Introduction to Scientific Programming using GPGPU and CUDA

CUDA Hands on

Luca Ferraro
l.ferraro@cineca.it

Sergio Orlandini
s.orlandini@cineca.it

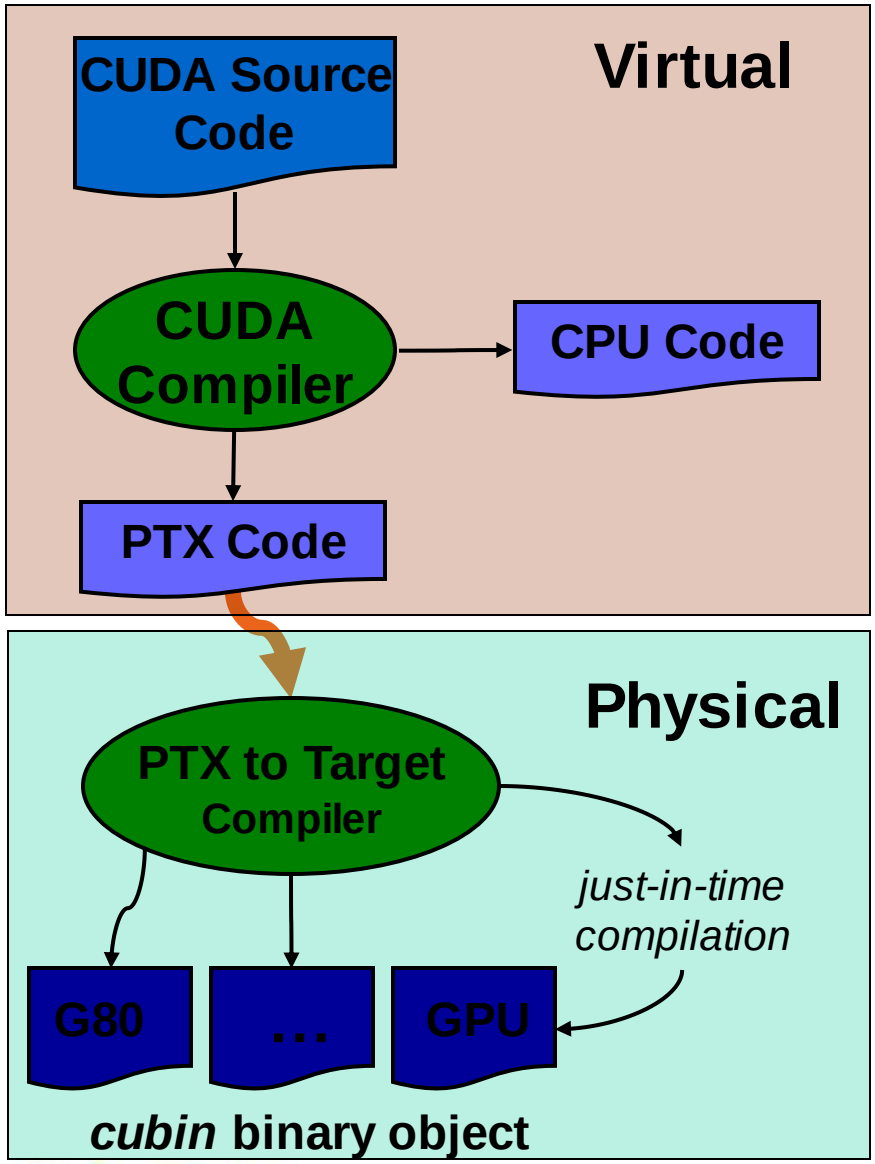
Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini

- Hands on:
  - Compiling a CUDA program
  - Environment and utility: deviceQuery and nvidia-smi
  - Vector Sum
  - Matrix Sum
  - Matrix Product

CUDA Compilation Workflow



- each source file with CUDA extension should be compiler with a proper CUDA aware compiler
  - `nvcc` CUDA C (NVIDIA)
  - `pgf90 -Mcuda` CUDA Fortran (PGI)
- CUDA compiler processes the source code, separating device code from host code:
  - *host* is modified replacing CUDA extensions by the necessary CUDA C runtime functions calls
  - the resulting *host* code is output to a host compiler
  - *device* code is compiled into the PTX assembly form
- starting from the PTX assembly code you can:
  - generate one or more object forms (*cubin*) specialized for specific GPU architectures
  - generate an executable which include both PTX and object code

