SciddicaT

Progetto esame GPGPU Programming

2020/2021

Marco Bellizzi 223966

Sommario

Abstract

In questo report sono analizzate le attività svolte nella realizzazione del progetto per l'esame di GPGPU programming. Il progetto consiste nella parallelizzazione dell'applicazione SciddicaT attraverso CUDA e l'uso delle GPGPU.

Sono state implementate 3 differenti versioni parallele:

- Versione monolitica
- Versione grid-stride
- Versione con memoria condivisa senza cello Halo
- Versione con memoria condivisa con celle Halo

In seguito sono state testate le versioni parallele con diverse configurazioni di griglie, sono stati confrontati i tempi di esecuzione e calcolati gli speed-up usando il tempo di esecuzione della versione sequenziale.

Introduzione

SciddicaT è un simulatore di flusso del fluido basato sul paradigma degli automi cellulari e sull'algoritmo di minimizzazione delle differenze.

Esso si basa su una mappa topografica delle altitudini, che consiste in una matrice cui ogni cella contiene un valore numerico rappresentante l'altitudine della cella. Si basa anche su una matrice degli spessori dei fluidi, cui ogni cella rappresenta lo spessore corrente del fluido della cella; e su 4 ulteriori matrici le cui celle rappresentano i deflussi provenienti dalla cella centrale verso le quattro celle posizionate a nord, sud, est o ovest rispetto la cella considerata. Le matrici hanno tutte le stesse dimensioni di righe e colonne.

La simulazione del deflusso del fluido da uno step all'altro è basata su 3 principali operazioni:

- Vengono azzerati i deflussi da ogni cella verso le 4 celle adiacenti.
- Vengono calcolati i deflussi di ogni cella verso le 4 celle adiacenti attraverso l'applicazione dell'algoritmo di minimizzazione delle differenze.
- Viene aggiornato il valore dello spessore del fluido di ogni cella in funzione della variazione dei deflussi dello step precedente.

Data una configurazione iniziale, applicando iterativamente questi 3 step uno dopo l'altro si è in grado di prevedere il l'andamento del flusso del fluido sulla mappa dopo un determinato periodo di tempo.

È stata fornita una versione sequenziale di SciddicaT, consistente in un programma C++ che, inizializzata l'applicazione e letto l'input, esegue iterativamente 3 funzioni rappresentanti i 3 step della simulazione per ogni cella.

L'input consiste in 3 file:

- L'header, che contiene il numero di righe e di colonne delle matrici.
- Il dem, che contiene la mappa topografica delle altitudini.
- Il source, che contiene la configurazione iniziale del fluido.

L'obbiettivo del progetto è di parallelizzare il programma fornendo versioni parallele delle 3 funzioni.

Versione sequenziale

La versione sequenziale consiste in un ciclo che itera sul numero degli step (4000) cui all'interno sono chiamate di seguito le 3 funzioni sequenziali in ogni cella.

```
for (int s = 0; s < steps; ++s) {
  for (int i = i_start; i < i_end; i++)
    for (int j = j_start; j < j_end; j++)
      sciddicaTResetFlows(i, j, r, c, nodata, Sf);

  for (int i = i_start; i < i_end; i++)
    for (int j = j_start; j < j_end; j++)
      sciddicaTFlowsComputation(i, j, r, c, nodata, Xi,Xj,Sz,Sh,Sf,p_r,p_epsilon);

  for (int i = i_start; i < i_end; i++)
    for (int j = j_start; j < j_end; j++)
      sciddicaTWidthUpdate(i, j, r, c, nodata, Xi, Xj, Sz, Sh, Sf);
}
```

Implementazioni parallele

Versione monolitica

Nella versione monolitica le matrici vengono allocate nella memoria globale.

```
cudaMallocManaged(&Sz, sizeof(double) * r * c);
cudaMallocManaged(&Sh, sizeof(double) * r * c);
cudaMallocManaged(&Sf, sizeof(double) * ADJACENT_CELLS * r * c);
```

Vengono poi creati blocchi di threads di dimensione **BLOCK_SIZE** x **BLOCK_SIZE** e un numero di blocchi pari al rapporto tra le dimensioni delle matrici e **BLOCK_SIZE**.

```
dim3 dimGrid(ceil(r/(float)BLOCK_SIZE), ceil(c/(float)BLOCK_SIZE), 1);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, 1);
```

Successivamente vengono chiamate le versioni parallele delle 3 funzioni, sincronizzando i threads tra una funzione e l'altra.

```
for (int s = 0; s < steps; ++s) {
  sciddicaTResetFlows_Kernel<<<dimGrid, dimBlock>>>(r, c, nodata, Sf, i_start,
i_end, j_start, j_end);
  cudaDeviceSynchronize();

  sciddicaTFlowsComputation_Kernel<<<dimGrid, dimBlock>>>(r, c, nodata, Xi, Xj, Sz,
Sh, Sf, p_r, p_epsilon, i_start, i_end, j_start, j_end);
  cudaDeviceSynchronize();

  sciddicaTWidthUpdate_Kernel<<<dimGrid, dimBlock>>>(r, c, nodata, Xi, Xj, Sz, Sh,
Sf, i_start, i_end, j_start, j_end);
  cudaDeviceSynchronize();
}
```

Ogni thread calcola i relativi indici della propria cella ed effettua le proprie operazioni in parallelo, escludendo gli indici delle celle che non appartengono alla matrice.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;

if(i < i_start || i >= i_end || j < j_start || j >= j_end)
  return;

// operazioni
```

Versione grid stride

La versione grid stride è adatta quando il numero di thread totali disponibile è inferiore al numero di celle; quindi, i thread ad ogni step devono effettuare la computazione delle 3 funzioni su più di una cella. A questo scopo è stata dichiarata una variabile **NUM_ITER**, usata per calcolare la dimensione della griglia e il numero dei blocchi, al fine di assegnare ad ogni thread un numero pari a **NUM_ITER** x **NUM_ITER** celle.

```
dim3 dimGrid(ceil(r/(float)BLOCK_SIZE/NUM_ITER),
      ceil(c/(float)BLOCK_SIZE/NUM_ITER), 1);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, 1);
```

All'interno delle funzioni parallele sono stati usati dei cicli che sono in grado calcolare gli indici delle celle sulle quali il thread deve effettuare le computazioni, al variare del numero e della grandezza dei blocchi.

```
for(int i=blockIdx.x*blockDim.x+threadIdx.x; i<r; i+=blockDim.x*gridDim.x) {
  for(int j=blockIdx.y*blockDim.y+threadIdx.y; j<c; j+=blockDim.y*gridDim.y) {
    if(i < i_start || i >= i_end || j < j_start || j >= j_end)
      continue;
    // operazioni
  }
}
```

Versione con memoria condivisa senza celle Halo

Questa versione usa la memoria condivisa. Le matrici vengono sempre allocate nella memoria globale, ma le funzioni usano la memoria condivisa per effettuare la computazione dei dati.

La prima funzione non richiede accesso in lettura a nessuna matrice; quindi non è stata utilizzata la memoria condivisa.

La seconda funzione richiede l'accesso in lettura alle matrici delle altitudini e dello spessore dei fluidi; quindi ogni thread copia il valore della corrispettiva cella delle due matrici nella memoria condivisa.

```
__shared__ double Sz_shared[BLOCK_SIZE][BLOCK_SIZE];
__shared__ double Sh_shared[BLOCK_SIZE][BLOCK_SIZE];

if (i < r && j < c) {
  Sz_shared[tx][ty] = GET(Sz, c, i, j);
  Sh_shared[tx][ty] = GET(Sh, c, i, j);
} else {
  Sz_shared[tx][ty] = 0.0;
  Sh_shared[tx][ty] = 0.0;
}

__syncthreads();

if(i < i_start || i >= i_end || j < j_start || j >= j_end) {
    return;
}
```

La terza funzione invece richiede l'accesso in lettura alle quattro matrici dei deflussi del fluido; quindi, ogni thread copia il valore delle corrispettive celle delle quattro matrici nella memoria condivisa.

```
__shared__ double shared[BLOCK_SIZE][BLOCK_SIZE][4];

if (i < r && j < c) {
  shared[tx][ty][0] = BUF_GET(Sf, r, c, 0, i, j);
  shared[tx][ty][1] = BUF_GET(Sf, r, c, 1, i, j);
  shared[tx][ty][2] = BUF_GET(Sf, r, c, 2, i, j);
  shared[tx][ty][3] = BUF_GET(Sf, r, c, 3, i, j);
} else {
  shared[tx][ty][0] = 0.0;
  shared[tx][ty][1] = 0.0;
  shared[tx][ty][2] = 0.0;
  shared[tx][ty][3] = 0.0;
}
```

```
__syncthreads();

if(i < i_start || i >= i_end || j < j_start || j >= j_end) {
  return;
}
```

Successivamente dove possibile le funzioni accederanno alla memoria condivisa e non alla memoria globale per ottenere i dati di cui ha bisogno per la computazione. Questo è stato fatto per limitare gli accessi alla memoria globale in quanto è più lenta della memoria condivisa.

Versione con memoria condivisa con celle Halo

Questa versione è simile alla precedente, ma leggermente diversa. In questa versione sono state inserite nella memoria condivisa anche le celle Halo. Nella versione precedente quando una cella lungo un bordo di un blocco doveva accedere alla cella confinante che appartiene ad un altro blocco, esso andava a leggere il contenuto della cella nella memoria globale, non essendo state memorizzate le celle halo nella memoria condivisa. In questa versione invece, la dimensione delle sottomatrici della memoria condivisa non corrisponde completamente alla dimensione dei blocchi, perché includono anche le celle halo, quindi le loro dimensioni sono leggermente più grandi. In questa versione ogni thread si ritrova nella memoria condivisa tutte le celle di cui ha bisogno senza dover mai avvedere alla memoria globale.

Valutazione delle prestazioni

Di seguito sono riportati i tempi di esecuzione e gli speed-up delle varie versioni con le varie configurazioni di griglie testate. La dimensione dell'input è 610 x 496 mentre il tempo d'esecuzione della versione sequenziale per il calcolo dello speed-up è 67,5.

Versione monolitica

dimBlock dimGrid	4 x 4 x 1 153 x 124 x 1	8 x 8 x 1 77 x 62 x 1	16 x 16 x 1 39 x 31 x 1	32 x 32 x 1 20 x 16 x 1
tempo	4,39 s	3,39 s	3,66 s	4,37 s
Speed-up	15,38	19,91	17,62	15,45

Versione grid stride

dimBlock dimGrid	4 x 4 x 1 77 x 62 x 1	4 x 4 x 1 51 x 42 x 1	8 x 8 x 1 39 x 31 x 1	8 x 8 x 1 26 x 21 x 1
tempo	4,33 s	4,05 s	3,17 s	3,54
Speed-up	15,59	16,67	21,29	19,07

dimBlock dimGrid	16 x 16 x 1 20 x 16 x 1	16 x 16 x 1 13 x 11 x 1	32 x 32 x 1 10 x 8 x 1	32 x 32 x 1 7 x 6 x 1
tempo	4,44 s	4,71 s	4,12 s	5,94 s
Speed-up	15,20	14,33	16,38	11,36

Versione con memoria condivisa senza celle Halo

dimBlock dimGrid	4 x 4 x 1 153 x 124 x 1	8 x 8 x 1 77 x 62 x 1	16 x 16 x 1 39 x 31 x 1	32 x 32 x 1 20 x 16 x 1
tempo	4,72 s	3,48 s	4,03 s	4,38 s
Speed-up	14,30	19,40	17,75	15,41

Versione con memoria condivisa con celle Halo

dimBlock dimGrid	8 x 8 x 1 77 x 62 x 1	16 x 16 x 1 39 x 31 x 1	32 x 32 x 1 20 x 16 x 1
tempo	4,83 s	4,34 s	4,28 s
Speed-up	13,98	15,55	15,77

Conclusioni

Abbiamo mostrato come attraverso CUDA e l'uso delle GPGPU si è stato in grado di parallelizzare l'applicazione SciddicaT. Dall'analisi delle performance si evince come i risultati migliori si ottengono utilizzando blocchi di dimensione 8 x 8 x 1. In particolare, il risultato migliore è stato ottenuto nella versione grid stride (assegnando ad ogni thread 4 celle) ottenendo uno speed-up di 21,29. Le versioni che utilizzano la memoria condivisa non sono state in grado di migliorare le performance in quanto il tempo impiegato dall'introduzione della memoria condivisa è risultato maggiore del tempo risparmiato dai minori accessi alla memoria globale. L'applicazione non fa un uso massiccio della memoria, in quando al più ogni funzione richiede l'accesso alla propria cella e alle celle adiacenti delle relative matrici; dunque, l'utilizzo della memoria condivisa non è risultato efficiente.