Introduction to High Performance Computing

\\

SDS406 – Fall semester, 2024 - 2025

\\

L05: Introduction to GPU programming, 1st November 2024

Outline

Introductory part

- Review of GPU architecture
- Review of GPU programming and CUDA
- Some details of the GPU nodes of our cluster

Outline

Introductory part

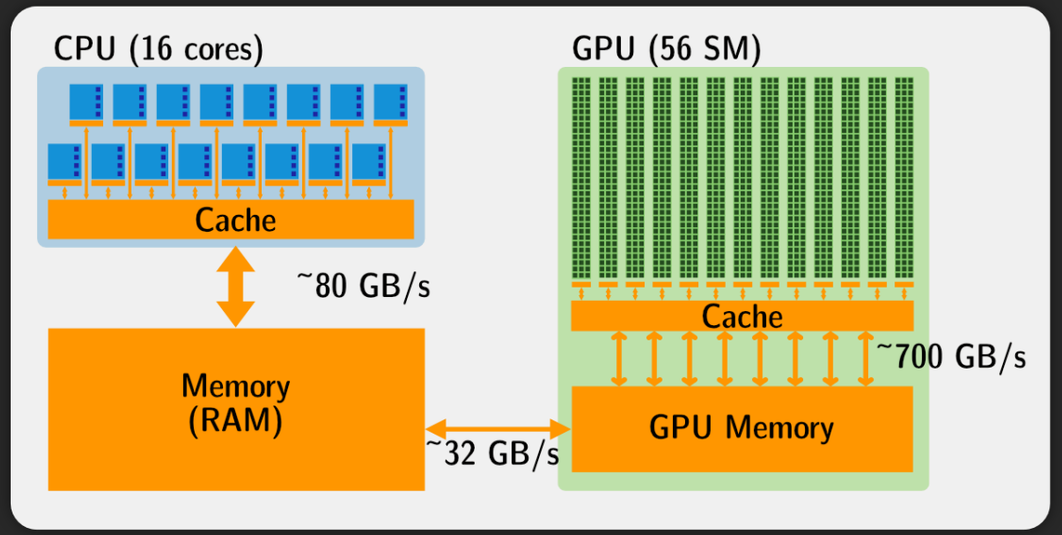
- Review of GPU architecture
- Review of GPU programming and CUDA
- Some details of the GPU nodes of our cluster

Practical examples on GPUs

Covering:

- GPU performance vs CPU performance —— this lecture
- Memory coalescing on GPUs —— this lecture
- Shared memory —— next week
- Details of GPU thread scheduling (warps) and why you should care —— next week

GPU architecture



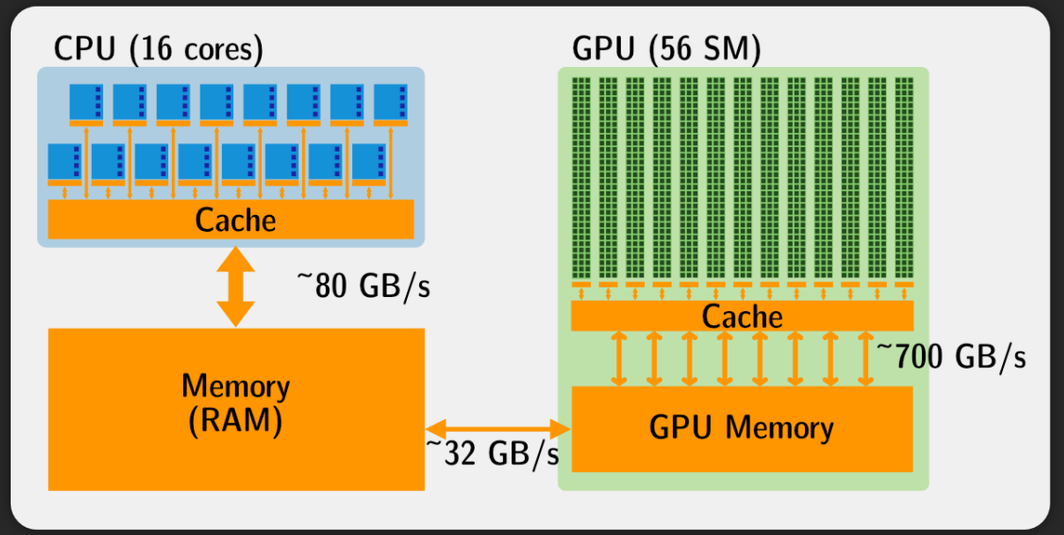
CPU

- Few heavy cores
- Large memory
- Moderate BW to memory
- Optimized for serial execution

GPU

- Many light "cores"
- Smaller memory
- High BW to memory
- Optimized for parallel execution

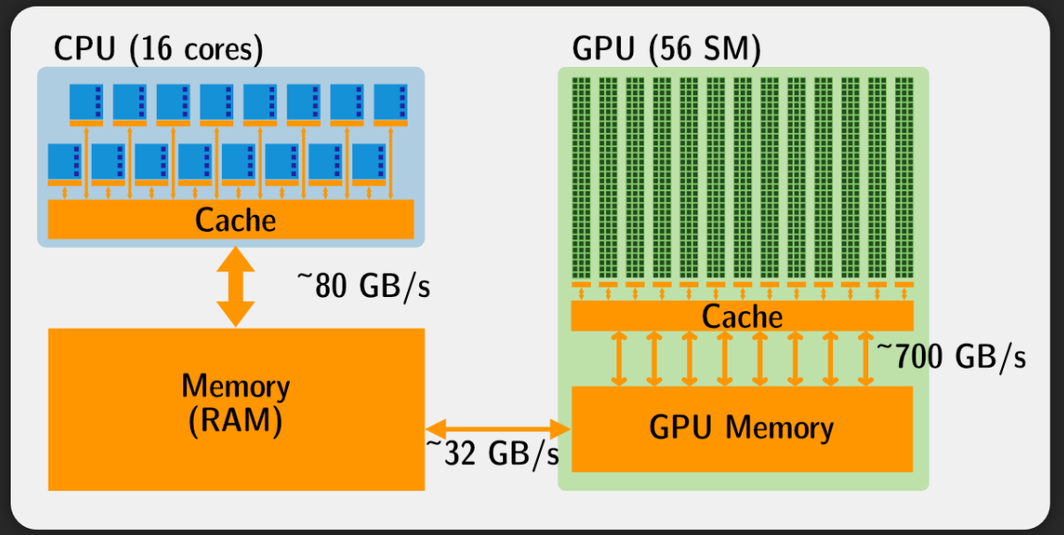
GPU programming model



Some numbers from our cluster's nodes: `cyc` nodes (NVIDIA definitions)

- NVIDIA P100 "Pascal" GPUs
- 56 Streaming Multiprocessors (SM) per GPU
- 32 FP64 or 64 FP32 "cores" per SM
- GPU memory: 16 GBytes
- Clock rate: ~1.5 GHz
- Cache (L2): 4 MBytes
- BW: 732 GB/s

