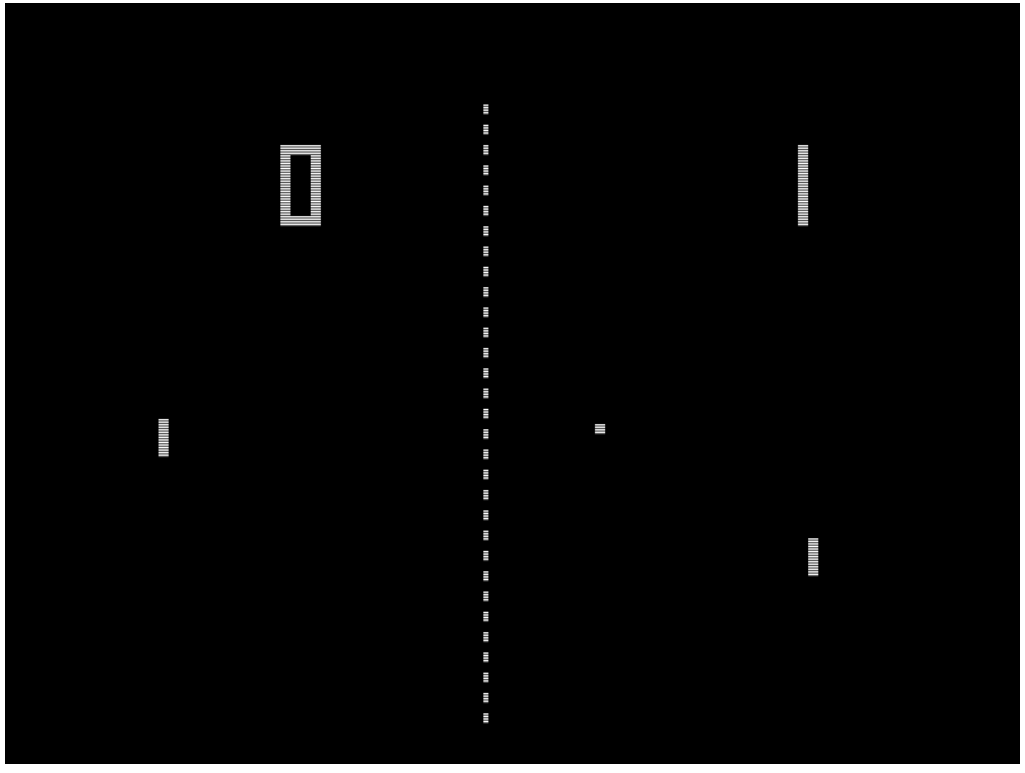
Programmazione su schede GPU in CUDA

Marco Sansottera

UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

Università degli Studi di Milano
Roma, 12.04.2016

Pong 1966



Le prime immagini di un videogioco di ping-pong appartengono al 1966 (Sanders Associates).

Nello stesso periodo Nolan Bushnell, neolaureato e futuro fondatore della *Atari*, realizza PONG. La versione domestica deve attendere il 1975.

Zork 1977

Che ci crediate o no, questo è stato uno dei primi videogiochi di successo per personal computer! Zork è un'*avventura testuale* ideata da alcuni componenti del *Dynamic Modelling Group* (MIT) tra il 1977 e il 1979.

Zork 1977

Che ci crediate o no, questo è stato uno dei primi videogiochi di successo per personal computer! Zork è un'*avventura testuale* ideata da alcuni componenti del *Dynamic Modelling Group* (MIT) tra il 1977 e il 1979.

dimenticavo . . . *"open mailbox"*.

Wild Gunman 1974

Se si era abbastanza fortunati da vivere vicino a qualche sala giochi le cose non erano poi così male! *Nintendo* stava rivoluzionando la grafica!

Wild Gunman 1974

Se si era abbastanza fortunati da vivere vicino a qualche sala giochi le cose non erano poi così male! *Nintendo* stava rivoluzionando la grafica!

"si devono usare le mani, allora è un gioco da bambini"

21.10.2015 — Ritorno al futuro II

Super Mario 1985

Processori limitati ed altre restrizioni imposte dall'hardware delle console obbligavano all'uso di patterns ripetitivi.

Le nuvole e i cespugli sono la stessa immagine!

De gustibus


SUPER MARIO BROS.
SUPER MARIO BROS. 2
SUPER MARIO BROS. 3
SUPER MARIO WORLD
SUPER MARIO 64
SUPER MARIO SUNSHINE
SUPER MARIO GALAXY
1985
1988
1990
1991
1996
2002
2007

Stessa storia, stesso posto . . . ma la grafica

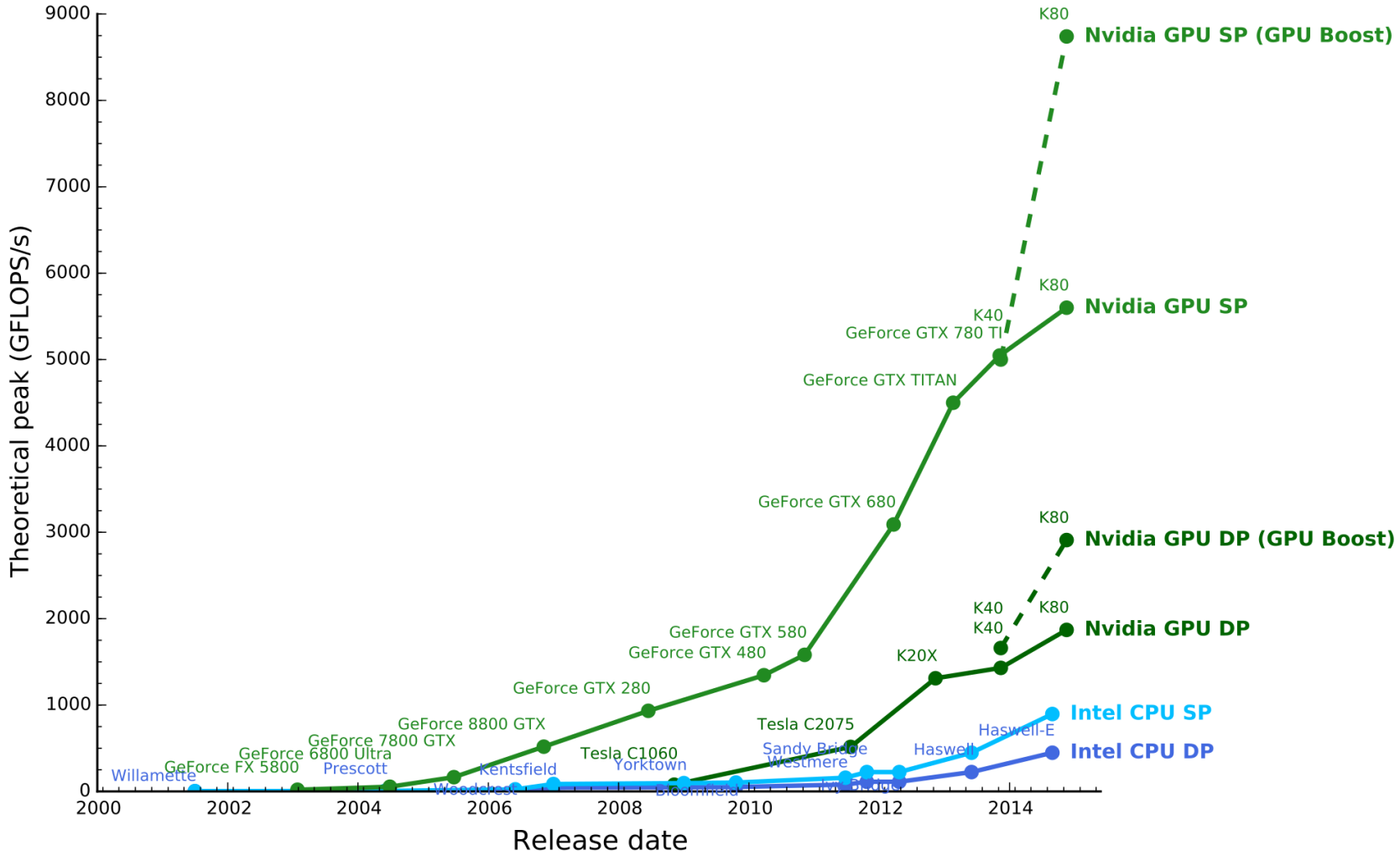
Cosa ha permesso questa evoluzione?

Acceleratore hardware

Un acceleratore è uno speciale componente hardware progettato per migliorare sensibilmente le prestazioni del computer in un settore specifico, tipicamente per le operazioni di calcolo in virgola mobile e le elaborazioni grafiche.

Gli acceleratori vengono spesso utilizzati nei videogiochi per aumentarne il livello di realismo grafico e la fluidità. Questo è possibile in quanto l'acceleratore hardware è stato progettato specificamente per le operazioni più spinte nel settore specifico a cui si rivolge.

CPU *vs.* GPU


Theoretical peak (GFLOPS/s)
Release date
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
2000
2002
2004
2006
2008
2010
2012
2014
K80
Nvidia GPU SP (GPU Boost)
K80
Nvidia GPU SP
K40
GeForce GTX 780 TI
GeForce GTX TITAN
GeForce GTX 680
K80
Nvidia GPU DP (GPU Boost)
K40
K40
K80
Nvidia GPU DP
GeForce GTX 580
GeForce GTX 480
K20X
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla C2075
Tesla C1060
Sandy Bridge
Westmere
Haswell
Haswell-E
Intel CPU SP
Intel CPU DP
Yorktown
Kentsfield
Prescott
Willamette

Prima legge di Moore

La complessità di un microcircuito, misurata ad esempio tramite il numero di transistori per chip, raddoppia ogni 18 mesi.

Limite della prima legge di Moore

La legge di Moore ha un limite imposto dalla fisica: esiste un limite fisico per la riduzione delle dimensioni dei transistor, al di sotto dei quali si genererebbero effetti indesiderati di natura quantistica nei circuiti elettronici.

Raggiunto quel limite (ci siamo quasi) l'unico modo possibile e praticabile per aumentare le prestazioni di calcolo è rappresentato dalla tecnologia multicore.

www.top500.org (dati Giugno 2015)

GPU NVIDIA vs. Intel Xeon Phi

Tianhe-2 (1) e Stampede (7) utilizzano acceleratori Intel Xeon Phi.
Titan (2) e Piz Daint (6) utilizzano acceleratori GPU NVIDIA.