Compute Capability

- the *compute capability* of a device describes its architecture
  - *registers, memory sizes, features and capabilities*
- the compute capability is identified by a code like "`compute_Xy`"
  - major number (X): identifies base line chipset architecture
  - minor number (y): indentifies variants and releases of the base line chipset
- a compute capability select the set of usable PTX instructions

compute capability	*feature support*
compute_20	FERMI architecture
compute_30	KEPLER K10 architecture (only single precision)
compute_35	KEPLER K20, K20X, K40 architectures
compute_37	KEPLER K80 architecture (two K40 on a single board)
compute_53	MAXWELL GM200 architecture (only single precision)
compute_60	PASCAL GP100 architecture

How to compile a CUDA program

- When compiling a CUDA executable, you must specify:
  - the compute capability: the virtual architecture for *PTX code*
  - actual architecture targets: the real GPU architectures where the executable will run (using the cubin code)

```
nvcc -arch=compute_35 -code=sm_35,sm_37
```

virtual architecture (*PTX code*)

real GPU architecture (*cubin*)

  - nvcc allows many shortcut switches as

`nvcc -arch=sm_35` to target Kepler K20/K40 architectures

which is equivalent to:

```
nvcc -arch=compute_35 -code=sm_35
```

Hands On

- `deviceQuery` (from the CUDA SDK): show information on CUDA devices
- `nvidia-smi` (NVIDIA System Management Interface): shows diagnostic information on present CUDA enabled devices (`nvidia-smi -q -d UTILIZATION -l 1`)
- `nvcc -V` shows current CUDA C compiler version
- Compile a CUDA program:
  - cd Exercises/VectorAdd. Try the following compiling commands:
  - `nvcc vectoradd_cuda.cu -o vectoradd_cuda`
  - `nvcc -arch=sm_35 vectoradd_cuda.cu -o vectoradd_cuda`
  - `nvcc -arch=sm_35 -ptx vectoradd_cuda.cu`
  - `nvcc -arch=sm_35 -keep vectoradd_cuda.cu -o vectoradd_cuda`
  - `nvcc -arch=sm_35 -keep -clean vectoradd_cuda.cu -o vectoradd_cuda`
  - Run resulting executable with:
  - `./vectoradd_cuda`

Hands On

- MatrixAdd:
  - write a program that performes square matrix sum: C = A + B
  - provide and compare results of the CPU and CUDA version of the kernel
  - try with different thread block sizes (16,16) (32,32) (64,64)
- variants
  - modify the previous kernel so to let inplace sum, that is: A= A + c*B

- Control and performances:
  - Error Handling
  - Measuring Performances

- Hands on:
  - measure data transfer performances
  - Matrix-Matrix product
    - simple implementation
    - performances

Checking Errors

- All CUDA API returns an error code of type `cudaError_t`
  - the special value `cudaSuccess` means that no error occurred
  - CUDA kernels are void type so they don't return any error code
  - CUDA kernels are asynchronous, returning an error which refers only on errors which may occur during the call on *host*

`char* cudaGetErrorString(cudaError_t code)`

  - returns a string (NULL-terminated) with a human understandable description of the type of error occured

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

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

Checking Errors

- The error status is also held in an internal variable, which is modified by each CUDA API call or kernel launch

`cudaError_t cudaGetLastError(void)`

  - returns the code status of internal error variable (cudaSuccess or other)
  - resets the internal error status to `cudaSuccess`
    - so the error code returned from a cudaGetLastError may refers to any other CUDA API preceeding current call to cudaGetLastError
  - ito check the error status of a CUDA kernel execution, we must instruct the host code to wait for kernel completition using one of the following synchronization API: **`cudaThreadSynchronize()or cudaDeviceSynchronize()`**

```
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));
```

