Programming paradigms for GPU devices

PTC course

@CINECA

15-17 April 2020

Sergio Orlandini
s.orlandini@cineca.it

Luca Ferraro
l.ferraro@cineca.it

- Control and performances:
  - Error Handling
  - Measuring Performances

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

Checking CUDA Errors

- All CUDA API returns an error code of type **`cudaError_t`**
  - Special value **`cudaSuccess`** means that no error occurred
- CUDA runtime has a convenience function that translates a CUDA error into a readable string with a human understandable description of the type of error occured

```
char* cudaGetErrorString(cudaError_t code)
```

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

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

- CUDA Asynchronous API returns an error which refers only on errors which may occur during the call on *host*
- CUDA kernels are asynchronous and void type so they don't return any error code

Checking Errors for CUDA kernels

- The error status is also held in an internal variable, which is modified by each CUDA API call or kernel launch.
- CUDA runtime has a function that returns the status of internal error variable.

`cudaError_t cudaGetLastError(void)`

1. Returns the status of internal error variable (**`cudaSuccess`** or other)
2. Resets the internal error status to **`cudaSuccess`**

- Error code from **`cudaGetLastError`** may refers to any other preceeding CUDA API runtime calls
- To check the error status of a CUDA kernel execution, we have to wait for kernel completition using the following synchronization API: **`cudaDeviceSynchronize()`**

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

Checking CUDA Errors

- Error checking is strongly encouraged during developer phase
- Error checking may introduce overhead and unpleasant synchronizations during production run
- Error check code can become very verbose and tedious

  A common approach is to define a assert style preprocessor macro which can be turned on/off in a simple manner

```
#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) );
```

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)
  - identify 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 preceding CUDA kernels are really completed
    - we will provide further information on synchronization techniques during the rest of the course

CUDA Events for Measuring Elapsed Time

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

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

float elapsed;
// execution time between events
// in milliseconds
cudaEventElapsedTime(&elapsed,
 start, stop);

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

```fortran
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

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

- A common metric for measuring performances of a computational intensive kernel (***compute-buond*** kernel)
- Common units are: Mflops, Gflops, …

Bandwidth:

Amount of data transfered per second

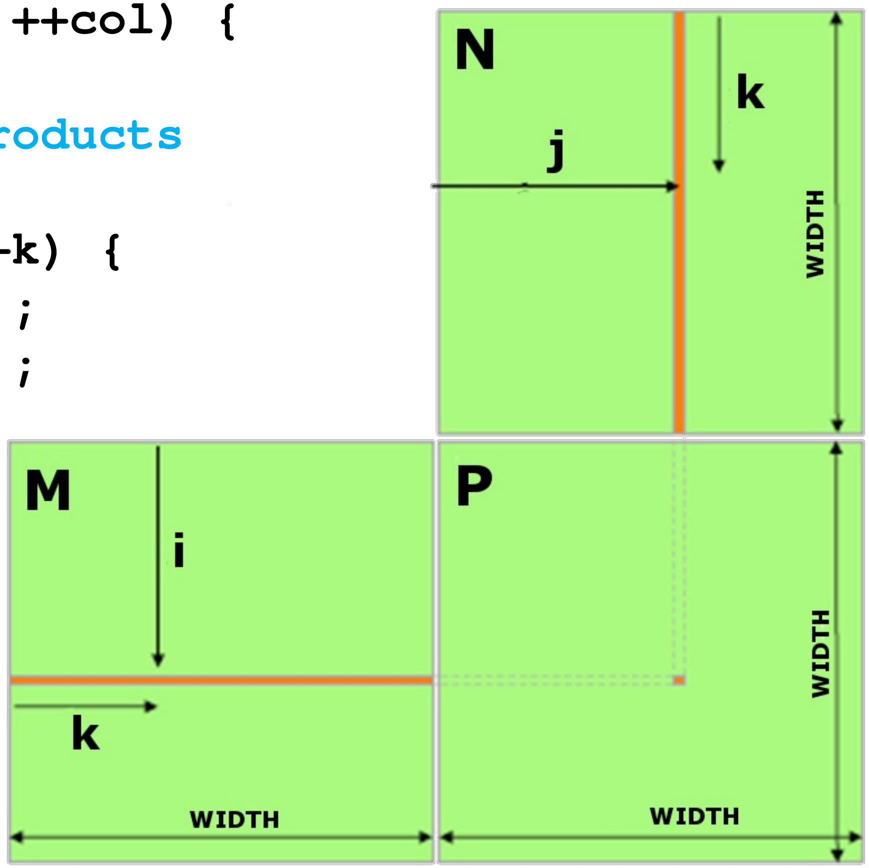
$$\text{bandwidth} = \frac{\text{Size of transfered data (byte)}}{\text{Elapsed Time (s)}}$$

- A common metric for kernel that spent the most of time in executing memory instructions (***memory-bound*** kernel).
- Common unit of performance is GB/s. Reference value depends on peak bandwidth performances provided by the bus or network hardware involved in the data transfer

Matrix-Matrix product: HOST Kernel