GPU programming model

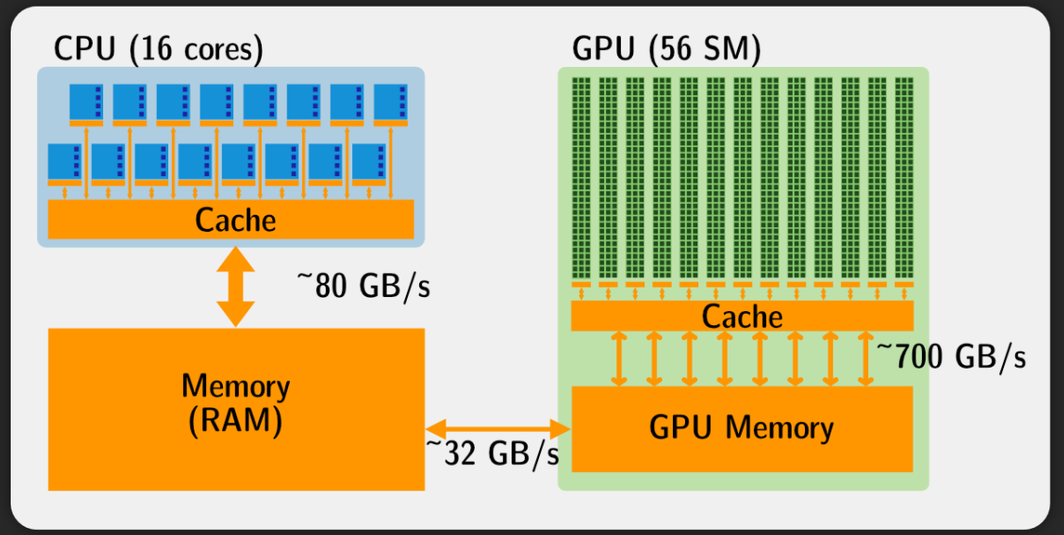


Some numbers from our cluster's nodes: `cyc` nodes (NVIDIA definitions)

- NVIDIA P100 "Pascal" GPUs
- 56 Streaming Multiprocessors (SM) per GPU
- 32 FP64 or 64 FP32 "cores" per SM
- GPU memory: 16 GBytes
- Clock rate: ~1.5 GHz
- Cache (L2): 4 MBytes
- BW: 732 GB/s

We will come back to these numbers during the practical

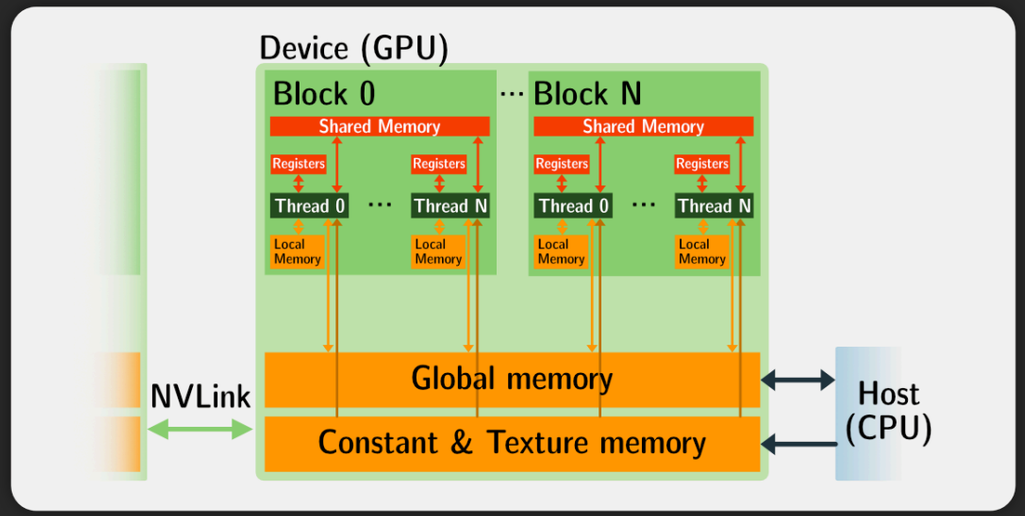
GPU programming model



"Offload" model of programming

- CPU starts program (runs `main()`)
- CPU copies data to GPU memory (over e.g. PCIe, ~16 GB/s)
- CPU dispatches "kernels" for execution on GPU
  - Kernels read/write to GPU memory (~732 GB/s)
  - Kernels run on GPU threads (thousands) which share *fast* memory [$O(10)$ times faster compared to GPU memory]
- Kernel completes; CPU copies data back from GPU (over e.g. PCIe, ~16 GB/s)

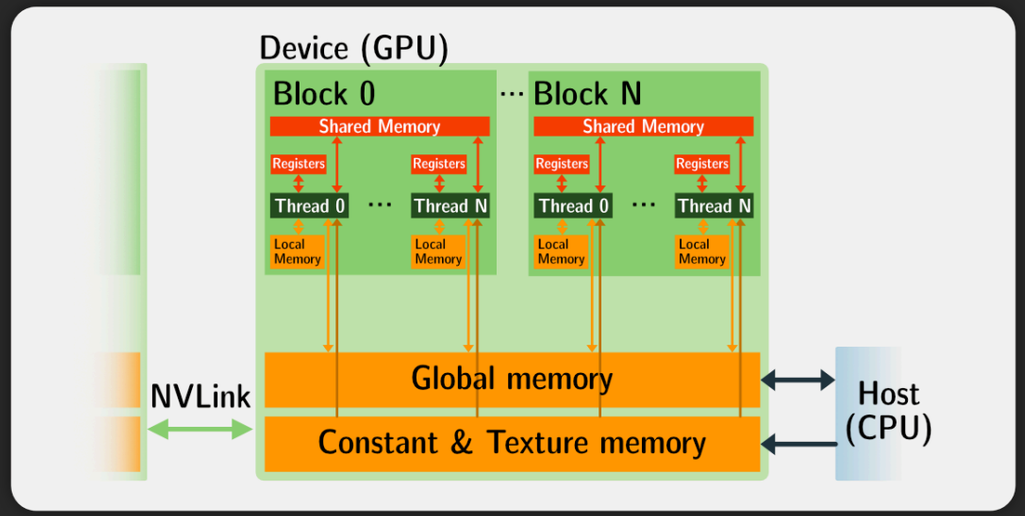
GPU programming model



GPU memory model (NVIDIA model)

- GPU threads: *slow* access to global, constant, and texture memory
- Each thread has *registers* (fast) and *local memory* (slow)
- Threads are grouped into *blocks*; Threads within the same block: *shared memory* (fast)

