GPU Programming Workshop

Introduction to OpenACC and GPU-Aware MPI

NCAR Special Technical Projects
&
Client Support Group

May 20, 2021

UCAR

Overview

- Introduction to GPU architecture, key concepts, and terminologies
- OpenACC programming model
- Connecting to Cloud & Accessing Code
- Section 1
  - Coding Practice - Matrix Multiplication with OpenACC
  - Profiling - NVprof and PCAST
- Section 2
  - Coding Practice - MPI Stencil Operation with OpenACC

The code samples and versions of the slides are available at the link below:
https://github.com/NCAR/GPU_workshop

Task 1: Accessing the Workshop Instance

- Download the ssh key given to you by email yesterday
- Change file permissions of the private key

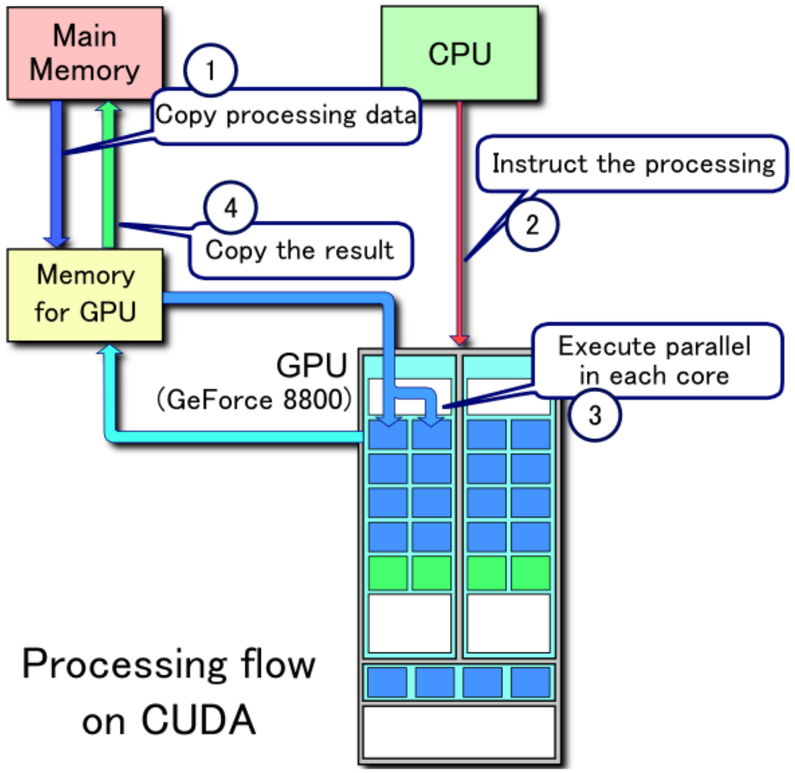
```
chmod 600 <path_to_private_key>
```

- Use ssh from a terminal window to connect to the instance IP address given to you
  - -i : Specifies identity file/private key
  - -p : Specifies the port number to use to connect

```
ssh -i <path_to_private_key> -p 22 <instance IP>
```

Find your instance IP and username here or ask your mentor later:
ADD link here

Developing a GPU Program



https://en.wikipedia.org/wiki/CUDA

Example of a GPU process flow*

1. Copy data from main memory to GPU memory

2. CPU initiates threads on the GPU and launches the kernel on GPU

3. GPU executes parallel code

4. Copy the results from GPU memory to main memory

* This flow occurs in both OpenACC/CUDA.

OpenACC

OpenACC Introduction

- OpenACC is a directives-based parallel programming model that is designed for performance and portability
- Enables easy conversion of sequential CPU based code to GPUs by adding pragmas
- General syntax:
  - C - #pragma acc [directive] [clauses]
  - FORTRAN - !$acc [directive] [clauses]
- Multiple directives can be chained together in the same line and applied to the same blocks of code
- There is no need for explicit allocation/copying of data to and from the GPU in OpenACC

OpenACC Introduction

- A vector is a group of threads that execute an operation on a chunk of data at once
- A worker executes one vector or SIMD operation. Workers in the same gang are able to share resources such as memory
- Gangs are a group of workers which in turn execute vectorized / SIMD operations

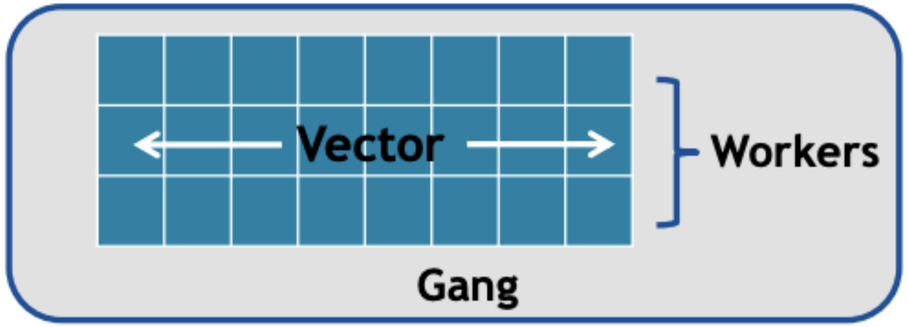


Image from https://www.olcf.ornl.gov tutorial slides

Kernels vs Parallel construct

- Kernels construct - The compiler is able to detect parts of the code that can be parallelized and parallelizes it. Sometimes it can parallelize the code better than the programmer.
- The downside is the compiler is conservative, and may not parallelize code that it deems unsafe

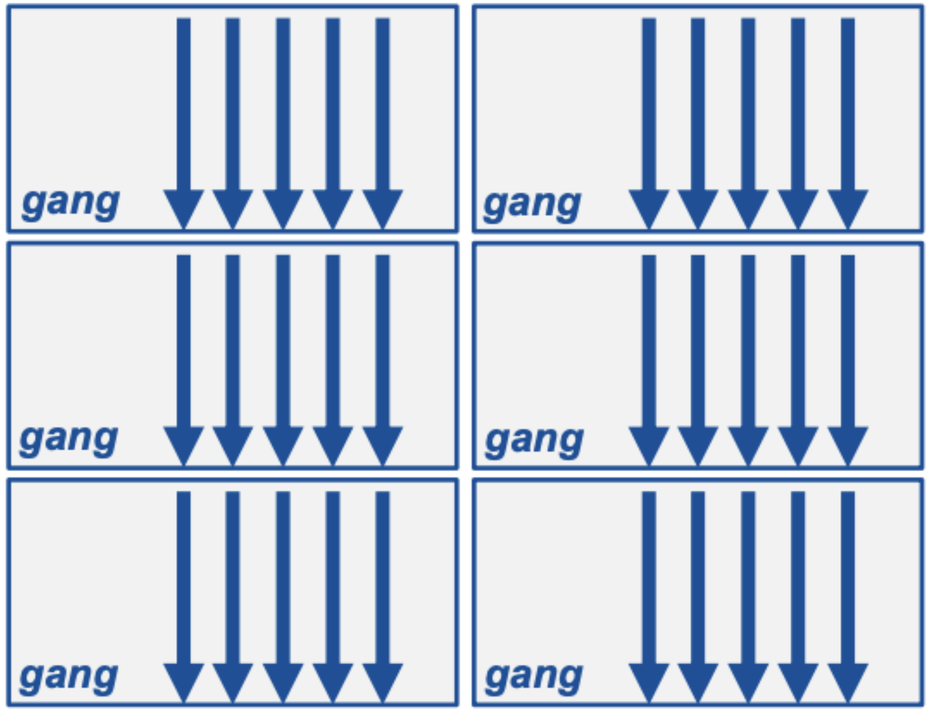
OpenACC Introduction

When you use "#pragma acc parallel", multiple gangs of workers are launched. The "#pragma acc loop" directive tells the compiler how to evenly split the work across the gangs.

C/C++

```
#pragma acc parallel
{
  #pragma acc loop
  for(int i = 0; j < N; i++)
    a[i] = 0;
}
```

- parallel directive tells the compiler to launch one or more parallel gangs.
- without the loop directive each of the gangs will run the same loop redundantly which is not efficient.
- when the compiler encounters the loop directive it parallelizes the block of code immediately after it, and splits the work across each of the gangs equally.



OpenACC Directives/Constructs

Construct	C code	Description
Kernels	#pragma acc kernels [clauses]	Surrounds loops to be executed on the GPU
Parallel	#pragma acc parallel [clauses]	Launches a number of gangs in parallel each with a number of workers, each with vector or SIMD operations
Loop	#pragma acc loop [clauses]	Applies to the immediately following loop or nested loops and describes the type of parallelism to execute these loops on the GPU
Data	#pragma acc data [clauses]	Defines a region of the program within which data is accessible by the GPU
Host data	#pragma acc host_data [clauses]	Makes the address of the device data available on the host
Cache	#pragma acc cache (list)	Added to the top of the loop. The elements in the list are cached in the software managed data cache
Update	#pragma acc update [clauses]	Copies data between the host memory and the data allocated on the GPU memory, and vice-versa
Clause - Collapse	#pragma acc loop collapse(n)	The collapse clause is used to specify how many tightly nested loops are associated with the loop clause.
Clause - Reduction	#pragma acc loop reduction(+:temp)	The reduction clause specifies a reduction operator and one or more vars.

Data construct

- *#pragma acc data [clauses]* defines a region below it within which the data is accessible by the GPU.

