Introduction to GPU programming: When and how to use GPU-acceleration?

GPU hardware and CUDA basics

Mirko Myllykoski
mirkom@cs.umu.se

Department of Computing Science / HPC2N
Umeå University

5 November 2019

SNIC UMEÅ UNIVERSITY HPC2N

Example codes

```
$ cd ~/pfs
$ git clone https://git.cs.umu.se/mirkom/gpu_course.git
$ cd gpu_course
$ ml purge
$ ml intelcuda/2019a buildenv
$ make
```

Lets go through some CUDA basics...

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 intelcuda/2019a buildenv
```

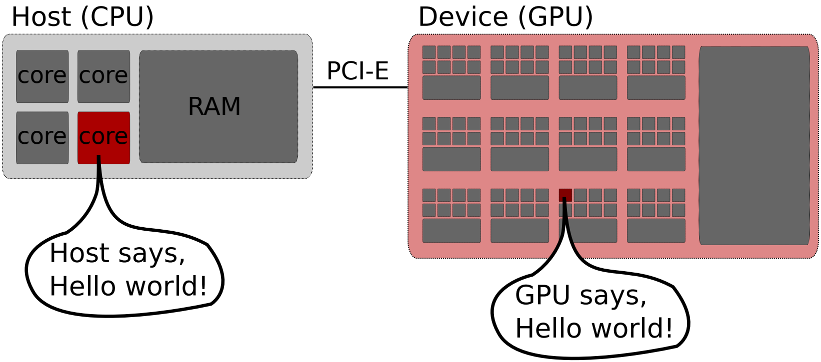
- Compile the source code with `nvcc`:

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

- Queue a job:

```
$ srun -A SNIC2019-5-142 --gres=gpu:v100:1,gpuexcl \
--time=00:05:00 --ntasks=1 ./hello.cuda
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**.

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");
}
```

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 host launches the `say_hello` kernel as follows:

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

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 host launches the `say_hello` kernel as follows:

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

- This places the kernel call into a queue known as **stream**.
  - The `cudaDeviceSynchronize();` call causes the host to wait until the kernel has finished.

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 multiplication)

- Lets try something more complicated:

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

$$\boldsymbol{x} \leftarrow \alpha \boldsymbol{x}$$

AX example (scalar multiplication)

- Lets try something more complicated:

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

$$\boldsymbol{x} \leftarrow \alpha \boldsymbol{x}$$

- The kernel is still relatively simple:

```
__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 (scalar multiplication)

- Lets try something more complicated:

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

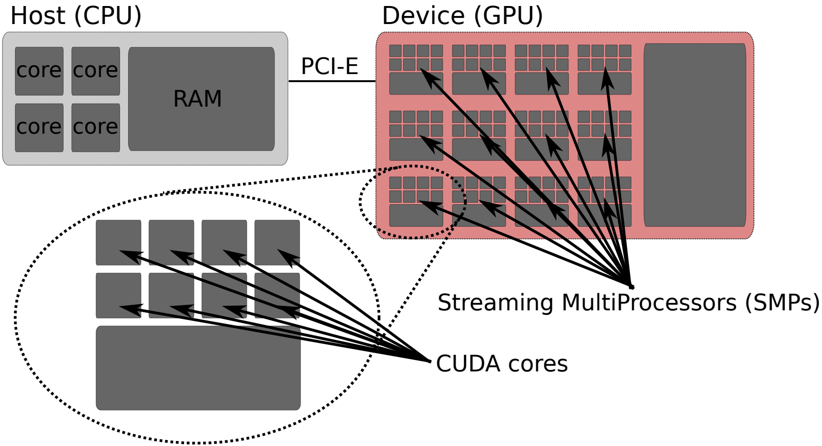
- The kernel is still relatively simple:

```
__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];
}
```

- What are `blockIdx.x`, `blockDim.x`and `threadIdx.x`?

AX example (CUDA cores and SMPs)



AX example (CUDA cores and SMPs)

- Each **Streaming MultiProcessor** (SMP) consist from several **CUDA cores**.

AX example (CUDA cores and SMPs)

- Each **Streaming MultiProcessor** (SMP) consist from several **CUDA cores**.
- Each CUDA core **can execute several threads simultaneously**.
  - The scheduler select the next instruction among a pool of active threads.

AX example (CUDA cores and SMPs)

- Each **Streaming MultiProcessor** (SMP) consist from 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.

AX example (CUDA cores and SMPs)

- Each **Streaming MultiProcessor** (SMP) consist from 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?

AX example (CUDA cores and SMPs)

- Each **Streaming MultiProcessor** (SMP) consist from 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?

AX example (CUDA cores and SMPs)

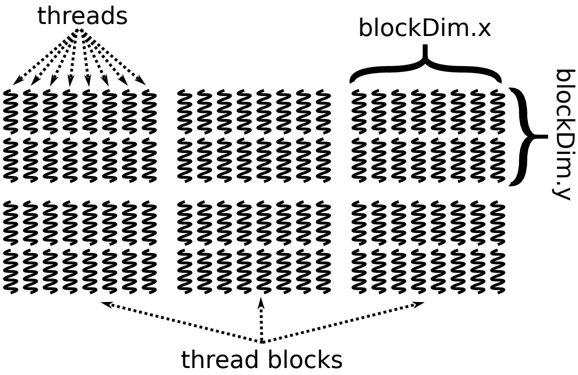
- Each **Streaming MultiProcessor** (SMP) consist from several **CUDA cores**.
- Each CUDA core **can execute several threads simultaneously**.
    - The scheduler select the next instruction among a pool of active threads.
- Thus, the total number of threads can be in the millions.
- How do we decide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

AX example (CUDA cores and SMPs)

- Each **Streaming MultiProcessor** (SMP) consist from 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.

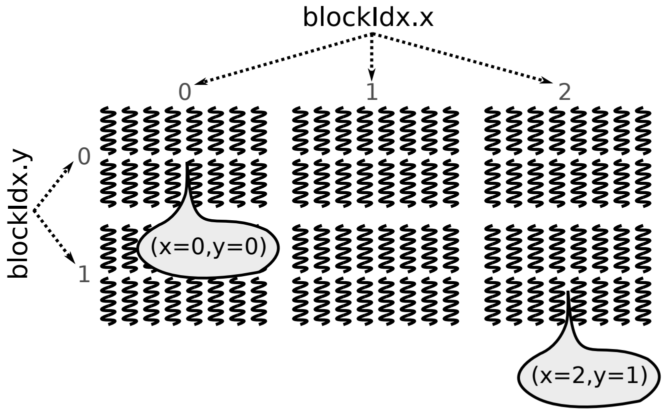
AX example (threads and thread blocks)

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



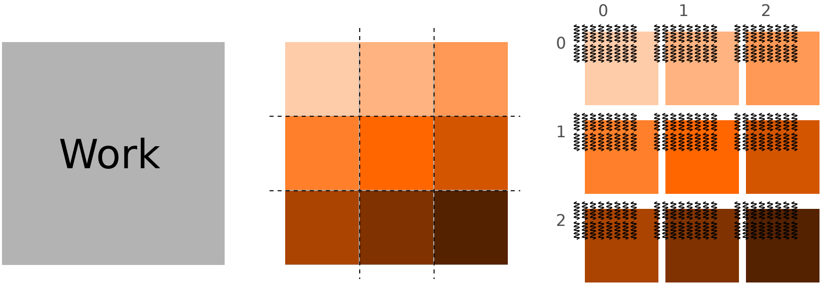
AX example (threads and thread blocks)

- Each tread block gets an **index number**:



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.

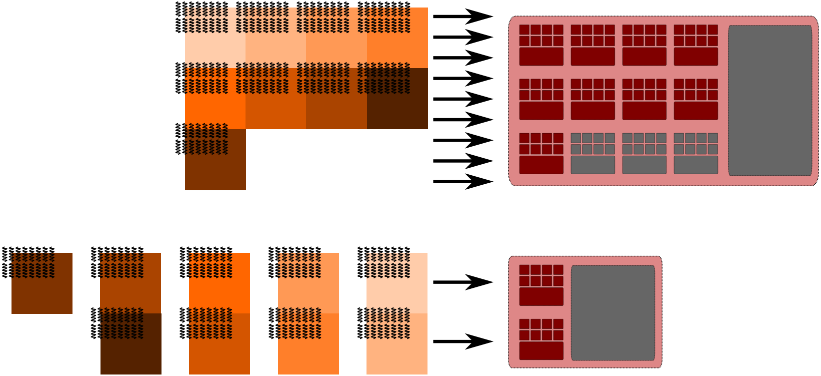


AX example (threads and thread blocks)

- The CUDA runtime is responsible from scheduling the thread blocks.

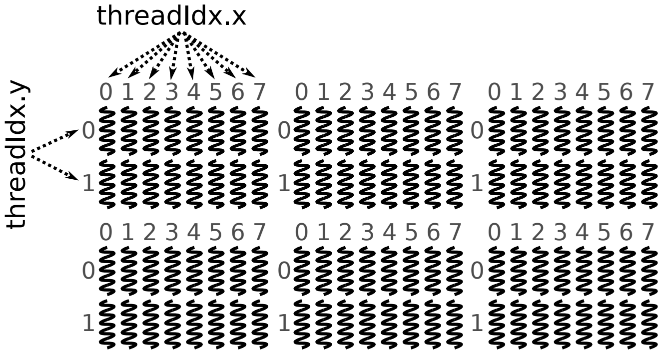
AX example (threads and thread blocks)

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



AX example (threads and thread blocks)

- Each tread gets a (local) **index number**:

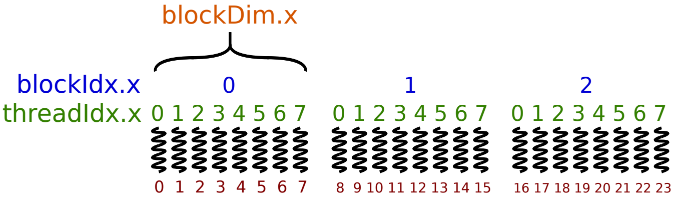


AX example (threads and thread blocks)

- An unique global 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;

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



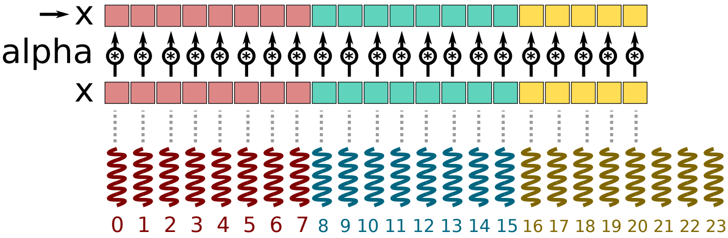
thread_id = blockIdx.x * blockDim.x + threadIdx.x

AX example (threads and thread blocks)

- An unique global 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;

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

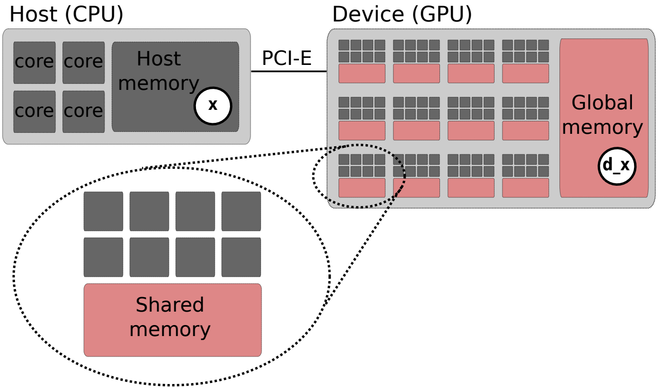


AX example (memory spaces)

- 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));
```

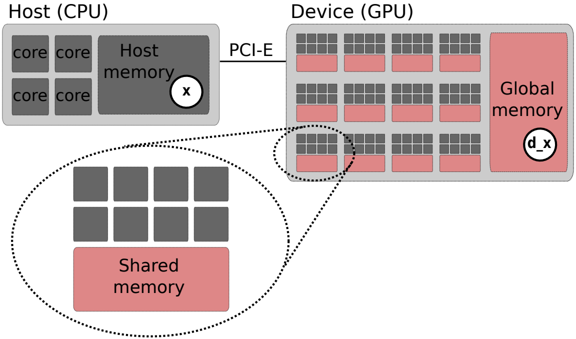


AX example (memory spaces)

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**.



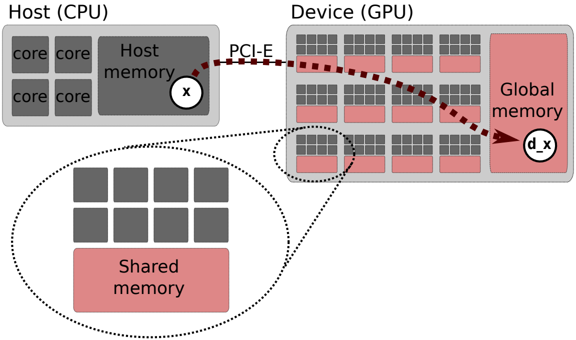
AX example (memory transfers)

- The host initializes a data transfer from the host memory to the global 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));

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



AX example (kernel launch)

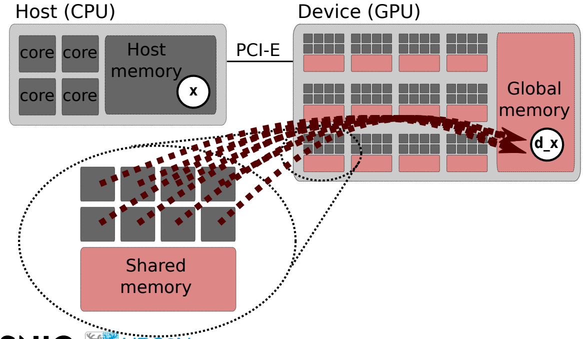
- The host launches the `ax_kernel` kernel:

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

// number of threads per thread block (blockDim.x)
dim3 threads = 256;

// number of thread blocks (gridDim.x)
dim3 blocks = (n+threads.x)/threads.x;

// launch the kernel
ax_kernel<<<blocks, threads>>>(n, alpha, d_x);
```



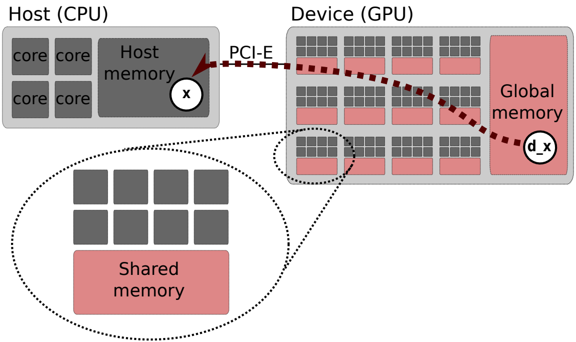
AX example (memory transfers)

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

```
...

dim3 threads = 256;
dim3 blocks = (n+threads.x)/threads.x;
ax_kernel<<<blocks, threads>>>(n, alpha, d_x);

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



AX example (compile and run)

- Load the correct toolchain:

```
$ ml intelcuda/2019a buildenv
```

- Compile the source code with `nvcc`:

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

- Queue a job:

```
$ srun -A SNIC2019-5-142 --gres=gpu:v100:1,gpuexcl \
--time=00:05:00 --ntasks=1 ./ax.cuda
The result was correct.
```

Error handling (queries)

- Most CUDA functions return an error code of the type `cudaError_t`.

Error handling (queries)

- Most CUDA functions return an error code of the type `cudaError_t`.
- A successful function call returns `cudaSuccess`.

Error handling (queries)

- Most CUDA functions return an error code of the type `cudaError_t`.
- A successful function call returns `cudaSuccess`.
- The error code can be turned into a string:

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

Error handling (queries)

- Most CUDA functions return an error code of the type `cudaError_t`.
- A successful function call returns `cudaSuccess`.
- The error code can be turned into a string:

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

- The error code can be turned into a longer description:

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

Error handling (queries)

- Most CUDA functions return an error code of the type `cudaError_t`.
- A successful function call returns `cudaSuccess`.
- The error code can be turned into a string:

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

- The error code can be turned into a longer description:

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

- The previous error code can be checked and **resetted** with:

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

Error handling (queries)

- Most CUDA functions return an error code of the type `cudaError_t`.
- A successful function call returns `cudaSuccess`.
- The error code can be turned into a string:

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

- The error code can be turned into a longer description:

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

- The previous error code can be checked and **resetted** with:

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

- The previous error code can be checked without resetting:

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

Error handling (some notes)

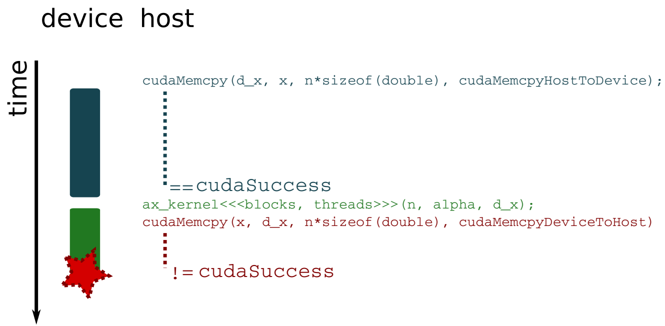
- Kernel launches and many other CUDA functions (`*Async`) are **asynchronous**.
  - The kernel or function call is simply placed into a queue (stream).

Error handling (some notes)

- Kernel launches and many other CUDA functions (`*Async`) are **asynchronous**.
  - The kernel or function call is simply placed into a queue (stream).
- It is possible that the returned error code is **related to one of the earlier kernels or function calls!**

Error handling (some notes)

- Kernel launches and many other CUDA functions (`*Async`) are **asynchronous**.
  - The kernel or function call is simply placed into a queue (stream).
- It is possible that the returned error code is **related to one of the earlier kernels or function calls!**



AXPY hands-on (scalar vector update)

- You try something more complicated:

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

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$$

AXPY hands-on (scalar vector update)

- You try something more complicated:

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

- Modify `ax.cu` such that it computes $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$.

AXPY hands-on (scalar vector update)

- You try something more complicated:

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

- Modify `ax.cu` such that it computes $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$.
  - Allocate and initialize $\boldsymbol{y}$.

AXPY hands-on (scalar vector update)

- You try something more complicated:

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

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$$

- Modify `ax.cu` such that it computes $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$.
  - Allocate and initialize $\boldsymbol{y}$.
  - Transfer $\boldsymbol{y}$ to global memory.

AXPY hands-on (scalar vector update)

- You try something more complicated:

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

- Modify `ax.cu` such that it computes $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$.
  - Allocate and initialize $\boldsymbol{y}$.
  - Transfer $\boldsymbol{y}$ to global memory.
  - Write a AXPY kernel.

AXPY hands-on (scalar vector update)

- You try something more complicated:

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

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$$

- Modify `ax.cu` such that it computes $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$.
  - Allocate and initialize $\boldsymbol{y}$.
  - Transfer $\boldsymbol{y}$ to global memory.
  - Write a AXPY kernel.
  - Transfer the updated $\boldsymbol{y}$ from global memory.

AXPY hands-on (scalar vector update)

- You try something more complicated:

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

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$$

- Modify `ax.cu` such that it computes $\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$.
  - Allocate and initialize $\boldsymbol{y}$.
  - Transfer $\boldsymbol{y}$ to global memory.
  - Write a AXPY kernel.
  - Transfer the updated $\boldsymbol{y}$ from global memory.
  - Validate the updated $\boldsymbol{y}$.

Streams (create, destroy and syncronize)

- The kernels and asynchronous function calls (`*Async`) are placed into a **stream**.

Streams (create, destroy and syncronize)

- The kernels and asynchronous function calls (`*Async`) are placed into a **stream**.
- A stream is created with

```
__host__ cudaError_t cudaStreamCreate(cudaStream_t* pStream)
```

Streams (create, destroy and syncronize)

- The kernels and asynchronous function calls (`*Async`) are placed into a **stream**.
- A stream is created with

```
__host__ cudaError_t cudaStreamCreate(cudaStream_t* pStream)
```

- A stream is destroyed with

```
__host__ __device__ cudaError_t cudaStreamDestroy(cudaStream_t stream)
```

Streams (create, destroy and syncronize)

- The kernels and asynchronous function calls (`*Async`) are placed into a **stream**.
- A stream is created with

```
__host__ cudaError_t cudaStreamCreate(cudaStream_t* pStream)
```

- A stream is destroyed with

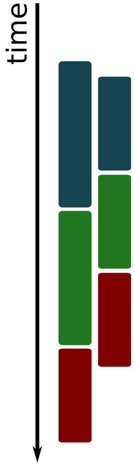
```
__host__ __device__ cudaError_t cudaStreamDestroy(cudaStream_t stream)
```

- A stream is synchronized with

```
__host__ cudaError_t cudaStreamSynchronize(cudaStream_t stream)
```

Streams (example)

device host




```
cudaStream_t p1, p2;
cudaStreamCreate(p1);
cudaStreamCreate(p2);
cudaMemcpyAsync(d_x1, x1, n*sizeof(double), cudaMemcpyHostToDevice, p1);
cudaMemcpyAsync(d_x2, x2, n*sizeof(double), cudaMemcpyHostToDevice, p2);
ax_kernel<<<blocks, threads, 0, p1>>>(n, alpha, d_x1);
ax_kernel<<<blocks, threads, 0, p2>>>(n, alpha, d_x2);
cudaMemcpyAsync(x1, d_x1, n*sizeof(double), cudaMemcpyDeviceToHost, p1);
cudaMemcpyAsync(x2, d_x2, n*sizeof(double), cudaMemcpyDeviceToHost, p2);
cudaStreamSynchronize(p1);
    ==cudaSuccess
cudaStreamSynchronize(p2);
    ==cudaSuccess
```

Unified Memory Programming

- 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, d_x);
```

Unified Memory Programming

- 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, d_x);
```

- The array `x` **resides in the host memory**.

Unified Memory Programming

- 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, d_x);
```

- The array `x` **resides in the host memory**.
- Access from the device side causes a page fault and triggers a data transfer.

Unified Memory Programming

- 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, d_x);
```

- The array `x` **resides in the host memory**.
- Access from the device side causes a page fault and triggers a data transfer.
- Make things simpler but has some limitations...