Checking Errors

```
#define CUDA_CHECK(X) {\
 cudaError_t _m_cudaStat = X;\
 if(cudaSuccess != _m_cudaStat) {\
    fprintf(stderr,"\nCUDA_ERROR: %s in file %s line %d\n",\
    cudaGetErrorString(_m_cudaStat), __FILE__, __LINE__);\
    exit(1);\
 } }

...
CUDA_CHECK( cudaMemcpy(d_buf, h_buf, buffSize, cudaMemcpyHostToDevice) );
```

- Error checking is strongly encouraged during developer phase
  - yet, error checking may introduce overhead during production run
  - and error check code can become very verbose and boring
- A common approach is to define a preprocessor macro, as in the above example, which can be turned on/off in a very simple manner

CUDA Events

- CUDA Events are special objects which can be used as mark points in your code
- CUDA events markers can be used to:
  - measure the elapsed time between two markers (providing very high precision measures)
  - indentify synchronization point in the code between CPU and GPU execution flow:
    - for example we can prevent CPU to go any further until some or all preceeding CUDA kernels are really completed
    - we will provide further information on synchronization techniques during the rest of the course

Using CUDA Events for Measuring Elapsed Time

```
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)
```

Performances

- Which metric should we use to measure performances?

- Flops (floating point operations per second):

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

- A common metric for measuring performances of a computational intensive code is reporting:
  Mflops, Gflops,...

- Bandwidth (amount of data transfered per second)

$$bandwidth = \frac{Size\ of\ transfere\ d\ data\ (byte)}{Elapsed\ Time\ (s)}$$

- A common metric is GB/s
  Reference value depends on peak bandwidth performances provided by the bus or network hardware involved in the data transfer

D2H and H2D Data Transfers

- GPU devices are connected to the host with a PCIe bus
  - PCIe bus is characterized by very low latency, but also by a low bandwidth with respect to other bus

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)

- data transfer can easily become a bottleneck in heterogeneous environment equipped with accelerators
  - strive to minimize transfers or execute them in overlap with computations (advanced technique, more on this later)

Hands on: measuring bandwidth

- Measure memory bandwidth versus increasing data size, for Host to Device, Device to Host and Device to Device transfers
- you can write a simple program for that, using CUDA events
- or rely on the `bandwidthTest` provied with the CUDA SDK

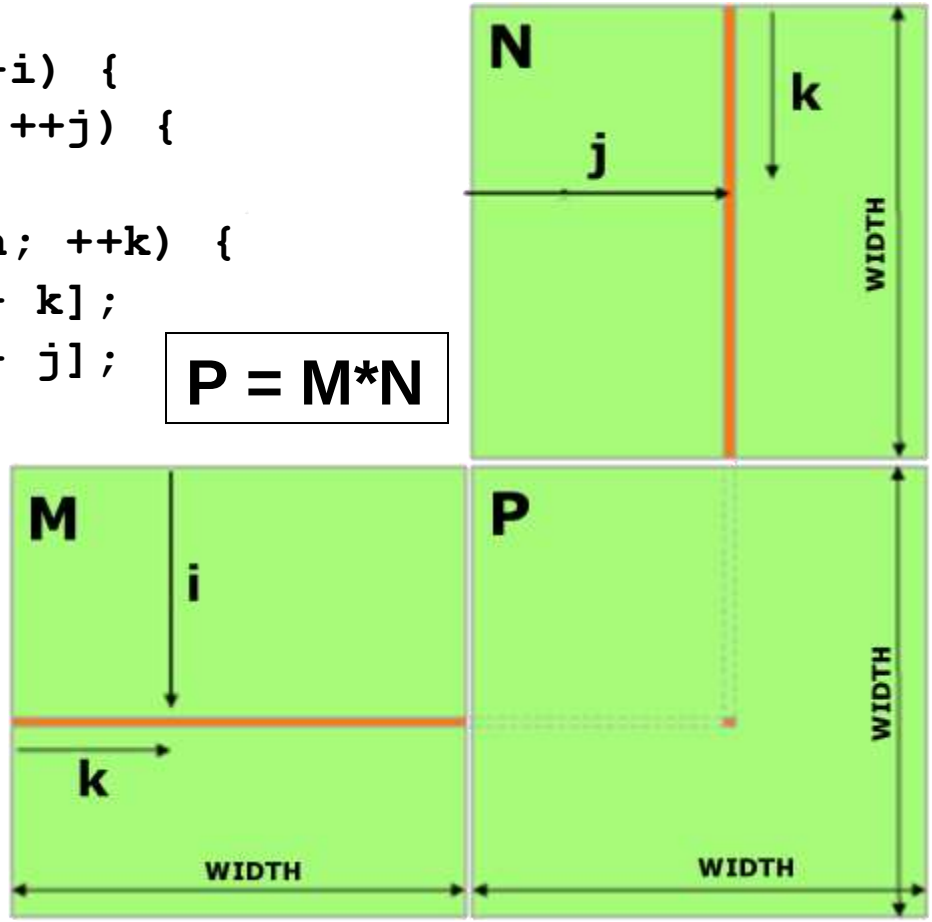
```
./bandwidthTest --mode=range --start=<B> --end=<B> --increment=<B>
```