Alcuni dati interessanti

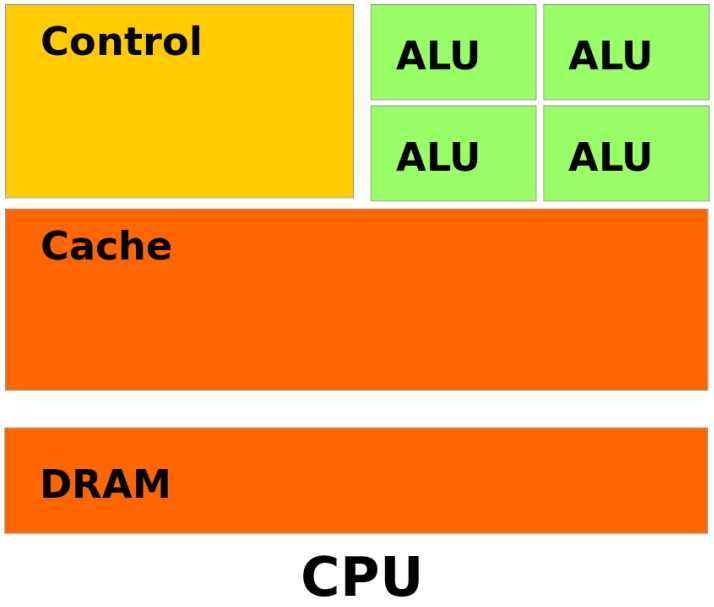
Ad oggi, un totale di 90 sistemi della lista TOP500 utilizzano acceleratori e/o coprocessori. A Novembre 2014 i sistemi erano 75.

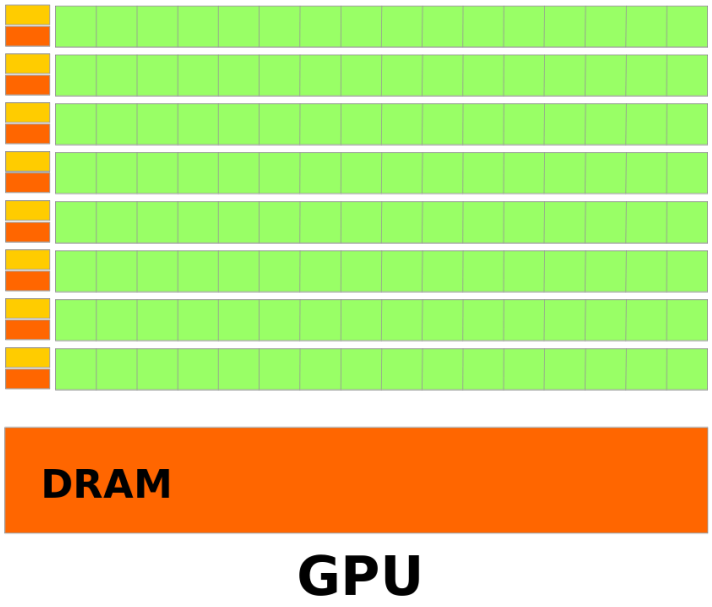
52 di questi sistemi utilizzano GPU NVIDIA, 4 usano ATI Radeon e 35 sfruttano la tecnologia Intel MIC (Xeon Phi).

GPU NVIDIA & Intel Xeon Phi

Quattro sistemi usano sia GPU Nvidia che Intel Xeon Phi.

CPU *vs.* GPU





Le CPU sono dispositivi *general purpose*: in linea di principio devono essere in grado di eseguire *qualunque* operazione.

Le GPU sono dispositivi *special purpose*: sono ottimizzate per processare (tramite operazioni elementari) grandi quantità di dati in parallelo.

Diversi scopi . . . diversi design.

CPU: minimizzare la latenza, l'intervallo di tempo che impiega la CPU ad eseguire una particolare operazione.

GPU: massimizzare il throughput, la capacità di elaborazione dei dati.

Diversi scopi . . . diversi design.

CPU: minimizzare la latenza, l'intervallo di tempo che impiega la CPU ad eseguire una particolare operazione.

GPU: massimizzare il throughput, la capacità di elaborazione dei dati.

CPU

GPU

CPU & GPU $\Rightarrow$ CUDA

CUDA (Compute Unified Device Architecture) è un'architettura hardware per l'elaborazione parallela creata da NVIDIA.

CUDA è un modello di programmazione che permette di aumentare le prestazioni di calcolo sfruttando la potenza di calcolo delle GPU.

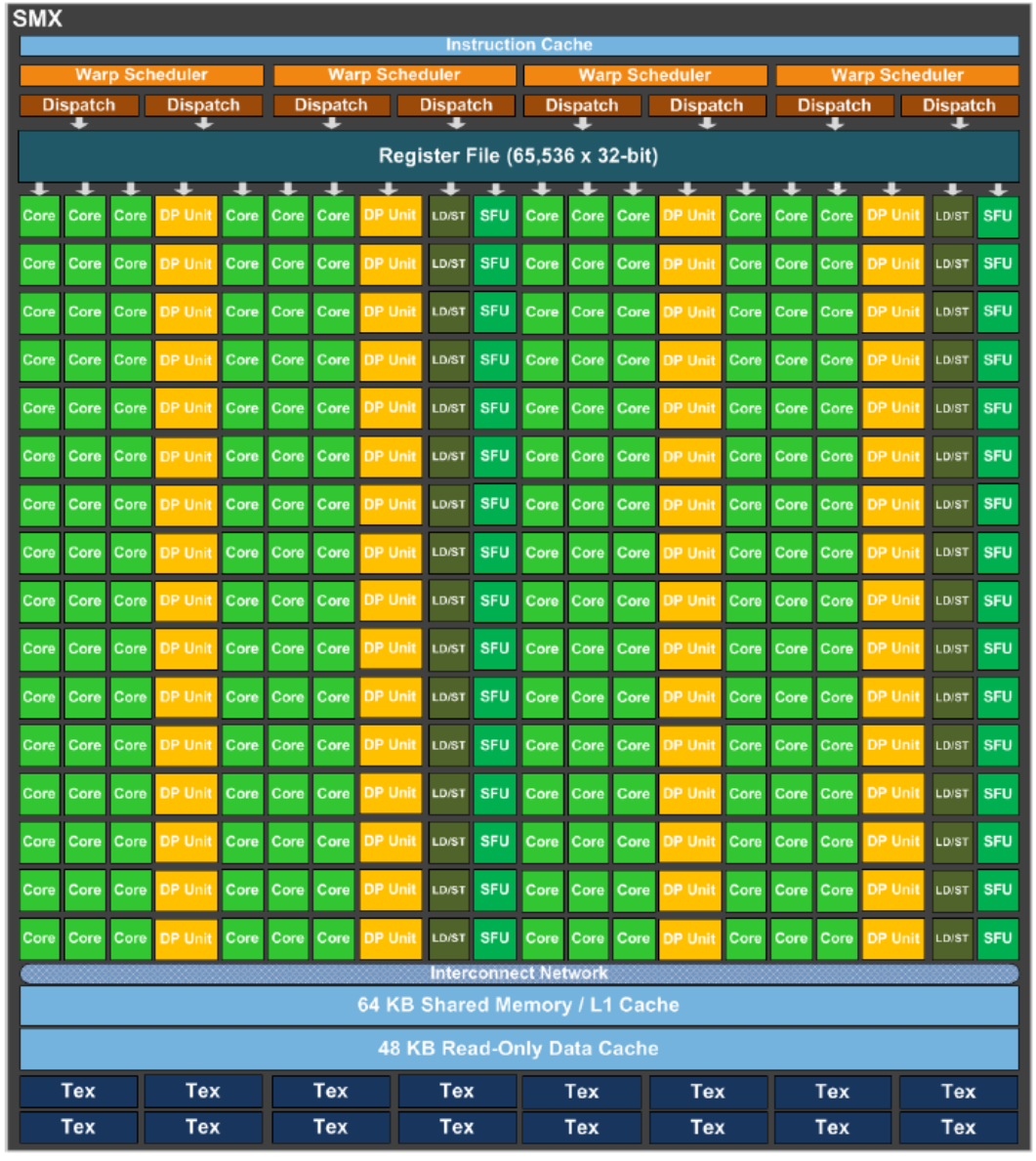
I linguaggi di programmazione disponibili nell'ambiente di sviluppo per CUDA, costituiscono delle estensioni dei linguaggi di programmazione più diffusi. Il principale è `CUDA-C` (C con estensioni NVIDIA), altri sono estensioni di Python, Fortran, Java e MATLAB.

In questo laboratorio utilizzeremo **esclusivamente** `CUDA-C`.

NVIDIA GeForce GTX 980

NVIDIA Kepler GK110

Architettura dello Streaming Multiprocessor (SMX)



Single thread vs. Threads

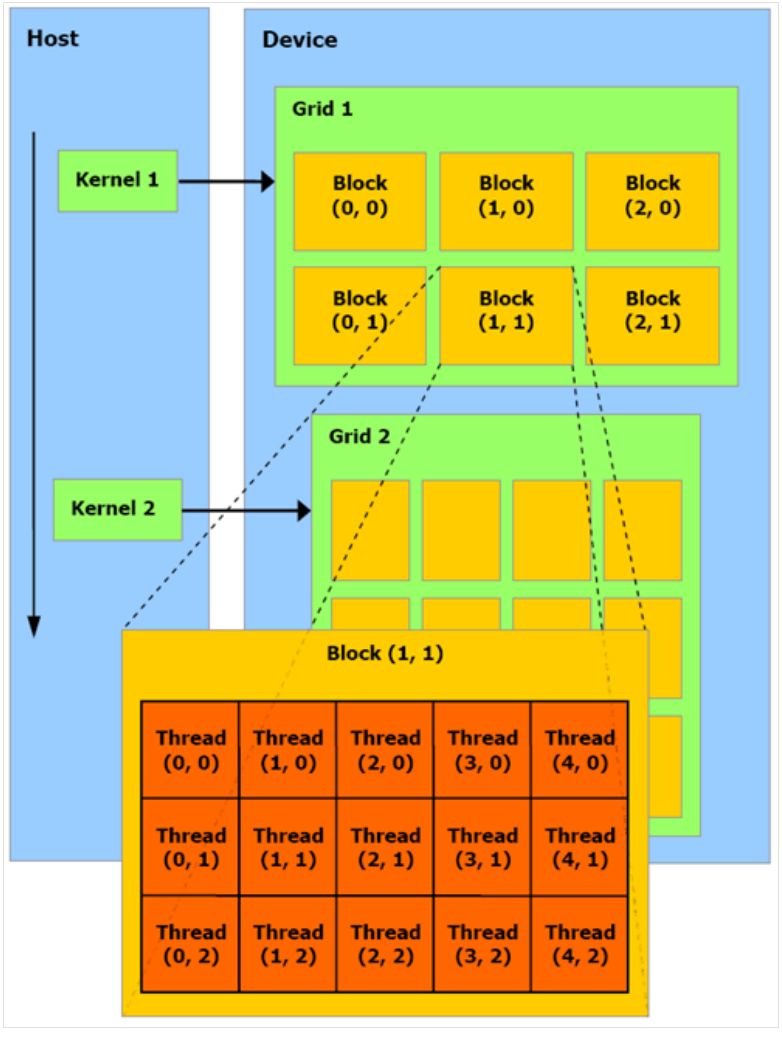
CPU (single core): un solo processore.