```
void MatrixMulOnHost (float* M, float* N, float* P, int Width)
{
  // loop on rows
  for (int row = 0; row < Width; ++row) {
    // loop on columns
    for (int col = 0; col < Width; ++col) {

      // accumulate element-wise products
      float pval = 0;
      for (int k = 0; k < Width; ++k) {
        float a = M[row * Width + k];
        float b = N[k * Width + col];
        pval += a * b;
      }

      // store final results
      P[row * Width + col] = pval;
    }
  }
}
```

P = M * N



Matrix-Matrix product: CUDA Kernel

```
__global__ void MMKernel (float* dM, float *dN, float *dP,
                          int width)
{
  // row,col from built-in thread indeces (2D block of threads)
  int col = threadIdx.x;
  int row = threadIdx.y;

  // accumulate element-wise products
  // NB: pval stores the dP element computed by the thread
  float pval = 0;
  for (int k=0; k < width; k++) {
     float a = dM[row * width + k];
     float b = dN[k * width + col];
     pval += a * b;
  }

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

SuperComputing Applications and Innovation

Matrix-Matrix product: HOST code

```
void MatrixMultiplication (float* hM, float *hN, float *hP,
                           int width) {
  float *dM, *dN, *dP;
  cudaMalloc((void**)&dM, width*width*sizeof(float));
  cudaMalloc((void**)&dN, width*width*sizeof(float));
  cudaMalloc((void**)&dP, width*width*sizeof(float));

  cudaMemcpy(dM, hM, size, cudaMemcpyHostToDevice);
  cudaMemcpy(dN, hN, size, cudaMemcpyHostToDevice);

  dim3 gridDim(1,1);
  dim3 blockDim(width,width);

  MMKernel<<<gridDim, blockDim>>>(dM, dN, dP, width);

  cudaMemcpy(hP, dP, size, cudaMemcpyDeviceToHost);

  cudaFree(dM); cudaFree(dN); cudaFree(dP);
}
```

Matrix-Matrix product: launch grid

WARNING:

- There is a limit on the maximum number of allowed threads per block
  - It depends on the device architecture (on the **compute capability**)
  - On the newest chipset the maximum number of threads per block is **1024**!
    - In our case we are not able to perform matrix multiplication between matrix with more than 1024 elements
  - Using a single block to cover all the matrix is not a good choice

Compute Capability

- ***compute capability*** of a device describes its architecture
  - *registers, memory sizes, features and capabilities*
- *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

compute capability	*feature support*
compute_10	basic CUDA support
compute_20	FERMI architecture
compute_30	KEPLER K10 architecture
compute_35	KEPLER K20, K20X, K40 architectures
compute_37	KEPLER K80 architecture
compute_60	PASCAL P100 architecture
compute_70	VOLTA V100 architecture