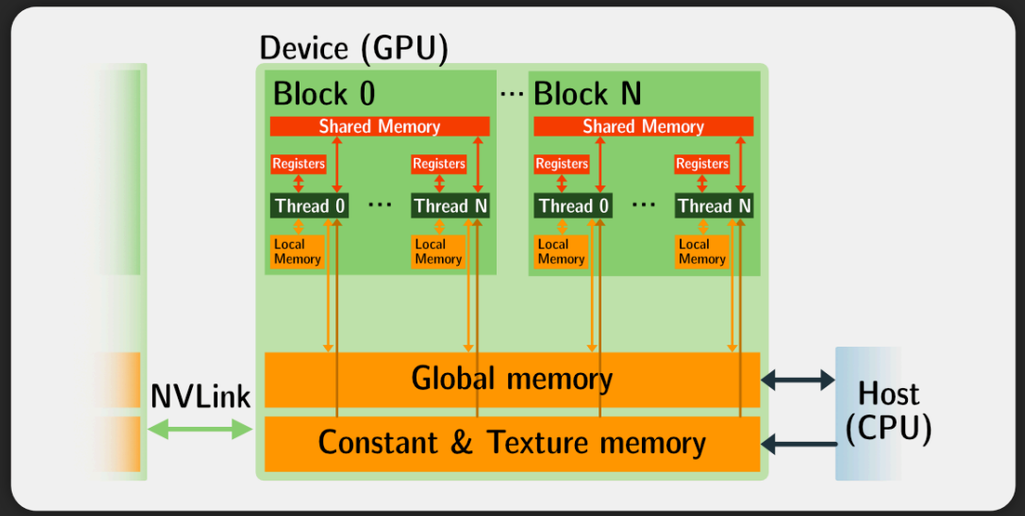
GPU programming model



GPU memory model (NVIDIA model); some numbers for context

- Threads per block: 1024 (max)
- Register memory (per block): 256 KB
- Shared memory (per block): 64 KB

GPU programming model



GPU memory model (NVIDIA model)

- Assumptions about execution order
  - Threads within the same block can be assumed to run concurrently
  - No assumption about the order by which blocks are executed

CUDA programming model

NVIDIA programming framework for NVIDIA GPUs

- Compute Unified Device Architecture
- C-like programming language for writing *CUDA Kernels*
  - Includes C/C++ and Fortran variants
  - Compiler for C/C++: `nvcc`
- Functions for transferring data to/from GPUs, starting kernels, etc.
- Some higher-level functionality also available (linear algebra, random number generations, etc.)
- Concepts generalizable to other accelerator programming frameworks (OpenCL, OpenACC, HiP, etc.)

CUDA programming basics

Nomenclature

- "Host" is the CPU
- "Device" is the GPU

Allocate memory on GPU

```
err = cudaMalloc(&d_ptr, size);
```

- Call from *host* (CPU)
- Allocate `size` bytes of memory on GPU and store the starting address in `d_ptr`
- `d_ptr` is a variable that holds an address to GPU memory i.e. a "device pointer"
- If `err` $\neq$ `cudaSuccess` then something went wrong

Free GPU memory

```
cudaFree(d_ptr);
```

CUDA programming basics

Nomenclature

- "Host" is the CPU
- "Device" is the GPU

Copy data to GPU

```
cudaMemcpy(d_ptr, ptr, size, cudaMemcpyHostToDevice);
```

- Call from *host* (CPU)
- Copy data on host pointed to by `ptr` to device at address pointed to by `d_ptr`
- Device memory should have been allocated using `cudaMalloc()` to obtain `d_ptr`

Copy data from GPU

```
cudaMemcpy(ptr, d_ptr, size, cudaMemcpyDeviceToHost);
```

- Call from *host* (CPU)
- Copy data on device pointed to by `d_ptr` to host at address pointed to by `ptr`
- Host memory should have been allocated using e.g. `malloc()` to obtain `ptr`

CUDA programming basics

Declare a CUDA kernel

Example:

```
__global__ void
func(int n, double a, double *x)
{
    ...
    return;
}
```

Call a CUDA kernel

- Call from host. Example:

  ```
  func<<<nblck, nthr>>>(n, a, x);
  ```

- `nthr`: number of threads per block; can be scalar or a `dim3` type
- `nblck`: number of blocks; can be scalar or a `dim3` type
- Example of `dim3` type:

  ```
  dim3 nthr(1024, 8, 8); /* No. of threads in (x, y, z) */
  ```

CUDA programming basics

Call a CUDA kernel

- Call from host. Example:

```
func<<<nblck, nthr>>>(n, a, x);
```

- `nthr`: number of threads per block; can be scalar or a `dim3` type
- `nblck`: number of blocks; can be scalar or a `dim3` type
- Example of `dim3` type:

```
dim3 nthr(1024, 8, 8); /* No. of threads in (x, y, z) */
```

Thread coordinates within kernel

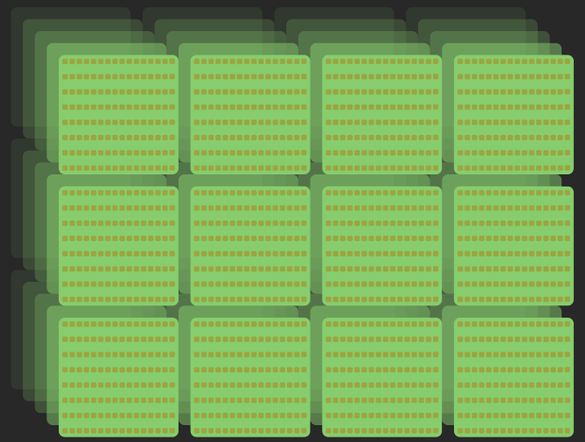
Example:

```
__global__ void
func(int n, double a, double *x)
{
    int idx = threadIdx.x + blockIdx.x*blockDim.x;
    ...
    return;
}
```

CUDA programming basics

Threads, blocks, grids

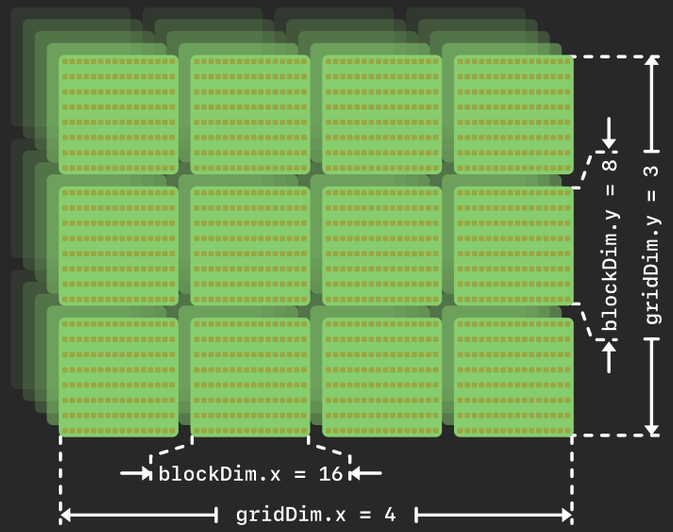
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

