Lab 1: A Gentle Introduction to CUDA

Cosmin E. Oancea

Department of Computer Science (DIKU)
University of Copenhagen

September 2025, PMPH Lecture Slides

Accessing the CUDA Servers

How do we measure/reason about Performance?

GPUs: Short Motivation, Desing & Challenges

Basic CUDA Programming

Get CUDA Up and Running

Option 1: Personal computer

- `https://developer.nvidia.com/cuda-downloads`
- Don't do this now!

Get CUDA Up and Running on the Official Servers

The information is provided on the Course github
`https://github.com/diku-dk/pmph-e2025-pub?tab=readme-ov-file#gpu--multicore-machines`

All but a few of you, that have registered late, should already have the accounts set up.

Importantly:

- You need to be connected to VPN, if you are confused please see `https://github.com/diku-dk/howto/blob/main/vpn.md`,
- You need to successfuly ssh to our servers hendrixfut01fl or hendrixfut03fl (see github, previous to that you will need to update your ssh config with a paragraph).
- Once you are there you need to execute:

  ```
  $ module load cuda;
  $ module load futhark;
  ```

  You should probably place the later two in your `$HOME/.bash_profile` or `$HOME/.bashrc` so you do not have to execute them every time.
- You are ready to go if nvcc is found, i.e., try `$ nvcc`

Accessing the CUDA Servers

How do we measure/reason about Performance?

GPUs: Short Motivation, Desing & Challenges

Basic CUDA Programming

What Is Performance? How to Measure it?

(1) What is performance?

Performance measures the degree to which hardware resources are utilized.

(2) How do we measure performance?

2.1 So as to compare the performance of an implementation across datasets?

What Is Performance? How to Measure it?

(1) What is performance?

Performance measures the degree to which hardware resources are utilized.

(2) How do we measure performance?

2.1 So as to compare the performance of an implementation across datasets?

- If program has low arithmetic intensity $\Longrightarrow$ **memory bandwidth/throughput**:

$$\frac{\text{total number of bytes accessed}}{\text{Running time } (\mu s) \cdot 10^3} \quad \text{(GB/sec)}$$

- If program has high arithmetic intensity $\Longrightarrow$ **computational performance**:

$$\frac{\text{total number of float operations}}{\text{Running time } (\mu s) \cdot 10^3} \quad \text{(GFlop/sec)}$$

- If in between $\Longrightarrow$ roofline model.

2.2 How to reason about the degree of hardware utlization?

What Is Performance? How to Measure it?

(1) What is performance?

Performance measures the degree to which hardware resources are utilized.

(2) How do we measure performance?

2.1 So as to compare the performance of an implementation across datasets?

- If program has low arithmetic intensity $\Longrightarrow$ **memory bandwidth/throughput**:

$$\frac{\text{total number of bytes accessed}}{\text{Running time } (\mu s) \cdot 10^3} \quad \text{(GB/sec)}$$

- If program has high arithmetic intensity $\Longrightarrow$ **computational performance**:

$$\frac{\text{total number of float operations}}{\text{Running time } (\mu s) \cdot 10^3} \quad \text{(GFlop/sec)}$$

- If in between $\Longrightarrow$ roofline model.

2.2 How to reason about the degree of hardware utlization?

- compute the percentage achieved by your implementation relative to the peak memory bandwidth or peak flops performance of the hardware.
- if these are not listed, compare your performance with the best-known implementation of your algorithm for a certain hardware type, e.g., Cublas for MMM.

Comparing Performance Across Different Implementations

- . . .

2.3 How to compare performance across datasets & different implementations?

Comparing Performance Across Different Implementations

- ...

2.3 How to compare performance across datasets & different implementations?

- Use the total number of bytes (or float ops) of the "golden sequential" implem!
- If top hardware performance not listed, sometimes it is useful to compare with simpler algorithms that have the same characteristics and are known to have near-optimal performance.

```
// Inclusive Prefix Sum:
// Input:  A = {a_0, ..., a_{n-1}}
// Result: X = {a_0 , a_0 + a_1 , ..., Σ_{i=0}^{n-1} a_i}
float acc = 0;
for(int i=0; i<n; i++) {
    acc = acc + A[i];
    X[i] = acc;
}
```

```
// Memcpy
// Input:  A = {a_0, ..., a_{n-1}}
// Result: X = {a_0, ..., a_{n-1}}

for(int i=0; i<n; i++) {
    X[i] = A[i];
}
```

- **Prefix sum** is challenging to implement efficiently for GPU;
- **Memcpy** is trivial and has the same access pattern: n reads + n writes.
- If your prefix scan reaches 80% of memcpy's parallel performance $\Longrightarrow$ happy!

