



# GPU Computing on the FASRC Cluster

Plamen Krastev, PhD

Harvard - FAS Research Computing

# Objectives

- To advise you on the best practices for running GPU applications on the FASRC cluster
- To provide the basic knowledge required for building your own GPU apps

# FAS Research Computing (FASRC)

Faculty of Arts and Sciences (FAS) department that handles non-enterprise IT requests from researchers.

- **RC Primary Services:**
  - Cannon Supercomputing Environment
  - Lab Storage
  - Instrument Computing Support
  - Hosted Machines (virtual or physical)
- **RC Staff:**
  - 25+ staff with backgrounds ranging from systems administration to development-operations to PhD research scientists
  - Supporting 600+ research groups and 5500+ users across FAS, SEAS, HSPH, HBS, GSE
  - For Bioinformatics researchers the Harvard Informatics group is closely tied to RC and is there to support the specific problems for that domain



# Cannon Cluster

## Compute:

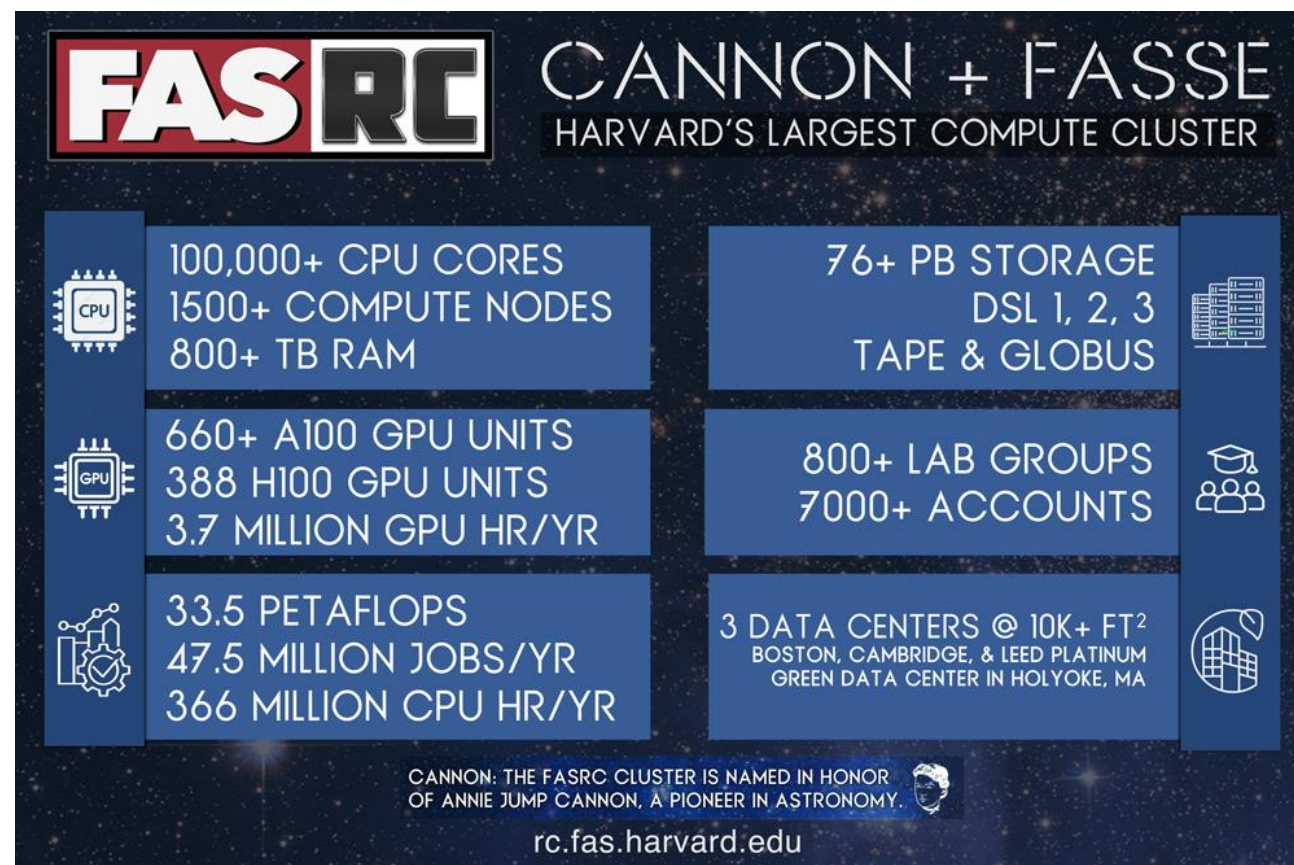
- 100,000+ compute cores
- Cores/node: 48 to 112
- Memory/node: 184GB to 2000GB
- GPUs/node: 4 (A100, H100)

