中国科学院大学计算机学院专业选修课（硕博通用课程）

GPU架构与编程

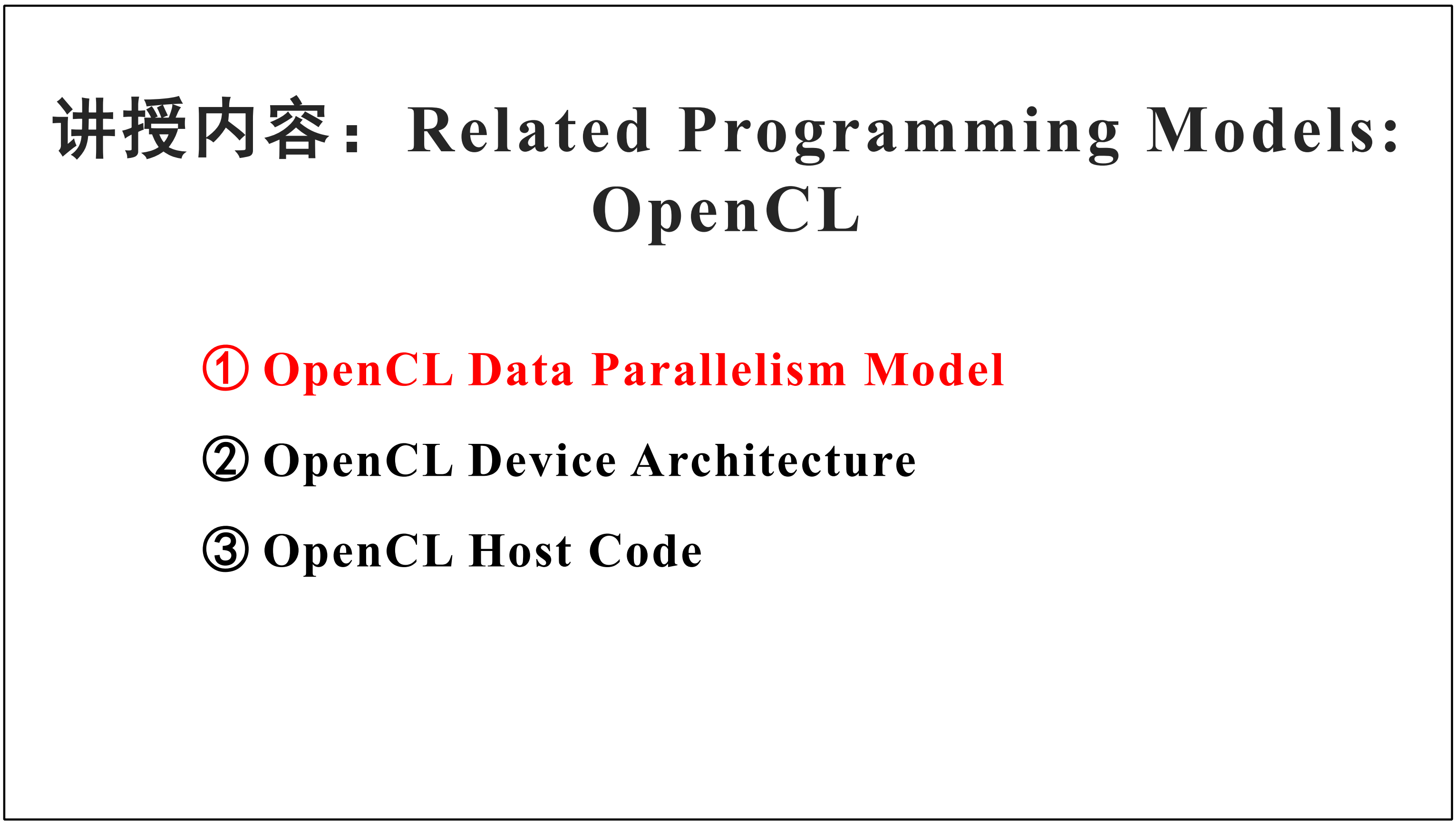
第五课：CUDA编程（四）

赵地
中科院计算所

2024年秋季学期

讲授内容

- ➢Related Programming Models: OpenCL
- ➢Related Programming Models: OpenACC
- ➢Multi-GPU: OpenMP
- ➢Related Programming Models: MPI

