Introduction to Scientific Programming using GP-GPU (Graphics Processing Unit) and CUDA

Day 1

- Esercitazione:
  - Compilazione CUDA
  - Ambiente e utility: deviceQuery e nvidia-smi
  - Somma di vettori
  - Somma Matrici
  - Prodotto Matrici

I passi del compilatore CUDA

- Al compilatore va sempre specificata:
  - l'architettura virtuale con cui generare il *PTX code*
  - l'architettura reale per creare il codice oggetto (*cubin*)

```
nvcc -arch=compute_20 -code=sm_20,sm_21
```

selezione architettura virtuale (*PTX code*)

selezione architetture reali (*cubin*)

- nvcc ammette l'uso dello shortcut `nvcc -arch=sm_XX`.
  Esempio di GPU Fermi:

```
nvcc -arch=sm_20
```

equivalente a: `nvcc -arch=compute_20 -code=sm_20`

Esercizi

- deviceQuery (SDK): informazioni sui device grafici presenti
- nvidia-smi (NVIDIA System Management Interfage): Informazioni diagnostiche per GPU Tesla e alcune Quadro (*e.g.*: nvidia-smi –L; nvidia-smi –q; nvidia-smi –q –d UTILIZATION –l 1)
- nvcc –V
- “Compilare un programma CUDA”
  - cd Esercizi/VectorAdd. Compilare:
  - nvcc vectoradd_cuda.cu -o vectoradd_cuda
  - nvcc –arch=sm_20 vectoradd_cuda.cu -o vectoradd_cuda
  - nvcc –arch=sm_20 –ptx vectoradd_cuda.cu
  - nvcc –arch=sm_20 –keep vectoradd_cuda.cu -o vectoradd_cuda
  - nvcc –arch=sm_20 –keep -clean vectoradd_cuda.cu -o vectoradd_cuda
  - Eseguire:
  - ./vectoradd_cuda

Esercizi

- MatrixAdd:
  - il programma calcola la somma di due matrici: C = A + B
  - completare il sorgente CUDA C o CUDA Fortran
  - compilare ed eseguire
  - modificare la griglia di blocchi da (16,16) a (32,32) a (64,64)

- Controllo e prestazioni:
  - Gestione errore
  - Misura prestazioni

- Esercitazione:
  - Misura trasferimenti
  - Matrix-Matrix product
    - implementazione e misura prestazioni

Segnalazione degli errori CUDA alla CPU

- Tutte le chiamate API CUDA ritornano un codice di errore di tipo cudaError_t
  - il codice cudaSuccess significa che non si sono verificati errori
  - i kernel sono asincroni e void (non ritornano codice di errore)
  - le funzioni asincrone resituiscono un errore che si riferisce solo alla fase di chiamata da host

`char* cudaGetErrorString(cudaError_t code)`

- ritorna una stringa di caratteri (NULL-terminated) che descrive l'errore a partire dal codice passato

```
cudaError_t cerr = cudaMalloc(&d_a,size);

 if (cerr != cudaSuccess)
  fprintf(stderr, "%s\n", cudaGetErrorString(cerr));
```

Segnalazione degli errori CUDA alla CPU

- L'errore viene comunque mantenuto in una variabile interna modificata dalle funzioni in errore

`cudaError_t cudaGetLastError(void)`

- ritorna il codice di stato della variabile interna (cudaSuccess o altro)
- resetta la variabile interna: quindi l'errore che dà cudaGetLastError può riferirsi a una qualunque delle funzioni dalla precedente chiamata di cudaGetLastError stesso
- è necessaria prima una sincronizzazione chiamando **`cudaThreadSynchronize()`** in caso di controllo di funzioni asincrone (e.g., kernel)

```
cudaError_t cerr;
...
cerr = cudaGetLastError(); // reset internal state
kernelGPU<<<dimGrid,dimBlock>>>(arg1,arg2,...);
cudaThreadSynchronize();
cerr = cudaGetLastError();
if (cerr != cudaSuccess)
  fprintf(stderr, "%s\n", cudaGetErrorString(cerr));
```

API CUDA: gli “eventi”

- Gli eventi sono una sorta di marcatori che possono essere utilizzati nel codice per:
  - misurare il tempo trascorso (elapsed) durante l’esecuzione di chiamate CUDA (precisione a livello di ciclo di clock)
  - bloccare la CPU fino a quando le chiamate CUDA precedenti l’evento sono state completate (maggiori dettagli su chiamate asincrone in seguito...)