## Software:

- Operating System Rocky 8
- Slurm job manager
- Spack software package manager

## Interconnect:

- 2 underlying networks connecting 3 data centers
- TCP/IP network
- Low-latency 200 GB/s HDR InfiniBand (IB) and 56 GB/s FDR IB network:
  - inter-node parallel computing
  - fast access to Lustre mounted storage

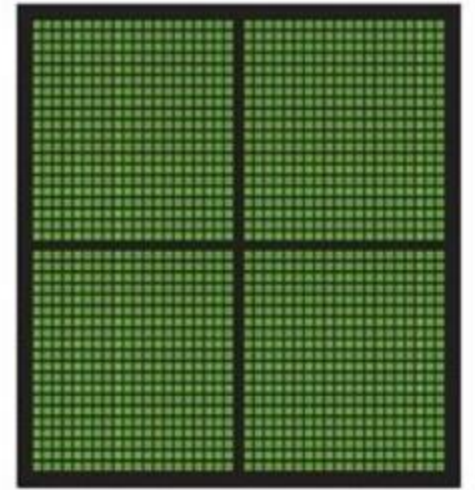


# What is a GPGPU?

**General-Purpose Graphics Processing Unit (GPGPU)** is a graphics processing unit (GPU) that is programmed for purposes *beyond* graphics processing, such as performing computations typically conducted by a Central Processing Unit (CPU).



CPU  
MULTIPLE CORES



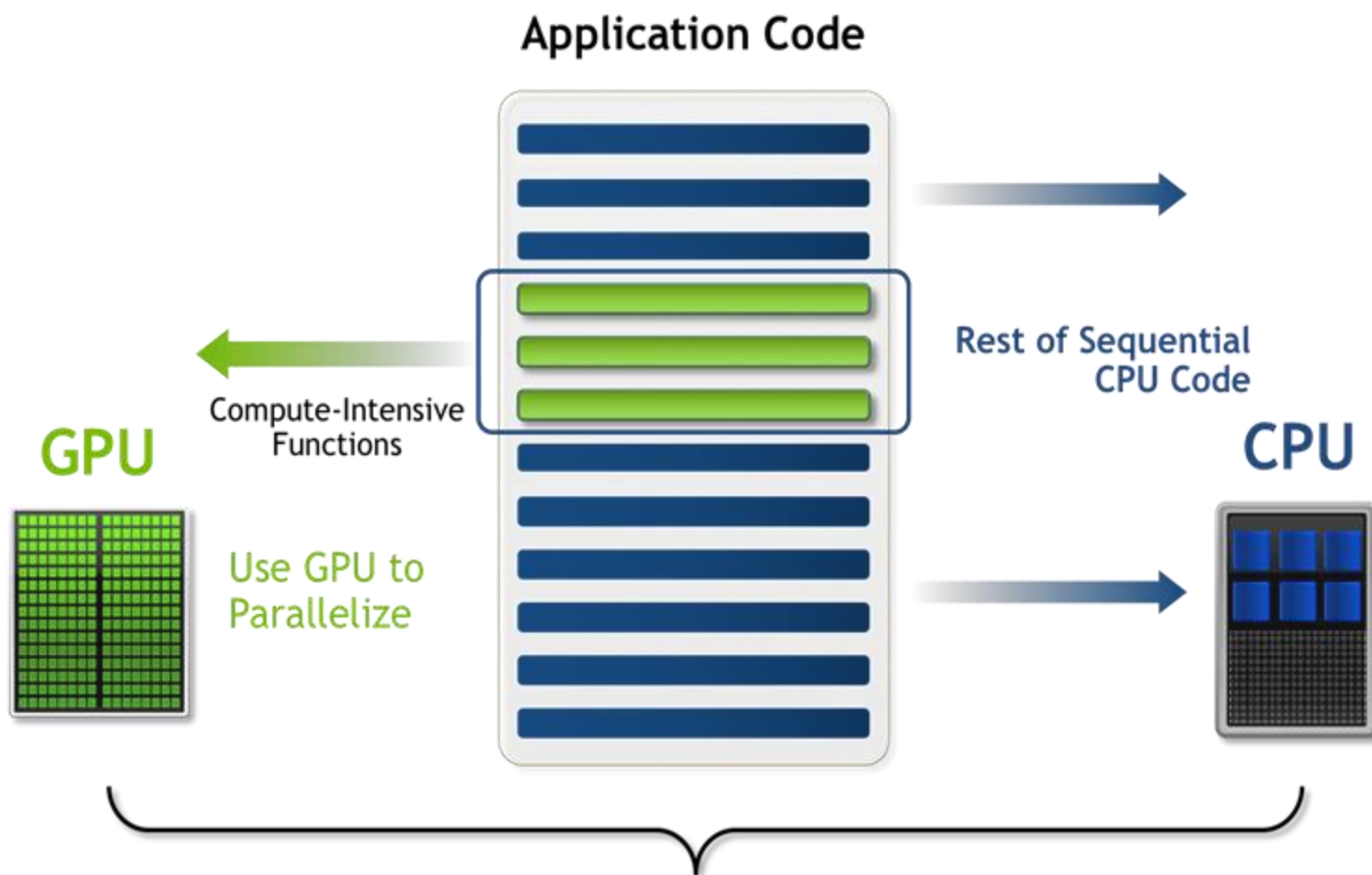
GPU  
THOUSANDS OF CORES

# GPU vs CPU

| <b>CPU</b>                             | <b>GPU</b>                             |
|--|--|
| Central Processing Unit                | Graphics Processing Unit               |
| Several cores                          | Many cores                             |
| Low latency                            | High throughput                        |
| Good for serial processing             | Good for parallel processing           |
| Can do a handful of operations at once | Can do thousands of operations at once |

<https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/>

# Heterogeneous Computing



# Using GPGPUs

- GPU enabled applications require a parallel computing platform and application programming interface (API) that allows software developers and software engineers to build algorithms to modify their application and map compute-intensive kernels to the GPU
- GPGPU supports several types of memory in a memory hierarchy for designers to optimize their programs
- GPGPU memory is used for transferring data between device and host - shared memory is an efficient way for threads in the same block to share their runtime and data



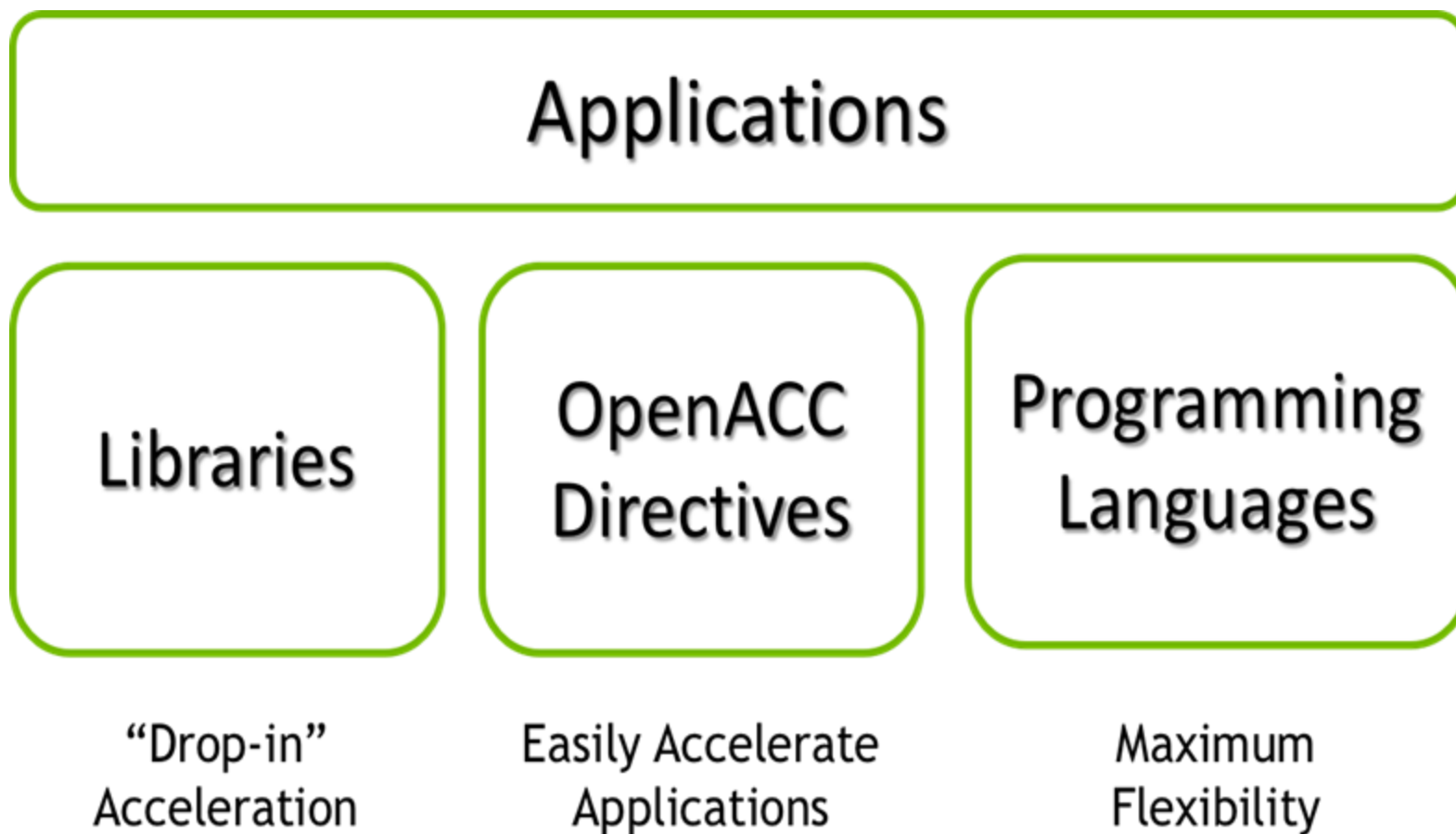
# GPU accelerated applications

Large number of applications and frameworks in different scientific and engineering domains have been adapted to take advantage of GPUs.