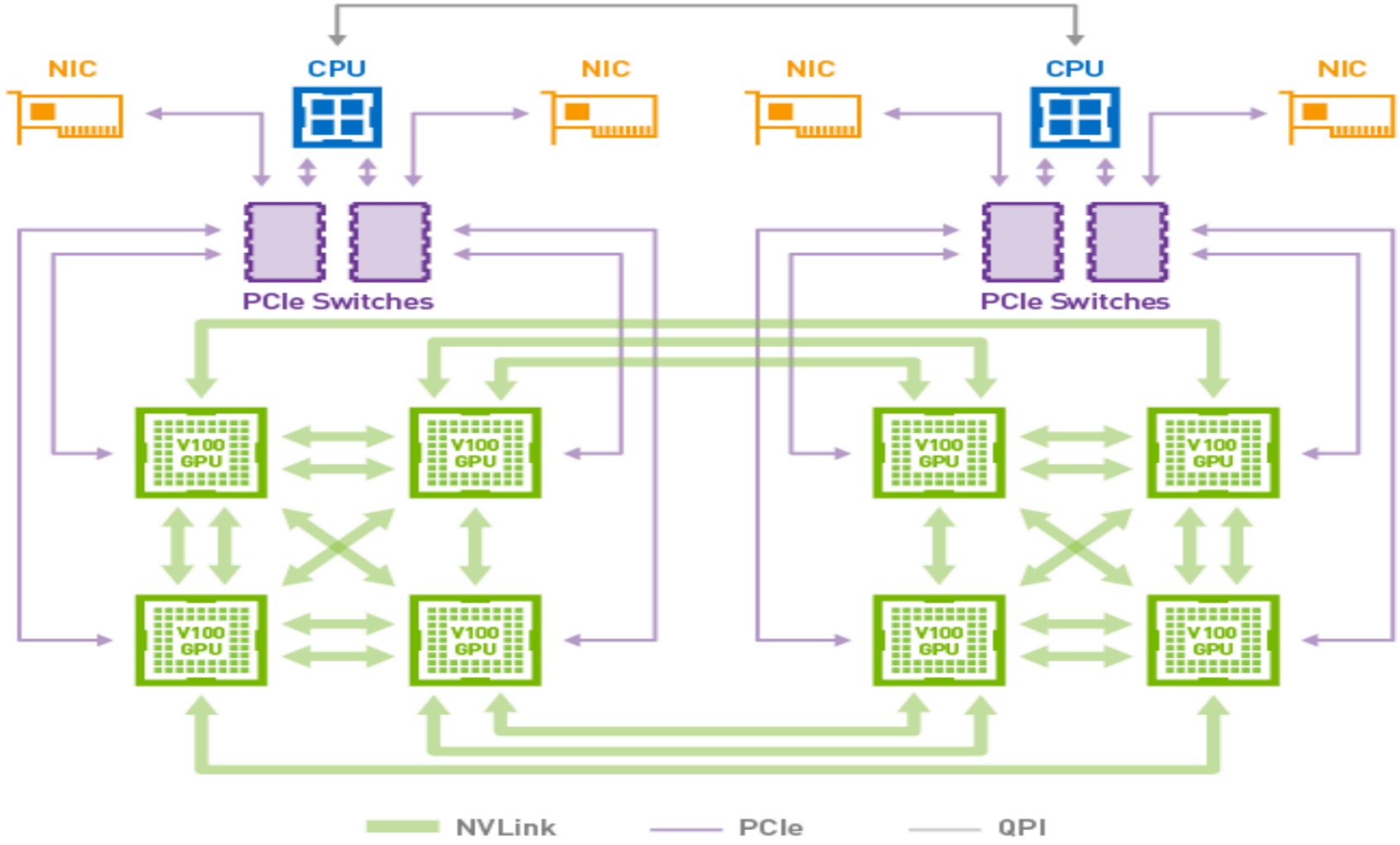
Clause	Description
copy(list)	1. Allocates the data in list on the GPU 2. Copies the data from the host to the GPU when entering the region 3. Copies the data from the GPU to the host when exiting the region
copyin(list)	Allocates the data in list on the GPU and copies the data from the host to the GPU when entering the region.
copyout(list)	Allocates the data in list on the GPU and copies the data from the GPU to the host when exiting the region.
create(list)	Allocates the data in list on the GPU, but does not copy data between the host and device.
present(list)	The data in the list must be already present on the GPU from some containing data region, which then can be found and used.

The information in this table and more about different directives and clauses at:
https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf

Coding practice~30 mins

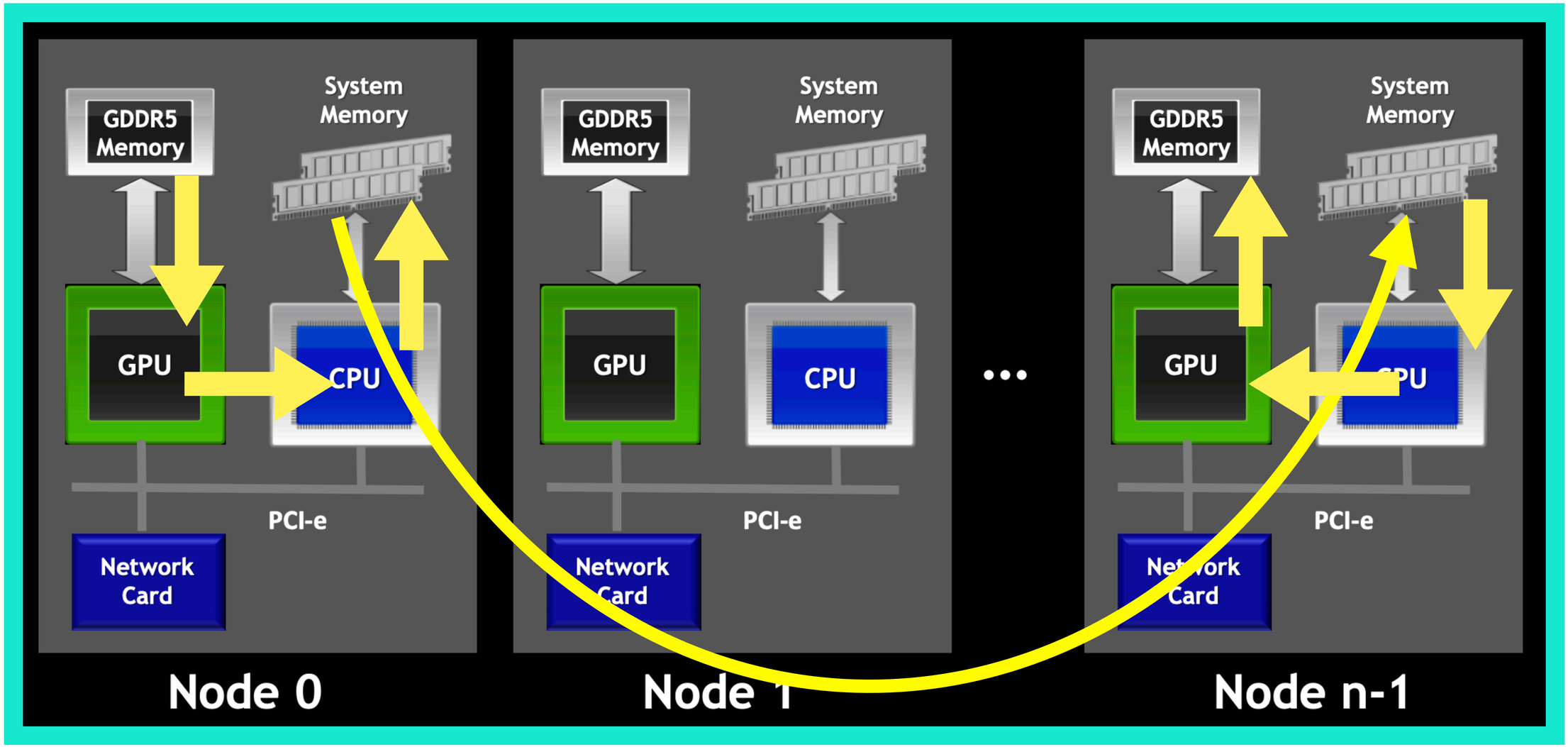
Multi-GPU

Multi-GPU node architecture



Naive GPU-GPU MPI

-Implements a double copy message passing

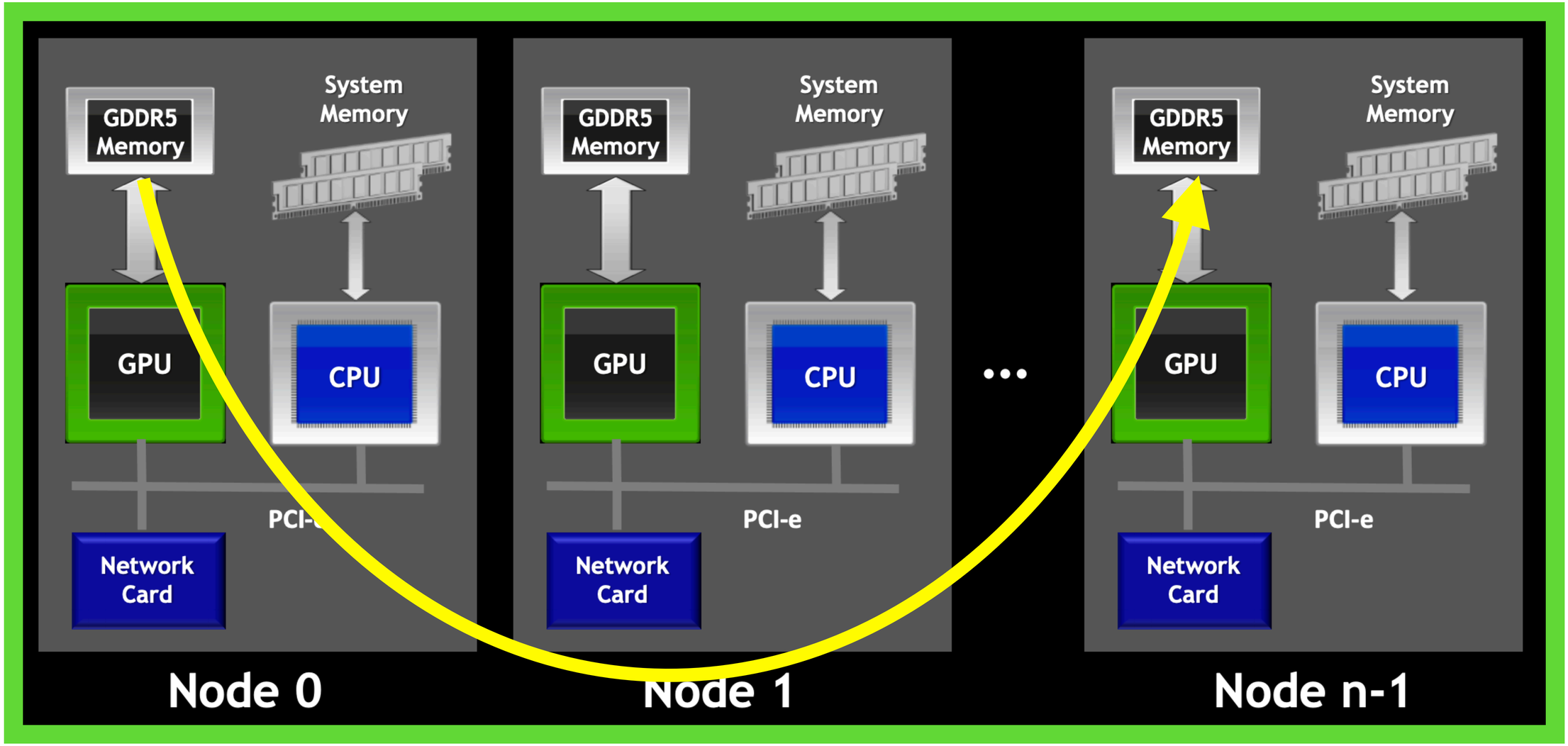


https://on-demand.gputechconf.com/gtc/2013/presentations/S3047-Intro-CUDA-Aware-MPI-NVIDIA-GPUDirect.pdf

https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

CUDA-aware MPI with GPU Direct RDMA

- Implements “zero copy” message passing!



