

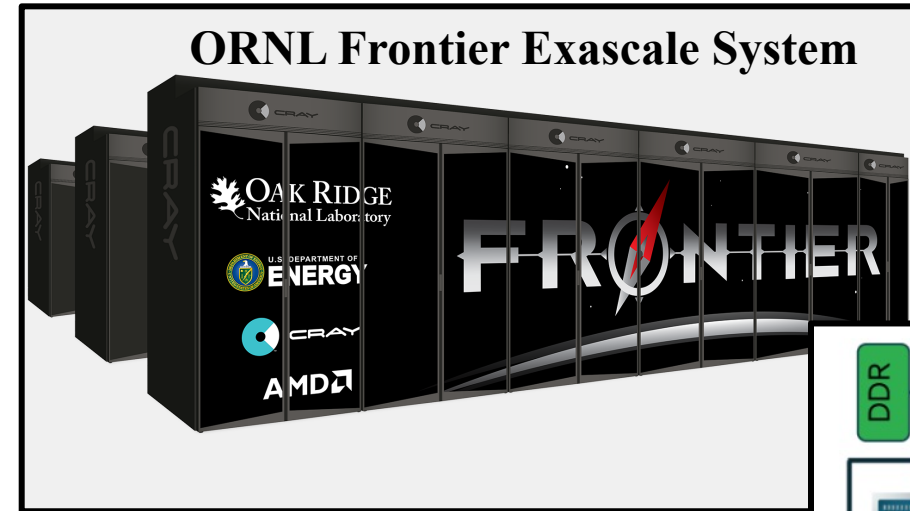
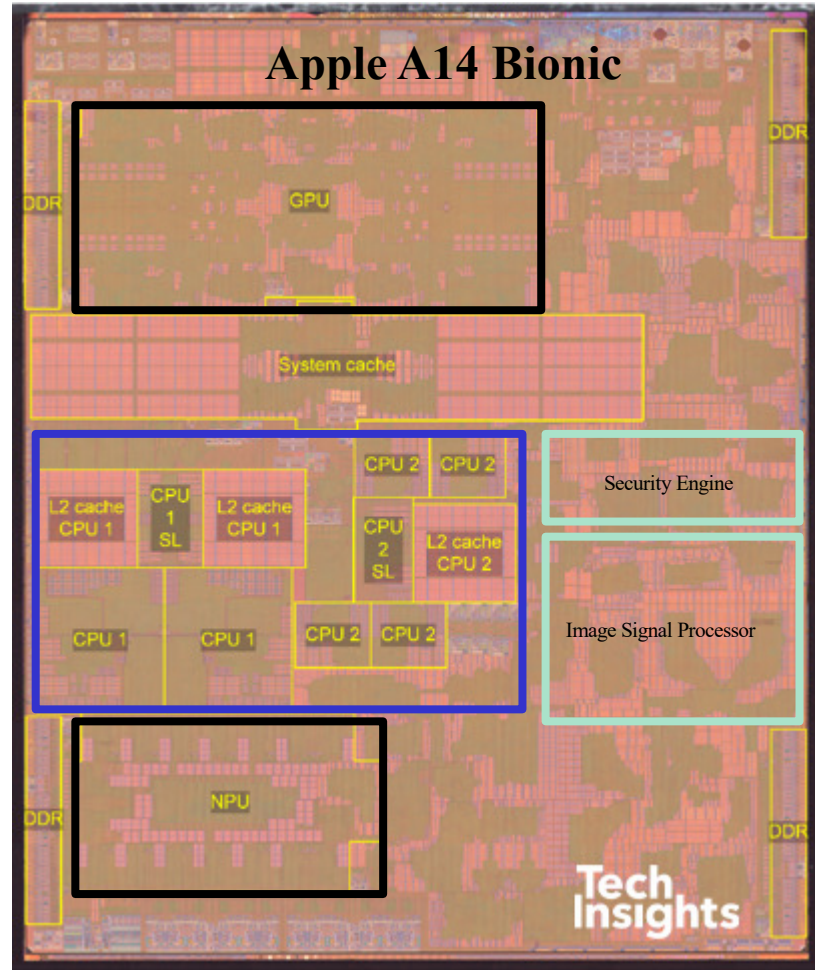
Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

ECE408 / CS483/CSE408

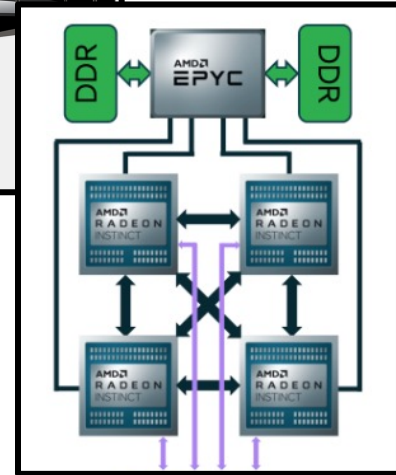
Applied Parallel Programming

Lecture 23:
Alternatives to CUDA

Accelerated Computing is no longer a question



From the smallest to the largest, computing today involves accelerators architectures.



CUDA is just one model for Compute Acceleration

OpenGL (1992)					OpenACC (2012)	Metal (2014)
DirectX (1995)	GPGPU (2002)	CUDA (2007)	OpenCL (2008)			SYCL (2014)
					C++AMP (2013)	Vulkan (2016)
					RenderScript (2013)	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.

NVIDIA CUDA-X Ecosystem

USE-CASES



Speech

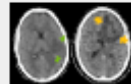


Translate



Recommender

CONSUMER INTERNET



Healthcare



Manufacturing

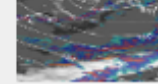


Finance

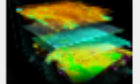
INDUSTRIAL APPLICATIONS



Molecular
Simulations



Weather
Forecasting



Seismic
Mapping

SUPERCOMPUTING

APPS & FRAMEWORKS



RAPIDS



PYTORCH



Chainer



ONNX

Amber

NAMD

+600
Applications



AUTODESK
3DS MAX



CUDA-X LIBRARIES

MACHINE LEARNING

cuDF

cuML

cuGRAPH

DL / HPC

cuDNN

CUTLASS

TENSORRT

CUDA Math Libraries

LANGUAGES



python OpenACC



LLVM Compiler
For CUDA

CUDA

CUDA TOOLKIT

CUDA
COMPILER

DEVELOPER TOOLS

DEBUGGERS

PROFILERS

CUDA C++
CORE

CUDA DRIVER

MEMORY
MANAGEMENT

WINDOWS &
GRAPHICS

COMMS
LIBRARIES

OS PLATFORMS

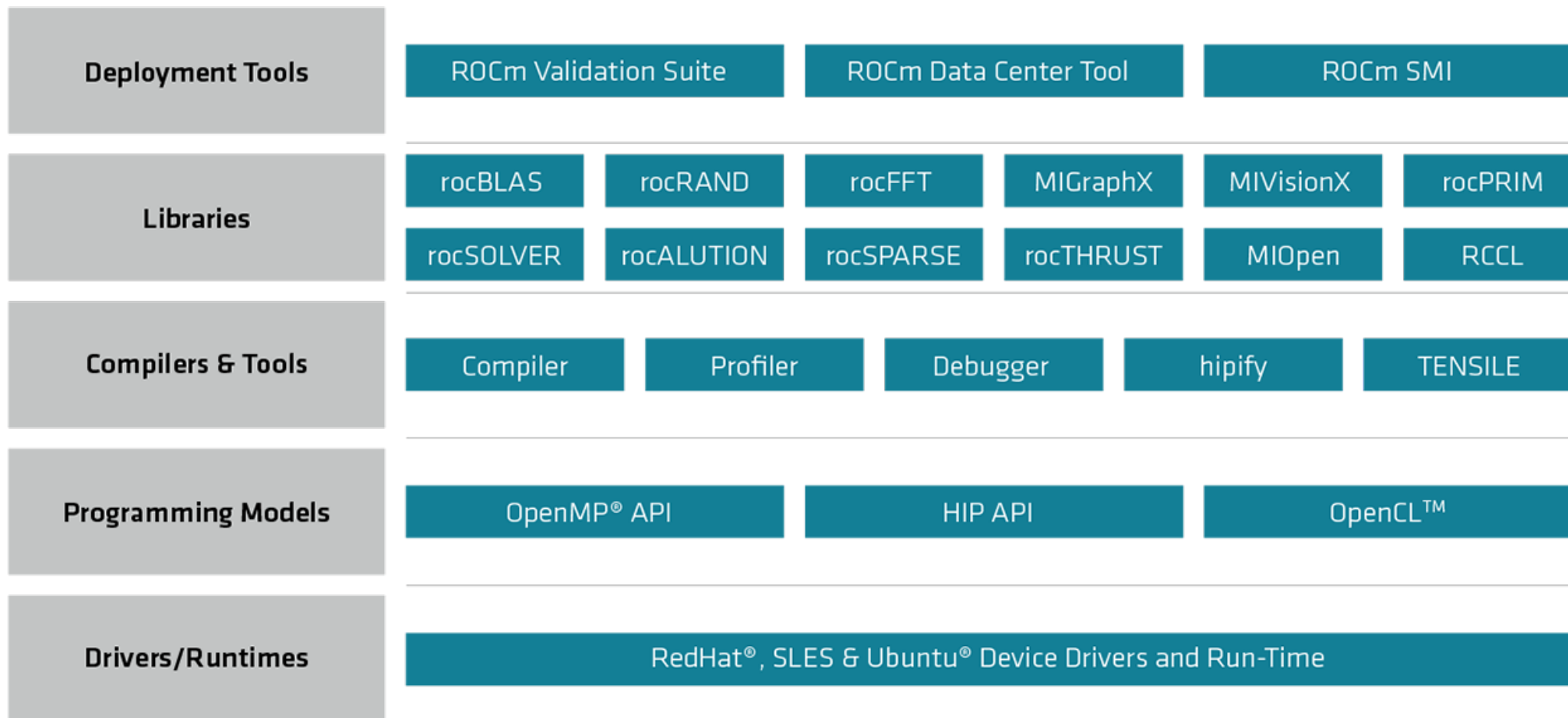


CentOS



Windows Server

AMD ROCm Ecosystem



”Universal” Accelerator API

- Massive Fine-grained Threading
 - Bulk Synchronized (eg., synctreads)
 - Thread hierarchies (eg., warps, blocks, grids)
- Non-uniform, non-coherent software controlled memory hierarchy
- Custom instructions and datatypes
- CPU-side: Kernels, mem-copies, task graphs and work streams
- Support for low-effort/good-gain & high-effort/high-gain coding