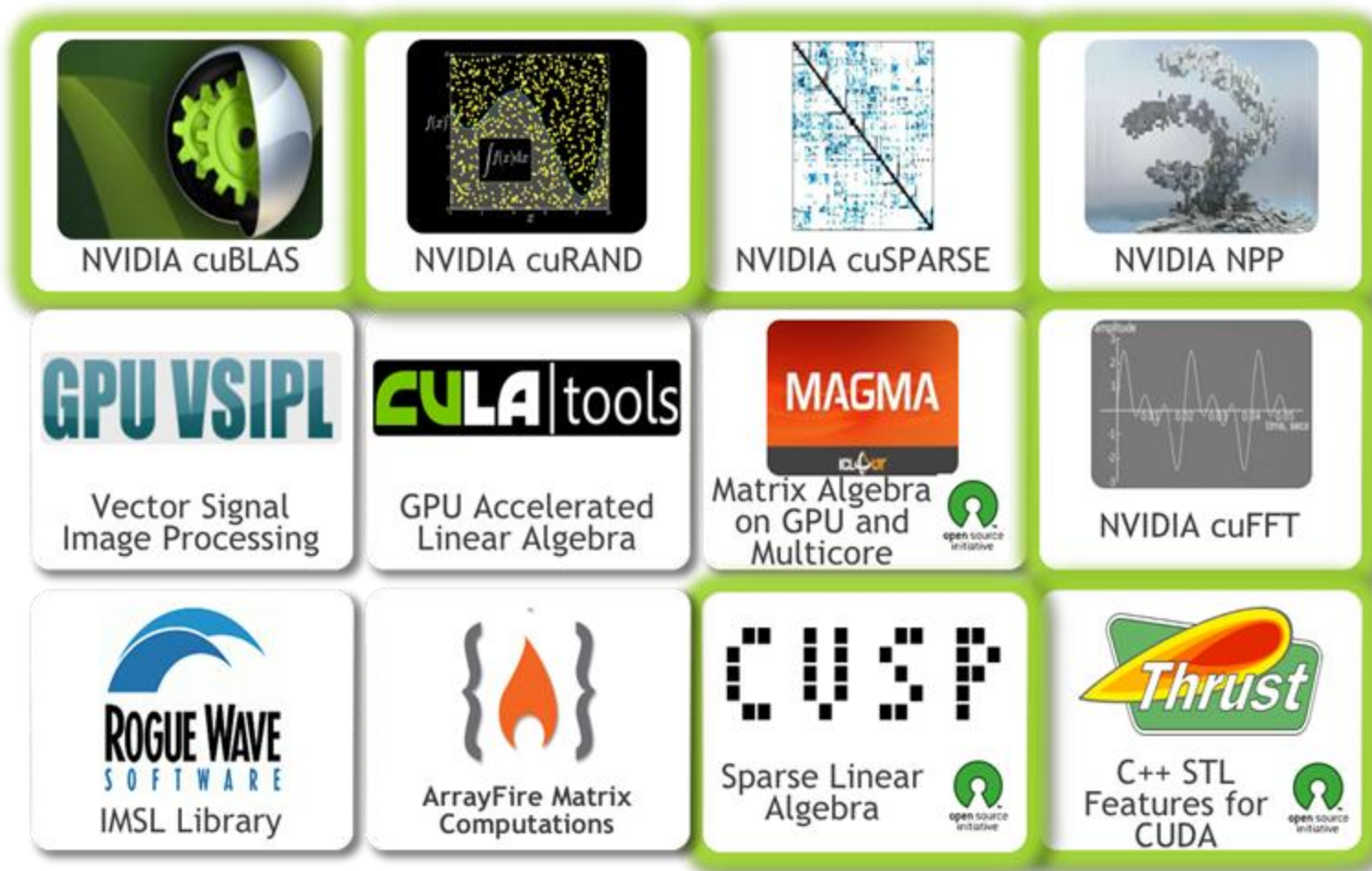
## Examples:

- Deep Learning frameworks (TensorFlow, PyTorch, JAX, etc.) have transparent Python APIs  
Typically, these can be easily installed via `pip` or `conda/mamba`
- MATLAB supports GPU computing via PCT
- Specific applications built with CUDA
- Containers available via Nvidia Cloud Computing platform  
<https://ngc.nvidia.com/catalog/all>

# Three Ways to Accelerate Your Applications



# Some GPU-Accelerated Libraries



# Drop-in Library: cuBLAS (1)

## Example: SAXPY

Stands for “Single-Precision  $A * X$  Plus  $Y$ ” (a function in the standard BLAS library)

