ECE408 / CS483 / CSE408
Summer 2025

Applied Parallel Programming

Lecture 22:
Alternatives to CUDA

What Will You Learn Today?

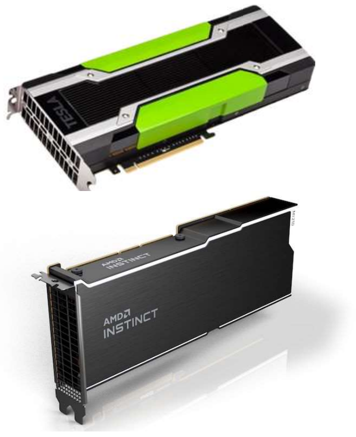
programming alternatives to CUDA

- more general models
- pragma-style models
- “built-in” library/interpreter approaches

more general heterogeneous computing models

Accelerated Computing is no longer a question

CRAY
OAK RIDGE National Laboratory
U.S. DEPARTMENT OF ENERGY
Hewlett Packard Enterprise
AMD
FRONTIER



GPU vendors include:
Nvidia
AMD
Intel
Samsung
Apple
Qualcomm
ARM
etc….

CUDA is just one model for Compute Acceleration

OpenGL (1992)
DirectX (1995)

GPGPU (2002)

CUDA (2007)

OpenCL (2008)

OpenACC (2012)
C++AMP (2013)
RenderScript (2013)

OpenMP (2016)
Metal (2014)
SYCL (2014)
Vulkan (2016)
ROCm HIP (2016)

Existing frameworks such as MPI, TBB, OpenCV adapted to provide support.
New frameworks such as Caffe, TensorFlow, R, PyCUDA natively support acceleration.

OpenCL, HIP, OpenACC, MPI

- OpenCL: An Open Standard Acceleration API
- Heterogeneous-Computing Interface for Portability (HIP)
- OpenACC: A “Low-Code” Acceleration API
- MPI: A Large Scale, Multi-Node Parallel API

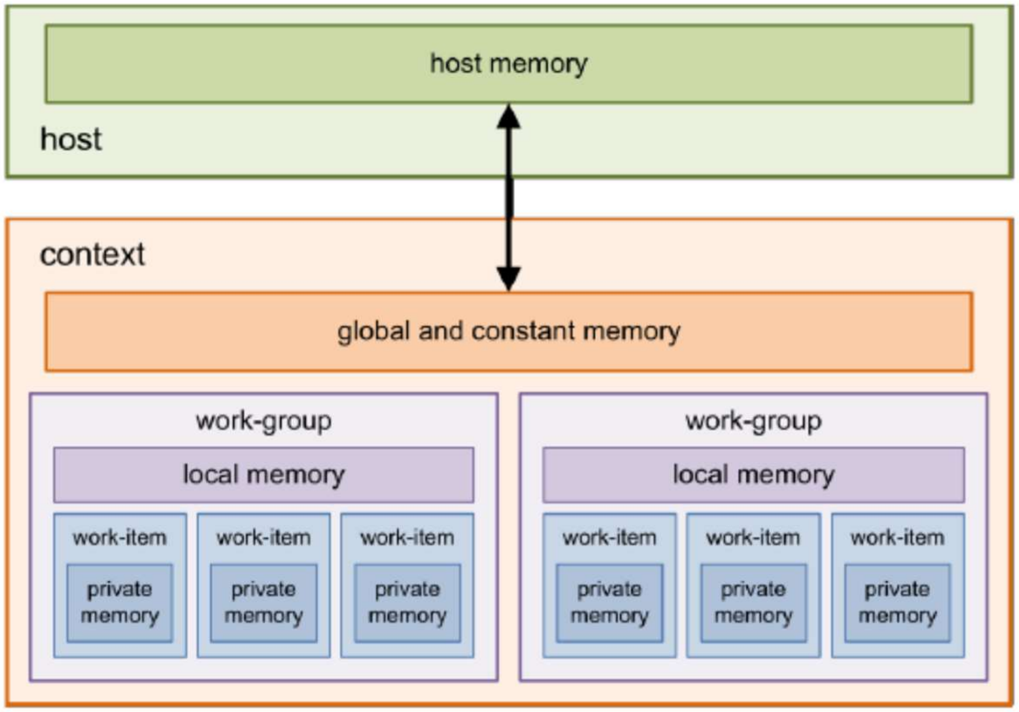
Common Traits for Acceleration APIs

- HARDWARE
  - Hierarchy of lightweight cores
  - Local scratchpad memories
  - Lack of HW coherence
  - Slow global atomics
  - Threading
- SOFTWARE
  - Kernel oriented acceleration
  - Device memory vs. Host memory
  - Software managed memory
  - Grids, Blocks, Threads
  - Bulk Synchronous Parallelism

OpenCL

- Framework for CPUs, GPUs, DSPs, FPGAs, etc. (not just NVIDIA GPUs)
- Initially developed by Apple with support from AMD, IBM, Qualcomm, Intel, and Nvidia. OpenCL 1.0 launched in 2008.
- OpenCL 2.2 launched in May 2017
- Apple announces dropping of OpenCL in 2018

OpenCL Memory Model



OpenCL MatMult

- Notice similarity to CUDA
- WorkGroup similar to Block
- WorkItem similar to Thread
- __local similar to __shared

```
// Tiled and coalesced version
__kernel void myGEMM2(int M, int N, int K, __global float*A, __global float*B, __global float*C) {

   // Thread identifiers
   const int row = get_local_id(0); // Local row ID (max: TS)
   const int col = get_local_id(1); // Local col ID (max: TS)
   const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
   const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[TS][TS];
    __local float Bsub[TS][TS];

    // Initialise the accumulation register
    float acc = 0.0f;
    // Loop over all tiles
    const int numTiles = K/TS;
    for (int t=0; t<numTiles; t++) {

       // Load one tile of A and B into local memory
       const int tiledRow = TS*t + row;
       const int tiledCol = TS*t + col;
       Asub[col][row] = A[tiledCol*M + globalRow];
       Bsub[col][row] = B[globalCol*K + tiledRow];

       // Synchronise to make sure the tile is loaded
       barrier(CLK_LOCAL_MEM_FENCE);

       // Perform the computation for a single tile
       for (int k=0; k<TS; k++)
          acc += Asub[k][row] * Bsub[col][k];

       // Synchronise before loading the next tile
       barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store the final result in C
    C[globalCol*M + globalRow] = acc;
}
```