Using Nvidia GPUs with PyTorch

```
import torch
import torch.nn as nn
from torch.nn import functional as F

# hyperparameters
batch_size = 64 # how many independent sequences will we process in parallel?
block_size = 256 # what is the maximum context length for predictions?
max_iters = 5000
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 6
dropout = 0.2
# -----

torch.manual_seed(1337)

#wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
> with open('input.txt', 'r', encoding='utf-8') as f:--

|# data loading
> def get_batch(split):--

@torch.no_grad()
> def estimate_loss():--

> class Head(nn.Module):--

> class MultiHeadAttention(nn.Module):--

> class FeedFoward(nn.Module):--

> class Block(nn.Module):--

> class GPTLanguageModel(nn.Module):--

model = GPTLanguageModel()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

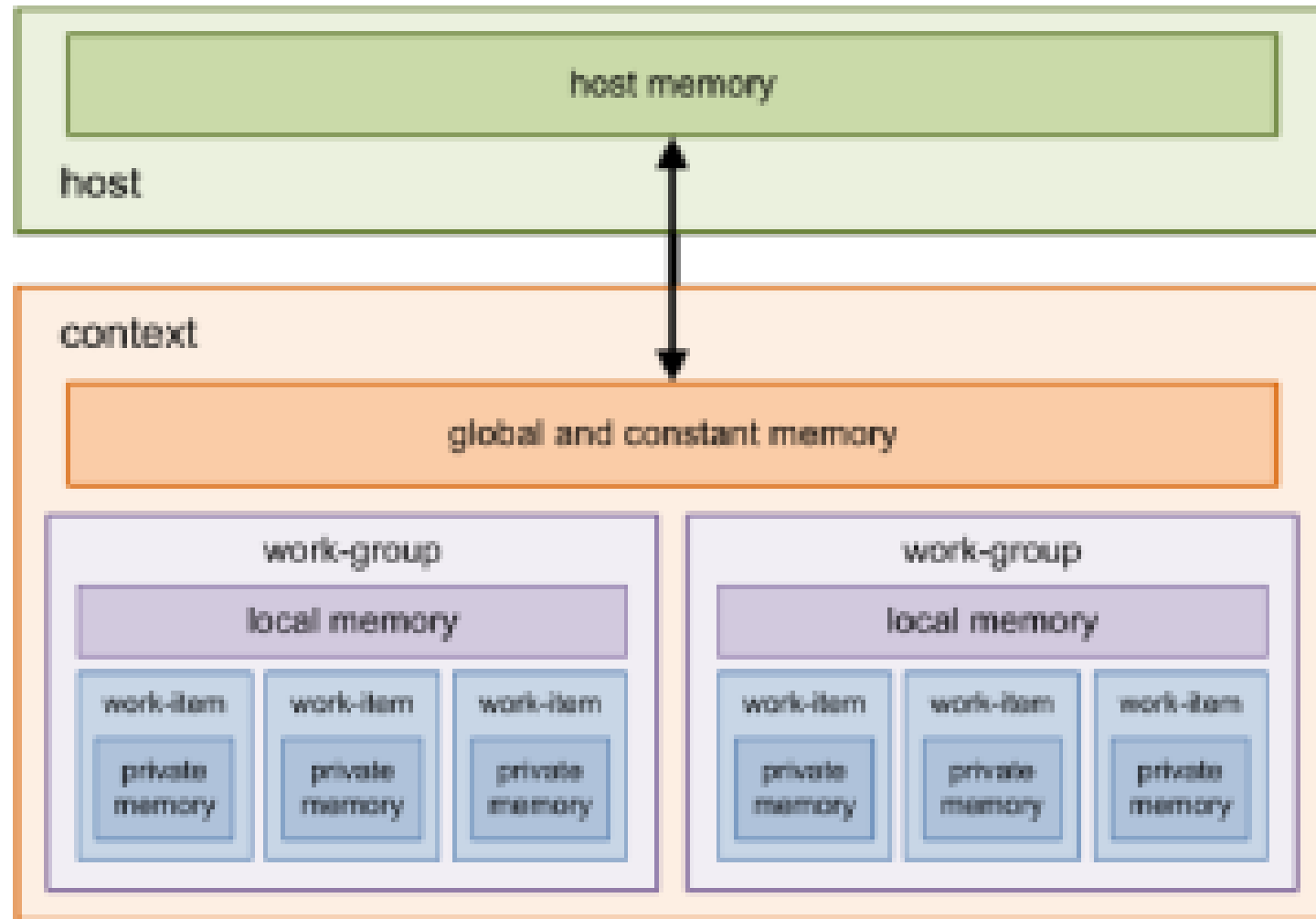
OpenCL, OpenACC, MPI

- OpenCL: An Open Standard Acceleration API
- OpenACC: A “Low-Code” Acceleration API
- MPI: A Large Scale, Multi-Node Parallel API

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`

```
1.// Tiled and coalesced version
2.__kernel void myGEMM2(int M, int N, int K, __global float* A, __global float* B, __global float* C) {
3.
4.    // Thread identifiers
5.    const int row = get_local_id(0); // Local row ID (max: TS)
6.    const int col = get_local_id(1); // Local col ID (max: TS)
7.    const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
8.    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)
9.
10.   // Local memory to fit a tile of TS*TS elements of A and B
11.   __local float Asub[TS][TS];
12.   __local float Bsub[TS][TS];
13.
14.   // Initialise the accumulation register
15.   float acc = 0.0f;
16.   // Loop over all tiles
17.   const int numTiles = K/TS;
18.   for (int t=0; t<numTiles; t++) {
19.
20.       // Load one tile of A and B into local memory
21.       const int tiledRow = TS*t + row;
22.       const int tiledCol = TS*t + col;
23.       Asub[col][row] = A[tiledCol*M + globalRow];
24.       Bsub[col][row] = B[globalCol*K + tiledRow];
25.
26.       // Synchronise to make sure the tile is loaded
27.       barrier(CLK_LOCAL_MEM_FENCE);
28.
29.       // Perform the computation for a single tile
30.       for (int k=0; k<TS; k++)
31.           acc += Asub[k][row] * Bsub[col][k];
32.
33.       // Synchronise before loading the next tile
34.       barrier(CLK_LOCAL_MEM_FENCE);
35.   }
36.
37.   // Store the final result in C
38.   C[globalCol*M + globalRow] = acc;
39.}
```

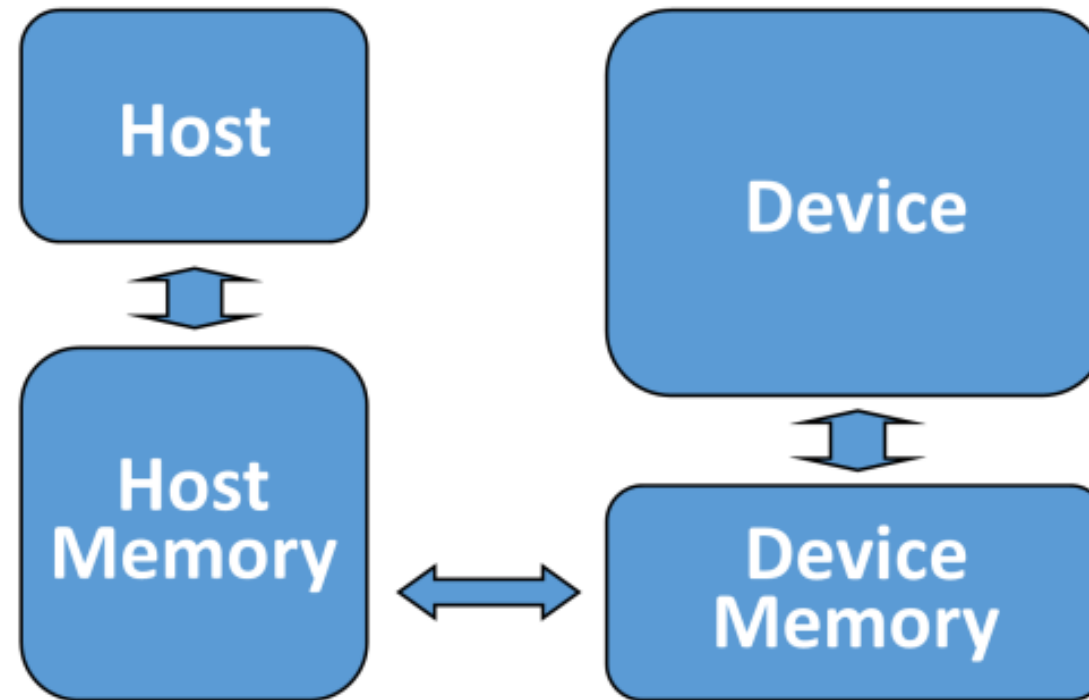
OpenACC

- The OpenACC Application Programming Interface (API) provides a set of
 - compiler directives (pragmas)
 - library routines and
 - environment variablesthat can be used to write data parallel FORTRAN, C and C++ programs that run on accelerator devices including GPUs and CPUs

OpenACC Pragmas

- In C and C++, the `#pragma` directive is the means to provide, to the compiler, information that is not specified in the standard language.

The OpenACC Abstract Machine Model



Simple Matrix-Matrix Multiplication in OpenACC