Capability: resources constraints

	Compute Capability							
Technical Specifications	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**	**5.0**	**5.2**
Maximum dimensionality of grid of thread blocks	2			3				
Maximum x-dimension of a grid of thread blocks	65535				2^{31}-1			
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512			1024				
Maximum z-dimension of a block	64							
Maximum number of threads per block	512			1024				
Warp size	32							
Maximum number of resident blocks per multiprocessor	8				16		32	
Maximum number of resident warps per multiprocessor	24	32		48	64			
Maximum number of resident threads per multiprocessor	768	1024		1536	2048			
Number of 32-bit registers per multiprocessor	8 K	16 K		32 K	64 K			
Maximum number of 32-bit registers per thread	128			63		255		
Maximum amount of shared memory per multiprocessor	16 KB			48 KB			64 KB	96 KB
Maximum amount of shared memory per thread block	16 KB			48 KB				
Number of shared memory banks	16			32				
Amount of local memory per thread	16 KB			512 KB				
Constant memory size	64 KB							

Capability: resources constraints

Technical Specifications	Compute Capability											
	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128	
Maximum dimensionality of grid of thread blocks	3											
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$											
Maximum y- or z-dimension of a grid of thread blocks	65535											
Maximum dimensionality of thread block	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block [27]	48 KB										96 KB	64 KB
Number of shared memory banks	32											
Amount of local memory per thread	512 KB											
Constant memory size	64 KB											
Cache working set per multiprocessor for constant memory	8 KB							4 KB	8 KB			
Cache working set per multiprocessor for texture memory	Between 12 KB and 48 KB							Between 24 KB and 48 KB			32 - 128 KB	32 or 64 KB
Maximum width for a 1D texture reference bound to a CUDA array	65536											
Maximum width for a 1D texture reference bound to linear memory	2^{27}											
Maximum width and number of layers for a 1D layered texture reference	16384 x 2048											

https://docs.nvidia.com/cuda/cuda-c-programming-guide/#features-and-technical-specifications

CINECA SCAI SuperComputing Applications and Innovation

Matrix-Matrix product: launch grid

<u>WARNING</u>:

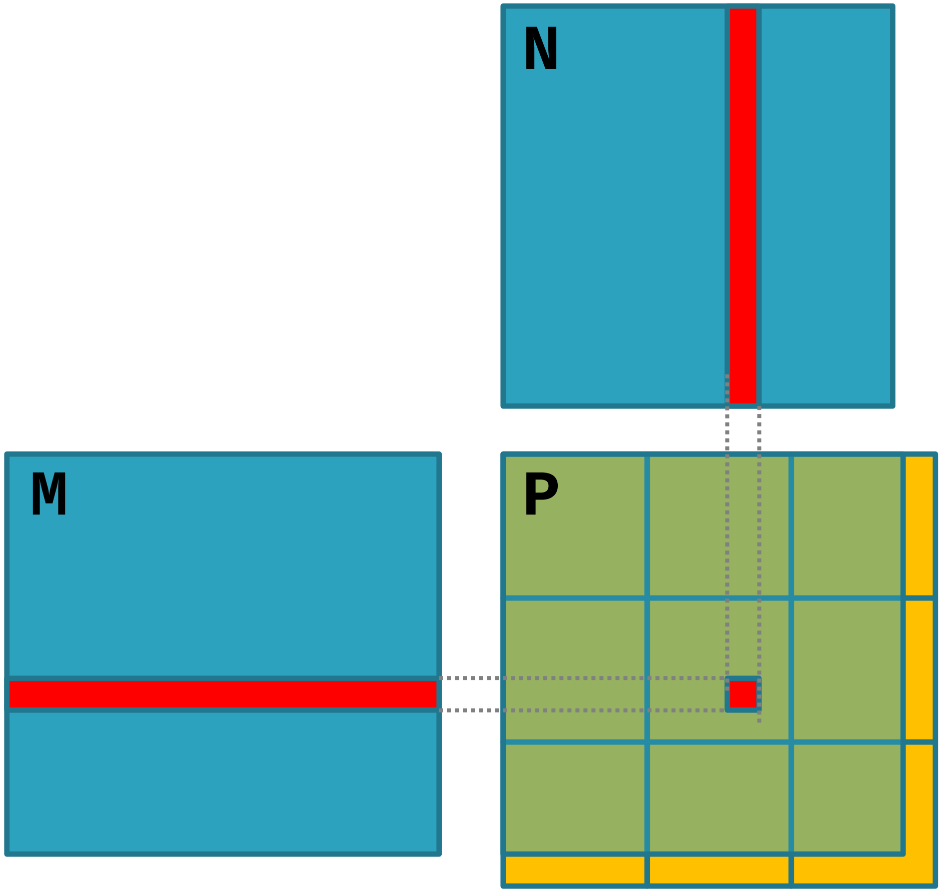
- There is a limit on the maximum number of allowed threads per block
  - It depends on the device architecture (on the **compute capability**)
  - On the newest chipset the maximum number of threads per block is **1024**!
    - In our case we are not able to perform matrix multiplication between matrix with more than 1024 elements
  - Using a single block to cover all the matrix is not a good choice