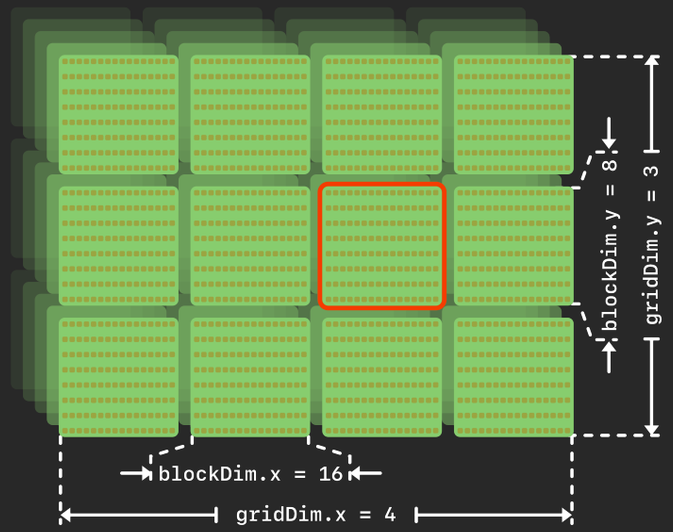
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

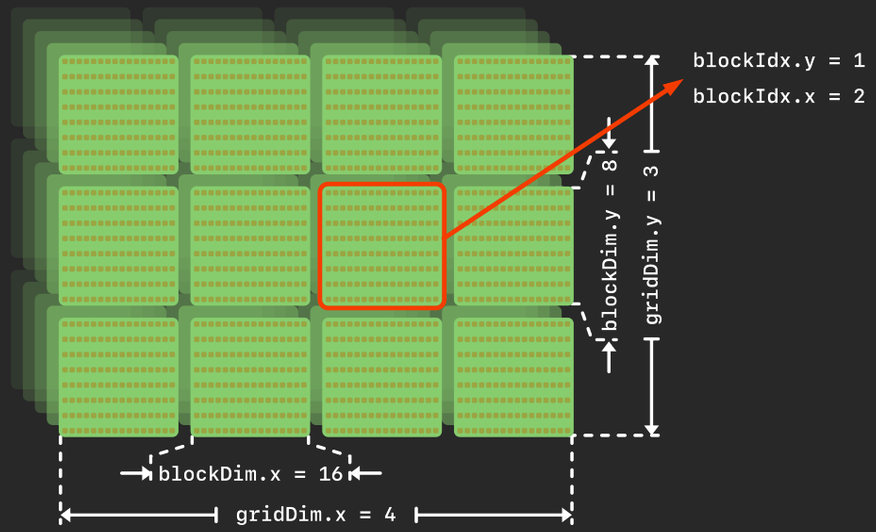
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

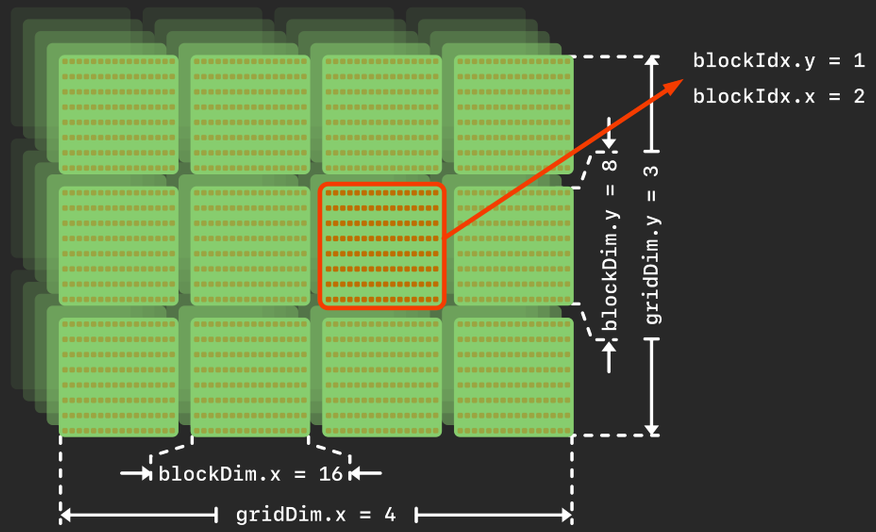
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

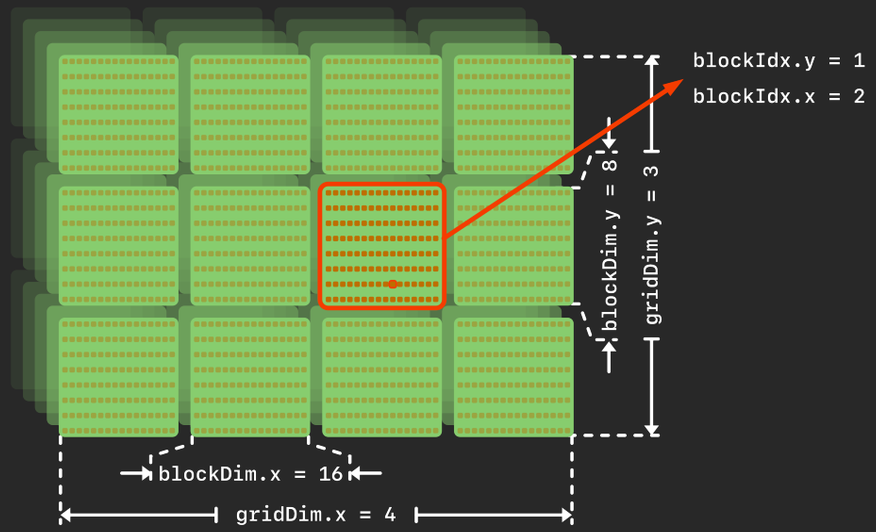
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

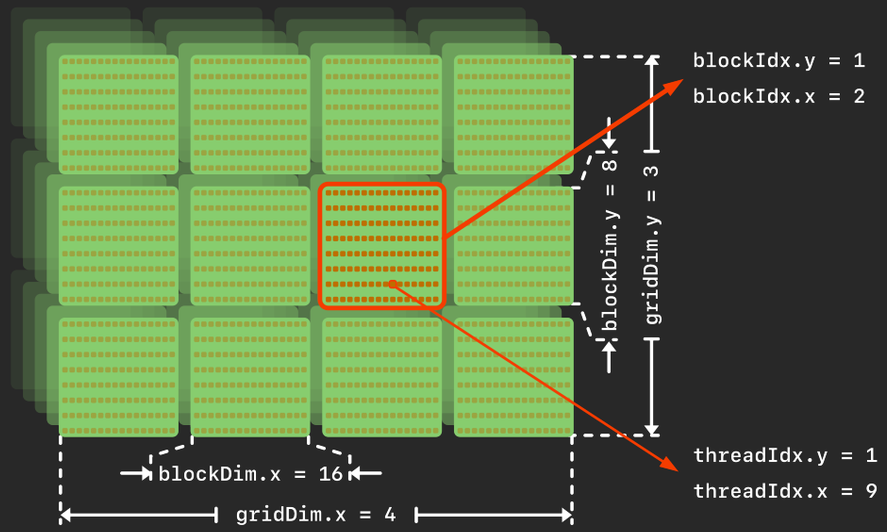
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

