GPU Computing: Leveraging GPGPUs on the HBSGrid Cluster

Bob Freeman, PhD
Research Computing Services

5 April, 2024

Harvard Business School

Agenda and Goals

Agenda

- Introductions / Q&A
- Brief background on GPUs
- Running GPU jobs
  - Job options
  - GPU options
  - GUI vs command-line comparison
- Leveraging GPUs in your code
  - Frameworks
  - GPU coding
  - Containers
- Memory usage on CPUs and GPUs
- Considerations for:
  - Large memory projects
  - Running multiple jobs
- Debugging code / performance monitoring
- Do's and Don'ts

Goals

- Understanding how the technology works
- Learn how run your analyses with appropriate resource asks (cores, RAM, # GPUs, etc)
- Learn how to troubleshoot in case of code, workflow, and job problems

Ultimately…

- **To help you be more efficient and successful with your research!**
- **Minimize any negative impact on other people using these limited, shared resources**

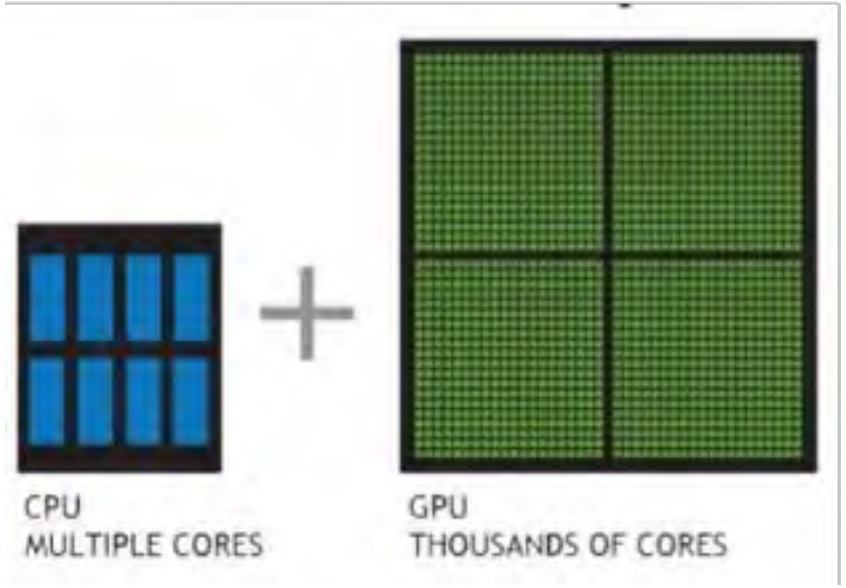
Credits

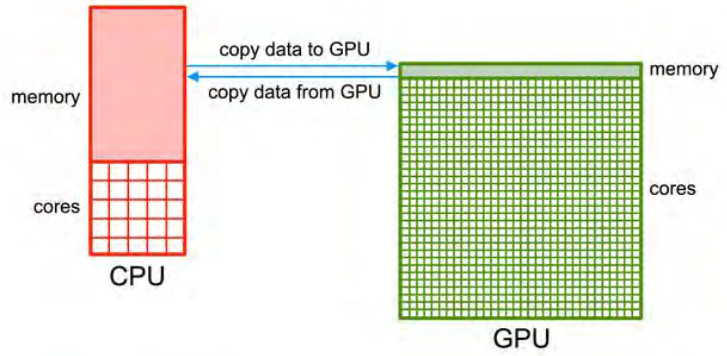
- FASRC GPU materials and example codes
- Princeton Research Computing: GPU Computing
- Digital Alliance of Canada's AI & Machine Learning Guide

Harvard Business School

Brief Background

- GPUs on the HBSGrid cluster are more correctly named GPGPUs: General-Purpose Graphics Processing Unit
  - Originally used to process data for computer displays – rendering engines that would execute concurrently
  - Beginning ~ 2008, GPUs were designed for purposes beyond graphics processing – typically that which is done by the CPU
  - Often these are referred to as ”hardware accelerators” to significantly boost computing performance.
- Embedded GPUs (such as our Tesla V100s) cannot be used independently – the CPUs are required for functionality. Typical flow includes:
  - Pull the data into main (CPU) memory
  - Transferring the data to the GPU memory
  - Performing the processing on the GPU
  - Transferring the data back to main memory
- In addition, code & job performance is tied to:
  - Coding & algorithm pattern
  - Topology of the hardware
  - Communication between the CPU & GPU
  - “Other constraints” – e.g. other users on the GPUs & node