```
1 void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2 {
3
4     #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw]) copyout(P[0:Mh*Nw])
5     for (int i=0; i<Mh; i++) {
6         #pragma acc loop
7         for (int j=0; j<Nw; j++) {
8             float sum = 0;
9             for (int k=0; k<Mw; k++) {
10                 float a = M[i*Mw+k];
11                 float b = N[k*Nw+j];
12                 sum += a*b;
13             }
14             P[i*Nw+j] = sum;
15         }
16     }
17 }
```

Some Observations

- The code is almost identical to the sequential version, except for the two lines with `#pragma` at line 4 and line 6.
- OpenACC uses compiler directives to extend the base language.
 - `#pragma` at line 4 tells the compiler to generate code for the ‘i’ loop at line 5 through 16 so that the loop iterations are executed in parallel on the accelerator.
 - The `copyin` clause and the `copyout` clause specify how the matrix data should be transferred between the host and the accelerator. The `#pragma` at line 6 instructs the compiler to map the inner ‘j’ loop to the second level of parallelism on the accelerator.

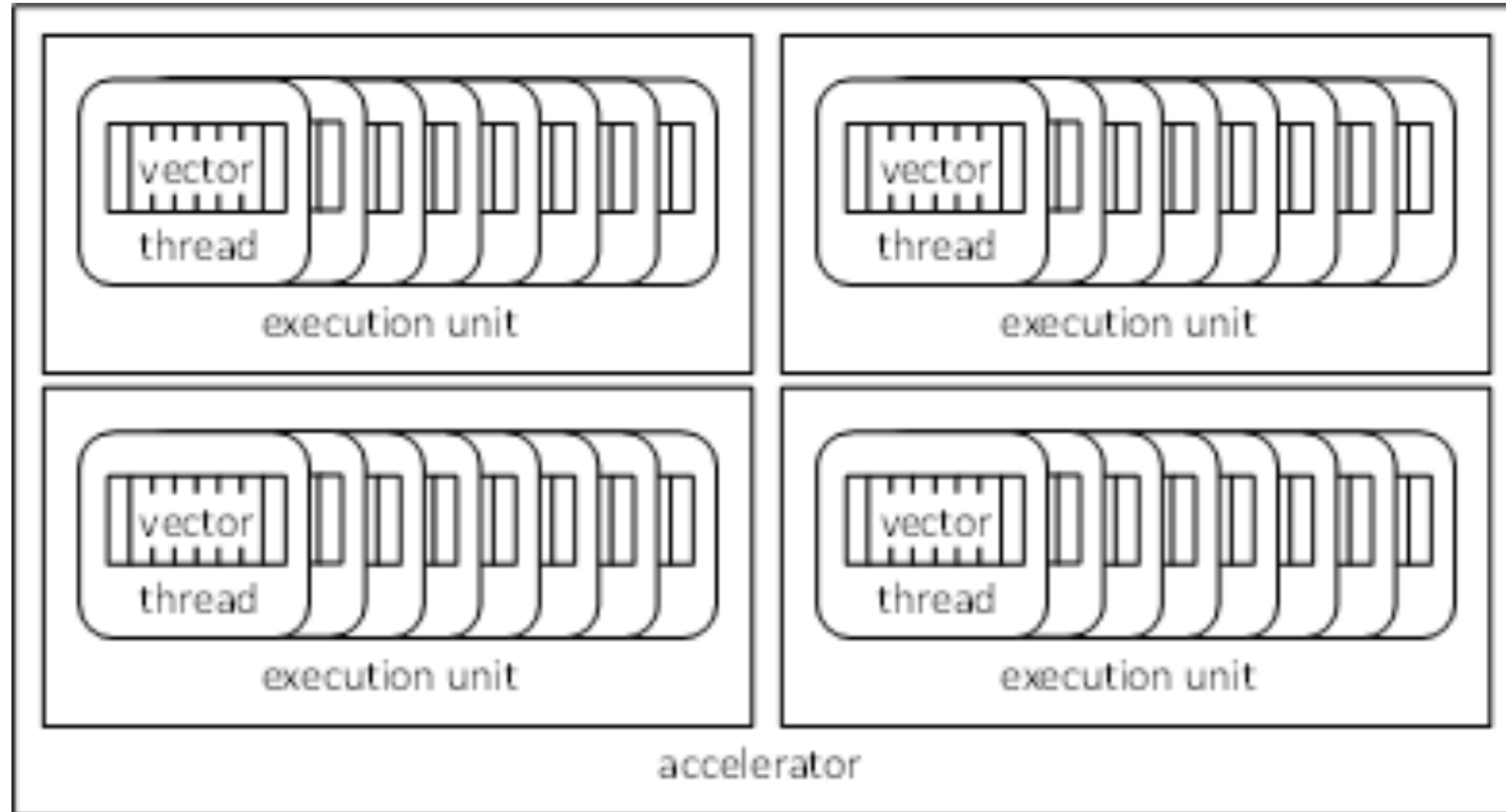
Motivation

- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
 - leave most of the details in generating a kernel and data transfers to the OpenACC compiler.
- OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

Frequently Encountered Issues

- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
 - The performance of an OpenACC depends heavily on the quality of the compiler.
 - Much less so in CUDA or OpenCL, which are explicit
- Some OpenACC programs may behave differently or even incorrectly if pragmas are ignored

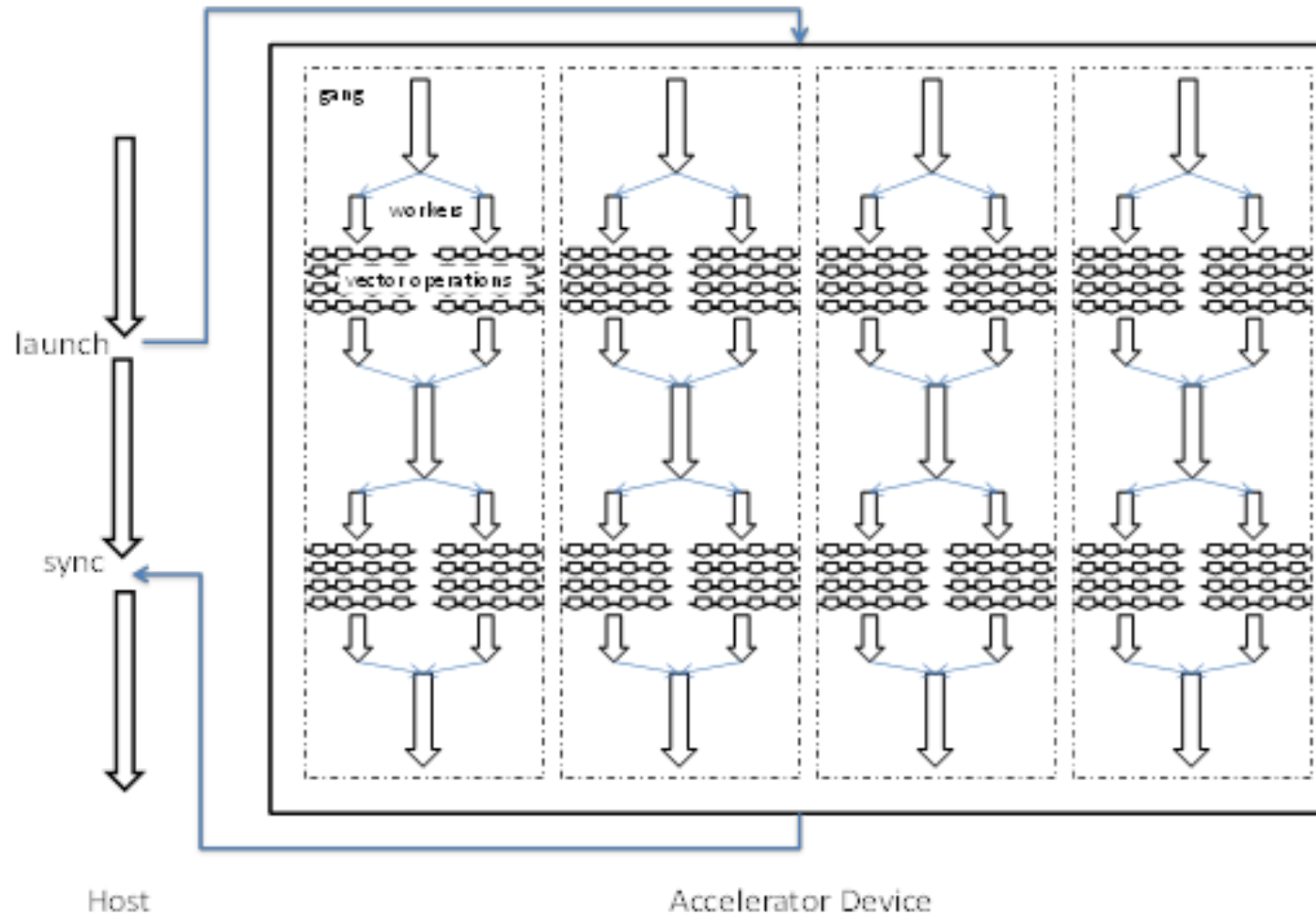
OpenACC Device Model



There are is also a minimal set of APIs in OpenACC, for things like synchronization, memory management, etc

OpenACC Execution Model