<u>How to select an appropriate (or best) thread grid?</u>

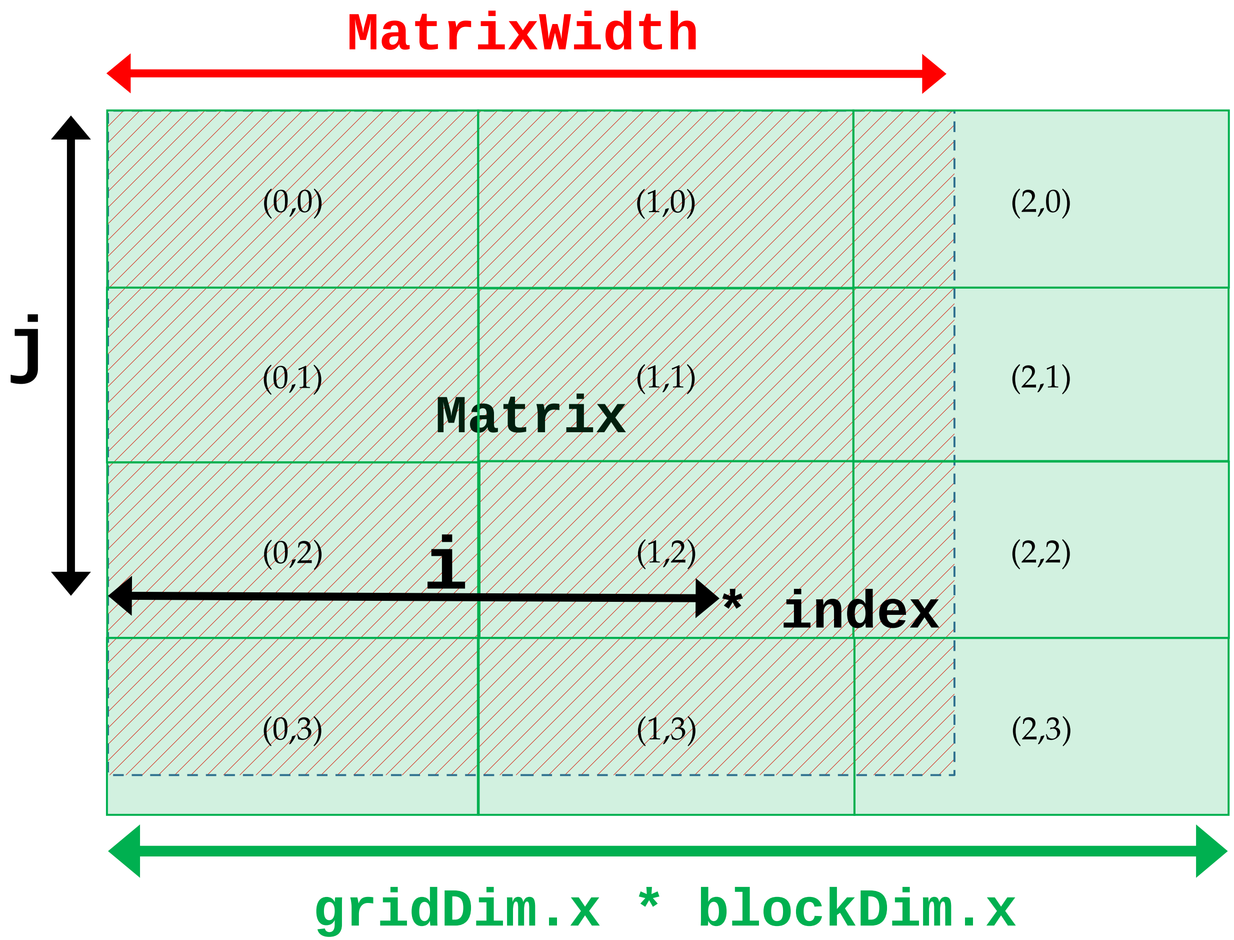
- respect compute capability limits for threads per block
- select the block grid so to cover all elements to be processed
- select block size so that each thread can process one or more data elements without raise conditions with other threads
  - use *builtin* variables `blockIdx` and `blockDim` to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

