GPGPU

GPGPU

- **GPGPU** (**G**eneral **P**urpose computation using **GPU**)**:** uso del processore delle schede grafice (GPU) per scopi differenti da quello tradizionale delle generazione di immagini 3D

- Primi esperimenti d'uso delle GPU nell'ambito del calcolo General Purpose, sono stati fatti usando direttamente libreria per la grafica come ad esempio OpnGL

- In anni piu' recenti e' naturamente aumentata la versatilita' ed usabilita' delle GPU

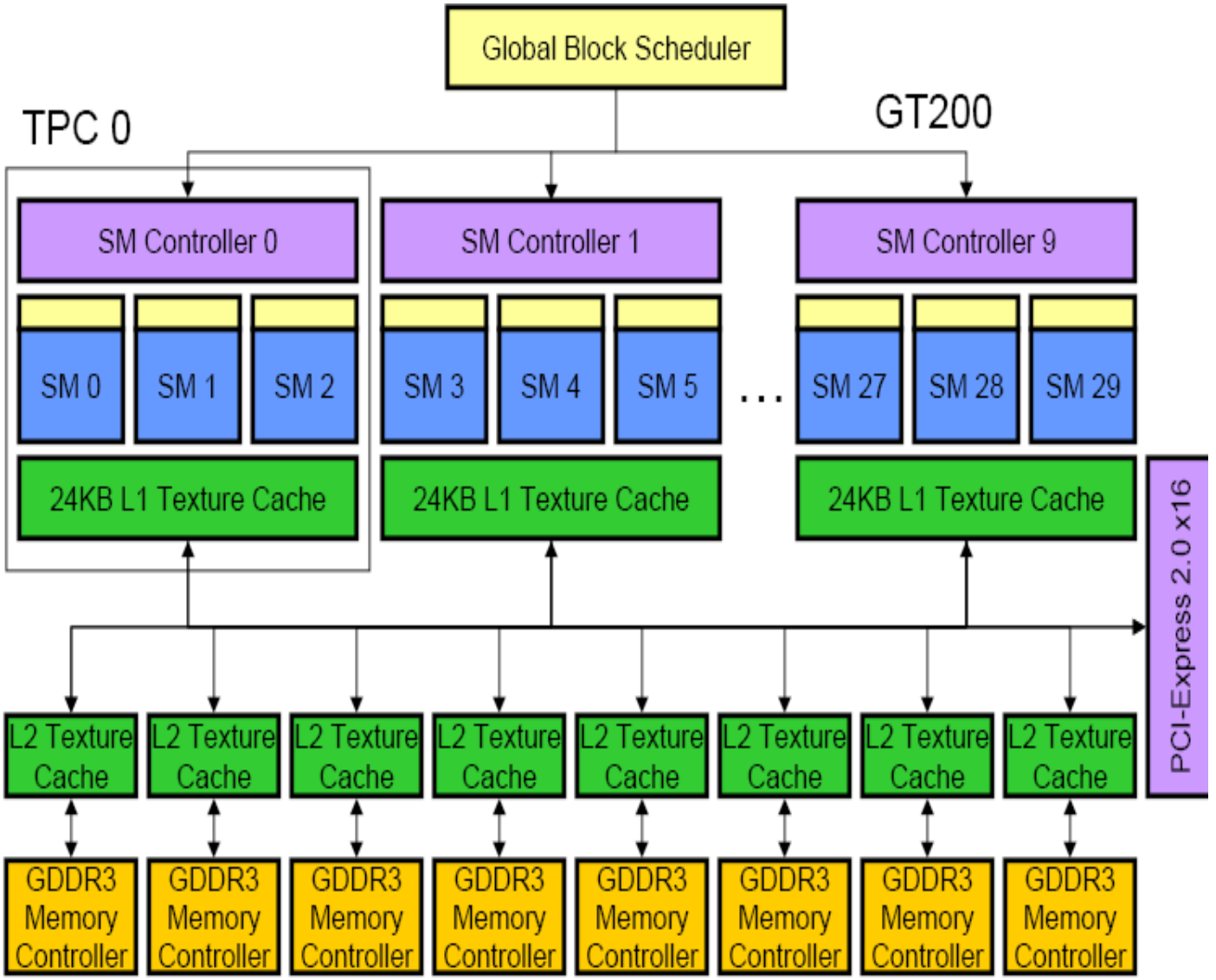
-

GPGPU - CUDA

- **CUDA** (un acronimo per **Compute Unified Device Architecture)** e' un'architettura di calcolo parallelo sviluppato da NVIDIA. CUDA e' il motore di calcolo per le GPU NVIDIA accessibile allo sviluppatore di software attraverso linguaggio di programmazione C.