Eventi per misurare il tempo

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
...
kernel<<<grid, block>>>(...);
...
cudaEventRecord(stop);
cudaEventSynchronize(stop);

float elapsed;
// tempo tra i due eventi
// in millisecondi
cudaEventElapsedTime(&elapsed,
 start, stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

```
integer ierr
type (cudaEvent) :: start, stop
real elapsed

ierr = cudaEventCreate(start)
ierr = cudaEventCreate(stop)

ierr = cudaEventRecord(start, 0)
...
call kernel<<<grid,block>>>()
...
ierr = cudaEventRecord(stop, 0)
ierr = cudaEventSynchronize(stop)

ierr = cudaEventElapsedTime&
    (elapsed,start, stop)

ierr = cudaEventDestroy(start)
ierr = cudaEventDestroy(stop)
```

Performance

- Quale metrica utilizziamo per misurare le performance?

- Flops (numero di operazioni floating point per secondo):

$$\text{flops} = \frac{N_{\text{OPERAZIONI FLOATING POINT}}\ (\text{flop})}{\text{Elapsed Time (s)}}$$

- Più comodo di solito usare Mflops, Gflops,... Interessante valutare questa grandezza a confronto con il valore di picco della macchina. Per schede NVIDIA Fermi valori di picco: 1Tflops in singola precisione, 0.5 Tflops in doppia precisione

- Bandwidth (larghezza di banda): quantità di dati trasferiti al secondo

$$\text{bandwidth} = \frac{\text{Quantità di dati trasferiti (byte)}}{\text{Elapsed Time (s)}}$$

- Di solito si usano GB/s o. Nel contesto GPU Computing possibili trasferimenti tra diverse aree di memoria (HostToDevice, DeviceToHost, DeviceToDevice)

Trasferimento dati D2H e H2D

- i device GPU sono connessi all'HOST tramite bus PCIe
  - bassa latenza, ma bassa bandwidth

Technology	Peak Bandwidth
PCIex GEN2 (16x, full duplex)	8 GB/s (peak)
PCIex GEN3 (16x, full duplex)	16 GB/s (peak)
DDR3 (full duplex)	26 GB/s (single channel)

- il trasferimento dati può diventare un collo di bottiglia
  - minimizzare i trasferimenti o eseguirli in overlap con il calcolo (advanced techniques)

Misura della bandwidth: esercitazione

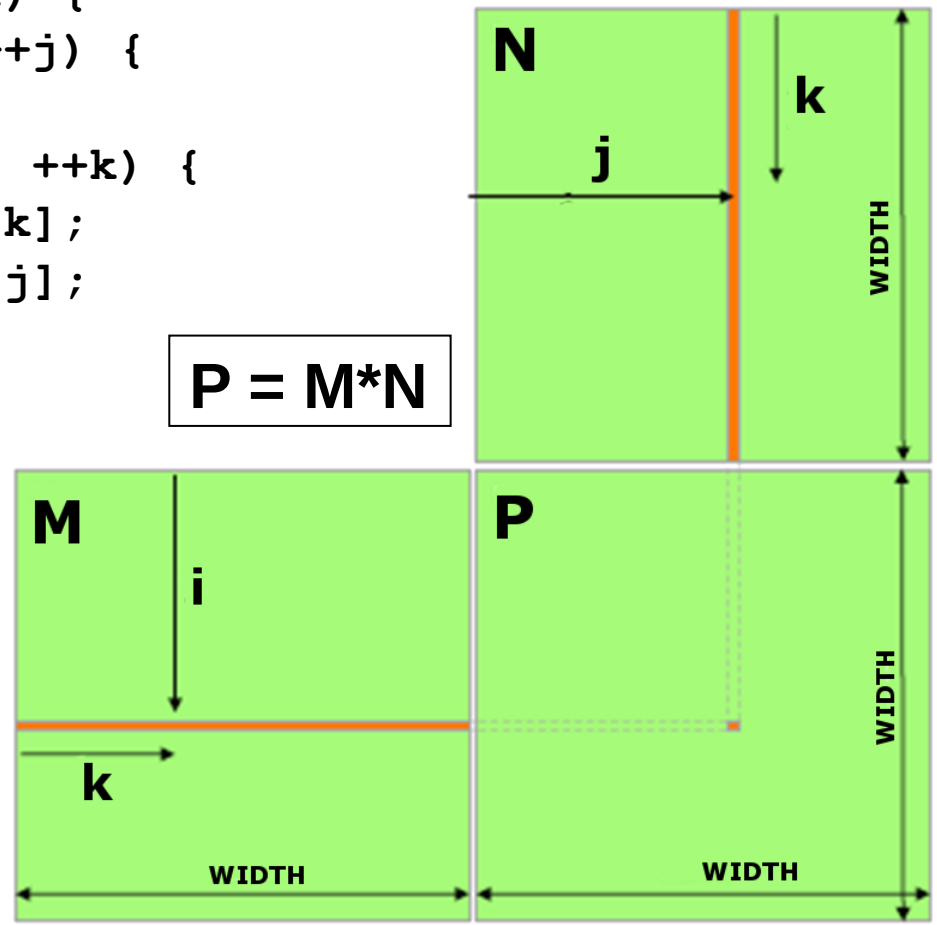
- riportare la bandwidth misurata all'aumentare del pacchetto dati trasferito da Host a Device, da Device to Host, da Device to Device
- utilizzare il tool bandwidthTest del CUDA SDK

```
./bandwidthTest --mode=range --start=<B> --end=<B> --increment=<B>
```