### CPU version:

```
#include <stdio.h>
#include <gsl/gsl_cblas.h>

int main(){
    const int n = 5;
    const float alpha = 2.0;
    float x[] = {1.0, 2.0, 3.0, 4.0, 5.0};
    float y[] = {2.0, 4.0, 6.0, 8.0, 10.0};

    // Perform SAXPY operation
    cblas_saxpy(n, alpha, x, 1, y, 1);

    // Print final values
    printf("SAXPY result: ");
    for (int i = 0; i < n; i++) {
        printf("%f ", y[i]);
    }
    printf("\n");

    return 0;
}
```



# Drop-in Library: cuBLAS (2)

## Example: SAXPY

Stands for “Single-Precision  $A * X$  Plus  $Y$ ” (a function in the standard BLAS library)

### GPU version:

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>

int main(){
    ...
    // Initialize cuBLAS context
    cublasHandle_t handle;
    cublasCreate(&handle);

    // Allocate memory on device
    float *d_x, *d_y;
    cudaMalloc((void**)&d_x, n * sizeof(float));
    cudaMalloc((void**)&d_y, n * sizeof(float));
    // Copy data to device
    cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyHostToDevice);

    // Perform SAXPY operation
    cublasSaxpy(handle, n, &alpha, d_x, 1, d_y, 1);

    // Copy data back to host
    cudaMemcpy(y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);

    // Destroy cuBLAS context
    cublasDestroy(handle);

    // Free memory on device
    cudaFree(d_x);
    cudaFree(d_y);
    ...
}
```

# Drop-in Library: cuBLAS (3)

## Compiling

**CPU version:** saxpy\_blas.c

```
# Load software modules
```

```
$ module load gcc/14.2.0-fasrc01
```

```
# Compile the code
```

```
$ gcc -o saxpy_blas.x saxpy_blas.c -lgslcblas
```

**GPU version:** saxpy\_cublas.c

```
# Load software modules
```

```
$ module load cuda/12.4.1-fasrc01 gcc/14.2.0-fasrc01
```

```
# Compile the code
```

```
$ gcc -o saxpy_cublas.x saxpy_cublas.c -lcudart -lcublas
```

# OpenACC

- **OpenACC** (for Open Accelerators) is a programming standard for parallel computing on accelerators (mostly on NIVIDIA GPU)
- It is designed to simplify GPU programming
- The basic approach is to insert special comments (directives) into the code to offload computation onto GPUs and parallelize the code at the level of GPU (CUDA) cores
- It is possible for programmers to create an efficient parallel OpenACC code with only minor modifications to a serial CPU code

# OpenACC

## OpenACC COMPILER DIRECTIVES

### *Parallel C Code*

```
void saxpy(int n,  
          float a,  
          float *x,  
          float *y)  
{  
#pragma acc kernels  
  for (int i = 0; i < n; ++i)  
    y[i] = a*x[i] + y[i];  
}  
  
...  
// Perform SAXPY on 1M elements  
saxpy(1<<20, 2.0, x, y);  
...
```

### *Parallel Fortran Code*

```
subroutine saxpy(n, a, x, y)  
  real :: x(:), y(:), a  
  integer :: n, i  
!$acc kernels  
  do i=1,n  
    y(i) = a*x(i)+y(i)  
  enddo  
!$acc end kernels  
end subroutine saxpy  
  
...  
! Perform SAXPY on 1M elements  
call saxpy(2**20, 2.0, x_d, y_d)  
...
```

<http://developer.nvidia.com/openacc> or <http://openacc.org>



# OpenACC

## Compiling

**GPU version:** `example_acc.f90`

```
# Load software modules (NVIDIA HPC SDK)
```

```
$ module load nvhpc/24.11-fasrc01
```

```
# Compile command
```

```
$ nvfortran -o example_acc.x example_acc.f90 -acc
```

[https://github.com/fasrc/User\\_Codes/tree/master/GPU\\_Computing/OpenACC](https://github.com/fasrc/User_Codes/tree/master/GPU_Computing/OpenACC)

# Using Programming Languages

- **Numerical frameworks:** MATLAB, Mathematica
- **Fortran:** OpenACC, CUDA Fortran, OpenCL/CLFORTRAN
- **C:** OpenACC, CUDA C, OpenCL
- **C++:** CUDA C++, Thrust, OpenCL C++
- **Python:** PyCUDA / Numba, Copperhead, PyOpenCL

# Compute Unified Device Architecture (CUDA)

- CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements for the execution of compute kernels
- Designed to work with programming languages such as C, C++, and Fortran
- CUDA is an accessible platform, requiring no advanced skills in graphics programming, and available to software developers through CUDA-accelerated libraries and compiler directives
- CUDA-capable devices are typically connected with a host CPU and the host CPUs are used for data transmission and kernel invocation for CUDA devices
- The CUDA Toolkit includes GPU-accelerated libraries, a compiler, programming guides, API references, and the CUDA runtime

<https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html>

# Using Programming Languages

## CUDA C

```
void saxpy(int n, float a,
           float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```



# Using Programming Languages

## Compiling

### CUDA C

```
# Load software modules
$ module load cuda/12.4.1-fasrc01

# Compile command
$ nvcc -o saxpy.x saxpy.cu
```

[https://github.com/fasrc/User\\_Codes/tree/master/GPU\\_Computing/CUDA](https://github.com/fasrc/User_Codes/tree/master/GPU_Computing/CUDA)

# Using Programming Languages

## PYTHON

### *Standard Python*

```
import numpy as np

def saxpy(a, x, y):
    return [a * xi + yi
            for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)

cpu_result = saxpy(2.0, x, y)
```

<http://numpy.scipy.org>

### *Numba Parallel Python*

```
import numpy as np
from numba import vectorize

@vectorize(['float32(float32, float32,
float32)'], target='cuda')
def saxpy(a, x, y):
    return a * x + y

N = 1048576

# Initialize arrays
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays onGPU
C = saxpy(2.0, X, Y)
```

<https://numba.pydata.org>

# GPU Computing on the FASRC Cluster

The FASRC cluster has nodes with NVIDIA general purpose graphics processing units (GPGPU).

It is possible to use Nvidia CUDA tools for running computational work, and in certain use cases achieve very significant speedups.

## GPU partitions:

- `gpu` partition: 36 nodes with 4 A100 GPUs per node
- `gpu_test` partition: 12 nodes with 8 A100 MIG GPUs per node
- `gpu_requeue` partition: nodes are owned by various research groups and are available when idle

Your lab may have access to other partitions with GPUs

<https://docs.rc.fas.harvard.edu/kb/gpgpu-computing-on-the-cluster/>

# Running GPU jobs

## Interactive Jobs:

```
# --- Start an interactive session on a partition with GPUs, e.g.,
[login-node]$ salloc -p gpu_test --gres=gpu:1 --mem=4G -N 1 -t 60
```

```
# --- Check CUDA runtime (use nvidia-smi or nvidia-smi)
```

```
[gpu-compute-node]$ nvidia-smi
```

```
+-----+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15          CUDA Version: 12.4          |
+-----+-----+-----+-----+-----+-----+
| GPU   Name                               Persistence-M | Bus-Id              Disp.A | Volatile Uncorr. ECC |
| Fan   Temp   Perf              Pwr:Usage/Cap |      Memory-Usage   | GPU-Util  Compute M. |
|                               |                      |              MIG M. |
+-----+-----+-----+-----+-----+-----+
|    0   NVIDIA A100-SXM4-40GB                On      | 00000000:31:00.0 Off  |           On         |
| N/A    24C    P0              40W / 400W |  74MiB / 40960MiB   |    N/A    Default   |
|                               |                      |              Enabled  |
+-----+-----+-----+-----+-----+-----+
```

```
+-----+
| Processes:                               |
|  GPU   GI    CI        PID   Type   Process name                      GPU Memory |
|        ID    ID                                   |              Usage                    |
+-----+-----+-----+-----+-----+-----+
| No running processes found              |
+-----+
```

# Running GPU jobs

```
# --- Check available CUDA versions
```

```
$ module available cuda/
```

```
----- /n/sw/helmod-rocky8/modulefiles/Core -----  
  cuda/9.1.85-fasrc01      cuda/11.8.0-fasrc01      cuda/12.2.0-fasrc01  
  cuda/11.3.1-fasrc01     cuda/12.0.1-fasrc01     cuda/12.4.1-fasrc01 (D)  
...
```

```
# --- Load required modules, e.g.,
```

```
$ module load cuda/12.4.1-fasrc01
```

```
$ which nvcc
```

```
/n/sw/helmod-rocky8/apps/Core/cuda/12.4.1-fasrc01/cuda/bin/nvcc
```

```
# --- Using CUDA-dependent software modules, e.g.,
```

```
$ module load cuda/12.4.1-fasrc01 cudnn/9.5.1.17_cuda12-fasrc01
```

# Running GPU jobs

## Example: Running tensorflow on a GPU node with Singularity:

```
# --- Start an interactive session on a partition with GPUs, e.g.,
[login-node ]$ salloc -p gpu_test --gres=gpu:1 --mem=4G -N 1 -t 60
# --- cd to your SCRATCH folder ---
[compute-node]$ cd $SCRATCH/your_lab/your_user/
# --- Pull the latest TF GPU version from the Docker registry ---
# We could use a local installation of python and TF, but we'll use singularity
[compute-node]$ singularity pull --name tf213_gpu.simg \
> docker://tensorflow/tensorflow:2.13.0-gpu
# --- Get examples from keras.io ---
[compute-node]$ git clone https://github.com/keras-team/keras-io.git
# --- Execute the code ---
[compute-node]$ singularity exec --nv tf213_gpu.simg python \
./keras-io/examples/vision/mnist_convnet.py
... (omitted output)
Test loss: 0.023846622556447983
Test accuracy: 0.9919000267982483
```

