Cluster Computing and GPU Programming Introduction

\\

Introduction to Parallel Computing for Beginners

15^{th} March 2024

\\

Giannis Koutsou,

Computation-based Science and Technology Research Center,

The Cyprus Institute

Outline for this hands-on part

Cluster computing

- Navigating an HPC cluster
- Build environment
- Submitting jobs

Introduction to GPU programming

- Outline of GPU architecture
- CUDA intro and hands-on practical

Outline for this hands-on part

Cluster computing

- Navigating an HPC cluster
- Build environment
- Submitting jobs

Introduction to GPU programming

- Outline of GPU architecture
- CUDA intro and hands-on practical

Assumptions about this hands-on session

- Some familiarity with programming in C or C++
- Familiarity with some common command-line tasks, e.g. compiling, navigating the file system
- Can edit text files on a remote server, e.g. text-based (emacs or vim) or VS Code Remote

Outline for this hands-on part

Cluster computing

- Navigating an HPC cluster
- Build environment
- Submitting jobs

Introduction to GPU programming

- Outline of GPU architecture
- CUDA intro and hands-on practical

Assumptions about this hands-on session

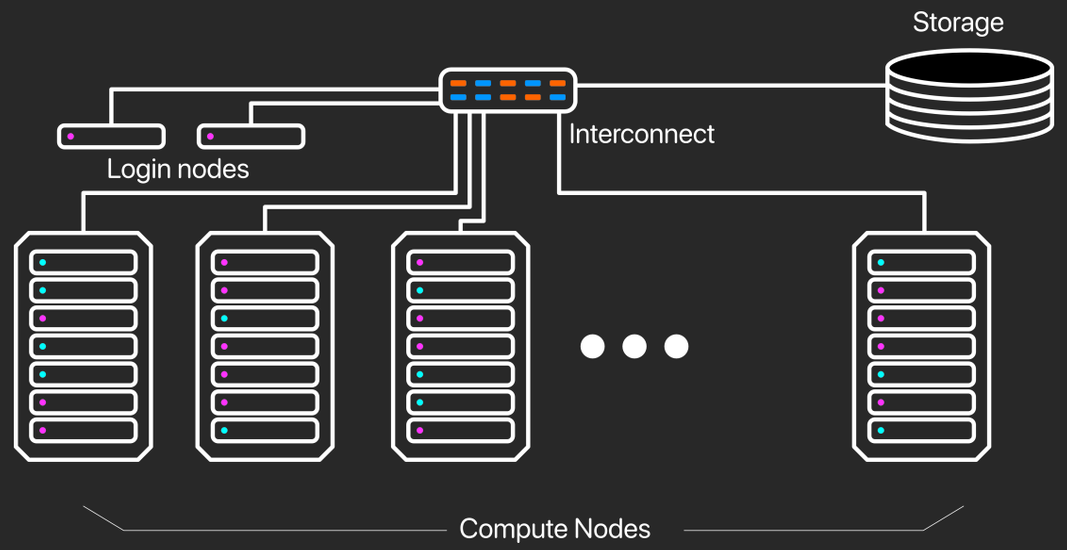
- Some familiarity with programming in C or C++
- Familiarity with some common command-line tasks, e.g. compiling, navigating the file system
- Can edit text files on a remote server, e.g. text-based (emacs or vim) or VS Code Remote

Ways of accessing these slides:

- As a PDF from this training event's github repo https://github.com/CaSToRC-CyI/NCC-training-202403
- Via web browser at slides.koutsou.net/EuroCC-2024-03-15 ⟵ **recommended**

Cluster Computing

Cluster Computing



Specific configuration of our local system

- Part of a larger system with ~100 nodes
- 2 nodes reserved for this training
  - Hostnames: `gpu{03,04}`
  - 2×20-core Intel Xeon
  - 186 GBytes RAM
  - 2×NVIDIA V100 GPUs with 32 GBytes Graphics memory each
- Common storage for our course: `/nvme/scratch/edu19/`

Cluster Computing

- Log in to a *login node* or *frontend node* — login node in our case has hostname `front02`
- To run programs on *compute nodes*, a *job scheduler* is available
- Distinguish between *interactive* and *batch* jobs

Cluster Computing

- Log in to a *login node* or *frontend node* — login node in our case has hostname `front02`
- To run programs on *compute nodes*, a *job scheduler* is available
- Distinguish between *interactive* and *batch* jobs

SLURM job scheduler

- See currently running and waiting jobs: `squeue`
- Ask for an interactive job: `salloc`
- Submit a batch job: `sbatch`
- Run an executable: `srun`

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@front02.hpcf.cyi.ac.cy
```

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@front02.hpcf.cyi.ac.cy
```

  ⟶ Should have gotten instructions how to get to this point before the training event

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@front02.hpcf.cyi.ac.cy
```

  ⟶ Should have gotten instructions how to get to this point before the training event

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front02 ~]$ hostname
front02
```

  this is the login node.

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@front02.hpcf.cyi.ac.cy
```

  ⟶ Should have gotten instructions how to get to this point before the training event

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front02 ~]$ hostname
front02
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front02 ~]$ salloc -N 1 -p gpu --reservation=edu19 -A edu19
salloc: Granted job allocation 397828
salloc: Waiting for resource configuration
salloc: Nodes gpu12 are ready for job
[ikoutsou@gpu03 ~]$
```

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@front02.hpcf.cyi.ac.cy
```

  ⟶ Should have gotten instructions how to get to this point before the training event

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front02 ~]$ hostname
front02
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front02 ~]$ salloc -N 1 -p gpu --reservation=edu19 -A edu19
salloc: Granted job allocation 397828
salloc: Waiting for resource configuration
salloc: Nodes gpu12 are ready for job
[ikoutsou@gpu03 ~]$
```

  - `-A edu19`: charge project with name `edu19`

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@front02.hpcf.cyi.ac.cy
```

  ⟶ Should have gotten instructions how to get to this point before the training event

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front02 ~]$ hostname
front02
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front02 ~]$ salloc -N 1 -p gpu --reservation=edu19 -A edu19
salloc: Granted job allocation 397828
salloc: Waiting for resource configuration
salloc: Nodes gpu12 are ready for job
[ikoutsou@gpu03 ~]$
```

  - `-A edu19`: charge project with name `edu19`
  - `--reservation=edu19`: use reservation `edu19` (reservation and project names need not be the same)

Cluster Computing Introductory Example

- Log in:

```
[localhost ~]$ ssh <username>@front02.hpcf.cyi.ac.cy
```

  ⟶ Should have gotten instructions how to get to this point before the training event

- Type `hostename`. This tells you the name of the node you are currently logged into:

```
[ikoutsou@front02 ~]$ hostname
front02
```

  this is the login node.

- Ask for one node (`-N 1`):

```
[ikoutsou@front02 ~]$ salloc -N 1 -p gpu --reservation=edu19 -A edu19
salloc: Granted job allocation 397828
salloc: Waiting for resource configuration
salloc: Nodes gpu12 are ready for job
[ikoutsou@gpu03 ~]$
```

  - `-A edu19`: charge project with name `edu19`
  - `--reservation=edu19`: use reservation `edu19` (reservation and project names need not be the same)
  - `-p=gpu`: requests a node from the `gpu` partition — the partition that includes all `gpuu??` nodes

Cluster Computing Introductory Example

- Type `hostname` again:

```
[ikoutsou@gpu03 ~]$ hostname
gpu03
```

`gpu03` is the hostname of a compute node

Cluster Computing Introductory Example

- Type `hostname` again:

```
[ikoutsou@gpu03 ~]$ hostname
gpu03
```

  `gpu03` is the hostname of a compute node

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@gpu03 ~]$ exit
salloc: Relinquishing job allocation 397828
[ikoutsou@front02 ~]$
```

  we're back on `front02`

Cluster Computing Introductory Example

- Type `hostname` again:

```
[ikoutsou@gpu03 ~]$ hostname
gpu03
```

  `gpu03` is the hostname of a compute node

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@gpu03 ~]$ exit
salloc: Relinquishing job allocation 397828
[ikoutsou@front02 ~]$
```

  we're back on `front02`

Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes.

Cluster Computing Introductory Example

- Type `hostname` again:

```
[ikoutsou@gpu03 ~]$ hostname
gpu03
```

  `gpu03` is the hostname of a compute node

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@gpu03 ~]$ exit
salloc: Relinquishing job allocation 397828
[ikoutsou@front02 ~]$
```

  we're back on `front02`

  Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes.

- Use `srun` instead of `salloc`:

```
[ikoutsou@front02 ~]$ srun -N 1 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
```

Cluster Computing Introductory Example

- Type `hostname` again:

```
[ikoutsou@gpu03 ~]$ hostname
gpu03
```

  `gpu03` is the hostname of a compute node

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@gpu03 ~]$ exit
salloc: Relinquishing job allocation 397828
[ikoutsou@front02 ~]$
```

  we're back on `front02`

  Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes.

- Use `srun` instead of `salloc`:

```
[ikoutsou@front02 ~]$ srun -N 1 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
```

- Allocates a node, runs the specified command (in this case `hostname`), and then exits the node, releasing the allocation

Cluster Computing Introductory Example

- Type `hostname` again:

```
[ikoutsou@gpu03 ~]$ hostname
gpu03
```

  `gpu03` is the hostname of a compute node

- Release the node (type `exit` or hit `ctrl-d`):

```
[ikoutsou@gpu03 ~]$ exit
salloc: Relinquishing job allocation 397828
[ikoutsou@front02 ~]$
```

  we're back on `front02`

  Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes.

- Use `srun` instead of `salloc`:

```
[ikoutsou@front02 ~]$ srun -N 1 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
```

- Allocates a node, runs the specified command (in this case `hostname`), and then exits the node, releasing the allocation
- The output, `gpu02`, reveals that we were allocated node `gpu02` for this specific — very short — job

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front02 ~]$ srun -N 1 -n 2 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
gpu02
```

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front02 ~]$ srun -N 1 -n 2 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
gpu02
```

  - `-N 1`: use one node
  - `-n 2`: use two processes

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front02 ~]$ srun -N 1 -n 2 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
gpu02
```

  - `-N 1`: use one node
  - `-n 2`: use two processes

- Run on more than one node:

```
[ikoutsou@front02 ~]$ srun -N 2 -n 2 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
gpu03
```

  runs one instance of `hostname` on each node

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front02 ~]$ srun -N 1 -n 2 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
gpu02
```

  - `-N 1`: use one node
  - `-n 2`: use two processes

- Run on more than one node:

```
[ikoutsou@front02 ~]$ srun -N 2 -n 2 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
gpu03
```

  runs one instance of `hostname` on each node

- Try:

```
[ikoutsou@front02 ~]$ srun -N 2 -n 1 -p gpu -A edu19 --reservation=edu19 hostname
```

Cluster Computing Introductory Example

- Run multiple instances of `hostname` in parallel:

```
[ikoutsou@front02 ~]$ srun -N 1 -n 2 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
gpu02
```

  - `-N 1`: use one node
  - `-n 2`: use two processes

- Run on more than one node:

```
[ikoutsou@front02 ~]$ srun -N 2 -n 2 -p gpu -A edu19 --reservation=edu19 hostname
gpu02
gpu03
```

  runs one instance of `hostname` on each node

- Try:

```
[ikoutsou@front02 ~]$ srun -N 2 -n 1 -p gpu -A edu19 --reservation=edu19 hostname
```

```
srun: Warning: can't run 1 processes on 2 nodes, setting nnodes to 1
gpu03
```

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front02 ~]$ mkdir eurocc-training
[ikoutsou@front02 ~]$ ls
eurocc-training
[ikoutsou@front02 ~]$
```

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front02 ~]$ mkdir eurocc-training
[ikoutsou@front02 ~]$ ls
eurocc-training
[ikoutsou@front02 ~]$
```

- Change into it:

```
[ikoutsou@front02 ~]$ cd eurocc-training/
[ikoutsou@front02 eurocc-training]$
```

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front02 ~]$ mkdir eurocc-training
[ikoutsou@front02 ~]$ ls
eurocc-training
[ikoutsou@front02 ~]$
```

- Change into it:

```
[ikoutsou@front02 ~]$ cd eurocc-training/
[ikoutsou@front02 eurocc-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front02 eurocc-training]$ pwd
/nvme/h/ikoutsou/eurocc-training
```

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front02 ~]$ mkdir eurocc-training
[ikoutsou@front02 ~]$ ls
eurocc-training
[ikoutsou@front02 ~]$
```

- Change into it:

```
[ikoutsou@front02 ~]$ cd eurocc-training/
[ikoutsou@front02 eurocc-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front02 eurocc-training]$ pwd
/nvme/h/ikoutsou/eurocc-training
```

- `/` is referred to as the *root directory*

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front02 ~]$ mkdir eurocc-training
[ikoutsou@front02 ~]$ ls
eurocc-training
[ikoutsou@front02 ~]$
```

- Change into it:

```
[ikoutsou@front02 ~]$ cd eurocc-training/
[ikoutsou@front02 eurocc-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front02 eurocc-training]$ pwd
/nvme/h/ikoutsou/eurocc-training
```

- `/` is referred to as the *root directory*
- `.` is an alias for the *current directory*

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front02 ~]$ mkdir eurocc-training
[ikoutsou@front02 ~]$ ls
eurocc-training
[ikoutsou@front02 ~]$
```

- Change into it:

```
[ikoutsou@front02 ~]$ cd eurocc-training/
[ikoutsou@front02 eurocc-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front02 eurocc-training]$ pwd
/nvme/h/ikoutsou/eurocc-training
```

- `/` is referred to as the *root directory*
- `.` is an alias for the *current directory*
- `..` is an alias for the directory one level above

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front02 ~]$ mkdir eurocc-training
[ikoutsou@front02 ~]$ ls
eurocc-training
[ikoutsou@front02 ~]$
```

- Change into it:

```
[ikoutsou@front02 ~]$ cd eurocc-training/
[ikoutsou@front02 eurocc-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front02 eurocc-training]$ pwd
/nvme/h/ikoutsou/eurocc-training
```

- `/` is referred to as the *root directory*
- `.` is an alias for the *current directory*
- `..` is an alias for the directory one level above
- `~` is an alias for your *home directory*

Cluster Computing Introductory Example

- Make a directory. List it, see that it is there:

```
[ikoutsou@front02 ~]$ mkdir eurocc-training
[ikoutsou@front02 ~]$ ls
eurocc-training
[ikoutsou@front02 ~]$
```

- Change into it:

```
[ikoutsou@front02 ~]$ cd eurocc-training/
[ikoutsou@front02 eurocc-training]$
```

- `pwd` will tell you where you are in the file system:

```
[ikoutsou@front02 eurocc-training]$ pwd
/nvme/h/ikoutsou/eurocc-training
```

- `/` is referred to as the *root directory*
- `.` is an alias for the *current directory*
- `..` is an alias for the directory one level above
- `~` is an alias for your *home directory*
- E.g.:

```
[ikoutsou@front02 eurocc-training]$ cd ../../
[ikoutsou@front02 h]$ pwd
/nvme/h
```

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front02 h]$ cd
[ikoutsou@front02 ~]$ pwd
/nvme/h/ikoutsou
```

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front02 h]$ cd
[ikoutsou@front02 ~]$ pwd
/nvme/h/ikoutsou
```

- Make a subdirectory under `eurocc-training` for our first C program

```
[ikoutsou@front02 ~]$ cd engage-training
[ikoutsou@front02 eurocc-training]$ mkdir 01
[ikoutsou@front02 eurocc-training]$ cd 01
[ikoutsou@front02 01]$
```

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front02 h]$ cd
[ikoutsou@front02 ~]$ pwd
/nvme/h/ikoutsou
```

- Make a subdirectory under `eurocc-training` for our first C program

```
[ikoutsou@front02 ~]$ cd engage-training
[ikoutsou@front02 eurocc-training]$ mkdir 01
[ikoutsou@front02 eurocc-training]$ cd 01
[ikoutsou@front02 01]$
```

- Copy the program from our shared space

```
[ikoutsou@front02 01]$ cp /nvme/scratch/edu19/01/prog-01.c .
```

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front02 h]$ cd
[ikoutsou@front02 ~]$ pwd
/nvme/h/ikoutsou
```

- Make a subdirectory under `eurocc-training` for our first C program

```
[ikoutsou@front02 ~]$ cd engage-training
[ikoutsou@front02 eurocc-training]$ mkdir 01
[ikoutsou@front02 eurocc-training]$ cd 01
[ikoutsou@front02 01]$
```

- Copy the program from our shared space

```
[ikoutsou@front02 01]$ cp /nvme/scratch/edu19/01/prog-01.c .
```

- Use `emacs` or `vim` (or any other text editor you're comfortable with) to inspect the program

Cluster Computing Introductory Example

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[ikoutsou@front02 h]$ cd
[ikoutsou@front02 ~]$ pwd
/nvme/h/ikoutsou
```

- Make a subdirectory under `eurocc-training` for our first C program

```
[ikoutsou@front02 ~]$ cd engage-training
[ikoutsou@front02 eurocc-training]$ mkdir 01
[ikoutsou@front02 eurocc-training]$ cd 01
[ikoutsou@front02 01]$
```

- Copy the program from our shared space

```
[ikoutsou@front02 01]$ cp /nvme/scratch/edu19/01/prog-01.c .
```

- Use `emacs` or `vim` (or any other text editor you're comfortable with) to inspect the program
- E.g.:

```
[ikoutsou@front02 01]$ emacs -nw prog-01.c
```

Cluster Computing Introductory Example

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>



int
main(int argc, char *argv[])
{
  char hname[256];
  pid_t p;
  gethostname(hname, 256);
  p = getpid();
  printf(" Hostname: %s, pid: %lu\n", hname, p);
  return 0;

}
```

Cluster Computing Introductory Example

```c
#include <unistd.h>           // ⟵ provides definitions for gethostname() and getpid()
#include <stdio.h>            // ⟵ provides definitions for printf()
#include <sys/types.h>        // ⟵ defines the pid_t type
// main()
//   → argv[] is an array of strings which holds all command line arguments
//   → argc holds the number of elements of argv
int
main(int argc, char *argv[])
{
  char hname[256];          // ⟵ declare hname[] as an array of 256 characters (a string of length 256)
  pid_t p;                  // ⟵ declare p as a pid_t type, in this case, an unsigned long integer
  gethostname(hname, 256); // ⟵ call gethostname(), return hostname in hname which is 256 characters long
  p = getpid();             // ⟵ call getpid(), return value in p
  printf(" Hostname: %s, pid: %lu\n", hname, p); // print statement
  return 0; // ⟵ return a value of 0 to the operating system. By convention 0 means success.
            //     Non zero values indicate errors.
}
```

Cluster Computing Introductory Example

Time to *compile* the program into an executable.

Cluster Computing Introductory Example

Time to *compile* the program into an executable.

- It is common on HPC systems, as is the case for this system, to have multiple versions and releases of software installed

Cluster Computing Introductory Example

Time to *compile* the program into an executable.

- It is common on HPC systems, as is the case for this system, to have multiple versions and releases of software installed
- This is also the case for *build environments*, i.e. various versions of compilers, linkers, debugers, and libraries

Cluster Computing Introductory Example

Time to *compile* the program into an executable.

- It is common on HPC systems, as is the case for this system, to have multiple versions and releases of software installed
- This is also the case for *build environments*, i.e. various versions of compilers, linkers, debugers, and libraries
- The user can select the version that is available to them in each session via a so-called *modules system*

Cluster Computing Introductory Example

Time to *compile* the program into an executable.

- It is common on HPC systems, as is the case for this system, to have multiple versions and releases of software installed
- This is also the case for *build environments*, i.e. various versions of compilers, linkers, debugers, and libraries
- The user can select the version that is available to them in each session via a so-called *modules system*
- List all available modules (long listing truncated in this slide)

```
[ikoutsou@front02 01]$ module avail

──────────────────────────────────────────── /eb/modules/all ────────────────────────────────────────────
   ABAQUS/2024                                           attr/2.5.1-GCCcore-11.3.0
   ANTLR/2.7.7-GCCcore-8.3.0-Java-11                     binutils/2.32-GCCcore-8.3.0

   ...

Use "module spider" to find all possible modules and extensions.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of the "keys".
```

Cluster Computing Introductory Example

Time to *compile* the program into an executable.

- It is common on HPC systems, as is the case for this system, to have multiple versions and releases of software installed
- This is also the case for *build environments*, i.e. various versions of compilers, linkers, debugers, and libraries
- The user can select the version that is available to them in each session via a so-called *modules system*
- List all available modules (long listing truncated in this slide)

```
[ikoutsou@front02 01]$ module avail

------------------------------------------------------------------------------ /eb/modules/all ------------------------------------------------------------------------------
   ABAQUS/2024                                                                  attr/2.5.1-GCCcore-11.3.0
   ANTLR/2.7.7-GCCcore-8.3.0-Java-11                                            binutils/2.32-GCCcore-8.3.0

    ...

Use "module spider" to find all possible modules and extensions.
Use "module keyword key1 key2  ..." to search for all possible modules matching any of the "keys".
```

- `module list` should show the currently loaded modules. In a fresh environment with no modules loaded, you should see:

```
[ikoutsou@front02 01]$ module list
No modules loaded
```

Cluster Computing Introductory Example

Time to *compile* the program into an executable.

- It is common on HPC systems, as is the case for this system, to have multiple versions and releases of software installed
- This is also the case for *build environments*, i.e. various versions of compilers, linkers, debugers, and libraries
- The user can select the version that is available to them in each session via a so-called *modules system*
- List all available modules (long listing truncated in this slide)

```
[ikoutsou@front02 01]$ module avail

------------------------------------------------------------------------------ /eb/modules/all ------------------------------------------------------------------------------
   ABAQUS/2024                                                                  attr/2.5.1-GCCcore-11.3.0
   ANTLR/2.7.7-GCCcore-8.3.0-Java-11                                            binutils/2.32-GCCcore-8.3.0

    ...

Use "module spider" to find all possible modules and extensions.
Use "module keyword key1 key2  ..." to search for all possible modules matching any of the "keys".
```

- `module list` should show the currently loaded modules. In a fresh environment with no modules loaded, you should see:

```
[ikoutsou@front02 01]$ module list
No modules loaded
```

- `module purge` will unload all modules

Cluster Computing Introductory Example

- We will be using the `gompi` module set. See info:

```
[ikoutsou@front02 ~]$ module show gompi
...
```

Cluster Computing Introductory Example

- We will be using the `gompi` module set. See info:

```
[ikoutsou@front02 ~]$ module show gompi
...
```

- GNU C/C++ compilers
- OpenMPI implementation of MPI library (not relevant here, but this is how I tested the exercises)

Cluster Computing Introductory Example

- We will be using the `gompi` module set. See info:

```
[ikoutsou@front02 ~]$ module show gompi
...
```

- GNU C/C++ compilers
- OpenMPI implementation of MPI library (not relevant here, but this is how I tested the exercises)

```
[ikoutsou@front02 01]$ module load gompi
[ikoutsou@front02 01]$ module list

Currently Loaded Modules:
  1) GCCcore/11.3.0                 5) numactl/2.0.14-GCCcore-11.3.0      9) hwloc/2.7.1-GCCcore-11.3.0      13) libfabric/1.15.1-GCCcore-11.3.0  1
  2) zlib/1.2.12-GCCcore-11.3.0     6) XZ/5.2.5-GCCcore-11.3.0           10) OpenSSL/1.1                     14) PMIx/4.1.2-GCCcore-11.3.0
  3) binutils/2.38-GCCcore-11.3.0   7) libxml2/2.9.13-GCCcore-11.3.0     11) libevent/2.1.12-GCCcore-11.3.0  15) UCC/1.0.0-GCCcore-11.3.0
  4) GCC/11.3.0                     8) libpciaccess/0.16-GCCcore-11.3.0  12) UCX/1.12.1-GCCcore-11.3.0       16) OpenMPI/4.1.4-GCC-11.3.0
```

Cluster Computing Introductory Example

- We will be using the `gompi` module set. See info:

```
[ikoutsou@front02 ~]$ module show gompi
...
```

- GNU C/C++ compilers
- OpenMPI implementation of MPI library (not relevant here, but this is how I tested the exercises)

