Lecture 20: GPU Computation

Current Assignments

- **Project 1: High Performance Linpack**
- You should have received an overleaf invitation.
- A link to the project description is on the website and posted in slack.
- The project is due by 11:59pm Nov 21st.
- We only have time for 2 projects, so they are each worth 20% of your final grade.

Evolution of Supercomputers

- Increasing processor speed (1960 - now)
- Processing vectors (1970s - now)
- Additional processors for single memory systems (monocomputer)
- Fast networks for distributed computation over many monocomputers (compute clusters or multicomputers) (1990s - now)
- GPUs (2000s - now)

- Supercomputing has evolved from a primarily hardware challenge (1960s-1990s – Cray Era) to creating software that runs on compute clusters (1990s-today – Multicomputer Era)
- (GPU accelerators recapitulate this pattern but over the past 10 years)

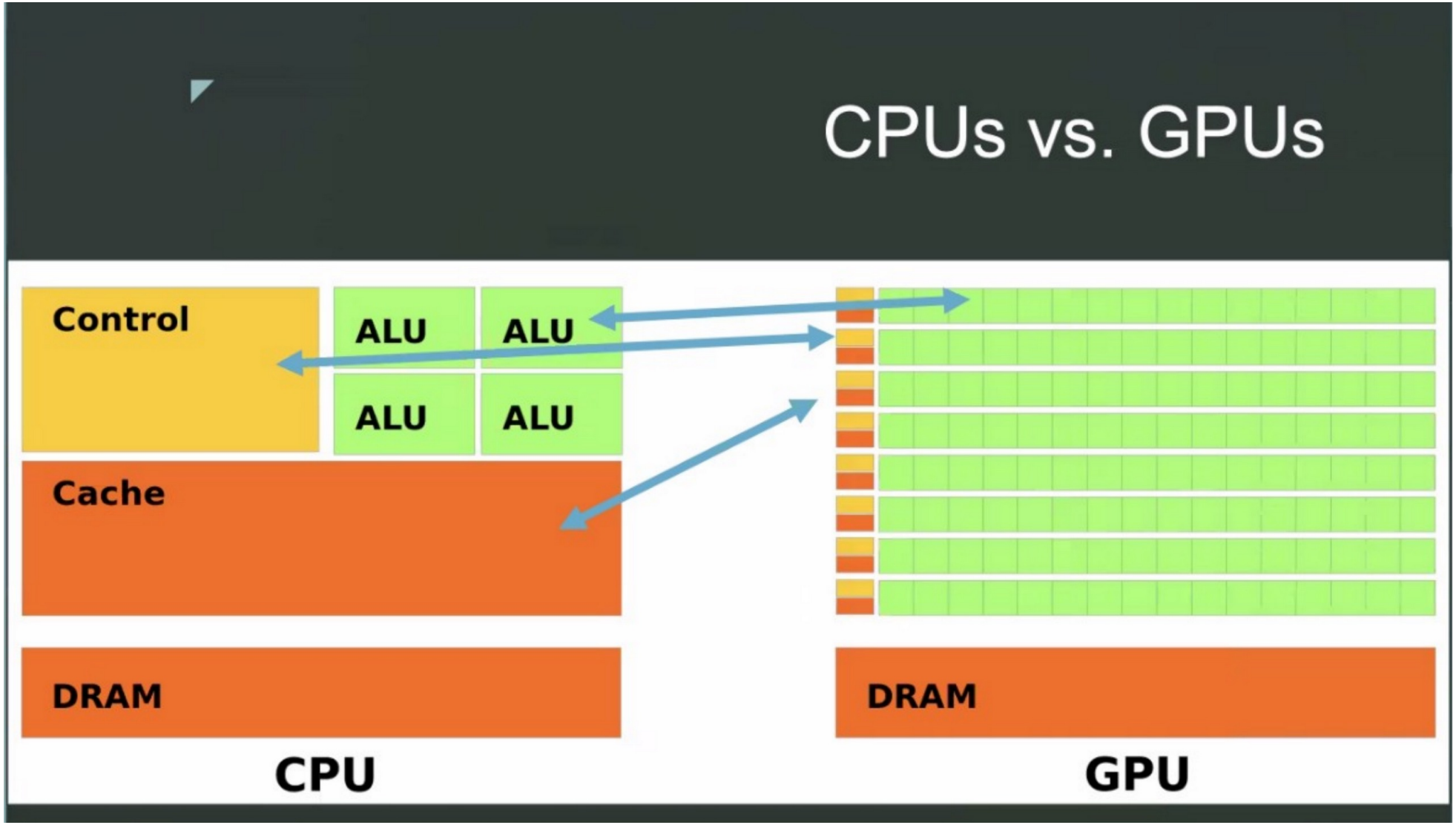
The GPU Era

- In 2001 Nvidia introduced the GeForce 3 video card.
- This card allowed the pixel renderers to be programmed instead of being hardcoded.
- Over time Nvidia made their graphic processing units (GPUs) more programmable with additional programmable engines for vertex and geometry shading.
- These different engines for programming different parts of th graphics pipeline became too complex.
- Nvidia came up with a general model for programming pixel math on the GPU (2009 GeForce 8800)
- The language Nvidia invented to program these pixel operations was the Compute Unified Compute Architecture (CUDA)
- GPUs are part of the reason clusters are moving towards havi fewer but more powerful nodes.

GPUs for Computation

- GPUs proved to be ideal for matrix math.
- The GPU architecture applies the same operation to thousands of elements at the same time.
- This is a bit like the earlier CPU vectorization operations on steroids (512 vector operations in the latest Intel CPUs, The Nvidia H100 GPU has 14,592 CUDA cores)
- The H100 can execute almost 15,000 instructions per clock cycle.
- This doesn’t work for all types of computations, if you can’t apply the same operation to a vector of values the GPU will be slower than a CPU.

CPU vs GPU comparison



Source: Jayce Dowell Presentation on GPU University of New Mexico, Fall 2021

CPU vs GPU comparison

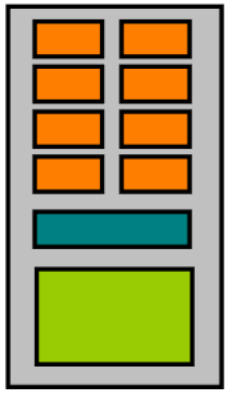
- DRAM is the global memory
- CPUs have few cores (ALU) when GPUs have a large number of cores.
- The CPU has lower latency and lower memory bandwidth compared to the GPU
- The GPU has high throughput and a higher memory bandwidth compared to the CPU

Source: https://thedatafrog.com/en/articles/cuda-kernel-python/

GPU Compute Model

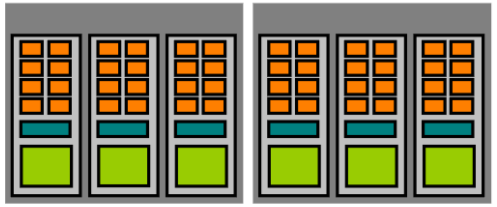
Scalar

Arithmetic logical unit (ALU)



Multiprocessor

Compute unit made of many ALUs



Device

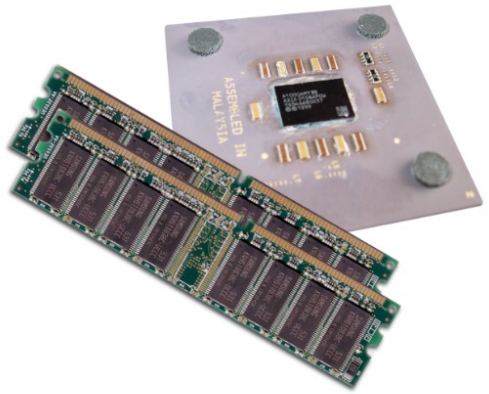
Many compute units across the AMD GPU

Source: https://www.olcf.ornl.gov/wp-content/uploads/2019/12/03-CUDA-Fundamental-Optimization-Part-1.pdf

CPU vs GPU programming model

HETEROGENEOUS COMPUTING

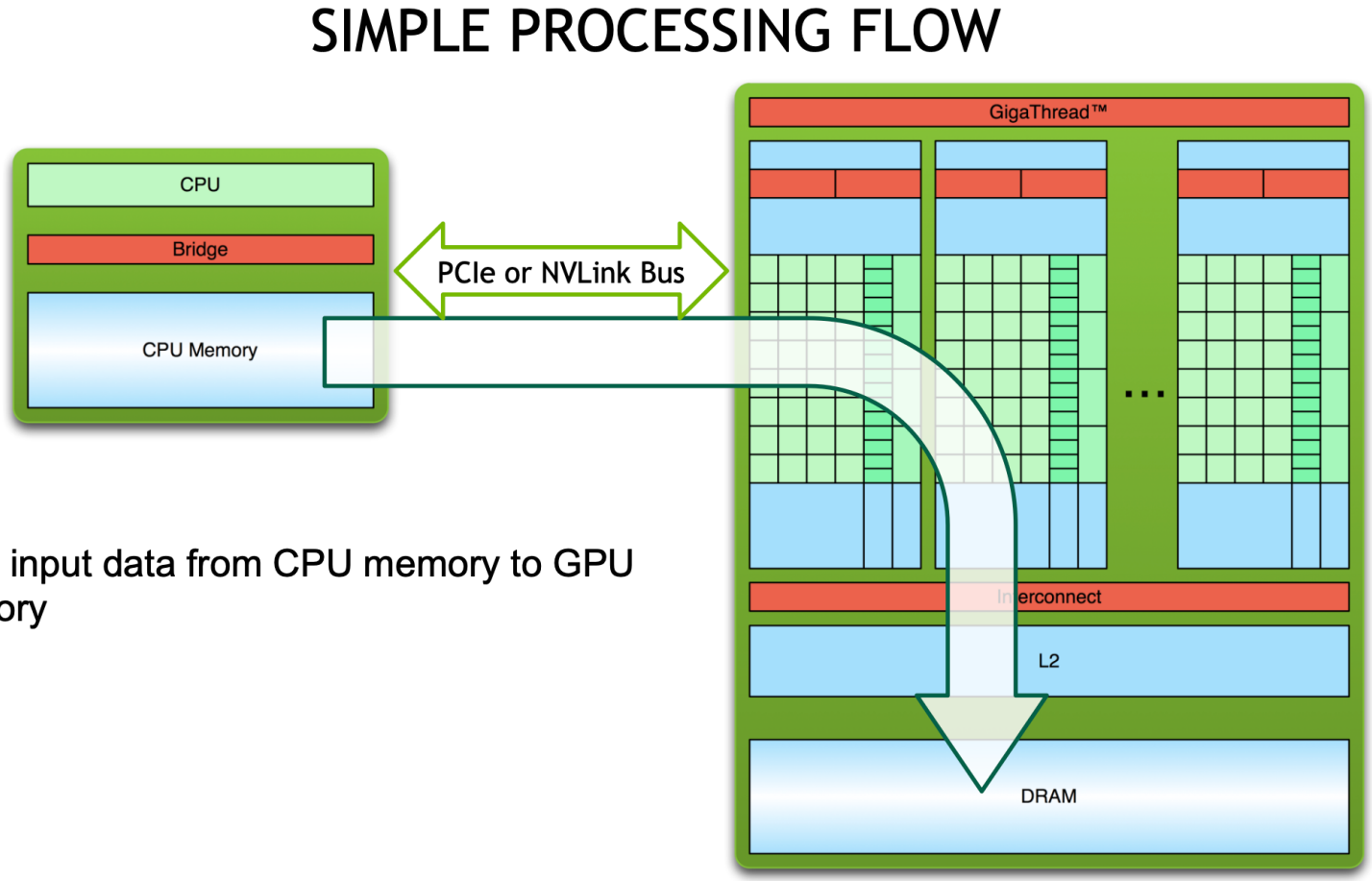
- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)



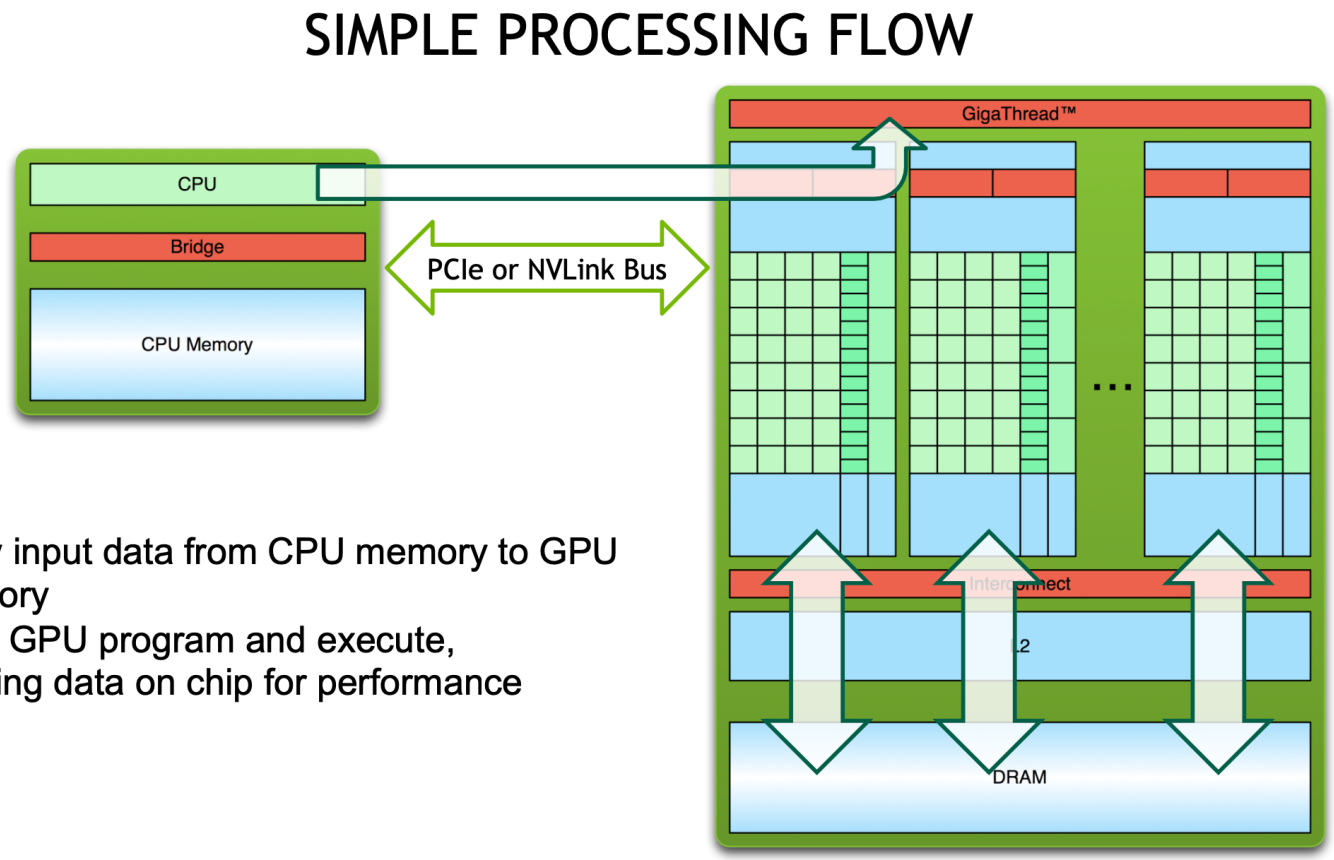
Source: https://www.olcf.ornl.gov/cuda-training-series/ ; CUDA C++

CPU vs GPU programming model

- Before the GPU computation, data has to be moved from CPU memory (host memory) to the GPU memory.
- After the computation the result has to be moved from GPU memory to the CPU memory.



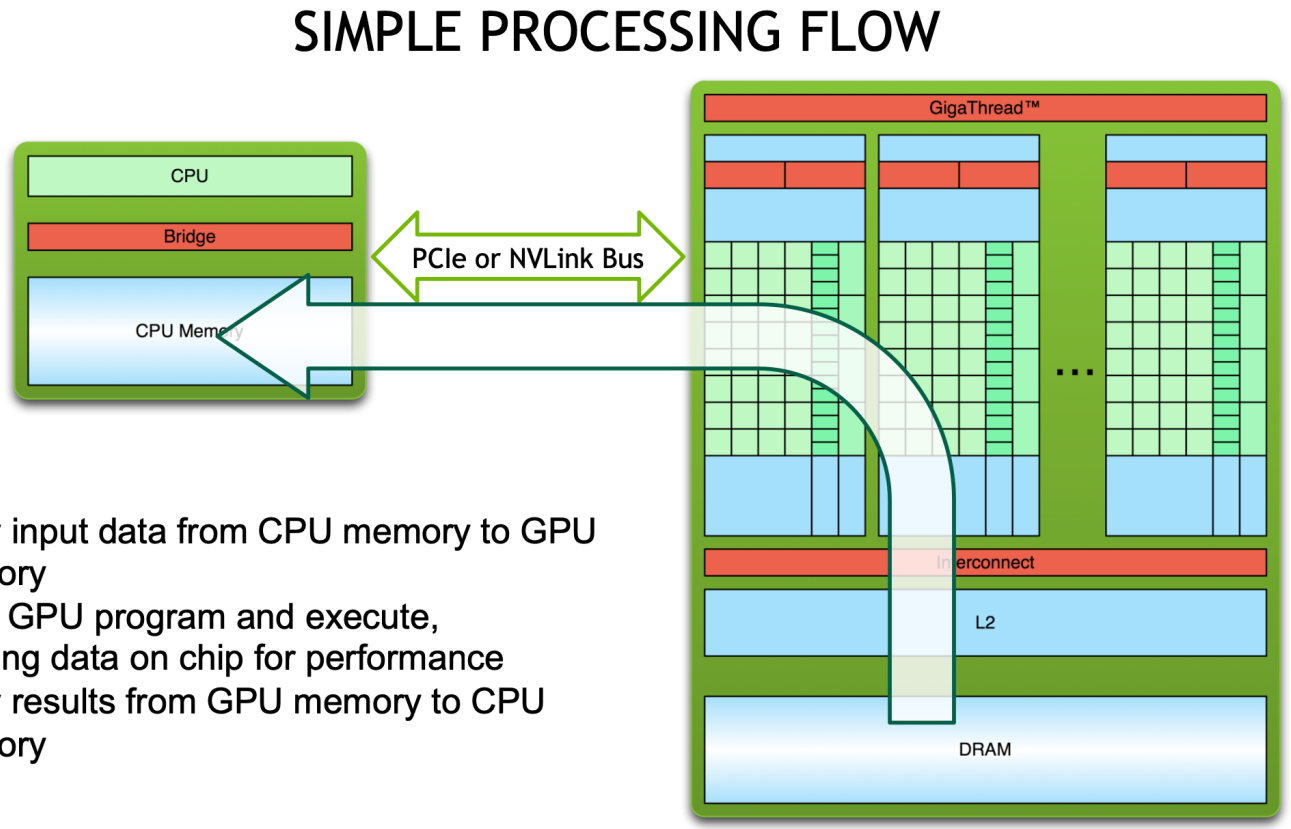
CPU vs GPU programming model



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Source: https://www.olcf.ornl.gov/cuda-training-series/ ; CUDA C++

CPU vs GPU programming model



Source: https://www.olcf.ornl.gov/cuda-training-series/ ; CUDA C++

Find the CUDA Version

```
mfricke@xena:~ nvidia-smi
Fri Apr 19 10:11:25 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.223.02   Driver Version: 470.223.02   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| 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  Tesla K40m           Off  | 00000000:04:00.0 Off |                    0 |
| N/A   30C    P0    64W / 235W |      0MiB / 11441MiB |     95%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

Create conda environment with cuda support and run it on a Xena GPU node

```
@cuda.jit
def Pi(step_vec, sum_vec, step_size):


    tx =  cuda.threadIdx.x # Unique thread ID within 1 block
    ty = cuda.blockIdx.x # Unique block ID
    block_size = cuda.blockDim.x # Number of threads per block
    grid_size = cuda.gridDim.x # Size of the grid


    # We define the grid size:
    startX = cuda.grid(1) # Starting point on the Grid
    gridX = cuda.gridDim.x * cuda.blockDim.x # stride in x
    stride = cuda.gridsize(1)


    # Or we could have written the following
    #startX = tx + ty * block_size
    #stride = block_size * grid_size


    for i in range(startX,  step_vec.shape[0], stride):
        step_vec[i] = (step_vec[i]+0.5) * step_size[0]
        sum_vec[i] +=  4.0 / (1.0 + step_vec[i] * step_vec[i])
```

Create conda environment with cuda support and run it on a Xena GPU node

```
# We convert the variables to
device
step_device = cuda.to_device(step_vec)
sum_device = cuda.to_device(sum_vec)
pi_device = cuda.to_device(pi)
step_size_device = cuda.to_device(step_size)

pi
start = time.time()
#Start
Pi[blocks_per_grid, thread_per_block](step_device, sum_device, step_size_device)

# I copy the devices to host and calculate the final pi
version
sum_vec = sum_device.copy_to_host()
pi = step_size * sum_vec.sum()
end = time.time() # End
```

Create conda environment with cuda support and run it on a Xena GPU node