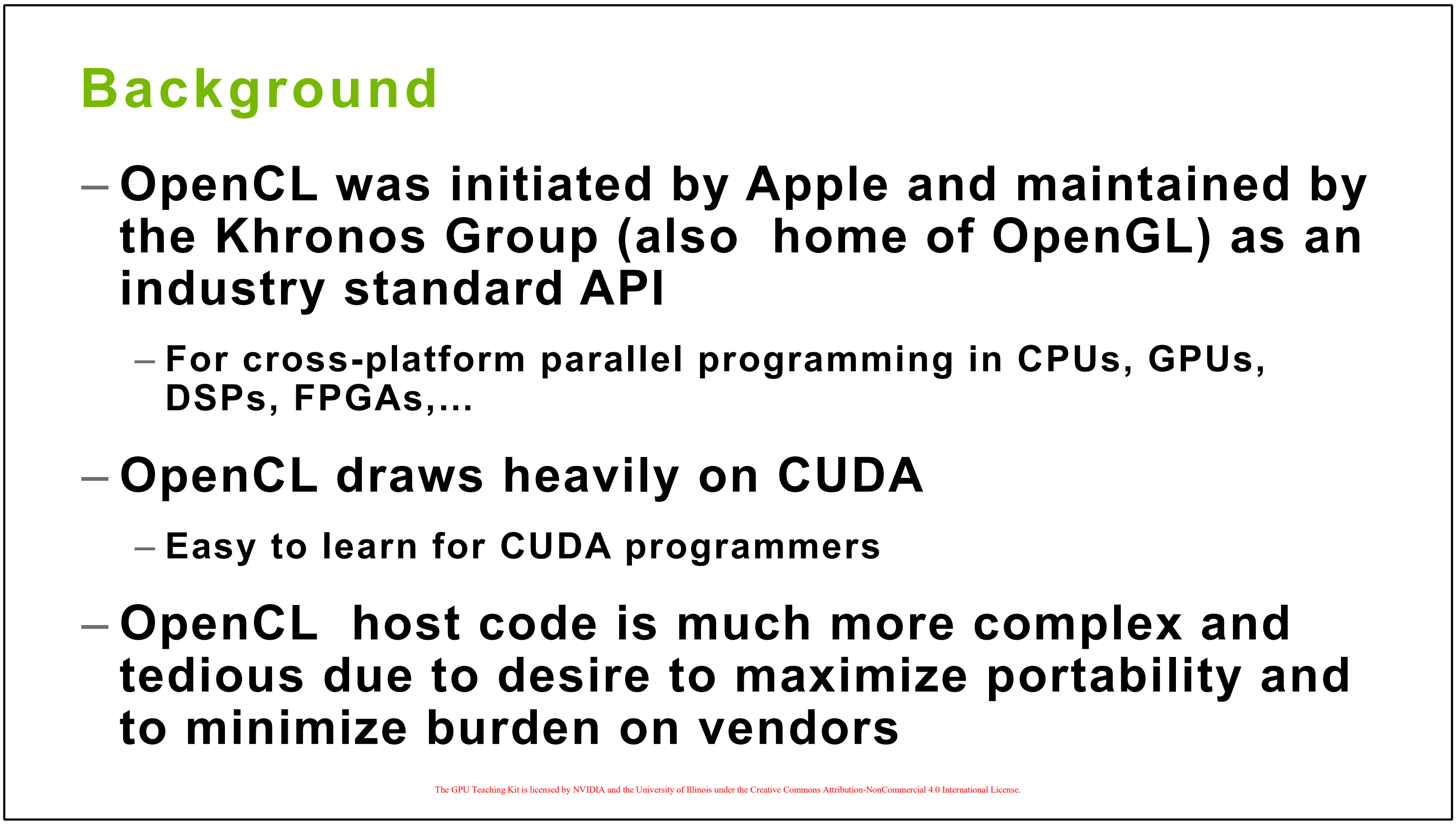


讲授内容：Related Programming Models: OpenCL

① OpenCL Data Parallelism Model

② OpenCL Device Architecture

③ OpenCL Host Code

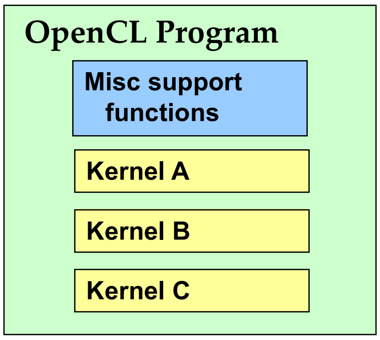


Background

- OpenCL was initiated by Apple and maintained by the Khronos Group (also home of OpenGL) as an industry standard API
  - For cross-platform parallel programming in CPUs, GPUs, DSPs, FPGAs,...
- OpenCL draws heavily on CUDA
  - Easy to learn for CUDA programmers
- OpenCL host code is much more complex and tedious due to desire to maximize portability and to minimize burden on vendors

OpenCL Programs

- An OpenCL "program" is a C program that contains one or more "kernels" and any supporting routines that run on a target device
- An OpenCL kernel is the basic unit of parallel code that can be executed on a target device