HIP

- Heterogeneous-Computing Interface for Portability (HIP)
  - C++ dialect designed to ease conversion of CUDA applications to portable C++ code.
  - Provides a C-style API and a C++ kernel language.
  - The C++ interface can use templates and classes across the host/kernel boundary.
- HIP code can run on AMD hardware (through the HCC compiler) or NVIDIA hardware (through the NVCC compiler).
- The HIPify tool automates much of the conversion work by performing a source-to-source transformation from CUDA to HIP.

vectorAdd with HIP

```
__global__ void vecAdd(double *a, double *b, double *c, int n) {
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if (id < n) c[id] = a[id] + b[id];
}
...
    hipMalloc(&d_a, nbytes);
    hipMalloc(&d_b, nbytes);
    hipMalloc(&d_c, nbytes);

    hipMemcpy(d_a, h_a, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_b, h_b, bytes, hipMemcpyHostToDevice);

    blockSize = 1024;
    gridSize = (int)ceil((float)n/blockSize);

    hipLaunchKernelGGL(vecAdd, dim3(gridSize), dim3(blockSize), 0, 0, d_a, d_b, d_c, n);
    hipDeviceSynchronize( );

    hipMemcpy(h_c, d_c, bytes, hipMemcpyDeviceToHost);
...
```

OpenACC

The OpenACC Application Programming Interface (API) provides a set of

- compiler directives (pragmas),
- library routines, and
- environment variables

that enable

- FORTRAN, C and C++ programs
- to execute on accelerator devices
- including GPUs and CPUs.

Pragmas Provide Extra Information

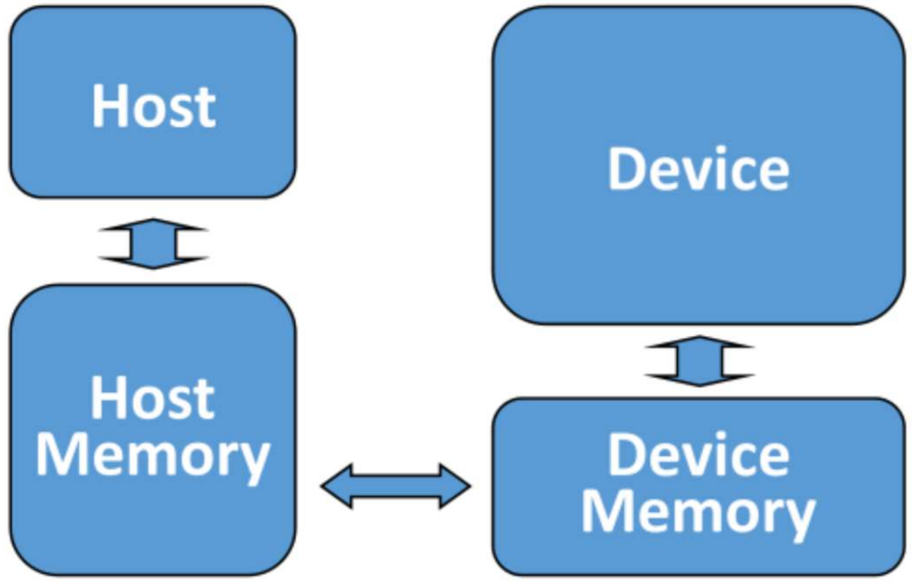
In C and C++,

- the `#pragma` directive
- provides the compiler with
- information not specified in the language.

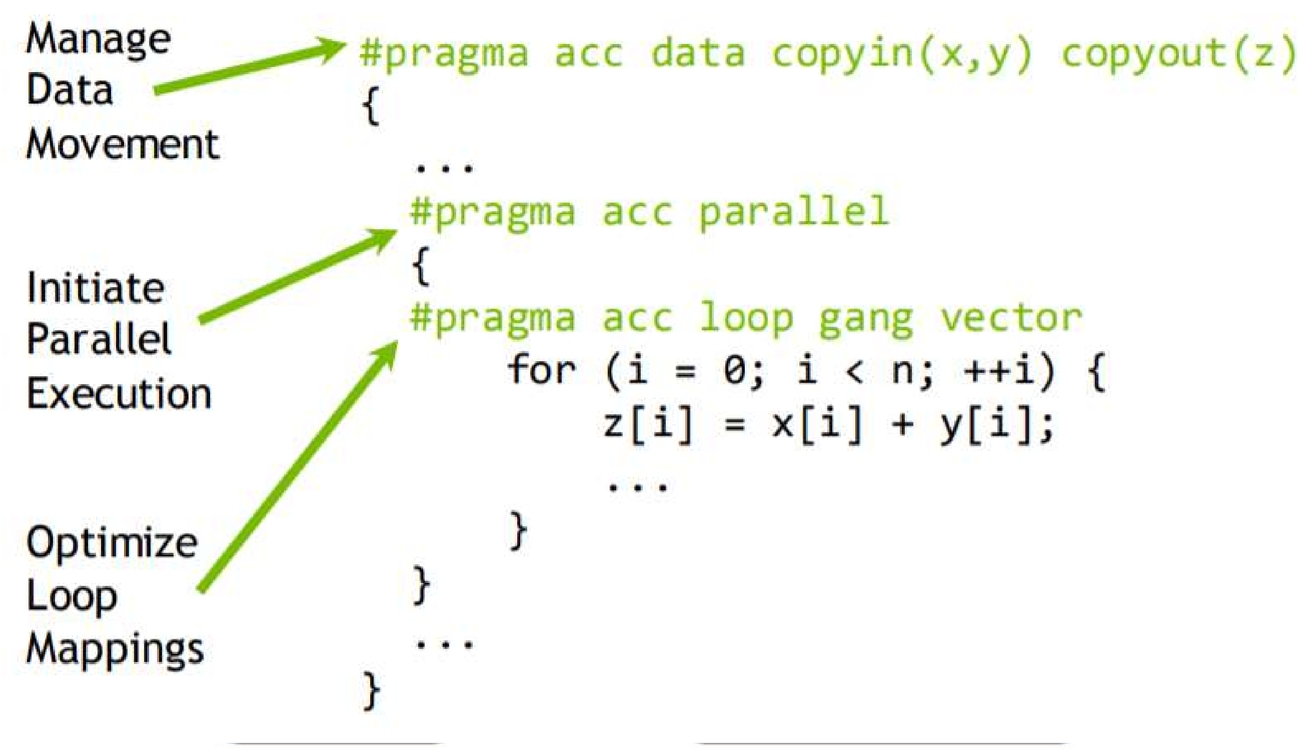
For OpenACC, they look like this:

#pragma acc [the information goes here]

The OpenACC Abstract Machine Model



The OpenACC Directives



```
#pragma acc data copyin(x,y) copyout(z)
{
   ...
   #pragma acc parallel
   {
   #pragma acc loop gang vector
       for (i = 0; i < n; ++i) {
           z[i] = x[i] + y[i];
           ...
       }
   }
   ...
}
```

Simple Matrix-Matrix Multiplication in OpenACC

```
void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw) {
 #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
 for (int i=0; i<Mh; i++) {
     #pragma acc loop
     for (int j=0; j<Nw; j++) {
         float sum = 0;
         for (int k=0; k<Mw; k++) {
             float a = M[i*Mw+k];
             float b = N[k*Nw+j];
             sum += a*b;
         }
         P[i*Nw+j] = sum;
     }
  }
}
```

Add Pragmas to Sequential Code

The **code** is

- **identical to** the **sequential** version
- **except for** the two **pragmas**
- at lines 2 and 4.

OpenACC uses the compiler directive mechanism to extend the base language.

Simple Matrix-Matrix Multiplication in OpenACC

```
1  void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw) {
2   #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
3   for (int i=0; i<Mh; i++) {
4       #pragma acc loop
5       for (int j=0; j<Nw; j++) {
6           float sum = 0;
7           for (int k=0; k<Mw; k++) {
8              float a = M[i*Mw+k];
9              float b = N[k*Nw+j];
10              sum += a*b;
11          }
12          P[i*Nw+j] = sum;
13      }
14    }
15 }
```

tells compiler
- to execute 'i' loop
- (lines 3 through 14)
- in parallel on accelerator.

copyin/copyout specify
- how matrix data
- should be transferred between memories.

Simple Matrix-Matrix Multiplication in OpenACC

```
1  void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw) {
2   #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
3   for (int i=0; i<Mh; i++) {
4       #pragma acc loop
5       for (int j=0; j<Nw; j++) {
6           float sum = 0;
7           for (int k=0; k<Mw; k++) {
8              float a = M[i*Mw+k];
9              float b = N[k*Nw+j];
10              sum += a*b;
11          }
12          P[i*Nw+j] = sum;
13      }
14    }
15 }
```

tells compiler

- to map ‘j’ loop
- (lines 5 through 13)
- to second level
- of parallelism on accelerator.

Motivating Goal: One Version of Code

OpenACC programmers

- can often start with a sequential version,
- then annotate their program with directives,
- leaving most kernel details and data transfers
- to the OpenACC compiler.

OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

Reality is More Complicated

Reality check:

- can be **difficult to write code**
- that works **correctly and well**
- **with and without pragmas**.

Some OpenACC programs

- behave differently or even incorrectly
- if pragmas are ignored.

Pitfall: Strong Dependence on Compiler

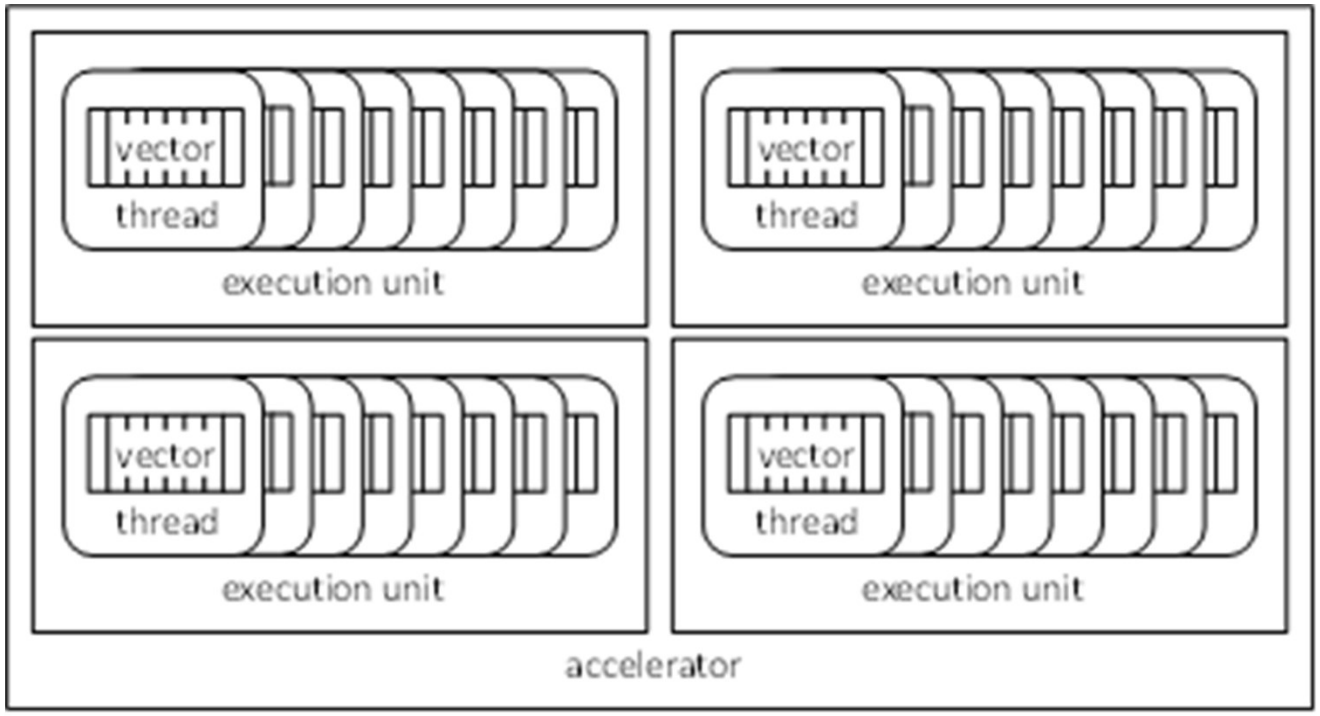
Some OpenACC pragmas

- are hints to the OpenACC compiler,
- which may or may not be able to act accordingly

Performance depends heavily

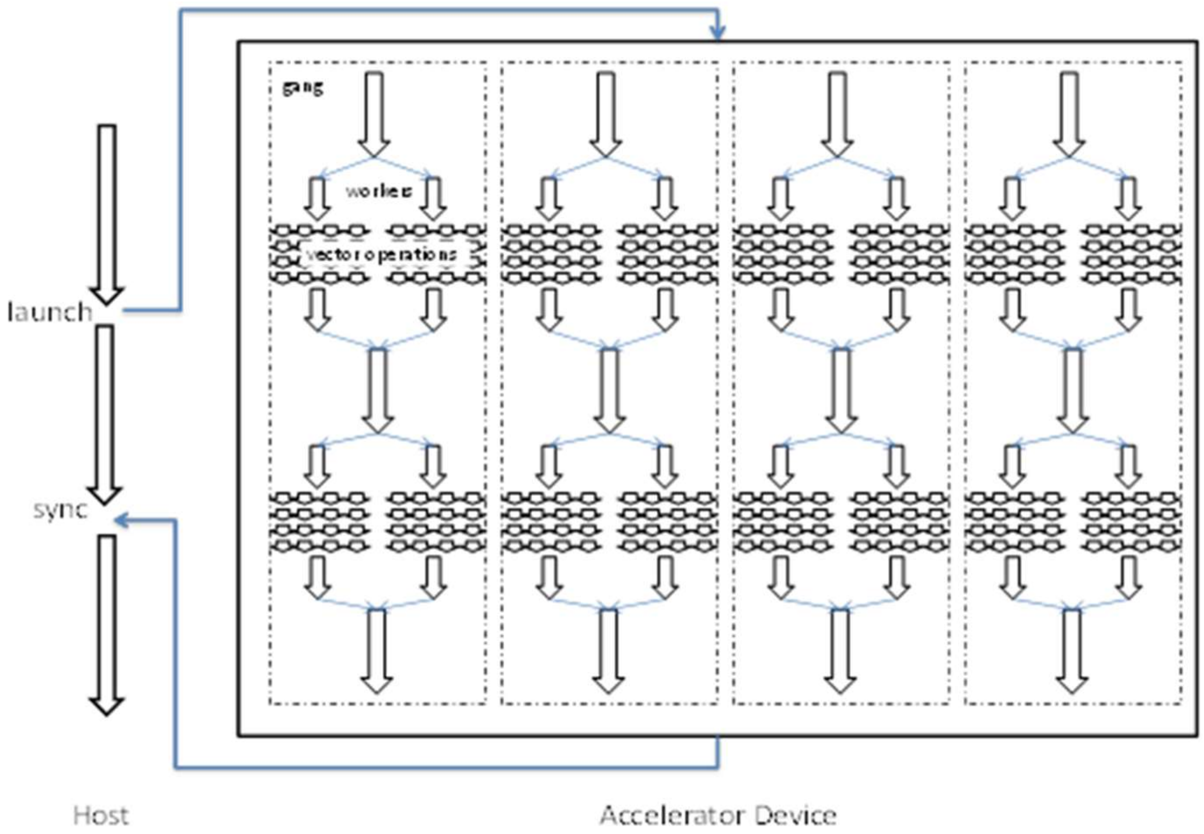
- **on** the **quality of** the **compiler**
- (more so than with CUDA or OpenCL).

OpenACC Device Model



Currently OpenACC does not allow user-specified synchronization across threads.

OpenACC Execution Model (Terminology: Gangs and Workers)



Parallel vs. Loop Constructs

```
#pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
for (int i=0; i<Mh; i++) {
…
}
```

is equivalent to:

```
#pragma acc parallel copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
{
    #pragma acc loop
    for (int i=0; i<Mh; i++) {
        …
    }
}
```

(a parallel region that consists of just a loop)

Parallel Construct

- A parallel construct is executed on an accelerator
- One can specify the number of gangs and number of workers in each gang
- Programmer’s directive

```
#pragma acc parallel copyout(a) num_gangs(1024) num_workers(32)
{
        a = 23;
}
```

1024*32 workers will be created. a=23 will be executed redundantly by all 1024 gang leads

What does each "Gang Loop" do?

```
#pragma acc parallel num_gangs(1024)
{
    for (int i=0; i<2048; i++) {
        …
    }
}
```

The for-loop will be redundantly executed by 1024 gangs

```
#pragma acc parallel num_gangs(1024)
{
#pragma acc loop gang
    for (int i=0; i<2048; i++) {
        …
    }
}
```

The 2048 iterations of the for-loop will be divided among 1024 gangs for execution

Worker Loop

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        #pragma acc loop worker
        for (int j=0; j<512; j++) {
            foo(i,j);
        }
    }
}
```

1024*32=32K workers will be created, each executing 1M/32K = 32 instance of foo()

A More Complex Example

```
#pragma acc parallel num_gangs(32)
{
    Statement 1; Statement 2;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
       Statement 3; Statement 4;
    }
    Statement 5;  Statement 6;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
       Statement 7;  Statement 8;
    }
    Statement 9;
    if (condition)
      Statement 10;
}
```

- Statements 1 and 2 are redundantly executed by 32 gangs
- The n for-loop iterations are distributed to 32 gangs

Kernel Regions