Size (MB)	HtoD	DtoH	DtoD
1			
10			
100			
1024			

Matrix-Matrix product: 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;
    }
  }
}
```

P = M*N



Matrix-Matrix product: CUDA Kernel

```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int width) {
  // 2D thread ID
  int col = threadIdx.x;
  int row = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the 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
  // (each thread writes one element)
  Pd[row * width + col] = Pvalue;
}
```

Matrix-matrix product using global memory: HOST code

```
void MatrixMultiplication(float* M, float *N, float *P, int width) {
  size_t size = width*width*sizeof(float);
  float* Md, Nd, Pd;

  cudaMalloc((void**)&Md, size);
  cudaMalloc((void**)&Nd, size);
  cudaMalloc((void**)&Pd, size);

  cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
  cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

  dim3 gridDim(1,1);
  dim3 blockDim(width,width);
  MNKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, width);

  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

  cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

Matrix-matrix using global memory: launch grid

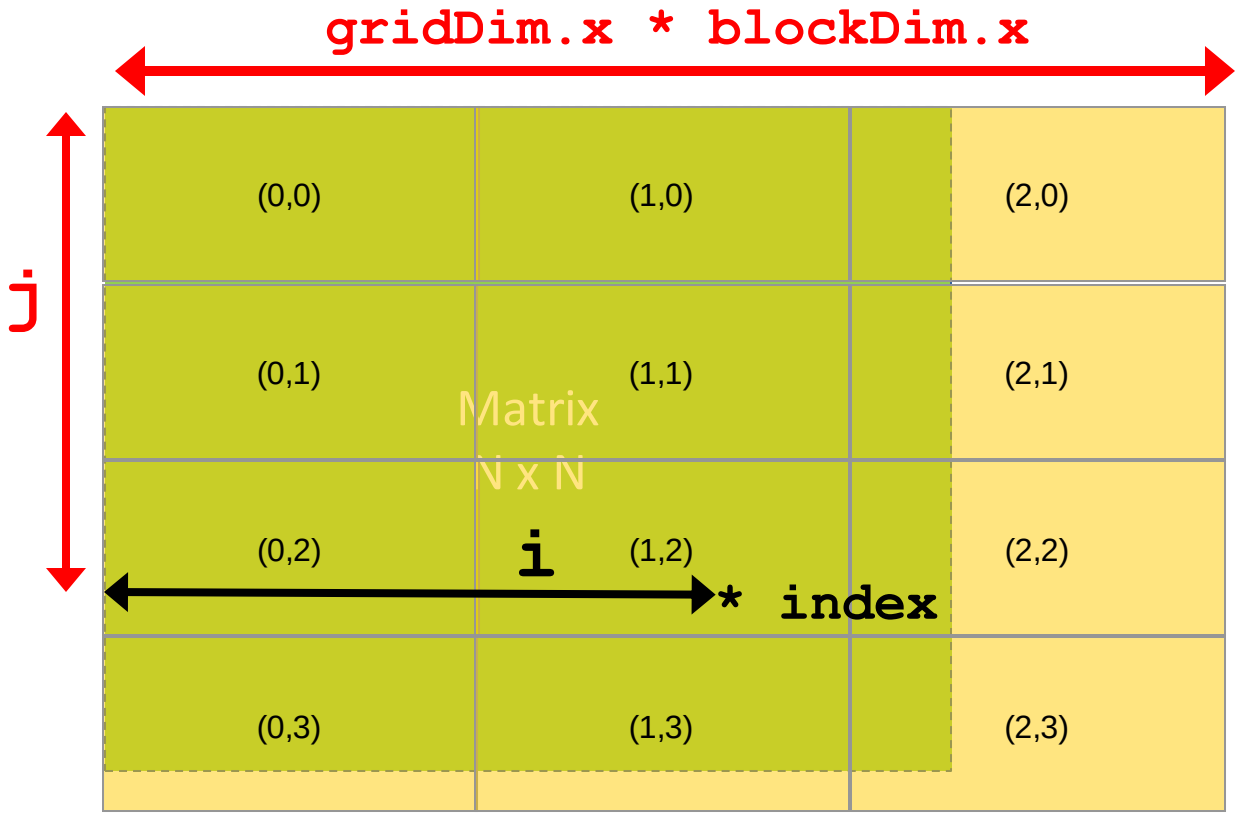
WARNING:

- there's a limit on the maximum number of allowed threads per block
  - depends on the compute capability

How to select an appropriate (or best) thread grid ?

- respect compute capability limits for threads per block
- select the block grid so to cover all elements to be processed
- select block size so that each thread can process one or more data elements without raise conditions with other threads
  - use *builtin* variables *blockIdx* and *blockDim* to identify which matrix subblock belong to current thread block

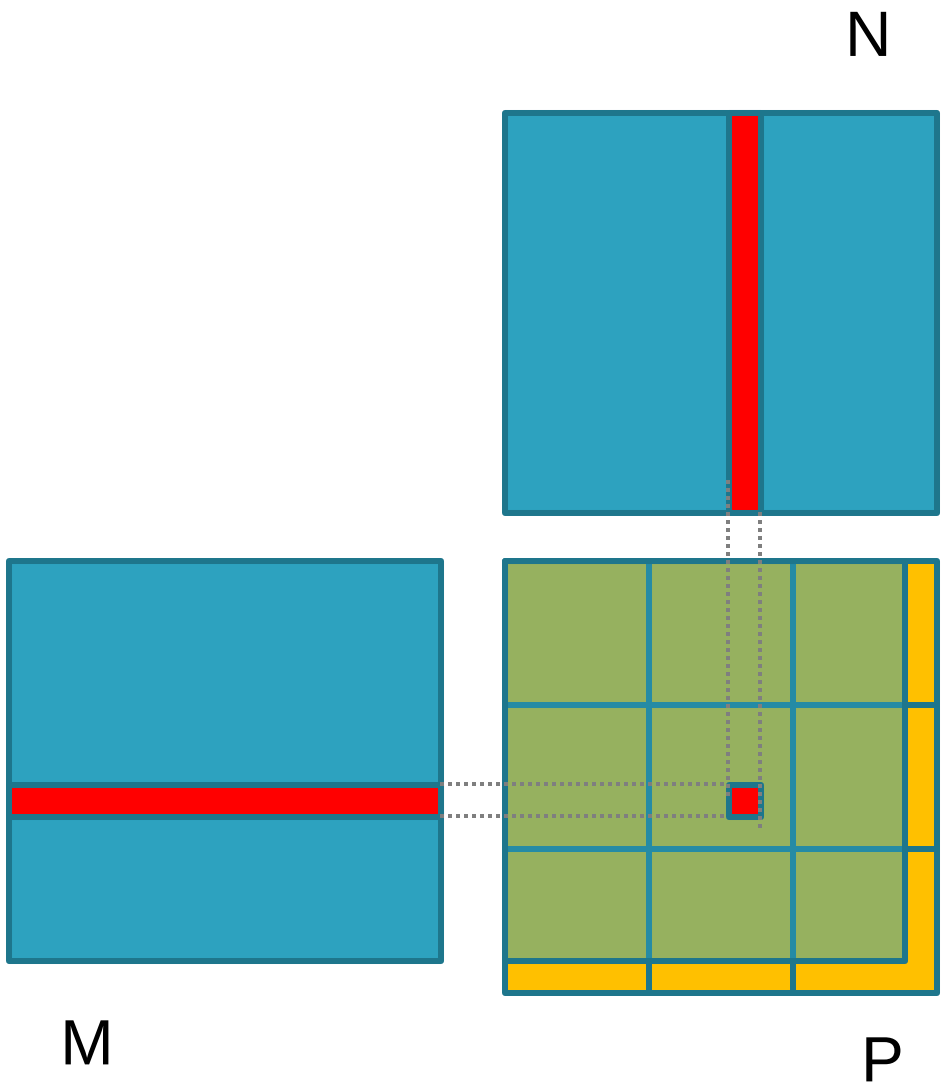
Matrix-matrix using global memory: launch grid