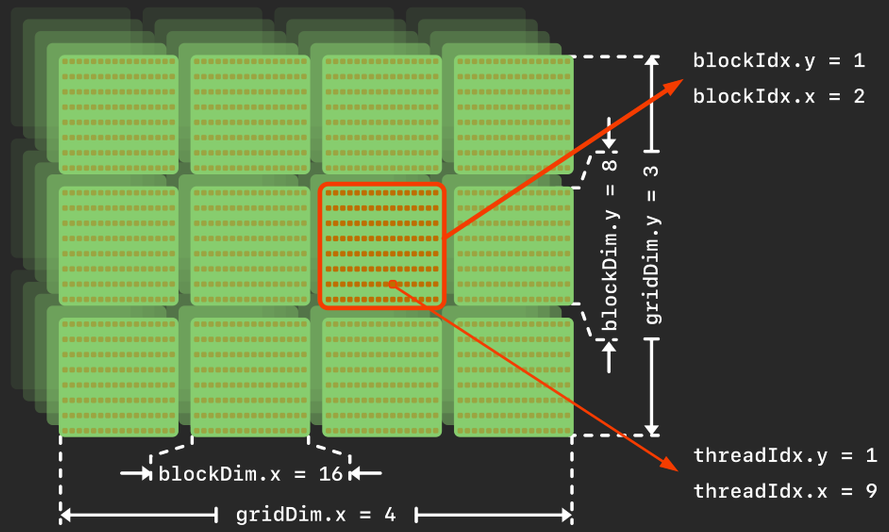
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



Variables available within kernel

- `threadIdx.{x,y,z}`
- `blockIdx.{x,y,z}`
- `blockDim.{x,y,z}`
- `gridDim.{x,y,z}`

CUDA Example

Exercise: port a simple code to GPU and investigate performance

Exercise directory: `/onyx/data/sds406f24/l05/ex01`

- `axpy.cu` implements this course's favorite BLAS operation, "axpy":

$$y_i \leftarrow a \cdot x_i + y_i, \quad i = 0, \ldots, n-1$$

  with a scalar and y and x vectors of length n.

CUDA Example

Exercise: port a simple code to GPU and investigate performance

Exercise directory: `/onyx/data/sds406f24/l05/ex01`

- `axpy.cu` implements this course's favorite BLAS operation, "axpy":

$$y_i \leftarrow a \cdot x_i + y_i, \ \ i = 0, \ldots, n-1$$

  with a scalar and y and x vectors of length n.

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 8 cores (per socket) of our educational nodes

CUDA Example

Exercise: port a simple code to GPU and investigate performance

Exercise directory: `/onyx/data/sds406f24/l05/ex01`

- `axpy.cu` implements this course's favorite BLAS operation, "axpy":

  $$y_i \leftarrow a \cdot x_i + y_i, \ \ i = 0, \ldots, n-1$$

  with a scalar and y and x vectors of length n.

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 8 cores (per socket) of our educational nodes
- We will proceed step-by-step, to port this simple application to the GPU using CUDA

CUDA Example

Exercise: port a simple code to GPU and investigate performance

Exercise directory: `/onyx/data/sds406f24/l05/ex01`

- `axpy.cu` implements this course's favorite BLAS operation, "axpy":

$$y_i \leftarrow a \cdot x_i + y_i, \ \ i = 0, \ldots, n-1$$

  with a scalar and y and x vectors of length n.

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 8 cores (per socket) of our educational nodes
- We will proceed step-by-step, to port this simple application to the GPU using CUDA

This will cover:

- Allocation of memory on the GPU;

CUDA Example

Exercise: port a simple code to GPU and investigate performance

Exercise directory: `/onyx/data/sds406f24/l05/ex01`

- `axpy.cu` implements this course's favorite BLAS operation, "axpy":

$$y_i \leftarrow a \cdot x_i + y_i, \ \ i = 0, \ldots, n-1$$

  with a scalar and y and x vectors of length n.

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 8 cores (per socket) of our educational nodes
- We will proceed step-by-step, to port this simple application to the GPU using CUDA

This will cover:

- Allocation of memory on the GPU;
- Transferring memory to/from GPU;

CUDA Example

Exercise: port a simple code to GPU and investigate performance

Exercise directory: `/onyx/data/sds406f24/l05/ex01`

- `axpy.cu` implements this course's favorite BLAS operation, "axpy":

$$y_i \leftarrow a \cdot x_i + y_i, \ \ i = 0, \ldots, n-1$$

  with a scalar and y and x vectors of length n.

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 8 cores (per socket) of our educational nodes
- We will proceed step-by-step, to port this simple application to the GPU using CUDA

This will cover:

- Allocation of memory on the GPU;
- Transferring memory to/from GPU;
- Invoking kernels;

CUDA Example

Exercise: port a simple code to GPU and investigate performance

Exercise directory: `/onyx/data/sds406f24/l05/ex01`

- `axpy.cu` implements this course's favorite BLAS operation, "axpy":

$$y_i \leftarrow a \cdot x_i + y_i, \ \ i = 0, \ldots, n-1$$

  with a scalar and y and x vectors of length n.

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 8 cores (per socket) of our educational nodes
- We will proceed step-by-step, to port this simple application to the GPU using CUDA

This will cover:

- Allocation of memory on the GPU;
- Transferring memory to/from GPU;
- Invoking kernels;
- Placement of threads and memory access

CUDA Example

File: `ex01/axpy.cu`

- Contains the C program we will begin with: `axpy.cu`
- Even though the file extension is `.cu`, the program contains no CUDA. Only OpenMP
- Allocates four arrays: `x0[n]`, `x1[n]`, `y0[n]`, and `y1[n]`, with `n` read from the command line
- `x0` and `y0` are initialized to random numbers
- `x1` and `y1` are initialized to `x0` and `y0` respectively
- The program:
  - performs `y0[:] = a*x0[:] + y0[:]` in the first part marked with `A:`
  - performs `y1[:] = a*x1[:] + y1[:]` in the second part marked with `B:`
  - reports the timing for part `A` and for `B`
  - reports the difference between `y0` and `y1`

CUDA Example

File: `ex01/axpy.cu`

- Contains the C program we will begin with: `axpy.cu`
- Even though the file extension is `.cu`, the program contains no CUDA. Only OpenMP
- Allocates four arrays: `x0[n]`, `x1[n]`, `y0[n]`, and `y1[n]`, with `n` read from the command line
- `x0` and `y0` are initialized to random numbers
- `x1` and `y1` are initialized to `x0` and `y0` respectively
- The program:
  - performs `y0[:] = a*x0[:] + y0[:]` in the first part marked with `A:`
  - performs `y1[:] = a*x1[:] + y1[:]` in the second part marked with `B:`
  - reports the timing for part `A` and for `B`
  - reports the difference between `y0` and `y1`