Size (MB)	HtoD	DtoH	DtoD
1			
10			
100			
1024			

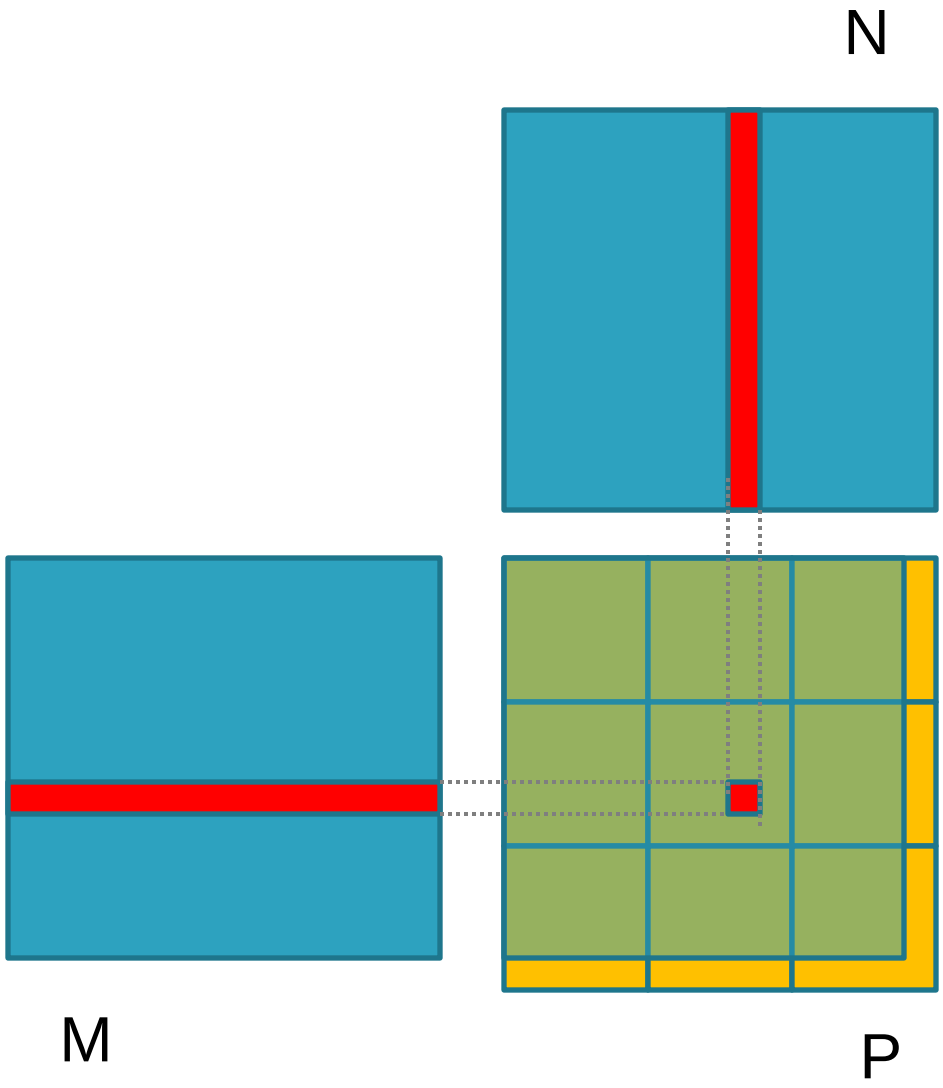
Prodotto Matrice Matrice: HOST Kernel

```
void MatrixMulOnHost(
float* M, float* N, float* P, int Width)
{
  for (int i = 0; i < Width; ++i) {
    for (int j = 0; j < Width; ++j) {
      float pval = 0;
      for (int k = 0; k < Width; ++k) {
        float a = M[i * Width + k];
        float b = N[k * Width + j];
        pval += a * b;
      }
      P[i * Width + j] = pval;
    }
  }
}
```