OpenCL Program

Misc support functions

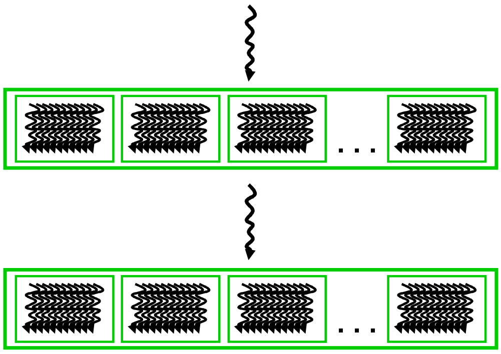
Kernel A

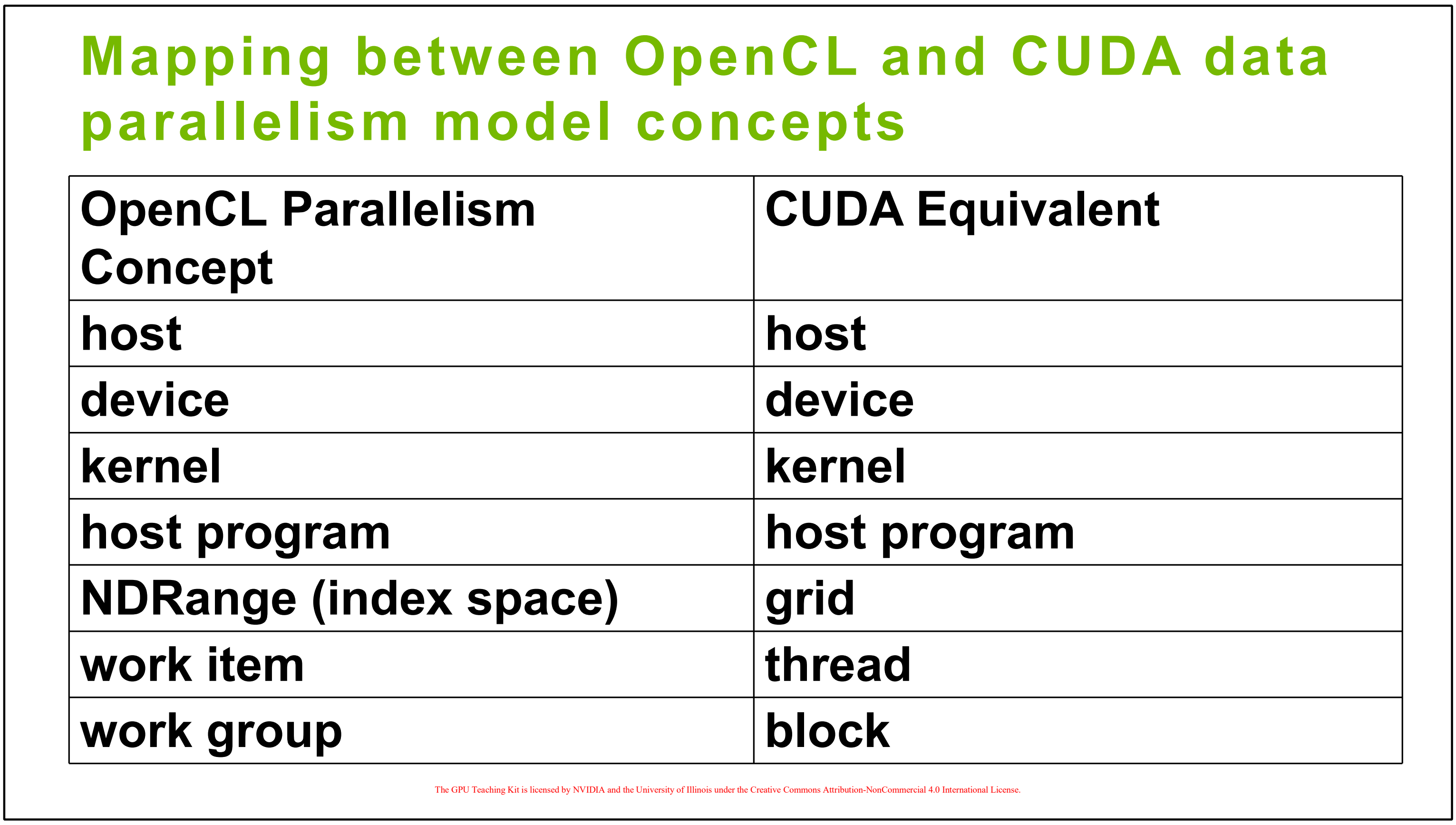
Kernel B

Kernel C

OpenCL Execution Model

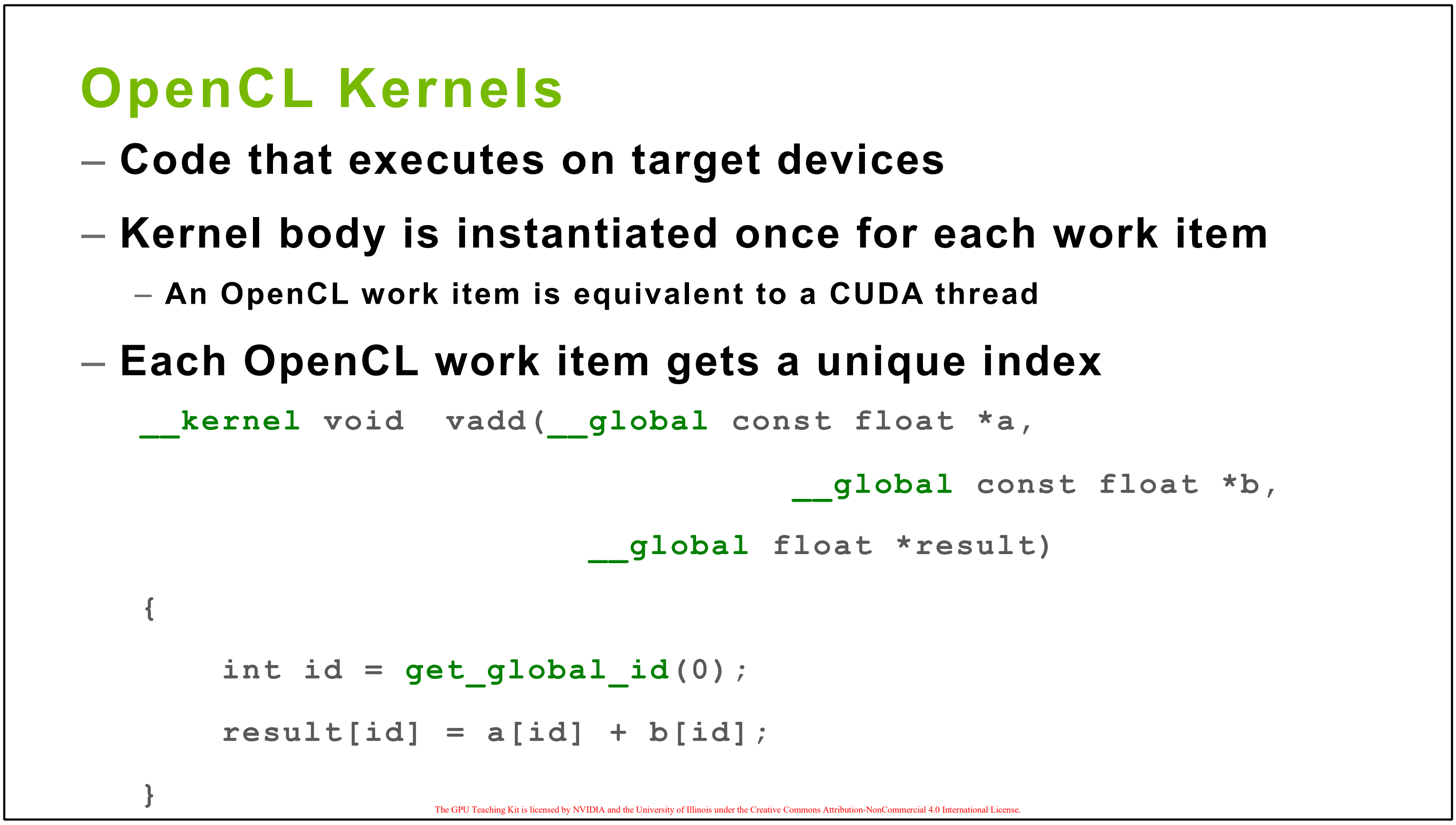
- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code





Mapping between OpenCL and CUDA data parallelism model concepts

OpenCL Parallelism Concept	CUDA Equivalent
host	host
device	device