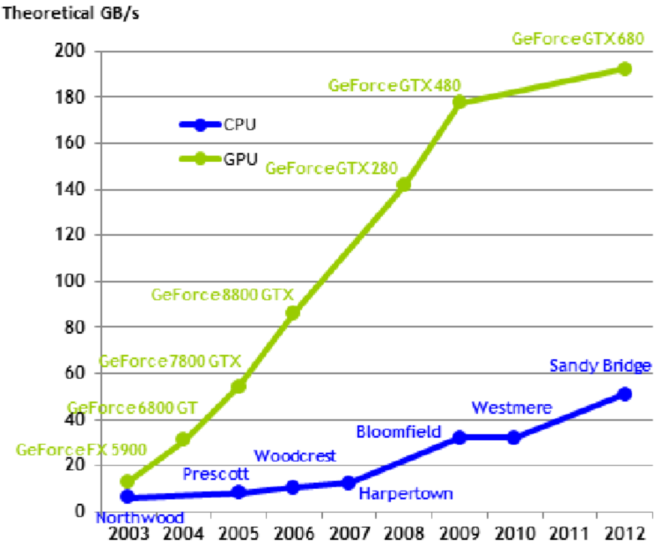
Accessing the CUDA Servers

How do we measure/reason about Performance?

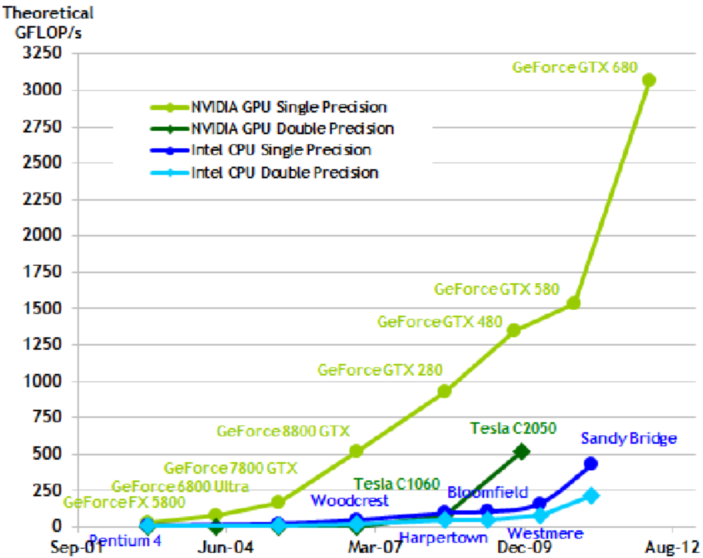
GPUs: Short Motivation, Desing & Challenges

Basic CUDA Programming

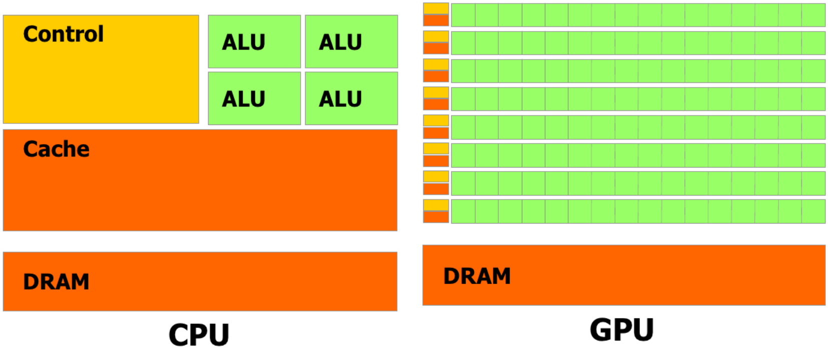
Peak Memory Performance: GPU vs CPU



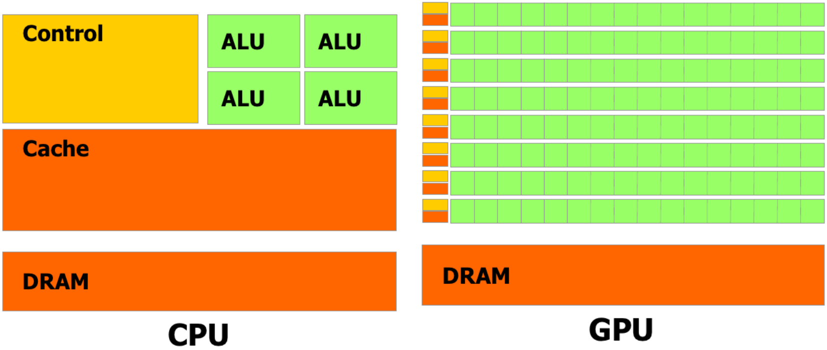
Peak Computational Performance: GPU vs CPU