- I driver piu' recenti contengono tutti i componenti necessari CUDA. CUDA funziona con tutte le GPU NVIDIA a partire dalla serie G8X.

GPGPU – GT200

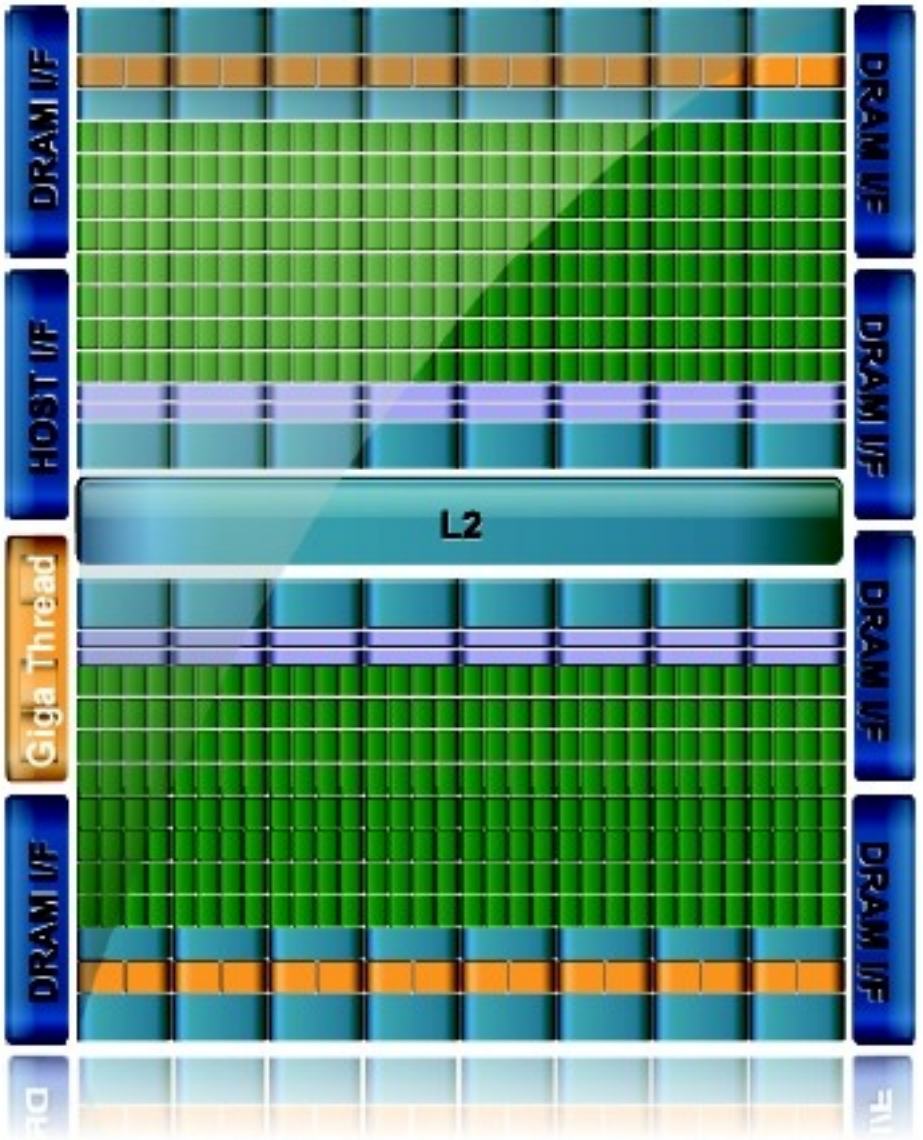


Thread Processing Clusters (TPC)

Streaming Multiprocessors (SM)

Ogni SM e' a sua volta composta da 8 Streaming Processors (SP)