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block



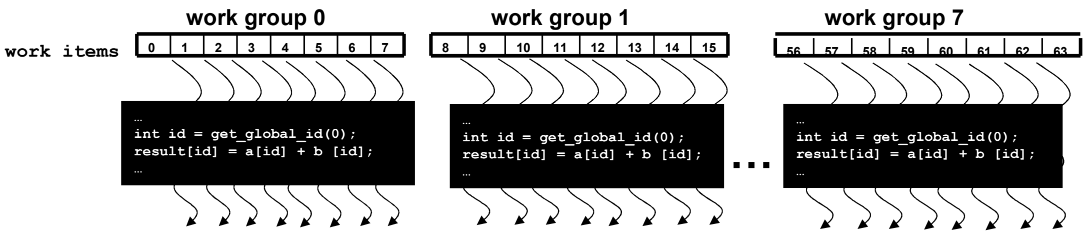
OpenCL Kernels

- Code that executes on target devices
- Kernel body is instantiated once for each work item
  - An OpenCL work item is equivalent to a CUDA thread
- Each OpenCL work item gets a unique index

```
__kernel void  vadd(__global const float *a,
                                __global const float *b,
                           __global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] + b[id];
}
```

Array of Work Items

- An OpenCL kernel is executed by an array of work items
  - All work items run the same code (SPMD)
  - Each work item can call get_global_id() to get its index for computing memory addresses and make control decisions



```
…
int id = get_global_id(0);
result[id] = a[id] + b [id];
…
```

