Introduction to CUDA programming models

Hands-On

September 26-30, 2022

University of Perugia

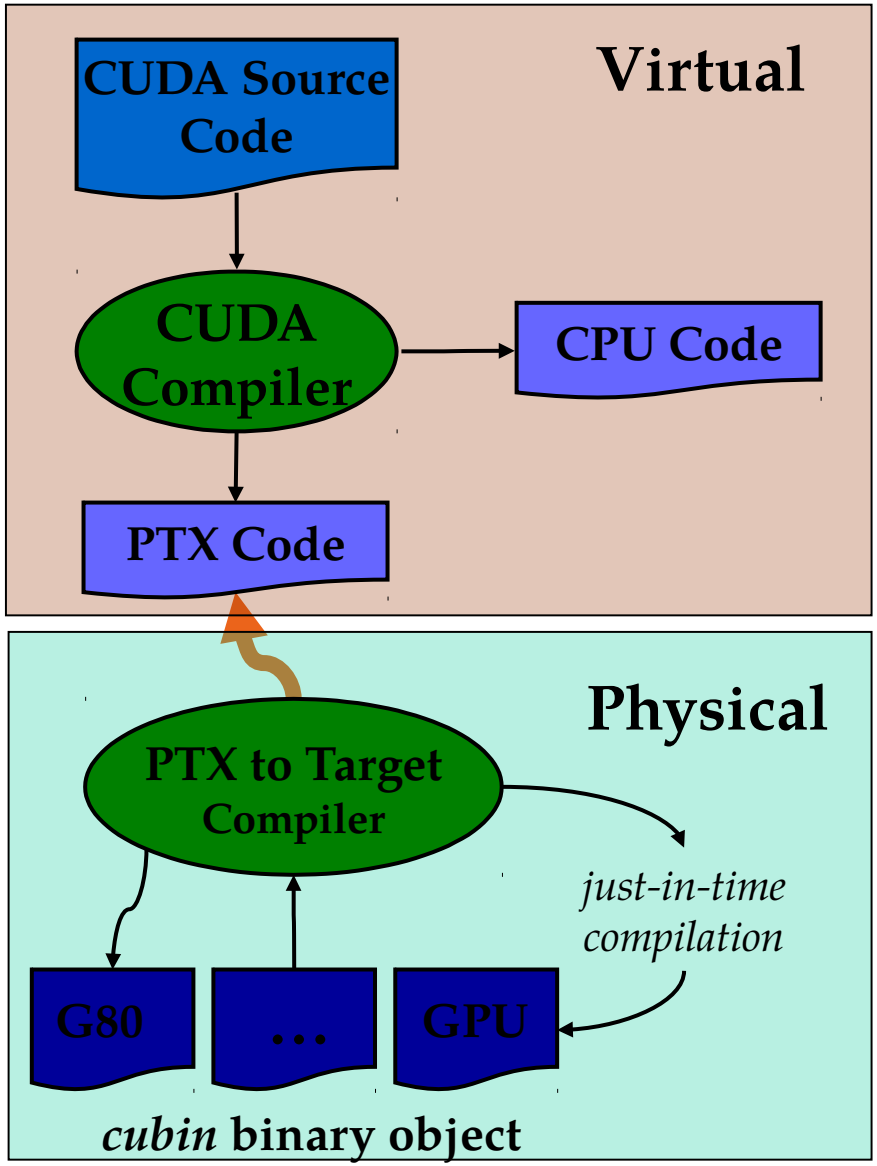
Sergio Orlandini

CINECA

s.orlandini@cineca.it

- Compiling a CUDA program
  - PTX, cubin and Compute Capability
- Hands on:
  - Matrix-Matrix product
  - Matrix Sum
  - Measuring Host/Device Bandwidth
- Measuring Performance
  - CUDA Events