Prodotto Matrice-Matrice usando memoria globale

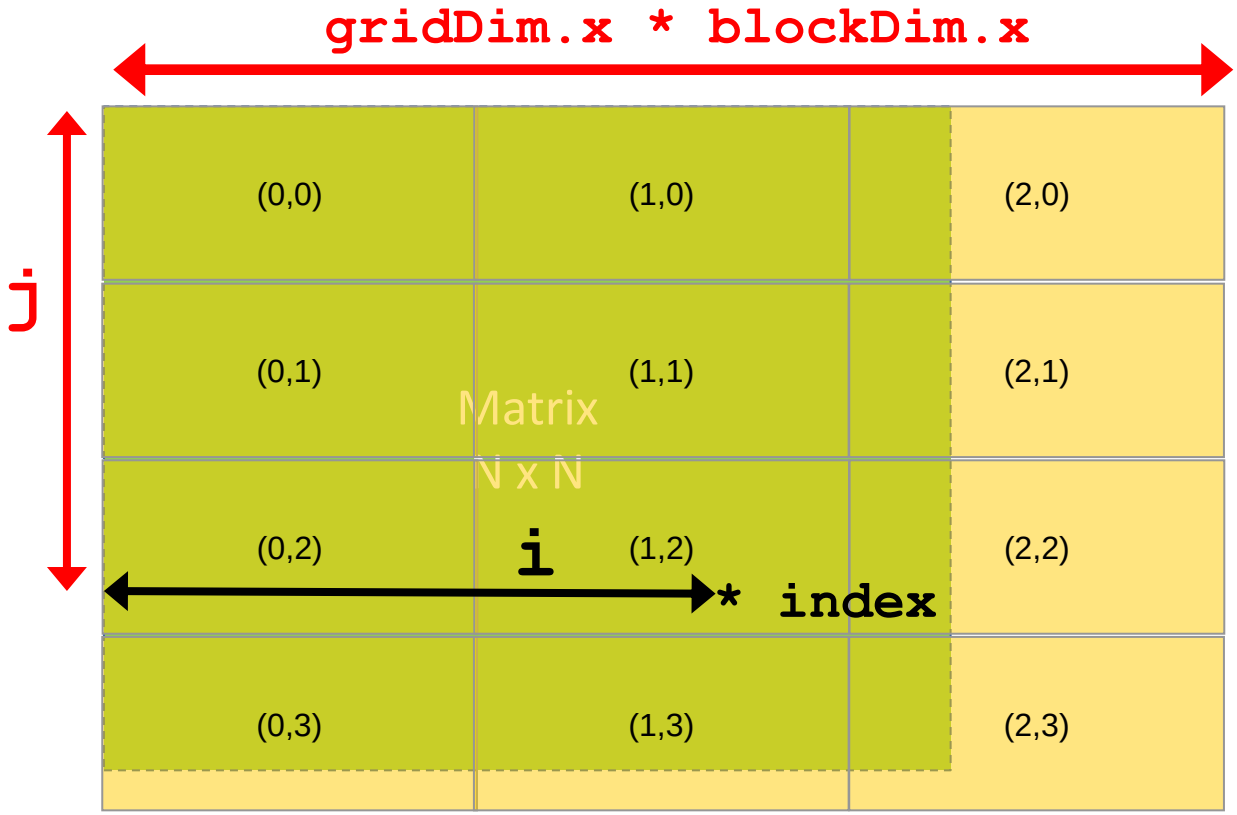
- Ogni thread calcola un elemento della matrice P prodotto
- Dimensionamento griglia di thread deve coprire tutti gli elementi della matrice da elaborare
- controllare quali thread cadono effettivamente all'interno dei limiti della matrice
- Ogni thread legge dalla memoria globale scorrendo gli elementi di una riga di M e di una colonna di N, li moltiplica e accumula il loro prodotto in P



Prodotto Matrice-Matrice in memoria globale: CUDA Kernel

```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int width) {

  // 2D thread ID
  int col = blockIdx.x*blockDim.x + threadIdx.x;
  int row = blockIdx.y*blockDim.y + threadIdx.y;

  // check if current CUDA thread is inside matrix borders
  if (row < width && col < width) {

    // Pvalue stores the Pd element that is computed by current thread
    float Pvalue = 0;
    for (int k=0; k < width; k++)
       Pvalue += Md[row * width + k] * Nd[k * width + col];

    // write the matrix to device memory
    Pd[row * width + col] = Pvalue;
  }
}
```

Prodotto Matrice-Matrice in memoria globale: griglia di lancio




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

index = j * realMatrixWidth + i;
```

Prodotto Matrice-Matrice in memoria globale: codice HOST