```
mfricke@xena:~ git clone https://github.com/gmfricke/CalcPiCUDA.git

mfricke@xena:~ conda create --name cuda numba cudatoolkit=11.4 --channel conda-forge

mfricke@xena:~ cd CalcPiCUDA

mfricke@xena:~ conda activate cuda

mfricke@xena:~ srun --gpus 1 python calculate_pi_gpu_opt_clean.py 10
```

TASK

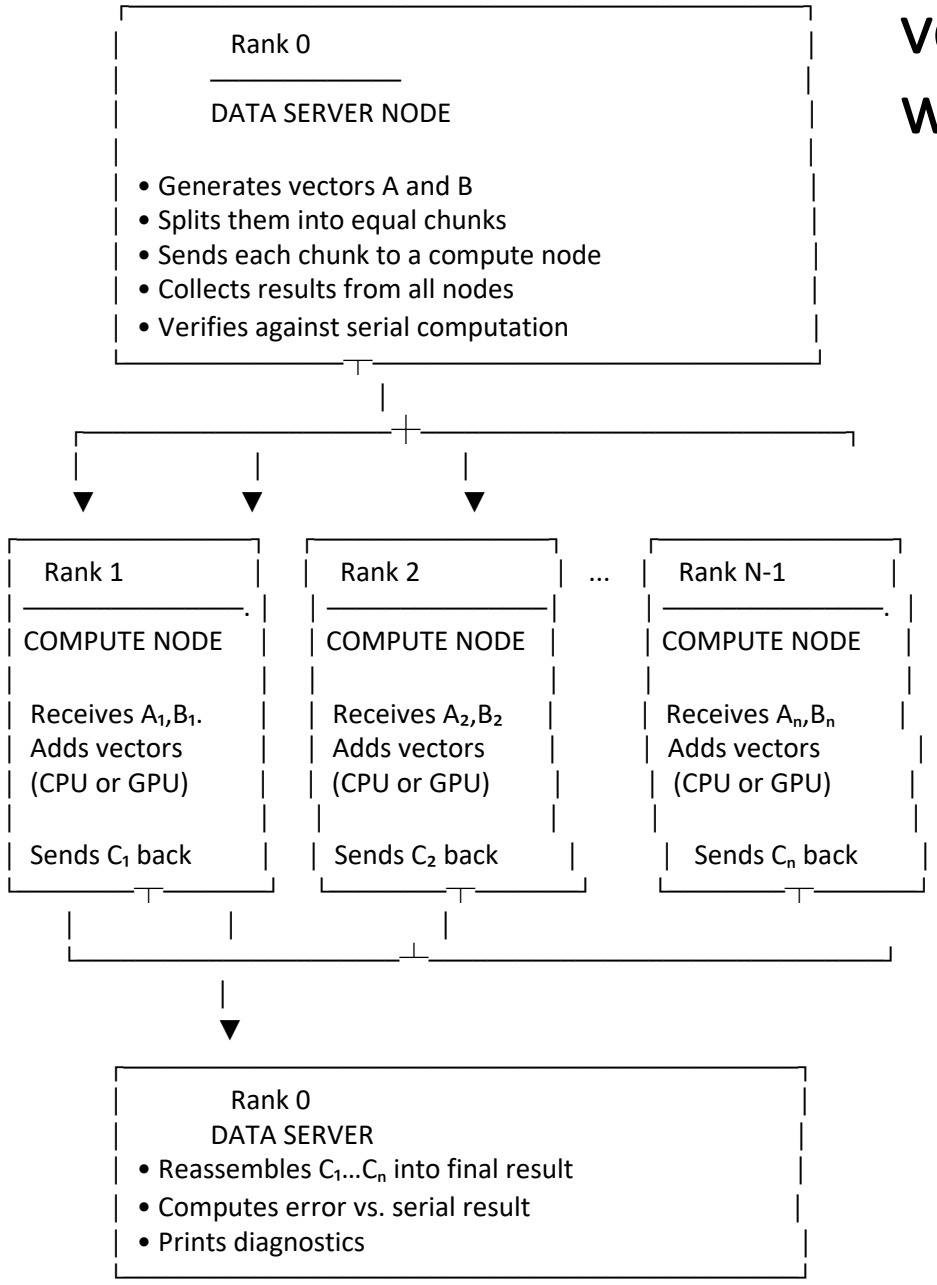
Write a Slurm script using an array job to compare the performance of the GPU accelerated code to the CPU code for various problem sizes.

C++ CUDA and MPI - VecAdd

```
vanilla@easley:~/ $ cd workshops
vanilla@easley:~/workshops $ git pull

vanilla@easley:~/workshops $
cd intro_workshops/code/vecadd
```

C++ CUDA and MPI - VecAdd

```
mfricke@easley:~/workshops/intro_workshop/code
/vecadd $ ls -1
Makefile
vecadd_gpu.cu
vecadd_mpi_cpu.c
vecaddmpi_cpu.sh
vecadd_mpi_gpu.c
vecaddmpi_gpu.sh
```



vecadd_mpi_gpu.c — MPI Data-Server Controller with Optional GPU Offload

- Acts as the **main control program** for distributed vector addition.
  - Rank 0 is the **data server**.
  - Ranks 1 … N-1 are **compute nodes***.

This is the **master/worker pattern** (sometimes called **manager/worker** or **task farm**), **a software engineering** and **parallel programming pattern**, and it maps naturally onto MPI.

* Another terminology collision: Nodes here does not mean compute nodes (hardware) it means computational nodes (software)

Initial setup:
Uses MPI_Init, MPI_Comm_rank, MPI_Comm_size.
Checks that there are at least 3 MPI processes (1 server + 2 workers).
Prints human-readable memory sizes with pretty_bytes().

Data server node responsibilities (rank 0):
Allocates and fills two large vectors (A, B) with random floats.
Divides them evenly among compute nodes.
Sends slices of both vectors to each node with MPI_Send.

Waits for completion (MPI_Barrier), then gathers results via MPI_Recv.
Performs a serial addition locally and compares with MPI results.
Reports percentage error and prints first 10 elements for inspection.

Compute node responsibilities (ranks > 0):

Receive corresponding segments of A and B.

If GPU mode: call compute_node_gpu() → offload addition to GPU kernel.

If CPU mode: call compute_node() → add elements in a simple for-loop.

Send partial results back to the data server.

CPU Compute node responsibilities (ranks > 0):

Calls MPI_Comm_rank and MPI_Comm_size to get its rank and total process count.
Determines the rank of the data server (rank 0).
Uses MPI_Comm_split_type(MPI_COMM_TYPE_SHARED) to find how many MPI processes share the same physical host (used mainly for reporting).

Receive its data
Waits for two vector segments (A and B) from the data server.
Uses MPI_Recv twice — once for each input vector.
Reports when each vector arrives.

GPU Compute node responsibilities (ranks > 0):

Queries how many GPUs are present with cudaGetDeviceCount().
Checks that the number of GPUs ≥ number of local compute processes.
Selects a GPU in a **round-robin** manner using local_rank % n_gpus.
Sets the active device with cudaSetDevice().

Allocate memory
On **host:** allocate input/output vectors using malloc().
On **device:** allocate corresponding arrays using cudaMalloc().

Receive data from the data server
Uses MPI_Recv twice to receive its segments of input vectors A and B.
Copy data to the GPU
Transfers host arrays → device arrays with cudaMemcpy(..., cudaMemcpyHostToDevice).
Launch CUDA kernel
Sets the grid and block sizes
Copies the result back to the data service node.

cat Makefile

```
MPICXX   = mpic++
CUDA_HOME ?= $(shell echo $$CUDA_HOME)
MPI_INC  = $(shell $(MPICXX) -show | tr ' ' '\n' | grep '^-I' | sed 's/^-I//')
NVFLAGS  = -arch=sm_80 $(addprefix -I,$(MPI_INC))
CXXFLAGS = -O3
LDFLAGS  = -L$(CUDA_HOME)/lib64 -lcudart

.PHONY: gpu cpu clean

gpu:
	nvcc $(NVFLAGS) -c vecadd_gpu.cu -o vecadd_gpu.o
	$(MPICXX) $(CXXFLAGS) -c vecadd_mpi_gpu.c -o vecadd_mpi_gpu.o
	$(MPICXX) $(CXXFLAGS) vecadd_mpi_gpu.o vecadd_gpu.o $(LDFLAGS) -o
vecadd_mpi_gpu


cpu:
	$(MPICXX) $(CXXFLAGS) vecadd_mpi_cpu.c -o vecadd_mpi_cpu


clean:
	rm -f vecadd_mpi_gpu vecadd_mpi_cpu vecadd_mpi_gpu.o vecadd_mpi_cpu.o vecadd_gpu.o
```

Nvidia Compute Architectures

Arch / SM	Example GPUs	
Tesla (1.x) sm_10–13	GTX 280 (legacy)	
Fermi (2.x) sm_20–21	GTX 480, C2050	
Kepler (3.x) sm_35	K40	Xena Cluster
Maxwell (5.x) sm_50–53	GTX 980, Titan X	
Pascal (6.x) sm_60–62	GTX 1080, P100	
Volta (7.0) sm_70	Tesla V100	
Turing (7.5) sm_75	RTX 2080, T4	
Ampere (8.x) sm_80	A100	Hopper Cluster
Ada (8.9) sm_89	RTX 4090, 4080	
Hopper (9.0) sm_90	H100, H200	Easley Cluster
Blackwell (9.1) sm_91	B100, GB200 (2025+)	

Our Code needs the MPI and CUDA Libraries and Compilers (Xena)

```
mfricke@xena:~/workshops/intro_workshop/code/vecadd $ module load gcc/8.3.0-wbma
openmpi cuda

mfricke@xena:~/workshops/intro_workshop/code/vecadd $ make cpu
mpic++ -O3 vecadd_mpi_cpu.c -o vecadd_mpi_cpu

mfricke@xena:~/workshops/intro_workshop/code/vecadd $ make gpu
nvcc -arch=sm_80 -I/opt/spack/opt/spack/linux-centos7-haswell/gcc-8.3.0/openmpi-
4.0.5-rla727qcwwopjb4aabszxwcz3dx2dkez/include -c vecadd_gpu.cu -o vecadd_gpu.o
mpic++ -O3 -c vecadd_mpi_gpu.c -o vecadd_mpi_gpu.o
mpic++ -O3 vecadd_mpi_gpu.o vecadd_gpu.o -L/opt/spack/opt/spack/linux-centos7-
haswell/gcc-8.3.0/cuda-11.6.2-jeohzfecd7oanuifzokki7bzw4dixy5i/lib64 -lcudart -o
vecadd_mpi_gpu
```

```
mfricke@xena:~/workshops/intro_workshop/code/vecadd $ srun -p dualGPU -N 1 -n 3 --gpus 2 ./vecadd_mpi_gpu 3
srun: Account not specified in script or ~/.default_slurm_account, using latest project
You have been allocated one or more GPUs.
Assigning compute node to rank 1.
Assigning compute node to rank 2.
GPU support set to: true.
Dataserver will try to allocate 3 vectors of size 400 MB each.
Assigning data server node to rank 0.
DataServer (0): There are 3 MPI proceses on host xena-dual02.xena.alliance.unm.edu, 2 are compute nodes.
ComputeNode (1): There are 3 MPI proceses on host xena-dual02.xena.alliance.unm.edu, 2 are compute nodes.
ComputeNode (1): Detected 2 GPUs on host xena-dual02.xena.alliance.unm.edu.
ComputeNode (2): There are 3 MPI proceses on host xena-dual02.xena.alliance.unm.edu, 2 are compute nodes.
ComputeNode (2): Detected 2 GPUs on host xena-dual02.xena.alliance.unm.edu.
ComputeNode (1): Using GPU 1.
ComputeNode (2): Using GPU 0.
DataServer: Starting with rank 0.
DataServer (0): filling two input vectors with random floats between 1.000000 and 10.000000...
ComputeNode (2): Waiting for data with 52428800 elements from data server 0.
ComputeNode (1): Waiting for data with 52428800 elements from data server 0.
DataServer (0): finished generating 104857600 random vector elements.
DataServer (0): Sending vector A of size 52428800 to compute node with id 1.
DataServer (0): Sending vector B of size 52428800 to compute node with id 1.
DataServer (0): Sending vector A of size 52428800 to compute node with id 2.
DataServer (0): Sending vector B of size 52428800 to compute node with id 2.
ComputeNode (1): GPU partial vector addition complete.
ComputeNode (2): GPU partial vector addition complete.
ComputeNode (1): GPU result differs from CPU result by 100.000000%.
ComputeNode (1): CPU result (first 10 elements):
12.279320 7.367366 15.577477 9.876278 12.755514 4.859811 7.072846 13.353083 11.715575 10.303871
ComputeNode (1): GPU result (first 10 elements):
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
ComputeNode (2): GPU result differs from CPU result by 100.000000%.
ComputeNode (2): CPU result (first 10 elements):
7.379248 10.080418 8.239628 8.643396 9.883593 10.065593 13.895368 7.023689 8.434000 11.415873
ComputeNode (2): GPU result (first 10 elements):
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
DataServer (0): All compute nodes finished. Receiving partial results.
DataServer (0): Comparing parallel computation to serial computation...
DataServer (0): Performing serial computation...
DataServer (0): Comparing results...
DataServer (0): Error is 100.000000%.
DataServer (0): MPI result (first 10 elements):
0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
DataServer (0): serial result (first 10 elements):
12.279320 7.367366 15.577477 9.876278 12.755514 4.859811 7.072846 13.353083 11.715575 10.303871
```

Our Code needs the MPI and CUDA Libraries and Compilers (Easley)