GPGPU – Nvidia Fermi



3 miliardi di transistor

Piu' di 4000 core

Prestazioni din doppia precisione di picco 8x

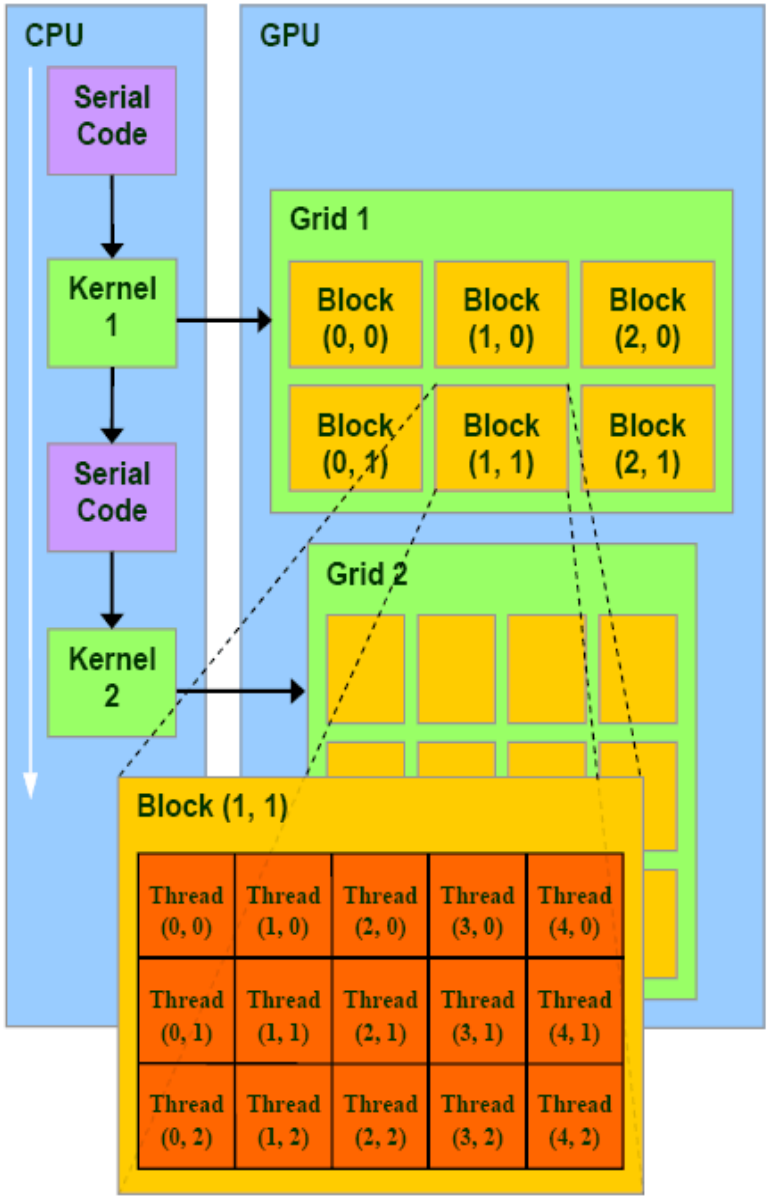
Memoria ECC

Caches L1 ed L2

Piu' banda verso la memoria

Fino ad 1 TB di memoria possibile

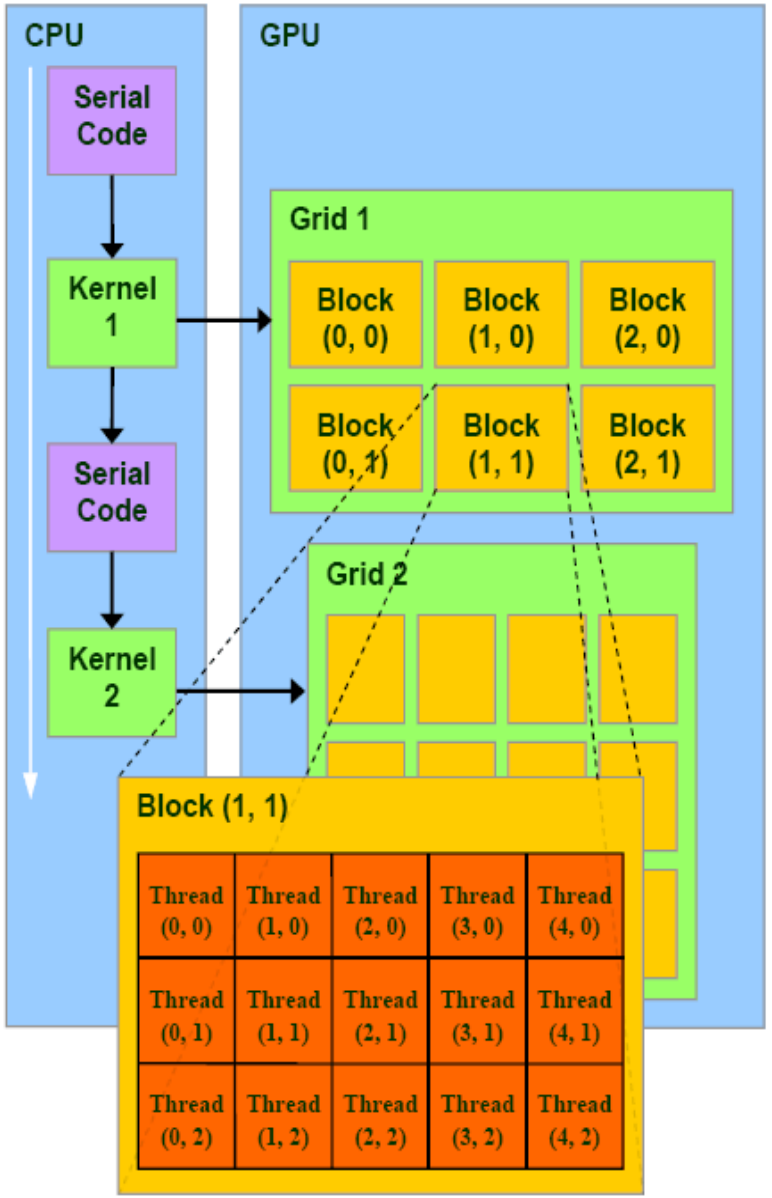
GPGPU – CUDA



Modello d'esecuzione (1)

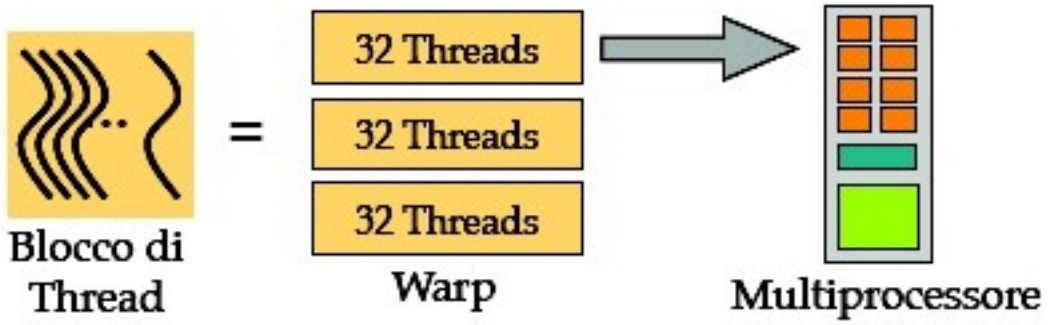
- Nella visione CUDA la GPU e' un coprocessore “highly threaded”
- Le GPU sono utili in casi di carichi di lavoro fortemente “data parallel”
- Le porzioni di codice che vengono eseguite dalla GPU sono note come Kernels
- Un kernel viene eseguito da una griglia di “thread blocks”
- Ogni “thread block” e' un groppo di threads che partono dallo stesso indirizzo di istruzioni, vengono eseguiti in parallelo e possono comunicare o mediante la “shared memory” oppure barriere di sincronizzazione
- Threads all'interno dello stesso blocco condividono dati, dunque devono essere eseguiti all'interno dello stesso Streaming Multiprocessor o SM
- Ogni singolo thread e' eseguito all'interno di un singolo Streaming Processor o SP core.