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

index = j * realMatrixWidth + i;
```

Matrix-matrix product using global memory

- Let each thread compute only one matrix element of resulting P matrix
- choose a block grid large enough to cover all elements to be computed
  - check if some thread is accessing elements outside of the domain
- Let heac thread read one element from global memory, cycling through the elements in a row of matrix M and elements in the a column of matrix N
- multiply and accumulate each signle element product into a scalar variable, and write the final result into corret location of matrix P



Matrix-matrix product: CUDA kernel

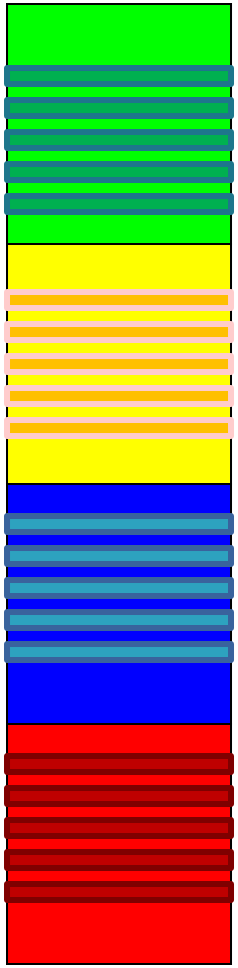
```
__global__ void MNKernel(float* Md, float *Nd, float *Pd, int width) {
    // 2D thread ID
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    int row = blockIdx.y*blockDim.y + threadIdx.y;

  // 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
     // (each thread writes one element)
     Pd[row * width + col] = Pvalue;
  }
}
```

kernel execution configuration:

```
dim3 gridDim( (width+TILE_WIDTH-1)/TILE_WIDTH, (width+TILE_WIDTH-1)/TILE_WIDTH );
dim3 blockDim( TILE_WIDTH, TILE_WIDTH );
MNKernel <<< dimGrid, dimBlock >>> (Md, Nd, Pd, width);
```

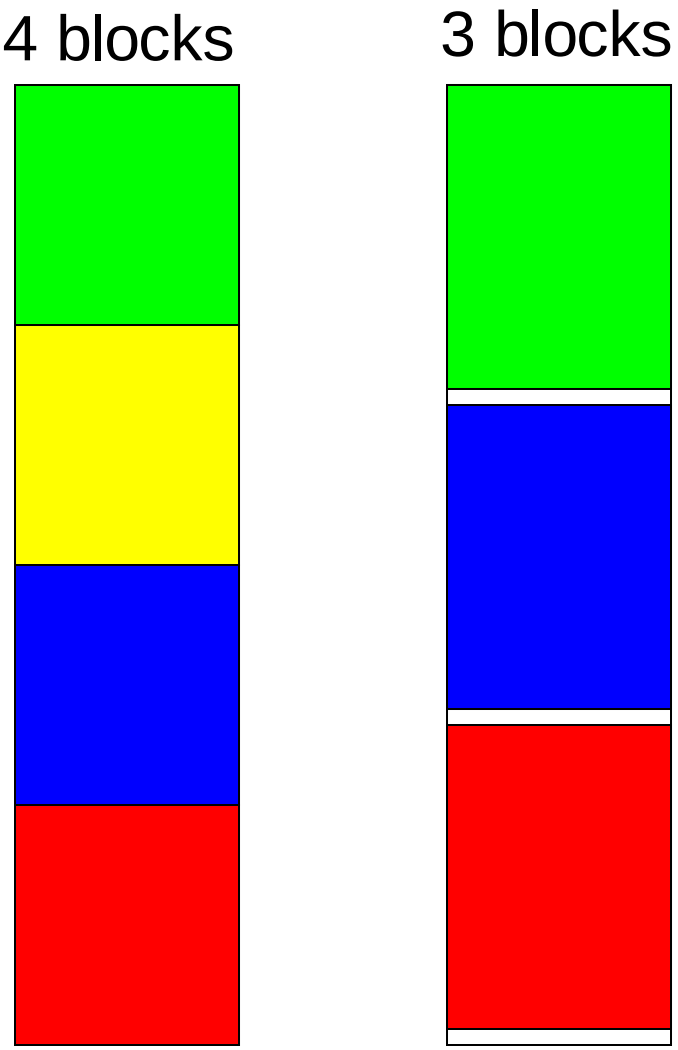
Resources per Thread Block

- each CUDA kernel needs a specific amount of resources to run
- Once blocks are assigned to the SM, registers are assigned to each thread block, depending on kernel required resources
- Once assigned, registers will belong to that thread until the thread block complete its work
- so that each thread can access only its own assigned registers
- allow for zero-overload schedule when content switching among different warp execution



Assigning Thread Blocks to SM

- Let's provide an example of block assignmend on a SM:
  - Fermi architecture: 32768 register per SM
  - CUDA kernel grid with 32x8 thread blocks
  - CUDA kernel needs 30 registers
- How many thread blocks can host a single SM?
  - each block requires 30x32x8 = 7680 registers
  - 32768/7680 = **4** blocks + "reminder"
  - only 4 blocks can be hosted (out of 8)
- What happen if we modify the kernel a little bit, moving to an implementation which requires 33 registers?
  - each block now requires 33x32x8 = 8448 registers
  - 32768/8448 = **3** blocks + "reminder"
  - only 3 blocks! (out of 8)
    - 25% reduction of potential parallelism



Matrix-matrix product: selecting thread block size

Which is the best thread block size to select (i.e. TILE_WIDTH)?

On Fermi architectures: each SM can handle up to 1536 total threads

- 8x8 = 64 threads ⇨ 1536/64 = 24 blocks needed to fully load a SM;
  ... yet there is a limit of maximum 8 blocks per SM on cc 2.0
  so we end up with just 64x8 = 512 threads per SM (33% occupancy)
- 16x16 = 256 threads ⇨ 1536/256 = 6 blocks

  ... reaching full occupancy per SM
- 32x32 = 1024 threads ⇨ 1536/1024 = 1.5 ⇨ 1 block

⇨ TILE_WIDTH = 16

Matrix-matrix product: selecting thread block size

Which is the best thread block size to select (i.e. TILE_WIDTH)?

On Kepler architectures: each SM can handle up to 2048 total threads

- 8x8 = 64 threads ⇨ 2048/64 = 32 blocks needed to fully load a SM;
  ... yet there is a limit of maximum 16 blocks per SM on cc 3.0
  so we end up with just 64x16 = 1024 threads per SM (50% occupancy)
- 16x16 = 256 threads ⇨ 2048/256 = 8 blocks

  ... reaching full occupancy per SM
- 32x32 = 1024 threads ⇨ 2048/1024 = 2 blocks

➡ TILE_WIDTH = 32 or 16

Matrix-matrix product: checking error

- Now, hands on and implement the matrix-matrix product
- Use the proper CUDA API to check error codes
  - use cudaGetLastError() to check that kernel has been completed with no errors

```
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
```

- Try to use block size greater than allower limit of 32x32.
- What kind of error is reported?

Matrix-matrix product: performances

- Measure performances of your matrix-matrix product implementation, both for CPU and GPU version, using CUDA Events
- Follow these steps:
  - Decleare a start and stop cuda event and initialize them with: *cudaEventCreate*
  - Plase start and stop events at proper place in the code
  - Record the start event using: *cudaEventRecord*
  - Launch the CPU or GPU (remember to check for errors)
  - Record the stop event using: *cudaEventRecord*
  - Synchronize host code just after the stop event with: *cudaEventSynchronize*
  - Measure the elapsed time between the two events with: *cudaEventElapsedTime*
  - Distruzione eventi: cudaEventDestroy
- Express performance metric using Gflops, knowing that the matrix-matrix product algorithm requres 2N^3 operations

	C	Fortran
Gflops		