Key Ideas in GPU Design

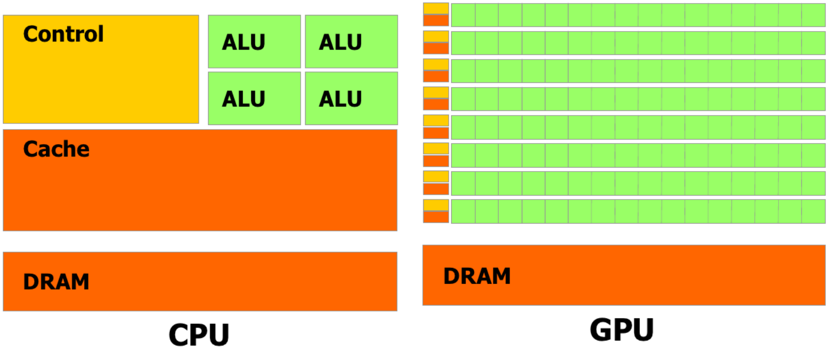


Key Ideas in GPU Design



1. Remove the hardware components that help a single instruction stream run fast,
2. SIMD: amortizes the management of an instruction stream across many ALUs,
3. Aggressively use hardware(-supported) multi-threading to hide latency.

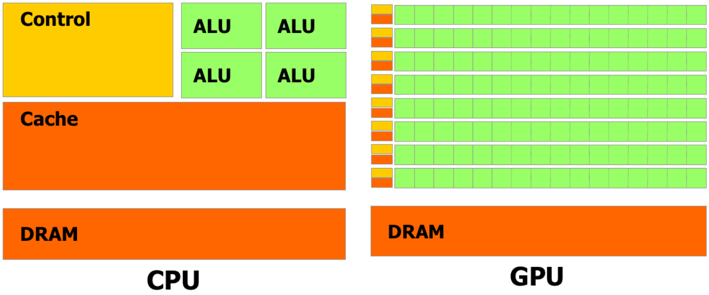
Key Ideas in GPU Design



1. Remove the hardware components that help a single instruction stream run fast,
2. SIMD: amortizes the management of an instruction stream across many ALUs,
3. Aggressively use hardware(-supported) multi-threading to hide latency.

Spatial locality to global memory means "coalesced accesses": threads executing in lock step a load/store SIMD instruction access consecutive memory locations!

CPUs compared to CPUs



- GPUs have *thousands* of simple cores and taking full advantage of their compute power requires *tens/hundred of thousands* of threads.
- GPU threads are very *restricted* in what they can do: no stack, no allocation, limited control flow, etc.
- Potential *very high performance* and *lower power usage* compared to CPUs, but programming them is *hard*.

Accessing the CUDA Servers

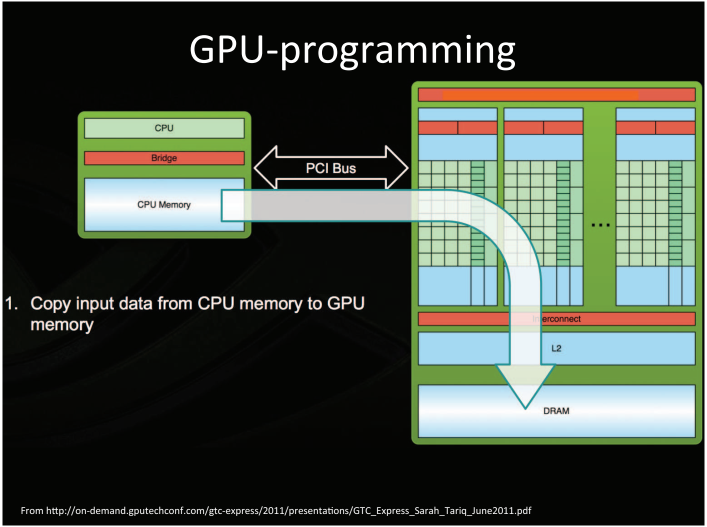
How do we measure/reason about Performance?