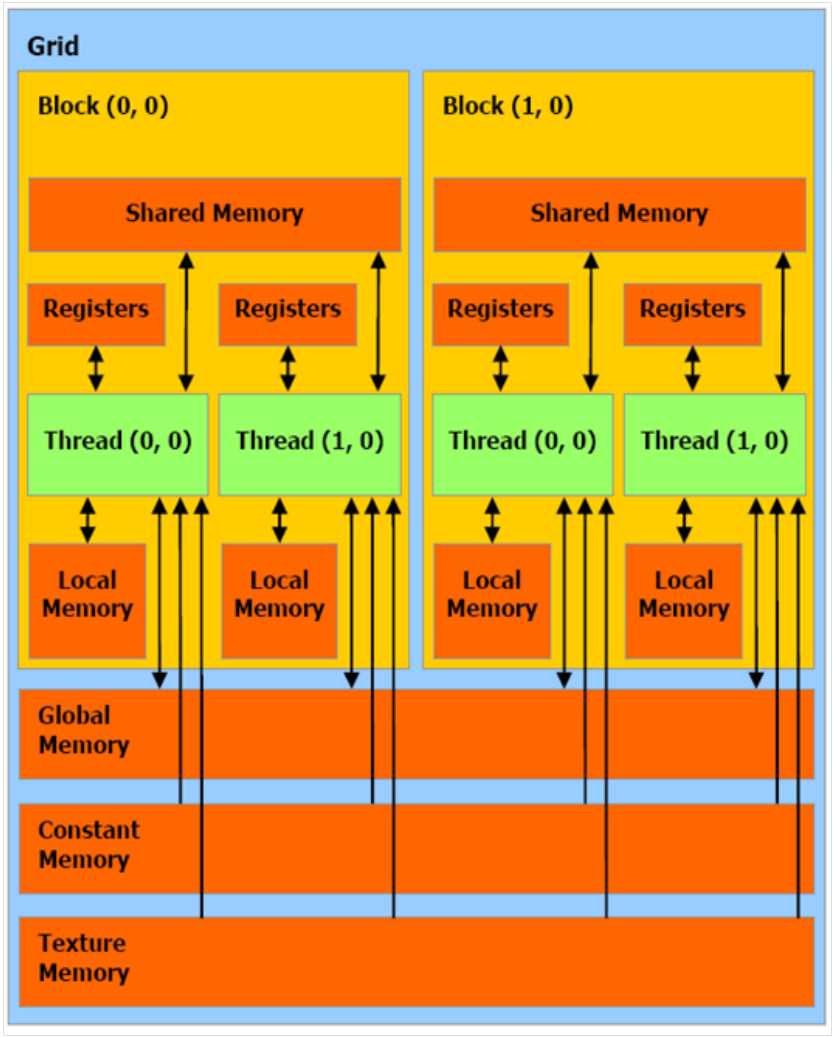
CPU (multi-core): più processori (2, 4, 8, 16).

CPU (cluster): più macchine.

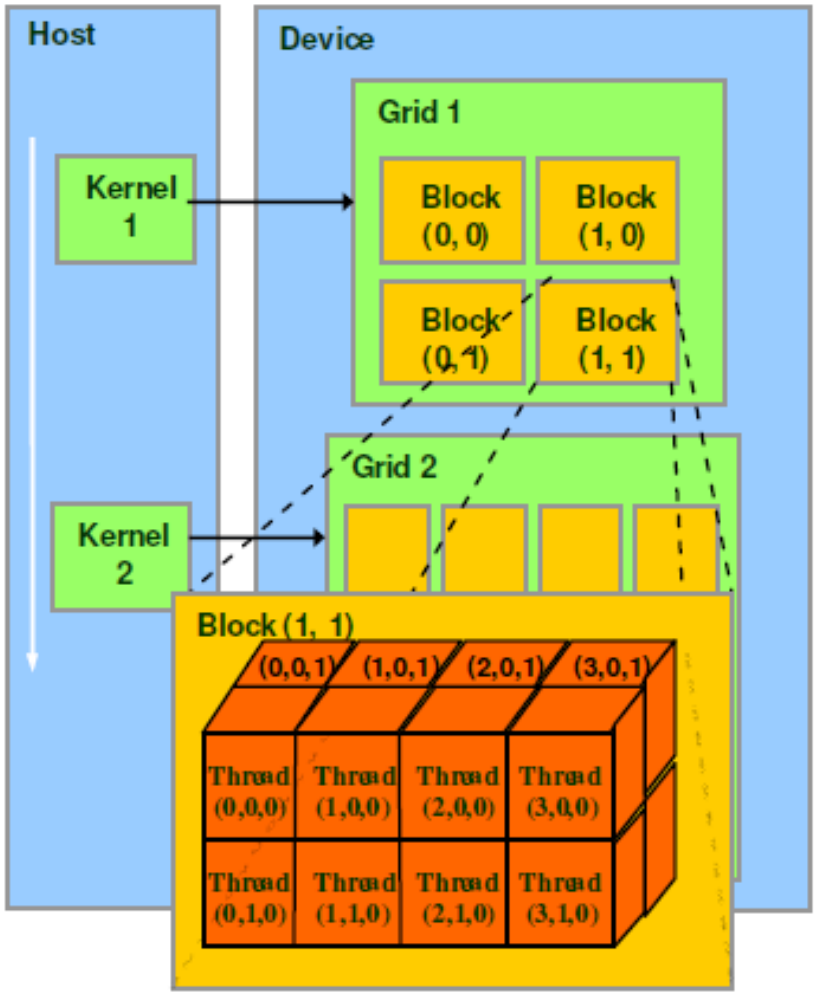
CPU + GPU: la CPU, sfrutta i *microprocessori* della GPU.

Organizzazione dei threads

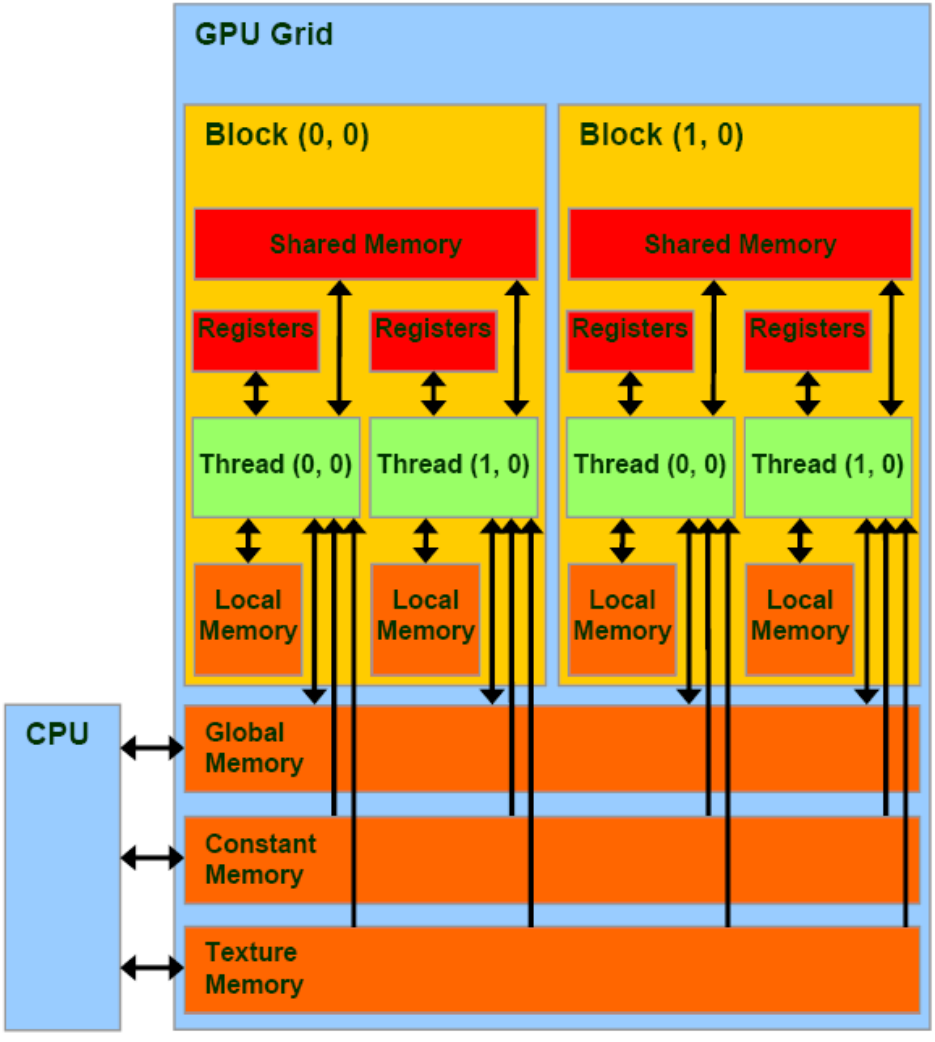




Grid, Blocks e Threads



Gestione delle memorie



NVIDIA Kepler GK110

13 Streaming Multiprocessor (SMX)

CUDA Compute Capability **3.5**

SMX

- 192 single-precision CUDA cores
- 64 double-precision units
- 32 special function units (SFU)
- 32 load/store units (LD/ST)