```
mfricke@easley:~/workshops/intro_workshop/code/vecadd $ module load cuda openmpi
mfricke@easley:~/workshops/intro_workshop/code/vecadd module list


Currently Loaded Modules:
  1) openmpi/4.1.7-3ilj   2) cuda/12.9.0-rir3
mfricke@easley:~/workshops/intro_workshop/code/vecadd [master !x?]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2025 NVIDIA Corporation
Built on Wed_Apr__9_19:24:57_PDT_2025
Cuda compilation tools, release 12.9, V12.9.41
Build cuda_12.9.r12.9/compiler.35813241_0
mfricke@easley:~/workshops/intro_workshop/code/vecadd [master !x?]$ mpic++ --version
g++ (GCC) 11.5.0 20240719 (Red Hat 11.5.0-5)
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Our Code needs the MPI and CUDA Libraries and Compilers

```
mfricke@easley:~/workshops/intro_workshop/code/vecadd $ make
mfricke@easley:~/workshops/intro_workshop/code/vecadd $ srun --partition
debug --gpus 1 ./vecadd_mpi_gpu

You have been allocated one or more GPUs.
Job 109686 running on easley051
/users/mfricke/workshops/intro_workshop/code/vecadd/./vecadd_mpi_gpu: error while
loading shared libraries: libcudart.so.12: cannot open shared object file: No such
file or directory
srun: error: easley051: task 0: Exited with exit code 127
```

Let’s try to find this missing CUDA runtime library…

Our Code needs the MPI and CUDA Libraries and Compilers

```
mfricke@easley:~/workshops/intro_workshop/code/vecadd $ srun --partition debug --gpus 1 ./vecadd_mpi_gpu
```

```
mfricke@easley:~/workshops/intro_workshop/code/vecadd ldd ./vecadd_mpi_gpu
linux-vdso.so.1 (0x00007ffcd6bbb000)
libcudart.so.12 => not found
libmpi_cxx.so.40 => /opt/spack/opt/spack/linux-sapphirerapids/openmpi-4.1.7-3iljxdmzw2og3uklpqbbxgpujm4qtrar/lib/libmpi_cxx.so.40 (0x0000146c2d1f9000)
libmpi.so.40 => /opt/spack/opt/spack/linux-sapphirerapids/openmpi-4.1.7-3iljxdmzw2og3uklpqbbxgpujm4qtrar/lib/libmpi.so.40 (0x0000146c2d0b8000)
libstdc++.so.6 => /lib64/libstdc++.so.6 (0x0000146c2ce00000)
libm.so.6 => /lib64/libm.so.6 (0x0000146c2cd25000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000146c2d09c000)
```

Our Code needs the MPI and CUDA Libraries and Compilers

```
mfricke@easley:~/workshops/intro_workshop/code/vecadd $ module show cuda
-------------------------------------------------------------------------------------------
   /opt/spack/share/spack/lmod/linux-rocky9-x86_64/Core/cuda/12.9.0-rir3.lua:
-------------------------------------------------------------------------------------------
whatis("Name : cuda")
whatis("Version : 12.9.0")
whatis("Target : icelake")
whatis("Short description : CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables
dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).")
help([[Name   : cuda]])
help([[Version: 12.9.0]])
help([[Target : icelake]])
]])
help([[CUDA is a parallel computing platform and programming model invented by
NVIDIA. It enables dramatic increases in computing performance by
harnessing the power of the graphics processing unit (GPU).
…
setenv("CUDA_LIB64","/opt/spack/opt/spack/linux-icelake/cuda-12.9.0-rir3t44quppxkqmotgvfyvgarxgcydi2/lib64")
…
```

Our Code needs the MPI and CUDA Libraries and Compilers

```
mfricke@easley:~/workshops/intro_workshop/code/vecadd $ ls -l

$CUDA_LIB64/libcudart.so.12lrwxrwxrwx 1 tredfear systems 20 Jul 29 14:17
/opt/spack/opt/spack/linux-icelake/cuda-12.9.0-
rir3t44quppxkqmotgvfyvgarxgcydi2/lib64/libcudart.so.12 -> libcudart.so.12.9.37
```

Our Code needs the MPI and CUDA Libraries and Compilers

```
mfricke@easley:~/workshops/intro_workshop/code/vecadd $
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CUDA_LIB64

mfricke@easley:~/workshops/intro_workshop/code/vecadd $ ldd ./vecadd_mpi_gpu

linux-vdso.so.1 (0x00007ffe9ef49000)
libcudart.so.12 => /opt/spack/opt/spack/linux-icelake/cuda-12.9.0-
rir3t44quppxkqmotgvfyvgarxgcydi2/lib64/libcudart.so.12 (0x000014d30718e000)
libmpi_cxx.so.40 => /opt/spack/opt/spack/linux-sapphirerapids/openmpi-4.1.7-
3iljxdmzw2og3uklpqbbxgpujm4qtrar/lib/libmpi_cxx.so.40 (0x000014d307172000)
libmpi.so.40 => /opt/spack/opt/spack/linux-sapphirerapids/openmpi-4.1.7-
3iljxdmzw2og3uklpqbbxgpujm4qtrar/lib/libmpi.so.40 (0x000014d307031000)
```

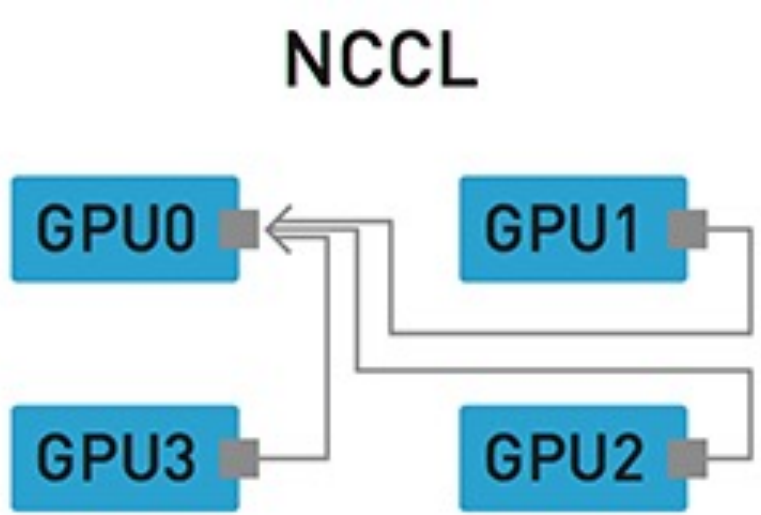
Our Code needs the MPI and CUDA Libraries and Compilers

```
srun --partition l40s --gpus 2 --ntasks 3 ./vecadd_mpi_gpu
```

```
srun --partition l40s --gpus 2 --ntasks 3
./vecadd_mpi_gpu
You have been allocated one or more GPUs.
Job 109771 running on easley055
Assigning compute node to rank 2.
GPU support set to: true.
Dataserver will try to allocate 3 vectors of size 400 MB each.
Assigning data server node to rank 0.
Assigning compute node to rank 1.
DataServer (0): There are 3 MPI proceses on host easley055, 2 are compute nodes.
DataServer: Starting with rank 0.
ComputeNode (1): There are 3 MPI proceses on host easley055, 2 are compute nodes.
ComputeNode (2): There are 3 MPI proceses on host easley055, 2 are compute nodes.
DataServer (0): filling two input vectors with random floats between 1.000000 and 10.000000...
ComputeNode (2): Detected 2 GPUs on host easley055.
ComputeNode (1): Detected 2 GPUs on host easley055.
ComputeNode (1): Using GPU 1.
ComputeNode (2): Using GPU 0.
ComputeNode (1): Waiting for data with 52428800 elements from data server 0.
ComputeNode (2): Waiting for data with 52428800 elements from data server 0.
DataServer (0): finished generating 104857600 random vector elements.
DataServer (0): Sending vector A of size 52428800 to compute node with id 1.
DataServer (0): Sending vector B of size 52428800 to compute node with id 1.
DataServer (0): Sending vector A of size 52428800 to compute node with id 2.
DataServer (0): Sending vector B of size 52428800 to compute node with id 2.
ComputeNode (1): GPU partial vector addition complete.
ComputeNode (2): GPU partial vector addition complete.
ComputeNode (1): GPU result differs from CPU result by 0.000000%.
ComputeNode (1): CPU result (first 10 elements):
12.279320 7.367366 15.577477 9.876278 12.755514 4.859811 7.072846 13.353083 11.715575 10.303871
ComputeNode (1): GPU result (first 10 elements):
12.279320 7.367366 15.577477 9.876278 12.755514 4.859811 7.072846 13.353083 11.715575 10.303871
ComputeNode (2): GPU result differs from CPU result by 0.000000%.
ComputeNode (2): CPU result (first 10 elements):
7.379248 10.080418 8.239628 8.643396 9.883593 10.065593 13.895368 7.023689 8.434000 11.415873
ComputeNode (2): GPU result (first 10 elements):
7.379248 10.080418 8.239628 8.643396 9.883593 10.065593 13.895368 7.023689 8.434000 11.415873
DataServer (0): All compute nodes finished. Receiving partial results.
DataServer (0): Comparing parallel computation to serial computation...
DataServer (0): Performing serial computation...
DataServer (0): Comparing results...
DataServer (0): Error is 0.000000%.
DataServer (0): MPI result (first 10 elements):
12.279320 7.367366 15.577477 9.876278 12.755514 4.859811 7.072846 13.353083 11.715575 10.303871
DataServer (0): serial result (first 10 elements):
12.279320 7.367366 15.577477 9.876278 12.755514 4.859811 7.072846 13.353083 11.715575 10.303871
```

Distributed GPU Computation (NCCL)

- The next obvious step is to parallelise our GPU computation using multiple GPUs. (GPU interconnects was the most common topic at the Hot Interconnects conference you saw at the beginning of the semester)
- Nvidia intentionally mirrored the API and terminology that MPI uses to make life easier for GPU programmers.
- Their distributed communication tool for GPUs is called NCCL (**NVIDIA* Collective Communication Library**) - 2017



RCCL — Radeon Collective Communications Library

*AMD and others have RCCL but in 2025 Nvidia is the undisputed ruler of GPU computation.