GPUs: Short Motivation, Desing & Challenges

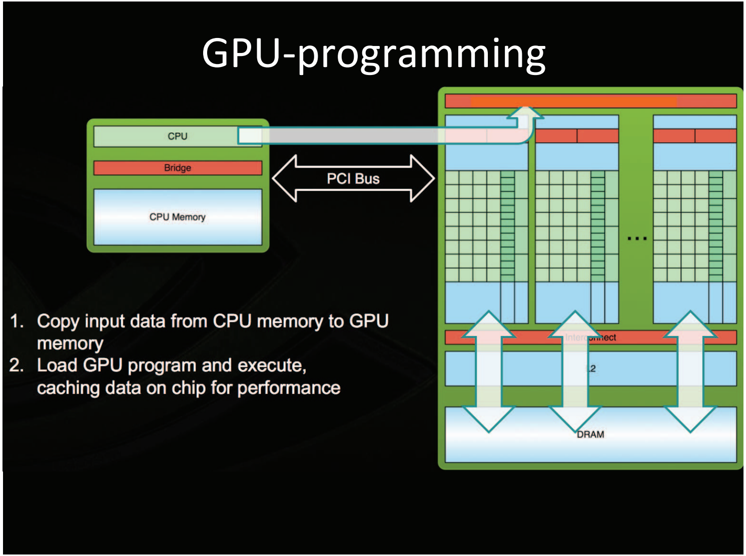
Basic CUDA Programming

Basic GPU Programming

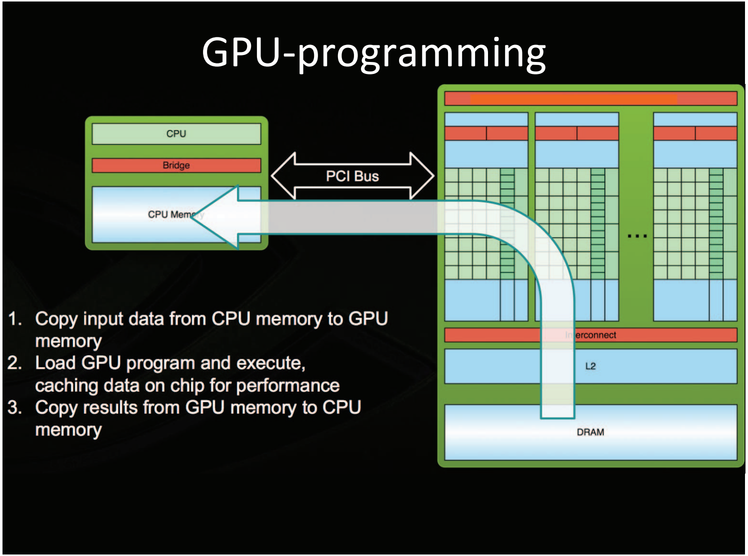
The device (GPU) and host (CPU) have different memory spaces!



Basic GPU Programming

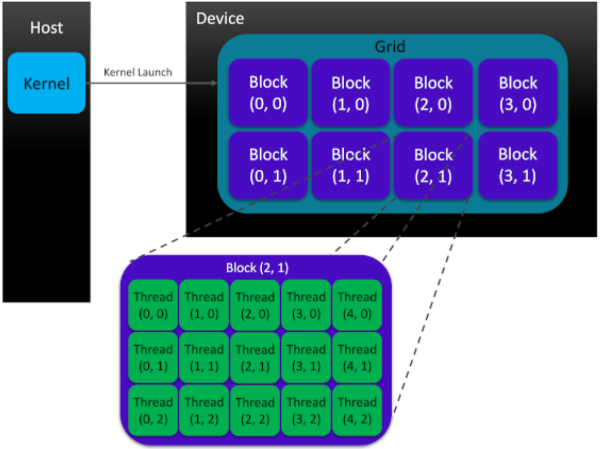


Basic GPGPU programming



Cuda: Grid-Block Structure of Threads

Credit: pictures taken from `http://education.molssi.org/gpu_programming_beginner/03-cuda-`

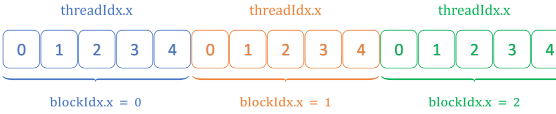


Blocks and Grids have at most three dimensions—denoted x, y, z, with x innermost and z outermost. Their sizes are specified at kernel launch. Inside the **kernel** you may use:

- `blockDim.x`: block size in dim x
- `blockIdx.x`: current block index (in x)
- `threadIdx.x`: local index of the current thread inside its block (in dim x)
- `gridDim.x` number of blocks on dim x
- Ditto for dimensions y and z.