Matrix-Matrix product: launch grid

- 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 each 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 single element product into a scalar variable, and write the final result into correct location of matrix P



Matrix-Matrix product: launch grid




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

index = j * MatrixWidth + i;
```

Matrix-Matrix product: CUDA Kernel

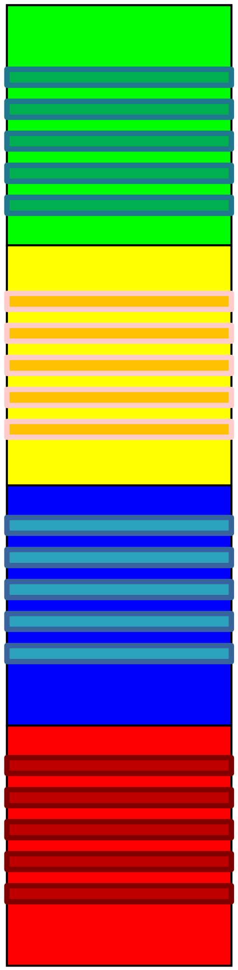
```
__global__ void MMKernel (float* dM, float *dN, float *dP,
                          int width) {
  // row,col from built-in thread indeces(2D block of threads)
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;

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

    // accumulate element-wise products
    // NB: pval stores the dP element computed by the thread
    float pval = 0;
    for (int k=0; k < width; k++)
       pval += dM[row * width + k] * dN[k * width + col];

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

Matrix-Matrix product: HOST code

```
void MatrixMultiplication (float* hM, float *hN, float *hP,
                                        int width) {
 float *dM, *dN, *dP;
 cudaMalloc((void**)&dM, width*width*sizeof(float));
 cudaMalloc((void**)&dN, width*width*sizeof(float));
 cudaMalloc((void**)&dP, width*width*sizeof(float));

 cudaMemcpy(dM, hM, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dN, hN, size, cudaMemcpyHostToDevice);

 dim3 blockDim( TILE_WIDTH, TILE_WIDTH );
 dim3 gridDim( (width-1)/TILE_WIDTH+1,(width-1)/TILE_WIDTH+1 );

 MMKernel<<<gridDim, blockDim>>>(dM, dN, dP, width);

 cudaMemcpy(hP, dP, size, cudaMemcpyDeviceToHost);

 cudaFree(dM); cudaFree(dN); cudaFree(dP);
}
```