GPGPU – CUDA

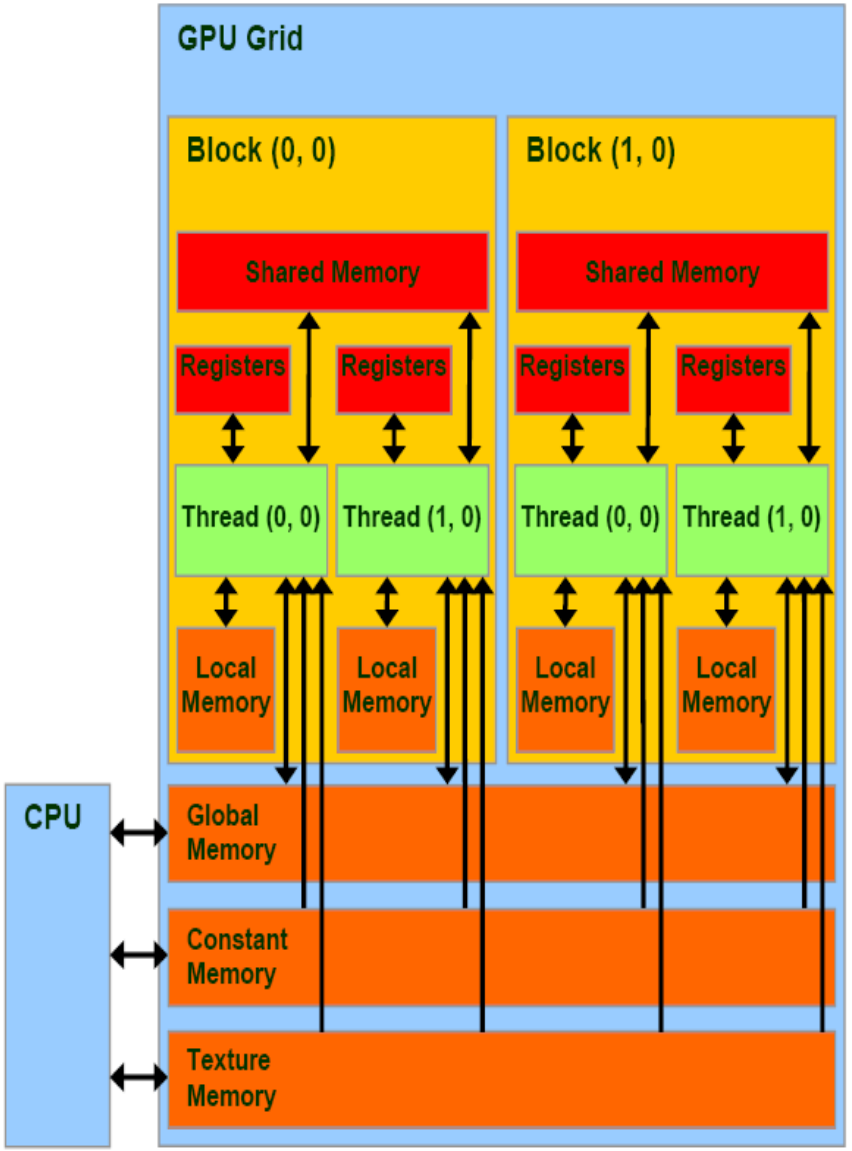


Modello d'esecuzione (2)

- La “griglia” è lanciata sulla GPU
- I blocchi di thread sono serialmente distribuiti su tutti gli SM
  - Potentialmente >1 blocco di thread per SM
- Ogni SM esegue “warp” di thread
  - 2 livelli di parallelismo
- Ogni SM schedula i warp “pronti” per l’esecuzione
- Quando i warp ed i blocchi completano vengono liberate le relative risorse
- La GPU può distribuire nuovi blocchi di thread