<https://docs.rc.fas.harvard.edu/kb/singularity-on-the-cluster/>

[https://github.com/fasrc/User\\_Codes/tree/master/Al/TensorFlow#gpu-version](https://github.com/fasrc/User_Codes/tree/master/Al/TensorFlow#gpu-version)



# Running GPU batch jobs

```
#!/bin/bash
#SBATCH -p gpu
#SBATCH -n 1
#SBATCH -c 1
#SBATCH --gres=gpu:1
#SBATCH --mem=12000
#SBATCH -J example_tf
#SBATCH -t 30

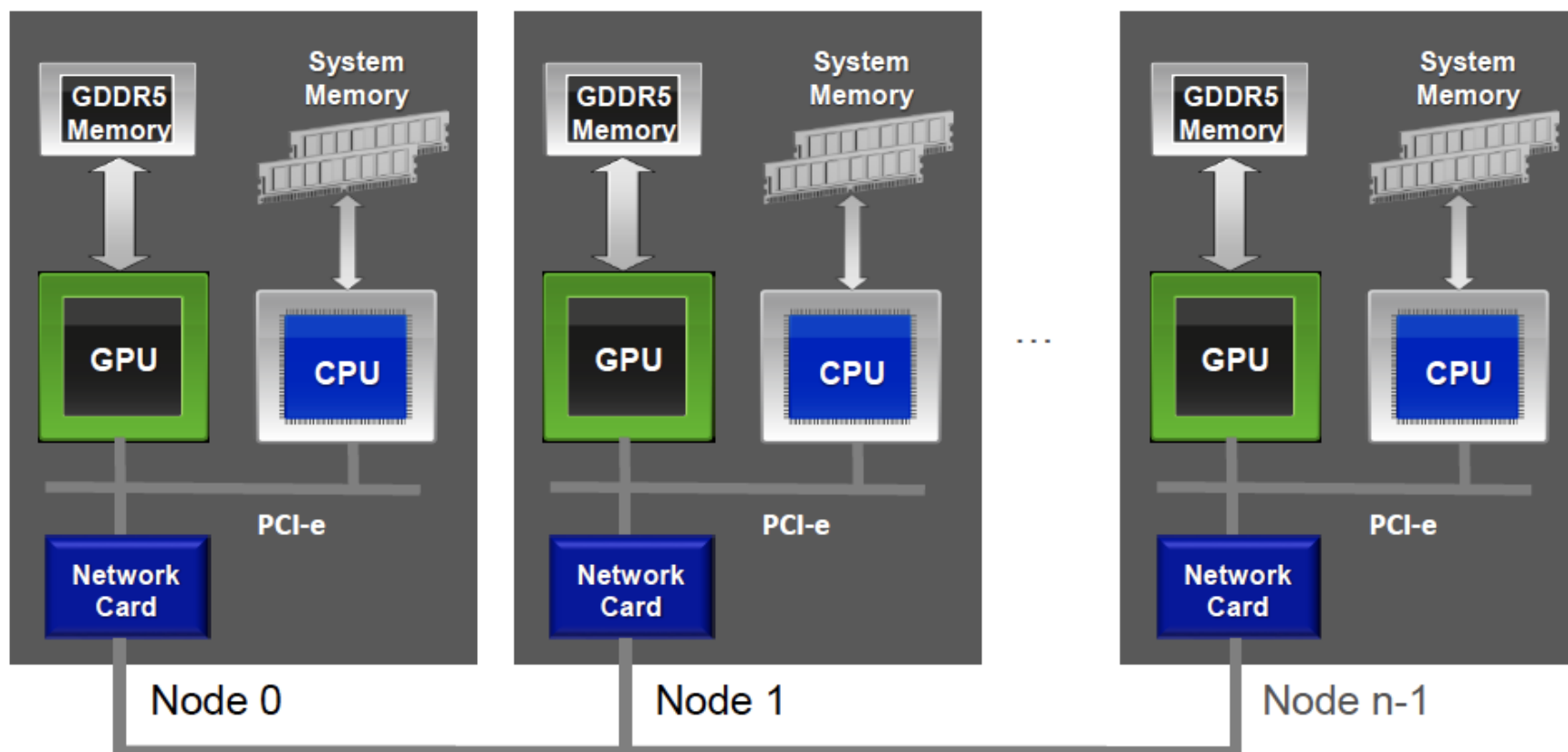
## you could use a local installation of python and tensorflow,
## but we'll use singularity

singularity exec --nv docker://tensorflow/tensorflow:latest-gpu \
    python myCNN.py > output.tf
```