Naive vs CUDA aware MPI support

MPI-Implementation with CUDA-Aware

- MVAPICH2 v(1.8/1.9b+)
- OpenMPI v(1.7+)
- CRAY MPI v(MPT 5.6.2+)
- IBM Spectrum MPI v(8.3+)
- SGI MPI v(1.08+)

Naive GPU MPI

```
#pragma acc update host(s_buf[0:size] )
MPI_Send(s_buf,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);
```

CUDA-Aware with GPU-Direct RDMA

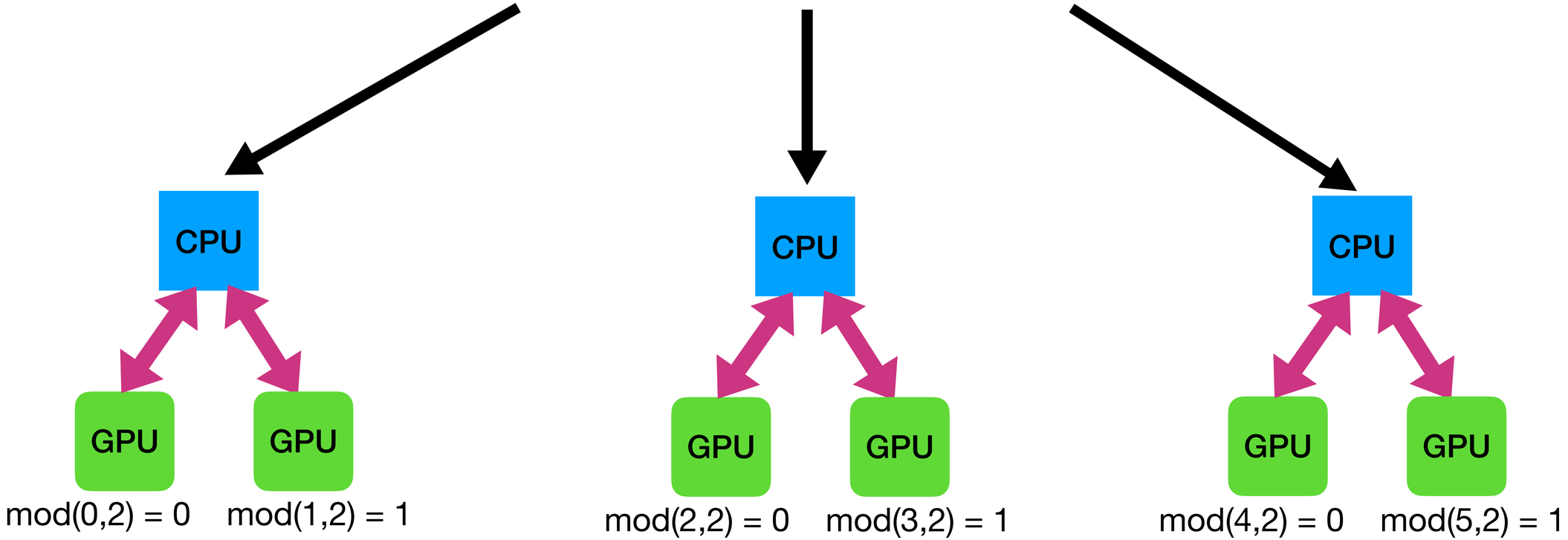
```
#pragma acc host_data use_device(s_buf)
MPI_Send(s_buf,size,MPI_CHAR,1,tag,MPI_COMM_WORLD);
```

Launching MPI ranks and pinning task to GPU

acc_set_device_num(gpu_r , acc_device_nvidia)

mod(rank,D)=gpu_r
D = number of gpus/node

mpirun -n 6 -npernode 2 <args> ./mpi_test



npernode = 2
Ranks = 0, 1
D = 2

npernode = 2
Ranks = 2, 3
D = 2

npernode = 2
Ranks = 4, 5
D = 2

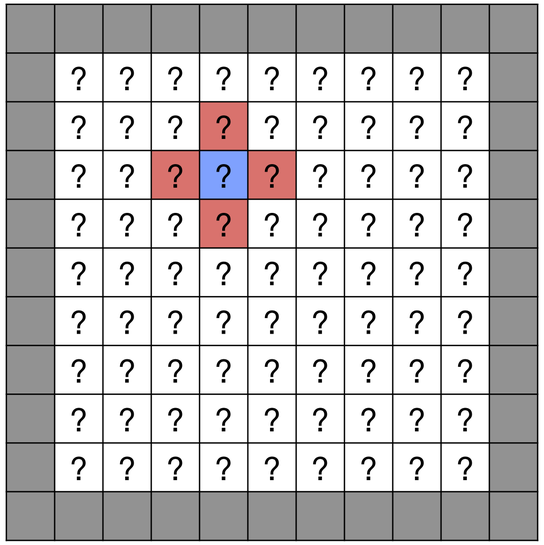
MPI-OpenACC: Constructs and Clauses Used

Construct	Clause	C code	Description
Loop	collapse	#pragma acc loop [clauses]	Applies to the immediately following loop or nested loops and describes the type of parallelism to execute these loops on the GPU.
	reduction	#pragma acc loop reduction(+:temp)	The reduction clause specifies a reduction operator and one or more vars (e.g. +, -, max, min).
	present	#pragma acc loop present(A[start:count])	Compiler hint that the variable is already in device memory. If not a runtime error occurs.
Data	copyin	#pragma acc enter data copyin(A[start:count])	Allocates the data in list on the GPU and copies the data from the host to the GPU when entering the region.
	create	#pragma acc enter data create(A[start:count])	Allocates the data in list on the GPU, but does not copy data between the host and device.
	copyout	#pragma acc enter data copyin(A[start:count])	Allocates the data in list on the GPU and copies the data from the GPU to the host when exiting the region.
Update	device	#pragma acc update device(A[start:count])	Updates the GPU copy of a variable with the version that is on the host.
	self or host	#pragma acc update self(A[start:count])	Updates the host copy of a variable with the version that is on the GPU.
Host data	use_device	#pragma acc host_data use_device(A)	Directs the compiler to use the device address of any entry in list.

Most text copied from https://www.openacc.org/sites/default/files/inline-files/OpenACC_2.5_ref_guide_1.pdf or https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf

MPI-OpenACC: Problem Conception

Given a 2D grid of points with a border of known values, find the steady-state of the interior points



The value of the interior points will be found by iteratively taking the average of the 4 points surrounding it

$$M_{new}[i\ ,\ j] = 1/4(M[i+1\ ,\ j]+M[i\ ,\ j+1]+M[i\ ,\ j-1]+M[i-1\ ,\ j])$$

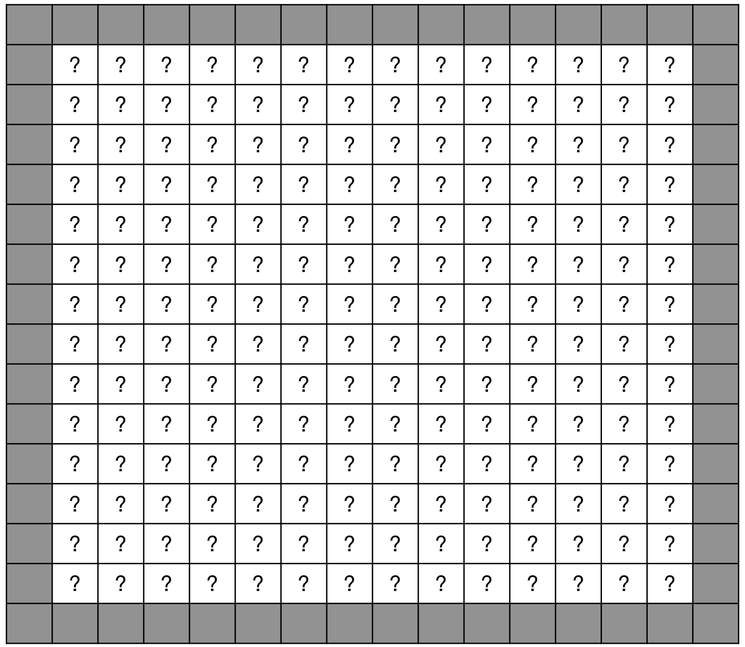
This forms a "stencil" that is applied to all points in the interior

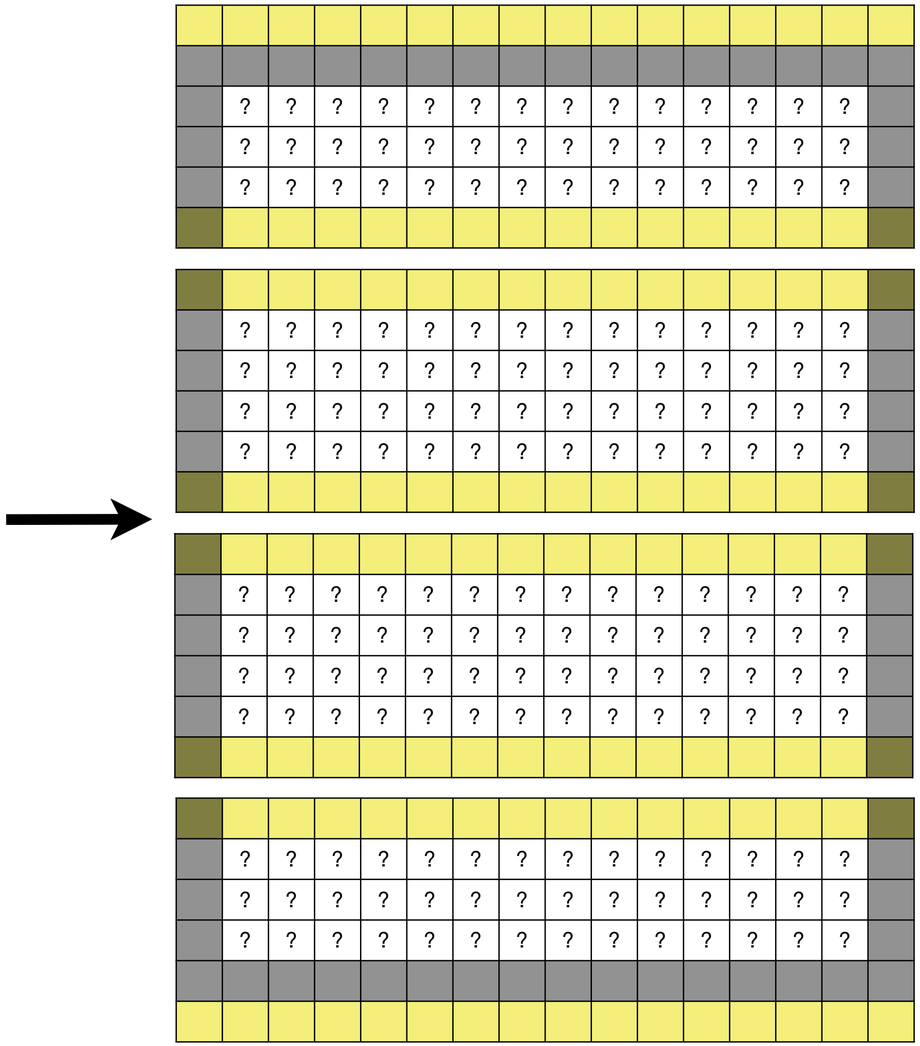
Iteration will cease once the difference between M_new[i , j] and M[i , j] is below a certain threshold