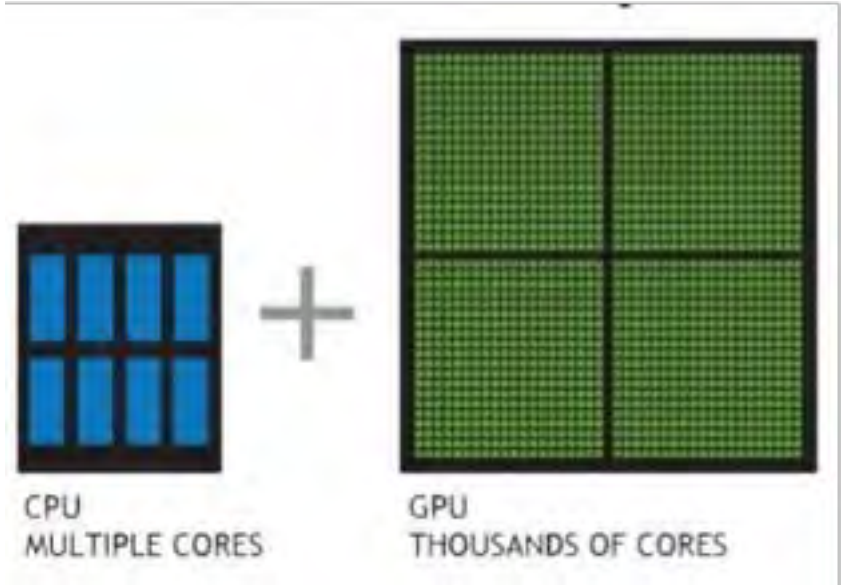


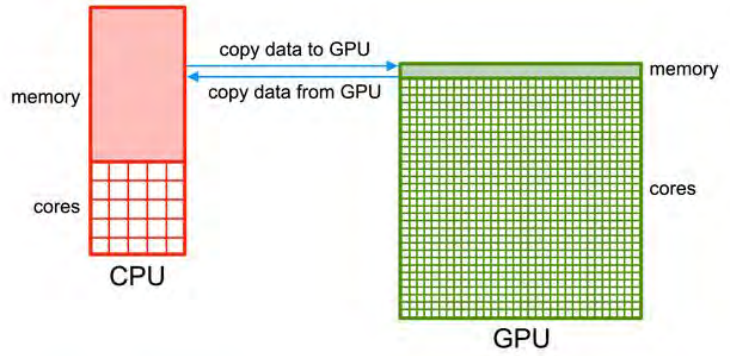

```
data = open("input.dat");          # read the data on the CPU
copyToGPU(data);                   # copy the data to the GPU
matrix_inverse(data.gpu);          # perform a matrix operation on the GPU
copyFromGPU(data);                 # copy the resulting output to the CPU
write(data, "output.dat");         # write the output to file on the CPU
```

Harvard Business School

Using GPUs

- GPU enabled applications requires 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
- Without additional work, this data transfer can decrease code performance by up to 2x due to gating between main and GPU memory
  - Use NVidia's MultiProcess Server (MPS) to overcome this limit