GPGPU – CUDA

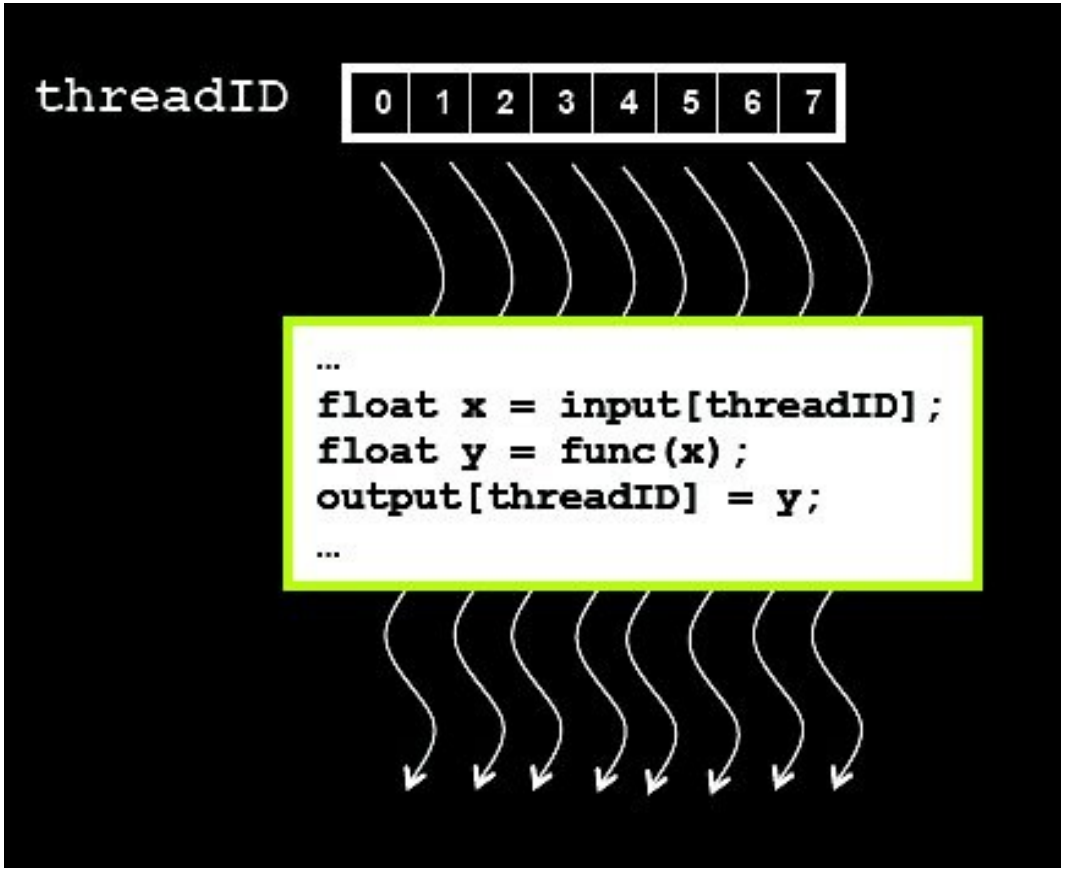


Modello di memoria

- Registi (fast on-chip) e “Local Memory” (DRAM) sono privati per ogni thread.
- “Shared Memory” (fast on-chip) e' un'area di memoria usata per la comunicazione fra thread
- “Constant memory” generalmente di piccole dimensioni usata per l'accesso random (ad esempio istruzioni)
- “Texture memory” anche questa e' read-only ma di dimensioni decisamente maggiori
- Sia la “Texture memory” che la “Constant Memory” sono cached on-chip