```
#pragma acc kernels
{
    #pragma acc loop num_gangs(1024)
    for (int i=0; i<2048; i++) {
        a[i] = b[i];
    }
    #pragma acc loop num_gangs(512)
    for (int j=0; j<2048; j++) {
        c[j] = a[j]*2;
    }
    for (int k=0; k<2048; k++) {
        d[k] = c[k];
    }
}
```

- Kernel constructs are descriptive of programmer intentions (suggestions)

Reduction

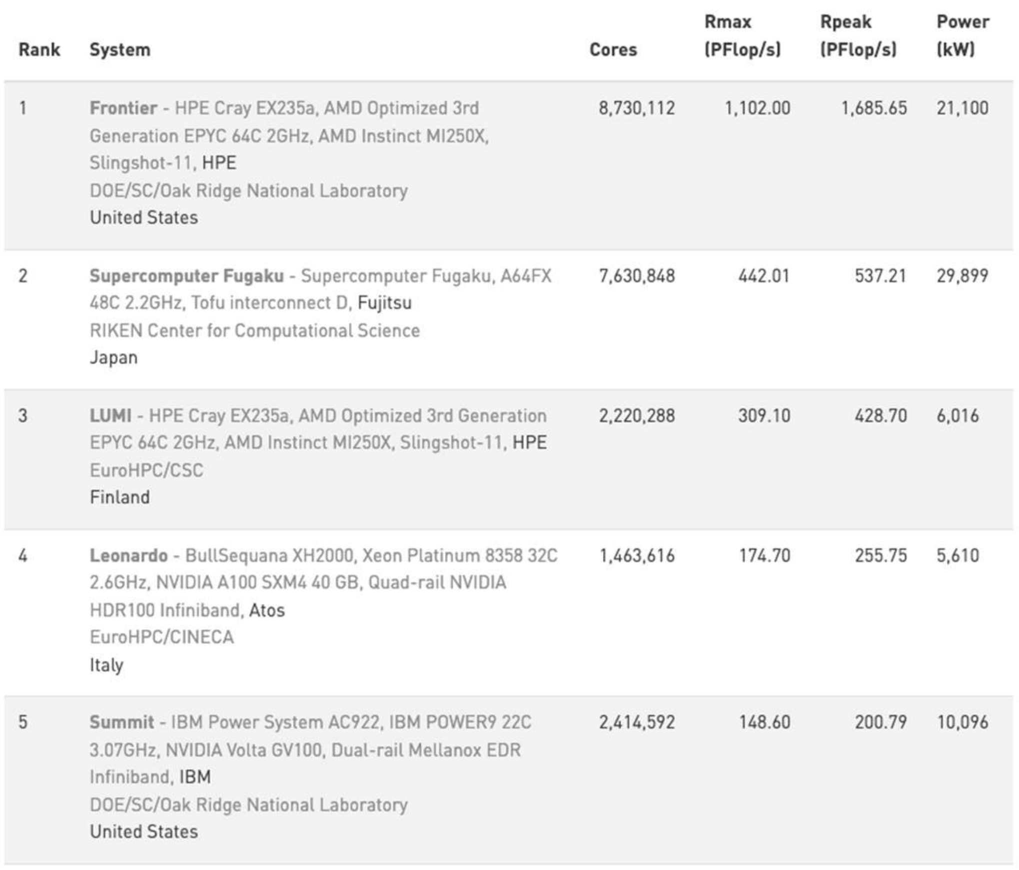
```
#pragma acc parallel loop
 reduction(+:sum)
  for(int i=0;i<n;i++) {
    sum +=
      xcoefs[i]*ycoefs[i];
  }
```

- Because each iteration of the loop adds to the variable sum, we must declare a reduction.
- A parallel reduction may return a slightly different result than a sequential addition due to floating point limitations.

C/C++ vs. FORTRAN