NVIDIA Tesla K20c

- Name: Tesla K20c
- Compute capability: 3.5
- Global memory available on device in bytes: 5032706048
- Constant memory available on device in bytes: 65536
- Maximum pitch in bytes allowed by memory copies: 2147483647
- Shared memory available per block in bytes: 49152
- Size of L2 cache in bytes: 1310720
- 32-bit registers available per block: 65536

NVIDIA Tesla K20c

- Clock frequency in kilohertz: 705500
- Number of multiprocessors on device: 13
- Maximum number of threads per block: 1024
- Maximum size of each dimension of a block: (1024, 1024, 64)
- Maximum size of each dimension of a grid: (2147483647, 65535, 65535)
- Warp size in threads: 32
- Maximum resident threads per multiprocessor: 2048

CUDA-C: alcuni concetti di base

Qualificatori per le **routines**:

```
__global__ void routine_da_CPU_a_GPU() { }
__device__ float routine_da_GPU_a_GPU() { }
```

Qualificatori per le **variabili**:

```
__constant__ float vettore_memoria_costant[32];
__shared__ float vettore_memoria_shared[32];
```

Configurare l'esecuzione:

```
dim3 grid_dim(10, 5);
dim3 block_dim(4, 8, 8);
my_kernel <<< grid_dim, block_dim >>> ();
```

Variabili **built-in**:

```
dim3 gridDim;
dim3 blockDim;
dim3 blockIdx;
dim3 threadIdx;
void __syncthreads();
```

Gestione delle variabili sul DEVICE (GPU)

Dichiarazione	**Memoria**	**Visibilità**	**Durata**
int var	registro	thread	thread
int var[10]	local	thread	thread
`__shared__ int var`	shared	block	block
`__global__ int var`	global	grid	programma

Ok, ma come funziona?

Bene, ora siamo pronti a *sporcarci* le mani!

Vediamo come sfruttare la GPU per accelerare alcune semplici routine (*kernel*) tramite CUDA.

Le implementazioni presentate **NON** sono ottimali!

my_cuda.h

```
1  void HandleError(cudaError_t err, const char *file,
       int line)
2  {
3    if (err != cudaSuccess)
4      {
5        printf("%s in %s at line %d\n",
       cudaGetErrorString( err ), file, line);
6        exit( EXIT_FAILURE );
7      }
8  }
9
10 #define HANDLE_ERROR( err ) (HandleError( err,
       __FILE__, __LINE__ ))
```

somma_CPU.cu I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /*  */
5  void sum(int x, int y, int *z);
6
7  /* ### */
8  int main()
9  {
10   int x = 17, y = 13, *z;
11
12   z = (int *) malloc(sizeof(int));
13
14   sum(x, y, z);
15
16   printf("%d + %d = %d\n", x, y, *z);
17   free(z);
18   return 0;
19 }
```

somma_CPU.cu II

```
20 /* ### */
21 void sum(int x, int y, int *z)
22 {
23    *z = x+y;
24    return;
25 }
```

somma_GPU.cu I

```
1  #include <stdio.h>
2  #include <cuda.h>
3  #include "my_cuda.h"
4
5  /*  */
6  __global__ void sum(int x, int y, int *z);
7
8  /* ### */
9  int main()
10 {
11    int x = 17, y = 13, *z;
12    int *d_z;
13
14    z = (int *) malloc(sizeof(int));
15    HANDLE_ERROR(cudaMalloc((void**)&d_z, sizeof(int)));
16
17    sum<<<1, 1>>>(x, y, d_z);
18
```

somma_GPU.cu II

```
19    HANDLE_ERROR(cudaMemcpy(z, d_z, sizeof(int),
        cudaMemcpyDeviceToHost));
20
21    printf("%d + %d = %d\n", x, y, *z);
22
23    free(z);
24    cudaFree(d_z);
25
26    return (0);
27  }
28  /* ### */
29  __global__ void sum(int x, int y, int *z)
30  {
31    *z = x+y;
32    return;
33  }
```

somma_vettori_CPU.cu I

```
1  #include <stdio.h>
2
3  #define SIZE 10000000
4
5  /* */
6  void add(double *x, double *y, double *z);
7
8  /* ### */
9  int main()
10 {
11   long  int i, istep;
12   double *x, *y, *z;
13
14   x = (double *) malloc(SIZE*sizeof(double));
15   y = (double *) malloc(SIZE*sizeof(double));
16   z = (double *) malloc(SIZE*sizeof(double));
17
18
19
```

somma_vettori_CPU.cu II

```
20    for(i = 0; i < SIZE; i++) {
21      x[i] = (double)rand()/RAND_MAX*10;
22      y[i] = (double)rand()/RAND_MAX*10;
23    }
24
25    add(x, y, z);
26
27    istep = SIZE/10;
28    for(i = 0; i < SIZE; i += istep) {
29      printf("[%ld] %le + %le = %le\n", i, x[i], y[i], z
        [i]);
30    }
31
32    free(x);
33    free(y);
34    free(z);
35
36    return 0;
37  }
38
```

somma_vettori_CPU.cu III

```
39 /* ### */
40 void add(double *x, double *y, double *z)
41 {
42   int id = 0;
43
44   while(id < SIZE)  {
45     z[id] = x[id]+y[id];
46     id += 1;
47   }
48   return;
49 }
```

somma_vettori_GPU.cu I

```
1  #include <stdio.h>
2  #include <cuda.h>
3  #include "my_cuda.h"
4
5  #define SIZE 10000000
6
7  /*  */
8  __global__ void add(double *x, double *y, double *z);
9
10 /* ### */
11 int main()
12 {
13   long  int i, istep;
14   double *x, *y, *z;
15   double *d_x, *d_y, *d_z;
16
17   x = (double *) malloc(SIZE*sizeof(double));
18   y = (double *) malloc(SIZE*sizeof(double));
19   z = (double *) malloc(SIZE*sizeof(double));
```

somma_vettori_GPU.cu II

```
20
21    HANDLE_ERROR(cudaMalloc((void**)&d_x, SIZE*sizeof(
        double)));
22    HANDLE_ERROR(cudaMalloc((void**)&d_y, SIZE*sizeof(
        double)));
23    HANDLE_ERROR(cudaMalloc((void**)&d_z, SIZE*sizeof(
        double)));
24
25    for(i = 0; i < SIZE; i++) {
26      x[i] = (double)rand()/RAND_MAX*10.;
27      y[i] = (double)rand()/RAND_MAX*10.;
28    }
29
30    HANDLE_ERROR(cudaMemcpy(d_x, x, SIZE*sizeof(double),
        cudaMemcpyHostToDevice));
31    HANDLE_ERROR(cudaMemcpy(d_y, y, SIZE*sizeof(double),
        cudaMemcpyHostToDevice));
32
33    add<<<2048*10,1024>>>(d_x, d_y, d_z);
34
```

somma_vettori_GPU.cu III

```
35    HANDLE_ERROR(cudaMemcpy(z, d_z, SIZE*sizeof(double),
          cudaMemcpyDeviceToHost));
36
37    istep = SIZE/10;
38    for(i = 0; i < SIZE; i += istep) {
39      printf("[%ld] %le + %le = %le\n", i, x[i], y[i], z
        [i]);
40    }
41
42    free(x);
43    free(y);
44    free(z);
45
46    cudaFree(d_x);
47    cudaFree(d_y);
48    cudaFree(d_z);
49
50    return(0);
51  }
52
```

somma_vettori_GPU.cu IV

```
53 /* ### */
54 __global__ void add(double *x, double *y, double *z)
55 {
56   int id = blockIdx.x*blockDim.x+threadIdx.x;
57   while(id < SIZE)  {
58     z[id] = x[id]+y[id];
59     id += blockDim.x*gridDim.x;
60   }
61   return;
62 }
```

prodotto_scalare.cu I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda.h>
4  #include "my_cuda.h"
5
6  #define MIN(a,b) ((a)<(b)?(a):(b))
7  #define N 10000000
8  #define NBLOCKS  1024
9  #define NTHREADS 512
10
11 typedef double DOUBLE;
12
13 __global__ void scalar_product(DOUBLE *x, DOUBLE *y,
       DOUBLE *z);
14
15
16
17
18
```

prodotto_scalare.cu II

```
19 /* ### */
20 int main()
21 {
22   int i;
23   DOUBLE *x, *y, *ztmp, z, zcheck;
24   DOUBLE *d_x, *d_y, *d_ztmp;
25
26   x = (DOUBLE *) malloc(N*sizeof(DOUBLE));
27   y = (DOUBLE *) malloc(N*sizeof(DOUBLE));
28   ztmp = (DOUBLE *) malloc(NBLOCKS*sizeof(DOUBLE));
29
30   HANDLE_ERROR(cudaMalloc((void**)&d_x, N*sizeof(
       DOUBLE)));
31   HANDLE_ERROR(cudaMalloc((void**)&d_y, N*sizeof(
       DOUBLE)));
32   HANDLE_ERROR(cudaMalloc((void**)&d_ztmp, NBLOCKS*
       sizeof(DOUBLE)));
33
34
35
```

prodotto_scalare.cu III

```
36    for(i = 0; i < N; ++i) {
37      x[i] = (DOUBLE) i;
38      y[i] = i*2.;
39    }
40
41    HANDLE_ERROR(cudaMemcpy(d_x, x, N*sizeof(DOUBLE),
        cudaMemcpyHostToDevice));
42    HANDLE_ERROR(cudaMemcpy(d_y, y, N*sizeof(DOUBLE),
        cudaMemcpyHostToDevice));
43
44    scalar_product<<<NBLOCKS, NTHREADS>>>(d_x, d_y,
        d_ztmp);
45
46    HANDLE_ERROR(cudaMemcpy(ztmp, d_ztmp, NBLOCKS*sizeof
        (DOUBLE), cudaMemcpyDeviceToHost));
47
48    z = 0;
49    for(i = 0; i < NBLOCKS; ++i)
50      z += ztmp[i];
51
```

prodotto_scalare.cu IV

```
52    zcheck = (N-1.)*(N)*(2.*N-1)/3.;
53    printf("GPU value %24.16le\nExpected value %24.16le\
        n", z, zcheck);
54
55    HANDLE_ERROR(cudaFree(d_x));
56    HANDLE_ERROR(cudaFree(d_y));
57    HANDLE_ERROR(cudaFree(d_ztmp));
58
59    free(x);
60    free(y);
61    free(ztmp);
62  }
63
64
65
66
67
68
69
70
```