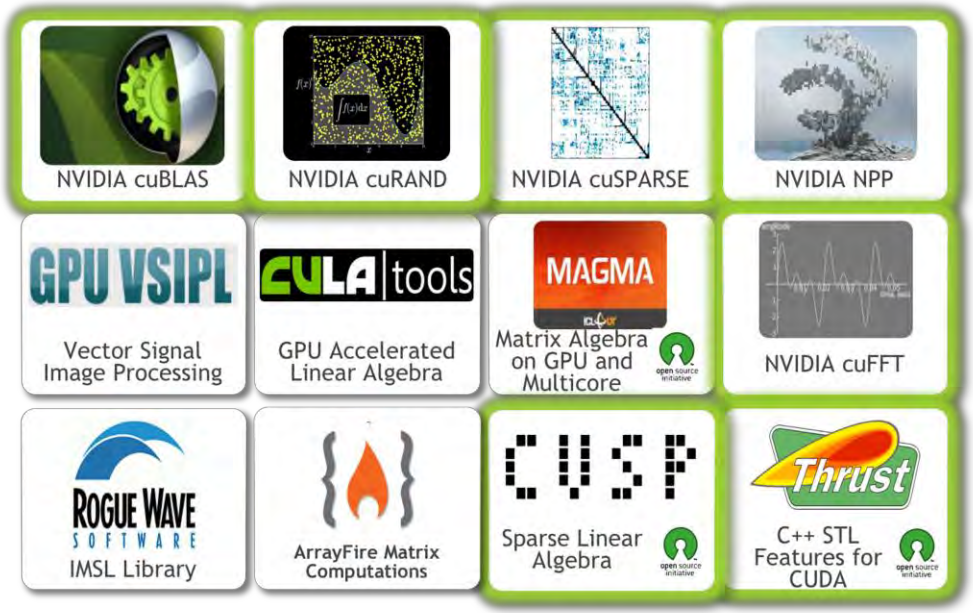



```
data = open("input.dat");          # read the data on the CPU
copyToGPU(data);                   # copy the data to the GPU
matrix_inverse(data.gpu);          # perform a matrix operation on the GPU
copyFromGPU(data);                 # copy the resulting output to the CPU
write(data, "output.dat");         # write the output to file on the CPU
```

Harvard Business School

Using GPUs

- Four ways to leverage GPU capabilities
- Applications already programmed to utilize the GPUs
  - MATLAB supports GPU computing via PCT
  - Containers available from NVidia's Cloud Computing platform
- Math/Hardware Libraries
  - "Drop-in Acceleration", cuBLAS
- OpenACC (Open Accelerative) directives:
  - additional options for C, C++, Fortran compilers
  - Simplifies GPU programming
  - Programming language



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

Slides from FASRC; and Jeff Larkin – NVidia
https://www.bu.edu/tech/files/2017/04/OpenACC-2017Spring.pdf

Using GPUs

- Using Traditional 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

- Underpinning this is the CUDA software layer (Compute Unified Device Architecture)
  - software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements for the execution of compute kernels
  - 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
  - CUDA Toolkit includes GPU-accelerated libraries, a compiler, programming guides, API references, and the CUDA runtime

Using GPUs

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);
```

Compiling

CUDA C

```
# Load software modules
$ module load nvhpc/23.7-fasrc01

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

CUDA Fortran

```
# Load software modules
$ module load nvhpc/23.7-fasrc01

# Compile command
$ nvfortran -o saxpy.x saxpy.cuf
```

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

Using GPUs

- Check the three additional frameworks (at least) that easily leverage GPU resources
  - PyTorch
    - Princeton PyTorch job https://github.com/PrincetonUniversity/gpu_programming_intro/tree/master/03_your_first_gpu_job#pytorch
    - Digital Canada's PyTorch: https://docs.alliancecan.ca/wiki/PyTorch
  - Tensorflow
    - Princeton TensorFlow job https://github.com/PrincetonUniversity/gpu_programming_intro/tree/master/03_your_first_gpu_job#tensorflow
    - Digital Canada's TensorFlow: https://docs.alliancecan.ca/wiki/TensorFlow
  - NVidia RAPIDS
    - Drop-in replacement for Python's Pandas frameworks
    - https://resources.nvidia.com/en-us-nvidia-rapids
- Other 'meta' frameworks that can leverage GPUs and abstract the underlying CUDA complexities:
  - XGBoost, Scikit-Learn, SnapML: see the DC AI/ML page
  - Large-scale Machine Learning: see the relevant DC page
- Using containers
  - NVidia GPU-enabled container library at https://docs.nvidia.com/ngc/

