

# Heterogeneous computing with performance modelling

## GPU programming basics

Mirko Myllykoski  
mirkom@cs.umu.se

Department of Computing Science and HPC2N  
Umeå University

4-5. November 2020



UMEÅ  
UNIVERSITY



**SNIC**



**HPC2N**



Why should I use GPUs?

Why should I use GPUs?

# Why should I use GPUs?

- ▶ Nvidia says that when compared to CPUs, GPUs provides significantly higher
  - ▶ **instruction throughput** and
  - ▶ **memory bandwidth.**
- ▶ In particular, this performance is delivered in a relatively small price and power envelope.

What does this mean?

# Why should I use GPUs? (flop rate, double precision)

- ▶ Quad-core Intel Skylake CPU:

~ 200 GFlops

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 1200 GFlops

- ▶ Nvidia Tesla V100 GPU:

~ **7 000** GFlops

# Why should I use GPUs? (memory bandwidth)

- ▶ Quad-core Intel Skylake CPU:

~ 35 GB/s

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 100 GB/s

- ▶ Nvidia Tesla V100 GPU:

~ **900** GB/s

# Why should I use GPUs? (power usage)

- ▶ Quad-core Intel Skylake CPU:

~ 90W

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 140 W

- ▶ Nvidia Tesla V100 GPU:

~ **250 W**

- ▶ These numbers are not entirely accurate.

# Why should I use GPUs? (price)

- ▶ Quad-core Intel Skylake CPU:

~ \$400

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ \$2 000

- ▶ Nvidia Tesla V100 GPU:

~ **\$10 000**

- ▶ These numbers are not entirely accurate.

# Why should I use GPUs? (per Watt)

- ▶ Quad-core Intel Skylake CPU:

~ 2 GFlops per Watt, ~ 0.4 GB/s per Watt

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 9 GFlops per Watt, ~ 0.7 GB/s per Watt

- ▶ Nvidia Tesla V100 GPU:

~ **28** GFlops per Watt, ~ **3.6** GB/s per Watt



# Why should I use GPUs? (per dollar)

- ▶ Quad-core Intel Skylake CPU:

~ 0.5 GFlops per \$, ~ 0.09 GB/s per \$

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 0.6 GFlops per \$, ~ 0.05 GB/s per \$

- ▶ Nvidia Tesla V100 GPU:

~ **0.7** GFlops per \$, ~ **0.09** GB/s per \$

# Why should I use GPUs? (reduced precision)

- ▶ Quad-core Intel Skylake CPU:

~ 400 GFlops (single precision)

- ▶ 14-core Intel Xeon Gold 6132 CPU:

~ 2400 GFlops (single precision)

- ▶ Nvidia Tesla V100 GPU:

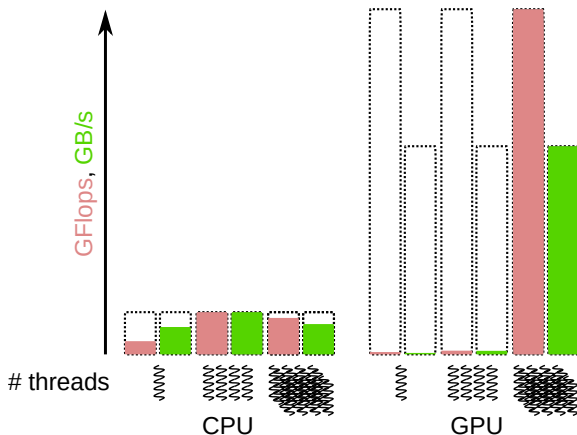
~ 14 000 GFlops (single precision)  
~ **112 000** GFlops (half precision, tensor cores)

What is the catch?

What is the catch?

# What is the catch? (highly parallel)

- ▶ GPUs are highly parallel. Obtaining peak performance requires **thousands of threads**.
- ▶ The algorithms and the software must be highly parallel.



# What is the catch? (long list continues)

- ▶ GPU programming is difficult.
  - ▶ As mentioned, requires a lot of parallelism.
  - ▶ Usually, the GPU-side code is written in "C++".
- ▶ The CUDA cores are not "true" cores.
  - ▶ The cores share resources such as schedulers and caches.
  - ▶ The tensor cores (half precision) are even more limited.
- ▶ Limited amount of memory.
  - ▶ Data must be moved between memories (RAM, VRAM, ...).
  - ▶ Certain memories are significantly faster than others.
  - ▶ Certain memories are fast only when used correctly.
- ▶ Not all algorithms are suitable for GPUs.
  - ▶ Some algorithms are just slightly faster on GPUs.
  - ▶ Does it make sense to pay the extra money?
- ▶ Not all GPUs are equally powerful.
  - ▶ Some GPUs have a very low double-precision flop rate.
  - ▶ Some GPUs do not have tensor cores.

Lets go through some CUDA basics...

# What is CUDA?

- ▶ **Compute Unified Device Architecture (CUDA)** is a parallel computing platform and an API created by Nvidia.
  - ▶ Only for Nvidia GPUs.
- ▶ Can be used
  - ▶ directly through CUDA C/C++ and CUDA Fortran,
  - ▶ directly through wrappers (Python, Perl, Fortran, Java, Ruby, Lua, etc),
  - ▶ indirectly through compiler directives (OpenACC), and
  - ▶ indirectly through other computational interfaces (OpenCL, DirectCompute, OpenGL, etc).
- ▶ During this training course, we will use CUDA C/C++.

# What is CUDA? (nvcc and libraries)

- ▶ CUDA comes with its own compiler, `nvcc`.
  - ▶ GPU-specific code is compiled to **PTX** (Parallel Thread Execution) "assembly".
  - ▶ The PTX code is translated to binary code by the driver.
  - ▶ The rest is offloaded to the host compiler (e.g., `g++`).
- ▶ CUDA also comes with a large set of libraries:
  - ▶ CUDA Runtime library (CUDART), basic functionality.
  - ▶ CUDA Basic Linear Algebra Subroutines (cuBLAS).
  - ▶ CUDA Fast Fourier Transform library (cuFFT).
  - ▶ CUDA Sparse Matrix library (cuSPARSE).
  - ▶ CUDA Deep Neural Network library (cuDNN).
- ▶ Many 3rd party libraries use these vendor libraries.
- ▶ In most cases, you want to use these libraries instead of implementing your own.



# Hello world

- ▶ A "Hello world" program (hello.cu) is a good place to start:

```
#include <stdlib.h>
#include <stdio.h>

__global__ void say_hello()
{
    printf("GPU says, Hello world!\n");
}

int main()
{
    printf("Host says, Hello world!\n");
    say_hello<<<1,1>>>>();
    cudaDeviceSynchronize();

    return EXIT_SUCCESS;
}
```

# Hello world (compile and run)

- ▶ Load the correct toolchain:

```
$ ml purge  
$ ml fosscuda/2019b buildenv
```

- ▶ Compile the source code with `nvcc`:

```
$ nvcc -o hello hello.cu
```

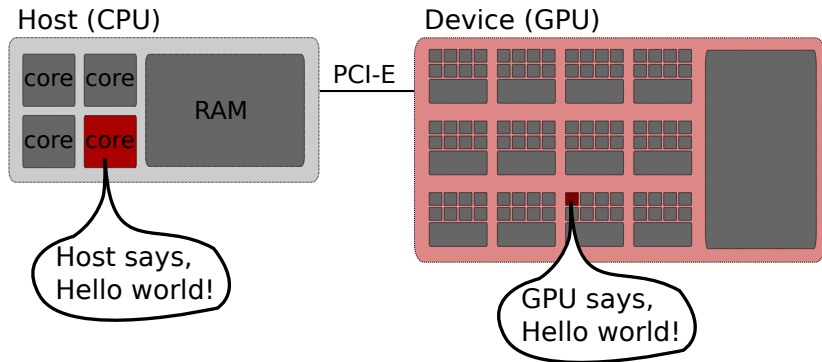
- ▶ Queue a job:

```
$ srun --account=SNIC2020-9-161 \  
--reservation=snic2020-9-161-day1 \  
--ntasks=1 --gres=gpu:v100:1,gpuexcl \  
--time=00:02:00 ./hello
```

Host says, Hello world!

GPU says, Hello world!

# Hello world (what is happening)



We have three objects:

**Host** CPUs + main memory (RAM)

**Device** A GPU consisting of several CUDA cores

**PCI-E** Fast interconnect between the host and the device

# Hello world (kernels)

- ▶ The device-side code is written inside special functions called **kernels**.
- ▶ A kernel is declared with the `__global__` keyword:

```
__global__ void say_hello()  
{  
    printf("GPU says, Hello world!\n");  
}
```

- ▶ Kernel's return type is always `void`.
- ▶ All threads enter the kernel from the beginning of the body of the function.
  - ▶ Single Instruction, Multiple Thread (SIMT)
  - ▶ Threads are not spawned fork-join style (except when a kernel launches other kernel).

# Hello world (kernel launch)

- ▶ The host **issues** the `say_hello` kernel as follows:

```
|| say_hello<<<1,1>>>();
```

- ▶ This places the kernel call into a queue known as **stream**.
  - ▶ Note that the kernel is not guaranteed to be executed!
- ▶ We will return to the `<<< . , . >>>` brackets later...
  - ▶ For now, you need to know that the number of threads is one.
- ▶ The `cudaDeviceSynchronize` call causes the host to wait until stream is empty, i.e., the kernel has finished:

```
|| cudaDeviceSynchronize();
```

# Hello world (summary)

```
#include <stdlib.h>
#include <stdio.h>

// kernel
__global__ void say_hello()
{
    // the device (GPU) executes these lines
    printf("GPU says, Hello world!\n");
}

int main()
{
    // the host (CPU) executes these lines

    printf("Host says, Hello world!\n");

    // issue the say_hello kernel
    say_hello<<<1,1>>>>();

    // wait until the kernel has finished
    cudaDeviceSynchronize();

    return EXIT_SUCCESS;
}
```

# AX example (scalar-vector multiplication)

- ▶ Let's try something more complicated:

$$\alpha \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^n$$

$$\mathbf{x} \leftarrow \alpha \mathbf{x}$$

- ▶ A host function would look something like this:

```
void ax(int n, double alpha, double *x)
{
    for (int i = 0; i < n; i++)
        x[i] = alpha * x[i];
}
```

# AX example (kernel)

- ▶ A matching kernel is still relatively simple:

```
__global__ void ax_kernel(int n, double alpha, double *x)
{
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

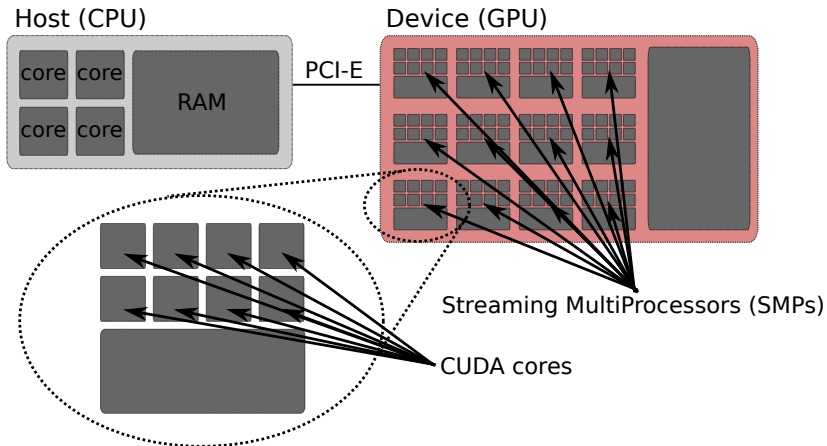
    if (thread_id < n)
        x[thread_id] = alpha * x[thread_id];
}
```

- ▶ What are blockIdx.x, blockDim.x and threadIdx.x?
- ▶ Where is the for loop?
- ▶ Why is there a if block?



# AX example (CUDA cores and SMPs)

- ▶ Let's look at the earlier figure again...

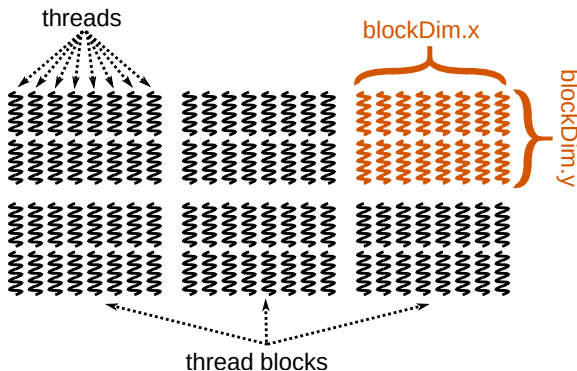


# AX example (CUDA cores and SMPs)

- ▶ Each **Streaming MultiProcessor** (SMP) consists of several **CUDA cores**.
- ▶ Each CUDA core **can execute several threads simultaneously**.
  - ▶ The scheduler select the next instruction among a pool of active threads.
- ▶ Thus, the total number of threads can be in the millions.
- ▶ How do we decide what each thread should do?
- ▶ How do we manage all these threads?
  - ▶ Different problems sizes might require different number of threads.
  - ▶ Different GPUs might have different number of SMPs and CUDA cores.

# AX example (threads and thread blocks)

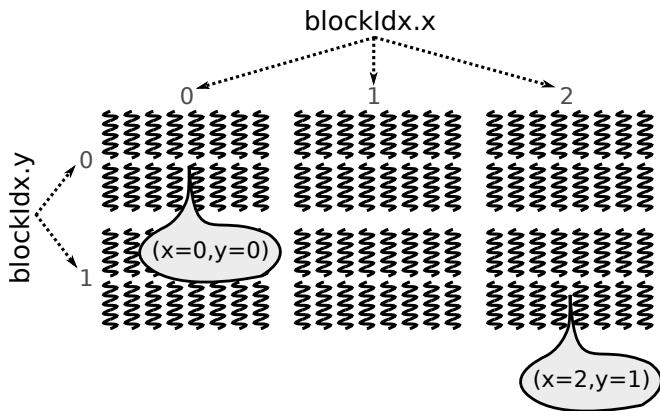
- ▶ The threads are divided into **thread blocks**:



- ▶ `blockDim.x` and `blockDim.y` are the dimensions of the thread blocks.

# AX example (threads and thread blocks)

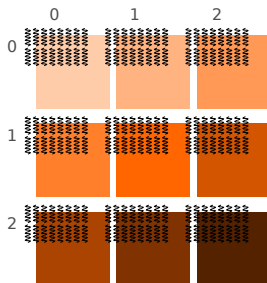
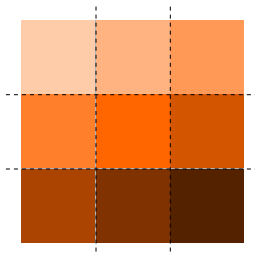
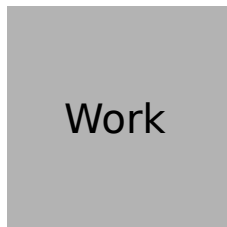
- ▶ Each thread block gets an **index number** in a **grid**:



- ▶ `gridDim.x` and `gridDim.y` are the dimensions of the grid.

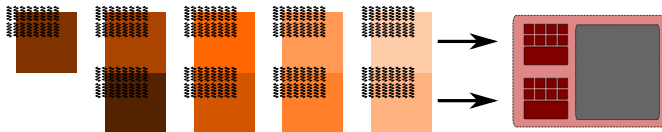
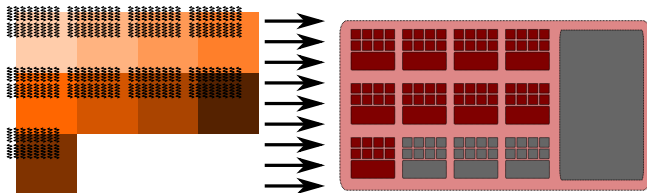
# AX example (threads and thread blocks)

- ▶ The overall idea is to **partition** the work into self-contained tasks.
- ▶ **Each task is assigned to one thread block.**
  - ▶ The thread block indices are used to identify the task.



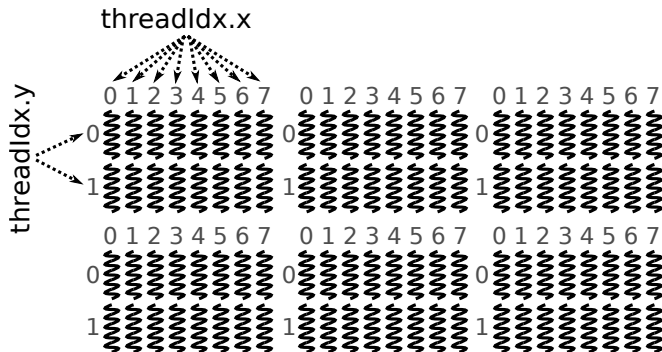
# AX example (threads and thread blocks)

- ▶ The CUDA runtime is responsible for scheduling the thread blocks to SMPs.
- ▶ The execution order of the thread blocks is **relaxed**.
  - ▶ The code can therefore adapt to different GPUs:



# AX example (threads and thread blocks)

- Each thread gets a **local** index number:

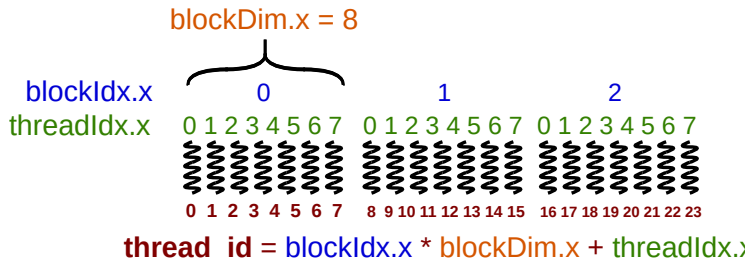


# AX example (global indexing)

- A **unique** global index number can be calculated for each thread:

```
--global__ void ax_kernel(int n, double alpha, double *x)
{
    // query the global thread index
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    if (thread_id < n)
        x[thread_id] = alpha * x[thread_id];
}
```



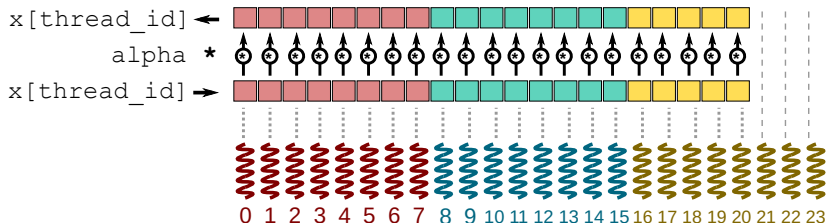


# AX example (global indexing)

- ▶ The if block is used to filter out excess threads:

```
__global__ void ax_kernel(int n, double alpha, double *x)
{
    // query the global thread index
    int thread_id = blockIdx.x * blockDim.x + threadIdx.x;

    // each thread updates one row
    if (thread_id < n)
        x[thread_id] = alpha * x[thread_id];
}
```



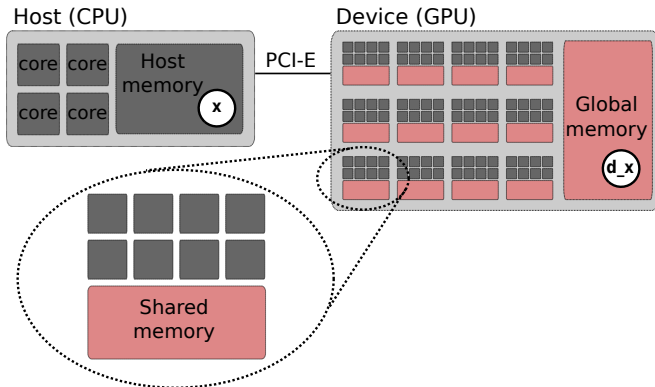
## AX example (remarks)

- ▶ The thread blocks should be reasonably large.
  - ▶ Usually 32 threads is too small for practical use.
  - ▶ At the moment, the upper limit is 1024 thread.
  - ▶ My personal recommendation is to **start from 256 threads**, and tune the number if necessary.
  - ▶ Multiples of 32 are preferred (warps, more on that later).
- ▶ The grid should be reasonably large.
  - ▶ **Each thread block runs on a single SMP.**
  - ▶ For optimal performance, each SMP should get a thread block.
  - ▶ Nvidia Tesla V100 GPU has 80 SMPs.
- ▶ Given a thread block of the size (Dx, Dy, Dz), the hardware indexes a thread of index (x, y, z) as  $(x + y \cdot Dx + z \cdot Dx \cdot Dy)$ .

# AX example (memory spaces)

- ▶ The host manages the memory:

```
double *x = (double *) malloc(n*sizeof(double));  
for (int i = 0; i < n; i++)  
    x[i] = i;  
  
double *d_x;  
cudaMalloc(&d_x, n*sizeof(double));
```

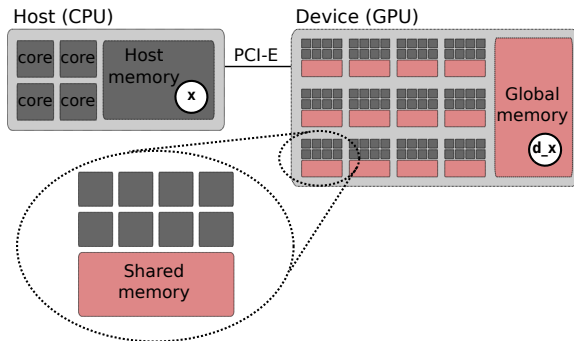


# AX example (memory spaces)

**Host memory** is accessible by the **host** (and sometimes by all threads in all thread blocks).

**Global memory** is accessible by **all threads** in **all thread blocks**.

**Shared memory** is accessible by threads that **belong to the same thread block**.

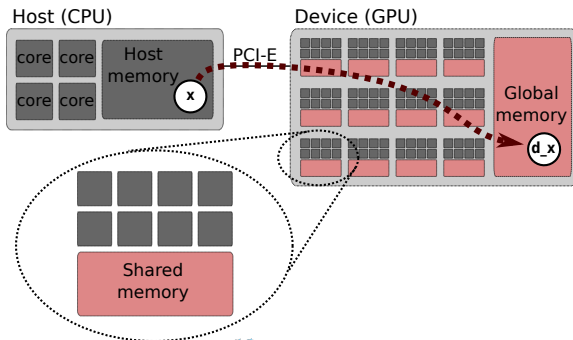


# AX example (memory transfer to device)

- ▶ The host initializes a data transfer from the host memory to the global memory:

```
...  
cudaMemcpy(d_x, x, n*sizeof(double), cudaMemcpyHostToDevice);
```

- ▶ The cudaMemcpy call is **blocking**.
  - ▶ The host waits until the transfer is ready.



# AX example (kernel launch)

- ▶ The host issues the `ax_kernel` kernel:

```
...  
dim3 threads = 256; // blockDim.x  
dim3 blocks = (n+threads.x-1)/threads.x; // gridDim.x  
ax_kernel<<<blocks, threads>>>(n, alpha, d_x);
```

- ▶ If the kernel used multi-dimensional threads blocks, then

```
...  
dim3 threads(Dx, Dy, Dz);  
dim3 blocks(Gx, Gy, Gz)  
ax_kernel<<<blocks, threads>>>(n, alpha, d_x);
```

would create a  $Gx \times Gy \times Gz$  grid of thread blocks, with  $Dx \times Dy \times Dz$  threads in each block.

# AX example (kernel launch)

- ▶ Alternatively, we could have written

```
...  
ax_kernel<<<(n+255)/256, 256>>>(n, alpha, d_x);
```

- ▶  $(n + \text{blockDim.x} - 1) / \text{blockDim.x}$  is simply a convenient way of making sure that

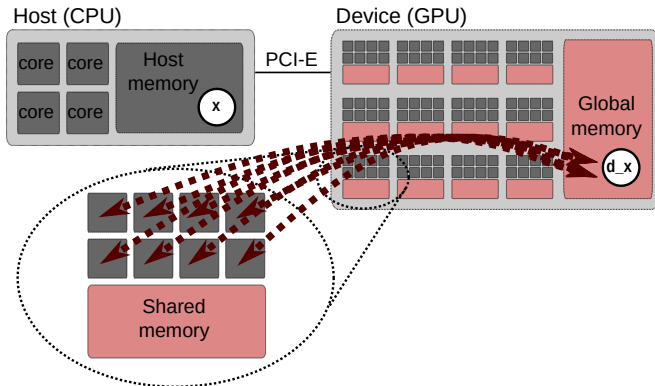
$$n \leq \text{gridDim.x} * \text{blockDim.x}.$$

- ▶ I personally prefer the following approach:

```
// a function that returns the ceil of a/b. That is,  
//   DIVCEIL(5, 2) = ceil(5/2) = ceil(2.5) = 3.  
static int DIVCEIL(int a, int b)  
{  
    return (a+b-1)/b;  
}  
  
...  
ax_kernel<<<DIVCEIL(n, 256), 256>>>(n, alpha, d_x);
```

# AX example (kernel launch)

- ▶ When the kernel finally starts, the threads access the data from the global memory:

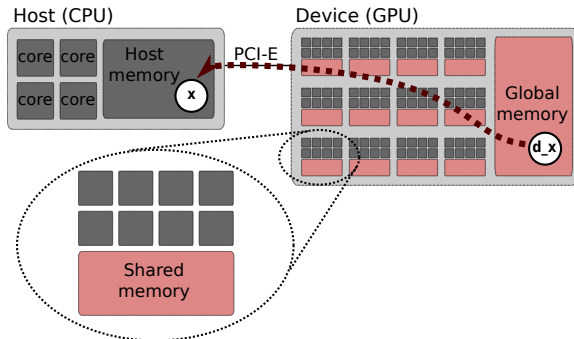




# AX example (memory transfers from device)

- ▶ The host initializes a data transfer from the global memory to the host memory:

```
...  
cudaMemcpy(x, d_x, n*sizeof(double), cudaMemcpyDeviceToHost)
```



# AX example (cleanup)

- ▶ Finally, we must free the allocated memory:

```
|| ...  
|| free(x);  
|| cudaFree(d_x);
```

- ▶ All memory that has been allocated using the cudaMalloc function **must** be freed with the cudaFree function.
- ▶ The following will cause a **segmentation fault**:

```
|| ...  
|| free(d_x);
```

# AX example (compile and run)

- ▶ Load the correct toolchain:

```
$ ml purge  
$ ml fosscuda/2019b buildenv
```

- ▶ Compile the source code with `nvcc`:

```
$ nvcc -o ax ax.cu
```

- ▶ Queue a job:

```
$ srun --account=SNIC2020-9-161 \  
--reservation=snic2020-9-161-day1 \  
--ntasks=1 --gres=gpu:v100:1,gpuexcl \  
--time=00:02:00 ./ax 10000  
Residual = 0.000000e+00
```

# Error handling (queries)

- ▶ Most CUDA functions return an error code of the type `cudaError_t`.
- ▶ A successful function call returns `cudaSuccess`. Other values indicate error.
- ▶ A kernel launch does not return anything.
  - ▶ Any errors must be queried separately. See below.
- ▶ The previous error code can be checked and **resetted** with:

```
|| __host__ __device__ cudaError_t cudaGetLastError()
```

# Error handling (queries)

- ▶ The previous error code can be checked without resetting:

```
|| __host__ __device__ cudaError_t cudaPeekAtLastError()
```

- ▶ An error code can be turned into a string:

```
|| __host__ __device__ const char* cudaGetErrorName(cudaError_t error)
```

- ▶ An error code can be turned into a longer description:

```
|| __host__ __device__ const char* cudaGetErrorString(cudaError_t error)
```

# Error handling (some notes)

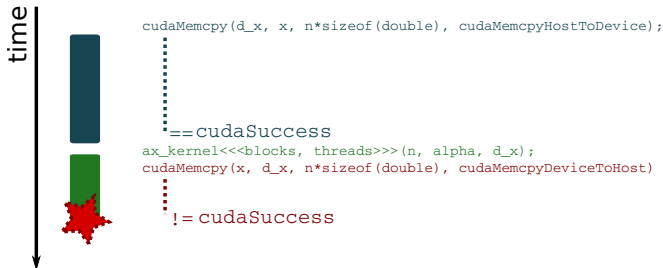
- ▶ Kernel launches and many other CUDA functions (\*Async) are non-blocking / **asynchronous**.
  - ▶ The kernel or the function call is simply placed into a stream.



# Error handling (some notes)

- It is possible that the returned error code is **related to one of the earlier kernels or function calls!**

device   host



# Error handling (some notes)

- ▶ This can happen even when the outcome of the kernel launch is checked:



- ▶ Only errors that occurred during the kernel launch are returned by `cudaGetLastError()`.



# Hands-ons

- ▶ Materials: [https://git.cs.umu.se/mirkom/gpu\\_course/](https://git.cs.umu.se/mirkom/gpu_course/)
- ▶ Three hands-ons under `hands-ons/1.basics`:
  - `1.threads` Learn how to launch a kernel. Learn how to coordinate threads and thread blocks.
  - `2.errors` Learn how to detect and handle errors.
  - `3.memory` Learn how to allocate global memory and transfer data to/from a device.
- ▶ Solutions can be found under `solutions/1.basics`.