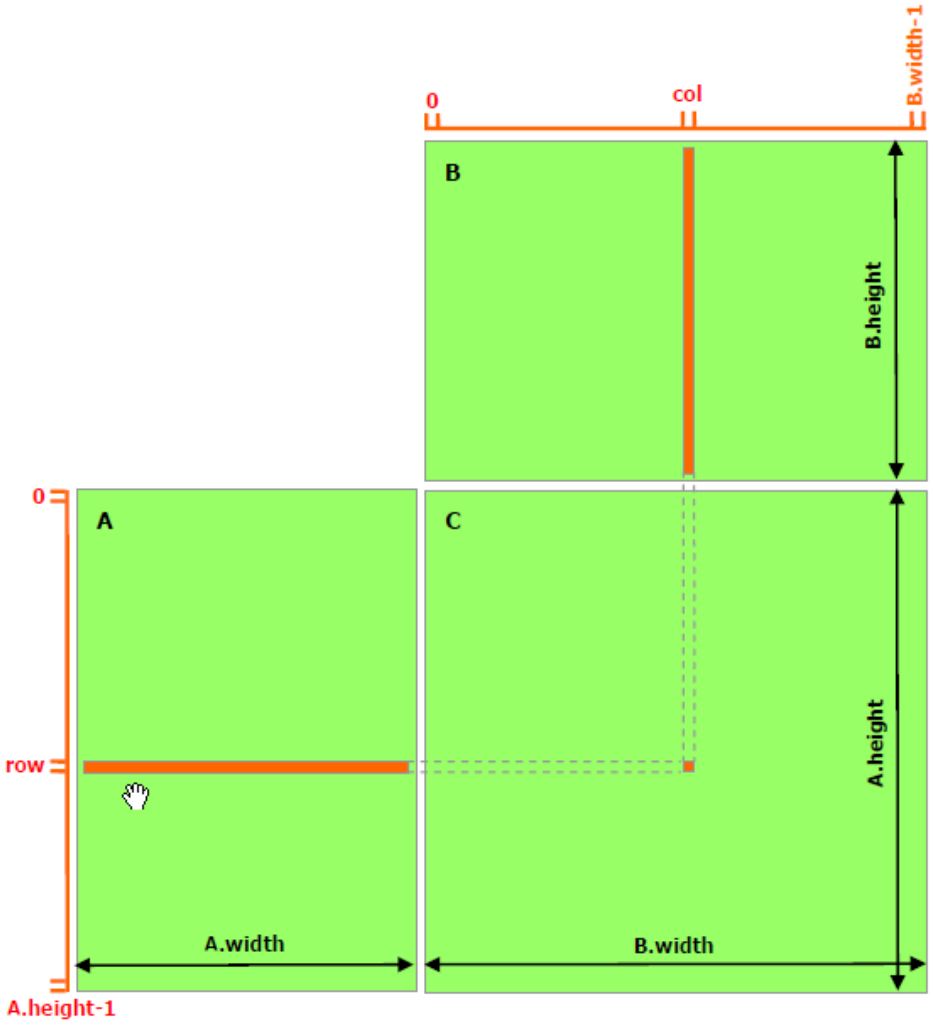
prodotto_scalare.cu V

```
71 /* ### */
72 __global__ void scalar_product(DOUBLE *x, DOUBLE *y,
       DOUBLE *z)
73 {
74   __shared__ float ztmp[NTHREADS];
75   int id, tid;
76   unsigned int j;
77   DOUBLE tmp;
78
79   id = threadIdx.x+blockIdx.x*blockDim.x;
80   tid = threadIdx.x;
81
82   tmp = 0.;
83   while(id < N) {
84     tmp += x[id]*y[id];
85     id += blockDim.x*gridDim.x;
86   }
87
88
89
```

prodotto_scalare.cu VI

```
90    ztmp[tid] = tmp;
91
92    __syncthreads();
93
94    j = blockDim.x;
95    while((j >>= 1) != 0) {
96      if(tid < j)
97        ztmp[tid] += ztmp[tid+j];
98      __syncthreads();
99    }
100
101   if(tid == 0)
102     z[blockIdx.x] = ztmp[tid];
103
104   return;
105 }
```

Prodotto di Matrici (global memory)



prodotto_matrici_CPU-1.cu I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  typedef struct {
6    unsigned int h;
7    unsigned int w;
8    unsigned int offset;
9    double *el;
10 } MATRIX;
11
12 #define NTEST 10
13
14
15
16
17
18
19
```

prodotto_matrici_CPU-1.cu II

```
20 /* ### */
21 int matrix_create_CPU(unsigned int w, unsigned int h,
       MATRIX *a);
22 void matrix_destroy_CPU(MATRIX *a);
23 void matrix_randomfill_CPU(MATRIX *a);
24 int matrix_product_CPU(MATRIX *a, MATRIX *b, MATRIX *c
       );
25 void matrix_export_octave(MATRIX *a, char *varname,
       char *filename);
26 void matrix_product_octave(char *filename);
27 void matrix_clear_octave(char *filename);
28
29
30
31
32
33
34
35
36
```

prodotto_matrici_CPU-1.cu III

```
37 /* ### */
38 int main()
39 {
40   MATRIX a, b, c;
41   unsigned int l, n;
42   clock_t start0, end0, delta0;
43   double msec;
44   char filename[256];
45   FILE *ofp;
46
47   sprintf(filename, "matrix_product_CPU-1_times.dat");
48   ofp = fopen(filename, "w");
49
50   for(l = 2; l <= 2048; l *= 2) {
51
52     if(matrix_create_CPU(l, l, &a) != 0) exit(-1);
53     if(matrix_create_CPU(l, l, &b) != 0) exit(-1);
54
55
56
```

prodotto_matrici_CPU-1.cu IV

```
57        /*   */
58        delta0 = 0.;
59        for(n = 0; n < NTEST; ++n) {
60
61          sprintf(filename, "octave_matrixprodCPU1_%04u
          -%02u.txt", l, n);
62
63          matrix_randomfill_CPU(&a);
64          matrix_randomfill_CPU(&b);
65
66          start0 = clock();
67          if(matrix_product_CPU(&a, &b, &c) != 0) exit(-1)
          ;
68          end0 = clock();
69          delta0 += end0-start0;
70
71          if(l <= 100 && n == 0) {
72            matrix_clear_octave(filename);
73            matrix_export_octave(&a, "A", filename);
74            matrix_export_octave(&b, "B", filename);
```

prodotto_matrici_CPU-1.cu V

```
75            matrix_export_octave(&c, "C", filename);
76            matrix_product_octave(filename);
77          }
78
79          matrix_destroy_CPU(&c);
80        }
81
82        matrix_destroy_CPU(&a);
83        matrix_destroy_CPU(&b);
84
85        msec = (double) delta0*1000./CLOCKS_PER_SEC/NTEST;
86
87        fprintf(ofp, "%08u\t%24.16le\n", l, msec);
88      }
89
90      fclose(ofp);
91
92      return 0;
93    }
94
```

prodotto_matrici_CPU-1.cu VI

```
95  /* ### */
96  int matrix_create_CPU(unsigned int h, unsigned int w,
        MATRIX *a)
97  {
98    size_t size;
99
100   a->h = h;
101   a->w = w;
102   a->offset = w;
103   size = h*w*sizeof(double);
104   a->el = malloc(size);
105
106   if(a->el == NULL) {
107     perror("ERROR: malloc failed (matrix_create_CPU)\n
        ");
108     return -1;
109   }
110
111   return 0;
112 }
```

prodotto_matrici_CPU-1.cu VII

```
113 /* ### */
114 void matrix_destroy_CPU(MATRIX *a)
115 {
116   a->h = 0;
117   a->w = 0;
118   a->offset = 0;
119   free(a->el);
120   a->el = NULL;
121   return;
122 }
123 /* ### */
124 void matrix_randomfill_CPU(MATRIX *a)
125 {
126   size_t i, size;
127 
128   size = a->w*a->h;
129   for(i = 0; i < size; ++i)
130     a->el[i] = drand48();
131   return;
132 }
```

prodotto_matrici_CPU-1.cu VIII

```
133 /* ### */
134 int matrix_product_CPU(MATRIX *a, MATRIX *b, MATRIX *c
        )
135 {
136   unsigned int i, j, l;
137   double tmp;
138 
139   if(a->w != b->h) {
140     perror("ERROR: incompatible matrices sizes (
        matrix_product_CPU)\n");
141     return -1;
142   }
143 
144   if(matrix_create_CPU(a->h, b->w, c) != 0) {
145     perror("ERROR: matrix creation for the product
        failed (matrix_product_CPU)\n");
146     return -2;
147   }
148 
149 
```

prodotto_matrici_CPU-1.cu IX

```
150     for(i = 0; i < c->h; ++i)
151       for(j = 0; j < c->w; ++j) {
152         tmp = 0.;
153         for(l = 0; l < a->w; ++l)
154           tmp += a->el[i*a->w+l]*b->el[l*b->w+j];
155         c->el[i*c->w+j] = tmp;
156       }
157 
158     return 0;
159 }
160 /* ### */
161 void matrix_export_octave(MATRIX *a, char *varname,
        char *filename)
162 {
163     int i, j;
164     FILE *ofp;
165 
166     ofp = fopen(filename, "a");
167 
168     fprintf(ofp, "%s", varname);
```

prodotto_matrici_CPU-1.cu X

```
169     fprintf(ofp, " = [");
170
171     for(i = 0; i < a->h; ++i) {
172       for(j = 0; j < a->w; ++j)
173         fprintf(ofp, "%24.16le%s", a->el[i*a->w+j], (j==
        a->w-1)?";\n":",");
174     }
175     fprintf(ofp, "];\n");
176
177     fclose(ofp);
178
179     return;
180 }
181 /* ### */
182 void matrix_product_octave(char *filename)
183 {
184     FILE *ofp;
185
186     ofp = fopen(filename, "a");
187     fprintf(ofp, "C1 = A*B;\n");
```

prodotto_matrici_CPU-1.cu XI