```
void MatrixMultiplication(float* M, float *N, float *P, int width) {
  size_t size = width*width*sizeof(float);
  float* Md, Nd, Pd;
  // allocate M, N and P on the device
  cudaMalloc((void**)&Md, size);
  cudaMalloc((void**)&Nd, size);
  cudaMalloc((void**)&Pd, size);
  // transfer M and N to the device memory
  cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
  cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
  // kernel invocation
  dim3 block(BLOCK_DIM, BLOCK_DIM);
  dim3 grid((width-1)/block.x+1, (width-1)/block.x+1);
  MNKernel<<<grid, block>>>(Md, Nd, Pd, width);
  // transfer P from the device to the host
  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
  // free device matrices
  cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

Prodotto Matrice-Matrice in memoria globale: griglia di lancio

- Qual è la dimensione ottimale di un blocco di thread (BLOCK_DIM) nel caso del kernel prodotto Matrice-Matrice ?
  - Esistono dei vincoli strutturali:
    - Un blocco di CUDA thread può contenere al massimo **1024** thread
    - La griglia di blocchi non può contenere più di **65535** blocchi per **Fermi** / 2^{31}**-1** blocchi per **Kepler**
  - BLOCK_DIM deve essere scelto in modo da saturare le risorse a disposizione dello Streaming Multiprocessor (SM)

Prodotto Matrice-Matrice in memoria globale: griglia di lancio

Fermi :

- ogni SM può gestire fino a **1536 thread**
- Numero massimo di **8 blocchi residenti per SM**
- **8x8** = 64 thread : 1536/64 = 24 blocchi per occupare tutto un SM

  Con un massimo di 8 blocchi per SM : 64x8 = **512** thread per SM

  Su un totale di 1536 disponibili
- **16x16** = 256 thread : 1536/256 = 6 blocchi

  256x6 = **1536** thread per SM (Piena occupazione dello SM!)
- **32x32** = 1024 thread : 1536/1024 = 1.5 = 1 blocco.

  1024x1 = **1024** thread per SM

BLOCK_DIM = 16

Prodotto Matrice-Matrice in memoria globale: griglia di lancio

Kepler:

- ogni SM può gestire fino a **2048 thread**
- Numero massimo di **16 blocchi residenti per SM**
- **8x8** = 64 thread : 2048/64 = 32 blocchi per occupare tutto un SM

  Con un massimo di 16 blocchi per SM : 64x16 = **1024** thread per SM

  Su un totale di 2048 disponibili
- **16x16** = 256 thread : 2048/256 = 8 blocchi

  256x8 = **2048** thread per SM (Piena occupazione dello SM!)
- **32x32** = 1024 thread : 2048/1024 = 2 blocco.

  1024x2 = **2048** thread per SM (Piena occupazione dello SM!)

BLOCK_DIM = 16 o 32

Prodotto Matrice-Matrice in memoria globale: errore

- Implementare il prodotto matrice-matrice
- utilizzare le funzioni di controllo dell'errore
  - utilizzare cudaGetLastError() per controllare la corretta esecuzione del kernel

```
mycudaerror=cudaGetLastError() ;
          <chiamata kernel>
cudaThreadSynchronize() ;
mycudaerror=cudaGetLastError() ;
if(mycudaerror != cudaSuccess)
  fprintf(stderr,"%s\n",
  cudaGetErrorString(mycudaerror)) ;
```

```
mycudaerror=cudaGetLastError()
          <chiamata kernel>
ierr = cudaThreadSynchronize()
mycudaerror=cudaGetLastError()
if(mycudaerror .ne. 0) write(*,*)  &
  'Errore in kernel: ',mycudaerror
```

- Provare a lanciare un kernel con blocchi di thread più grandi del limite (32x32). Vedere quale tipo di errore si rileva.

Prodotto Matrice-Matrice in memoria globale: performance

- Implementare la misura del tempo del kernel tramite eventi e funzioni di timing Cuda
- Fasi:
  - Creazione eventi start e stop: *cudaEventCreate*
  - Record dell'evento start: *cudaEventRecord*
  - Chiamata kernel, eventualmente con controllo dell'errore
  - Record dell'evento stop: *cudaEventRecord*
  - Sincronizzazzione eventi: *cudaEventSynchronize*
  - Misura elapsed time tra eventi: *cudaEventElapsedTime*
  - Distruzione eventi: *cudaEventDestroy*
- Calcolare i Gflops (floating point operations per second) considerando che l'algoritmo richiede $2N^3$ operazioni

	C	Fortran
Gflops		