CUDA Compilation Workflow



- Each source file with CUDA extension should be compiled with a proper CUDA aware compiler
  - `nvcc` for CUDA C/C++
  - `pgf90 -Mcuda` for CUDA Fortran
- 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

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

How to compile a CUDA program

- When compiling a CUDA executable, you must specify:
  - compute capability and the architecture targets
- `nvcc` allows many shortcut switches to select architecture as:

  `nvcc -arch=sm_70` to target VOLTA V100 architecture

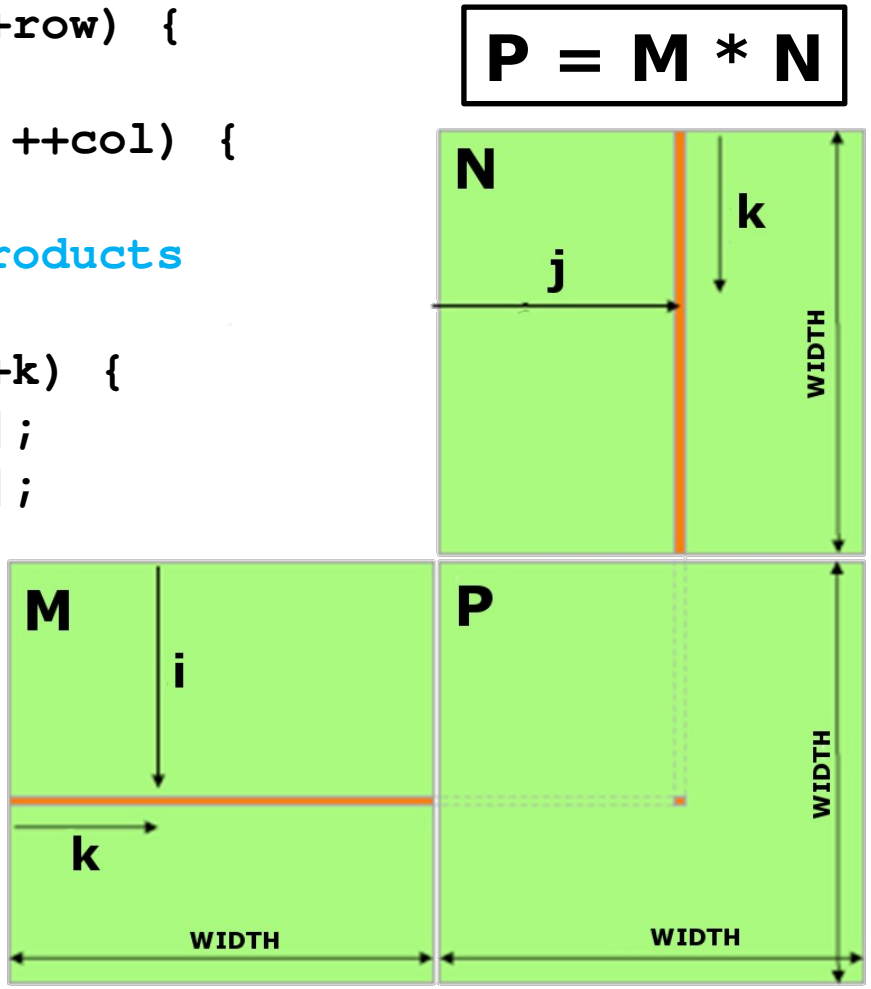
- **CUDA Fortran**: does not require a new or separate compiler
  - CUDA features are supported by the same nvidia Fortran compiler
  - Use `-Mcuda` option: `nvfortran -Mcuda=cc70`

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;
    }
  }
}
```



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;
}
```

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

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

	Compute Capability											
Technical Specifications	**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

Matrix-Matrix product: launch grid

WARNING:

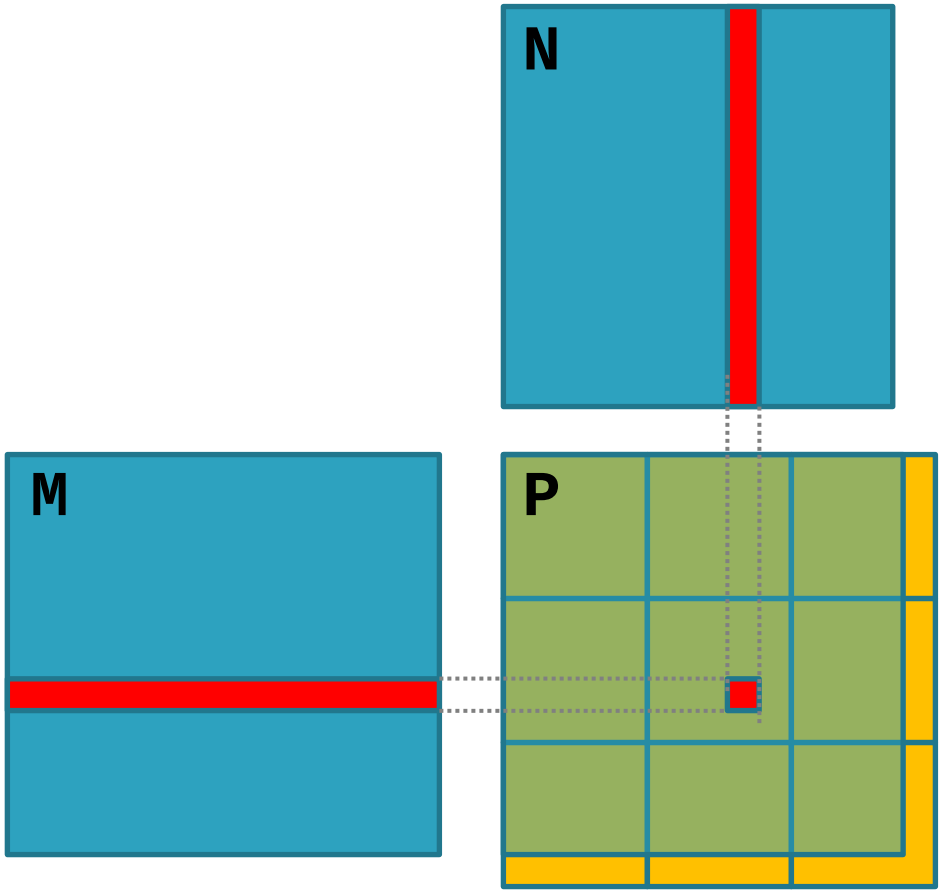
- There is a limit on the maximum number of allowed threads per block
  - It depends on the device architecture (on the **compute capability**)
  - On the newest chipset the maximum number of threads per block is **1024**!
    - In our case we are not able to perform matrix multiplication between matrix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