GPUs on the HBSGrid Cluster

- Our GPU node `rhrcsnod12`
  - 64 Intel cores
  - 250 GB RAM (usable)
- 5 GPUs are NVidia Tesla V100s
  - NVidia Volta architecture
  - *Each has 32 GB on-board RAM
  - Connected via PCIe or SMP interconnect if on same socket
  - 5 GPUs arranged on 3 of 4 CPU sockets (cores 0 – 47)
  - ***NB!** Jobs that land on cores 48-63 will not be able to access the GPUs! (But you needn't worry about this, as LSF won't let your job run if these are the only cores available)

```
[08:31:30, rfreeman@rhrcsnod12:~]$ nvidia-smi topo -m
        GPU0    GPU1    GPU2    GPU3    GPU4    CPU Affinity    NUMA Affinity   GPU NUMA ID
GPU0     X      SYS     SYS     SYS     SYS     0-15,64-79      0               N/A
GPU1    SYS      X      SYS     SYS     SYS     16-31,80-95     1               N/A
GPU2    SYS     SYS      X      SYS     SYS     16-31,80-95     1               N/A
GPU3    SYS     SYS     SYS      X      SYS     32-47,96-111    2               N/A
GPU4    SYS     SYS     SYS     SYS      X      32-47,96-111    2               N/A

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

	V100 PCIe	V100 SXM2	V100S PCIe
GPU Architecture	NVIDIA Volta		
NVIDIA Tensor Cores	640		
NVIDIA CUDA® Cores	5,120		
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS	8.2 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS	16.4 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS	130 TFLOPS
GPU Memory	32 GB /16 GB HBM2		32 GB HBM2
Memory Bandwidth	900 GB/sec		1134 GB/sec
ECC	Yes		
Interconnect Bandwidth	32 GB/sec	300 GB/sec	32 GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink™	PCIe Gen3
Form Factor	PCIe Full Height/Length	SXM2	PCIe Full Height/Length
Max Power Comsumption	250 W	300 W	250 W
Thermal Solution	Passive		
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC®		

Running GPU Jobs

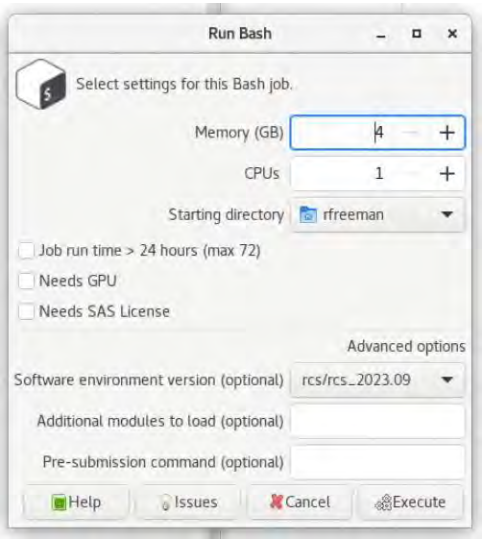
Interactive Sessions

- (OK) Use Grid3 GUI launchers (ok, but not best)
- (Best) Run via terminal: get session on login node (4G RAM + 1 CPU), and then submit job

```
[cli1]$ bsub -q gpu_int -Is -gpu - -R "rusage[mem=5000]" -M 5000 -hl /bin/bash

Job <2477991> is submitted to queue <gpu_int>.
<<Waiting for dispatch ...>>
<<Starting on rhrcsnod12>>

[nod12]$ nvidia-smi   # show us what we've got
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 545.23.08              Driver Version: 545.23.08    CUDA Version: 12.3     |
|-------------------------------+----------------------+----------------------+
| 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  Tesla V100-PCIE-32GB           Off | 00000000:25:00.0 Off |                    0 |
| N/A   66C    P0             245W / 250W |   3718MiB / 32768MiB |     90%      Default |
|                                         |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     96703      C   ...jharvard/conda//bin/python             3714MiB |
+-----------------------------------------------------------------------------+

[nod12]$ module load rcs/rcs_2023.09; spyder &  # load latest grid3 just in case & start our spyder session