- “Global Memory” e' di grandi dimensioni e visibile a tutti i thread della griglia. Puo' essere scritta e letta da GPU e CPU.

GPGPU – CUDA

•Ogni CUDA kernel e' eseguito da un array di threads
  • Tutti i threads eseguono lo stesso codice
  • Ogni thread ha un ID che viene usato per calcolare indirizzi di memoria e prendere decisioni



```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```

```
dim3 grid, block;
grid.x = 2; grid.y = 4;
block.x = 8; block.y = 16;

kernel<<<grid, block>>>(...);
```

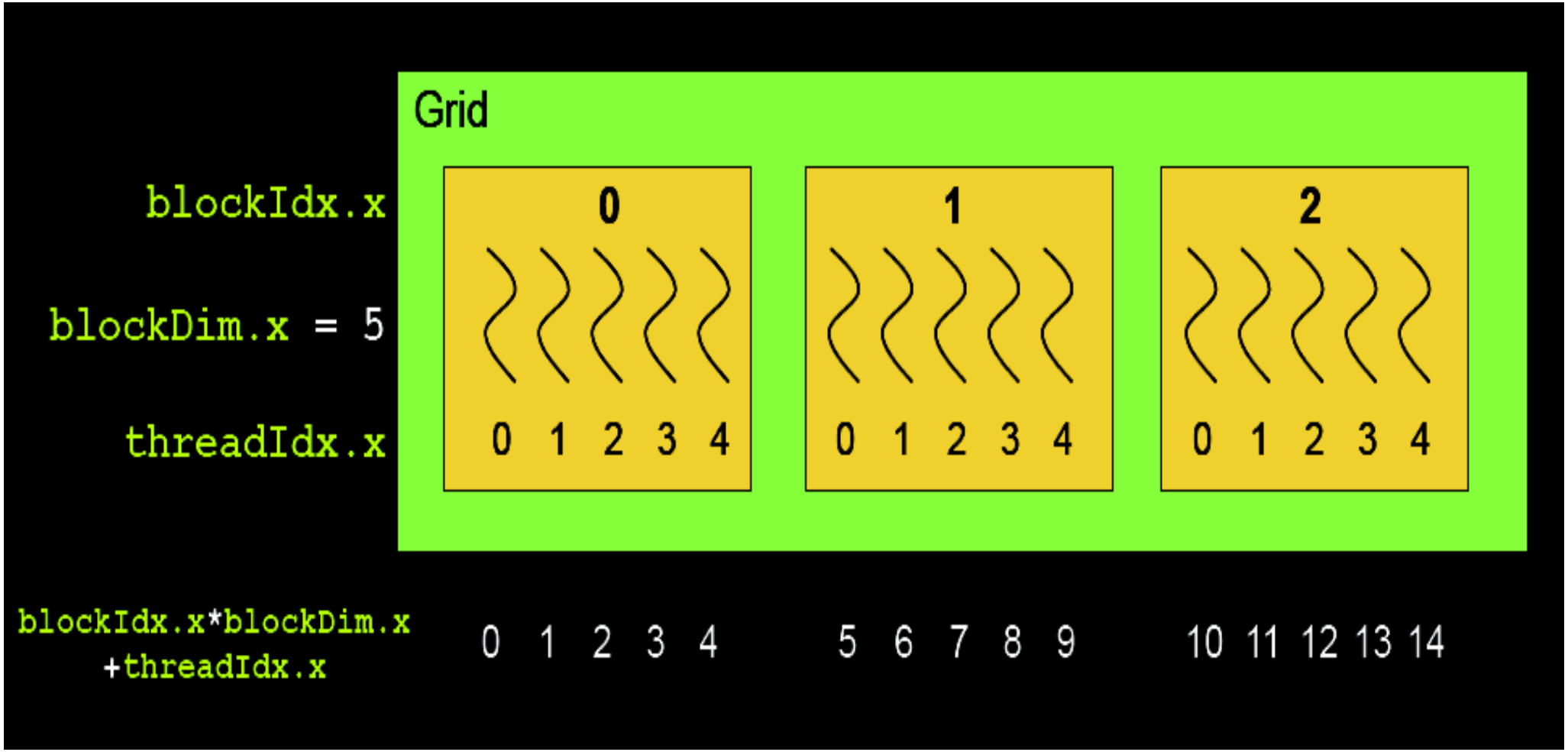
```
dim3 grid(2, 4), block(8,16);