(Terminology: Gangs and Workers)



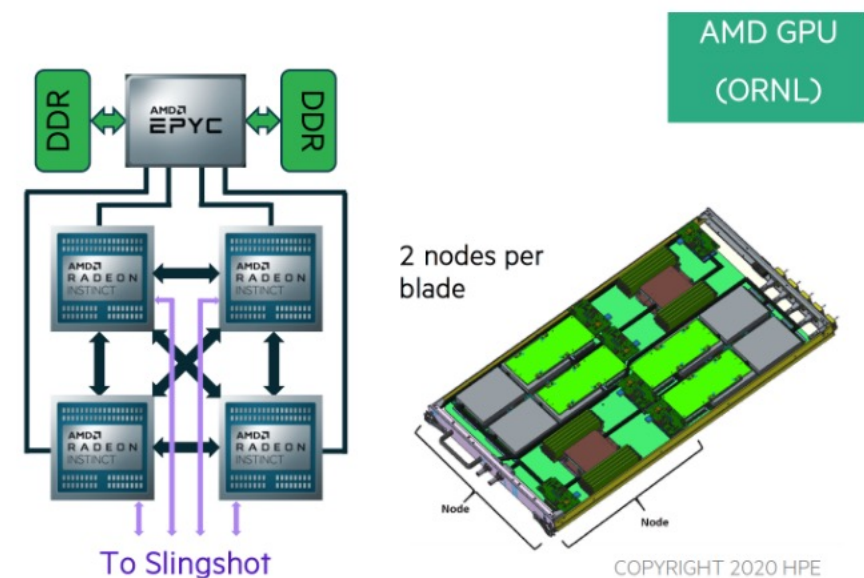
Top 5 Supercomputers (Fall 2022)

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

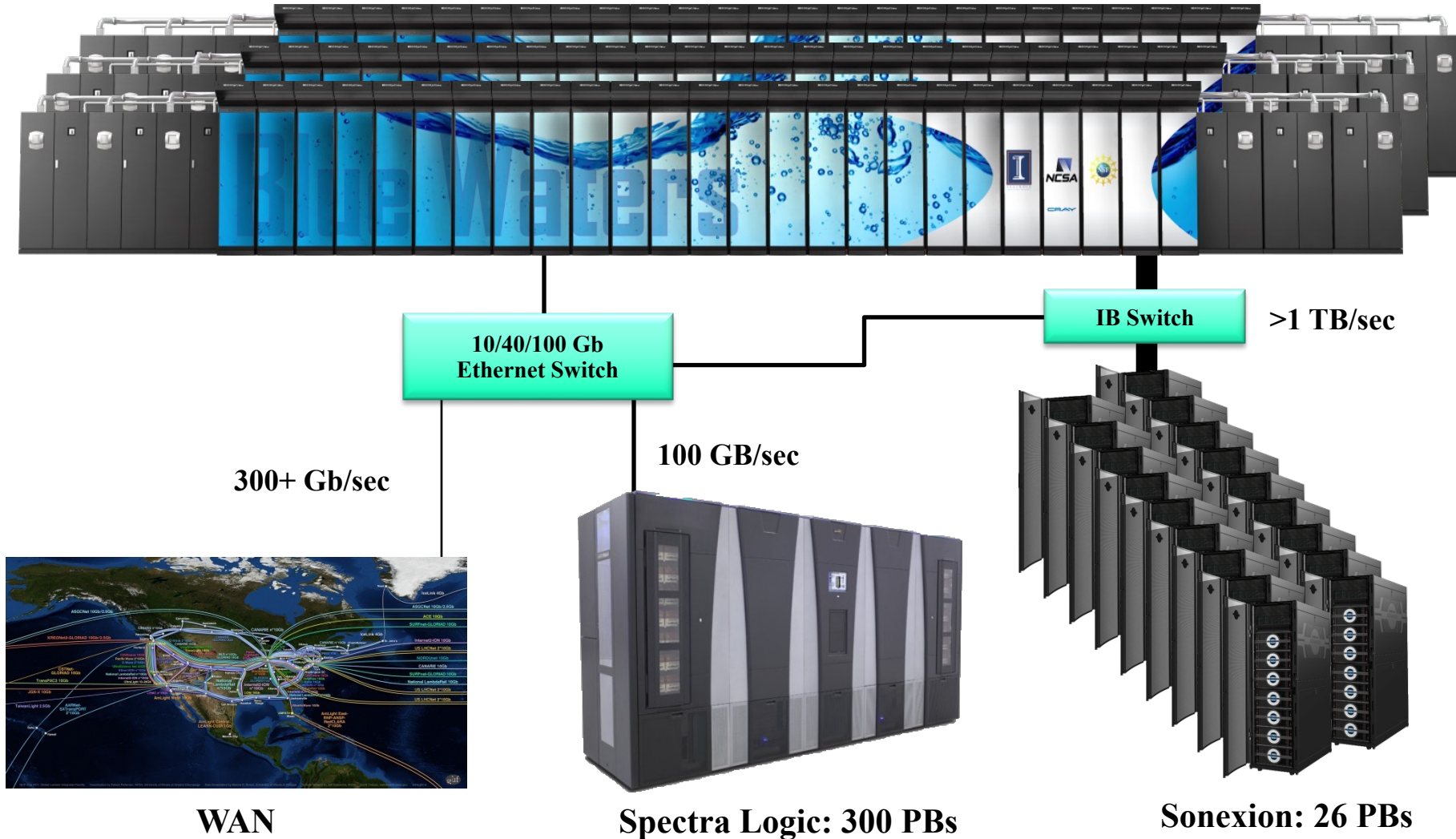
ORNL Frontier

9472 AMD CPUs

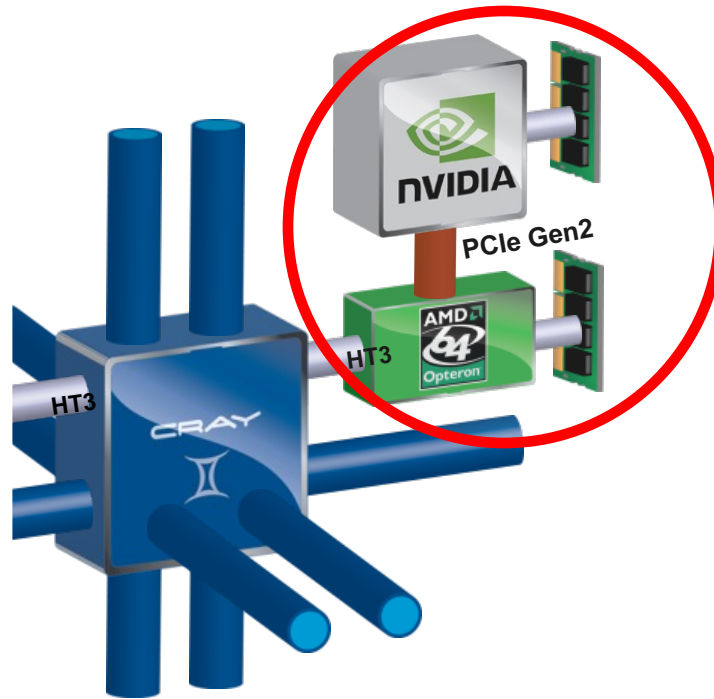
37,888 AMD GPUs



Blue Waters @ UIUC (2013-2021)



Cray XK7 Nodes



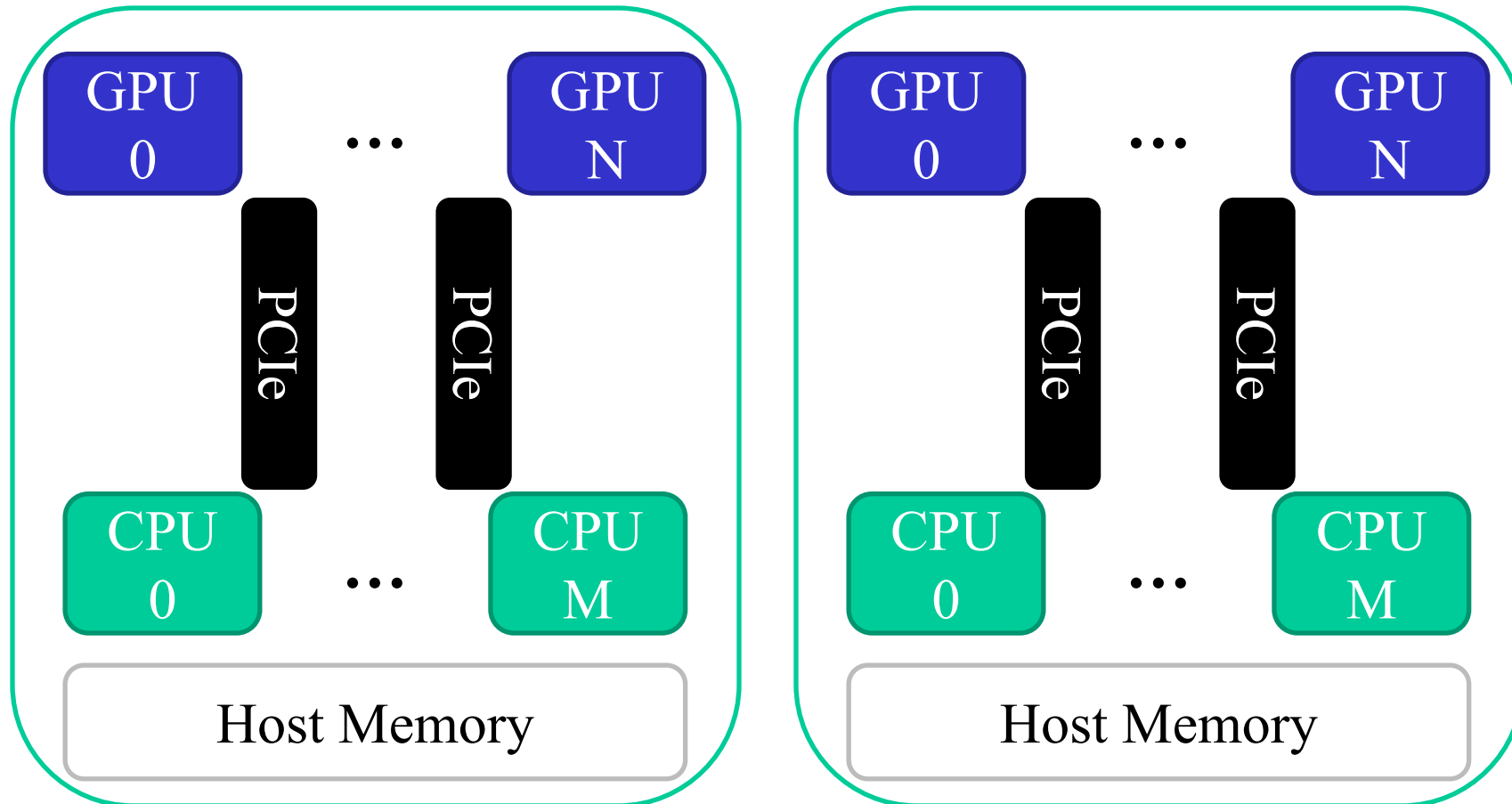
**Blue Waters contains 4,224
Cray XK7 compute nodes.**

- 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

Science Area	Number of Teams	Codes	Struct Grids	Unstruct Grids	Dense Matrix	Sparse Matrix	N-Body	Monte Carlo	FF T	PIC	Significant I/O
Climate and Weather	3	CESM, GCRM, CM1/WRF, HOMME	X	X		X		X			X
Plasmas/Magnetosphere	2	H3D(M), VPIC, OSIRIS, Magtail/UPIC	X				X		X		X
Stellar Atmospheres and Supernovae	5	PPM, MAESTRO, CASTRO, SEDONA, ChaNGa, MS-FLUKSS	X			X	X	X		X	X
Cosmology	2	Enzo, pGADGET	X			X	X				
Combustion/Turbulence	2	PSDNS, DISTUF	X						X		
General Relativity	2	Cactus, Harm3D, LazEV	X			X					
