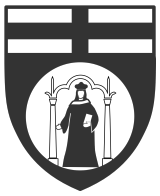

Università degli Studi di Genova
Anno accademico 2020/2021

Consorzio IANUA

Relazione sul modulo Pervasive Computing

Pietro Firpo, Matteo Littardi

Settembre 2021

1 Introduzione

A termine del modulo *Pervasive computing* proposto dal Consorzio IANUA nell'A. A. 2020/2021 abbiamo voluto esplorare le possibilità offerte dalla parallelizzazione tramite CUDA Toolkit, software sviluppato da Nvidia per le proprie schede grafiche.
Tutti i test presenti nella relazione sono stati eseguiti su una scheda Nvidia GTX 1050Ti Max-Q e un processore Intel Core i7-9750H. Tutti file sorgente utilizzati per ottenere i dati riportati nella relazione sono reperibili al link `https://github.com/CatoMaior/HPC-tests.git`.

2 Primo esempio: somma di vettori

Iniziando tramite un esercizio molto semplice per provare le funzionalità del CUDA Toolkit abbiamo provato a implementare la somma tra vettori sfruttando i 768 CUDA Cores della GPU. Il programma viene parallelizzato suddividendo il compito in blocchi divisi a loro volta in threads. Abbiamo implementato la somma in modo che ogni singolo thread si occupi degli elementi che hanno indice congruo al proprio id modulo il numero di thread, utilizzando il codice seguente:

```
8  __global__ void sumVector(float *a, float *b, float *result, int n) {
9      int firstIndex = blockIdx.x * blockDim.x + threadIdx.x;
10     for(int i = firstIndex; i < n; i += BLOCK_SIZE * N_BLOCKS) {
11         result[i] = a[i] + b[i];
12     }
13 }
```

Abbiamo fatto più prove variando il numero dei blocchi e la loro dimensione. Rispetto alla situazione in cui l'intera somma vettoriale è effettuata da un unico thread siamo riusciti ad ottenere uno speed-up di circa 33 volte.

Numero blocchi	Dimensione blocchi	Tempo di esecuzione
1	1	14.70 secondi
1	100	0.84 secondi
100	1	0.75 secondi
500	100	0.44 secondi

Tabella 1: Tempi di esecuzione delle implementazioni della somma fra vettori

3 Secondo esempio: il modello di Ising

Dopo aver preso confidenza con il CUDA Toolkit abbiamo provato a parallelizzare del codice scritto da noi qualche anno fa. Si tratta di un programma di simulazione con il metodo Monte Carlo del modello di Ising, usato per analizzare la transizione della materia da ferromagnetismo a paramagnetismo con il crescere della temperatura. Il modello considera un reticolo di particelle il cui stato evolve nel tempo, scandito dall'aumentare della temperatura, secondo relazioni stocastiche che coinvolgono le particelle vicine. L'energia del sistema considerato viene calcolata tramite i valori delle forze elettromagnetiche tra le particelle; queste sono determinate dallo spin di ogni singola particella, che ammette come valori solamente `-1` o `1`. Quindi, chiamando J la costante di accoppiamento (nel nostro caso, $J = 1$), $S_{i,j}$ lo spin della particella nella posizione (i, j) e H il valore del campo magnetico esterno a cui il reticolo è sottoposto (nel nostro caso $H = 0$) l'energia totale del sistema è:

$$E = -J \sum_{\substack{i+j<i'+j' \\ |i-i'|+|j-j'|=1}} S_{i,j} S_{i',j'} + H \sum_{i,j} S_{i,j}$$

Ogni thread sceglie casualmente una particella e osserva cosa succederebbe se il suo spin venisse invertito: se l'energia del sistema diminuisse, tale spin viene effettivamente invertito; altrimenti viene invertito con una probabilità che dipende dalla variazione di energia che l'inversione causerebbe. Questo procedimento viene effettuato con il seguente codice:

```
39 __global__ void updateBoard(char *gpuS, float *T, curandState *states) {
40     int id = threadIdx.x;
41     int x = ((int) (generate(states, id) * 1000000)) % (N - 1);
42     int y = ((int) (generate(states, id) * 1000000)) % (N - 1);
43     float deltaE = -2 * J * *(gpuS + N * x + y) *(
44                                     *(gpuS + ((x + 1) % N) * N + y % N) +
45                                     *(gpuS + ((x - 1) % N) * N + y % N) +
46                                     *(gpuS + (x % N) * N + (y + 1) % N) +
47                                     *(gpuS + (x % N) * N + (y - 1) % N)) -
48                                     2 * H * *(gpuS + N * x + y);
49     __syncthreads();
50     if (deltaE < 0 || exp((float) - deltaE / (Kb * *T)) > generate(states, id))
51         *(gpuS + N * x + y) *= -1;
52 }
```

Questo stesso programma è stato scritto anche nei linguaggi Python e C per confrontare le prestazioni con i seguenti risultati:

Linguaggio	Tempo di esecuzione
Python	177.02 secondi
C	3.99 secondi
Cuda	11,36 secondi

Tabella 2: Tempi di esecuzione delle implementazioni del modello di Ising

4 Conclusioni

Nel caso della somma vettoriale, vista la semplicità del compito da svolgere, abbiamo potuto sperimentare modificando il numero e le dimensioni dei blocchi di thread, ottenendo una notevole riduzione del tempo di esecuzione.
Nel caso del modello di Ising l'esecuzione su scheda video non ha permesso di scendere al di sotto del tempo di esecuzione dello stesso programma in linguaggio C, probabilmente per il tempo impiegato a trasferire i dati sulla scheda video e a sincronizzare i thread. Possiamo inoltre ipotizzare che questo sia dovuto al fatto che, per natura del modello simulato, è impossibile utilizzare un elevato numero di thread in quanto aumentandone il numero cresce la probabilità che due o più particelle adiacenti vengano aggiornate simultaneamente, facendo in modo che il programma fornisca risultati significativamente diversi a quelli ottenuti dal programma non parallelizzato. Per il modello di Ising, quindi, il vantaggio proveniente dalla maggiore disponibilità di unità di calcolo non è sufficiente a controbilanciare la perdita di tempo dovuta agli spostamenti dei dati tra le memorie.
Possiamo quindi giungere alla conclusione che alcuni programmi, a causa della loro struttura, non possono trarre giovamento dalla parallelizzazione su scheda video.

5 Programmi utilizzati

Per comodità del lettore alleghiamo qui il codice CUDA utilizzato.

5.1 Somma di vettori

```
#define LEN_ARR 100000000
#define BLOCK_SIZE 100
#define N_BLOCKS 500

#include <stdio.h>
#include <stdlib.h>

__global__ void sumVector(float *a, float *b, float *result, int n) {
    int firstIndex = blockIdx.x * blockDim.x + threadIdx.x;
    for(int i = firstIndex; i < n; i += BLOCK_SIZE * N_BLOCKS) {
        result[i] = a[i] + b[i];
    }
}

int main() {
    float *a = (float*)malloc(sizeof(float) * LEN_ARR);
    float *b = (float*)malloc(sizeof(float) * LEN_ARR);
    float *result = (float*)malloc(sizeof(float) * LEN_ARR);
    float *gpuA, *gpuB, *gpuResult;
```

```
20
21      for(int i = 0; i < LEN_ARR; i++){
22          a[i] = 30.0f;
23          b[i] = 12.0f;
24      }
25
26      cudaMalloc((void **) &gpuA, sizeof(float) * LEN_ARR);
27      cudaMalloc((void **) &gpuB, sizeof(float) * LEN_ARR);
28      cudaMalloc((void **) &gpuResult, sizeof(float) * LEN_ARR);
29
30      cudaMemcpy(gpuA, a, sizeof(float) * LEN_ARR, cudaMemcpyHostToDevice);
31      cudaMemcpy(gpuB, b, sizeof(float) * LEN_ARR, cudaMemcpyHostToDevice);
32
33      sumVector<<<N_BLOCKS, BLOCK_SIZE>>>(gpuA, gpuB, gpuResult, LEN_ARR);
34
35      cudaMemcpy(result, gpuResult,
36          sizeof(float) * LEN_ARR, cudaMemcpyDeviceToHost
37      );
38
39      cudaFree(gpuA);
40      cudaFree(gpuB);
41      cudaFree(gpuResult);
42
43      free(a);
44      free(b);
45      free(result);
46
47      return 0;
48  }
```

5.2 Modello di Ising

```
1   #include <math.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <curand_kernel.h>
5   #include <curand.h>
6   #include <time.h>
7
8   #define MAX_TEMP 4
9   #define NUM_STEP 100
10  #define MIN_TEMP 0.01
11  #define N 50
12  #define J -1
13  #define H 0
14  #define Kb 1
15  #define TRIGGER 1000
16  #define SAMPLE_DELAY 10
17  #define N_SAMPLES 800
```

```
18 #define N_THREAD 40
19
20 char S[N][N];
21
22 void randomizeS() {
23     for (int i = 0; i < N; i++)
24         for (int j = 0; j < N; j++)
25             S[i][j] = rand() > RAND_MAX / 2 ? 1 : -1;
26 }
27
28 __device__ float generate(curandState *globalState, int ind) {
29     curandState localState = globalState[ind];
30     float randNum = curand_uniform(&localState);
31     globalState[ind] = localState;
32     return randNum;
33 }
34
35 __global__ void setup_kernel(curandState *state, unsigned long seed) {
36     int id = threadIdx.x;
37     curand_init(seed, id, 0, &state[id]);
38 }
39
40 __global__ void updateBoard(char *gpuS, float *T, curandState *states) {
41     int id = threadIdx.x;
42     int x = ((int) (generate(states, id) * 1000000)) % (N - 1);
43     int y = ((int) (generate(states, id) * 1000000)) % (N - 1);
44     float deltaE = -2 * J * *(gpuS + N * x + y) *(
45                                     *(gpuS + ((x + 1) % N) * N + (y + 1) % N) +
46                                     *(gpuS + ((x + 1) % N) * N + (y - 1) % N) +
47                                     *(gpuS + ((x - 1) % N) * N + (y + 1) % N) +
48                                     *(gpuS + ((x - 1) % N) * N + (y - 1) % N)) -
49                                     2 * H * *(gpuS + N * x + y);
50     __syncthreads();
51     if (deltaE < 0 || exp((float) - deltaE / (Kb * *T)) > generate(states, id))
52         *(gpuS + N * x + y) *= -1;
53 }
54
55 float *runCycles(float T, curandState *devStates, char *gpuS) {
56     float *magnArr = (float *)malloc(N_SAMPLES * sizeof(float));
57     float *energyArr = (float *)malloc(N_SAMPLES * sizeof(float));
58     float magn, energy;
59     int insertedSamples = 0;
60
61     float* Ts;
62     cudaMalloc((void **) &Ts, sizeof(float));
63     cudaMemcpy(Ts, S, sizeof(float), cudaMemcpyHostToDevice);
64
65     for (unsigned int i = 0; i < TRIGGER + N_SAMPLES; i++) {
66         magn = 0;
```

```
67          energy = 0;
68          for (unsigned int j = 0; j < SAMPLE_DELAY; j++) {
69              updateBoard<<<1, N_THREAD>>>(gpuS, Ts, devStates);
70              cudaDeviceSynchronize();
71          }
72          cudaMemcpy(S, gpuS, N * N, cudaMemcpyDeviceToHost);
73          if (i > TRIGGER) {
74              for (int j = 0; j < N; j++) {
75                  for (int k = 0; k < N; k++)
76                      magn += S[j][k];
77              }
78              magnArr[insertedSamples] = magn / (N * N);
79              for (int j = 0; j < N; j++) {
80                  for (int k = 0; k < N; k++)
81                      energy += J * S[j][k] * (S[(j + 1) % N][k] +
82                                   S[j][(k + 1) % N]) - 2 * H * S[j][k];
83              }
84              energyArr[insertedSamples] = energy;
85              insertedSamples++;
86          }
87      }
88
89      cudaFree(Ts);
90
91      float susc = 0, sq_av = 0, av_sq = 0, cal = 0;
92      magn = 0;
93
94      for (int i = 0; i < N_SAMPLES; i++) {
95          sq_av += magnArr[i];
96          av_sq += sq_av * sq_av;
97      }
98      sq_av /= N_SAMPLES;
99      av_sq /= N_SAMPLES;
100
101     magn = abs(sq_av);
102     sq_av = sq_av * sq_av;
103     susc = (av_sq - sq_av) / T;
104     sq_av = 0;
105     av_sq = 0;
106
107     for (int i = 0; i < N_SAMPLES; i++) {
108         sq_av += energyArr[i];
109         av_sq += sq_av * sq_av;
110     }
111     sq_av /= N_SAMPLES;
112     av_sq /= N_SAMPLES;
113     sq_av = sq_av * sq_av;
114     cal = (av_sq - sq_av) / T;
115
```

```
116     free(magnArr);
117     free(energyArr);
118 
119     float *retArr = (float *)malloc(4 * sizeof(float));
120     retArr[0] = T;
121     retArr[1] = susc;
122     retArr[2] = cal;
123     retArr[3] = magn;
124     return retArr;
125 }
126 
127 int main() {
128     srand(time(NULL));
129     randomizeS();
130     char* gpuS;
131     cudaMalloc((void **) &gpuS, N * N * sizeof(char));
132     cudaMemcpy(gpuS, S, N * N, cudaMemcpyHostToDevice);
133     curandState* devStates;
134     cudaMalloc(&devStates, N * sizeof(curandState));
135     int seed = rand();
136     setup_kernel<<<1, N_THREAD>>>(devStates, seed);
137     cudaDeviceSynchronize();
138     float *resArr;
139     float susc[NUM_STEP], temp[NUM_STEP], heat[NUM_STEP], arrMagn[NUM_STEP];
140     for (int i = 0; i < NUM_STEP; i++) {
141         float t = MIN_TEMP + (MAX_TEMP - MIN_TEMP) / NUM_STEP * i;
142         resArr = runCycles(t, devStates, gpuS);
143         temp[i] = resArr[0];
144         susc[i] = resArr[1];
145         heat[i] = resArr[2];
146         arrMagn[i] = resArr[3];
147         free(resArr);
148     }
149     cudaFree(gpuS);
150     return 0;
151 }
152 
```