```
188     fprintf(ofp, "diff=C-C1;\n");
189     fclose(ofp);
190 
191     return;
192 }
193 /* ### */
194 void matrix_clear_octave(char *filename)
195 {
196     FILE *ofp;
197 
198     ofp = fopen(filename, "w");
199     fprintf(ofp, "# Created by matrix_product_CPU-1\n");
200     fclose(ofp);
201 
202     return;
203 }
```

prodotto_matrici_CPU-2.cu I

```
1  /* ### */
2  int matrix_transpose_CPU(MATRIX *a, MATRIX *at)
3  {
4    int i, j;
5
6    if(matrix_create_CPU(a->w, a->h, at) != 0) {
7      perror("ERROR: matrix creation for the transpose
       failed (matrix_transpose_CPU)\n");
8      return -1;
9    }
10
11   for(i = 0; i < at->h; ++i)
12     for(j = 0; j < at->w; ++j)
13       at->el[i*at->w+j] = a->el[j*a->w+i];
14
15   return 0;
16 }
17
18
```

prodotto_matrici_CPU-2.cu II

```
19 /* ### */
20 int matrix_product_CPU(MATRIX *a, MATRIX *b, MATRIX *c
       )
21 {
22   unsigned int i, j, l;
23   MATRIX bt;
24   double tmp;
25
26   if(a->w != b->h) {
27     perror("ERROR: incompatible matrices sizes (
       matrix_product_CPU)\n");
28     return -1;
29   }
30
31   if(matrix_create_CPU(a->h, b->w, c) != 0) {
32     perror("ERROR: matrix creation for the product
       failed (matrix_product_CPU)\n");
33     return -2;
34   }
35
```

prodotto_matrici_CPU-2.cu III

```
36    if(matrix_transpose_CPU(b, &bt) != 0) {
37      perror("ERROR: matrix transpose for the product
        failed (matrix_product_CPU)\n");
38      return -3;
39    }
40
41    for(i = 0; i < c->h; ++i)
42      for(j = 0; j < c->w; ++j) {
43        tmp = 0.;
44        for(l = 0; l < a->w; ++l)
45  tmp += a->el[i*a->w+l]*bt.el[j*bt.w+l];
46        c->el[i*c->w+j] = tmp;
47      }
48
49    matrix_destroy_CPU(&bt);
50
51    return 0;
52  }
53
```

prodotto_matrici_GPU-1.cu I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include "my_cuda.h"
5
6  typedef struct {
7    unsigned int h;
8    unsigned int w;
9    unsigned int offset;
10   double *el;
11 } MATRIX;
12
13 #define NTEST 10
14 #define NBLOCKS 1024
15 #define NTHREADS 256
16
17
18
19
```

prodotto_matrici_GPU-1.cu II

```
20 /* ### */
21 int matrix_create_CPU(unsigned int w, unsigned int h,
       MATRIX *a);
22 void matrix_destroy_CPU(MATRIX *a);
23 void matrix_randomfill_CPU(MATRIX *a);
24 int matrix_product_CPU(MATRIX *a, MATRIX *b, MATRIX *c
       );
25 void matrix_export_octave(MATRIX *a, char *varname,
       char *filename);
26 void matrix_product_octave(char *filename);
27 void matrix_clear_octave(char *filename);
28 void matrix_create_GPU(unsigned int h, unsigned int w,
        MATRIX *d_a);
29 void matrix_destroy_GPU(MATRIX *d_a);
30 void matrix_copy_to_GPU(MATRIX *a, MATRIX *d_a);
31 void matrix_copy_from_GPU(MATRIX *d_a, MATRIX *a);
32 int matrix_product_GPU(MATRIX *a, MATRIX *b, MATRIX *c
       );
33 __global__ void ker_matrix_product(MATRIX d_a, MATRIX
       d_b, MATRIX d_c);
```

prodotto_matrici_GPU-1.cu III

```
34 /* ### */
35 int main()
36 {
37   MATRIX a, b, c;
38   unsigned int l, n;
39   clock_t start0, end0, delta0;
40   double msec;
41   char filename[256];
42   FILE *ofp;
43
44   sprintf(filename, "matrix_product_GPU-1_times.dat");
45   ofp = fopen(filename, "w");
46
47   for(l = 2; l <= 2048; l *= 2) {
48
49     if(matrix_create_CPU(l, l, &a) != 0) exit(-1);
50     if(matrix_create_CPU(l, l, &b) != 0) exit(-1);
51
52
53
```

prodotto_matrici_GPU-1.cu IV

```
54        /*   */
55        delta0 = 0.;
56        for(n = 0; n < NTEST; ++n) {
57
58          sprintf(filename, "octave_matrixprodGPU1_%04u
          -%02u.txt", l, n);
59
60          matrix_randomfill_CPU(&a);
61          matrix_randomfill_CPU(&b);
62
63          start0 = clock();
64          if(matrix_product_GPU(&a, &b, &c) != 0) exit(-1)
        ;
65          end0 = clock();
66          delta0 += end0-start0;
67
68          if(l <= 100 && n == 0) {
69  matrix_clear_octave(filename);
70  matrix_export_octave(&a, "A", filename);
71  matrix_export_octave(&b, "B", filename);
```

prodotto_matrici_GPU-1.cu V

```
72  matrix_export_octave(&c, "C", filename);
73  matrix_product_octave(filename);
74        }
75
76        matrix_destroy_CPU(&c);
77      }
78
79      matrix_destroy_CPU(&a);
80      matrix_destroy_CPU(&b);
81
82      msec = (double) delta0*1000./CLOCKS_PER_SEC/NTEST;
83
84      fprintf(ofp, "%08u\t%24.16le\n", l, msec);
85    }
86
87    fclose(ofp);
88
89    return 0;
90  }
91
```

prodotto_matrici_GPU-1.cu VI

```
92  /* ### */
93  void matrix_create_GPU(unsigned int h, unsigned int w,
        MATRIX *d_a)
94  {
95    size_t size;
96
97    d_a->h = h;
98    d_a->w = w;
99    d_a->offset = w;
100   size = h*w*sizeof(double);
101   HANDLE_ERROR(cudaMalloc((void **)&(d_a->el), size));
102
103   return;
104 }
105
106
107
108
109
110
```

prodotto_matrici_GPU-1.cu VII

```
111 /* ### */
112 void matrix_destroy_GPU(MATRIX *d_a)
113 {
114   d_a->h = 0;
115   d_a->w = 0;
116   HANDLE_ERROR(cudaFree(d_a->el));
117   d_a->el = NULL;
118 }
119
120
121
122
123
124
125
126
127
128
129
130
```

prodotto_matrici_GPU-1.cu VIII

```
131 /* ### */
132 void matrix_copy_to_GPU(MATRIX *a, MATRIX *d_a)
133 {
134   size_t size;
135
136   size = a->w*a->h*sizeof(double);
137   HANDLE_ERROR(cudaMemcpy(d_a->el, a->el, size,
        cudaMemcpyHostToDevice));
138   return;
139 }
140 /* ### */
141 void matrix_copy_from_GPU(MATRIX *d_a, MATRIX *a)
142 {
143   size_t size;
144
145   size = d_a->w*d_a->h*sizeof(double);
146   HANDLE_ERROR(cudaMemcpy(a->el, d_a->el, size,
        cudaMemcpyDeviceToHost));
147   return;
148 }
```

prodotto_matrici_GPU-1.cu IX

```
149 /* ### */
150 int matrix_product_GPU(MATRIX *a, MATRIX *b, MATRIX *c
        )
151 {
152   MATRIX d_a, d_b, d_c;
153
154   if(a->w != b->h) {
155     perror("ERROR: incompatible matrices sizes (
        matrix_product_GPU)\n");
156     return -1;
157   }
158
159   if(matrix_create_CPU(a->h, b->w, c) != 0) {
160     perror("ERROR: matrix creation for the product
        failed (matrix_product_GPU)\n");
161     return -2;
162   }
163
164   matrix_create_GPU(a->h, a->w, &d_a);
165   matrix_create_GPU(b->h, b->w, &d_b);
```

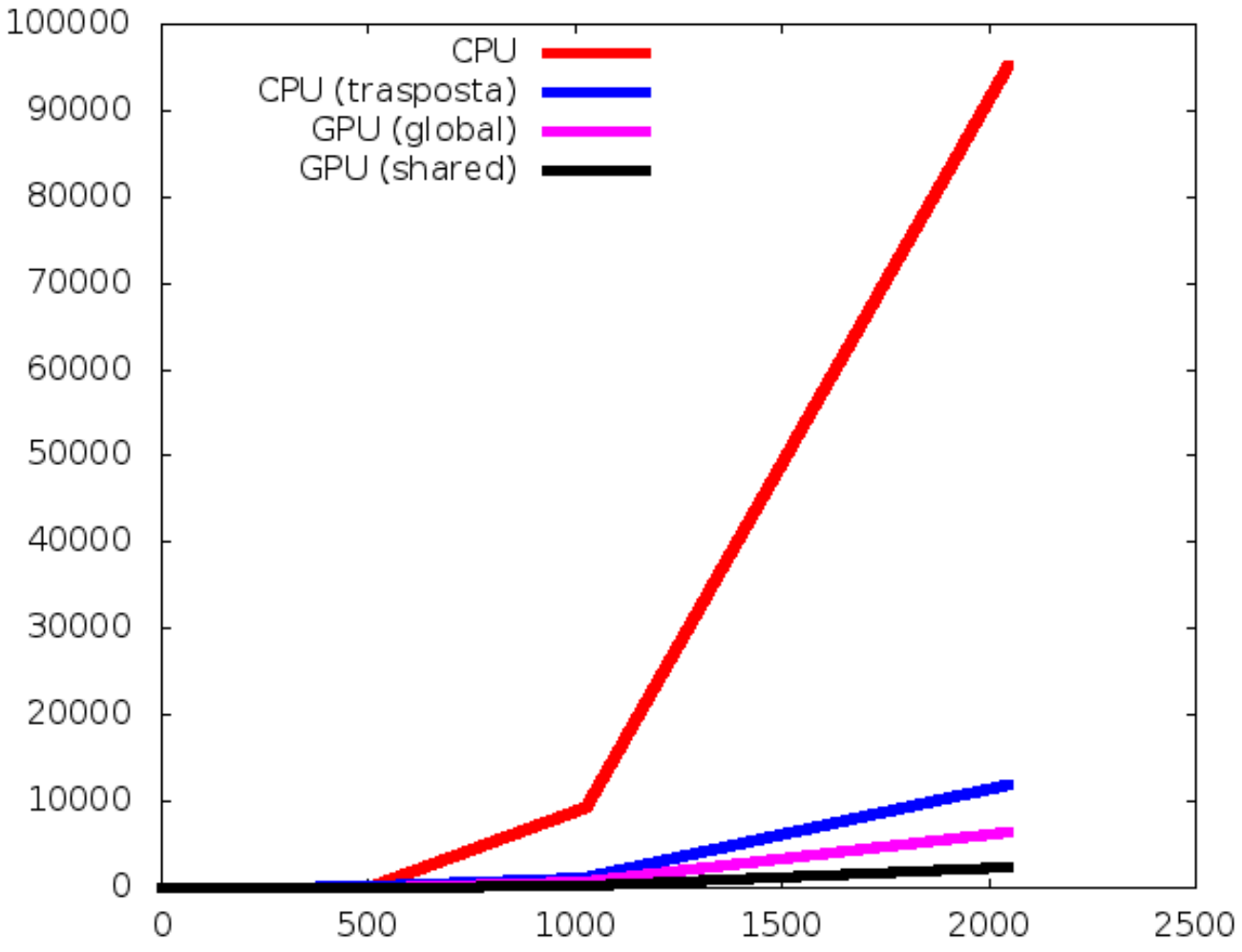
prodotto_matrici_GPU-1.cu X

```
166     matrix_create_GPU(c->h, c->w, &d_c);
167
168     matrix_copy_to_GPU(a, &d_a);
169     matrix_copy_to_GPU(b, &d_b);
170
171     ker_matrix_product<<< NBLOCKS, NTHREADS >>>(d_a, d_b
          , d_c);
172
173     matrix_destroy_GPU(&d_a);
174     matrix_destroy_GPU(&d_b);
175
176     matrix_copy_from_GPU(&d_c, c);
177
178     matrix_destroy_GPU(&d_c);
179
180     return 0;
181 }
182
183
184
```