```
// C or C++
#pragma acc <directive> <clauses>
{ … }

! Fortran
!$acc <directive> <clauses>
…
!$acc end <directive>
```

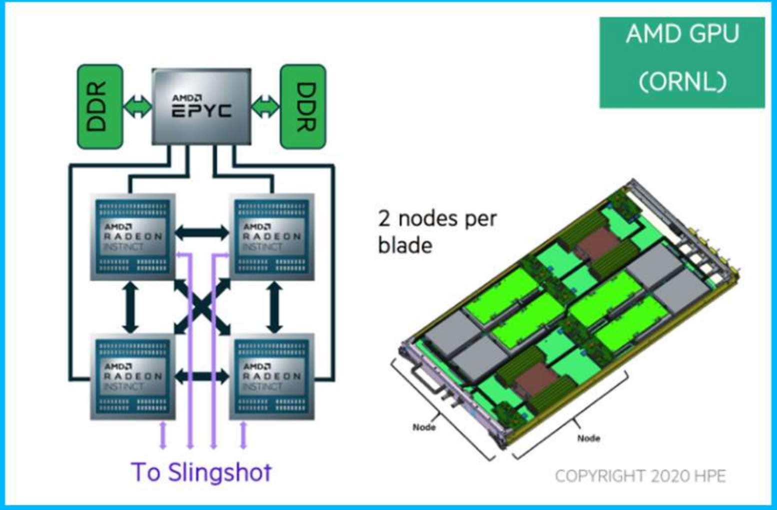
Top 5 Supercomputers (Fall 2022)



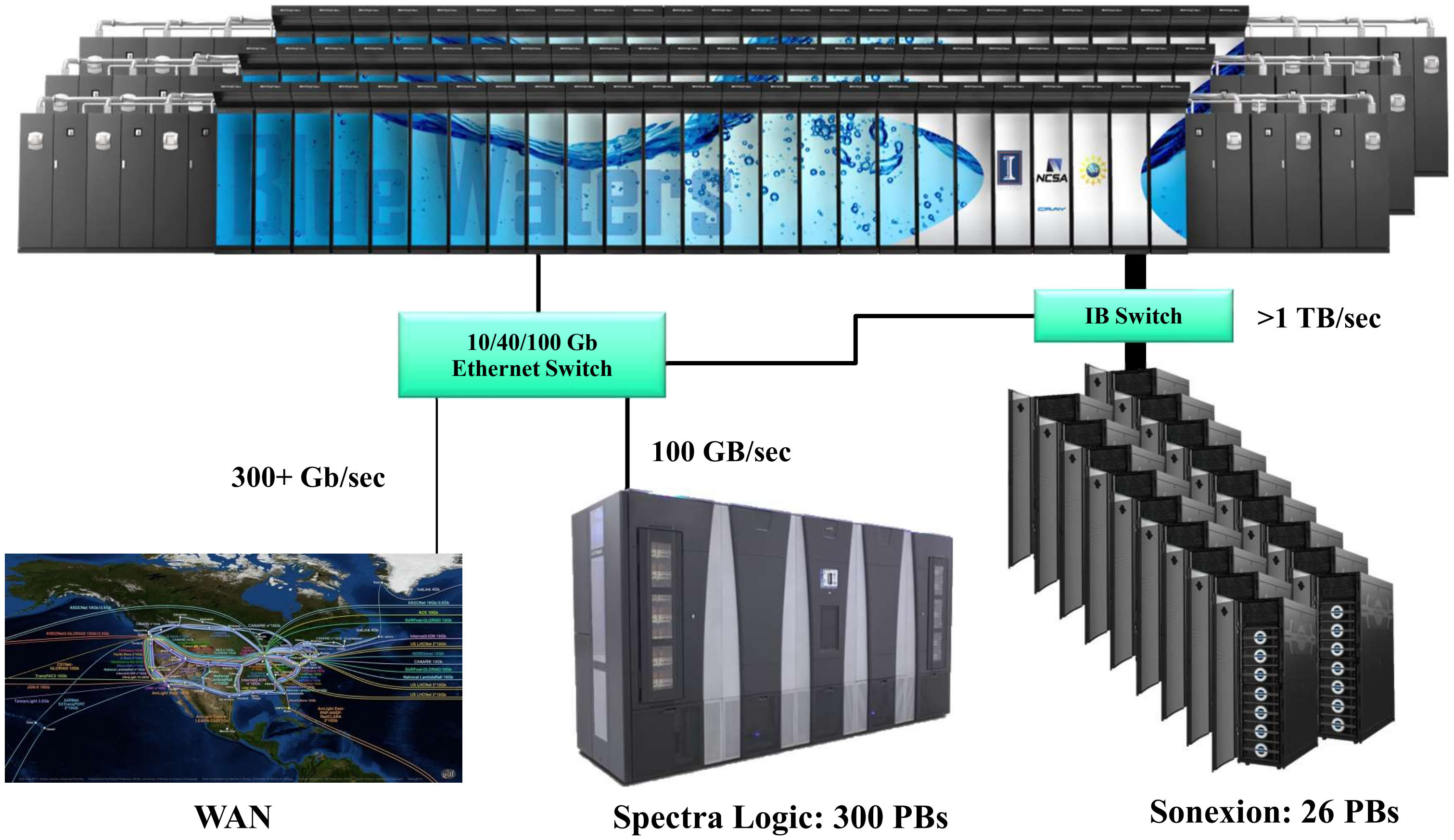
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 40 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096

ORNL Frontier

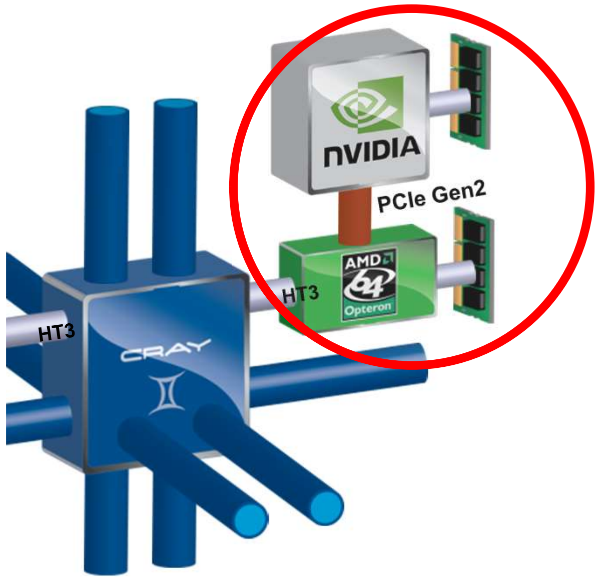
9472 AMD CPUs
37,888 AMD GPUs



Blue Waters @ UIUC (2013-2021)



Cray XK7 Nodes

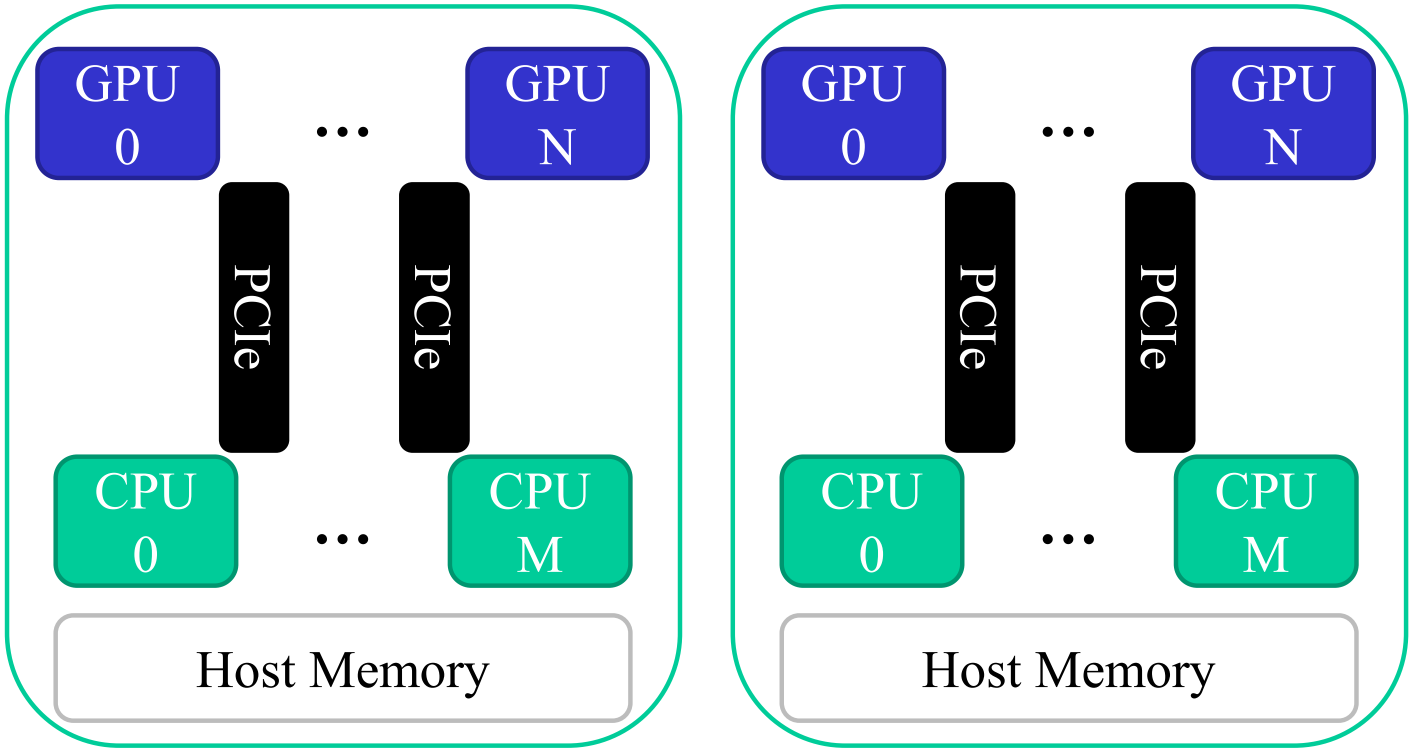


Blue Waters contained 4,224 Cray XK7 compute nodes.

- Dual-socket Node
  - One AMD Interlagos chip
    - 8 core modules, 32 threads
    - 156.5 GFs peak performance
    - 32 GBs memory
      - 51 GB/s bandwidth
  - One NVIDIA Kepler chip
    - 1.3 TFs peak performance
    - 6 GBs GDDR5 memory
      - 250 GB/sec bandwidth
  - Gemini Interconnect
    - Same as XE6 nodes

Abstract CUDA-based Node

- Each node contains *N* GPUs



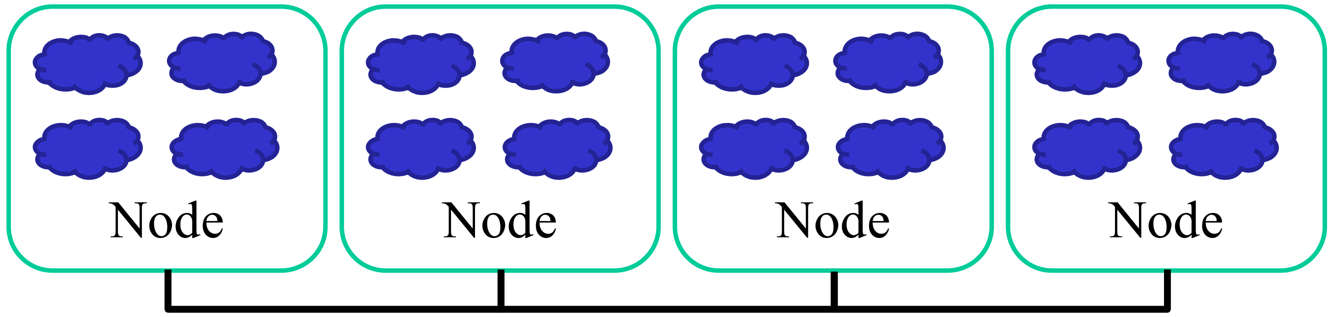
CUDA Support for Multiple GPUs

- **cudaSetDevice()** sets the current GPU
- Current GPU can be changed while async calls (kernels, memcopies) are running
  - It is also OK to queue up a bunch of async calls to a GPU and then switch to another GPU