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

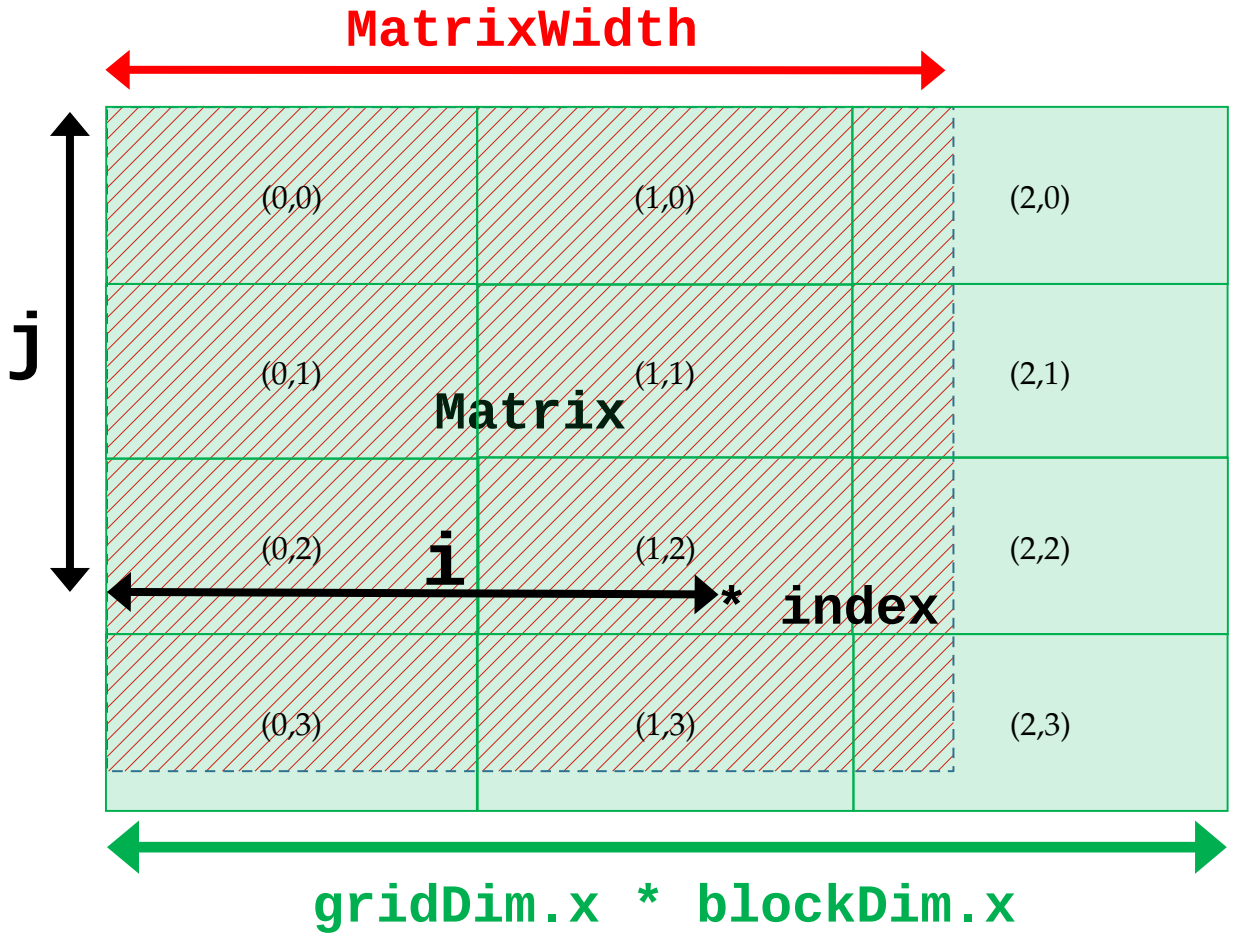
- respect compute capability limits for threads per block
- select the block grid so to cover all elements to be processed
- select block size so that each thread can process one or more data elements without raise conditions with other threads
  - use *builtin* variables `blockIdx` and `blockDim` to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