...
```

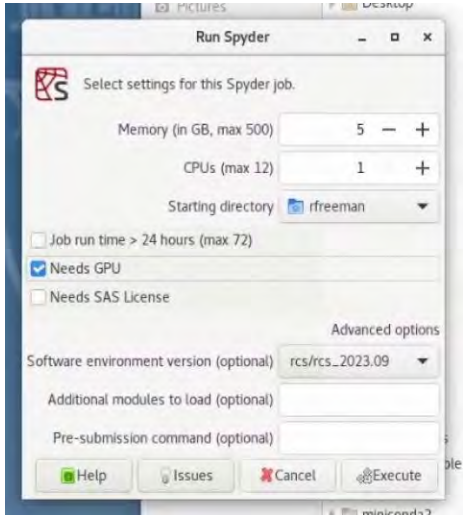
Spyder on GPU node

Login node shell to submit GPU job



Harvard Business School

Running GPU Jobs



Use the Grid3 GUI launchers

- Only works for interactive sessions (*gpu_int*)
- Uses cluster defaults for GPU configuration
- There is no ability to specify additional options for GPU configuration, pre-job execution setup, or post-job execution teardown
- Ability to execute code & monitoring execution will vary by GUI application

```
$ bsub -q gpu_int -Is -R "rusage[mem=5000]" \
   -gpu - \
   -M 6000 -hl /bin/bash
$ nvidia-smi
$ spyder &
```

Use shell / terminal / command-line

- Useful for interactive (*gpu_int*) and batch jobs (*gpu*)
- Can use cluster default GPU configuration or change any number of GPU-related options at job submission
- Ability to run code & commands as pre-job execution setup and/or post-job execution teardown
- Can execute GUI applications in addition to monitoring code execution via shell

Harvard Business School

GPU Options

Default / GPU Configuration

- The configuration can be seen in the bottom of any `bjobs -l jobID` or `bhist -l jobID` job details
- The default is used when launch apps via Grid3 launchers and when specifying '... `-gpu -` ...' in `bsub` commands
- `-gpu - "num=1:aff=no:mode=shared:mps=no:j_exclusive=no"`

GPU Options

- **num** (default = 1): # of GPUs one is requesting. Once the job is dispatched, GPUs will always be numbered from 0.
- **aff** (default = no | yes): CPU-GPU affinity. This indicates whether or not the CPU & GPU should be on the same socket (same group of CPU cores)
  - Set to **yes** if trying to increase the efficiency of high-throughput batch jobs
- **mode** (default = shared | exclusive): Indicates whether or not multiple processes share using the GPU & its resources simultaneously; or if each executing processes has solitary, exclusive use and control
  - Set to **exclusive** to collect stats from LSF on GPU usage or if one wishes to ensure that the GPU is reserved exclusively for the current job process; complete context switches will occur if multiple processes wish to use it. **Shared** is generally recommended.
- **mps** (default = no | yes): Enables or disables the NVIDIA Multi-Process Service (MPS) for the GPU(s) allocated to the job(s). This can be extremely useful for performance, esp. with large data- or compute-intensive work
- **j_exclusive** (default = no | yes): Specifies whether the allocated GPU(s) can be used by jobs from other users.

https://hbs-rcs.github.io/hbsgrid-docs/tutorials/scaling-work#gpu-computing
https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=jobs-submitting-that-require-gpu-resources
https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=o-gpu

GPU Options

GPU Options (cont'd)

- **mps** (default = no | yes): Well, there's actually a series of additional options that you may wish to use:
  - If mps=yes is set, LSF starts one MPS daemon per host per job.
  - If mps=yes,shared is set, LSF starts one MPS daemon per host for all jobs that are submitted by the same user with the same resource requirements. These jobs all use the same MPS daemon on the host.
  - If mps=per_gpu is set, LSF starts one MPS daemon per GPU per job. When enabled with share (that is, if mps=per_gpu,shared is set), LSF starts one MPS daemon per GPU for all jobs that are submitted by the same user with the same resource requirements. These jobs all use the same MPS daemon for the GPU.
- **gmem**=*mem_value* Specify the GPU memory on each GPU required by the job. The format of *mem_value* is the same to other resource value (for example, mem or swap) in the rusage section of the job resource requirements (-R).

Example submissions:

```
# submit job with default GPU (1 gpu, shared, no MPS) & RAM/CPU requests (1 core, 5-8 GB RAM max) to batch queue
bsub -gpu - python myscript.py