Molecular Dynamics	4	AMBER, Gromacs, NAMD, LAMMPS				X	X		X		
Quantum Chemistry	2	SIAL, GAMESS, NWChem			X	X	X	X			X
Material Science	3	NEMOS, OMEN, GW, QMCPACK			X	X	X	X			
Earthquakes/Seismology	2	AWP-ODC, HERCULES, PLSQR, SPECFEM3D	X	X			X				X
Quantum Chromo Dynamics	1	Chroma, MILC, USQCD	X		X	X					
Social Networks	1	EPISIMDEMICS									
Evolution	1	Eve									
Engineering/System of Systems	1	GRIPS, Revisit						X			
Computer Science	1			X	X	X			X		X

Abstract CUDA-based Node

- Each node contains N GPUs



CUDA Support for Multiple GPUs

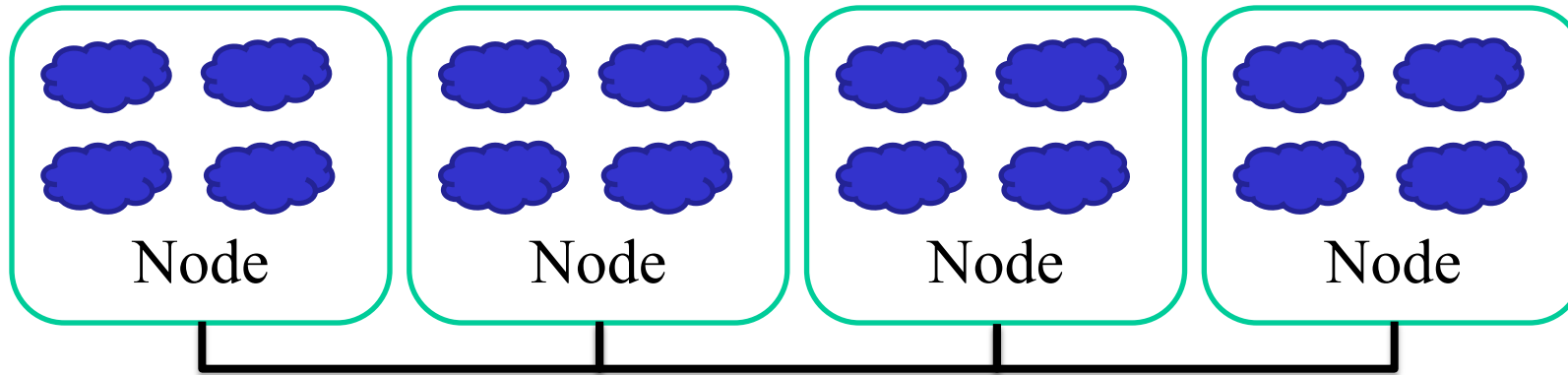
Managing multiple GPUs from a single CPU thread

`cudaSetDevice()` - sets the current 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
 - `MPRun -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("Needed 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);

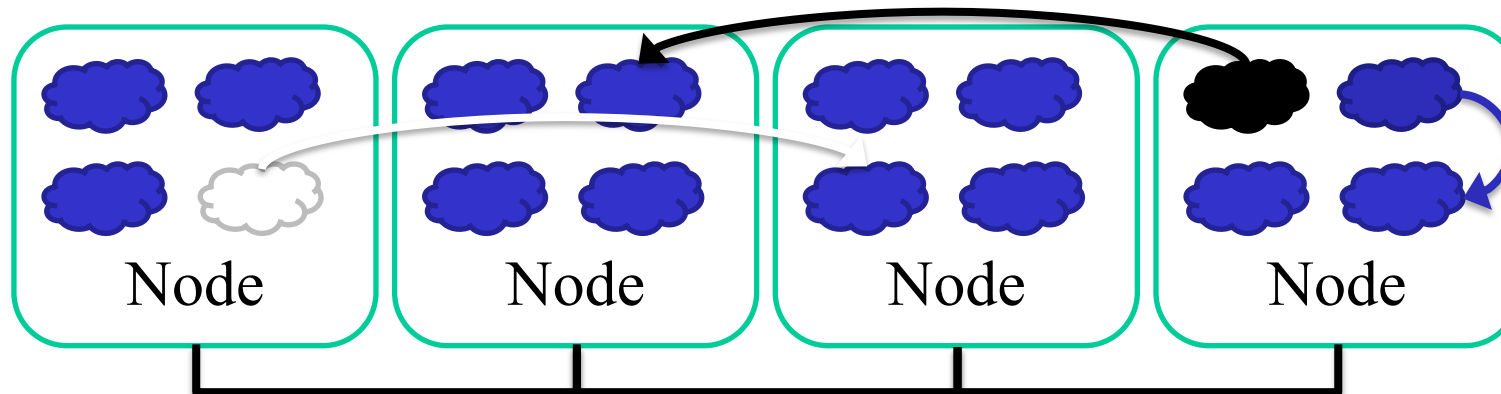
    MPI_Finalize();
    return 0;
}
```

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)

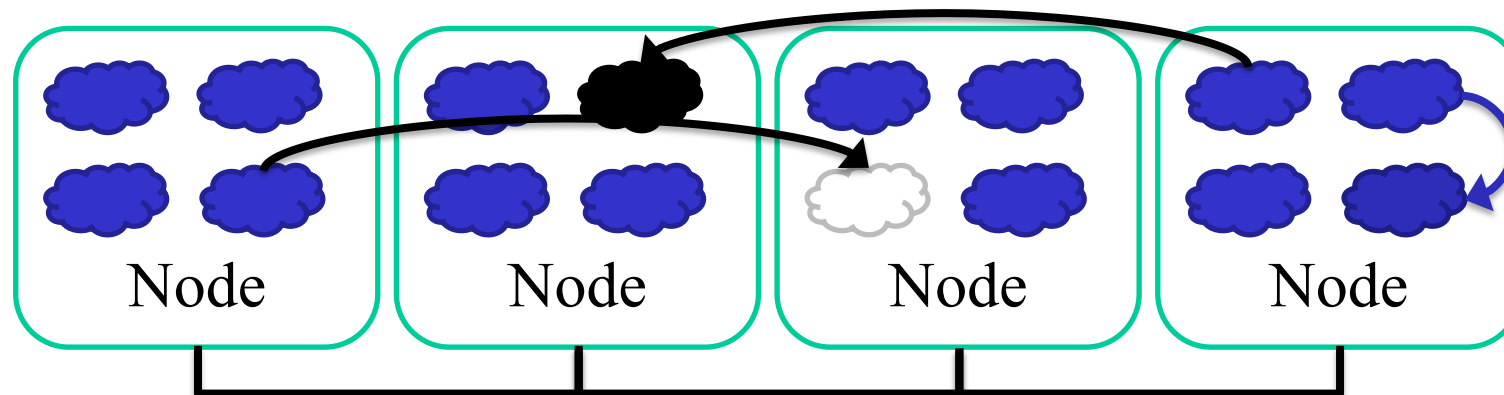


MPI Receiving Data

- `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);
}
```



ANY MORE QUESTIONS?
READ CHAPTER 15