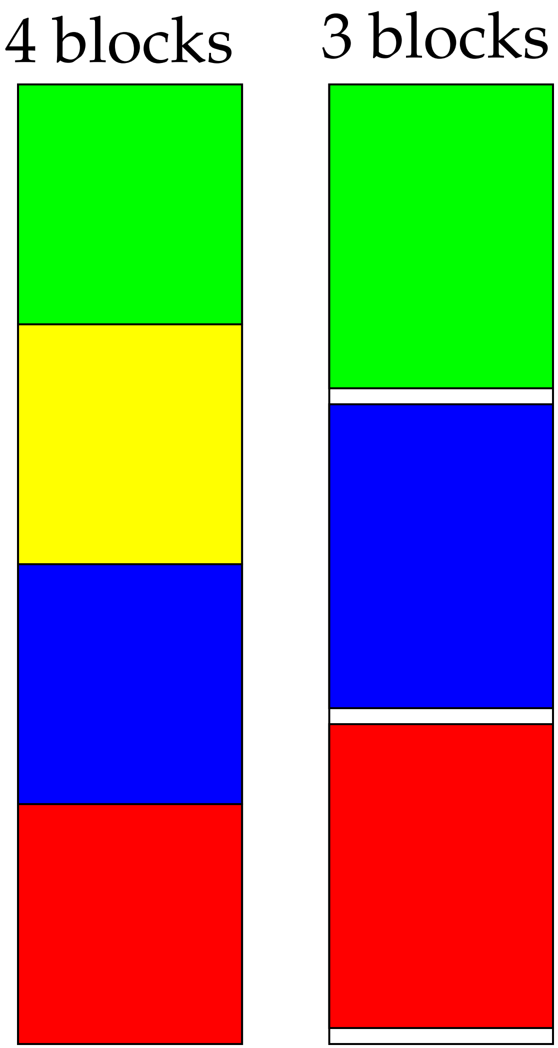
Resources per Thread Block

- 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: thread block size

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

On <u>Fermi</u> architectures: each SM can handle up to ***1536*** total threads

- **TILE_WIDTH** = 8

  8x8 = 64 threads >>> 1536/64 = 24 blocks needed to fully load a SM

  … yet there is a limit of maximum 8 resident blocks per SM for cc 2.x
  so we end up with just 64x8 = 512 threads per SM on a maximum of 1536
  (only **33%** occupancy)

- **TILE_WIDTH** = 16

  16x16 = 256 threads >>> 1536/256 = 6 blocks to fully load a SM

  6x256 = 1536 threads per SM … reaching **full occupancy** per SM!

- **TILE_WIDTH** = 32

  32x32 = 1024 threads >>> 1536/1024 = 1.5 = 1 block fully loads SM

  1024 threads per SM (only **66%** occupancy)

TILE_WIDTH = 16

Matrix-Matrix product: 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

- **`TILE_WIDTH`** = 8

  8x8 = 64 threads >>> 2048/64 = 32 blocks needed to fully load a SM

  … yet there is a limit of maximum 16 resident blocks per SM for cc 3.x
  so we end up with just 64x16 = 1024 threads per SM on a maximum of 2048 (only **50%** occupancy)

- **`TILE_WIDTH`** = 16

  16x16 = 256 threads >>> 2048/256 = 8 blocks to fully load a SM

  8x256 = 2048 threads per SM … reaching **full occupancy** per SM!

- **`TILE_WIDTH`** = 32

  32x32 = 1024 threads >>> 2048/1024 = 2 blocks fully load a SM

  2x1024 = 2048 threads per SM … reaching **full occupancy** per SM!