The global thread index in dim $q \in \{x, y, z\}$**:**

$$\texttt{threadIdx.q} + \texttt{blockIdx.q} \cdot \texttt{blockDim.q}$$



Multiply with 2 Each Element of an Array in CUDA

Golden Sequential:

```
// Y and X are arrays of length N
for(int i=0; i<N; i++) {
    Y[i] = 2.0 * X[i];
}
```

How do we do this in CUDA?

- We provide a very naive version in this folder `https://github.com/diku-dk/pmph-e2024-pub/tree/main/HelperCode/Lab-1-Cuda`
  Please be advised that it works correctly only for arrays of ≤ 1024 elements.
- Then we will provide instruction/code in the slides so that you can type in a generally-correct solution.

First let's examine the available code. The actual code is a bit larger than the one on the slides since it also performs runtime measurement and validation.

A Simple CUDA Program

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cuda_runtime.h>

// CUDA kernel:
__global__ void mul2Kernel(float* X, float *Y) {
    const unsigned int gid = threadIdx.x; // threadIdx.x is the local id inside a
                                          // CUDA block; Ok since we only have 1 block.
    Y[gid] = 2 * X[gid];                  // computes and updates the result array
}
int main(int argc, char** argv) {
    unsigned int N = atoi(argv[1]);
    unsigned int mem_size = N*sizeof(float);

    // allocate host memory
    float* h_in  = (float*) malloc(mem_size);
    float* h_out = (float*) malloc(mem_size);

    // initialize the memory
    for(unsigned int i=0; i<N; ++i) { h_in[i] = (float)i; }
```

A Simple CUDA Program (continuation)

```
    float *d_in, *d_out;
    cudaMalloc((void**)&d_in,  mem_size);                      // allocate device memory
    cudaMalloc((void**)&d_out, mem_size);


    cudaMemcpy(d_in, h_in, mem_size, cudaMemcpyHostToDevice); // copy host input to device

    for(int r = 0; r < GPU_RUNS; r++) {
        mul2Kernel<<< 1, N>>>(d_in, d_out);                    // execute the kernel
    }
    cudaDeviceSynchronize();
    // ^ Needed only for measuring runtime (see longer comment in actual code)
    //   Please note that the execution of multiple kernels in Cuda runs correctly
    //     without such explicit synchronizations (which are expensive).

    gpuAssert( cudaPeekAtLastError() );                        // check for errors

    cudaMemcpy(h_out, d_out, mem_size,cudaMemcpyDeviceToHost);//copy device result to host

    free(h_in);   free(h_out);   cudaFree(d_in);   cudaFree(d_out); // clean-up memory
}
```

Save, Compile, Run

```
$ nvcc -O3 -o trivial trivial.cu

$ ./trivial
```

Or even better, use and adjust the Makefile.

Trouble Ahead

What if our array contains a larger number of elements, say 32757?

This shouldn't be a problem with our program (adapt the kernel)

- GPU logical threads organized in a grid of blocks, in which the grid and the block can have up to three dimensions.
- However CUDA does not accept a block of size 32757
  - a *CUDA warp* is formed by 32 threads that execute SIMD.
  - a *a CUDA block* may contain up to 1024 threads (included); ideally the block size is a multiple of 32, but not necessarily.
  - Synchronization/communication is possible inside a CUDA block by means of barriers & scratchpad memory (shared memory).
  - Barrier synchronization is not possible across threads in different CUDA blocks, i.e., only by finishing the kernel!

Trouble Ahead

What if our array contains a larger number of elements, say 32757?

This shouldn’t be a problem with our program (adapt the kernel)

- GPU logical threads organized in a grid of blocks, in which the grid and the block can have up to three dimensions.
- However CUDA does not accept a block of size 32757
  - a *CUDA warp* is formed by 32 threads that execute SIMD.
  - a *a CUDA block* may contain up to 1024 threads (included); ideally the block size is a multiple of 32, but not necessarily.
  - Synchronization/communication is possible inside a CUDA block by means of barriers & scratchpad memory (shared memory).
  - Barrier synchronization is not possible across threads in different CUDA blocks, i.e., only by finishing the kernel!
- Finally if the size of the computation does not matches exactly a multiple of block size, then you need to spawn extra threads, hence you need to add an `if` inside the kernel code, to make the extra threads iddle!

GPGPU in More Detail

- A set of Streaming Multiprocessors (SMs)

```
From deviceQuery:
 (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
```

- Each SM executes 1 ‘thread block’ at a time.
- Each block has access to
  - Global memory (function arguments)

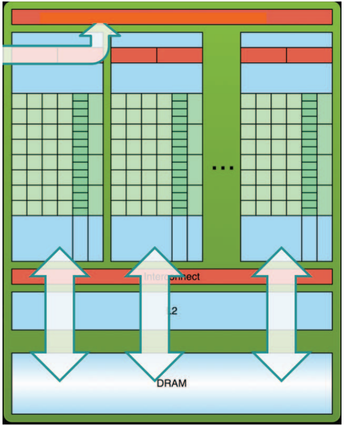
```
From deviceQuery:
 Total amount of global memory:                  3072 MBytes
```

  - Shared memory (`__shared__ int array[512]`)

```
From deviceQuery:
Total amount of shared memory per block:       49152 bytes
```

  - Local memory (local variables)

```
From deviceQuery:
 Total number of registers available per block: 65536
```



Step 1 in Fixing Our CUDA Program

Golden Sequential:

```
// Y and X are arrays of length N
for(int i=0; i<N; i++) {
    Y[i] = 2.0 * X[i];
}
```

Step 1 in Fixing Our CUDA Program

Golden Sequential:

```
// Y and X are arrays of length N
for(int i=0; i<N; i++) {
    Y[i] = 2.0 * X[i];
}
```

In order to handle values of *N* greater than the CUDA-block size (1024), we need to modify the host code to orchestrate the kernel into a grid of multiple blocks:

- We use a 1D grid and a 1D block,
- We chose a “good block size”, for example 128, 256,
- we compute how many blocks our parallel dimension requires; this is ”the grid”,
- we update the kernel call with the new grid/block and also pass *N* as parameter!

Calling the kernel from host/CPU-executed code:

```
unsigned int B = 256;  // chose a suitable block size in dimension x
unsigned int numblocks = (N + B - 1) / B; // number of blocks in dimension x
dim3 block(B,1,1), grid(numblocks,1,1);  // total number of threads (numblocks*B) may overshoot N!
mul2Kernel<<<grid, block>>>(d_in, d_out, N); // pass N as parameter as well;
                                             // d_in and d_out are in device memory
```

Step 2 in Fixing Our CUDA Program

Golden Sequential:

```
// Y and X are arrays of length N
for(int i=0; i<N; i++) {
    Y[i] = 2.0 * X[i];
}
```

Step 2 in Fixing Our CUDA Program

Golden Sequential:

```
// Y and X are arrays of length N
for(int i=0; i<N; i++) {
    Y[i] = 2.0 * X[i];
}
```

We modify the CUDA Kernel:

1. to contain as extra/third parameter the array length *N*
2. to compute correctly the global thread id
   (now that we have multiple CUDA blocks, we cannot use `threadIdx.x`)
3. to perform the write to global memory *if and only if* the global id is within the array bounds.

```
__global__ void mul2Kernel(float* X, float* Y, int N) {
  // compute global thread id in dimension x
  const unsigned int gid = blockIdx.x * blockDim.x + threadIdx.x;
  if( gid < N ) {  // don't access out of bounds
    Y[gid] = 2.0 * X[gid];
  }
}
```

How fast does it go?

The program already contains built-in validation and measuring of performance in terms of runtime and more importantly of Gflops/sec. Once it validates, run your program for a big *N*, e.g., 200 millions; what performance do you obtain?

The peak bandwidth of our A100 GPU is about 1.55 TBytes/sec. What do you get?

How is performance affected if you change the line of your kernel

```
Y[gid] = 2.0 * X[gid];
```

to

```
Y[threadIdx.x*gridDim.x + blockIdx.x] = 2.0 * X[gid];
```

The reason is that good spatial locality for global memory is achieved when a warp of consecutive threads access in their SIMD load/store instruction contiguous memory locations. This is dubbed "coalesced access". The latter access pattern is uncoalesced, hence performance should suffer.