# an attempt to run
bsub -gpu "num=1::gmem=8G" python myscript.py
```

Other options can be specified (e.g. RAM, GPU type) as part of the bsub job submission command. Please see LSF Submitting jobs that require GPU resources documentation for these and example job submission commands.

https://hbs-rcs.github.io/hbsgrid-docs/tutorials/scaling-work#gpu-computing
https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=jobs-submitting-that-require-gpu-resources
https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=o-gpu

Job Options & Requesting Resources

- Same rules apply for requesting resources -- ask for what you need
- Guidelines on our GitHub HBSGrid User Guide website for CPUs and RAM
  - Be thoughtful on your RAM ask! **RAM wastage is our biggest problem and obstructs research**
  - Also be thoughtful on your core requests, again for the same reason.
- Ensure that you've done scaling tests to inform your ask for multiple cores (see here, here, and example)
- Before asking for more than 1 GPU, demonstrate through performance analysis (see later slides) that one requires it

https://hbs-rcs.github.io/hbsgrid-docs/commandline/#resource-requirements
https://hpc.mediawiki.hull.ac.uk/Scaling
https://waterprogramming.wordpress.com/2021/06/07/scaling-experiments-how-to-measure-the-performance-of-parallel-code-on-hpc-systems/
https://hemelb-dev.github.io/HemeLB-Carpentries/03-benchmarking-and-scaling/index.html

Leveraging GPUs in your Code

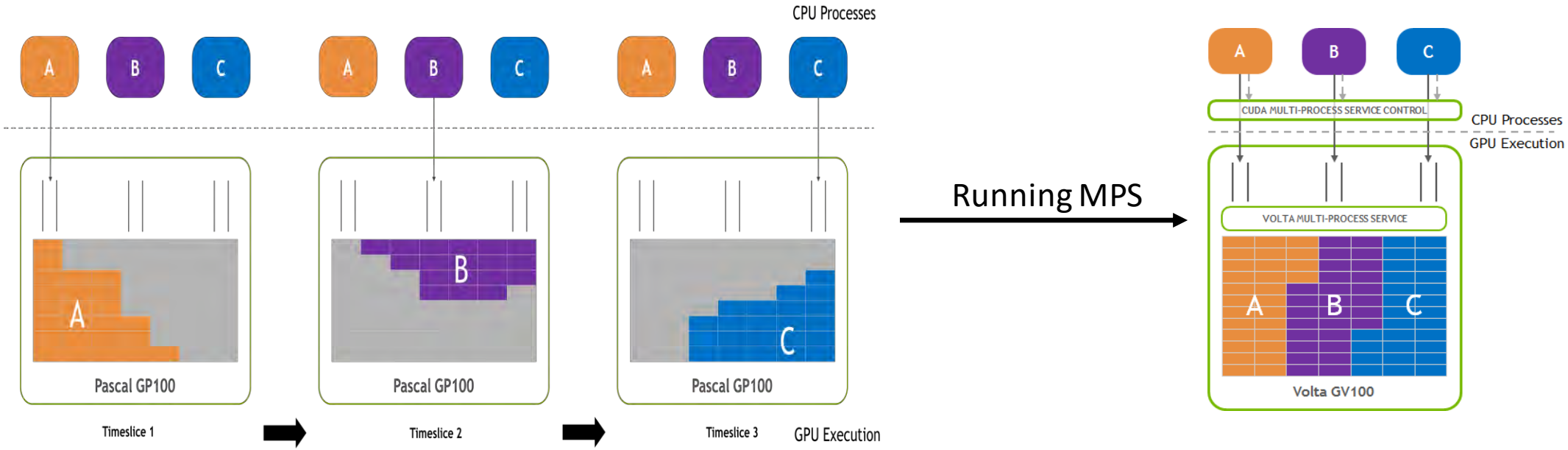
- Please see previous examples for using GPUs in Python code via Numba directives
- Consider using RAPIDS as a drop-in Python Pandas replacement
- Please see also the previously mentioned Python AI/ML frameworks
- Using containers
  - NVidia GPU-enabled container library at https://docs.nvidia.com/ngc/
- Memory usage is can be a big problem
  - All GPU jobs must fit inside 250 GB RAM
  - Each GPU has only 32 GB RAM to fit data structures - both input and output
  - Use 'chunking' capabilities for both Python Pandas and R's dataframes
    - https://pandas.pydata.org/docs/user_guide/scale.html
    - https://pandas.pydata.org/docs/user_guide/io.html#performance-considerations
  - Use memory optimization techniques
    - https://pandas.pydata.org/docs/user_guide/enhancingperf.html
- Storage utilization is also a key efficiency component! Please see DC's Handling Large Collection of Files page.

Running Multiple Jobs on GPUs

- Shared used is easiest way to go
- Should consider using NVidia's MultiProcess Server (MPS) as a front to the GPU(s) allocated to one's job
- This is not enabled by default, but can decrease runtime by up to 2x
- MPS is an **alternative, binary-compatible implementation** of the CUDA Application Programming Interface (API). The MPS runtime architecture is **designed to transparently enable co-operative multi-process CUDA applications**, typically MPI jobs, to utilize Hyper-Q capabilities on the latest NVIDIA (Kepler and later) GPUs. Hyper-Q **allows CUDA kernels to be processed concurrently on the same GPU; this can benefit performance when the GPU compute capacity is underutilized by a single application process.**
  - *GPU utilization:* A single process may not utilize all the compute and memory-bandwidth capacity available on the GPU. MPS allows kernel and memcopy operations from different processes to overlap on the GPU, achieving higher utilization and shorter running times.
  - *Reduced on-GPU context storage:* Without MPS each CUDA processes using a GPU allocates separate storage and scheduling resources on the GPU. In contrast, the MPS server allocates one copy of GPU storage and scheduling resources shared by all its clients. Volta MPS supports increased isolation between MPS clients, so the resource reduction is to a much lesser degree.
  - *Reduced GPU context switching:* Without MPS, when processes share the GPU their scheduling resources must be swapped on and off the GPU. The MPS server shares one set of scheduling resources between all of its clients, eliminating the overhead of swapping when the GPU is scheduling between those clients.

Running Multiple Jobs on GPUs

- Shared used is easiest way to go
- Should consider using NVidia's MultiProcess Server (MPS) as a front to the GPU(s) allocated to one's job
- This is not enabled by default, but can decrease runtime by 2x



The Multi-Process Service (MPS) enables multiple processes (e.g. MPI ranks) to concurrently share the resources on a single GPU. This is accomplished by starting an MPS server process, which funnels the work from multiple CUDA contexts (e.g. from multiple MPI ranks) into a single CUDA context. In some cases, this can increase performance due to better utilization of the resources.

Harvard Business School

Using MultiProcess Server

- *The MPS capabilities are currently not working properly. Stay tuned for updates on this.*

Performance Monitoring

Interactive Sessions

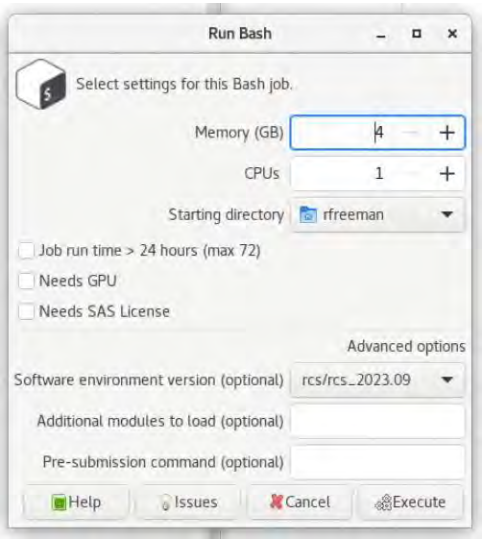
- (OK) Use Grid3 GUI launchers (ok, but not best)

- (Best) Run via terminal: get session on login node (4G RAM + 1 CPU), and then submit job

```
[cli1]$ bsub -q gpu_int -Is -n 1 -gpu - -R "rusage[mem=5000]" -M 5000 -hl /bin/bash

Job <2477991> is submitted to queue <gpu_int>.
<<Waiting for dispatch ...>>
<<Starting on rhrcsnod12>>

[nod12]$ nvidia-smi   # show us what we've got
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 545.23.08              Driver Version: 545.23.08    CUDA Version: 12.3     |
|-----------------------------------------+----------------------+----------------------+
| 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  Tesla V100-PCIE-32GB           Off | 00000000:25:00.0 Off |                    0 |
| N/A   66C    P0             245W / 250W |   3718MiB / 32768MiB |     90%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=========================================================================================|
|    0   N/A  N/A     96703      C   ...jharvard/conda//bin/python               3714MiB |
+-----------------------------------------------------------------------------------------+

[nod12]$ module load rcs/rcs_2023.09; spyder &  # load latest grid3 just in case & start our spyder session

...
```

Spyder on GPU node

Login node shell to submit GPU job



Harvard Business School

Performance Monitoring

```
[nod12]$ nvidia-smi pmon # and (before this command) execute script in background or press Run in Spyder
# gpu        pid   type     sm    mem    enc    dec    jpg    ofa    command
# Idx          #    C/G      %      %      %      %      %      %    name
    0     284596      C      0      0      -      -      -      -    python
    0     284596      C      0      0      -      -      -      -    python
    0     284596      C      3      0      -      -      -      -    python
    0     284596      C      5      0      -      -      -      -    python
    0     284596      C      5      0      -      -      -      -    python
    0     284596      C      5      0      -      -      -      -    python
    0     284596      C      5      0      -      -      -      -    python
    0     284596      C      5      0      -      -      -      -    python
    0     284596      C      5      0      -      -      -      -    python
    0     284596      C      5      0      -      -      -      -    python
    0     284596      C      5      0      -      -      -      -    python
    0     284596      C      1      0      -      -      -      -    python
    0     284596      C      0      0      -      -      -      -    python
    0     284596      C      0      0      -      -      -      -    python
 (…press ctrl-C to exit out…)

[nod12]$ nvidia-smi dmon
# gpu    pwr  gtemp  mtemp     sm    mem    enc    dec    jpg    ofa   mclk   pclk
# Idx      W      C      C      %      %      %      %      %      %    MHz    MHz
    0     36     30     29      0      0      0      0      -      -    877   1230
    0     36     30     29      0      0      0      0      -      -    877   1230
    0     57     30     29      4      1      0      0      -      -    877   1230
    0     43     30     30      7      1      0      0      -      -    877   1230
    0     51     30     30      4      1      0      0      -      -    877   1230
    0     36     30     29      4      1      0      0      -      -    877   1230
    0     37     30     30      4      1      0      0      -      -    877   1230
    0     37     30     30      7      1      0      0      -      -    877   1230
    0     36     30     30      4      1      0      0      -      -    877   1230
    0     36     30     30      7      1      0      0      -      -    877   1230
    0     37     30     30      7      1      0      0      -      -    877   1230
    0     36     30     30      7      1      0      0      -      -    877   1230
    0     36     30     30      0      0      0      0      -      -    877   1230
    0     36     30     29      0      0      0      0      -      -    877   1230
    0     36     30     29      0      0      0      0      -      -    877   1230
 (…press ctrl-C to exit out…)
```

% GPU Utilization and
% GPU Memory Used

Harvard Business School

Additional Resources

The following resources offer information, reference, self-paced tutorials, and programming examples for many different GPU programming models, toolkits, frameworks, and test suites:

- GPU Computing (HBS RCS)
- Submitting and monitoring GPU jobs (IBM Spectrum LSF)
- Tesla V100 GPU documentation landing page (NVidia)
- AI and HPC Containers and NVidia GPU-optimized containers catalog for AI/ML & HPC (NVidia)
- RAPIDS as a Pandas drop-in replacement (NVidia)
- Understanding GPU Architectures (Cornell Virtual Workshop series)
- GPU Computing on the FASRC Cluster: slides, docs, example codes
- Princeton University, Research Computing:
  - GPU Computing documentation,
  - Intro to GPU Programming workshop (self-paced)
  - Workshop series (with GitHub repos) archives
- Digital Alliance Canada – AI and Machine Learning (PyTorch, Tensorflow, and other helpful guides)
- GPU Programming series training materials landing page (OpenHackathon.org)

Any questions, please contact research@hbs.edu.