Aside: Finding the steady-state is an application of the Laplace equation. Rearranging the equation above and creating a flattened version of the interior points puts the problem in a form that fits for the Jacobi method. So the code is an iterative Laplace Jacobi solver

MPI-OpenACC: Laplace-Jacobi Distributed Solver Code

Matrix with 16x16 points split among 4 processors:





A large matrix can take a long time to update, but the code is embarrassingly parallel. So, we can divide the matrix among separate ranks with a "ghost area" around each to enable exchanging of information.

Coding practice~30 mins

Recap

- Matrix multiplication with shared memory
- OpenACC and CUDA coding practice
  - PCAST
  - FMA flags
  - Nsys profiling

Extra Coding Practice: Matrix Mult Scaling Curves

- Scenario: We'd like you to gather performance data to compare how the GPU and CPU Matrix Multiplication code performs as the size of the matrices increases. We know it's not fair to compare the times for a single GPU to a single-threaded CPU, so **we'd like try using OpenACC's Multicore CPU capabilities** to run the code on a fully-subscribed CPU node. (Documented here: https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/index.html#using-openacc)
- Then, for increasing sizes of square matrix multiplication, you can fill out this example table with performance data. Feel free to optimize the GPU and CPU code - the fastest times (with correct results) win!

Square Mat Dimension	Single GPU Execution Time	Single CPU Node Execution Time
256		
512		
1024		
2048		
4096		
8192		

References

1. https://www.olcf.ornl.gov/wp-content/uploads/2020/04/OpenACC-Course-2020-Module-1.pdf
2. https://www.olcf.ornl.gov/wp-content/uploads/2020/02/OpenACC_Course_2020_Module_2.pdf
3. https://www.olcf.ornl.gov/wp-content/uploads/2020/06/OpenACC_Course_2020_Module_3_updated.pdf
4. https://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf
5. https://docs.nvidia.com/hpc-sdk/compilers/openacc-gs/
6. https://docs.computecanada.ca/wiki/OpenACC_Tutorial_-_Optimizing_loops

Reference Slides

- Hardware Introduction - Slides 27 to 35
- Coding practice 1 - Slides 36 to 48
- Coding practice 2 - Slides 49 to 58

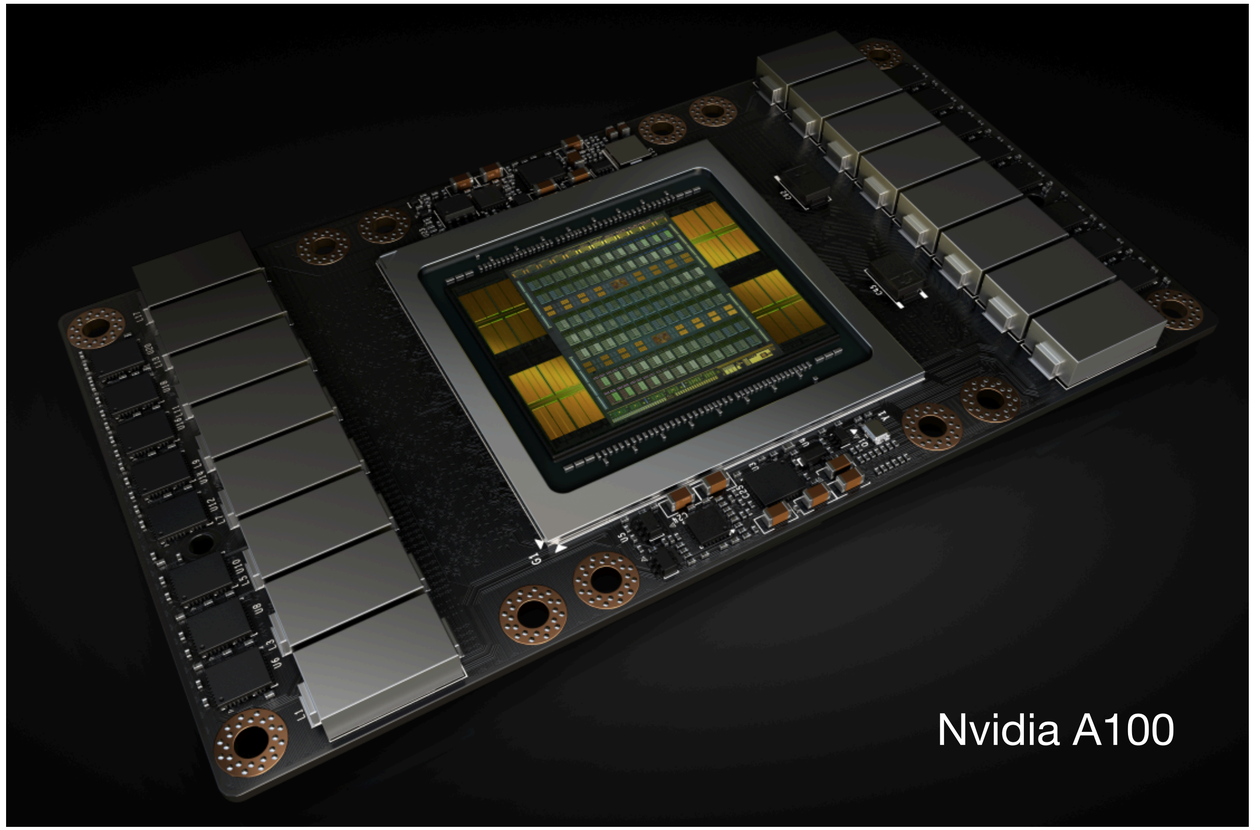
Thank you for your time!

GitHub Repository with slides:**https://github.com/NCAR/GPU_workshop**

GPU Architecture

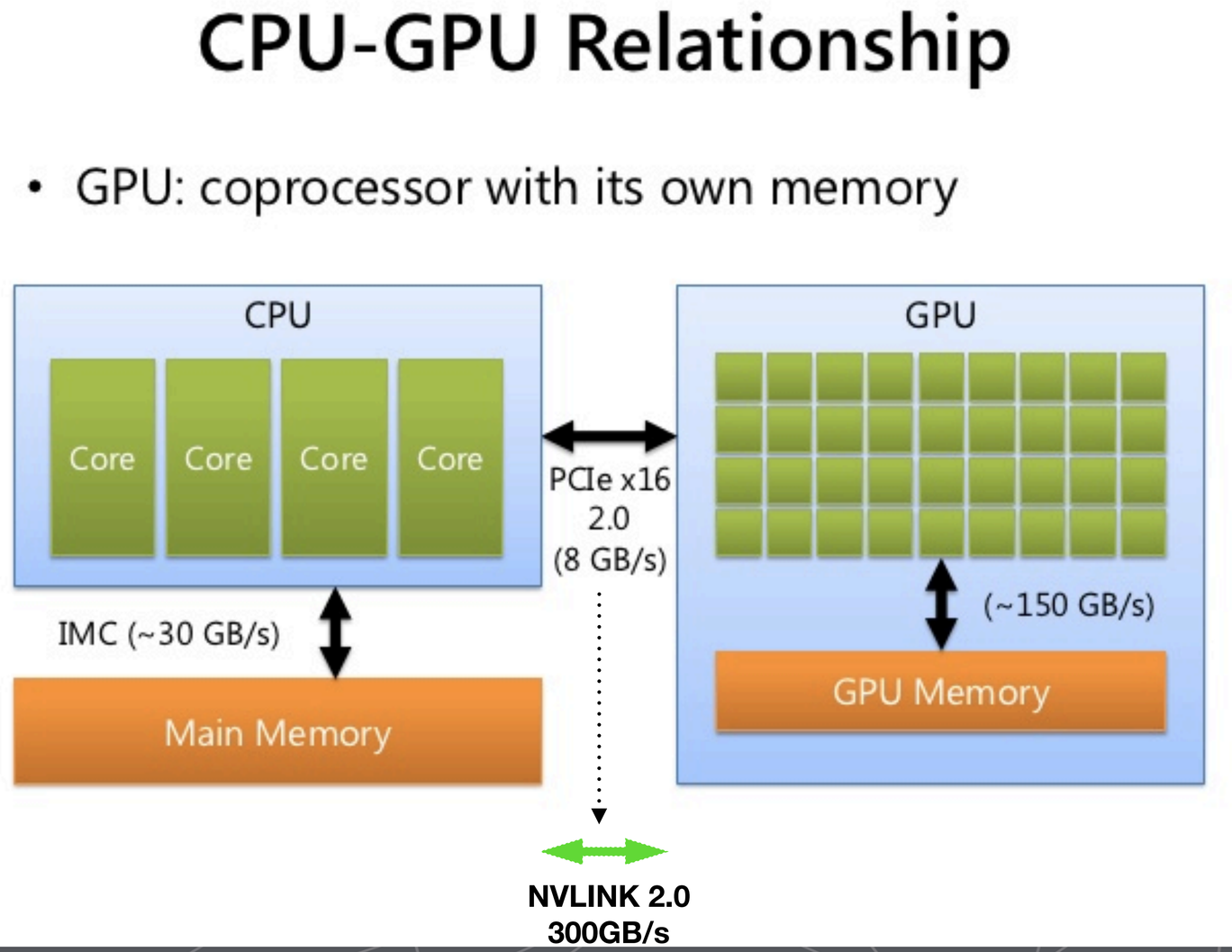
Why GPUs?