$\boxed{\texttt{TILE\_WIDTH} = 16 \text{ or } 32}$

Matrix-Matrix product: checking error

- Hands on: matrix-matrix product
- Use the proper CUDA API to check error codes
  - use **cudaGetLastError()** to check that kernel has been completed with no errors
- Try to use block size greater than 32x32. What kind of error is reported?

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

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

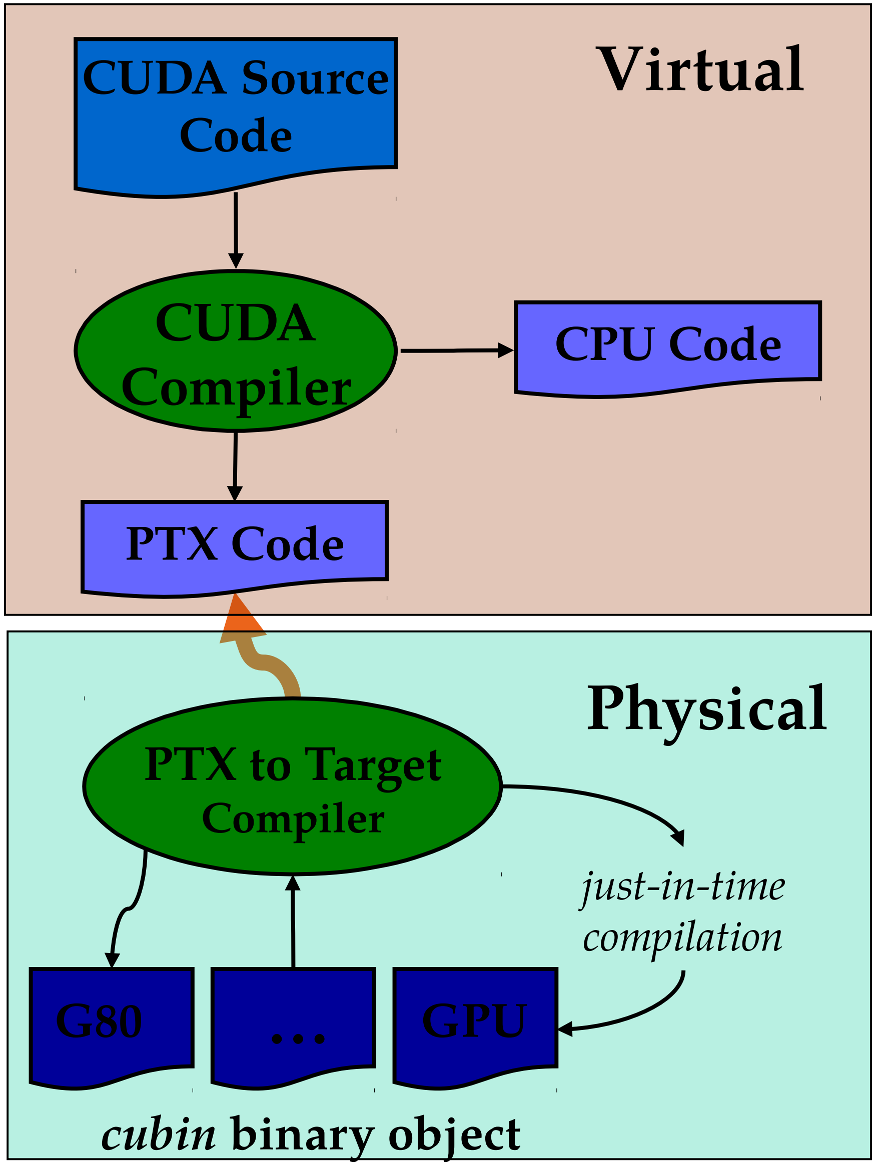
Matrix-Matrix product: performances

- Measure performances of matrix-matrix product, both for CPU and GPU version, using CUDA Events
- Follow these steps:
  - Declare 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 events with: `cudaEventElapsedTime`
  - Destroy events with: `cudaEventDestroy`
- Express performance metric using Gflops, knowing that the matrix-matrix product algorithm requires $2N^3$ operations