```
[ikoutsou@front02 01]$ module load gompi
[ikoutsou@front02 01]$ module list

Currently Loaded Modules:
  1) GCCcore/11.3.0                 5) numactl/2.0.14-GCCcore-11.3.0      9) hwloc/2.7.1-GCCcore-11.3.0      13) libfabric/1.15.1-GCCcore-11.3.0  1
  2) zlib/1.2.12-GCCcore-11.3.0     6) XZ/5.2.5-GCCcore-11.3.0           10) OpenSSL/1.1                     14) PMIx/4.1.2-GCCcore-11.3.0
  3) binutils/2.38-GCCcore-11.3.0   7) libxml2/2.9.13-GCCcore-11.3.0     11) libevent/2.1.12-GCCcore-11.3.0  15) UCC/1.0.0-GCCcore-11.3.0
  4) GCC/11.3.0                     8) libpciaccess/0.16-GCCcore-11.3.0  12) UCX/1.12.1-GCCcore-11.3.0       16) OpenMPI/4.1.4-GCC-11.3.0
```

- `gompi` is really a collection of modules as listed above

Cluster Computing Introductory Example

- We will be using the `gompi` module set. See info:

```
[ikoutsou@front02 ~]$ module show gompi
...
```

- GNU C/C++ compilers
- OpenMPI implementation of MPI library (not relevant here, but this is how I tested the exercises)

```
[ikoutsou@front02 01]$ module load gompi
[ikoutsou@front02 01]$ module list

Currently Loaded Modules:
  1) GCCcore/11.3.0                 5) numactl/2.0.14-GCCcore-11.3.0      9) hwloc/2.7.1-GCCcore-11.3.0      13) libfabric/1.15.1-GCCcore-11.3.0  1
  2) zlib/1.2.12-GCCcore-11.3.0     6) XZ/5.2.5-GCCcore-11.3.0           10) OpenSSL/1.1                     14) PMIx/4.1.2-GCCcore-11.3.0
  3) binutils/2.38-GCCcore-11.3.0   7) libxml2/2.9.13-GCCcore-11.3.0     11) libevent/2.1.12-GCCcore-11.3.0  15) UCC/1.0.0-GCCcore-11.3.0
  4) GCC/11.3.0                     8) libpciaccess/0.16-GCCcore-11.3.0  12) UCX/1.12.1-GCCcore-11.3.0       16) OpenMPI/4.1.4-GCC-11.3.0
```

- `gompi` is really a collection of modules as listed above
- To compile:

```
[ikoutsou@front02 01]$ module load gompi
[ikoutsou@front02 01]$ gcc prog-01.c -o p01
```

Cluster Computing Introductory Example

- We will be using the `gompi` module set. See info:

```
[ikoutsou@front02 ~]$ module show gompi
...
```

- GNU C/C++ compilers
- OpenMPI implementation of MPI library (not relevant here, but this is how I tested the exercises)

```
[ikoutsou@front02 01]$ module load gompi
[ikoutsou@front02 01]$ module list

Currently Loaded Modules:
  1) GCCcore/11.3.0                 5) numactl/2.0.14-GCCcore-11.3.0      9) hwloc/2.7.1-GCCcore-11.3.0      13) libfabric/1.15.1-GCCcore-11.3.0  1
  2) zlib/1.2.12-GCCcore-11.3.0     6) XZ/5.2.5-GCCcore-11.3.0           10) OpenSSL/1.1                     14) PMIx/4.1.2-GCCcore-11.3.0
  3) binutils/2.38-GCCcore-11.3.0   7) libxml2/2.9.13-GCCcore-11.3.0     11) libevent/2.1.12-GCCcore-11.3.0  15) UCC/1.0.0-GCCcore-11.3.0
  4) GCC/11.3.0                     8) libpciaccess/0.16-GCCcore-11.3.0  12) UCX/1.12.1-GCCcore-11.3.0       16) OpenMPI/4.1.4-GCC-11.3.0
```

- `gompi` is really a collection of modules as listed above
- To compile:

```
[ikoutsou@front02 01]$ module load gompi
[ikoutsou@front02 01]$ gcc prog-01.c -o p01
```

- `-o p01` means "name the resulting executable `p01`". If you don't specify `-o` the executable name defaults to `a.out`

Cluster Computing Introductory Example

- Type `ls` to make sure it has been created. Then run it on the frontend node:

```
[ikoutsou@front02 01]$ ls
p01  prog-01.c
[ikoutsou@front02 01]$ ./p01
Hostname: front02, pid: 14848
```

Cluster Computing Introductory Example

- Type `ls` to make sure it has been created. Then run it on the frontend node:

```
[ikoutsou@front02 01]$ ls
p01  prog-01.c
[ikoutsou@front02 01]$ ./p01
Hostname: front02, pid: 14848
```

- Run your new program `p01` using `srun` on two nodes with two processes each

Cluster Computing Introductory Example

- Type `ls` to make sure it has been created. Then run it on the frontend node:

```
[ikoutsou@front02 01]$ ls
p01  prog-01.c
[ikoutsou@front02 01]$ ./p01
Hostname: front02, pid: 14848
```

- Run your new program `p01` using `srun` on two nodes with two processes each

```
[ikoutsou@front02 01]$ srun -N 2 -n 4 -p gpu --reservation=edu19 -A edu19 ./p01
Hostname: gpu02, pid: 129145
Hostname: gpu03, pid: 241544
Hostname: gpu03, pid: 241543
Hostname: gpu03, pid: 241545
[ikoutsou@front02 01]$
```

Cluster Computing Introductory Example

- Type `ls` to make sure it has been created. Then run it on the frontend node:

```
[ikoutsou@front02 01]$ ls
p01  prog-01.c
[ikoutsou@front02 01]$ ./p01
Hostname: front02, pid: 14848
```

- Run your new program `p01` using `srun` on two nodes with two processes each

```
[ikoutsou@front02 01]$ srun -N 2 -n 4 -p gpu --reservation=edu19 -A edu19 ./p01
Hostname: gpu02, pid: 129145
Hostname: gpu03, pid: 241544
Hostname: gpu03, pid: 241543
Hostname: gpu03, pid: 241545
[ikoutsou@front02 01]$
```

  1 process ran on `gpu02` and 3 processes ran *in parallel* on `gpu03`

Cluster Computing Introductory Example

- Time for step 2 — a simple program to compute the Fletcher 32 checksum of several files

Cluster Computing Introductory Example

- Time for step 2 — a simple program to compute the Fletcher 32 checksum of several files
- First, make a new directory under `~/eurocc-training/.`

```
[ikoutsou@front02 01]$ cd ../
[ikoutsou@front02 eurocc-training]$ mkdir 02
[ikoutsou@front02 eurocc-training]$ cd 02/
[ikoutsou@front02 02]$
```

Cluster Computing Introductory Example

- Time for step 2 — a simple program to compute the Fletcher 32 checksum of several files
- First, make a new directory under `~/eurocc-training/.`

```
[ikoutsou@front02 01]$ cd ../
[ikoutsou@front02 eurocc-training]$ mkdir 02
[ikoutsou@front02 eurocc-training]$ cd 02/
[ikoutsou@front02 02]$
```

- Copy the program `fletcher32.c` from `/nvme/scratch/edu19/02/fletcher32.c`

```
[ikoutsou@front02 02]$ cp /nvme/scratch/edu19/02/fletcher32.c .
```

Cluster Computing Introductory Example

- Time for step 2 — a simple program to compute the Fletcher 32 checksum of several files
- First, make a new directory under `~/eurocc-training/.`

```
[ikoutsou@front02 01]$ cd ../
[ikoutsou@front02 eurocc-training]$ mkdir 02
[ikoutsou@front02 eurocc-training]$ cd 02/
[ikoutsou@front02 02]$
```

- Copy the program `fletcher32.c` from `/nvme/scratch/edu19/02/fletcher32.c`

```
[ikoutsou@front02 02]$ cp /nvme/scratch/edu19/02/fletcher32.c .
```

- Inspect `fletcher32.c`, e.g.:

```
[ikoutsou@front02 02]$ emacs -nw fletcher32.c
```

Cluster Computing Introductory Example

- Compile and run on the frontend:

```
[ikoutsou@front02 02]$ gcc fletcher32.c -o f32
[ikoutsou@front02 02]$ ./f32
Usage: ./f32 FNAME
```

Cluster Computing Introductory Example

- Compile and run on the frontend:

```
[ikoutsou@front02 02]$ gcc fletcher32.c -o f32
[ikoutsou@front02 02]$ ./f32
Usage: ./f32 FNAME
```

- Requires as argument the filename to compute the checksum over. There are 10 files you can use under `/nvme/scratch/edu19/data/`:

```
[ikoutsou@front02 02]$ ls -1 /nvme/scratch/edu19/data
00.txt
01.txt
02.txt
03.txt
04.txt
05.txt
06.txt
07.txt
08.txt
09.txt
[ikoutsou@front02 02]$
```

Cluster Computing Introductory Example

- Compile and run on the frontend:

```
[ikoutsou@front02 02]$ gcc fletcher32.c -o f32
[ikoutsou@front02 02]$ ./f32
Usage: ./f32 FNAME
```

- Requires as argument the filename to compute the checksum over. There are 10 files you can use under `/nvme/scratch/edu19/data/`:

```
[ikoutsou@front02 02]$ ls -1 /nvme/scratch/edu19/data
00.txt
01.txt
02.txt
03.txt
04.txt
05.txt
06.txt
07.txt
08.txt
09.txt
[ikoutsou@front02 02]$
```

- For example:

```
[ikoutsou@front02 02]$ ./f32 /nvme/scratch/edu19/data/03.txt
/nvme/scratch/edu19/data/03.txt: 0c40e2d2
```

Cluster Computing Introductory Example

- Compile and run on the frontend:

```
[ikoutsou@front02 02]$ gcc fletcher32.c -o f32
[ikoutsou@front02 02]$ ./f32
Usage: ./f32 FNAME
```

- Requires as argument the filename to compute the checksum over. There are 10 files you can use under `/nvme/scratch/edu19/data/`:

```
[ikoutsou@front02 02]$ ls -1 /nvme/scratch/edu19/data
00.txt
01.txt
02.txt
03.txt
04.txt
05.txt
06.txt
07.txt
08.txt
09.txt
[ikoutsou@front02 02]$
```

- For example:

```
[ikoutsou@front02 02]$ ./f32 /nvme/scratch/edu19/data/03.txt
/nvme/scratch/edu19/data/03.txt: 0c40e2d2
```

- Takes ~1.5 - 2 seconds per file

Cluster Computing Introductory Example

- Our objective is to compute the checksums of all 10 files in parallel

Cluster Computing Introductory Example

- Our objective is to compute the checksums of all 10 files in parallel
- We will use the compute nodes available to us and the 40 available cores per node

Cluster Computing Introductory Example

- Our objective is to compute the checksums of all 10 files in parallel
- We will use the compute nodes available to us and the 40 available cores per node
- We will start by using a *Slurm batch script* as opposed to running *interactively* which we have been doing so far

Cluster Computing Introductory Example

- Copy from `/nvme/scratch/edu19/02/fletcher.sh`

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 1
#SBATCH -N 1
#SBATCH --ntasks-per-node=1

date +"%T.%6N"
srun ./f32 /nvme/scratch/edu19/data/03.txt
date +"%T.%6N"
```

Cluster Computing Introductory Example

- Copy from `/nvme/scratch/edu19/02/fletcher.sh`

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 1
#SBATCH -N 1
#SBATCH --ntasks-per-node=1

date +"%T.%6N"
srun ./f32 /nvme/scratch/edu19/data/03.txt
date +"%T.%6N"
```

- `date +"%T.%6N"` prints the current time, including milliseconds (you can try this on the command line interactively)

Cluster Computing Introductory Example

- Copy from `/nvme/scratch/edu19/02/fletcher.sh`

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 1
#SBATCH -N 1
#SBATCH --ntasks-per-node=1

date +"%T.%6N"
srun ./f32 /nvme/scratch/edu19/data/03.txt
date +"%T.%6N"
```

- `date +"%T.%6N"` prints the current time, including milliseconds (you can try this on the command line interactively)
- The lines starting with `#SBATCH` are parsed by Slurm, even though they are ignored as comments by bash.

  The options are the same as you would pass to `srun`. Notice that `srun` now takes no options

Cluster Computing Introductory Example

- Submit the script via `sbatch`:

```
[ikoutsou@front02 02]$ sbatch fletcher.sh
Submitted batch job 397847
```

Cluster Computing Introductory Example

- Submit the script via `sbatch`:

```
[ikoutsou@front02 02]$ sbatch fletcher.sh
Submitted batch job 397847
```

- Check status via `squeue`:

```
[ikoutsou@front02 02]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            397850       gpu fletcher ikoutsou  R       0:01      1 gpu12
```

Cluster Computing Introductory Example

- Submit the script via `sbatch`:

```
[ikoutsou@front02 02]$ sbatch fletcher.sh
Submitted batch job 397847
```

- Check status via `squeue`:

```
[ikoutsou@front02 02]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            397850       gpu fletcher ikoutsou  R       0:01      1 gpu12
```

- `R` means "Running". You may see `PD` for "Pending" (not yet running) or `CG` for "Completing"

Cluster Computing Introductory Example

- Submit the script via `sbatch`:

```
[ikoutsou@front02 02]$ sbatch fletcher.sh
Submitted batch job 397847
```

- Check status via `squeue`:

```
[ikoutsou@front02 02]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            397850       gpu fletcher ikoutsou  R       0:01      1 gpu12
```

- `R` means "Running". You may see `PD` for "Pending" (not yet running) or `CG` for "Completing"
- When the job has finished, `squeue -u $USER` should not show any jobs

Cluster Computing Introductory Example

- Submit the script via `sbatch`:

```
[ikoutsou@front02 02]$ sbatch fletcher.sh
Submitted batch job 397847
```

- Check status via `squeue`:

```
[ikoutsou@front02 02]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            397850       gpu fletcher ikoutsou  R       0:01      1 gpu12
```

- `R` means "Running". You may see `PD` for "Pending" (not yet running) or `CG` for "Completing"
- When the job has finished, `squeue -u $USER` should not show any jobs
- The output is now in `f32.txt` (which was specified in the batch script). Any errors will be in `f32.err`

Cluster Computing Introductory Example

- Submit the script via `sbatch`:

```
[ikoutsou@front02 02]$ sbatch fletcher.sh
Submitted batch job 397847
```

- Check status via `squeue`:

```
[ikoutsou@front02 02]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            397850       gpu fletcher ikoutsou  R       0:01      1 gpu12
```

- `R` means "Running". You may see `PD` for "Pending" (not yet running) or `CG` for "Completing"
- When the job has finished, `squeue -u $USER` should not show any jobs
- The output is now in `f32.txt` (which was specified in the batch script). Any errors will be in `f32.err`
- This is the output that would be shown on the terminal had you run the program interactively. Slurm redirects the output and error to those two files

Cluster Computing Introductory Example

- Submit the script via `sbatch`:

```
[ikoutsou@front02 02]$ sbatch fletcher.sh
Submitted batch job 397847
```

- Check status via `squeue`:

```
[ikoutsou@front02 02]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            397850       gpu fletcher ikoutsou  R       0:01      1 gpu12
```

- `R` means "Running". You may see `PD` for "Pending" (not yet running) or `CG` for "Completing"
- When the job has finished, `squeue -u $USER` should not show any jobs
- The output is now in `f32.txt` (which was specified in the batch script). Any errors will be in `f32.err`
- This is the output that would be shown on the terminal had you run the program interactively. Slurm redirects the output and error to those two files
- Check the output:

```
[ikoutsou@front02 02]$ cat f32.txt
16:47:53.325712
/nvme/scratch/edu19/data/03.txt: 0c40e2d2
16:47:54.756334
```

Cluster Computing Introductory Example

Now let's loop over the 10 files

- Could just copy-paste the line to have 10 `srun` lines

Cluster Computing Introductory Example

Now let's loop over the 10 files

- Could just copy-paste the line to have 10 `srun` lines
- But here we will go with something a little more elegant:

```
date +"%T.%6N"
for((i=0; i<10; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun ./f32 /nvme/scratch/edu19/data/$filename
done
date +"%T.%6N"
```

- The `#SBATCH` lines not shown are unchanged compared to before

Cluster Computing Introductory Example

Now let's loop over the 10 files

- Could just copy-paste the line to have 10 `srun` lines
- But here we will go with something a little more elegant:

```
date +"%T.%6N"
for((i=0; i<10; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun ./f32 /nvme/scratch/edu19/data/$filename
done
date +"%T.%6N"
```

- The `#SBATCH` lines not shown are unchanged compared to before
- Submit and check output once completed (will overwrite previous output). Note that the 10 iterations need about 15 seconds:

```
[ikoutsou@front02 02]$ cat f32.txt
17:12:25.469723
 /nvme/scratch/edu19/data/00.txt: 04d70552
 /nvme/scratch/edu19/data/01.txt: 19708cd4
 /nvme/scratch/edu19/data/02.txt: ed737a1c
 /nvme/scratch/edu19/data/03.txt: 0c40e2d2
 /nvme/scratch/edu19/data/04.txt: f7bde74d
 /nvme/scratch/edu19/data/05.txt: 562ddd6c
 /nvme/scratch/edu19/data/06.txt: 6f2cd2f1
 /nvme/scratch/edu19/data/07.txt: 016db6c6
 /nvme/scratch/edu19/data/08.txt: 700c1fd1
 /nvme/scratch/edu19/data/09.txt: f1c7b711
17:12:39.638734
```

Cluster Computing Introductory Example

Now we will modify the script to run the ten iterations *in parallel*

Cluster Computing Introductory Example

Now we will modify the script to run the ten iterations *in parallel*

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 10
#SBATCH -N 2
#SBATCH --ntasks-per-node=5
date +"%T.%6N"
for((i=0; i<10; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun --exclusive -n 1 -N 1 ./f32 /nvme/scratch/edu19/data/$filename &
done
wait
date +"%T.%6N"
```

Cluster Computing Introductory Example

Now we will modify the script to run the ten iterations *in parallel*

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 10
#SBATCH -N 2
#SBATCH --ntasks-per-node=5
date +"%T.%6N"
for((i=0; i<10; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun --exclusive -n 1 -N 1 ./f32 /nvme/scratch/edu19/data/$filename &
done
wait
date +"%T.%6N"
```

- 10 processes in total (`-n 10`), 2 nodes (`-N 2`), 5 processes per node (`--ntasks-per-node=5`)

Cluster Computing Introductory Example

Now we will modify the script to run the ten iterations *in parallel*

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 10
#SBATCH -N 2
#SBATCH --ntasks-per-node=5
date +"%T.%6N"
for((i=0; i<10; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun --exclusive -n 1 -N 1 ./f32 /nvme/scratch/edu19/data/$filename &
done
wait
date +"%T.%6N"
```

- 10 processes in total (`-n 10`), 2 nodes (`-N 2`), 5 processes per node (`--ntasks-per-node=5`)
- Each instance of `srun` will run on one node and a single process (`srun -n 1 -N 1`)

Cluster Computing Introductory Example

Now we will modify the script to run the ten iterations *in parallel*

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 10
#SBATCH -N 2
#SBATCH --ntasks-per-node=5
date +"%T.%6N"
for((i=0; i<10; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun --exclusive -n 1 -N 1 ./f32 /nvme/scratch/edu19/data/$filename &
done
wait
date +"%T.%6N"
```

- 10 processes in total (`-n 10`), 2 nodes (`-N 2`), 5 processes per node (`--ntasks-per-node=5`)
- Each instance of `srun` will run on one node and a single process (`srun -n 1 -N 1`)
- `--exclusive` means that each process will be dedicated a processor (i.e., a "CPU core")

Cluster Computing Introductory Example

Now we will modify the script to run the ten iterations *in parallel*

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 10
#SBATCH -N 2
#SBATCH --ntasks-per-node=5
date +"%T.%6N"
for((i=0; i<10; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun --exclusive -n 1 -N 1 ./f32 /nvme/scratch/edu19/data/$filename &
done
wait
date +"%T.%6N"
```

- 10 processes in total (`-n 10`), 2 nodes (`-N 2`), 5 processes per node (`--ntasks-per-node=5`)
- Each instance of `srun` will run on one node and a single process (`srun -n 1 -N 1`)
- `--exclusive` means that each process will be dedicated a processor (i.e., a "CPU core")
- The ampersand "&" at the end of the `srun` line indicates that the command in this line should be *sent to the background*, i.e. the loop will go to the next iteration without waiting for the current iteration to complete

Cluster Computing Introductory Example

Now we will modify the script to run the ten iterations *in parallel*

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p gpu
#SBATCH -A edu19
#SBATCH --reservation=edu19
#SBATCH -t 00:02:00
#SBATCH -n 10
#SBATCH -N 2
#SBATCH --ntasks-per-node=5
date +"%T.%6N"
for((i=0; i<10; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun --exclusive -n 1 -N 1 ./f32 /nvme/scratch/edu19/data/$filename &
done
wait
date +"%T.%6N"
```

- 10 processes in total (`-n 10`), 2 nodes (`-N 2`), 5 processes per node (`--ntasks-per-node=5`)
- Each instance of `srun` will run on one node and a single process (`srun -n 1 -N 1`)
- `--exclusive` means that each process will be dedicated a processor (i.e., a "CPU core")
- The ampersand "&" at the end of the `srun` line indicates that the command in this line should be *sent to the background*, i.e. the loop will go to the next iteration without waiting for the current iteration to complete
- `wait` means wait for all background processes to finish before moving to the next line

Cluster Computing Introductory Example

Submit and check output once completed

```
[ikoutsou@front02 02]$ cat f32.txt
17:30:59.141054
 /nvme/scratch/edu19/data/05.txt: 562ddd6c
 /nvme/scratch/edu19/data/03.txt: 0c40e2d2
 /nvme/scratch/edu19/data/07.txt: 016db6c6
 /nvme/scratch/edu19/data/00.txt: 04d70552
 /nvme/scratch/edu19/data/09.txt: f1c7b711
 /nvme/scratch/edu19/data/08.txt: 700c1fd1
 /nvme/scratch/edu19/data/04.txt: f7bde74d
 /nvme/scratch/edu19/data/01.txt: 19708cd4
 /nvme/scratch/edu19/data/06.txt: 6f2cd2f1
 /nvme/scratch/edu19/data/02.txt: ed737a1c
17:31:00.866826
```

Cluster Computing Introductory Example

Submit and check output once completed

```
[ikoutsou@front02 02]$ cat f32.txt
17:30:59.141054
 /nvme/scratch/edu19/data/05.txt: 562ddd6c
 /nvme/scratch/edu19/data/03.txt: 0c40e2d2
 /nvme/scratch/edu19/data/07.txt: 016db6c6
 /nvme/scratch/edu19/data/00.txt: 04d70552
 /nvme/scratch/edu19/data/09.txt: f1c7b711
 /nvme/scratch/edu19/data/08.txt: 700c1fd1
 /nvme/scratch/edu19/data/04.txt: f7bde74d
 /nvme/scratch/edu19/data/01.txt: 19708cd4
 /nvme/scratch/edu19/data/06.txt: 6f2cd2f1
 /nvme/scratch/edu19/data/02.txt: ed737a1c
17:31:00.866826
```

- The 10 checksums should be identical to before

Cluster Computing Introductory Example

Submit and check output once completed

```
[ikoutsou@front02 02]$ cat f32.txt
17:30:59.141054
 /nvme/scratch/edu19/data/05.txt: 562ddd6c
 /nvme/scratch/edu19/data/03.txt: 0c40e2d2
 /nvme/scratch/edu19/data/07.txt: 016db6c6
 /nvme/scratch/edu19/data/00.txt: 04d70552
 /nvme/scratch/edu19/data/09.txt: f1c7b711
 /nvme/scratch/edu19/data/08.txt: 700c1fd1
 /nvme/scratch/edu19/data/04.txt: f7bde74d
 /nvme/scratch/edu19/data/01.txt: 19708cd4
 /nvme/scratch/edu19/data/06.txt: 6f2cd2f1
 /nvme/scratch/edu19/data/02.txt: ed737a1c
17:31:00.866826
```

- The 10 checksums should be identical to before
- The order is random. Which checksum completes first is undetermined. The only guarantee is that the last line in the script (the second `date +"%T.%6N`) is run after all 10 are complete (because of the preceding `wait`)

Cluster Computing Introductory Example

Submit and check output once completed

```
[ikoutsou@front02 02]$ cat f32.txt
17:30:59.141054
 /nvme/scratch/edu19/data/05.txt: 562ddd6c
 /nvme/scratch/edu19/data/03.txt: 0c40e2d2
 /nvme/scratch/edu19/data/07.txt: 016db6c6
 /nvme/scratch/edu19/data/00.txt: 04d70552
 /nvme/scratch/edu19/data/09.txt: f1c7b711
 /nvme/scratch/edu19/data/08.txt: 700c1fd1
 /nvme/scratch/edu19/data/04.txt: f7bde74d
 /nvme/scratch/edu19/data/01.txt: 19708cd4
 /nvme/scratch/edu19/data/06.txt: 6f2cd2f1
 /nvme/scratch/edu19/data/02.txt: ed737a1c
17:31:00.866826
```

- The 10 checksums should be identical to before
- The order is random. Which checksum completes first is undetermined. The only guarantee is that the last line in the script (the second `date +"%T.%6N`) is run after all 10 are complete (because of the preceding `wait`)
- All 10 checksums complete within ~1.7 seconds

Summary of Introductory Example

Outline of major points covered

- Job scheduling system for reserving and running on compute nodes
- Interactive versus batch jobs
- Modules system for selecting build environment
- Running programs concurrently on multiple cores and multiple nodes

Summary of Introductory Example

Outline of major points covered

- Job scheduling system for reserving and running on compute nodes
- Interactive versus batch jobs
- Modules system for selecting build environment
- Running programs concurrently on multiple cores and multiple nodes

Important points *not* covered

- Here we parallelized the execution of otherwise *scalar* programs

Summary of Introductory Example

Outline of major points covered

- Job scheduling system for reserving and running on compute nodes
- Interactive versus batch jobs
- Modules system for selecting build environment
- Running programs concurrently on multiple cores and multiple nodes

Important points *not* covered

- Here we parallelized the execution of otherwise *scalar* programs
- We have not covered implementing parallelism and concurrency *within* the programs, which is the standard approach in parallel computing
- Examples:
  - Multi-threading, using a shared memory paradigm. For example using OpenMP
  - Distributed memory parallelization, e.g. using MPI

Summary of Introductory Example

Outline of major points covered

- Job scheduling system for reserving and running on compute nodes
- Interactive versus batch jobs
- Modules system for selecting build environment
- Running programs concurrently on multiple cores and multiple nodes

Important points *not* covered

- Here we parallelized the execution of otherwise *scalar* programs
- We have not covered implementing parallelism and concurrency *within* the programs, which is the standard approach in parallel computing
- Examples:
  - Multi-threading, using a shared memory paradigm. For example using OpenMP
  - Distributed memory parallelization, e.g. using MPI

Up next: parallel processing on GPUs

GPU Programming Introduction

Outline

Some slides on:

- Review of GPU architecture
- Review of GPU programming and CUDA
- Some details of our training system, "Cyclone"

Outline

Some slides on:

- Review of GPU architecture
- Review of GPU programming and CUDA
- Some details of our training system, "Cyclone"

Hands-on — practical example on GPUs

Covering:

- GPU performance vs CPU performance
- Memory coalescing on GPUs

Outline

Some slides on:

- Review of GPU architecture
- Review of GPU programming and CUDA
- Some details of our training system, "Cyclone"

Hands-on — practical example on GPUs

Covering:

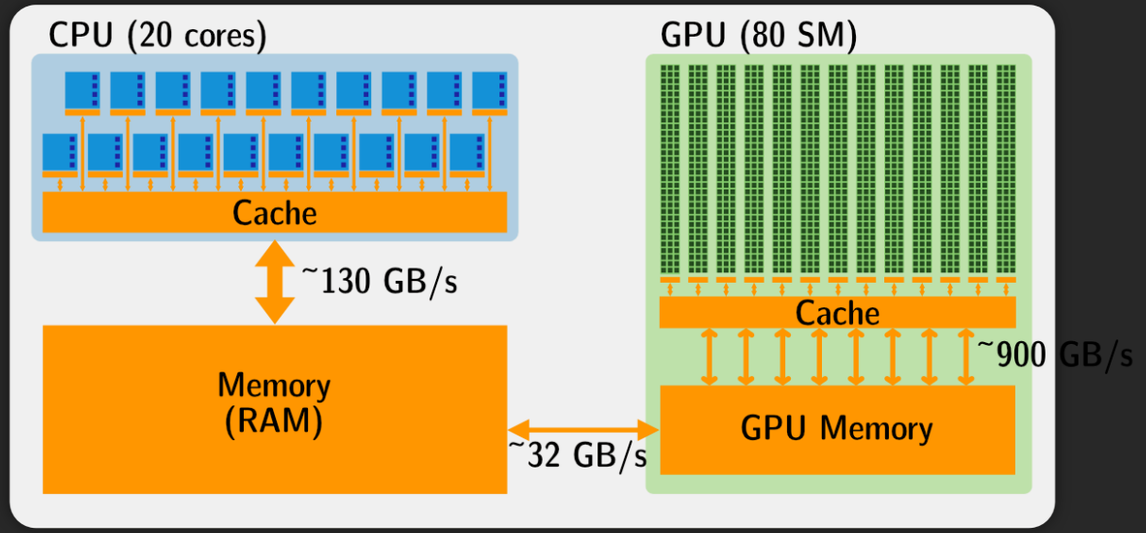
- GPU performance vs CPU performance
- Memory coalescing on GPUs

Not covering:

- Shared memory
- Warps and thread scheduling

GPU architecture

At a very high level:



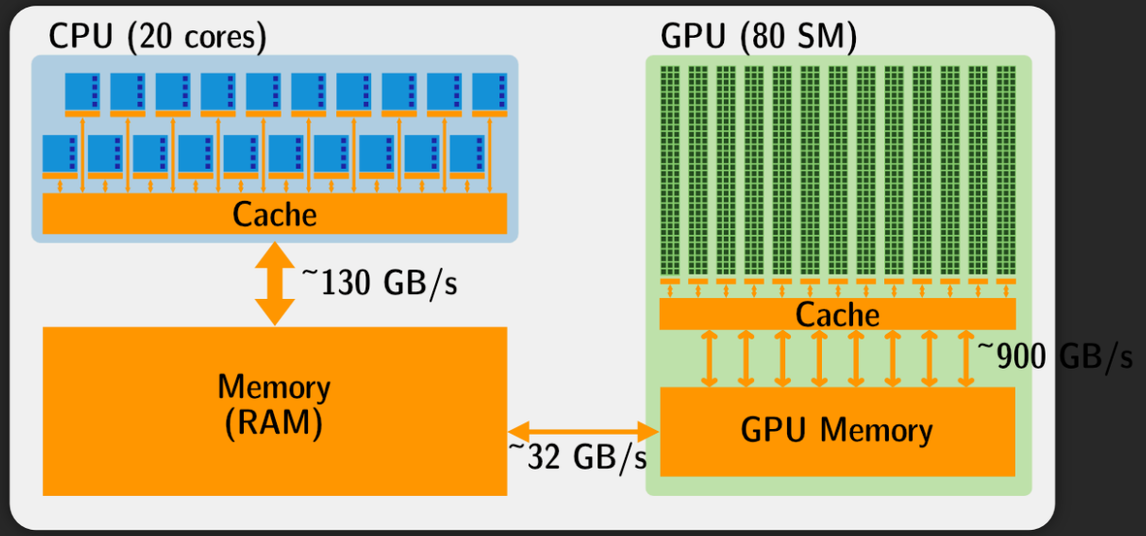
CPU

- 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

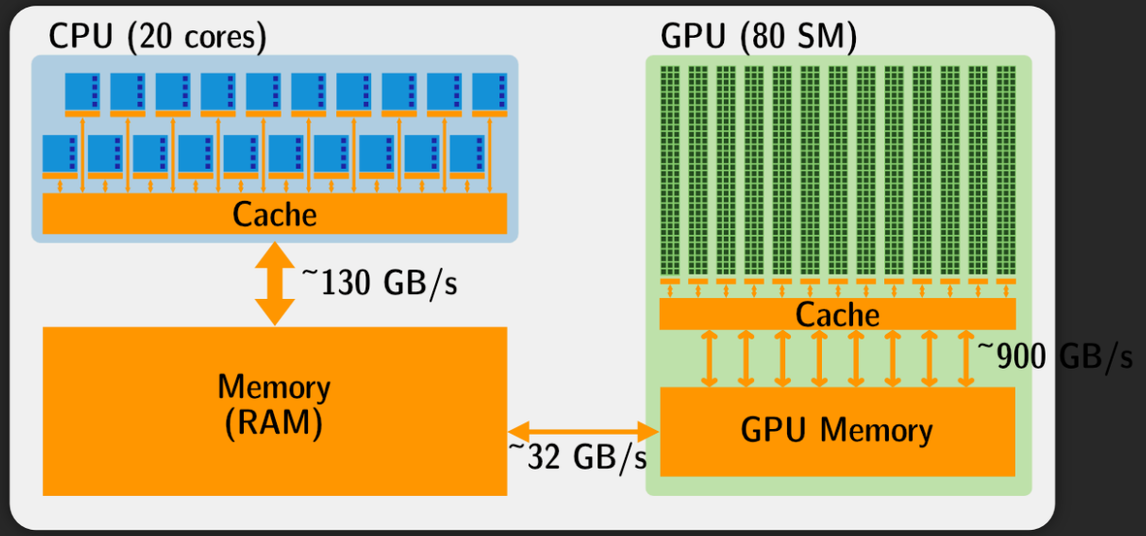


Some numbers from the GPU partition of our Cyclone cluster

NVIDIA V100 *Volta* GPUs

- 80 Streaming Multiprocessors (SM) per GPU
- 64 "cores" per SM
- GPU memory: 32 GBytes
- Memory bandwidth: 900 GB/s
- Peak performance: 7.8 Tflop/s (double precision)

GPU programming model



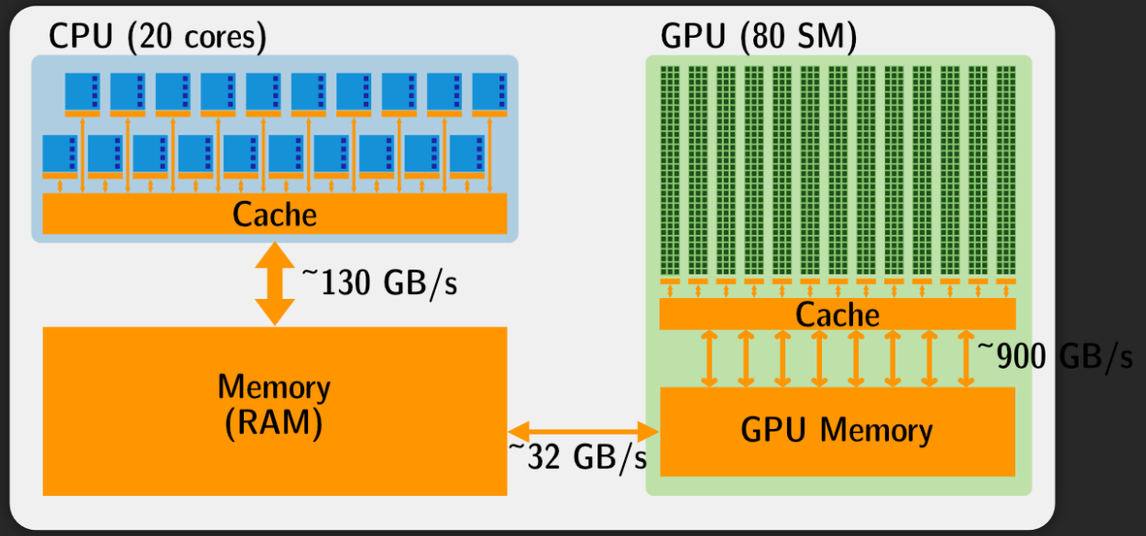
Some numbers from the GPU partition of our Cyclone cluster

NVIDIA V100 *Volta* GPUs

- 80 Streaming Multiprocessors (SM) per GPU
- 64 "cores" per SM
- GPU memory: 32 GBytes
- Memory bandwidth: 900 GB/s
- Peak performance: 7.8 Tflop/s (double precision)

We will come back to these numbers during the hands-on

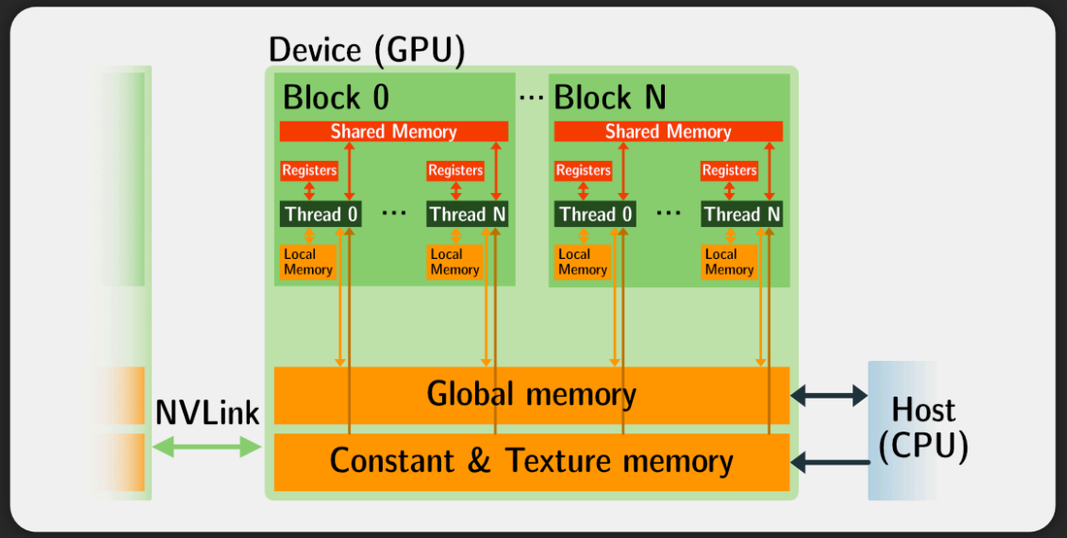
GPU programming model



"Offload" model of programming

- CPU starts program (runs `main()`)
- CPU copies data to GPU memory (over e.g. PCIe, ~32 GB/s)
- CPU dispatches "kernels" for execution on GPU
  - Kernels read/write to GPU memory (~900 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, ~32 GB/s)

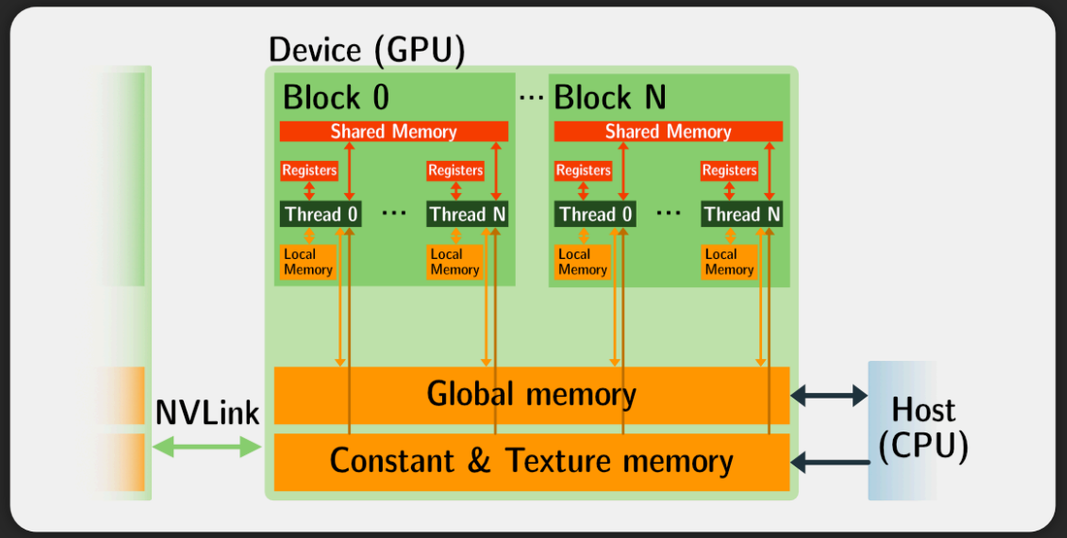
GPU programming model



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)
- Shared memory is limited. E.g. 96 KB per block for V100

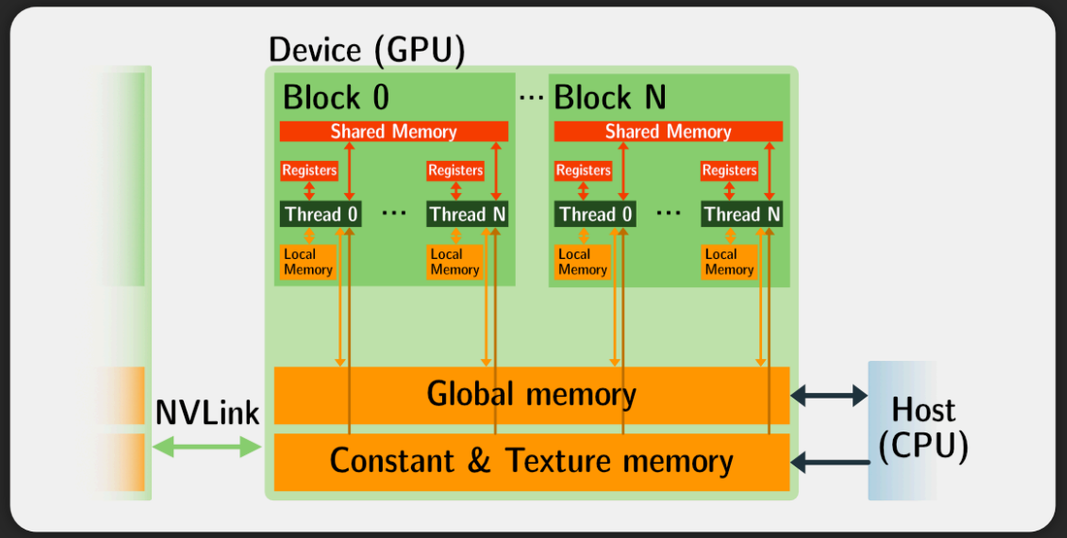
GPU programming model



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

- Threads per block: 1024 (max)
- Register memory (per block): 64 KB
- Shared memory (per block): 96 KB
- Also, max. 255 registers per thread

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 ≠ 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

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

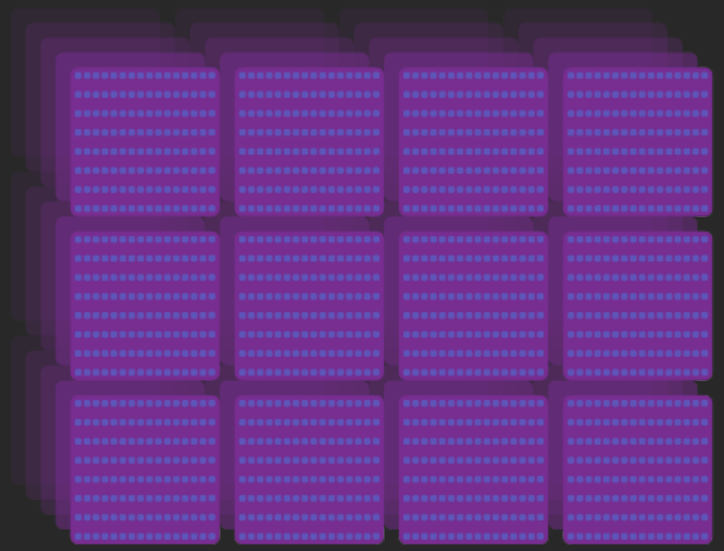
Example:

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

CUDA programming basics

Threads, blocks, grids

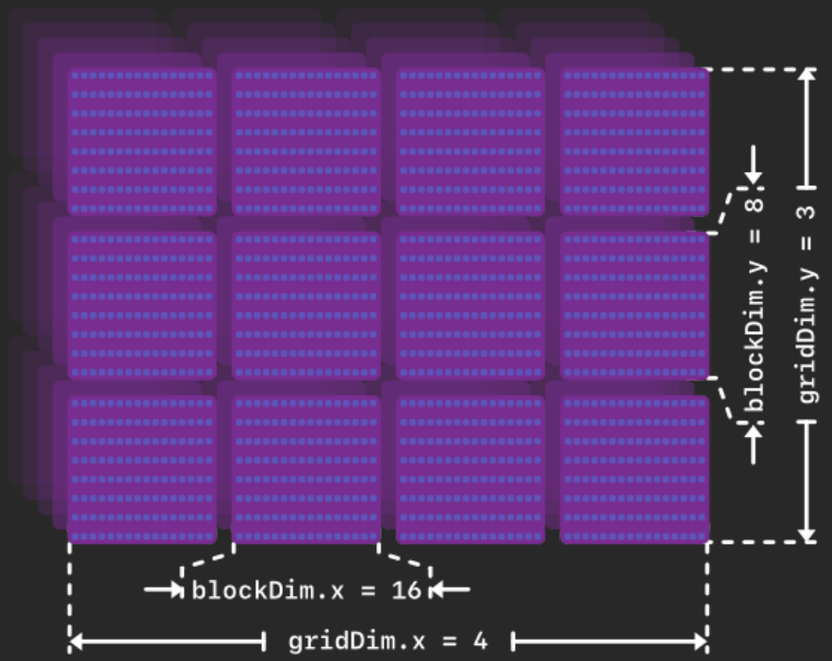
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

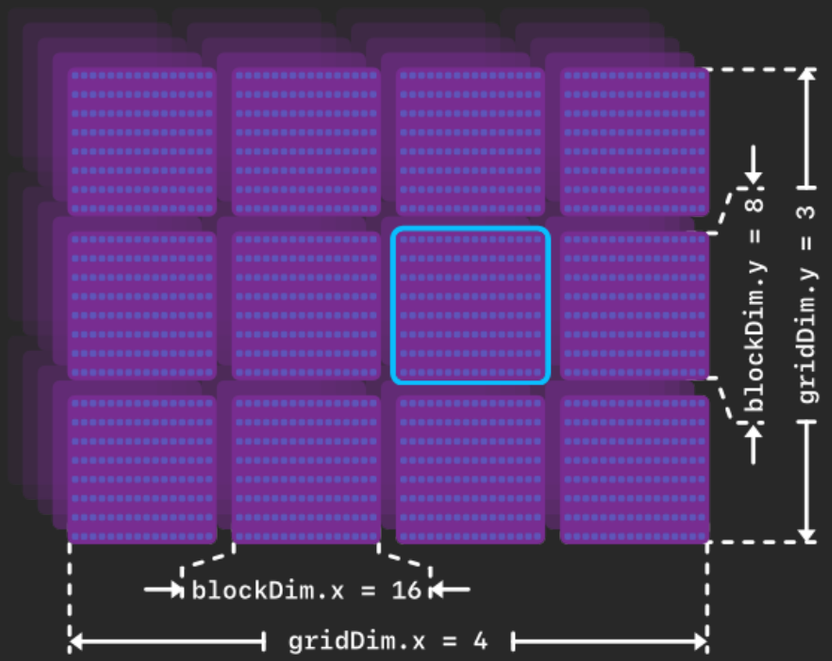
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

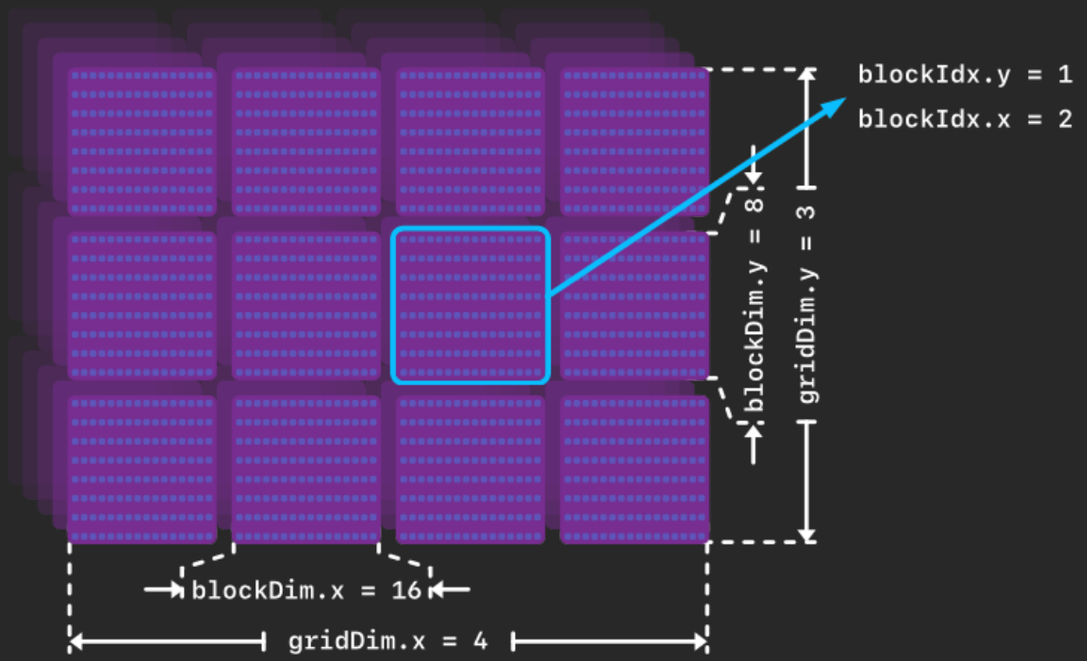
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

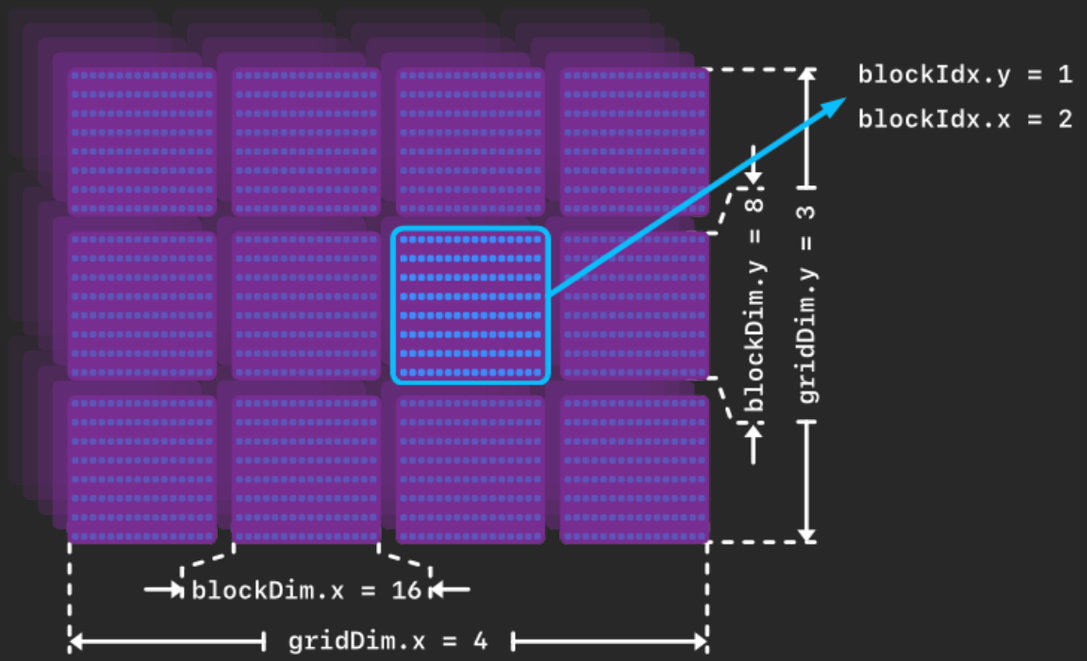
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

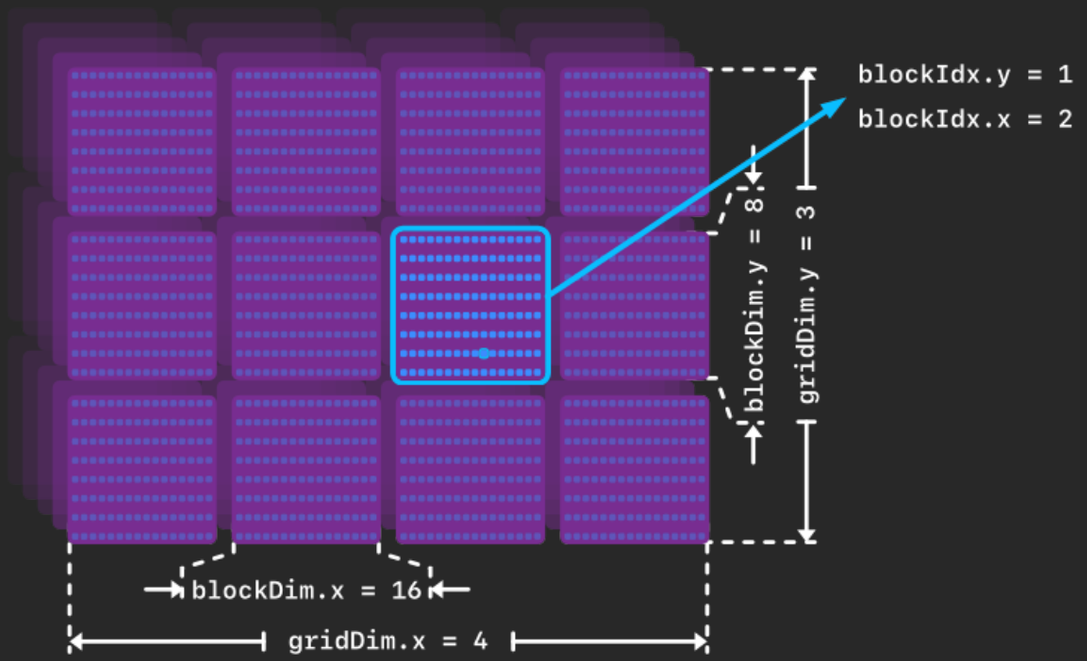
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

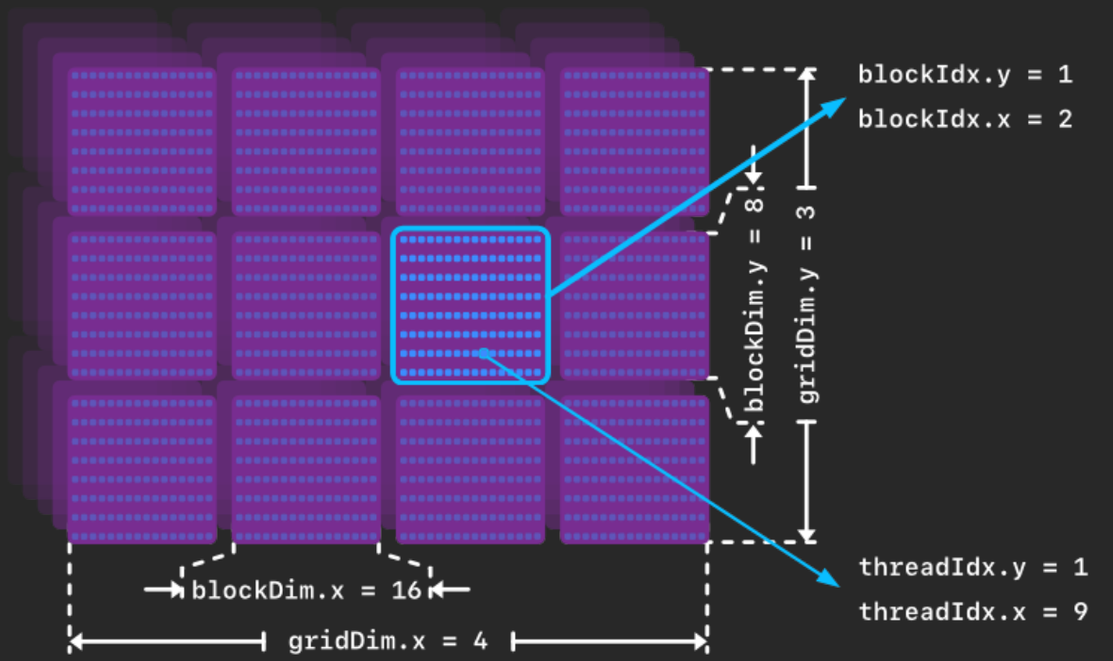
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



CUDA programming basics

Threads, blocks, grids

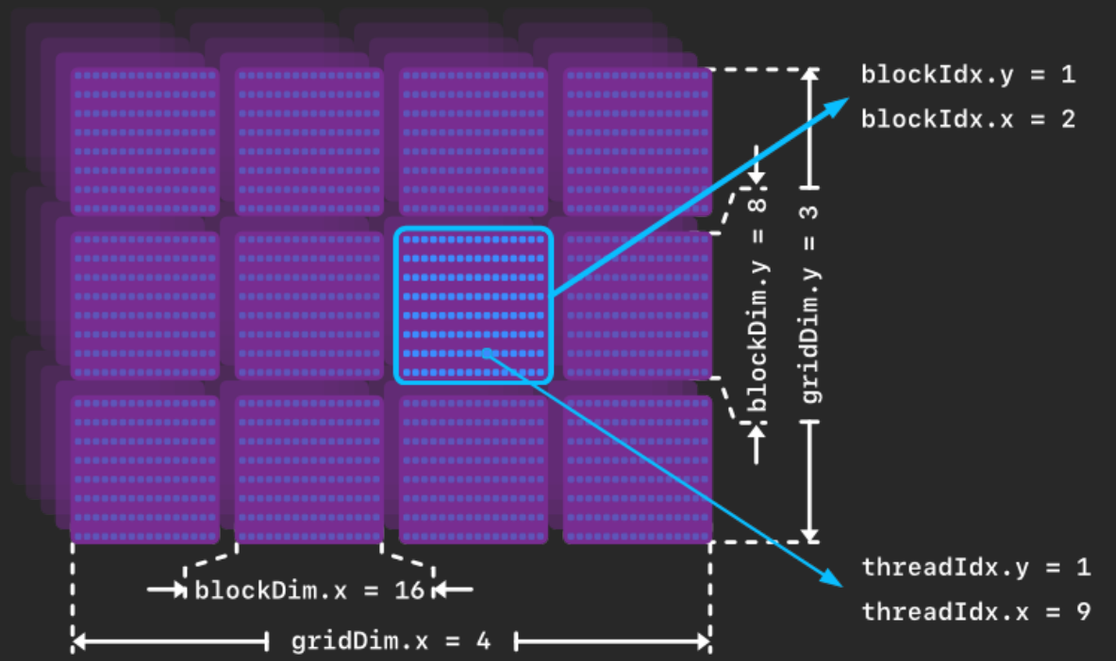
```
dim3 blcks( 4, 3, bz);
dim3 thrds(16, 8, tz);
func<<<blcks, thrds>>>( ... );
```



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, by example

Port a simple code to GPU and investigate performance

Sources:

```
[ikoutsou@front02 ~]$ ls -1 /nvme/scratch/edu19/03
axpy.cu
```

- `axpy.cu` implements a so-called "axpy" operation (a-times-x-plus-y):

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

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

CUDA, by example

Port a simple code to GPU and investigate performance

Sources:

```
[ikoutsou@front02 ~]$ ls -1 /nvme/scratch/edu19/03
axpy.cu
```

- `axpy.cu` implements a so-called "axpy" operation (a-times-x-plus-y):

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

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

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 20 cores (per socket) of the Cyclone cluster nodes

CUDA, by example

Port a simple code to GPU and investigate performance

Sources:

```
[ikoutsou@front02 ~]$ ls -1 /nvme/scratch/edu19/03
axpy.cu
```

- `axpy.cu` implements a so-called "axpy" operation (a-times-x-plus-y):

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

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

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 20 cores (per socket) of the Cyclone cluster nodes
- We will proceed step-by-step, to port this simple application to the GPU using CUDA

CUDA, by example

Port a simple code to GPU and investigate performance

Sources:

```
[ikoutsou@front02 ~]$ ls -1 /nvme/scratch/edu19/03
axpy.cu
```

- `axpy.cu` implements a so-called "axpy" operation (a-times-x-plus-y):

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

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

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 20 cores (per socket) of the Cyclone cluster 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, by example

Port a simple code to GPU and investigate performance

Sources:

```
[ikoutsou@front02 ~]$ ls -1 /nvme/scratch/edu19/03
axpy.cu
```

- `axpy.cu` implements a so-called "axpy" operation (a-times-x-plus-y):

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

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

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 20 cores (per socket) of the Cyclone cluster 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, by example

Port a simple code to GPU and investigate performance

Sources:

```
[ikoutsou@front02 ~]$ ls -1 /nvme/scratch/edu19/03
axpy.cu
```

- `axpy.cu` implements a so-called "axpy" operation (a-times-x-plus-y):

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

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

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 20 cores (per socket) of the Cyclone cluster 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, by example

Port a simple code to GPU and investigate performance

Sources:

```
[ikoutsou@front02 ~]$ ls -1 /nvme/scratch/edu19/03
axpy.cu
```

- `axpy.cu` implements a so-called "axpy" operation (a-times-x-plus-y):

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

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

- Currently Implements this on the **CPU**. Uses OpenMP to multi-thread over the 20 cores (per socket) of the Cyclone cluster 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: `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: `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 the shared space:

```
[ikoutsou@front02 eurocc-training]$ mkdir 03
[ikoutsou@front02 eurocc-training]$ cd 03
[ikoutsou@front02 03]$ cp /nvme/scratch/edu19/03/axpy.cu .
```

CUDA Example

- Copy the exercise from the shared space:

```
[ikoutsou@front02 eurocc-training]$ mkdir 03
[ikoutsou@front02 eurocc-training]$ cd 03
[ikoutsou@front02 03]$ cp /nvme/scratch/edu19/03/axpy.cu .
```

- Compile with `nvcc` including OpenMP:

```
[ikoutsou@front02 03]$ module load gompi
[ikoutsou@front02 03]$ module load CUDA
[ikoutsou@front02 03]$ nvcc -O3 -arch sm_70 -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 the shared space:

```
[ikoutsou@front02 eurocc-training]$ mkdir 03
[ikoutsou@front02 eurocc-training]$ cd 03
[ikoutsou@front02 03]$ cp /nvme/scratch/edu19/03/axpy.cu .
```

- Compile with `nvcc` including OpenMP:

```
[ikoutsou@front02 03]$ module load gompi
[ikoutsou@front02 03]$ module load CUDA
[ikoutsou@front02 03]$ nvcc -O3 -arch sm_70 -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 03] export OMP_PROC_BIND="close"
[ikoutsou@front02 03] export OMP_PLACES="cores"
[ikoutsou@front02 03] export OMP_NUM_THREADS=20
[ikoutsou@front02 03] srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0085 sec   P =  15.726 Gflop/s   B =  94.354 GB/s
 CPU: nthr =   20   t0 = 0.0077 sec   P =  17.470 Gflop/s   B = 104.819 GB/s
 Diff = 0.000000e+00
```

CUDA Example

- Copy the exercise from the shared space:

```
[ikoutsou@front02 eurocc-training]$ mkdir 03
[ikoutsou@front02 eurocc-training]$ cd 03
[ikoutsou@front02 03]$ cp /nvme/scratch/edu19/03/axpy.cu .
```

- Compile with `nvcc` including OpenMP:

```
[ikoutsou@front02 03]$ module load gompi
[ikoutsou@front02 03]$ module load CUDA
[ikoutsou@front02 03]$ nvcc -O3 -arch sm_70 -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 03] export OMP_PROC_BIND="close"
[ikoutsou@front02 03] export OMP_PLACES="cores"
[ikoutsou@front02 03] export OMP_NUM_THREADS=20
[ikoutsou@front02 03] srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0085 sec   P =  15.726 Gflop/s   B =  94.354 GB/s
 CPU: nthr =   20   t0 = 0.0077 sec   P =  17.470 Gflop/s   B = 104.819 GB/s
 Diff = 0.000000e+00
```

- Compare ~100 GB/s achieved vs ~130 GB/s peak memory bandwidth

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 03]$ nvcc -arch sm_70 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
```

- Run as before (I'm assuming `OMP_BIND`, `OMP_PLACES`, and `OMP_NUM_THREADS` were set before):

```
[ikoutsou@front02 03]$ srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0091 sec   P =  14.816 Gflop/s   B =  88.896 GB/s
 GPU:              t0 = 0.0000 sec   P = 2916.839 Gflop/s   B = 17501.035 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.
- 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 03]$ nvcc -arch sm_70 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 03]$ srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0092 sec   P =  14.622 Gflop/s   B =  87.732 GB/s
 GPU:               t0 = 3.5689 sec   P =   0.038 Gflop/s   B =   0.226 GB/s
 Diff = 1.021564e-15
```

CUDA Example

- Compile and run again:

```
[ikoutsou@front02 03]$ nvcc -arch sm_70 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 03]$ srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0092 sec   P =  14.622 Gflop/s   B =  87.732 GB/s
 GPU:               t0 = 3.5689 sec   P =   0.038 Gflop/s   B =   0.226 GB/s
 Diff = 1.021564e-15
```

- This performance is of course extremely poor;

CUDA Example

- Compile and run again:

```
[ikoutsou@front02 03]$ nvcc -arch sm_70 -O3 -Xcompiler -fopenmp -o axpy axpy.cu
[ikoutsou@front02 03]$ srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0092 sec   P =  14.622 Gflop/s   B =  87.732 GB/s
 GPU:               t0 = 3.5689 sec   P =   0.038 Gflop/s   B =   0.226 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:

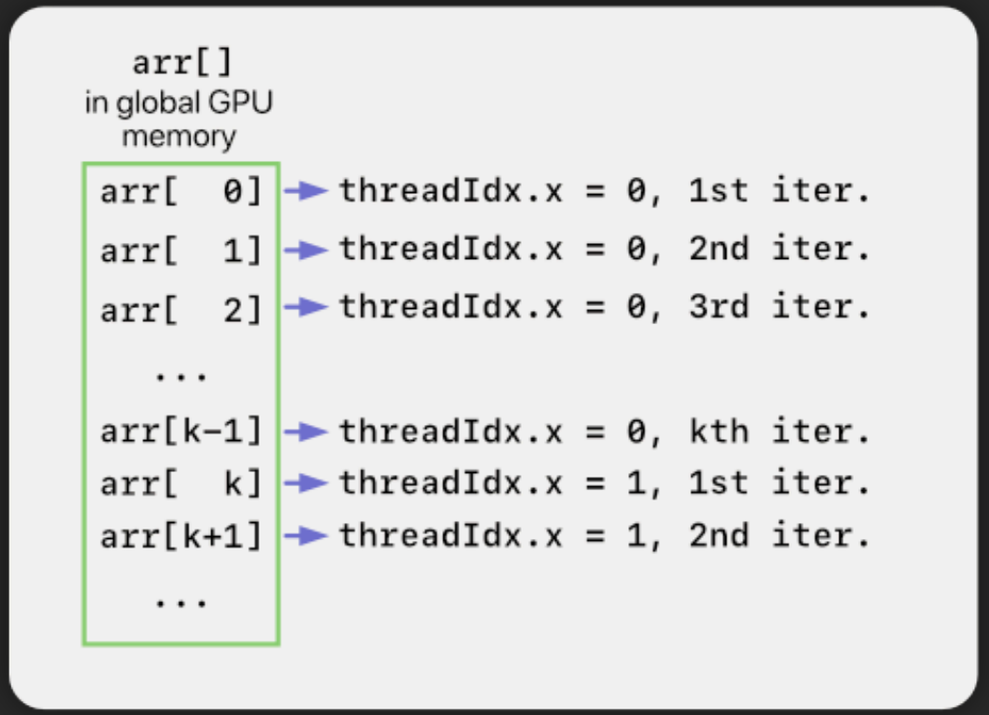
```
[ikoutsou@front02 03]$ srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0092 sec   P =  14.606 Gflop/s   B =  87.637 GB/s
 GPU:               t0 = 0.1864 sec   P =   0.720 Gflop/s   B =   4.321 GB/s
 Diff = 1.021564e-15
```

- 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



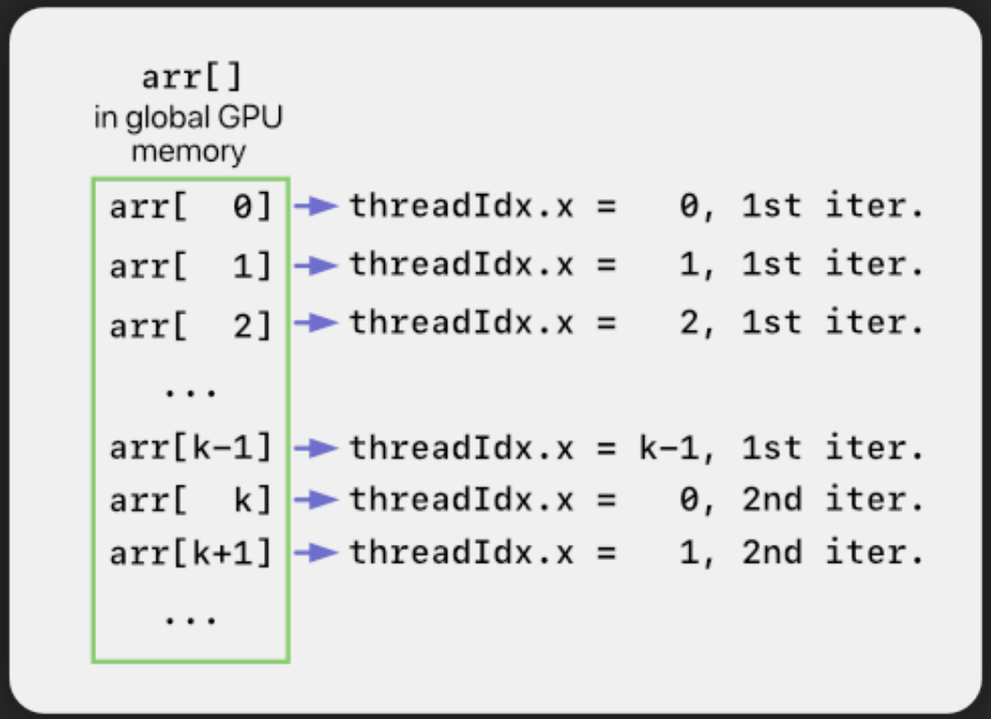
This represents the order by which elements are accessed currently

- 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 03]$ srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19  ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0092 sec   P =  14.662 Gflop/s   B =  87.972 GB/s
 GPU:              t0 = 0.0619 sec   P =   2.168 Gflop/s   B =  13.009 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

```
    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 03]$ srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0091 sec   P =  14.701 Gflop/s   B =  88.204 GB/s
 GPU:              t0 = 0.0011 sec   P = 120.701 Gflop/s   B = 724.207 GB/s
 Diff = 1.021564e-15
```

- ~720 GB/s is ~80% of peak bandwidth (which is 900 GB/s)

CUDA Example

Blocks and threads

- Compile and run:

```
[ikoutsou@front02 03]$ srun -n 1 --cpus-per-task=20 -p gpu -A edu19 --gres=gpu:1 --reservation=edu19 ./axpy $((1024*1024*64))
 CPU: nthr =   20   t0 = 0.0091 sec   P =  14.701 Gflop/s   B =  88.204 GB/s
 GPU:              t0 = 0.0011 sec   P = 120.701 Gflop/s   B = 724.207 GB/s
 Diff = 1.021564e-15
```

- ~720 GB/s is ~80% of peak bandwidth (which is 900 GB/s)
- Try varying the number of threads per block. E.g. with 512 threads I got ~735 GB/s.

Summary of CUDA Example

Main points covered

- Memory allocation and data transfer to and from GPU
- Launching kernels
- Memory access patterns of GPU threads

Summary of CUDA Example

Main points covered

- Memory allocation and data transfer to and from GPU
- Launching kernels
- Memory access patterns of GPU threads

Major points that were not covered

- *Shared memory*: memory that can be shared between GPU threads in the same block
- *Thread scheduling*
  - Thread synchronization
  - Thread warps
  - Considerations when using control flow constructs (e.g. `if`-statements)