kernel<<<grid, block>>>(...);
```

```
kernel<<<32,512>>>(...);
```

GPGPU – CUDA

•Alcune variabili sono usate per determinare un ID unico, tale ID potra' poi essere usato come indice di array



GPGPU – CUDA

- Esempio incremento elementi di un vettore

```
void increment_cpu(float *a, float b, int N)
{

        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}
```

```
__global__ void increment_gpu(float *a, float b, int N)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N)
            a[idx] = a[idx] + b;
}
```

```
void main()
{
    .....
      increment_cpu(a, b, N);
}
```

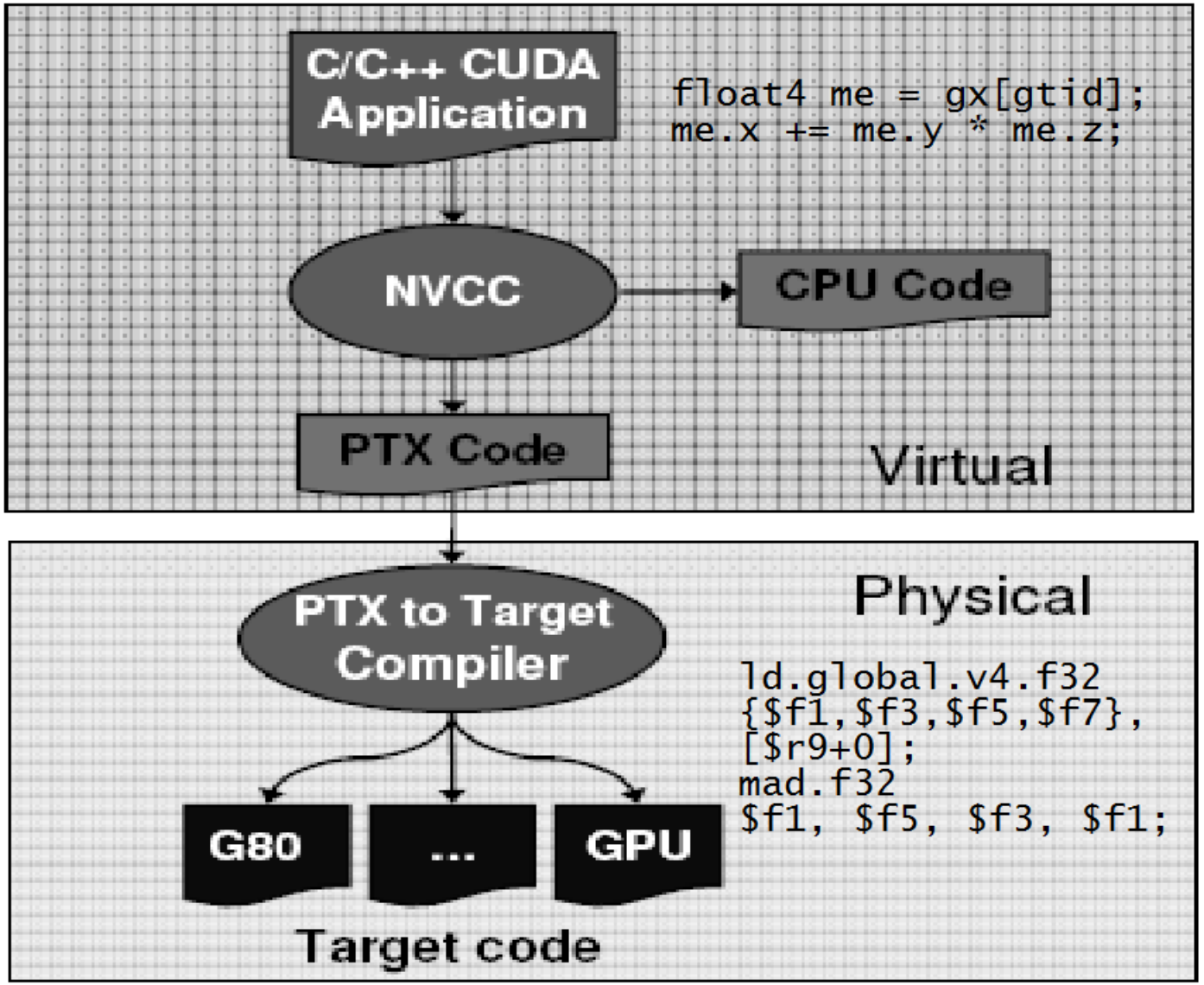
```
void main()
{
    .....
      dim3 dimBlock (blocksize);
      dim3 dimGrid( ceil( N / (float)blocksize)  );
      increment_gpu<<<dimGrid, dimBlock>>>(ad, bd, N);
}
```

GPGPU – CUDA

La compilazione alcuni dettagli:

- Ogni file sorgente contenente estensioni CUDA deve essere compilato con NVCC
- NVCC è un front-end per la compilazione
  - invoca tutti gli strumenti ed i compilatori richiesti g++, cl, cudacc, ...
  - NVCC produce codice C code (eseguito dalla CPU), da compilare utilizzando un compilatore C (GNU)
  - NVCC produce anche direttamente codice oggetto per la GPU, oppure sorgente PTX interpretato a tempo di esecuzione
- Alcune opzioni utili:
  - --ptxas-options=-v Mostra l'utilizzo dei registri e della memoria (shared e costante)
  - --maxrregcount <N> Limita il numero di registri (dettagli a proposito dello spilling dei registri)
  - -arch sm_13 (o sm_20) Attiva la possibilità di utilizzare la doppia precisione (se supportata)

GPGPU – CUDA



PTX: Parallel Trhead eXecution

Il PTX e' il set di istruzioni virtuali per i chip grafici NVIDIA, definisce anche risorse, stato etc.

Lo step successivo e' eseguito dal compilatore stesso o dall'interprete a runtime CUDA, ed il codice PTX viene trasformato in codice binario dell'architettura fisica

GPGPU – CUDA

•Qualsiasi eseguibile che contiene codice CUDA richiede almeno due librerie dinamiche:

- La libreria CUDA runtime (cudart)
- La libreria CUDA “base” (cuda)

•Sono disponibili molte altre librerie:

- cutil (utilità di base)
- cuBLAS (algebra lineare)
- cuFFT (Fast Fourier Transform)
- cudpp (primitive “data parallel”)

GPGPU – CUDA

Esempi, Esercizi e cenni di multigpu e shared memory.

```
__global__ void kernel(...)
{
    …
    __shared__ float sData[256];
    …
}

int main(void)
{
    …
    kernel<<<nBlocks, blockSize>>>(...);
    …
}
```

```
__global__ void kernel(...)
{
    …
    extern __shared__ float sData[];
    …
}

int main(void)
{
    …
    int smBytes = blockSize*sizeof(float);
    kernel<<<nBlocks, blockSize, smBytes>>>(...);
    …
}
```

OpenCL

•OpenCL (Open Computing Language) e' una libreria basata sul linguaggio di programmazione C99 che puo' esser eseguito su una molteplicita' di piattaforme, CPU, GPU, e altri tipi di processori.

•Standard iniziale di Apple successivamente ratificato anche da Intel, AMD e NVIDIA

•Lo standard e' stato poi completato dal gruppo no-profit Kronos

•Prevede una serie di chiamate per il discover delle risorse e successivamente per la compilazione ed esecuzione di kernel su una o piu' risorse