NCCL and MPI Correspondence

NCCL Function	MPI Equivalent	Purpose
ncclAllReduce	MPI_Allreduce	Every rank contributes a buffer and receives the reduced result (sum, max, min, etc). Used for synchronous averaging, gradient reduction, and global agreement.
ncclBroadcast	MPI_Bcast	One rank (the root) sends the same buffer to all other ranks. **In our pi calculator we distributed the number of rectangles to all the ranks with a broadcast.**
ncclAllGather	MPI_Allgather	Each rank contributes a buffer, and all ranks receive the concatenation of all buffers. Used for assembling full tensors from shards.
ncclReduceScatter	MPI_Reduce_scatter	Performs a reduction across all ranks, then scatters equal-sized segments of the reduced result back. Useful for sharded or partitioned model parallelism.
ncclReduce	MPI_Reduce	All ranks contribute data to a reduction, and only the root receives the reduced result. Used when only one rank needs the final value. **We gathered the partial sums with a reduce call in our pi calculator.**

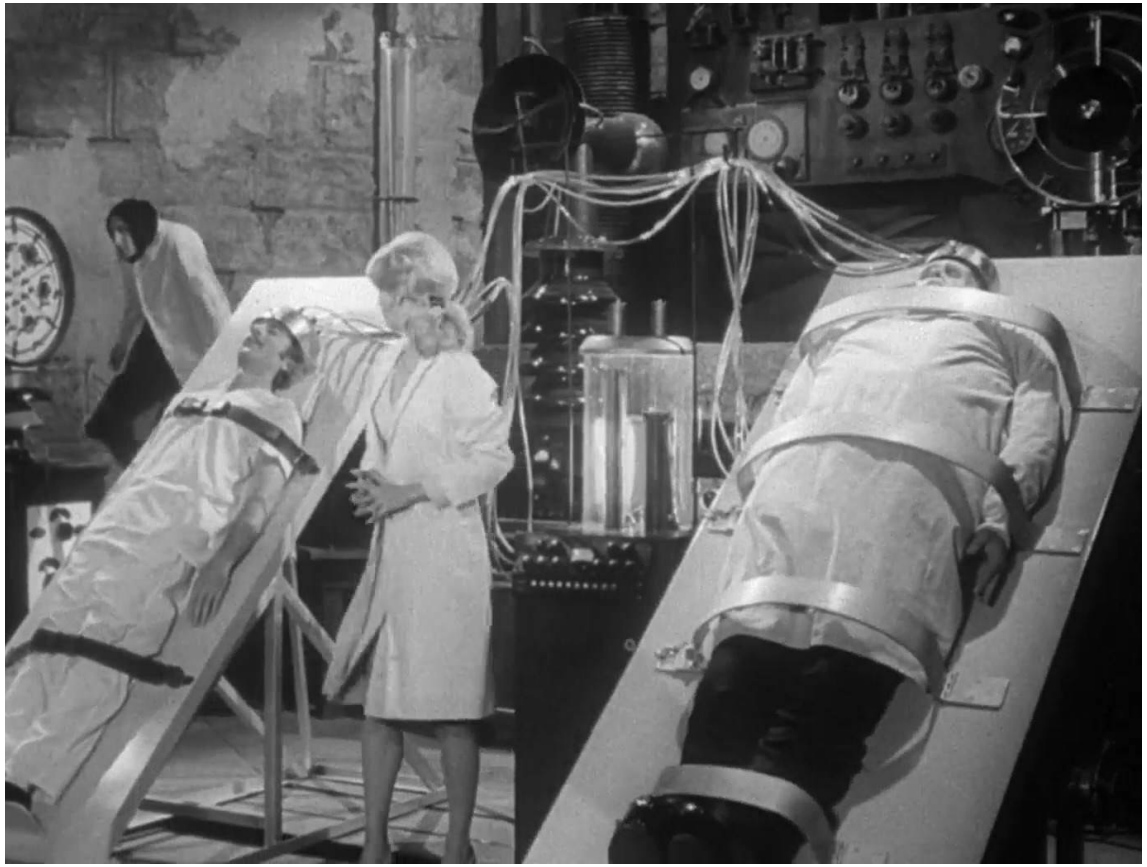
```
mfricke@easley:~/workshops/intro_workshop/code $ head calcPiMPI.py
import time
import sys
import numpy as np # Value of PI to compare to

###################### SETUP MPI - START ######################
from mpi4py import MPI           #Import the MPI library
comm = MPI.COMM_WORLD #Communication framework
root = 0 #Root process
rank = comm.Get_rank() #Rank of this process
num_procs = comm.Get_size() #Total number of processes


mfricke@easley:~/workshops/intro_workshop/code $ head calcPiMultiGPU.py
import sys
import time
import numpy as np
import cupy as cp
from mpi4py import MPI
from cupy.cuda import nccl

# ----------------------------------------------------------------
# Multi-GPU π estimation using NCCL and MPI
# ----------------------------------------------------------------
```

Your knowledge of MPI and CUDA come together
Comparison of calcPi implemented using NCCL and MPI

	MPI	NCCL version	Why it matters
Compute device	CPU loop	GPU vectorised CuPy ops	GPU does the heavy compute.
Work distribution	range(rank, N, size)	cp.arange(rank, N, size)	Same pattern, different execution model.
Communicator	Only MPI.COMM_WORLD	MPI + NcclCommunicator	NCCL needs its own GPU collective communicator.
Reduction	comm.reduce(mypi, MPI.SUM, root)	ncclComm.allReduce(...)	GPUs all need the global sum, so allreduce is used.
Memory location	Host numpy buffers	Device CuPy buffers	**NCCL operates directly on GPU memory.**

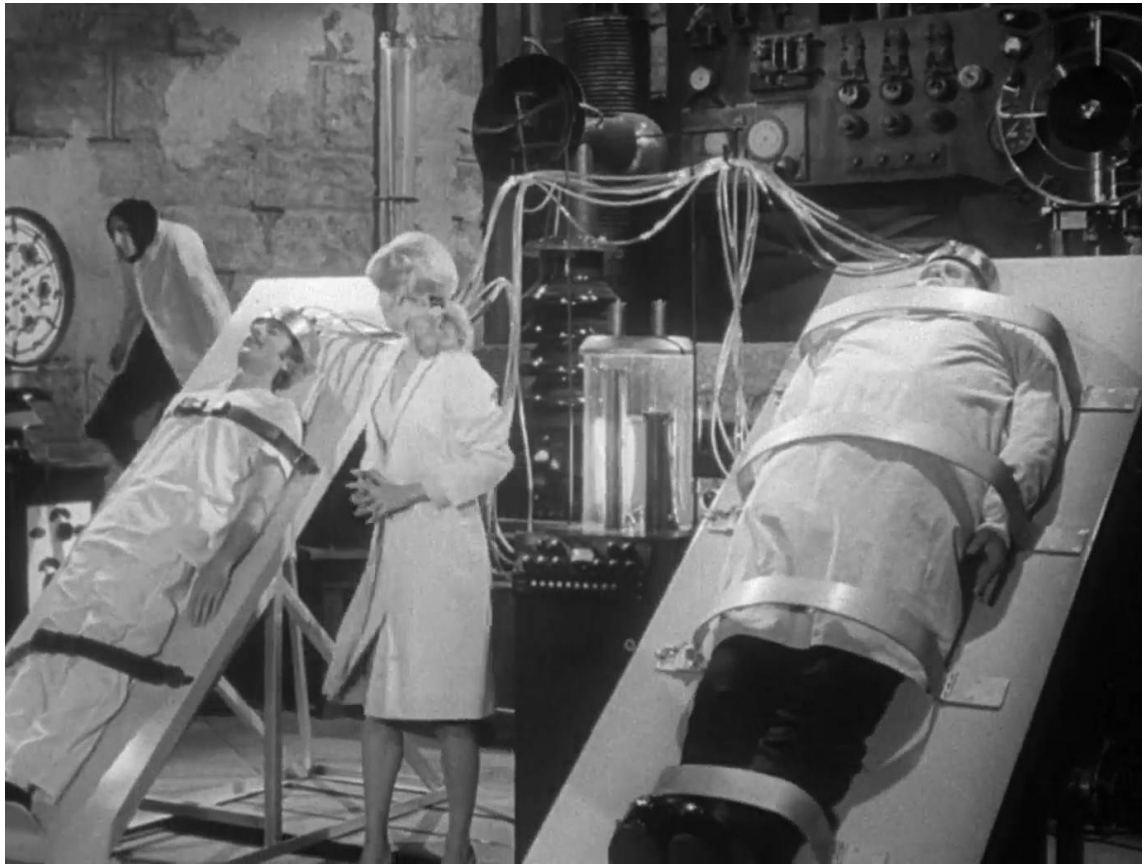
NCCL operates directly on GPU memory

- For speed NCCL bypasses the CPU to operate directly on VRAM (video RAM)
- When we scale up to multiple GPUs we need to use a network.
- You've seen one network technology that enables RDMA (Recall this is remote direct memory access)



NCCL used Infiniband

- Infiniband is built to provide RDMA
- Without Infiniband NCCL falls back to CPU based TCP communication which is an order of magnitude slower.
- Mellanox was the primary creator and vendor of Infiniband
- Nvidia bought Mellanox in 2020



Mellanox® TECHNOLOGIES + NVIDIA®

Core NCCL Code compared to MPI

Old MPI distributed Version

```
for i in range(rank, num_steps, num_procs):
    x = (i + 0.5) * step
    sum = sum + 4.0 / (1.0 + x * x)
    mypi = step * sum
```

NCCL Code: First allocate an array on the GPU

```
x = cp.arange(rank, N, size, dtype=cp.float64)
```

CuPy automatically runs these calculations on the GPU because the x array is on the GPU:

```
x = (x + 0.5) * step_size
partial_sum = (4.0 / (1.0 + x * x)).sum()
```

```
mfricke@easley:~/workshops/intro_workshop/code $ git pull
Already up to date.

mfricke@easley:~/workshops/intro_workshop/code $ module load miniconda3
This Miniconda3 installation uses the conda-forge channel. Conda-forge packages are generally
open-source and free for research and
educational use. Be sure to review the license agreements of software packages you install and
additional channels you enable.

mfricke@easley:~/workshops/intro_workshop/code $ conda activate intro_workshop
(intro_workshop) mfricke@easley:~/workshops/intro_workshop/code $$ salloc -p h100 -N1 -n2 --gpus-
per-task=1 --time=2:0
salloc: Granted job allocation 118402
salloc: Nodes easley051 are ready for job

mfricke@easley051:~/workshops/intro_workshop/code $ mpirun -np 2 python calcPiMultiGPU.py
1000000000
Pi = 3.1415926535897935 6009, (Diff=+4.4408920985006261616 9e-16)
Using 2 GPU(s): NVIDIA H100 NVL, NVIDIA H100 NVL
Setup:   0.3987 s
Compute: 0.0589 s
Comm:    1.6107 s
Total:   2.0683 s
mfricke@easley051:~/workshops/intro_workshop/code $ exit
exit
salloc: Relinquishing job allocation 118402
salloc: Job allocation 118402 has been revoked.
```