**NOTE:** Use `nvidia-smi` or `nvidia-top` to monitor GPU usage in real-time on the execution host

# Multi-GPU jobs with MPI

MPI+CUDA



Slide by Jiri Kraus and Peter Messmer

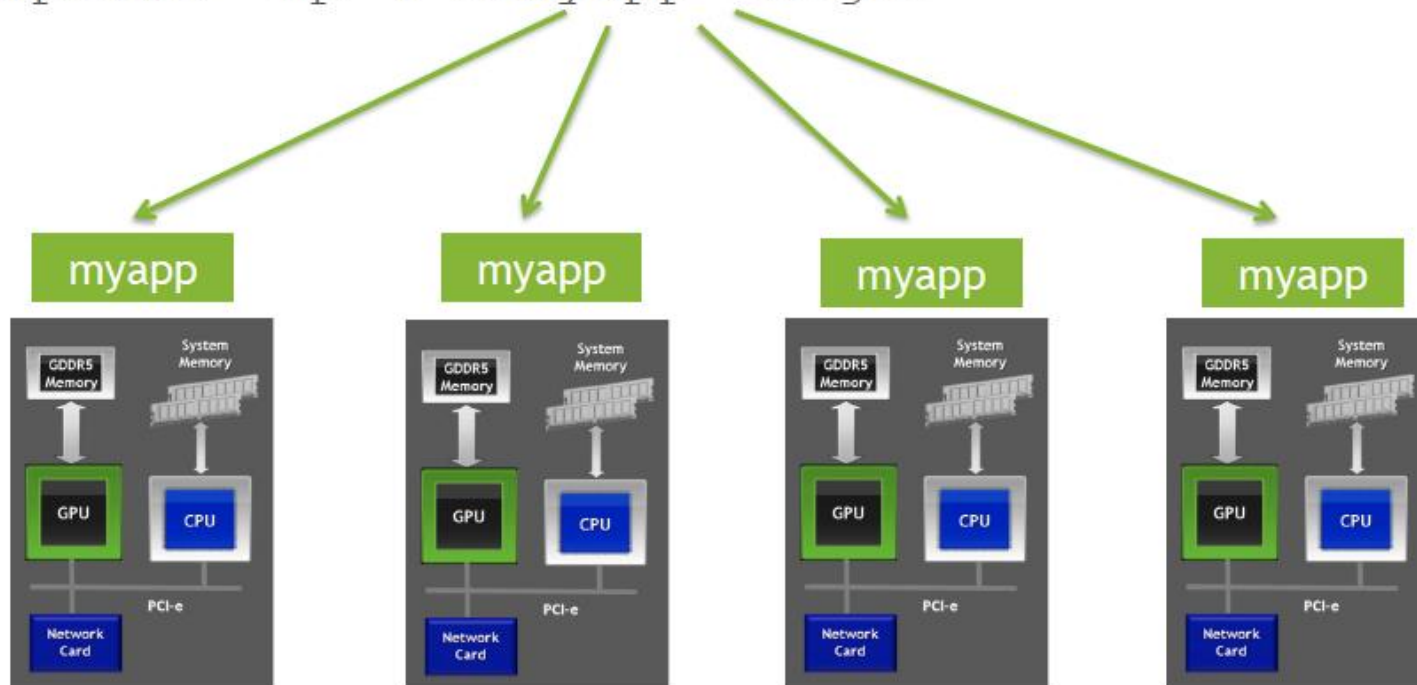
<https://on-demand.gputechconf.com/gtc/2014/presentations/S4236-multi-gpu-programming-mpi.pdf>

# Multi-GPU jobs with MPI

## Compiling and Launching

```
$ mpicc -o myapp myapp.c
```

```
$ mpirun -np 4 ./myapp <args>
```



# Multi-GPU jobs with MPI

**Example:** 2 nodes with 4 MPI tasks and 4 GPUs per node

```
#!/bin/bash
#SBATCH -N 2
#SBATCH --ntasks-per-node=4
#SBATCH --gres=gpu:4
#SBATCH --mem-per-cpu=8G
#SBATCH -J gpu_mpi_test
#SBATCH -t 1:00:00
#SBATCH -p gpu_test
#SBATCH -o gpu_mpi_test.out
#SBATCH -e gpu_mpi_test.err

# --- Load required modules
module load gcc/12.2.0-fasrc01
module load openmpi/4.1.5-fasrc02 # built against cuda/12.2.0-fasrc01

# --- Launch application
srun -n 8 --mpi=pmix ./app.mpi.cuda
```

# Request Help - Resources

- <https://docs.rc.fas.harvard.edu/kb/support/>
  - Rocky 8 Transition Guide
    - <https://docs.rc.fas.harvard.edu/kb/rocky-8-transition-guide/>
  - Email
    - [rchelp@rc.fas.harvard.edu](mailto:rchelp@rc.fas.harvard.edu)
  - Office Hours
    - **Wednesday noon-3pm** <https://harvard.zoom.us/j/97676134704>
  - Training Calendar
    - <https://www.rc.fas.harvard.edu/upcoming-training/>





**Thank you! Questions? Comments?**

**Plamen Krastev, PhD**

**Harvard - FAS Research Computing**