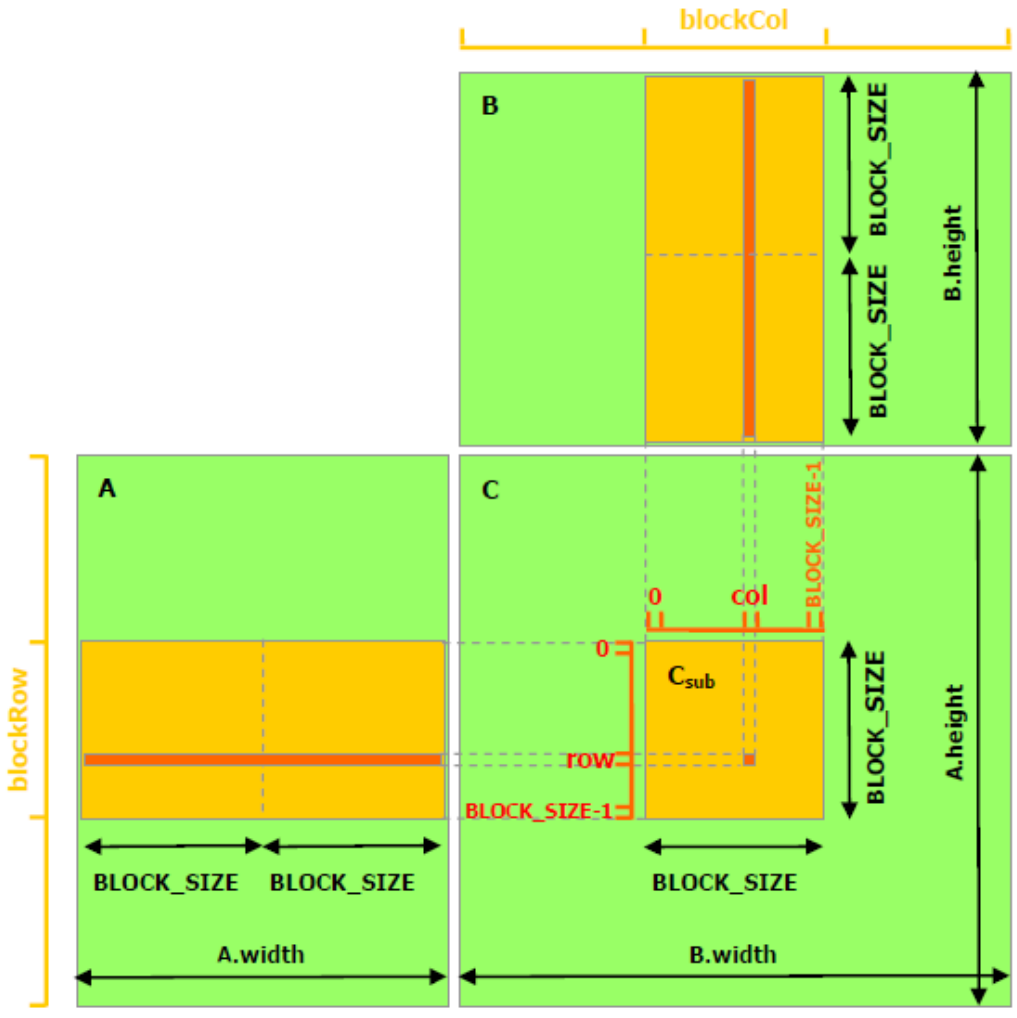
prodotto_matrici_GPU-1.cu XI

```
185 /* ### */
186 __global__ void ker_matrix_product(MATRIX d_a, MATRIX
        d_b, MATRIX d_c)
187 {
188   double tmp;
189   unsigned int idx, i, j, k, size;
190
191   idx = blockIdx.x*blockDim.x+threadIdx.x;
192   size = d_c.h*d_c.w;
193   while(idx < size) {
194     i = idx/d_c.w;  j = idx%d_c.w;
195     tmp = 0.;
196     for(k = 0; k < d_a.w; ++k)
197       tmp += d_a.el[i*d_a.w+k] * d_b.el[k*d_b.w+j];
198     d_c.el[idx] = tmp;
199     idx += blockDim.x*gridDim.x;
200   }
201   return;
202 }
```

Prodotto di Matrici (CPU vs. GPU)



Prodotto di Matrici (shared memory)



prodotto_matrici_GPU-1.cu I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include "my_cuda.h"
5
6  typedef struct {
7    unsigned int h;
8    unsigned int w;
9    unsigned int offset;
10   double *el;
11 } MATRIX;
12
13 #define NTEST 10
14 #define NBLOCKS 1024
15 #define NTHREADS 16
16
17
18
19
```

prodotto_matrici_GPU-1.cu II

```
20
21  /* ### */
22  __device__ double get_elem(const MATRIX d_a, int row,
        int col) {
23    return d_a.el[row*d_a.offset+col];
24  }
25  /* ### */
26  __device__ void set_elem(MATRIX d_a, int row, int col,
        double value)
27  {
28    d_a.el[row*d_a.offset+col] = value;
29  }
30
31
32
33
34
35
36
37
```

prodotto_matrici_GPU-1.cu III

```
38 /* ### */
39 __device__ MATRIX get_submatrix(MATRIX d_a, int row,
       int col)
40 {
41   MATRIX d_sa;
42   d_sa.w = NTHREADS;
43   d_sa.h = NTHREADS;
44   d_sa.offset = d_a.offset;
45   d_sa.el = d_a.el+(d_a.offset*NTHREADS*row+NTHREADS*
       col);
46   return d_sa;
47 }
48
49
50
51
52
53
54
55
```

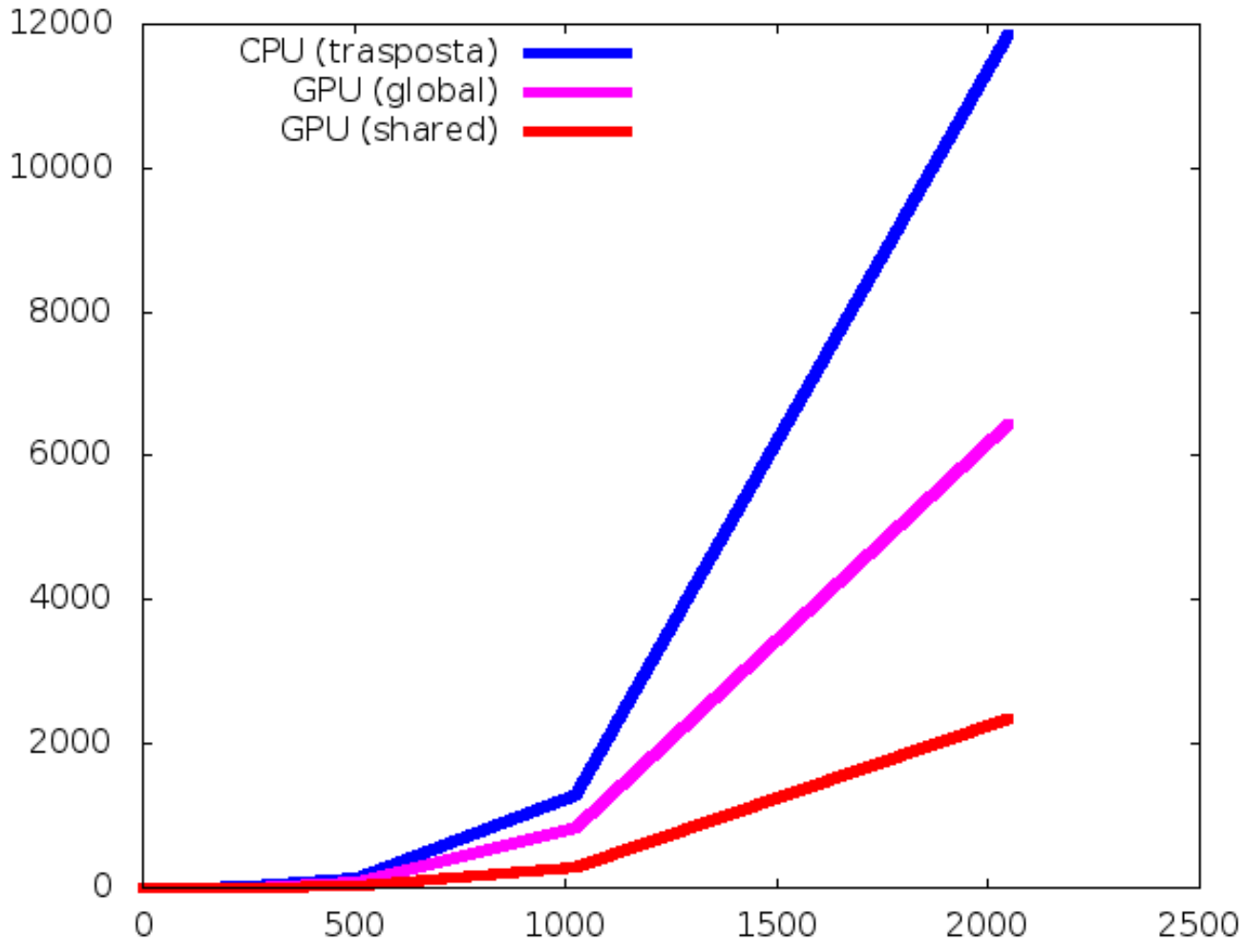
prodotto_matrici_GPU-1.cu IV