	C	Fortran
Gflops		

- Compiling a CUDA program
    - PTX, cubin, what's inside
    - Computing capability
- Hands on:
    - Compiling a CUDA program
    - Environment and utility: deviceQuery and nvidia-smi
    - Vector Sum
    - Matrix Sum

CUDA Compilation Workflow



- Each source file with CUDA extension should be compiled 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 PTC code and object code

How to compile a CUDA program

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

$$\texttt{nvcc}\ \underbrace{\texttt{-arch=compute\_30}}_{\substack{\text{virtual architecture}\\(\textit{PTX code})}}\ \underbrace{\texttt{-code=sm\_30,sm\_37}}_{\substack{\text{real GPU architecture}\\(\textit{cubin})}}$$

  - nvcc allows many shortcut switches as

    `nvcc -arch=sm_37` to target KEPLER K80 architecture

    which is equivalent to:

    `nvcc -arch=compute_37 -code=sm_37`

- **CUDA Fortran**: NVIDIA worked with The Portland Group (PGI) to develop a CUDA Fortran Compiler that provides Fortran language
  - PGI CUDA Fortran does not require a new or separate compiler
  - CUDA features are supported by the same PGI Fortran compiler
  - Use `-Mcuda` option: `pgf90 -Mcuda=cc30`

Hands On

- `deviceQuery` (from the CUDA SDK): show information on CUDA devices
- `nvidia-smi` (NVIDIA System Management Interface): shows diagnostic informations 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 -arch=sm_37 vectoradd_cuda.cu -o vectoradd_cuda`
  - `nvcc -arch=sm_37 -ptx vectoradd_cuda.cu`
  - `nvcc -arch=sm_37 -keep vectoradd_cuda.cu -o vectoradd_cuda`
  - `nvcc -arch=sm_37 -keep -clean vectoradd_cuda.cu -o vectoradd_cuda`
  - Run resulting executable with: `./vectoradd_cuda`

Hands On

- `deviceQuery` (from the CUDA SDK): show information on CUDA devices
- `nvidia-smi` (NVIDIA System Management Interface):
  shows diagnostic informations on present CUDA enabled devices
  (`nvidia-smi -q -d UTILIZATION -l 1`)
- Compile a CUDA program:
  - cd Exercises/VectorAdd. Try the following compiling commands:
  - `pgf90 -Mcuda=cc3x vectoradd_cuda.f90 -o vectoradd_cuda`
  - `pgf90 -Mcuda=cc3x,keepptx -ptx vectoradd_cuda.f90`
  - `pgf90 -Mcuda=cc3x,keepbin vectoradd_cuda.f90 -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 CPU and CUDA versions of the kernel
  - Try CUDA version with different thread block sizes
    (16,16) (32,32) (64,64)
- Home-works:
  - Modify the previous kernel to let in-place sum:
    A = A + c * B

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 transfers can easily become a bottleneck in heterogeneous environment equipped with accelerators
  - Best Practice: minimize transfers between host and device or execute them in overlap with computations

Hands on: measuring bandwidth

- Measure memory bandwidth versus increasing data size, for Host to Device, Device to Host and Device to Device transfers

1. Write a simple program using CUDA events
2. Use `bandwidthTest` provided with CUDA SDK

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

Size (MB)	HtoD	DtoH	DtoD
1			
10			
100			
1024			

Hands on: measuring bandwidth

- Measure memory bandwidth versus increasing data size, for Host to Device, Device to Host and Device to Device transfers

1. Write a simple program using CUDA events
2. Use `bandwidthTest` provided with CUDA SDK

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

Size (MB)	HtoD	DtoH	DtoD
1	2059	2024	69198
10	3493	3076	83274
100	3317	2869	86284
1024	3548	3060	86650

Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini