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

Why should I use GPUs?

Why should I use GPUs?

- When compared to CPUs, GPUs provides much higher
  - instruction throughput and
  - memory bandwidth.
- In particular, this computing power is delivered in a relatively small price and power envelope.

Why should I use GPUs? (flop rate, DP)

- Quad-core Intel Skylake CPU:

  $\sim$ 200 GFlops

- 14-core Intel Xeon Gold 6132 CPU:

  $\sim$ 1200 GFlops

- Nvidia Tesla V100 GPU:

  $\sim$ **7 000** GFlops

Why should I use GPUs? (memory bandwidth)

- Quad-core Intel Skylake CPU:

  $\sim$ 35 GB/s

- 14-core Intel Xeon Gold 6132 CPU:

  $\sim$ 100 GB/s

- Nvidia Tesla V100 GPU:

  $\sim$ **900** GB/s

Why should I use GPUs? (power usage)

- Quad-core Intel Skylake CPU:

  $\sim$ 90W

- 14-core Intel Xeon Gold 6132 CPU:

  $\sim$ 140 W

- Nvidia Tesla V100 GPU:

  $\sim$ **250** W

Why should I use GPUs? (price)

- Quad-core Intel Skylake CPU:

  $\sim$ \$400

- 14-core Intel Xeon Gold 6132 CPU:

  $\sim$ \$2 000

- Nvidia Tesla V100 GPU:

  $\sim$ **\$10 000**

Why should I use GPUs? (per Watt)

- Quad-core Intel Skylake CPU:

  $\sim$ 2 GFlops per Watt, $\sim$ 0.4 GB/s per Watt

- 14-core Intel Xeon Gold 6132 CPU:

  $\sim$ 9 GFlops per Watt, $\sim$ 0.7 GB/s per Watt

- Nvidia Tesla V100 GPU:

  $\sim$ **28** GFlops per Watt, $\sim$ **3.6** GB/s per Watt

Why should I use GPUs? (per dollar)

- Quad-core Intel Skylake CPU:

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

- 14-core Intel Xeon Gold 6132 CPU:

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

- Nvidia Tesla V100 GPU:

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

Why should I use GPUs? (reduced precision)

- Quad-core Intel Skylake CPU:

  $\sim$ 400 GFlops (single precision)

- 14-core Intel Xeon Gold 6132 CPU:

  $\sim$ 2400 GFlops (single precision)

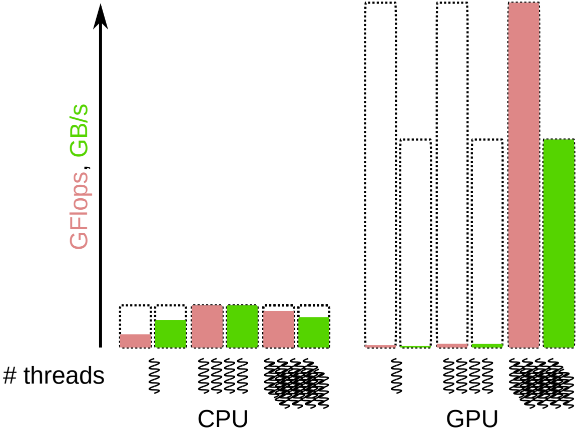
- Nvidia Tesla V100 GPU:

  $\sim$ 14 000 GFlops (single precision)
  $\sim$ **112 000** GFlops (half precision, tensor cores)

What is the catch?

What is the catch?

- GPUs are highly parallel. Peak performance requires hundreds or thousands of threads.
- The algorithms and codes must be highly parallel.



What is the catch?

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

CUDA basics

Lets go through some CUDA basics...

What is CUDA?

- **C**ompute **U**nified **D**evice **A**rchitecture (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?

- CUDA comes with it's own compiler, `nvcc`.
  - GPU-specific code is compiled to **PTX** (Parallel Thread Execution).
  - The PTX "assembly" code is translated to binary code by the graphics 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)
- 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
```

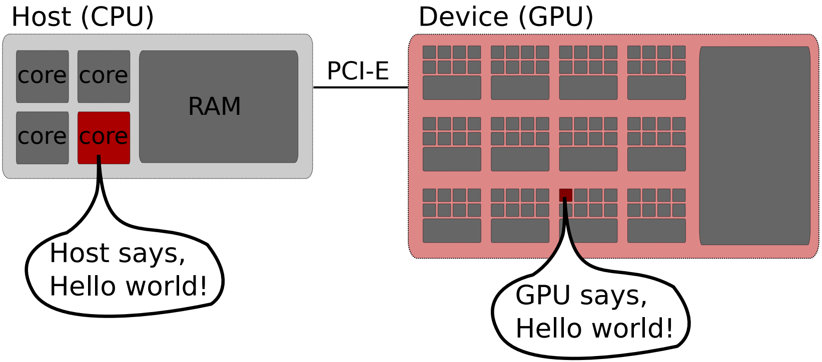
- Compile the source code with `nvcc`:

```
$ 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** CPU cores + RAM memory
- **Device** CUDA cores + VRAM
- **PCI-E** Fast interconnect between the host and the device

Hello world (kernels)

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

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

- The 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 **launches** 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 kernel is executed once.
- 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");

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

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

    return EXIT_SUCCESS;
}
```

AX example (scalar-vector multiplication)

- Lets try something more complicated:

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

- 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)

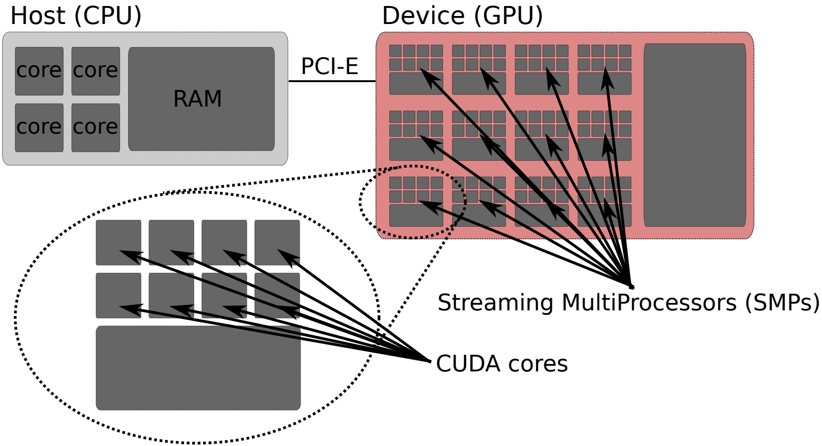
- A matching kernel is still relatively simple:

```
__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)

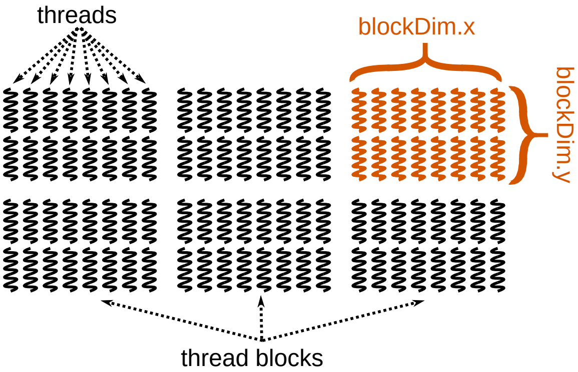


AX example (CUDA cores and SMPs)

- 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 which thread does what?
- 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.

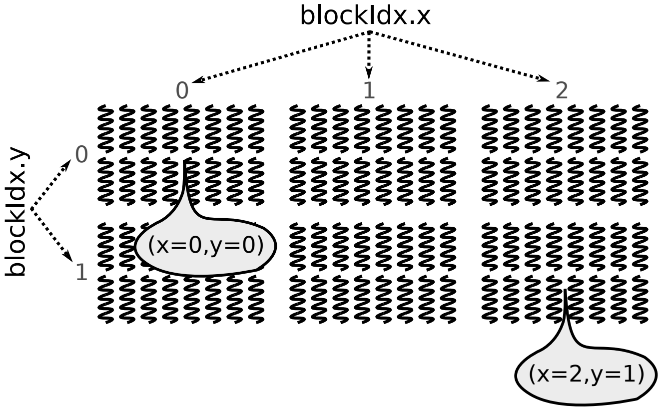
AX example (threads and thread blocks)

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



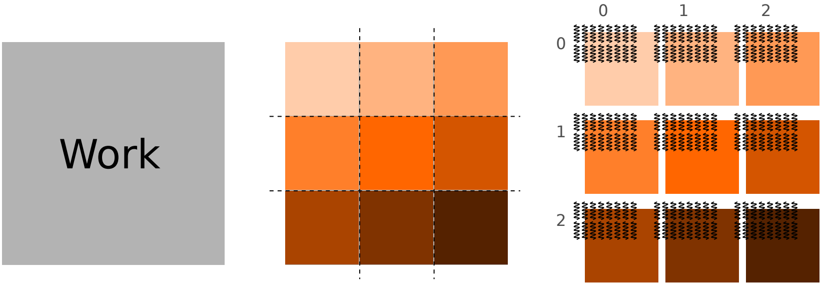
AX example (threads and thread blocks)

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



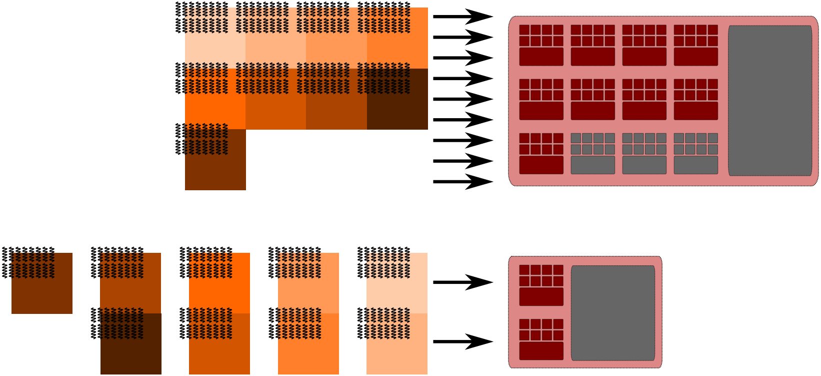
AX example (threads and thread blocks)

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



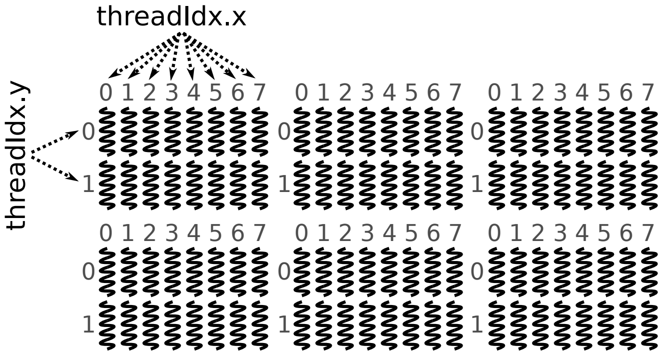
AX example (threads and thread blocks)

- 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:

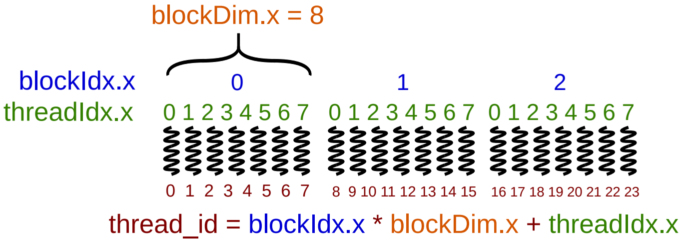


AX example (global indexing)

- 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];
}
```

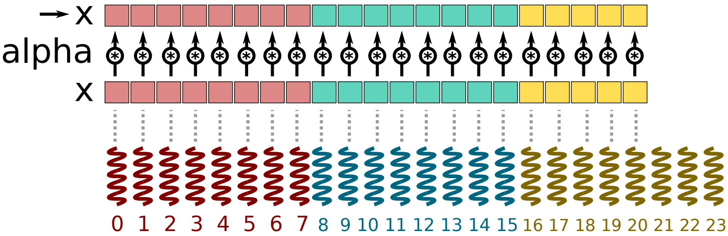


AX example (global indexing)

- 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)

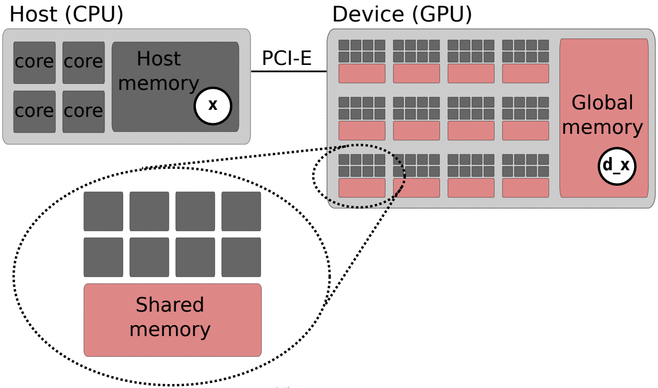
- The thread blocks should be reasonably large.
  - Usually 32 threads is too small for practical use.
  - At the moment, the upper limit is 1024 thread.
  - My personal recommendation is to start from 256 threads, and tune the number if necessary.
  - Multiples of 32 are preferred (warps).
- The number of thread block 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*Dx + z*Dx*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));
```

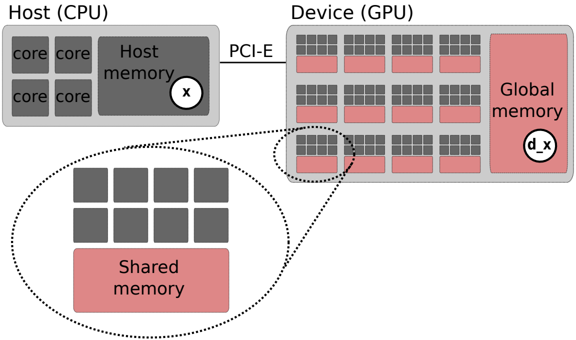


AX example (memory spaces)

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 a same thread block**.

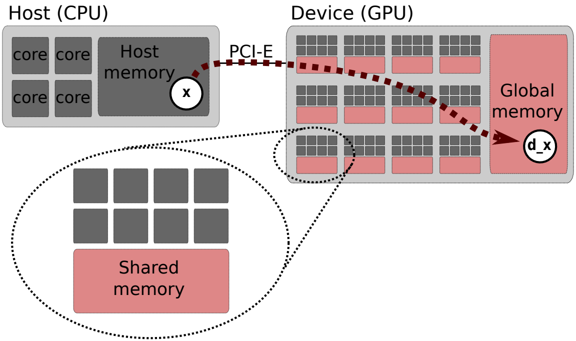


AX example (memory transfer to device)

- The host initializes a data transfer from the host to the device:

```
...
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 launches 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+blockDim.x-1)/blockDim.x` is simply a convenient way of making sure that

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

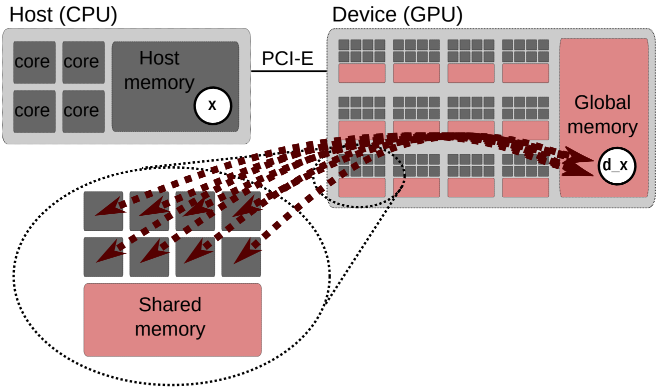
- I personally prefer the following approach:

```
// 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)

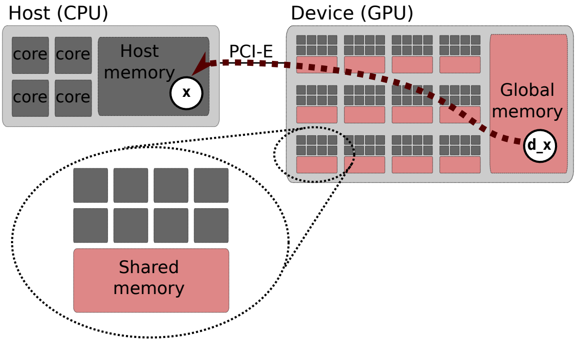
- The kernel can now access the data from the global memory:



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 CUDA API calls **must** the 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
```

More about memory (page-locked memory)

- Host and device share the same memory address space.
  - However, a device cannot always access the host memory and vice versa.
- A device can access page-locked host memory:

```
__host__ cudaError_t cudaMallocHost ( void** ptr, size_t size )
```

- Page-locked memory can be accessed with much higher bandwidth than pageable memory obtained with functions such as `malloc()`.
- However, page-locked memory is still much slower that the device-side memory.

More about memory (managed memory)

- Modern GPUs can manage the memory automatically:

```
// allocate managed memory
double *x;
cudaMallocManaged(&x, n*sizeof(double));

// initialize memory
for (int i = 0; i < n; i++)
    x[i] = 2.0 * rand()/RAND_MAX - 1.0;

// launch the kernel directly
dim3 threads = 256;
dim3 blocks = (n+threads.x)/threads.x;
ax_kernel<<<blocks, threads>>>(n, alpha, x);
```

- The array `x` is allocated when it is accessed.
- Access from either side causes a page fault and triggers a data transfer.
- Make things simpler but has some limitations...

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:

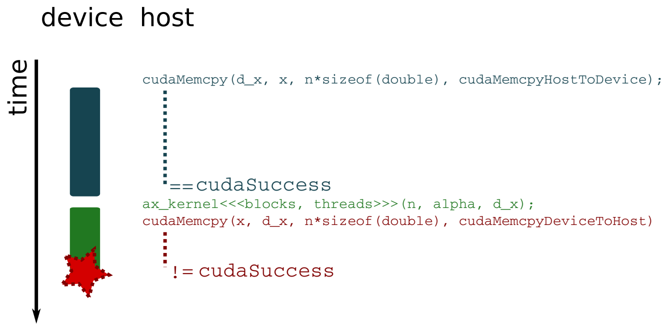
```
__host__ __device__ const char* cudaGetErrorString(cudaError_t error)
```

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.

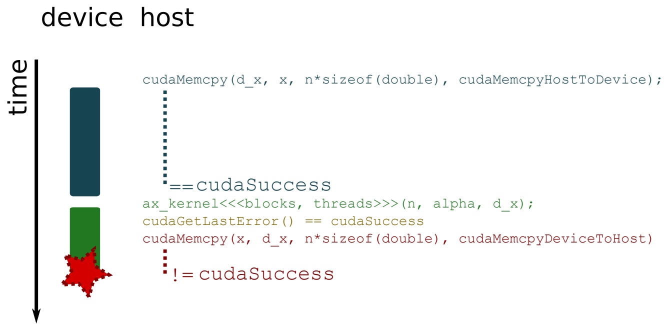
Error handling (some notes)

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



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 reported by `cudaGetLastError()`.

Hands-ons

- Four 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 device-side memory and transfer data to/from a device memory.

  4.managed
  : Learn how to use managed device memory. Learn how to use CUDA Basic Linear Algebra Subroutines.

- Solutions can be found under `solutions/1.basics`.