Take some time to inspect `axpy.cu` before we compile and run

CUDA Example

- Copy the exercise from this week's lesson directory:

```
[ikoutsou@front02 l05]$ cp -r /onyx/data/sds406f24/l05/ex01 .
[ikoutsou@front02 l05]$ cd ex01/
[ikoutsou@front02 ex01]$ ls -1
axpy.cu
```

CUDA Example

- Copy the exercise from this week's lesson directory:

```
[ikoutsou@front02 l05]$ cp -r /onyx/data/sds406f24/l05/ex01 .
[ikoutsou@front02 l05]$ cd ex01/
[ikoutsou@front02 ex01]$ ls -1
axpy.cu
```

- Compile with `nvcc` including OpenMP:

```
[ikoutsou@front02 ex01]$ module load gompi/2023a
[ikoutsou@front02 ex01]$ module load CUDA
[ikoutsou@front02 ex01]$ nvcc -O3 -Xcompiler -fopenmp -o axpy axpy.cu
```

- `-Xcompiler -fopenmp`: tells `nvcc` to pass `-fopenmp` to the underlying C compiler (here `gcc`)

CUDA Example

- Copy the exercise from this week's lesson directory:

```
[ikoutsou@front02 l05]$ cp -r /onyx/data/sds406f24/l05/ex01 .
[ikoutsou@front02 l05]$ cd ex01/
[ikoutsou@front02 ex01]$ ls -1
axpy.cu
```

- Compile with `nvcc` including OpenMP:

```
[ikoutsou@front02 ex01]$ module load gompi/2023a
[ikoutsou@front02 ex01]$ module load CUDA
[ikoutsou@front02 ex01]$ nvcc -O3 -Xcompiler -fopenmp -o axpy axpy.cu
```

- `-Xcompiler -fopenmp`: tells `nvcc` to pass `-fopenmp` to the underlying C compiler (here `gcc`)
- Run on the CPUs of a GPU node
- Use `srun` to run interactively, e.g.:

```
[ikoutsou@front02 ex01]$ export OMP_PROC_BIND="close"
[ikoutsou@front02 ex01]$ export OMP_PLACES="cores"
[ikoutsou@front02 ex01]$ export OMP_NUM_THREADS=16
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0145 sec   P =   9.232 Gflop/s   B =  55.393 GB/s
 CPU: nthr =   16   t0 = 0.0142 sec   P =   9.437 Gflop/s   B =  56.620 GB/s
 Diff = 0.000000e+00
```

CUDA Example

- Copy the exercise from this week's lesson directory:

```
[ikoutsou@front02 l05]$ cp -r /onyx/data/sds406f24/l05/ex01 .
[ikoutsou@front02 l05]$ cd ex01/
[ikoutsou@front02 ex01]$ ls -1
axpy.cu
```

- Compile with `nvcc` including OpenMP:

```
[ikoutsou@front02 ex01]$ module load gompi/2023a
[ikoutsou@front02 ex01]$ module load CUDA
[ikoutsou@front02 ex01]$ nvcc -O3 -Xcompiler -fopenmp -o axpy axpy.cu
```

- `-Xcompiler -fopenmp`: tells `nvcc` to pass `-fopenmp` to the underlying C compiler (here `gcc`)
- Run on the CPUs of a GPU node
- Use `srun` to run interactively, e.g.:

```
[ikoutsou@front02 ex01]$ export OMP_PROC_BIND="close"
[ikoutsou@front02 ex01]$ export OMP_PLACES="cores"
[ikoutsou@front02 ex01]$ export OMP_NUM_THREADS=16
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0145 sec   P =   9.232 Gflop/s   B =  55.393 GB/s
 CPU: nthr =   16   t0 = 0.0142 sec   P =   9.437 Gflop/s   B =  56.620 GB/s
 Diff = 0.000000e+00
```

- Compare ~56 GB/s achieved vs ~80 GB/s peak memory bandwidth (single socket)

CUDA Example

Use a GPU to replace part `B` of the calculation

- Edits outside of `main()`:
    1. Add the `cuda_runtime.h` header file
    2. Add the GPU `axpy` kernel, naming it `gpu_axpy()`
    3. Add a function similar to `ualloc()` that allocates memory on the GPU and checks whether an error occured
- Edits within `main()`:
    1. Allocate arrays on GPU
    2. Copy `x1[:]` and `y1[:]` to GPU
    3. Call `gpu_axpy()`
    4. Copy `y1[:]` from GPU

CUDA Example

Edits outside of `main()` 1/3

- Add the `cuda_runtime.h` header file on line 5:

```
#include <cuda_runtime.h>
```

CUDA Example

Edits outside of `main()` 2/3

- Add the GPU `axpy` kernel, naming it `gpu_axpy()`, after the CPU `axpy`, around line 64:

```
/***
 * Do y ← a*x + y on the GPU
 ***/
__global__ void
gpu_axpy(int n, float a, float *x, float *y)
{
  for(int i=0; i<n; i++)
    y[i] = a*x[i] + y[i];

  return;
}
```

CUDA Example

Edits outside of `main()` 3/3

- At around line 30 add a function similar to `ualloc()` that allocates memory on the GPU and checks whether an error occurred

```
/***
 * Allocate memory on GPU; print error if not successful
 ***/
void *
gpu_alloc(size_t size)
{
  void *ptr;
  cudaError_t err = cudaMalloc(&ptr, size);
  if(err ≠ cudaSuccess) {
    fprintf(stderr, "cudaMalloc() returned %d; quitting...\n", err);
    exit(-2);
  }
  return ptr;
}
```

CUDA Example

Edits within `main()` 1/4

- Allocate arrays on GPU, within `B` part. Free arrays before closing `B` part:

```
/*
 * B: Run axpy(), return to y1, report performance
 */
  {
    /* Allocate GPU memory */
    float *d_x = (float *)gpu_alloc(n*sizeof(float));
    float *d_y = (float *)gpu_alloc(n*sizeof(float));
    ...
    cudaFree(d_x);
    cudaFree(d_y);
  }
```

CUDA Example

Edits within `main()` 2/4

- Copy `x1[:]` and `y1[:]` to GPU

```
cudaMemcpy(d_x, x1, sizeof(float)*n, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y1, sizeof(float)*n, cudaMemcpyHostToDevice);
```

CUDA Example

Edits within `main()` 3/4

- Call `gpu_axpy()`. For the moment use 1 thread and 1 block. Replace `axpy(n, a, x, y)` of part `B` with:

```
double t0 = stop_watch(0);
gpu_axpy<<<1, 1>>>(n, a, d_x, d_y);
t0 = stop_watch(t0);
```

Note we need to pass the *device pointers* since it is these pointers that point to the memory allocated on the GPU

CUDA Example

Edits within `main()` 4/4

- Copy `y1[:]` from GPU:

```
/* Copy y1 back from GPU */
cudaMemcpy(y1, d_y, sizeof(float)*n, cudaMemcpyDeviceToHost);
```

- Also change:

```
printf(" CPU: nthr = %4d    ... );
```

  to:

```
printf(" GPU:              ... );
```

  and remove OpenMP parallel region.

CUDA Example

Compile and run

- Compile as before:

```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
```

- We need to specify `-arch=sm_60`, the architecture of the GPUs on the `cyc` nodes (P100 architecture), because by default `nvcc` compiles against a newer GPU version, compatible with newer nodes
- Run as before (I'm assuming `OMP_BIND`, `OMP_PLACES`, and `OMP_NUM_THREADS` were set before):

```
CPU: nthr =   16   t0 = 0.0150 sec   P =   8.945 Gflop/s   B =  53.670 GB/s
GPU:               t0 = 0.0001 sec   P = 2630.607 Gflop/s   B = 15783.644 GB/s
Diff = 1.021564e-15
```

CUDA Example

Compile and run

- Compile as before:

```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
```

- We need to specify `-arch=sm_60`, the architecture of the GPUs on the `cyc` nodes (P100 architecture), because by default `nvcc` compiles against a newer GPU version, compatible with newer nodes
- Run as before (I'm assuming `OMP_BIND`, `OMP_PLACES`, and `OMP_NUM_THREADS` were set before):

```
CPU: nthr =   16   t0 = 0.0150 sec   P =   8.945 Gflop/s   B =  53.670 GB/s
GPU:               t0 = 0.0001 sec   P = 2630.607 Gflop/s   B = 15783.644 GB/s
Diff = 1.021564e-15
```

This performance is infeasible. What's going on?

CUDA Example

Edits within `main()` 3/4

- The problem is here:

```
double t0 = stop_watch(0);
gpu_axpy<<<1, 1>>>(n, a, d_x, d_y);
t0 = stop_watch(t0);
```

- CUDA kernels return **immediately**; the kernel is still being executed on the device when `stop_watch(t0)` is called. We are **not** timing the kernel execution time, but the time it takes to dispatch the kernel to the GPU.

CUDA Example

Edits within `main()` 3/4

- The problem is here:

```
double t0 = stop_watch(0);
gpu_axpy<<<1, 1>>>(n, a, d_x, d_y);
t0 = stop_watch(t0);
```

- CUDA kernels return **immediately**; the kernel is still being executed on the device when `stop_watch(t0)` is called. We are **not** timing the kernel execution time, but the time it takes to dispatch the kernel to the GPU.
- Correct this by adding `cudaDeviceSynchronize();` after the CUDA kernel, which blocks until all running CUDA kernels are complete:

```
double t0 = stop_watch(0);
gpu_axpy<<<1, 1>>>(n, a, d_x, d_y);
cudaDeviceSynchronize();
t0 = stop_watch(t0);
```

CUDA Example

- Compile and run again:

```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0149 sec   P =   9.024 Gflop/s   B =  54.145 GB/s
 GPU:               t0 = 8.9564 sec   P =   0.015 Gflop/s   B =   0.090 GB/s
 Diff = 1.021564e-15
```

CUDA Example

- Compile and run again:

```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0149 sec   P =   9.024 Gflop/s   B =  54.145 GB/s
 GPU:               t0 = 8.9564 sec   P =   0.015 Gflop/s   B =   0.090 GB/s
 Diff = 1.021564e-15
```

- This performance is of course extremely poor;

CUDA Example

- Compile and run again:

```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0149 sec   P =   9.024 Gflop/s   B =  54.145 GB/s
 GPU:               t0 = 8.9564 sec   P =   0.015 Gflop/s   B =   0.090 GB/s
 Diff = 1.021564e-15
```

- This performance is of course extremely poor;
- We're using only one GPU thread in the kernel

CUDA Example

Use more threads

- In this step, we will use 512 GPU threads. First, change the call to the GPU kernel:

```
double t0 = stop_watch(0);
gpu_axpy<<<1, 512>>>(n, a, d_x, d_y);
cudaDeviceSynchronize();
t0 = stop_watch(t0);
```

CUDA Example

Use more threads

- In this step, we will use 512 GPU threads. First, change the call to the GPU kernel:

```
    double t0 = stop_watch(0);
    gpu_axpy<<<1, 512>>>(n, a, d_x, d_y);
    cudaDeviceSynchronize();
    t0 = stop_watch(t0);
```

- Then we need to change the kernel. We need in each GPU thread to calculate which elements it will operate on:

```
/***
 * Do y ← a*x + y on the GPU
 ***/
__global__ void
gpu_axpy(int n, float a, float *x, float *y)
{
  int ithr = threadIdx.x;
  int nthr = blockDim.x;
  int lt = n/nthr;
  for(int i=ithr*lt; i<(ithr+1)*lt; i++)
    y[i] = a*x[i] + y[i];
  return;
}
```

- With the above, each thread operated on `n/nthr` contiguous elements

CUDA Example

- Compile and run again:

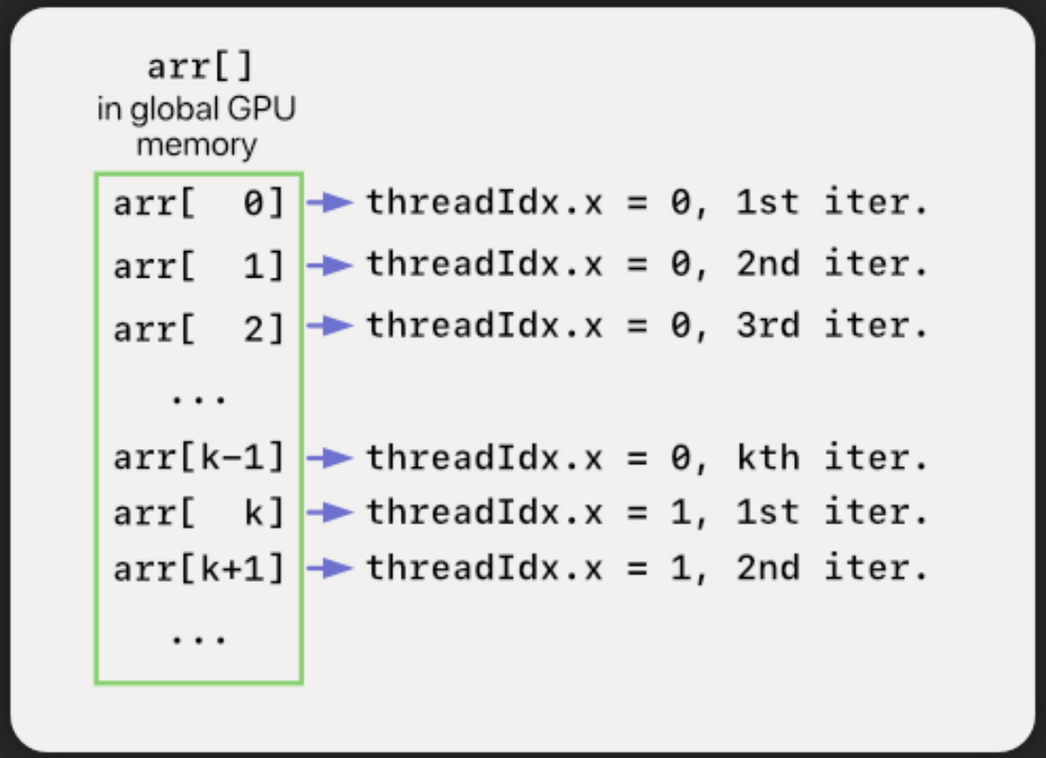
```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0150 sec   P =   8.940 Gflop/s   B =  53.637 GB/s
 GPU:               t0 = 0.2555 sec   P =   0.525 Gflop/s   B =   3.152 GB/s
 Diff = 1.021564e-15
```

- Better than before, but still very poor performance. Can we do better?