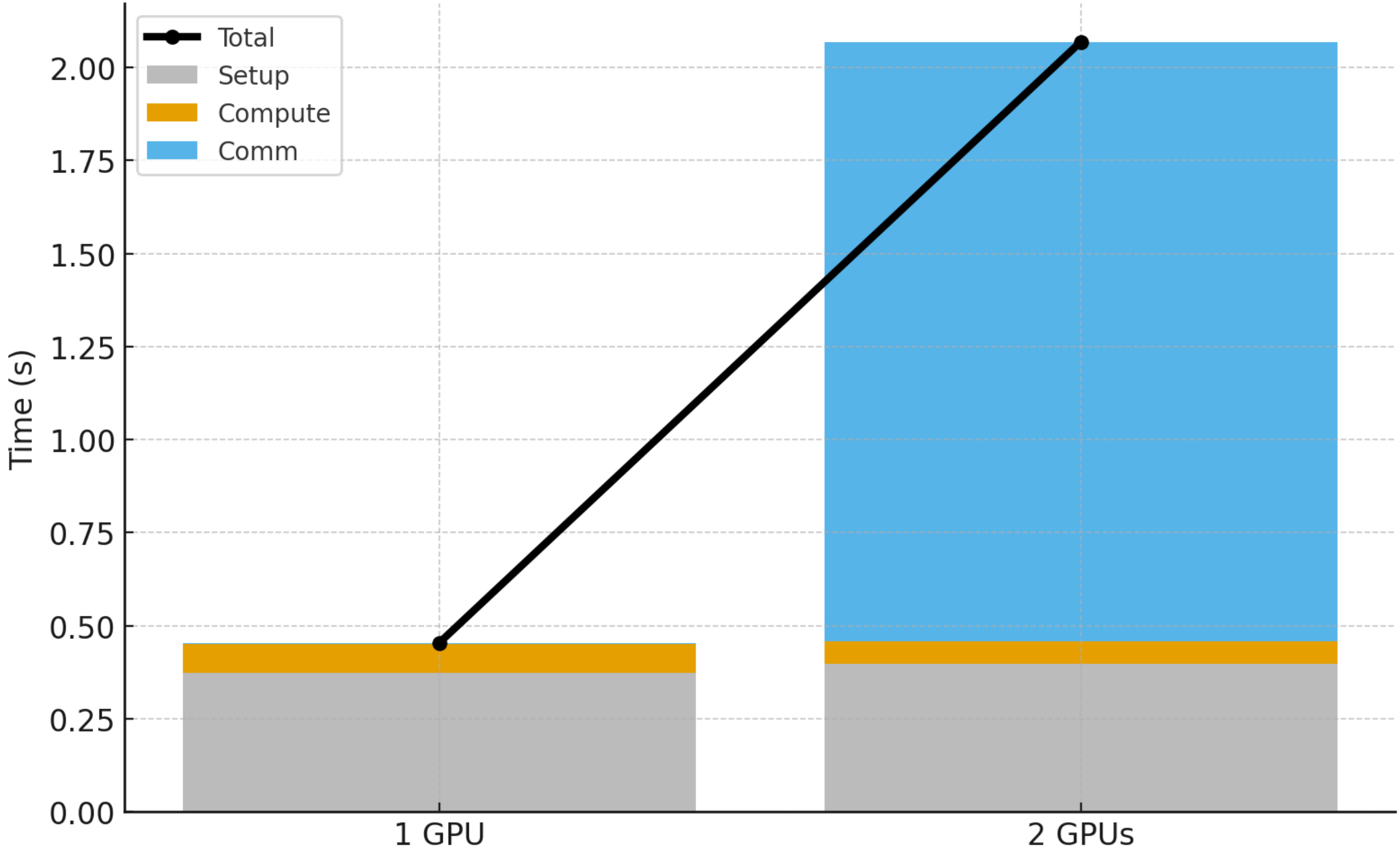
Version 1

Run our NCCL calcPiCode on 2 H100 GPUs

Version 1

```
(intro_workshop) mfricke@easley:~/workshops/intro_workshop/code [master !x?]$ salloc
-p h100 -N1 -n2 --gpus-per-task=1 --time=2:0
salloc: Granted job allocation 118407
salloc: Nodes easley051 are ready for job

mfricke@easley051:~/workshops/intro_workshop/code $ mpirun -np 1 python
calcPiMultiGPU.py 1000000000
Pi = 3.141592653589794004 18, (Diff=+8.881784197001252323 39e-16)
Using 1 GPU(s): NVIDIA H100 NVL
Setup:   0.3731 s
Compute: 0.0785 s
Comm:    0.0023 s
Total:   0.4539 s

mfricke@easley051:~/workshops/intro_workshop/code $ exit
exit
salloc: Relinquishing job allocation 118407
```

Run our NCCL calcPiCode on 1 H100 GPUs


Total
Setup
Compute
Comm
Time (s)
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
1 GPU
2 GPUs

Does not scale well. Even 10^9 rectangles are so easy for the GPU it doesn't stay busy enough to make multiple GPUs worth the overhead.

Each rectangle computes: $x = (i + 0.5)\,h, \qquad y = \dfrac{4}{1 + x^2}$

✓ **Each thread performs only about 4–6 floating point operations.**

✓ **A warp therefore performs about 128–192 FLOPs per π sample.**

✗ **But this is still far too little to keep a streaming processor busy.**

A modern GPU can perform trillions of FLOPs per second, so:

- Even one billion rectangles take only milliseconds of GPU compute.
- The GPU is vastly underutilised.
- There is no substantial work to distribute across multiple GPUs.

Does not scale well. When we try to scale up to bigger problems we run out of memory. (Gustafson scaling)

```
(intro_workshop) mfricke@easley:~/workshops/intro_workshop/code [master !x?]$ salloc -p h100 -N1 -n2 --gpus-per-task=1 --
time=2:0
salloc: Granted job allocation 118412
salloc: Nodes easley051 are ready for job
mfricke@easley051:~/workshops/intro_workshop/code $ mpirun -np 2 python calcPiMultiGPU.py 100000000000
Traceback (most recent call last):
  File "/users/mfricke/workshops/intro_workshop/code/calcPiMultiGPU.py", line 50, in <module>
    x = cp.arange(rank, N, size, dtype=cp.float64)
  File "/projects/shared/conda/envs/intro_workshop/lib/python3.10/site-packages/cupy/_creation/ranges.py", line 58, in arange
  File "cupy/cuda/memory.pyx", line 1118, in cupy.cuda.memory.SingleDeviceMemoryPool.malloc
  File "cupy/cuda/memory.pyx", line 1384, in cupy.cuda.memory.SingleDeviceMemoryPool._try_malloc
  File "cupy/cuda/memory.pyx", line 1387, in cupy.cuda.memory.SingleDeviceMemoryPool._try_malloc
cupy.cuda.memory.OutOfMemoryError: Out of memory allocating 400,000,000,000 bytes (allocated so far: 0 bytes).
  File "cupy/cuda/memory.pyx", line 1139, in cupy.cuda.memory.SingleDeviceMemoryPool._malloc
  File "cupy/cuda/memory.pyx", line 1384, in cupy.cuda.memory.SingleDeviceMemoryPool._try_malloc
  File "cupy/cuda/memory.pyx", line 1387, in cupy.cuda.memory.SingleDeviceMemoryPool._try_malloc
cupy.cuda.memory.OutOfMemoryError: Out of memory allocating 400,000,000,000 bytes (allocated so far: 0 bytes).
--------------------------------------------------------------------------
prterun detected that one or more processes exited with non-zero status,
thus causing the job to be terminated. The first process to do so was:


   Process name: [prterun-easley051-1653725@1,1]
   Exit code:    1
--------------------------------------------------------------------------
mfricke@easley051:~/workshops/intro_workshop/code $
```

10^{11}

Version 1

How We Make Multiple GPUs Worthwhile
solving the out of memory error will also solve our scaling problem)

Fix GPU memory use by processing the integration in batches.

- Rather than allocating an array of size *N* (often impossible), each GPU repeatedly computes π on a batch of rectangles sized to fit entirely in device memory.

Each batch represents a block of consecutive rectangles.

- We generate a batch, compute its contribution on the GPU, accumulate it locally, discard it, and generate the next batch.

Total computation grows in direct proportion to the number of batches.

- If the problem requires *K* batches, the total compute is $K \times compute\_per\_batch$.
- This allows *N* to grow arbitrarily large.

Batching enables scaling because more batches → more compute.

- As *N* increases, computation eventually dwarfs the single AllReduce communication, making multi-GPU usage effective.

Communication remains small, but its setup has high latency.

- NCCL must be initialised once per run; this setup cost dominates when the compute phase is short.
- Batching lengthens the compute phase, outweighing communication setup latency and allowing multiple GPUs to reduce the overall runtime.

How We Make Multiple GPUs Worthwhile

- **Batch the work** so GPU memory use stays fixed and N can grow arbitrarily large.
- We feed the GPU batches of rectangles to calculate.
- That allows us to grow the compute (N batches means N times the work) until compute outweighs communication overhead.
- The communication needed is tiny but is is very slow to setup.

  We set the size of each batch to 10 million.

  (You will be asked to find the sweet spot for this in HW5)