- Classic CPU single-threaded performance reached its limit
- GPUs naturally can handle massively parallel compute loads at high bandwidth
- GPUs is a **throughput-oriented**, many-core architecture, with thousands of cores bundled into dozens of SIMD multiprocessors
- Operational latencies for executions are higher on the GPU than on the CPU because of clock difference
- Operational latencies must be covered by thread-level and instruction-level parallelism
- GPUs have their own RAM, which can be accessed by the host



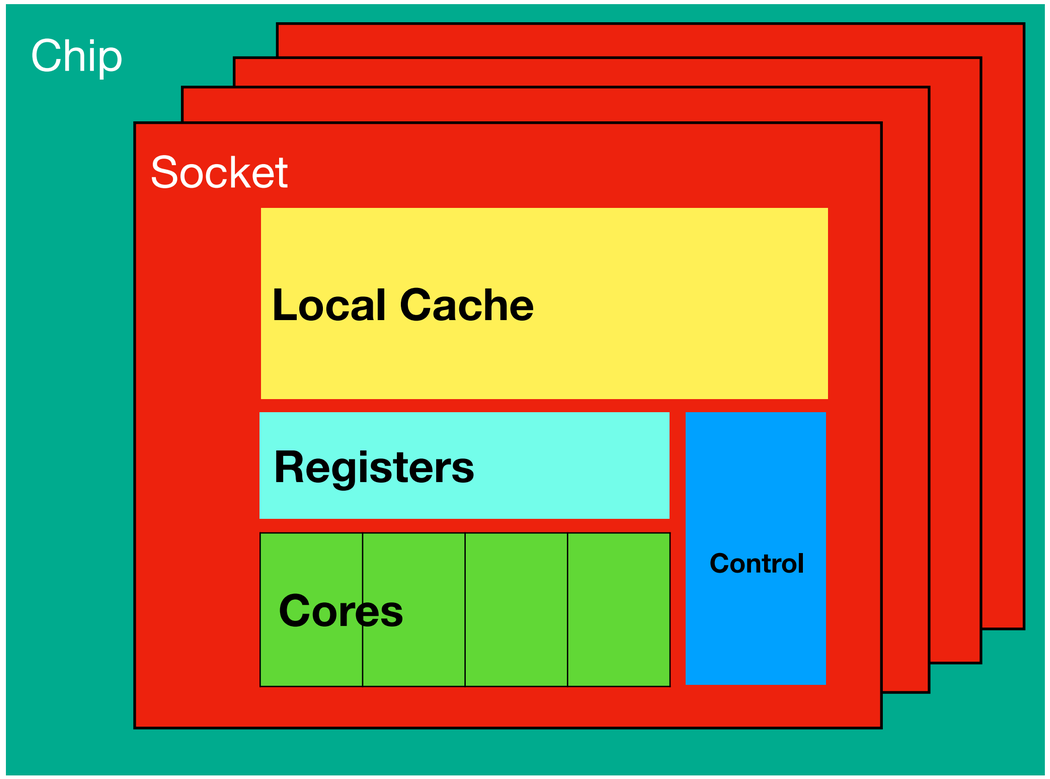
GPU Node Architecture

- Transfer between the chip and the GPU is conducted over PCI express bus or NvLink (Power systems)
- The third version of PCI-3
  - Delivers 32GB/s per lane for 'x' number of lanes
- The more lanes, the faster the transfer

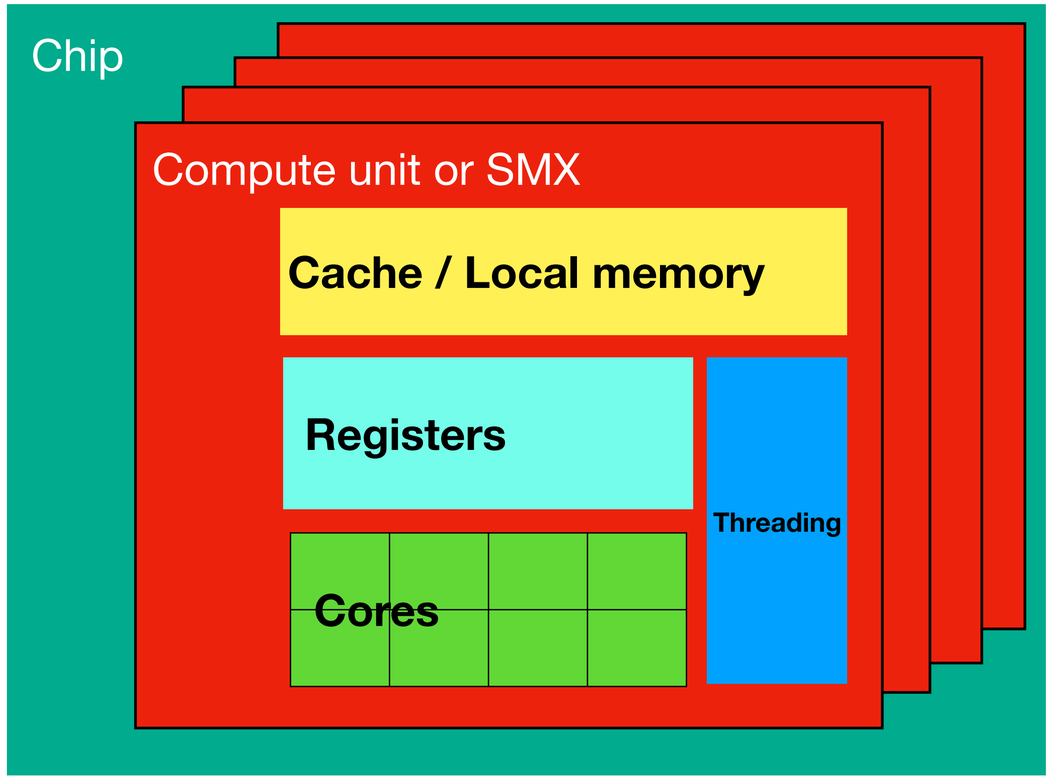


CPU and GPU are Designed Very Differently

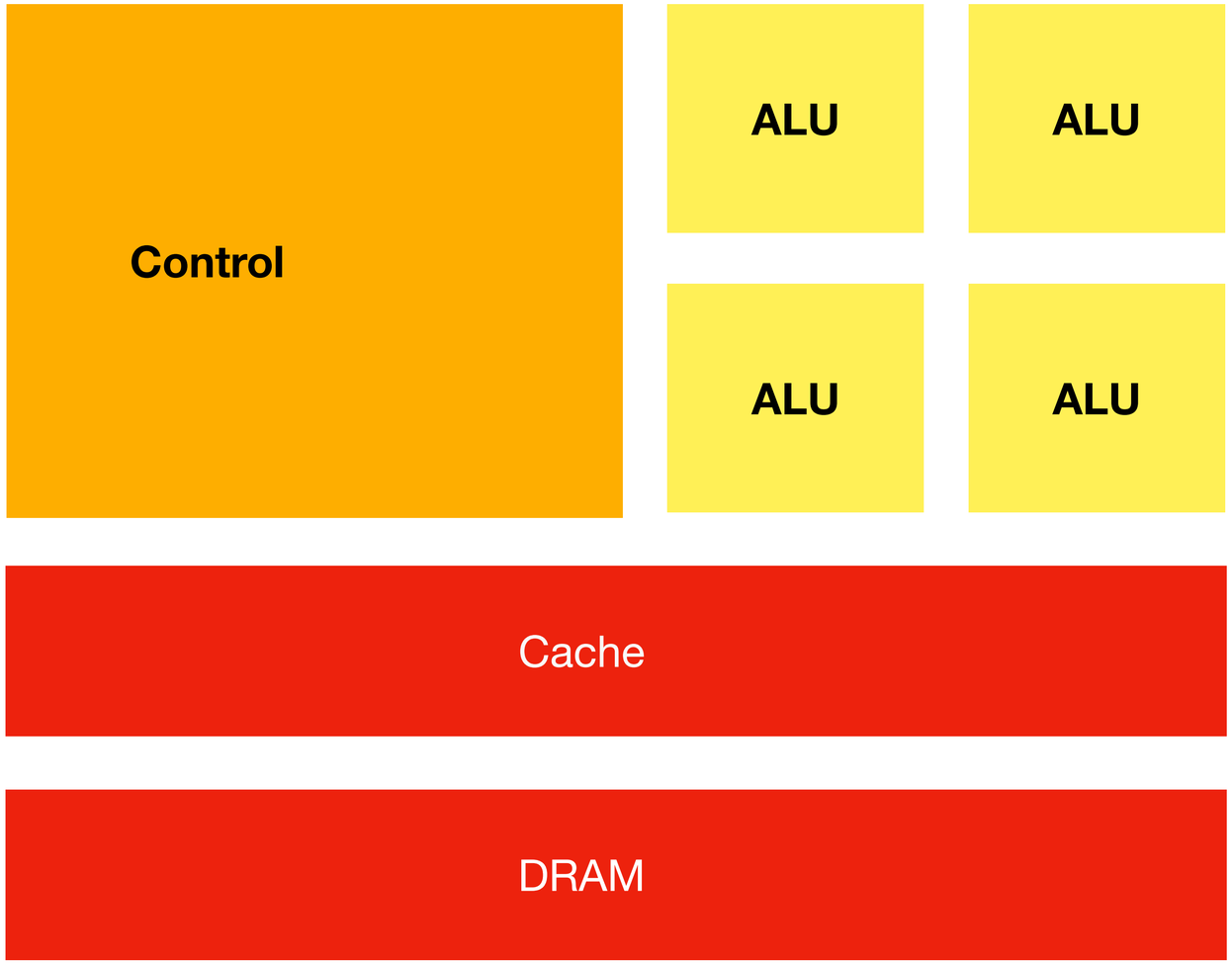
CPU
Latency Oriented Cores



GPU
Throughput Oriented Cores

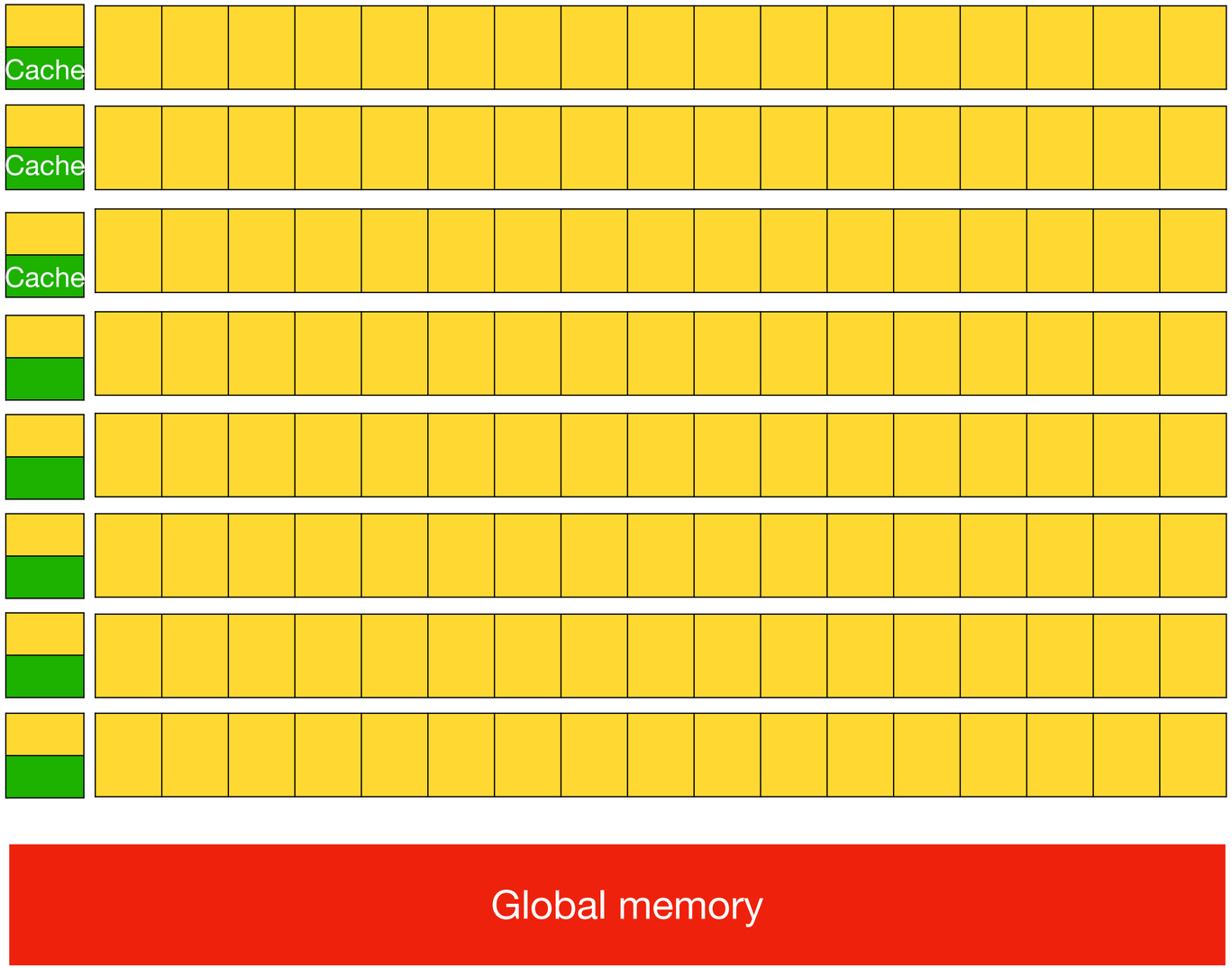


CPUs: Latency Oriented Design



- Powerful ALU
  - Great for serial operations
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency (Data prefetch)

GPUs: Throughput Oriented Design

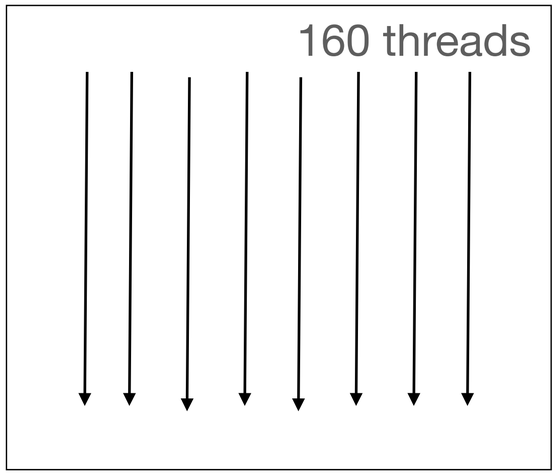


- Simple but a very large number of cores devoted to Floating Point operations.
- Small caches to boost memory throughput
- Simple control
  - No branch prediction
  - No data prefetching
- Require massive number of threads to hide operational latencies
  - Threading logic

Warps

- A Warp is a collection of 32 threads
- GPU always executes all 32 threads in a Single Instruction Multiple Threads (SIMT) approach
- Due to SIMT all threads in a warp normally execute the same instruction in the same single clock cycle

Logical view



Thread block

Hardware view

32 threads
32 threads
32 threads
32 threads
32 threads

Execution view

Warp scheduler			
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP
SP	SP	SP	SP

CPU vs GPU Threads & Register Usage per Thread

CPU threads	GPU threads
• Heavy weight threads • Plenty of resources per thread • Limited number of threads can be launched because number of cores are limited • Data cached through L1, L2 and L3 cache	• Light weight threads • Very limited resources per thread • A very high number of threads can be launched • Data cached through Shared memory and L2 cache • Requires special attention to number of registers allocated per thread

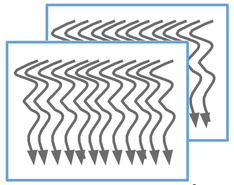
Software Abstraction

Software term

Threads

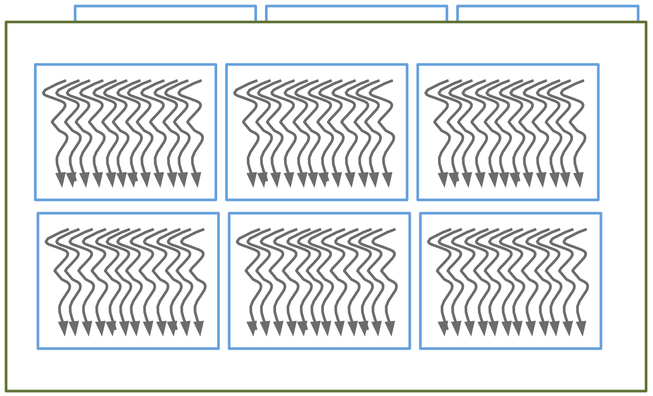
Vector (OpenACC)

Blocks



512–1024 threads / block

Grids



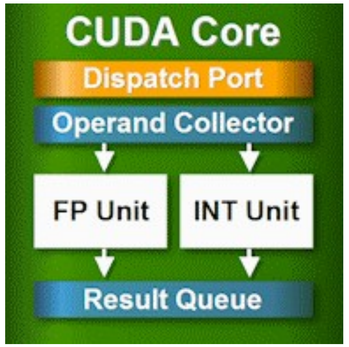
gangs (OpenACC)

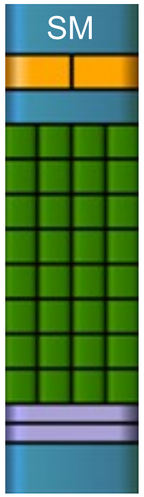
Kernel is executed by threads processed by CUDA Core

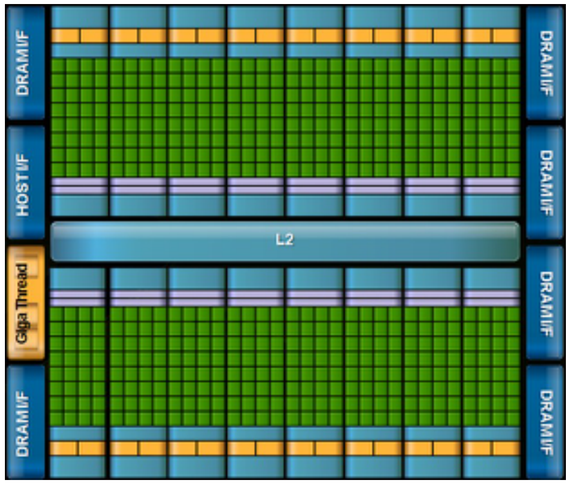
Maximum 8 blocks per SM
32 parallel threads are executed at the same time in a *WARP*

One grid per kernel with multiple concurrent kernels

Hardware term







Task 1: Using Container on Rescale

- Navigate to work/shared/<user>
- Clone the repo if you haven't already (from https://github.com/NCAR/GPU_workshop)
  - `git clone https://github.com/<GitHub Username>/GPU_workshop.git`

  -- or (only if you are unable to use “git clone”) --

  - `wget "https://github.com/NCAR/GPU_workshop/archive/refs/tags/V1.0.tar.gz"`
- Start a shell using the Singularity container
  - `singularity shell --nv <path_to_cuda_inst.sif>`
- Once inside the container, try the command `“nvidia-smi”.` It should print information about your instance’s GPU.
- Navigate to <user>/“add path here”

Task 1: File Structure

Example FORTRAN directory

```
.
├── exercise
│   ├── build.sh
│   ├── Makefile
│   ├── matrix_mult.f90
│   └── submit.sh
└── solution
    ├── build.sh
    ├── Makefile
    ├── matrix_mult.f90
    └── submit.sh
```

- Each lesson is has a C/C++ or FORTRAN example code. Within those, there are two sub-directories:
  - **Exercise** - the code that you will be working on.
  - **Solution** - a *minimal* working code

To run the exercise:

1. "cd" into desired directory
2. Complete your porting task
3. Run "`make`" to build executable
4. ./example.exe <arg1> <arg2> …

Example C/C++ directory

```
.
├── exercise
│   ├── build.sh
│   ├── common.cpp
│   ├── functions.cpp
│   ├── main.cpp
│   ├── Makefile
│   ├── matrix_mult.cc
│   ├── pch.h
│   └── submit.sh
└── solution
    ├── build.sh
    ├── common.cpp
    ├── functions.cpp
    ├── main.cpp
    ├── Makefile
    ├── matrix_mult.cc
    ├── pch.h
    └── submit.sh
```

Task 2: Matrix Multiplication with OpenACC

Task 2: Naive Matrix Multiplication with OpenACC

Parallelizing code with OpenACC involves adding directives to your sequential code. First of all, lets take a look at our CPU matrix multiplication code that runs sequentially.

```
// Calculate AxB=C
float temp = 0.0;
for (int i = 0; i < rowsA; i++)
  for (int j = 0; j < colsB; j++)
  {
      temp = 0.0;
      for (int k = 0; k < rowsB; k++)
      {
          temp += A[i*rowsB+k] * B[k*colsB+j];
      }

      C[i*colsB+j] = temp;
 }
```

We will be adding OpenACC directives to this base version of our matrix multiplication code.

Task 2: Naive Matrix Multiplication with OpenACC

We will be adding OpenACC directives to the matrix multiplication function:

- Add a data directive that copies in/out the required variables.

- Add a directive to tell the compiler what region needs to be parallelized

- Add a clause that collapses the two tightly-nested for loops. Note: Add the reduction clause.

- Add a directive that vectorizes the inner most loop. Note: We will also be adding the reduction clause to this directive.

Task 2: Naive Matrix Multiplication with OpenACC - C/C++

We will be adding OpenACC directives to offload the matrix multiplication to the GPU:

- Add a **data directive** that copies in/out the required Matrix variables.

- Add a directive to mark the **parallel region** for the compiler

- Add a directive that **collapses** the two tightly-nested for loops. Note: Add the reduction clause.

- Add a directive that vectorizes the inner most loop. Note: We will also be adding the reduction clause to this directive.

Close the data region

C/C++
In matrix_mult.cc

```
float temp = 0.0;
#pragma acc data copyout(C[0:rowsA*colsB]) \
   copyin(A[0:rowsA*rowsB],B[0:rowsB*colsB])
{
#pragma acc parallel loop collapse(2) \
   reduction(+:temp)
for (int i = 0; i < rowsA; i++)
  for (int j = 0; j < colsB; j++)
  {
       temp = 0.0;
 #pragma acc loop reduction(+:temp)
       for (int k = 0; k < rowsB; k++)
       {
            temp += A[i*rowsB+k] * B[k*colsB+j];
       }

       C[i*colsB+j] = temp;
  }
}
```

Task 2: Naive Matrix Multiplication with OpenACC - FORTRAN

We will be adding OpenACC directives to offload the matrix multiplication to the GPU:

- Add a **data directive** that copies in/out the required Matrix variables.
- Add a directive to mark the **parallel region** for the compiler
- Add a directive that **collapses** the two tightly-nested for loops. Note: Add the reduction clause.
- Add a directive that vectorizes the inner most loop. Note: We will also be adding the reduction clause to this directive
- Make sure to close parallel and data regions!

FORTRAN
In matrix_mult.f90, (line 90)

```
!$acc data copyin(a,b) copyout(c_gpu)
!$acc parallel loop collapse(2) reduction(+:tmp)
      do j=1,colsB
         do i=1,rowsA
            tmp = 0.0
!$acc loop reduction(+:tmp)
            do k=1,rowsB
               tmp = tmp + a(i,k) * b(k,j)
            enddo
            c_gpu(i,j) = tmp
         enddo
      enddo
!$acc end parallel
!$acc end data
```

Breakout Session 2

- Task 4: Profiling with NVprof
- Task 5: Error Checking with PCAST
- Task 6: Compiler OpenACC Profiling (NV_ACC_TIME)

Task 4: Introduction to nvprof (NVIDIA Profiler)

nvprof is a command line profiler that provides information about your program execution along with timing information.

Below is an example of the nvprof output from our Matrix Multiplication code. This output is generated by adding the nvprof command while running our executable. Example: nvprof --unified-memory-profiling off ./example.exe

```
pgc++ -c -g -O3 -std=c++11 -Wall   main.cpp
pgc++ -c -g -O3 -std=c++11 -Wall   functions.cpp
pgc++ -c -g -O3 -std=c++11 -Wall   common.cpp
nvcc -c -g -std=c++11 -O3 -I/glade/u/apps/dav/opt/cuda/11.0.3//include matrixMul.cu
pgc++ -o output.exe main.o functions.o common.o matrixMul.o  -L/glade/u/apps/dav/opt/cuda/11.0.3//lib64 -lcudart
==149693== NVPROF is profiling process 149693, command: ./output.exe

Entire CPU process took 3.360000 seconds.... Starting Cuda Process.
Done. Matrix multiplication on GPU took 0.360000 seconds.
==149693== Profiling application: ./output.exe
==149693== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   43.82%  1.4974ms         2  748.68us  745.47us  751.90us  [CUDA memcpy HtoD]
                   38.48%  1.3149ms         1  1.3149ms  1.3149ms  1.3149ms  mmul(float*, float*, float*, int, int, int)
                   17.71%  605.18us         1  605.18us  605.18us  605.18us  [CUDA memcpy DtoH]
      API calls:   97.41%  249.69ms         3  83.231ms  271.31us  249.02ms  cudaMalloc
                    1.15%  2.9541ms         3  984.69us  975.76us  1.0010ms  cudaMemcpy
                    0.53%  1.3542ms         1  1.3542ms  1.3542ms  1.3542ms  cudaDeviceSynchronize
                    0.29%  752.64us       101  7.4510us     130ns  333.63us  cuDeviceGetAttribute
                    0.29%  745.28us         3  248.43us  224.24us  289.07us  cudaFree
                    0.27%  696.19us         1  696.19us  696.19us  696.19us  cuDeviceTotalMem
                    0.04%  90.345us         1  90.345us  90.345us  90.345us  cuDeviceGetName
                    0.02%  44.011us         1  44.011us  44.011us  44.011us  cudaLaunchKernel
                    0.00%  7.4660us         1  7.4660us  7.4660us  7.4660us  cuDeviceGetPCIBusId
                    0.00%  3.3140us         4     828ns     190ns  2.3040us  cudaGetLastError
                    0.00%  2.0040us         3     668ns     223ns  1.1470us  cuDeviceGetCount
                    0.00%  1.5240us         2     762ns     511ns  1.0130us  cuDeviceGet
                    0.00%     234ns         1     234ns     234ns     234ns  cuDeviceGetUuid
```

Task 4: Introduction to nvprof (NVIDIA Profiler)

A more detailed output can be generated by using: nvprof --unified-memory-profiling off --print-gpu-trace ./example.exe

This command is most useful when running your code on two GPUs. It will display what GPU each part/Kernel of your code is running on. Below is an example of the output from running the command above.

```
rm *.o output.exe
pgc++ -c -g -O3 -std=c++11 -Wall   main.cpp
pgc++ -c -g -O3 -std=c++11 -Wall   functions.cpp
pgc++ -c -g -O3 -std=c++11 -Wall   common.cpp
nvcc -c -g -std=c++11 -O3 -I/glade/u/apps/dav/opt/cuda/11.0.3//include matrixMul.cu
pgc++ -o output.exe main.o functions.o common.o matrixMul.o  -L/glade/u/apps/dav/opt/cuda/11.0.3//lib64 -lcudart
==42032== NVPROF is profiling process 42032, command: ./output.exe

Entire CPU process took 3.260000 seconds.... Starting Cuda Process.
Done. Matrix multiplication on GPU took 0.270000 seconds.
==42032== Profiling application: ./output.exe
==42032== Profiling result:
   Start  Duration           Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput  SrcMemType  DstMemType          Device   Context    Stream
  Name
377.59ms  798.97us                    -               -         -         -         -  4.0000MB  4.8891GB/s    Pageable      Device  Tesla V100-SXM2         1         7
  [CUDA memcpy HtoD]
378.62ms  784.35us                    -               -         -         -         -  4.0000MB  4.9802GB/s    Pageable      Device  Tesla V100-SXM2         1         7
  [CUDA memcpy HtoD]
379.42ms  1.4475ms            (32 32 1)       (32 32 1)        32        0B        0B         -           -           -           -  Tesla V100-SXM2         1         7
  mmul(float*, float*, float*, int, int, int) [118]
380.88ms  625.09us                    -               -         -         -         -  4.0000MB  6.2491GB/s      Device    Pageable  Tesla V100-SXM2         1         7
  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler sh
ows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
SrcMemType: The type of source memory accessed by memory operation/copy
DstMemType: The type of destination memory accessed by memory operation/copy
```

Task 4: nvprof profiling Matrix Multiplication with tile

During the next breakout room we want you to get some experience with nvprof.

Within your matrix multiplication with tile exercise code folder, run a command like:

```
nvprof --unified-memory-profiling off --print-gpu-trace ./matmul.exe
```

Where is the majority of the time spent in this program? Discuss with your mentors.

Task 5: PGI Autocompare

- PCAST is used to check the validity of the GPU code
- When compiled with the **-ta=tesla:autocompare** flag, code in OpenACC compute regions will run redundantly on the CPU and GPU
- Simplest way to invoke PCAST is to use "**`-ta=tesla:autocompare,<other options>`**" compiler flag to enable the autocompare feature
- Set the environment variable "**`export PCAST_COMPARE=summary,abs=6,verboseautocompare`**" in the shell to get a summary of comparisons with absolute error tolerance up to 10^(-6) and verbose output
- Any data in a **copy**, **copyout**, or **update host** directive will be compared when it is copied off the device.

Source: https://www.pgroup.com/resources/pcast.htm

Task 6: Compiler OpenACC Profiling

- When not using nvprof, here are two environment variables you can set that will provide information about the OpenACC execution:
  - **export NV_ACC_TIME=1**
    - prints kernel timing data after program execution finishes
  - **export NV_ACC_NOTIFY=** [1,2,4,8,16] (bitmask)
    - 1 - launch
    - 2 - data upload/download
    - 4 - wait (explicit or implicit) for device
    - 8 - data/compute region
    - 16 - data create, allocate, delete, and free

Code Explanation

MPI-OpenACC: Laplace-Jacobi Base Solver Code

```
// Allocate the second version of the M matrix used for the computation
M_new = (float*)malloc(ny*nx*sizeof(float));

do {
        maxdiff = 0.0f;
        itr++;
        // Update M_new with M
        for(int i=1; i<ny-1; i++){
                for(int j=1; j<nx-1; j++){
                        M_new[i*nx+j] = 0.25f *(M[(i-1)*nx+j]+M[i*nx+j+1]+ \
                                                M[(i+1)*nx+j]+M[i*nx+j-1]);
                }
        }

        // Check for convergence while copying values into M
        for(int i=1; i<ny-1; i++){
                for(int j=1; j<nx-1; j++){
                        maxdiff = MAX(fabs(M_new[i*nx+j] - M[i*nx+j]), maxdiff);
                        M[i*nx+j] = M_new[i*nx+j];
                }
        }
} while(maxdiff > JACOBI_TOLERANCE);

// Free malloc'd memory
free(M_new);
```

Profiling Kernels with nvprof CLI

```
mpirun  -n 16 nvprof --print-gpu-trace --profile-api-trace none --kernels "::LaplaceJacobi_MPIACC:2"  --metrics
"achieved_occupancy,gld_transactions,gst_transactions,flop_count_sp"  ./mpi_acc_stencil.exe 64 4
```

- Profiles 2nd invocation of all kernels launched with “LaplaceJacobi_MPIACC” in name
- Gathers specified metrics (Use “nvprof -- query-metrics” for full list of options)

```
==74878== Profiling result:
         Device   Context    Stream                Kernel  achieved_occupancy  gld_transactions  gst_transactions  flop_count_sp

Tesla V100-SXM2         2        29  LaplaceJacobi_MPIACC            0.062388               587               160           4096
```

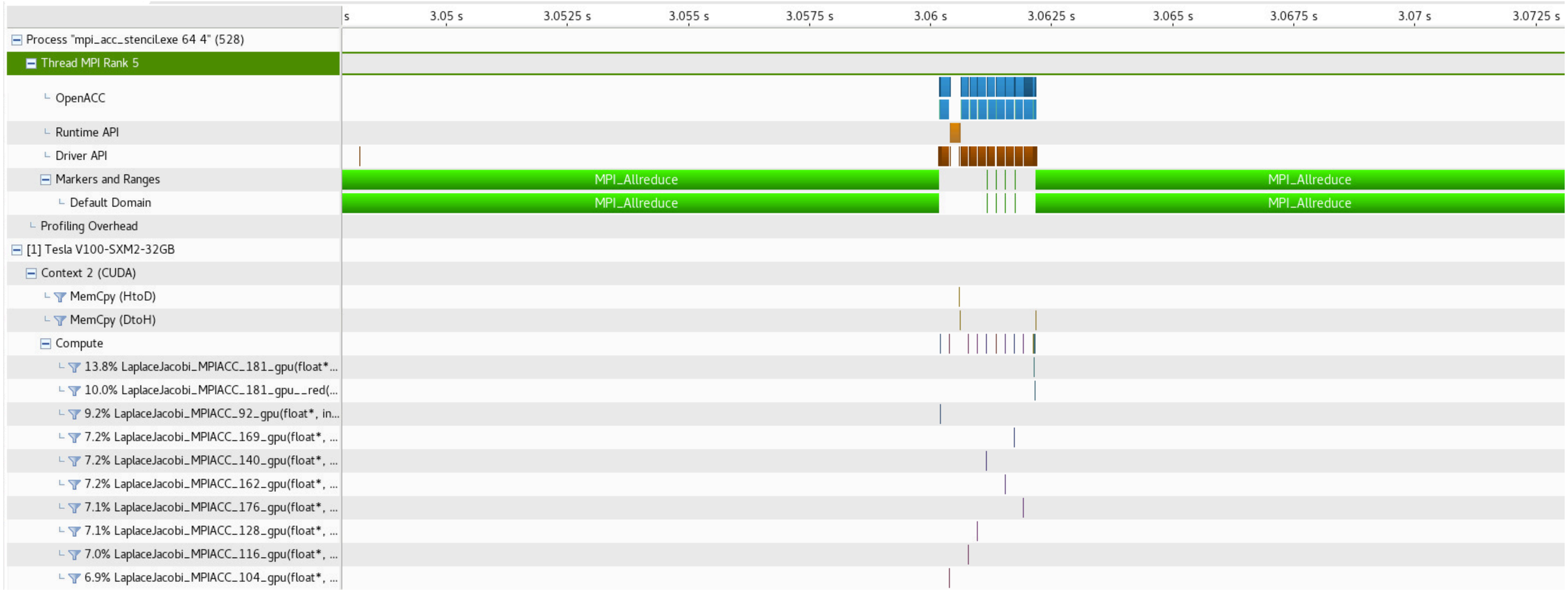
- Monitor the achieved occupancy as the data size increases
  - Ex: for a Matrix size 16,384 divided over 16 Ranks

```
==76531== Profiling result:
         Device   Context    Stream                Kernel  achieved_occupancy  gld_transactions  gst_transactions  flop_count_sp

Tesla V100-SXM2         2        29  LaplaceJacobi_MPIACC            0.936929           9630915           2621440       67108864
```

Profiling MPI with nvprof GUI

```
mpirun -n 16 nvprof --annotate-mpi openmpi -o Jacobi.%q{OMPI_COMM_WORLD_RANK}.nvprof ./mpi_acc_stencil.exe 64 4
```



Exercise: Distributed LaplaceJacobi Solver

Look for left-aligned comments starting with "//" for tips on what to do in the code

Steps:

- In main.cpp

- Add openacc.h header
- After acc_init call the function provided to map GPUs to MPI ranks
- Fill in the call to LaplaceJacobi_MPIACC with the correct arguments

- In stencil.cc

- Add openacc.h header file
- Add a set of unstructured data pragmas to copyin and create all the variables needed
- Add a set of unstructured data pragmas to copyout and delete variables
- Add pragmas to parallelize the update and convergence loops
- Add pragmas to ensure the MPI communication can see the send/receive buffers.
- Parallelize any other loops inside the LaplaceJacobi_MPIACC function

To build use **./build.sh**

To submit use **sbatch submit.sh**

Exercise Check: Distributed LaplaceJacobi Solver

What questions did you have?

Exercise Check: Distributed LaplaceJacobi Solver

Some things to think about:

- What happens when we don't map GPUs and MPI ranks?

- Do we need to do a halo exchange every iteration? What might change if we don't?

- The time for CPU code with the input `mpirun -n 16 ./mpi_acc_stencil.exe 64 4` is about 0.103006 seconds. If we run the same inputs on OpenACC code, is this a fair comparison?

- If we were to run this code on only 1 V100 GPU, how big of a matrix could we process? (recall global memory size).

Exercise Check: Distributed LaplaceJacobi Solver

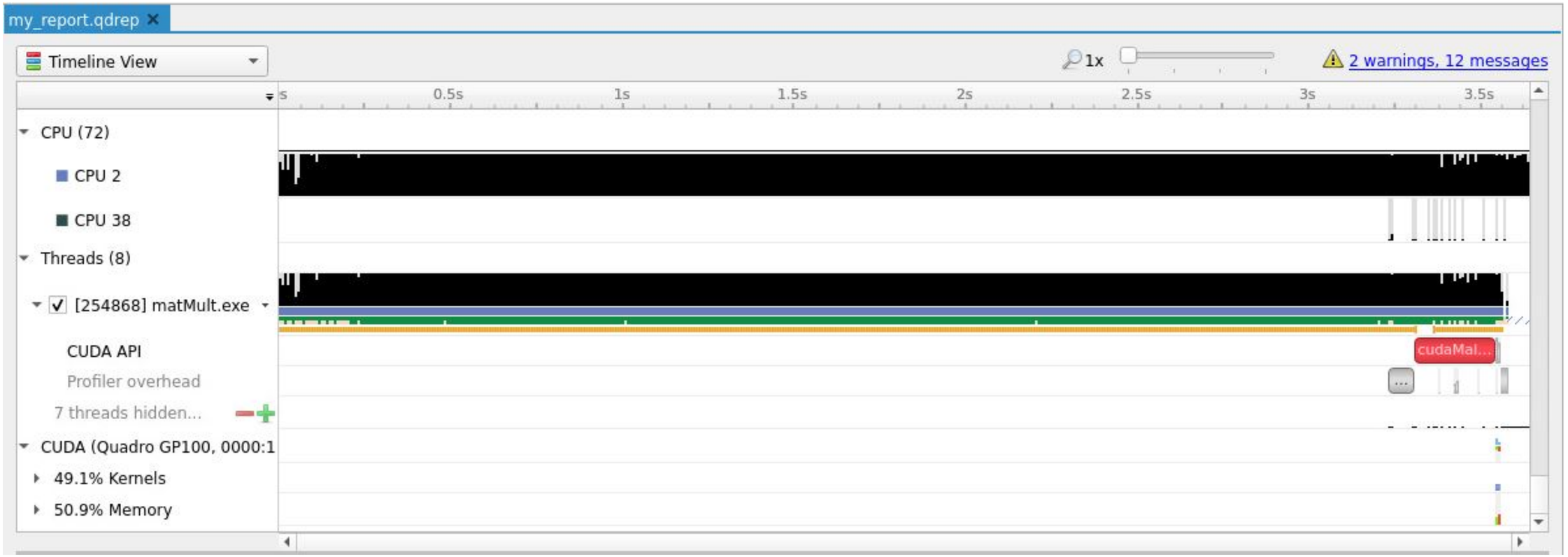
Some things to think about:

- What happens when we don't map GPUs and MPI ranks?
  - All code will execute on one GPU
- Do we need to do a halo exchange every iteration? What might change if we don't?
  - Consider a very large matrix, the most interior points will take a long time to converge regardless of what occurs at the edges. We might be able to save execution time without message passing
- The time for CPU code with the input `mpirun -n 16 ./mpi_acc_stencil.exe 64 4` is about 0.103006 seconds. If we run the same inputs on OpenACC code, is this a fair comparison?
  - No. Not only do we need optimized CPU code, the timers used in main.cpp aren't how we would get GPU speedup. We would look at output from NV_ACC_TIME=1 or profiling and average by the number of calls.
- If we were to run this code on only 1 V100 GPU, how big of a matrix could we process? (recall global memory size).
  - Theoretically about 46k by 46k matrices. This is half of what we expect because of M_new.

Nsight Profiler

Last session, we looked at the nvprof which is a command line profiler that prints out information about your program execution.

Today, we will be looking at Nsight which is also a profiler but is more user friendly as it displays information about your program execution on an interactive GUI as seen below.



Using Nsight on Casper

Step 6: Once you are on the terminal, load the cuda module using "module load cuda". Without loading this, the nsys commands will not be loaded into our environment.

Step 7: Run the build script and submit the job.

Now we profile!

Step 8: Run the following command to profile your code:

"nsys profile -o profileData ./exec_file.exe"

Step 9: Finally, run the command "nsight-sys profileData.qdrep" to view your profile on the GUI

Code practice 1

- **Task 1: Connecting & Accessing the Code**
- **Task 2: Coding Practice - Matrix Multiplication**

General guidelines for screen sharing

https://docs.google.com/document/d/1U1jJ1W58SVDrXy7hJK9Yw7pAXQelWomKbU5g_u_nXHk/edit?usp=sharing