```
56 /* ### */
57 __global__ void ker_matrix_product(MATRIX d_a, MATRIX
       d_b, MATRIX d_c)
58 {
59   MATRIX d_asub, d_bsub, d_csub;
60   unsigned int i, j, row, col, blkrow, blkcol;
61   double cvalue;
62   __shared__ double asub_el[NTHREADS][NTHREADS];
63   __shared__ double bsub_el[NTHREADS][NTHREADS];
64
65   row = threadIdx.y;
66   col = threadIdx.x;
67
68   blkrow = blockIdx.y;
69   blkcol = blockIdx.x;
70
71   d_csub = get_submatrix(d_c, blkrow, blkcol);
72   cvalue = 0.;
73
74
```

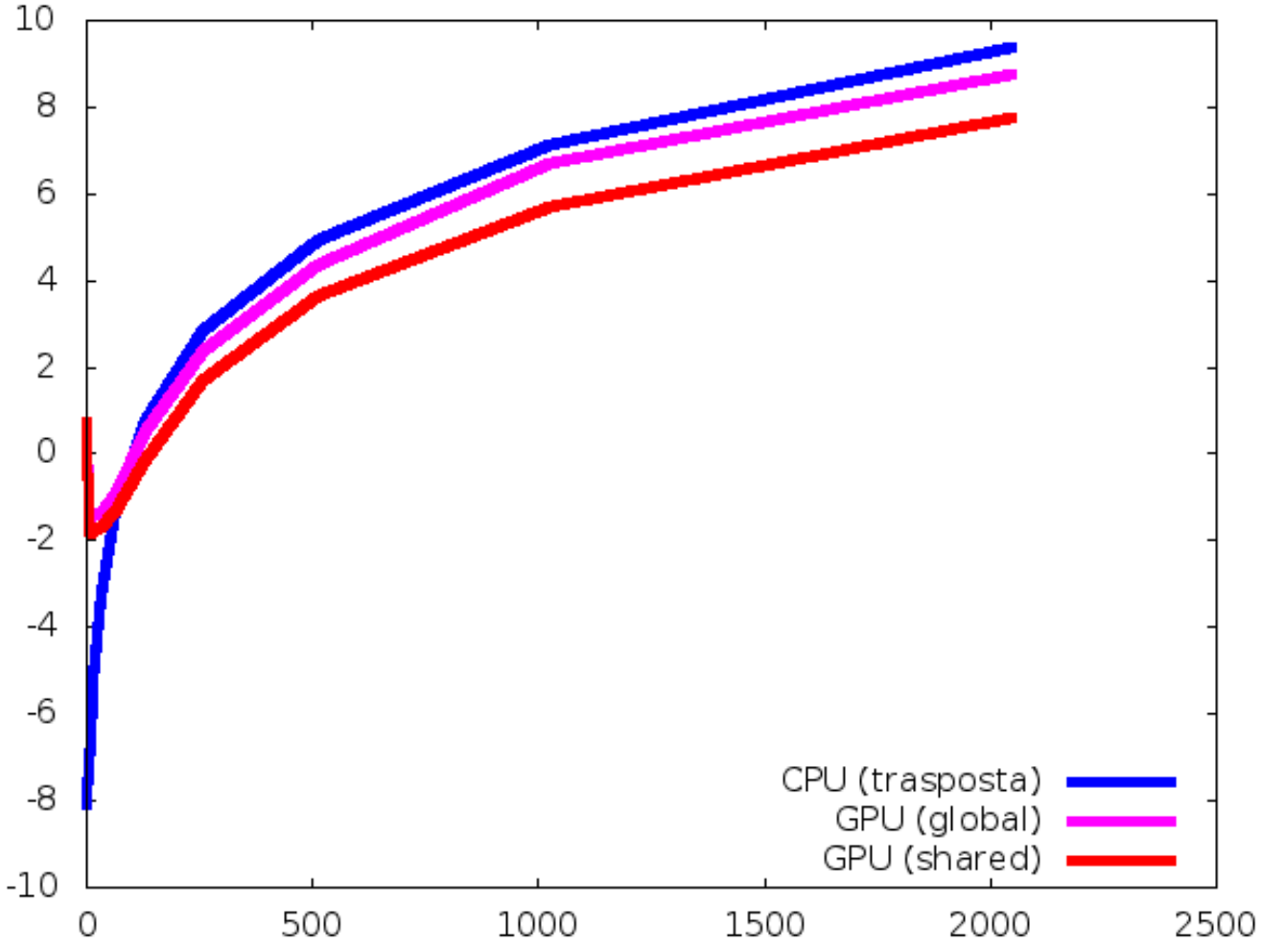
prodotto_matrici_GPU-1.cu V

```
75    for (i = 0; i < d_a.w/NTHREADS; ++i) {
76      d_asub = get_submatrix(d_a, blkrow, i);
77      d_bsub = get_submatrix(d_b, i, blkcol);
78
79      asub_el[row][col] = get_elem(d_asub, row, col);
80      bsub_el[row][col] = get_elem(d_bsub, row, col);
81
82      __syncthreads();
83
84      for(j = 0; j < NTHREADS; ++j)
85        cvalue += asub_el[row][j]*bsub_el[j][col];
86      __syncthreads();
87    }
88
89    set_elem(d_csub, row, col, cvalue);
90
91    return;
92  }
```

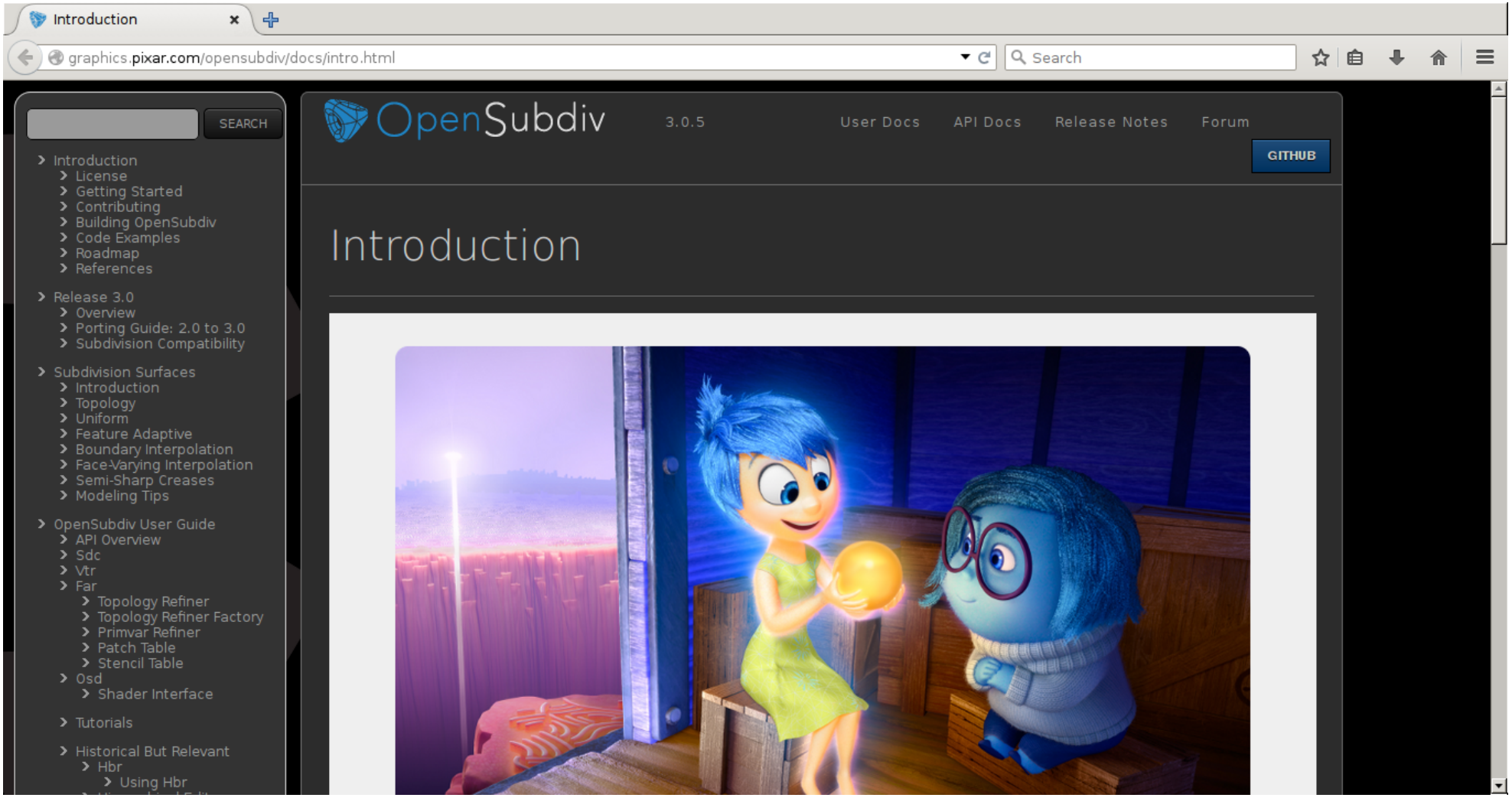
Prodotto di Matrici (CPU vs. global vs. shared)



Prodotto di Matrici (CPU vs. global vs. shared)



Non si vive di soli conti...



Grazie per l'attenzione!

Domande?

Commenti?

http://www.mat.unimi.it/users/sansotte/cuda/