```
cudaSetDevice( 0 );
kernel<<>>(...);
cudaMemcpyAsync(...);
cudaSetDevice( 1 );
kernel<<>>(...);
```

MPI Model

- Many processes distributed in a cluster



- Each process computes part of the output
- Processes communicate with each other through message passing (not global memory)
- Processes can synchronize through messages

MPI Initialization, Info

- `User launches an MPI job with X processes by executing in the command shell`
  - `MPIrun -np X`
- `int MPI_Init(int *argc, char ***argv)`
  - Initialize MPI
- `MPI_COMM_WORLD`
  - MPI group formed with all allocated nodes
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Rank of the calling process in group of comm
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Number of processes in the group of comm

Vector Addition: Main Process

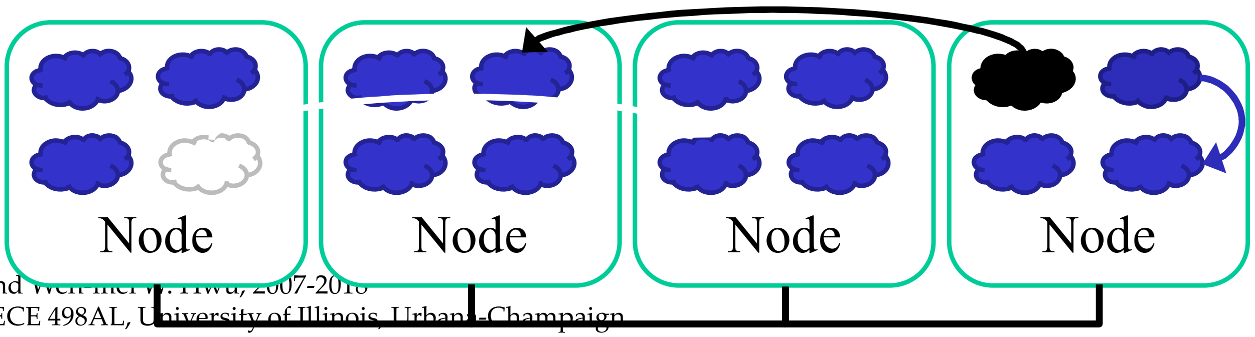
```
int main(int argc, char *argv[]) {
    int vector_size = 1024 * 1024 * 1024;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np < 3) {
        if(0 == pid) printf("Nedded 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_node(vector_size / (np - 1));
    else
        data_server(vector_size);

    MPI_Finalize();
    return 0;
}
```

MPI Sending Data

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - buf: Starting address of send buffer
  - count: Number of elements in send buffer (nonnegative integer)
  - datatype: Datatype of each send buffer element
  - dest: Rank of destination (integer)
  - tag: Message tag (integer)
  - comm: Communicator (handle)

MPI Sending Data

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Buf: Initial address of send buffer
  - Count: Number of elements in send buffer (nonnegative integer)
  - Datatype: Datatype of each send buffer element
  - Dest: Rank of destination (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)

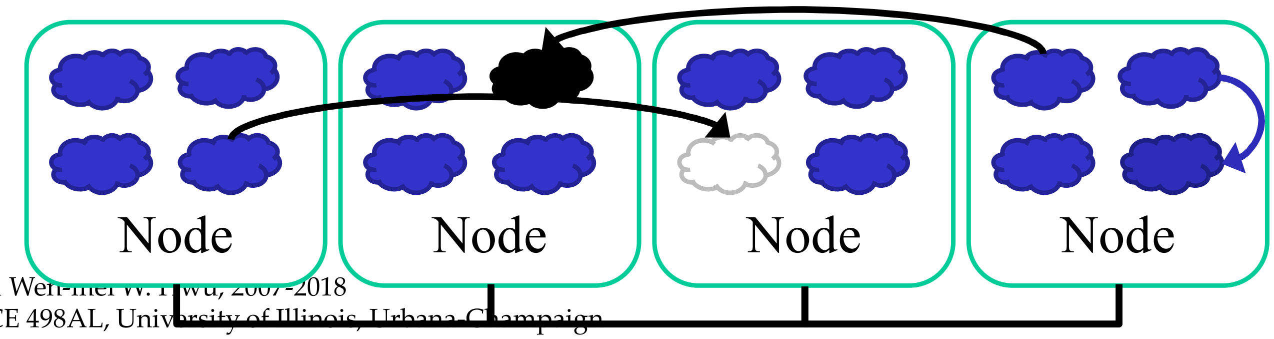


MPI Receiving Data

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - buf: Starting address of receive buffer
  - count: Maximum number of elements in receive buffer (non-negative integer)
  - datatype: Datatype of each receive buffer element
  - source: Rank of source (integer)
  - tag: Message tag (integer)
  - comm: Communicator (handle)
  - status: Status object

MPI Receiving Data

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - Buf: Initial address of receive buffer
  - Count: Maximum number of elements in receive buffer (non-negative integer)
  - Datatype: Datatype of each receive buffer element
  - Source: Rank of source (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)
  - Status: Status object (Status)



Vector Addition: Server Process (I)

```
void data_server(unsigned int vector_size) {
    int np, num_nodes = np – 1, first_node = 0, last_node = np - 2;
    unsigned int num_bytes  = vector_size * sizeof(float);
    float *input_a = 0, *input_b = 0, *output = 0;

    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    /* Allocate input data */
    input_a = (float *)malloc(num_bytes);
    input_b = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if(input_a == NULL || input_b == NULL || output == NULL) {
        printf(“Server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data */
    random_data(input_a, vector_size , 1, 10);
    random_data(input_b, vector_size ,
```

Vector Addition: Server Process (II)

```
    /* Send data to compute nodes */
    float *ptr_a = input_a;
    float *ptr_b = input_b;

    for (int process = 1; process < last_node; process++) {
        MPI_Send(ptr_a, vector_size / num_nodes, MPI_FLOAT,
                process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
        ptr_a += vector_size / num_nodes;

        MPI_Send(ptr_b, vector_size / num_nodes, MPI_FLOAT,
                process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
        ptr_b += vector_size / num_nodes;
    }
```

Vector Addition: Server Process (III)

```
    /* Wait for compute to complete*/
    MPI_Barrier(MPI_COMM_WORLD);

    /* Collect output data */
    MPI_Status status;
    for (int process = 0; process < num_nodes; process++) {
        MPI_Recv(output + process * num_points / num_nodes,
            num_points / num_comp_nodes, MPI_REAL, process,
            DATA_COLLECT, MPI_COMM_WORLD, &status );
    }

    /* Store output data */
    store_output(output, dimx, dimy, dimz);

    /* Release resources */
    free(input);
    free(output);
}
```

Vector Addition: Compute Process (I)

```
void compute_node(unsigned int vector_size ) {
    int np;
    unsigned int num_bytes = vector_size * sizeof(float);
    float *input_a, *input_b, *output;
    MPI_Status status;

    MPI_Comm_size(MPI_COMM_WORLD, &np);
    int server_process = np - 1;

    /* Alloc host memory */
    input_a = (float *)malloc(num_bytes);
    input_b = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);

    /* Get the input data from server process */
    MPI_Recv(input_a, vector_size, MPI_FLOAT, server_process,
            DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
    MPI_Recv(input_b, vector_size, MPI_FLOAT, server_process,
            DATA_DISTRIBUTE, MPI_COMM_WORLD, &status);
```

Vector Addition: Compute Process (II)

```
    /* Compute the partial vector addition */
    for (int i = 0; i < vector_size; ++i) {
        output[i] = input_a[i] + input_b[i];
    }

    /* Or, can offload to GPU here */
    /* cudaMalloc(), cudaMemcpy(), kernel launch, etc. */

    MPI_Barrier(MPI_COMM_WORLD);

    /* Send the output */
    MPI_Send(output, vector_size, MPI_FLOAT,
            server_process, DATA_COLLECT, MPI_COMM_WORLD);

    /* Release memory */
    free(input_a);
    free(input_b);
    free(output);
}
```

QUESTIONS?
READ CHAPTER 15!

Also see https://developer.nvidia.com/intro-to-openacc-course-2016