Matrix-Matrix 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);
}
```

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)

- MatrixMul:
  - Write a program that performs a matrix-matrix multiplication:
    C = A x B

- Home-works:
  - Modify the previous kernel to let in-place sum:
    A = A + c * B

Hands on: measuring bandwidth

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

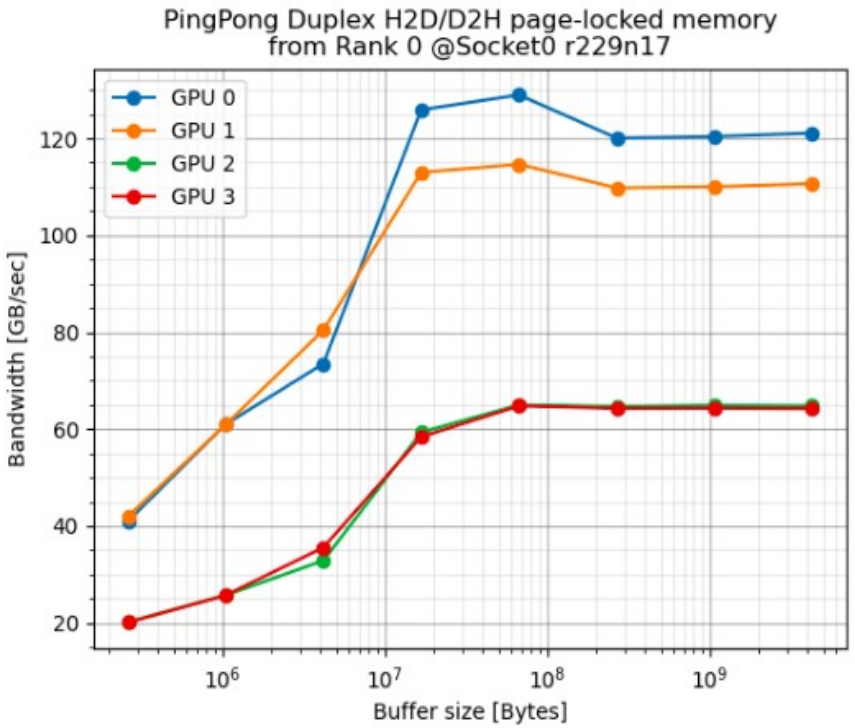
1. Write a simple program using CUDA events
2. Use `bandwidthTest` provided with CUDA SDK

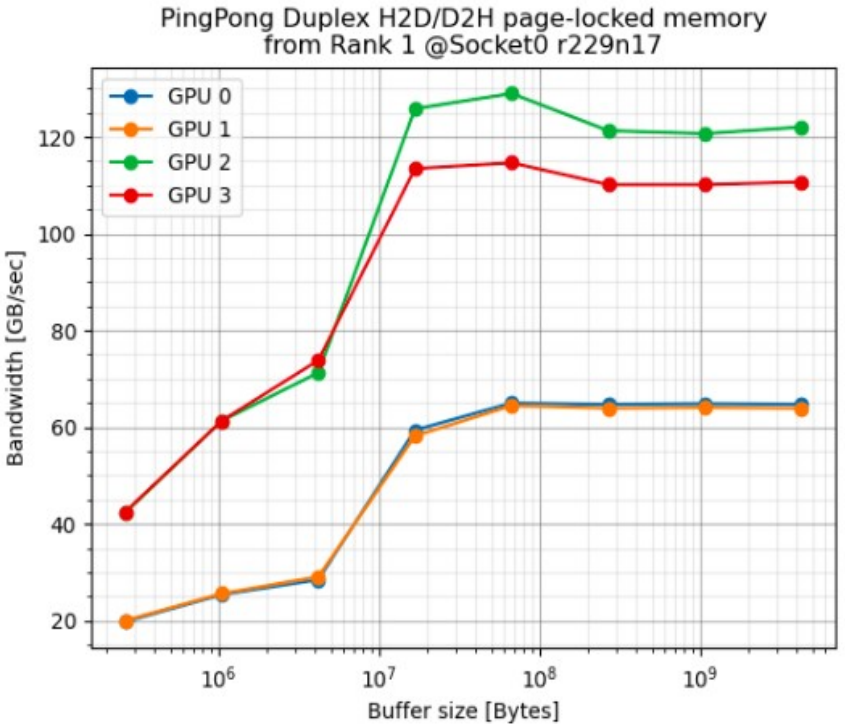
```
./bandwidthTest --mode=range --start=<B> --end=<B> --increment=<B>
```

Size (MB)	HtoD	DtoH	DtoD
1			
10			
100			
1024			

Hands on: measuring bandwidth

H2D/D2H bandwidth in duplex mode with **NVLink** @**Marconi100** with 4 V100





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

```
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);
```

```
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

Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini