Calcolo parallelo e distribuito mod. B

HW2

implementazione algoritmo prodotto tra matrici con CUDA

Boukara Djihad N97000275

Cicala Crispino N97000264

Indice

1 Descrizione del problema

Si vuole implementare un algoritmo per il prodotto tra matrici in un ambiente di calcolo parallelo che utilizzi la libreria CUDA.

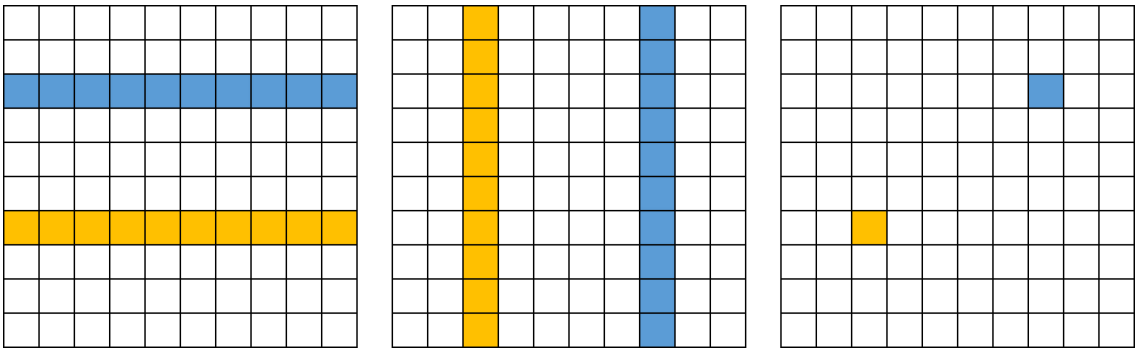
1.1 CUDA

Le GPU (Graphics processing unit) sono nate per andare incontro alle esigenze del mondo dei videogame, con lo scopo di migliorare, per esempio, le prestazioni legate al rendering degli oggetti tridimensionali, ma con il passare del tempo sono state utilizzate anche in settori diversi, tra i quali per esempio quello del calcolo scientifico e parallelo e quindi generalmente per la risoluzione di problemi matematici. Sulle GPU si trova un numero di microprocessori a memoria condivisa nell'ordine della migliaia, che abbinato all'ottimizzazione della larghezza di banda privilegia le fasi di calcolo rispetto agli accessi alla memoria. Alcune GPU, dette GPGPU(General-purpose computing on graphics processing units) hanno un numero elevato di processori simili ai multi-core generalmente adottati dalle CPU, questa scelta è stata sviluppata per operazioni complesse di calcolo e migliora la scalabilità delle prestazioni. Con il diffondersi delle GPGPU nascono i linguaggi ad alto livello per operare su questi dispositivi. CUDA (Compute Unified Device Architecture) è un'architettura hardware per l'elaborazione parallela creata da NVIDIA e può operare solo su macchine di quest'ultima. I linguaggi di programmazione disponibili nell'ambiente di sviluppo CUDA sono estensioni dei linguaggi più diffusi, quello utilizzato in questo progetto è CUDA-C, ma esistono estensioni anche per Python, Fortran, Java e MATLAB. Utilizzando CUDA abbiamo un parallelismo SIMT, ognuna delle ALU può eseguire un flusso di operazioni indipendente e i thread eseguiti da ognuna compiono esattamente le stesse operazioni, ottenendo un alto livello di parallelismo, visto che avendo **p** multiprocessori a memoria condivisa, ognuno dei quali avrà **q** ALU all'interno, il numero di processori risulta essere **p*q**.

2 Implementazione

2.1 Strategia

La strategia adottata per il calcolo del prodotto tra matrici prevede che ogni processore calcoli una componente della matrice risultate C, in particolare ogni processore K in posizione (i, j) ha il compito di calcolare il valore della cella $C(i, j)$ della matrice risultante come il prodotto tra la riga i della matrice A e la colonna j della matrice B. Al termine del calcolo tutte le cella della matrice C saranno state calcolate.



2.2 Analisi del problema

Verranno definite due matrici di input $A \in R^{mxn}$ e $B \in R^{mxn}$ ed una di output $C \in R^{mxn}$. L'implementazione finale dovrà infatti prevedere anche casi di matrici non quadrate. Verranno analizzati in risultati ottenuti al crescere dell'ordine delle matrici e del numero di thread per blocco.

2.3 Algoritmo

I processori vengono suddivisi in blocchi secondo la struttura illustrata in figura 2.1.

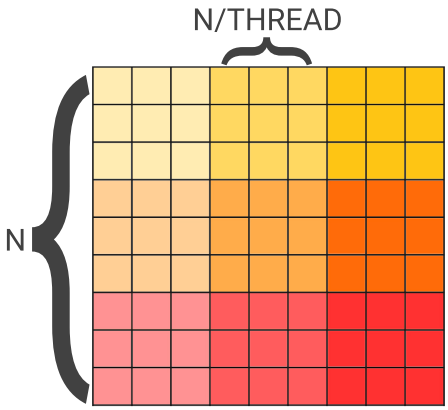


Figura 2.1: Divisione in blocchi della matrice. Ogni sfumatura rappresenta un blocco, ed ogni cella un thread.

Per ognuna delle tre matrici viene calcolata la più piccola matrice le cui dimensioni siano multiple della dimensione del blocco definita dalla variabile `THREADS`. Attraverso la struttura `dim3`, utilizzata in CUDA per passare i riferimenti della griglia e dei blocchi, vengono memorizzati nelle variabili `blocksA`,`blocksB` e `blocksC` i riferimenti dei blocchi delle tre matrici nei campi x e y della struttura. Sempre attraverso una variabile di tipo `dim3` vengono memorizzati il numero di thread per blocco.
Attraverso tre distinte chiamate a `cudaMalloc` viene allocato sulla memoria della GPU lo spazio richiesto per le tre matrici. Al termine di queste operazioni il contenuto delle matrici viene copiato dalla memoria principale a quella del device attraverso le chiamate a `cudaMemcpy2D`.

Per gestire il caso generale di matrici non quadrate, la parte in eccesso della matrice quadrata che contiene quella data in input viene riempita di 0. A questo punto può essere calcolato il prodotto attraverso la chiamata a `productKernel`, che avendo a disposizione gli id dei blocchi all'interno della griglia (`blockIdx`), e dei thread all'interno dei blocchi (`threadIdx` calcola l'indice della prima sottomatrice di A elaborata dal generico blocco m come $m * THREADS * indice$. Successivamente viene calcolato l'indice dell'ultima sottomatrice aggiungendo all'indice della prima $(m - 1)$. Il numero di colonne tra una sottomatrice e la successiva è uguale a $THREADS$. Leggermente

diverso il calcolo per matrice B, per la quale l'indice della prima sottomatrice è uguale a $THREADS * id_blocco_s u_a scissa$, e lo step tra una sottomatrice e l'altra è uguale a $THREADS * p$.
A questo punto per ogni sottomatrice viene suddiviso il calcolo degli elementi del blocco, caricando gli elementi di ciascuna sottomatrice in memoria condivisa, ogni thread del blocco carica un elemento. I processori vengono sincronizzati per assicurare che ogni thread del blocco abbia caricato gli elementi attraverso la chiamata a `__syncthreads()`. Dopo aver caricato tutti gli elementi si procede al calcolo dei risultati parziali, al termine i thread si sincronizzano per caricare nella memoria condivisa tra i thread il blocco successivo di A e di B. Una volta terminato il calcolo il risultato locale $C_{i,j}$ viene copiato nella memoria centrale.

3 Tempi

Ordine matrici	Tempo CPU	GFlops CPU	Tempo GPU 2*2	GFlops GPU 2*2	Speedup GPU 2*2
1000	0,95	2,11	0,43	4,61	2,21
2000	7,21	2,22	3,41	4,69	2,11
3000	24,88	2,17	11,47	4,71	2,17
4000	58,25	2,20	27,17	4,71	2,14
5000	113,13	2,21	53,17	4,70	2,13
6000	194,54	2,22	91,96	4,70	2,12
7000	307,67	2,23	146,33	4,69	2,10
8000	452,94	2,26	218,76	4,68	2,07
9000	645,14	2,26	311,77	4,68	2,07
10000	900,87	2,22	428,18	4,67	2,10

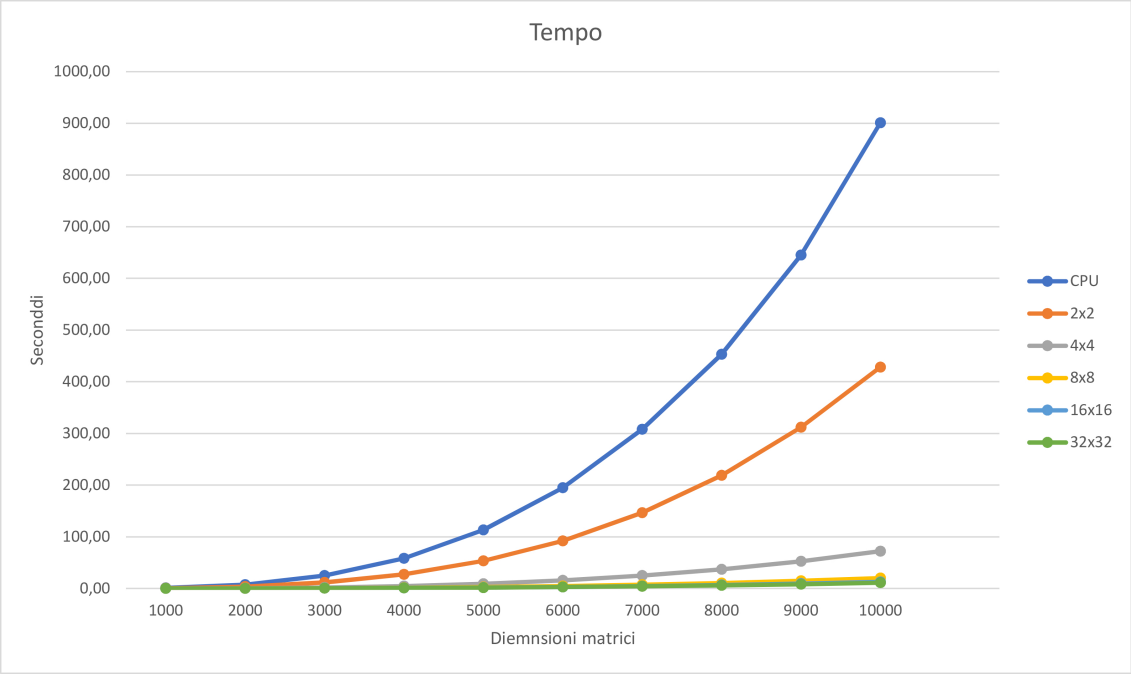
Ordine matrici	Tempo CPU	GFlops CPU	Tempo GPU 4*4	GFlops GPU 4*4	Speedup GPU 4*4
1000	0,95	2,11	0,08	24,02	11,88
2000	7,21	2,22	0,60	26,62	12,02
3000	24,88	2,17	1,99	27,19	12,50
4000	58,25	2,20	4,66	27,48	12,50
5000	113,13	2,21	9,06	27,58	12,49
6000	194,54	2,22	15,61	27,67	12,46
7000	307,67	2,23	24,74	27,73	12,44
8000	452,94	2,26	36,91	27,75	12,27
9000	645,14	2,26	52,49	27,78	12,29
10000	900,87	2,22	72,09	27,74	12,50

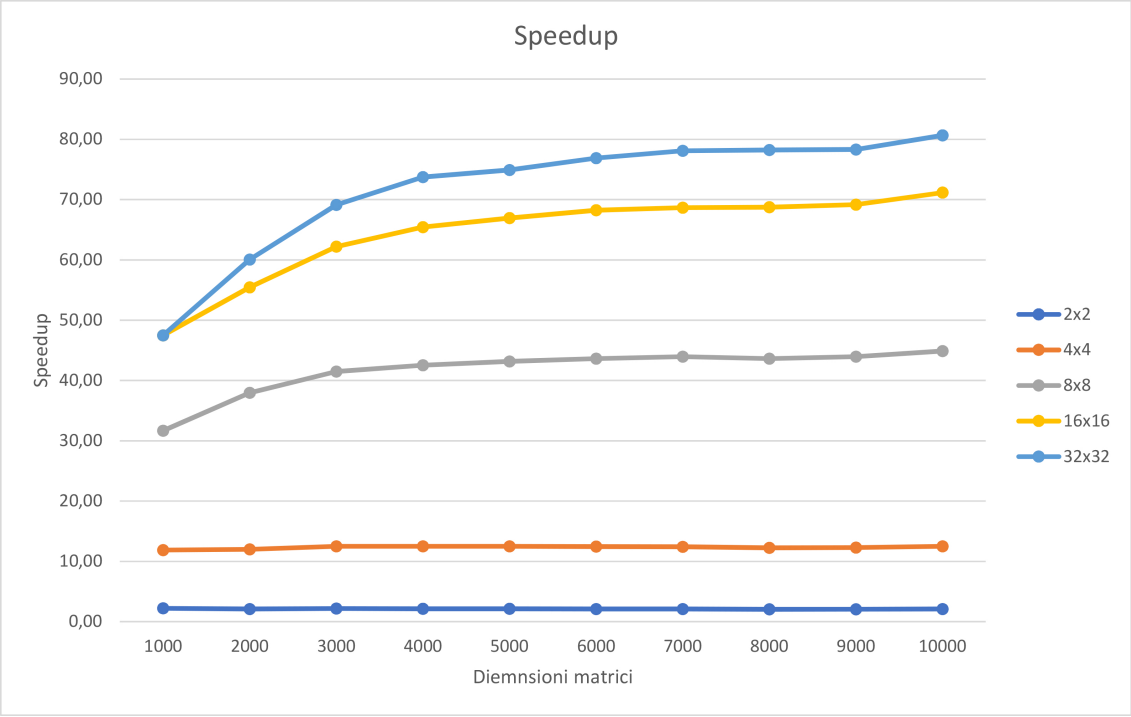
Ordine matrici	Tempo CPU	GFlops CPU	Tempo GPU 8*8	GFlops GPU 8*8	Speedup GPU 8*8
1000	0,95	2,11	0,03	62,96	31,67
2000	7,21	2,22	0,19	83,42	37,95
3000	24,88	2,17	0,60	90,01	41,47
4000	58,25	2,20	1,37	93,43	42,52
5000	113,13	2,21	2,62	95,48	43,18
6000	194,54	2,22	4,46	96,89	43,62
7000	307,67	2,23	7,00	97,93	43,95
8000	452,94	2,26	10,38	98,68	43,64
9000	645,14	2,26	14,68	99,30	43,95
10000	900,87	2,22	20,07	99,65	44,89

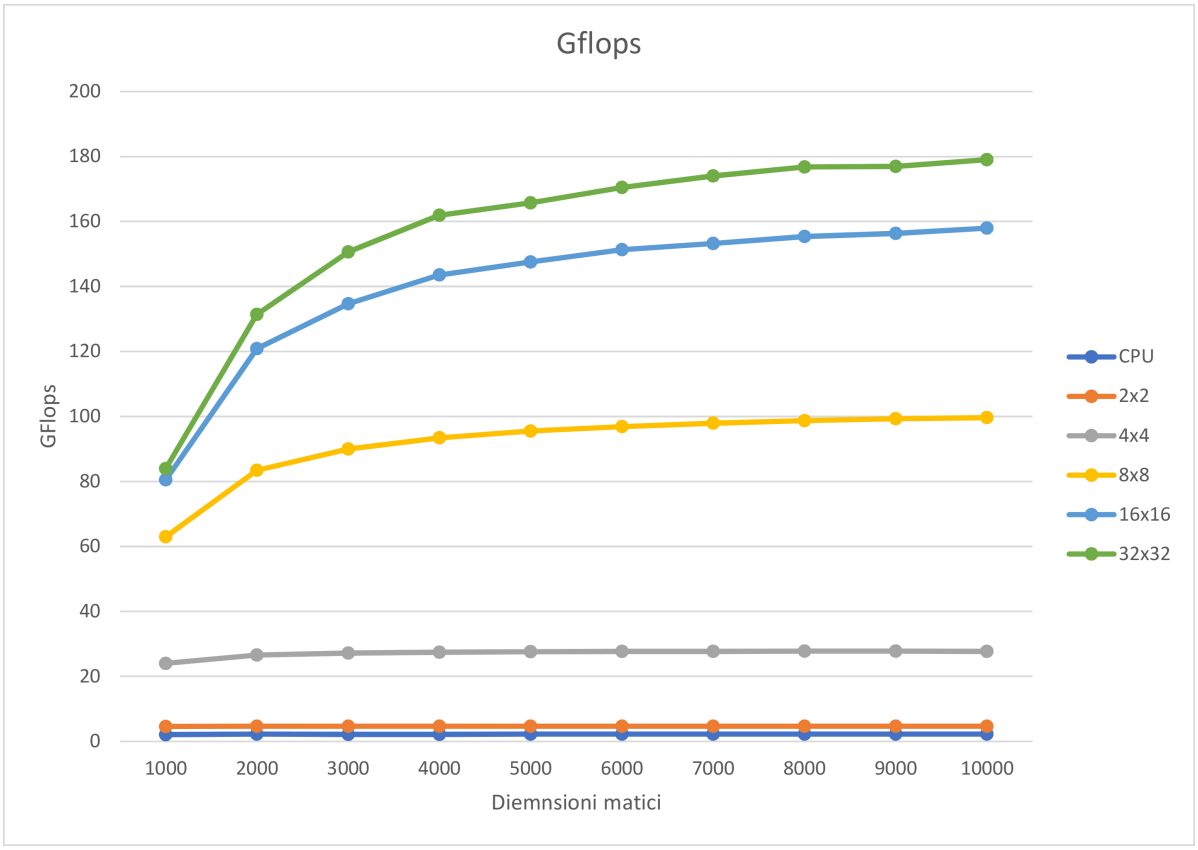
Ordine matrici	Tempo CPU	GFlops CPU	Tempo GPU 16*16	GFlops GPU 16*16	Speedup GPU 16*16
1000	0,95	2,11	0,02	80,46	47,50
2000	7,21	2,22	0,13	120,89	55,46
3000	24,88	2,17	0,40	134,63	62,20
4000	58,25	2,20	0,89	143,52	65,45
5000	113,13	2,21	1,69	147,56	66,94
6000	194,54	2,22	2,85	151,34	68,26
7000	307,67	2,23	4,48	153,20	68,68
8000	452,94	2,26	6,59	155,41	68,73
9000	645,14	2,26	9,33	156,32	69,15
10000	900,87	2,22	12,66	157,96	71,16

Ordine matrici	Tempo CPU	GFlops CPU	Tempo GPU 32*32	GFlops GPU 32*32	Speedup GPU 32*32
1000	0,95	2,11	0,02	83,94	47,50
2000	7,21	2,22	0,12	131,39	60,08
3000	24,88	2,17	0,36	150,67	69,11
4000	58,25	2,20	0,79	161,91	73,73
5000	113,13	2,21	1,51	165,71	74,92
6000	194,54	2,22	2,53	170,49	76,89
7000	307,67	2,23	3,94	173,97	78,09
8000	452,94	2,26	5,79	176,73	78,23
9000	645,14	2,26	8,24	176,93	78,29
10000	900,87	2,22	11,17	179,01	80,65

4 Grafici


Tempo
1000,00
900,00
800,00
700,00
600,00
500,00
400,00
300,00
200,00
100,00
0,00
Seconddi
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Diemnsioni matrici
CPU
2x2
4x4
8x8
16x16
32x32


Speedup
90,00
80,00
70,00
60,00
50,00
40,00
30,00
20,00
10,00
0,00
Speedup
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Diemnsioni matrici
2x2
4x4
8x8
16x16
32x32


Gflops
200
180
160
140
120
100
80
60
40
20
0
GFlops
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Diemnsioni matici
CPU
2x2
4x4
8x8
16x16
32x32

5 Codice

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <cuda.h>

#define MAT_SIZE 10000

#define GIGA 1000000000.
#define NSEC 1000000000.

#define THREADS 32

__host__ cudaError_t matmatCuda(int, int, int, int, int, int,
                                double *, double *, double *);
__global__ void productKernel(double*, double*, double*, int, int);
__global__ void adjustMatrix(double *, int, int);

double* rnd_flt_matrix(int n, int m);
double* zeros_flt_matrix(int n, int m);


int main(int argc, char const *argv[])
{
  // Strutture che memorizzano le matrici da moltiplicare
    double *A_h, *B_h, *C_d;
    double gflops_gpu;
    float elapsed;

    long nop;

    // variabili eventi utilizzate per stimare i tempi
    // di esecuzione sulla Gpu
    cudaEvent_t start_gpu, stop_gpu;

    double gpuTime; //tempo di esecuzione con Gpu

    cudaError_t cudaStatus;

  //allocazione matrici
    A_h = rnd_flt_matrix(MAT_SIZE, MAT_SIZE);
    B_h = rnd_flt_matrix(MAT_SIZE, MAT_SIZE);
    C_d = zeros_flt_matrix(MAT_SIZE, MAT_SIZE);

    // Inzializza gli eventi di inizio e fine
    cudaEventCreate(&start_gpu);
    cudaEventCreate(&stop_gpu);
```

```
48
49      printf("Ordine matrici; Tempo GPU; GFlops GPU\n");
50
51      for(int i = 100; i <= MAT_SIZE; i += 100)
52      {
53          // Prende il tempo di inizio
54          cudaEventRecord(start_gpu, 0);
55
56          // inizializza ed esegue il prodotto tra matrici
57          // sul device @see matmatCuda
58          cudaStatus = matmatCuda(MAT_SIZE, MAT_SIZE, MAT_SIZE,
59                                  i, i, i, A_h, B_h, C_d);
60
61          // Prende il tempo di fine
62          cudaEventRecord(stop_gpu, 0);
63          cudaEventSynchronize(stop_gpu);
64
65          if(cudaStatus != cudaSuccess){
66              fprintf(stderr, "matmatCuda failed!");
67              return 1;
68          }
69
70          // Calcolo il tempo trascorso tra i 2 eventi
71          cudaEventElapsedTime(&elapsed, start_gpu, stop_gpu);
72
73          // Trasforma il tempo in secondi
74          elapsed /= 1000.;
75
76          // Prende il tempo di fine
77          cudaEventRecord(stop_gpu, 0);
78          cudaEventSynchronize(stop_gpu);
79
80          // Calcolo il tempo trascorso tra i 2 eventi
81          cudaEventElapsedTime(&elapsed, start_gpu, stop_gpu);
82
83          // Trasforma il tempo in secondi
84          gpuTime = elapsed / 1000.;
85
86          nop = 2 * pow(i, 3);
87
88          gflops_gpu = (nop / gpuTime) / GIGA;
89
90          printf("%6d; %5.2lf; %5.2lf\n", i, gpuTime, gflops_gpu);
91      }
92
93      cudaEventDestroy(start_gpu);
94      cudaEventDestroy(stop_gpu);
95      free(A_h);
96      free(B_h);
97      free(C_d);
98
```

```
99      return 0;
100 }
101
102 double* rnd_flt_matrix(int n, int m)
103 {
104     int size = n * m;
105     double *A = (double*) malloc(sizeof(double) * size);
106
107     for(int i = 0; i < size; ++i)
108         A[i] = (float) rand() / RAND_MAX;
109
110     return A;
111 }
112
113 double* zeros_flt_matrix(int n, int m)
114 {
115     return (double*) calloc(n * m, sizeof(double));
116 }
117
118 // procedura che inserisce il valore 0
119 // per riempire gli spazi di matrice vuoti sulla cornice
120 __global__ void adjustMatrix(double *M, int n, int m)
121 {
122   int i = blockIdx.y * blockDim.y + threadIdx.y;
123   int j = blockIdx.x * blockDim.x + threadIdx.x;
124   int offset = gridDim.x * blockDim.x * i + j;
125
126     if(i >= n || j >= m)
127         M[offset] = 0.;
128 }
129
130 cudaError_t matmatCuda(int lda, int ldb, int ldc,
131                        int n, int m, int p, double *A, double *B, double *C)
132 {
133     double *A_d = NULL;
134     double *B_d = NULL;
135     double *C_d = NULL;
136
137   // Calcola il numero di blocchi della griglia
138     unsigned int nblock = (n + THREADS - 1) / THREADS;
139     unsigned int mblock = (m + THREADS - 1) / THREADS;
140     unsigned int pblock = (p + THREADS - 1) / THREADS;
141
142   // Calcola la piu' piccola matrice quadrata multipla
143   // di THREADS che possa contenere la matrice A e B
144     int nwidth = nblock * THREADS;
145     int mwidth = mblock * THREADS;
146     int pwidth = pblock * THREADS;
147
148     cudaError_t cudaStatus;
149
```

```
150     // Numero di blocchi per la griglia su A
151     dim3 blocksA;
152     blocksA.x = mblock;
153     blocksA.y = nblock;
154
155     // Numero di blocchi per la griglia su B
156     dim3 blocksB;
157     blocksB.x = pblock;
158     blocksB.y = mblock;
159
160   // Numero di blocchi per la griglia su C
161   dim3 blocksC;
162   blocksC.x = pblock;
163   blocksC.y = nblock;
164
165   // Numero di thread per blocco
166     dim3 threads;
167   threads.x = THREADS;
168     threads.y = THREADS;
169
170     // Scegli la GPU sul quale eseguire, in caso di un sistema multi-GPU.
171     cudaStatus = cudaSetDevice(0);
172     if (cudaStatus != cudaSuccess) {
173         fprintf(stderr, "1: cudaSetDevice failed!\n");
174         fprintf(stderr, "Do you have a CUDA-capable GPU installed?\n");
175         goto Error;
176     }
177
178   // **** Allocazione vettori all'interno della memoria del device ****
179
180     cudaStatus = cudaMalloc((void**)&A_d, nwidth * mwidth * sizeof(double));
181
182     if (cudaStatus != cudaSuccess) {
183         fprintf(stderr, "2: cudaMalloc failed!\n");
184         goto Error;
185     }
186
187     cudaStatus = cudaMalloc((void**)&B_d, mwidth * pwidth * sizeof(double));
188
189     if (cudaStatus != cudaSuccess) {
190         fprintf(stderr, "3: cudaMalloc failed!\n");
191         goto Error;
192     }
193
194     cudaStatus = cudaMalloc((void**)&C_d, nwidth * pwidth * sizeof(double));
195     if (cudaStatus != cudaSuccess) {
196         fprintf(stderr, "4: cudaMalloc failed!\n");
197         goto Error;
198     }
199   // **** Termine allocazione dei vettori ****
200
```

```
201   // **** Copia delle matrici in input nella memoria del device ****
202     cudaStatus = cudaMemcpy2D(A_d, mwidth * sizeof(double),
203                               A, lda * sizeof(double),
204                               m * sizeof(double), n, cudaMemcpyHostToDevice);
205
206     if (cudaStatus != cudaSuccess) {
207         fprintf(stderr, "5: cudaMemcpy2D failed!\n");
208         goto Error;
209     }
210
211     cudaStatus = cudaMemcpy2D(B_d, pwidth * sizeof(double),
212                               B, ldb * sizeof(double), p * sizeof(double),
213                               m, cudaMemcpyHostToDevice);
214
215     if (cudaStatus != cudaSuccess) {
216         fprintf(stderr, "6: cudaMemcpy2D failed!\n");
217         goto Error;
218     }
219
220   // **** Termina copia dei vettori ****
221
222   // Riempie la parte della matrice in eccesso con degli 0
223     if( (n % THREADS) || (m % THREADS) || (p % THREADS) ) {
224     adjustMatrix<<<blocksA, threads>>>(A_d, n, m);
225       adjustMatrix<<<blocksB, threads>>>(B_d, m, p);
226     }
227
228   // Calcolo il prodotto tra le 2 matrici
229     productKernel<<<blocksC, threads>>>(A_d, B_d, C_d, mwidth, pwidth);
230
231   // Copia del vettore risultante dalla memoria del device a quella RAM
232     cudaStatus = cudaMemcpy2D(C, ldc * sizeof(double),
233                               C_d, pwidth * sizeof(double), p * sizeof(double),
234                               n, cudaMemcpyDeviceToHost);
235
236     if (cudaStatus != cudaSuccess) {
237         fprintf(stderr, "7: cudaMemcpy2D failed!\n");
238         goto Error;
239     }
240
241
242   // Deallocazione vettori ed eventi dalla memoria del device
243     Error:
244         cudaFree(A_d);
245         cudaFree(B_d);
246         cudaFree(C_d);
247
248     return cudaStatus;
249 }
250
251 /**
```

```
252 * Esegue il prodotto scalare tra  una riga di A e una colonna di B
253 *
254 * @param A vettore rapprensentante la matrice A
255 * @param B vettore rapprensentante la matrice B
256 * @param C vettore risulatante reappresentante la matrice C
257 * @param width dimensione della matrice
258 */
259 __global__ void productKernel(double* A, double* B, double* C, int m, int p)
260 {
261     // id del blocco sull'ordinata all'interno della griglia
262     int ib = blockIdx.y;
263     // id del blocco sull'ascissa all'interno della griglia
264     int jb = blockIdx.x;
265     // id del thread sull'ordinata all'interno del blocco
266     int it = threadIdx.y;
267     // id del thread sull'ascissa allinterno del blocco
268     int jt = threadIdx.x;
269
270     int a, b, c, k;
271
272     // Indice della prima sottomatrice di A elaborata dal blocco
273     // m e' un multiplo intero di THREADS
274     // aBegin  include un certo numero ib  di gruppi di blocchi
275     // rettangolari THREADSxwidth
276     int aBegin = m * THREADS * ib;
277
278     //Indice dell'ultima sottomatrice di A elaborata dal blocco
279     int aEnd   = aBegin + m - 1;
280
281     // numero di colonne tra una sottomatrice e la successiva
282     int aStep  = THREADS;
283
284     // indice della prima sottomatrice di B elaborata dal blocco
285     // bBegin include un certo numero jb di blocchi di colonne,
286     // blocchi larghi THREADS
287     int bBegin = THREADS * jb;
288
289     // numero di elementi tra una sottomatrice e la successiva
290     int bStep  = THREADS * p;
291
292     // Csub e' usata come variabile in cui memorizzare
293     // il valore dell'elemento di C calcolato dal thread
294     // Viene aggiornato ripetutamente nel ciclo for seguente
295     double Csub = 0;
296
297     // Le matrici vengono divise in blocchi di dimensione THREADS X THREADS
298     // per ridurre il numero di accessi alla memoria pricipale del device
299     // che risultano costosi in termini di tempo
300
301     // Dichiarazione della variabile in cui salvare
302     // la sottomatrice di A in esame
```

```
303     __shared__ double As[THREADS][THREADS];
304
305     // Dichiarazione della variabile in cui salvare
306     // la sottomatrice di B in esame
307     __shared__ double Bs[THREADS][THREADS];
308
309     // Iterazione sulle sottomatrici
310     // in cui viene suddiviso il calcolo degli elementi del blocco
311     for (a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
312     {
313         // Vengono Caricati gli elementi di ciascuna
314         // sottomatrice in memoria condivisa:
315         // ogni thread del blocco carica un elemento!
316
317         As[it][jt] = A[a +  m * it + jt];
318         Bs[it][jt] = B[b +  p * it + jt];
319
320         // i processi vengono sincronizzati per assicurare
321         // che ogni thread del blocco abbia caricato gli elementi.
322         __syncthreads();
323
324         // vengono calcolati i contributi agli elementi di matrice di C
325         // dovute alla sottomatrici in esame
326         for( k = 0; k < THREADS ; ++k )
327             Csub += As[it][k]*Bs[k][jt];
328         // l'elemento C[it][jt] viene aggiornato in un numero di volte
329         // pari al numero di iterazioni del ciclo for
330
331         // i processi vengono sincronizzati per assicurare che il calcolo
332         // precedente sia terminato prima di caricare nuove
333         // sottomatrici
334         __syncthreads();
335     }
336
337     // vengono inseriti i risultati in C.
338     // Ogni thread elabora un elemento di C.
339     c = p * THREADS * ib + THREADS * jb;
340     C[c +  p * it + jt] = Csub;
341 }
```