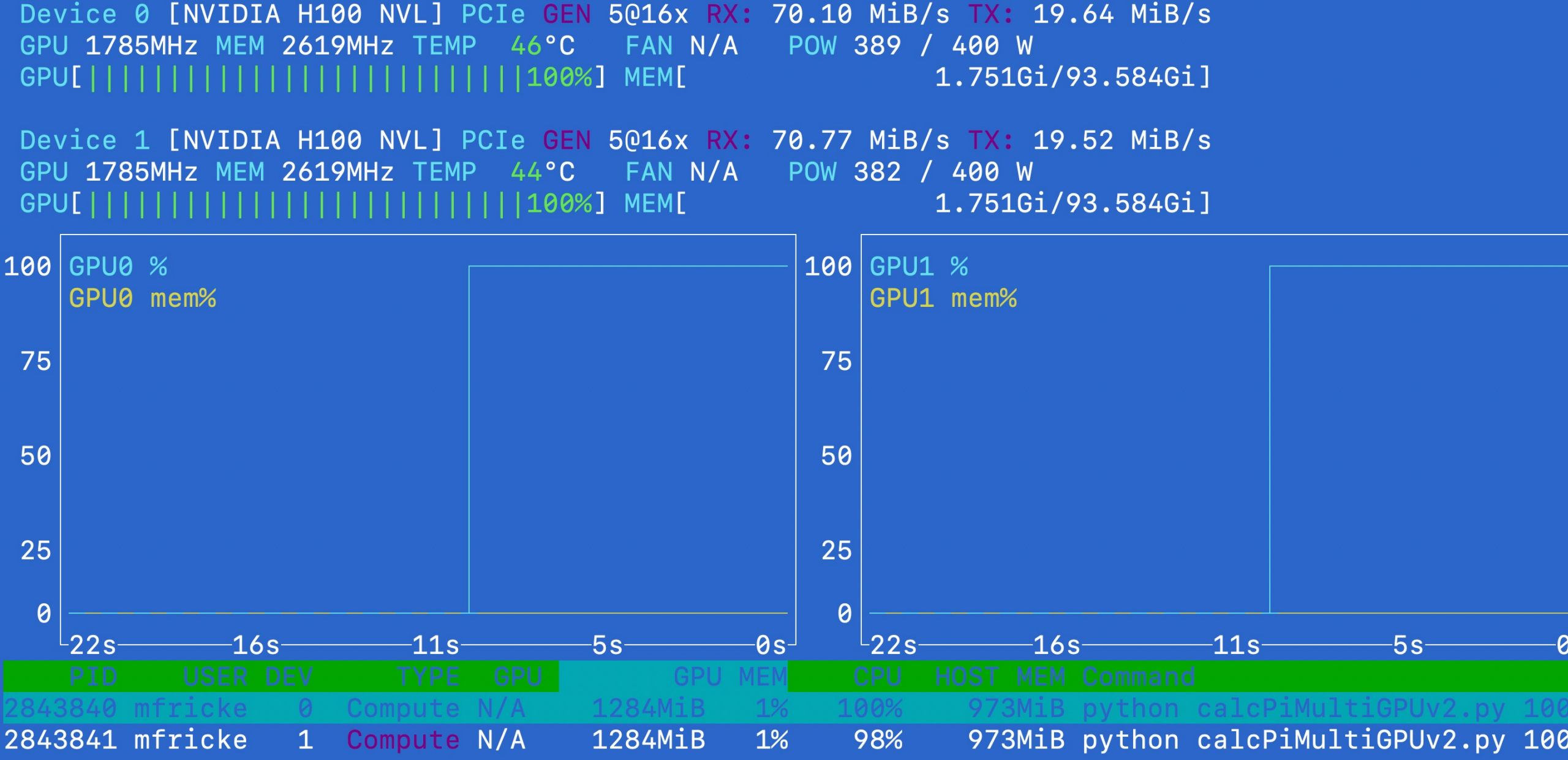
```
local_sum = cp.zeros((), dtype=cp.float64)

# Process this rank's indices in batches over k-space
# where global index is i = rank + k * size
for k_start in range(0, local_N, batch_size):
    k_end = min(local_N, k_start + batch_size)

    # k indices for this batch (local to this rank)
    k = cp.arange(k_start, k_end, dtype=cp.float64)

    # Global indices i handled by this rank in this batch
    i = rank + k * size

    # x positions in [0, 1]
    x = (i + 0.5) * step_size

    # Basic integrand
    y = 4.0 / (1.0 + x * x)

    local_sum += y.sum()

partial_sum = local_sum
```

Amdahl vs Gustafson for calcPi Multi GPU

- For fixed N (Amdahl's model), the π integrand is too cheap and the serial NCCL and launch overhead dominate.
- Adding GPUs provides no benefit at N ≈ 10^9 because the serial fraction stays large.
- With batching we shift into Gustafson's model where the problem size grows with GPU count.
- In this regime the parallel fraction dominates and scaling becomes nearly linear.
- This approach only becomes effective for **very large numbers of rectangles** (≈ 10^{10}–10^{11}) or when each thread performs substantial internal work.

Version 2

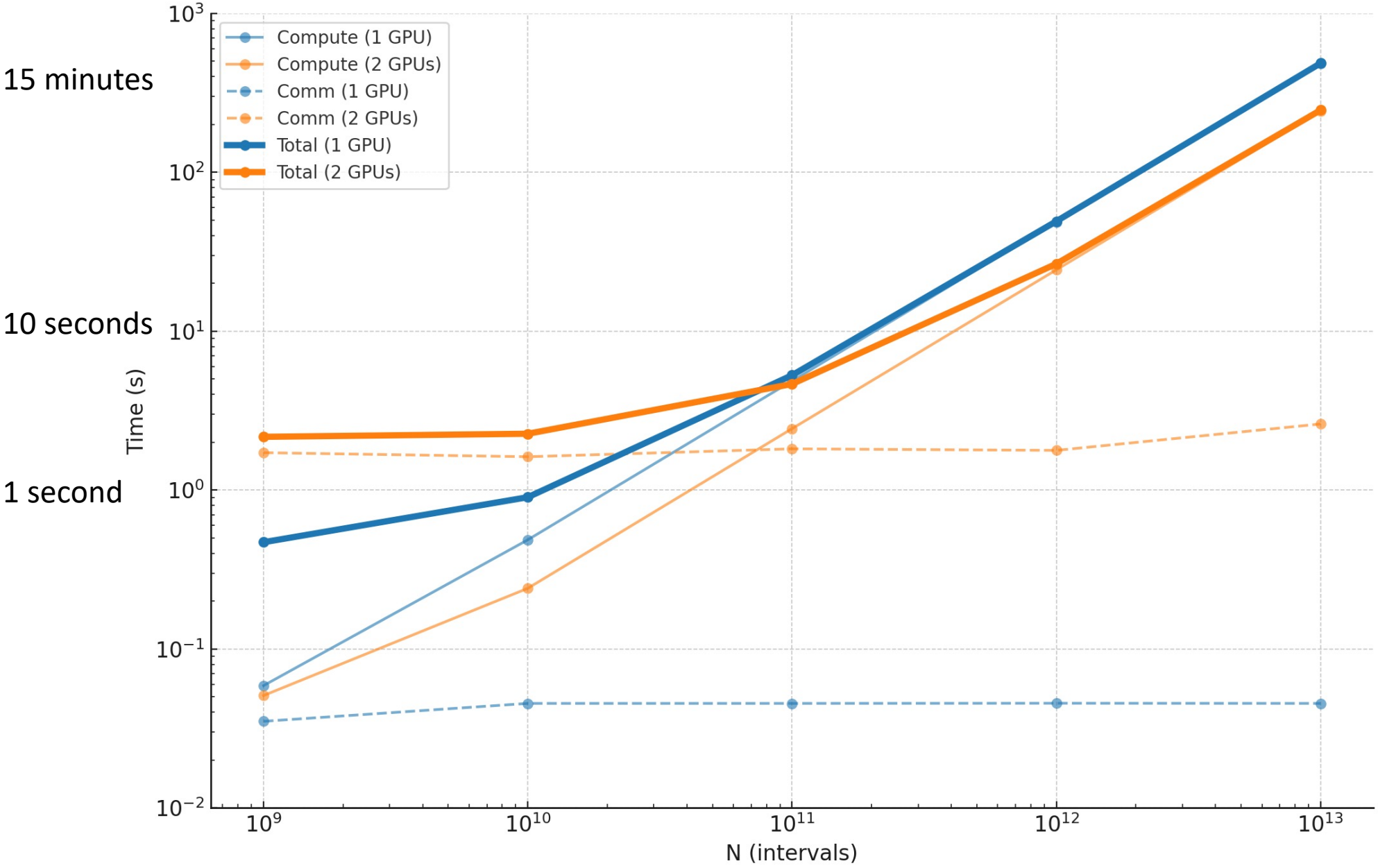
```
(intro_workshop) mfricke@easley054:~/workshops/intro_workshop/code $ mpirun -np 1 python
calcPiMultiGPUv2.py 1000000000000
Pi = 3.141592653589801997 78, (Diff=+8.881784197001252323 39e-15)
Using 1 GPU(s): NVIDIA H100 NVL
N = 1000000000000, batch_size = 10000000
Setup:   0.3724 s
Compute: 48.7816 s
Comm:    0.0456 s
Total:   49.1996 s

(intro_workshop) mfricke@easley054:~/workshops/intro_workshop/code $ mpirun -np 2 python
calcPiMultiGPUv2.py 1000000000000
Pi = 3.141592653589819761 35, (Diff=+2.664535259100375697 02e-14)
Using 2 GPU(s): NVIDIA H100 NVL, NVIDIA H100 NVL
N = 1000000000000, batch_size = 10000000
Setup:   0.4290 s
Compute: 24.3547 s
Comm:    1.7796 s
Total:   26.5633 s
```



Login to the node where calcPiMultiGPUv2.py is running and execute nvtop

- **1 GPU:** ~8.103 minutes
- **2 GPUs:** ~4.106 minutes

- **Difference:** ~4.00 minutes (2 GPUs are almost exactly twice as fast) for 10^{13}



Using 2 GPUs is faster for N > 10^{11} rectangles

Even with just 2 datapoints
We can estimate Amdahl scaling

Analysis depends a lot on the problem size

H100, $N = 10^{13}$

$$T_1 = 486.1675 \text{ s} \quad (1 \text{ GPU total time})$$
$$T_2 = 246.3418 \text{ s} \quad (2 \text{ GPU total time})$$

$$S(2) = \frac{T_1}{T_2} = \frac{486.1675}{246.3418} \approx 1.97$$

Amdahl's Law: $$S(p) = \frac{1}{f + \frac{1-f}{p}}$$

Fraction of the code that is inherently serial

Solve for f_{Amdahl} with $p = 2$:

$$f_{\text{Amdahl}} = \frac{\frac{1}{S(2)} - \frac{1}{2}}{1 - \frac{1}{2}} = \frac{\frac{1}{1.97} - 0.5}{0.5} \approx 0.0134$$

$$S_{\max} = \frac{1}{f_{\text{Amdahl}}} \approx \frac{1}{0.0134} \approx 74.6$$

This is the bound on estimated speedup no matter how many processors we provide.

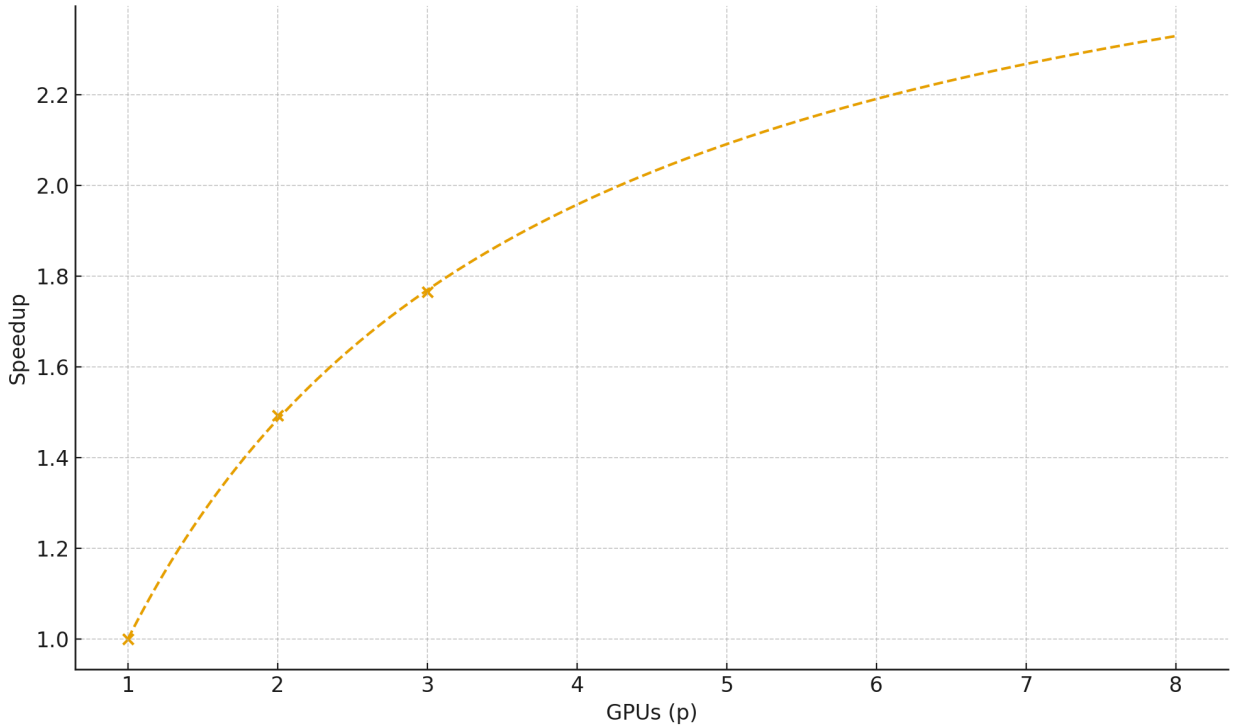
Conclusion: $f_{\text{Amdahl}} \approx 1.34\%$ serial under fixed-size workload.

Let's try with 3 L40s GPUs with 10^{11} rectangles

Version 2

```
(intro_workshop) mfricke@easley055:~/workshops/intro_workshop/code $ mpirun -np 1 python calcPiMultiGPUv2.py
100000000000
Pi = 3.14159265358980155369, (Diff=+8.437694987151189707 22e-15)
Using 1 GPU(s): NVIDIA L40S
N = 100000000000, batch_size = 10000000
Compute: 18.6714 s
Comm:    0.1621 s
Total:   18.9913 s
(intro_workshop) mfricke@easley055:~/workshops/intro_workshop/code $ mpirun -np 2 python calcPiMultiGPUv2.py
100000000000
Pi = 3.14159265358980466232, (Diff=+1.154631945610162802 04e-14)
Using 2 GPU(s): NVIDIA L40S, NVIDIA L40S
N = 100000000000, batch_size = 10000000
Setup:   0.1582 s
Compute: 8.4877 s
Comm:    0.4083 s
Total:   9.0542 s
(intro_workshop) mfricke@easley055:~/workshops/intro_workshop/code $ mpirun -np 3 python calcPiMultiGPUv2.py
100000000000
Pi = 3.14159265358978689875, (Diff=-6.217248937900876 62637e-15)
Using 3 GPU(s): NVIDIA L40S, NVIDIA L40S, NVIDIA L40S
N = 100000000000, batch_size = 10000000
Setup:   0.1575 s
Compute: 5.6725 s
Comm:    0.4082 s
Total:   6.2382 s
```

Amdahl Scaling Study



The speedup curves so we have to estimate
The percentage of the code that is inherently serial.

We also know it depends on the size of the problem

But with those caveats we estimate 35% for f.

Amdahl strong scaling (L40S, $N = 10^9$)

$$T_1 = 1.4804 \text{ s}, \quad T_2 = 0.9919 \text{ s}, \quad T_3 = 0.8390$$

$$S(2) = \frac{T_1}{T_2} = \frac{1.4804}{0.9919} \approx 1.4925,$$

$$S(3) = \frac{T_1}{T_3} = \frac{1.4804}{0.8390} \approx 1.7645.$$

Amdahl's law: $$S(p) = \frac{1}{f_{\text{Amdahl}} + \frac{1 - f_{\text{Amdahl}}}{p}}.$$

From $p = 2$: $$\frac{1}{S(2)} = f_{\text{Amdahl}} + \frac{1 - f_{\text{Amdahl}}}{2} = \frac{1}{2} + \frac{f_{\text{Amdahl}}}{2},$$

$$f^{(2)}_{\text{Amdahl}} = 2\left(\frac{1}{S(2)} - \frac{1}{2}\right) \approx 2\left(\frac{1}{1.4925} - 0.5\right) \approx 0.34,$$

From $p = 3$: $$\frac{1}{S(3)} = f_{\text{Amdahl}} + \frac{1 - f_{\text{Amdahl}}}{3} = \frac{1}{3} + \frac{2}{3} f_{\text{Amdahl}},$$

$$f^{(3)}_{\text{Amdahl}} = \frac{3}{2}\left(\frac{1}{S(3)} - \frac{1}{3}\right) \approx \frac{3}{2}\left(\frac{1}{1.7645} - 0.3333\right) \approx 0.35,$$

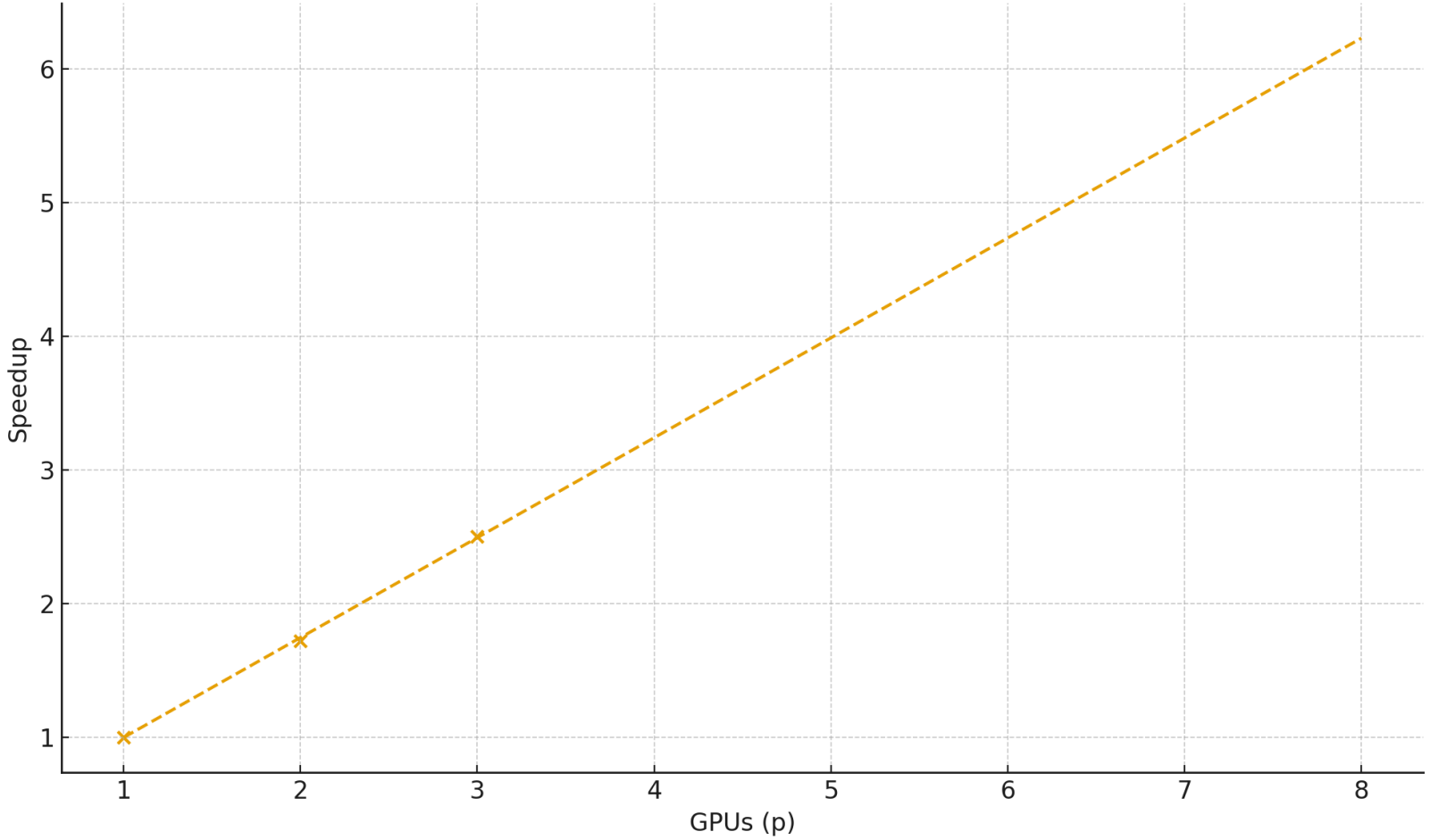
Combined estimate: $f_{\text{Amdahl}} \approx 0.35.$

Amdahl's fit will always be a parabola.
We have to do a non-linear fit to the curve to find *f*.

https://github.com/gmfricke/Juliaset.git contains a jupyter notebook to do the fit

```
mfricke@easley055:~/workshops/intro_workshop/code $ mpirun -np 1 python calcPiMultiGPUv3.py
1000000000
Pi = 3.141592653589794448 27, (Diff=+1.3322676295501878 4851e-15)
Using 1 GPU(s): NVIDIA L40S
N = 1000000000, batch_size = 10000000
Setup:   0.1576 s
Compute: 0.9625 s
Comm:    0.2389 s
Total:   1.3590 s
mfricke@easley055:~/workshops/intro_workshop/code $ mpirun -np 2 python calcPiMultiGPUv3.py
2000000000
Pi = 3.141592653589793116 00, (Diff=+0.0000000000000000 000e+00)
Using 2 GPU(s): NVIDIA L40S, NVIDIA L40S
N = 2000000000, batch_size = 10000000
Setup:   0.1617 s
Compute: 0.9639 s
Comm:    0.4497 s
Total:   1.5753 s
mfricke@easley055:~/workshops/intro_workshop/code $ mpirun -np 3 python calcPiMultiGPUv3.py
3000000000
Pi = 3.141592653589794004 18, (Diff=+8.8817841970012523 2339e-16)
Using 3 GPU(s): NVIDIA L40S, NVIDIA L40S, NVIDIA L40S
N = 3000000000, batch_size = 10000000
Setup:   0.1624 s
Compute: 0.9643 s
Comm:    0.5005 s
Total:   1.6272 s
```

Gustafson Scaling Study



Even though we have linear speedup the slope is about 0.75.

That tells us right away that Gustafson f value is about 25%

Gustafson weak scaling (L40S, $N, 2N, 3N$)

$$T_1 = 1.3590 \text{ s}, \quad T_2 = 1.5753 \text{ s}, \quad T_3 = 1.6272$$

$$S_G(1) = 1,$$

$$S_G(2) = 2\,\frac{T_1}{T_2} = 2\,\frac{1.3590}{1.5753} \approx 1.73,$$

$$S_G(3) = 3\,\frac{T_1}{T_3} = 3\,\frac{1.3590}{1.6272} \approx 2.51.$$

Gustafson’s fit will always be a line.
The slope gives us *f*.

$$\text{Gustafson model:} \quad S_G(p) = p - f_{\text{Gustafson}}(p-1).$$

$$\text{From } p = 2: \quad f^{(2)}_{\text{Gustafson}} = 2 - S_G(2) \approx 2 - 1.73 \approx 0.27,$$

$$\text{From } p = 3: \quad f^{(3)}_{\text{Gustafson}} = \frac{3 - S_G(3)}{2} \approx \frac{3 - 2.51}{2} \approx 0.25,$$

$$\text{Combined estimate:} \quad f_{\text{Gustafson}} \approx 0.25.$$