CUDA Example

Optimized GPU memory access

Always keep in mind that on GPUs, it is more optimal if contiguous threads access contiguous memory locations



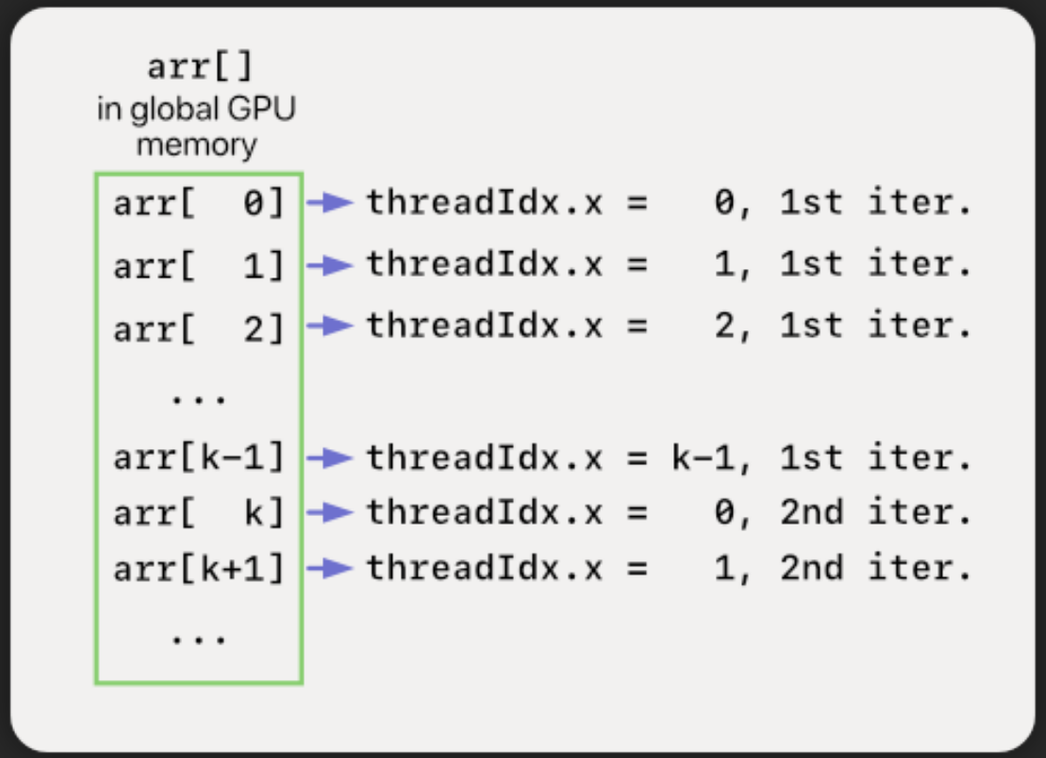
This represents the order by which elements are accessed currently

- The same thread accesses continuous elements
- Very common approach on CPUs
- On GPUs, this results in so-called *bank conflicts*
- *Suboptimal!*

CUDA Example

Optimized GPU memory access

Always keep in mind that on GPUs, it is more optimal if contiguous threads access contiguous memory locations



This represents an optimal data access pattern

- Different threads accesses continuous elements
- Each thread is served by a different memory bank

CUDA Example

Optimized GPU memory access

Always keep in mind that on GPUs, it is more optimal if contiguous threads access contiguous memory locations

In our example:

```
/***
 * Do y ← a*x + y on the GPU
 ***/
__global__ void
gpu_axpy(int n, float a, float *x, float *y)
{
  int ithr = threadIdx.x;
  int nthr = blockDim.x;
  for(int i=0; i<n; i+=nthr)
    y[i+ithr] = a*x[i+ithr] + y[i+ithr];
  return;
}
```

- Compile and run:

```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0150 sec   P =   8.950 Gflop/s   B =  53.698 GB/s
 GPU:               t0 = 0.0565 sec   P =   2.377 Gflop/s   B =  14.260 GB/s
 Diff = 1.021564e-15
```

CUDA Example

Blocks and threads

Now let's use blocks. Let's use as many blocks and threads as we can

- Upper limit of 1024 threads
- Upper limit of $2^{31} - 1$ blocks (practically infinite)

```
    double t0 = stop_watch(0);
    int nthr = 1024;
    gpu_axpy<<<n/nthr, nthr>>>(n, a, d_x, d_y);
    cudaDeviceSynchronize();
    t0 = stop_watch(t0);
```

```
/***
 * Do y ← a*x + y on the GPU
 ***/
__global__ void
gpu_axpy(int n, float a, float *x, float *y)
{
  int ithr = threadIdx.x;
  int nthr = blockDim.x;
  int iblk = blockIdx.x;
  int idx = ithr + iblk*nthr;
  y[idx] = a*x[idx] + y[idx];
  return;
}
```

CUDA Example

Blocks and threads

- Compile and run:

```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0144 sec   P =   9.303 Gflop/s   B =  55.816 GB/s
 GPU:               t0 = 0.0016 sec   P =  86.316 Gflop/s   B = 517.893 GB/s
 Diff = 1.021564e-15
```

- ~520 GB/s is ~70% of peak bandwidth (which is 732 GB/s)

CUDA Example

Blocks and threads

- Compile and run:

```
[ikoutsou@front02 ex01]$ nvcc -arch=sm_60 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 ex01]$ srun -n 1 --cpus-per-task=16 -p p100 --gres=gpu:1 ./axpy $((1024*1024*64))
 CPU: nthr =   16   t0 = 0.0144 sec   P =   9.303 Gflop/s   B =  55.816 GB/s
 GPU:               t0 = 0.0016 sec   P =  86.316 Gflop/s   B = 517.893 GB/s
 Diff = 1.021564e-15
```

- ~520 GB/s is ~70% of peak bandwidth (which is 732 GB/s)
- Try varying the number of threads per block. E.g. read it from the command line and scan for the optimal value.