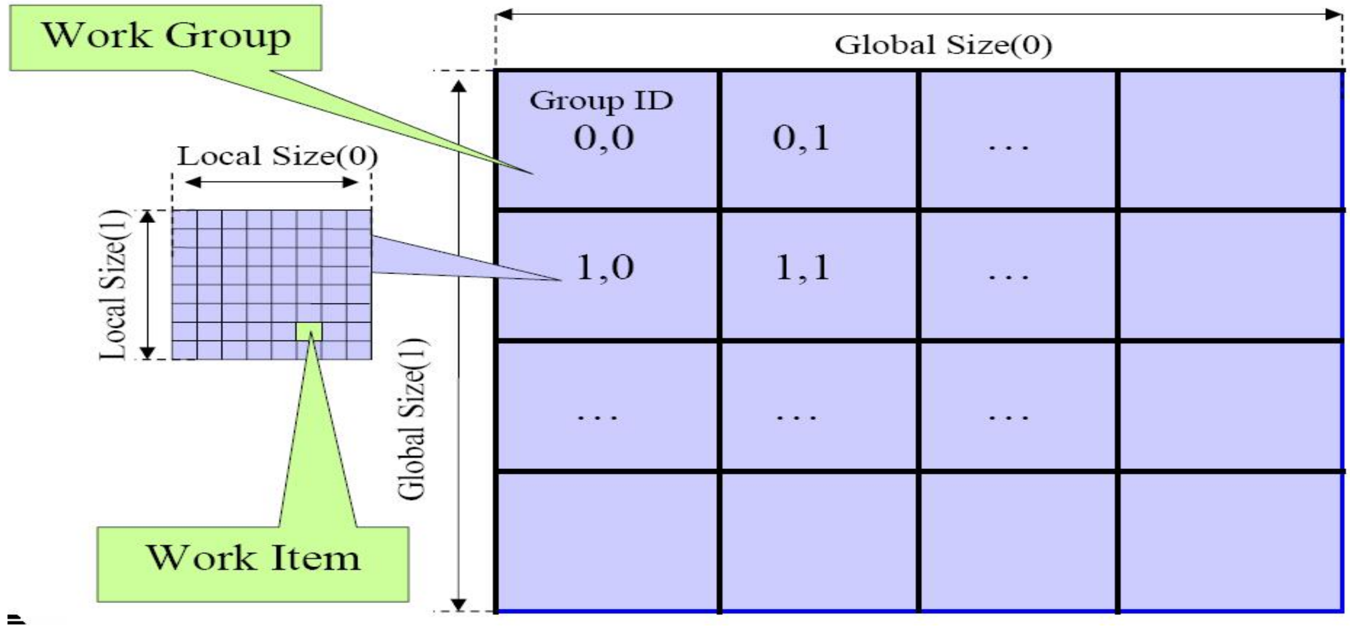
Work Groups: Scalable Cooperation

- Divide monolithic work item array into work groups
  - Work items within a work group cooperate via shared memory and barrier synchronization
  - Work items in different work groups cannot cooperate
- OpenCL counter part of CUDA Thread Blocks

OpenCL Dimensions and Indices

OpenCL API Call	Explanation	CUDA Equivalent
get_global_id(0);	global index of the work item in the x dimension	blockIdx.x*blockDim.x+threadIdx.x
get_local_id(0)	local index of the work item within the work group in the x dimension	threadIdx.x
get_global_size(0);	size of NDRange in the x dimension	gridDim.x*blockDim.x
get_local_size(0);	Size of each work group in the x dimension	blockDim.x

Multidimensional Work Indexing



OpenCL Data Parallel Model Summary

- Parallel work is submitted to devices by launching kernels
- Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”
- Work items executing within the same work group can synchronize with each other with barriers or memory fences
- Work items in different work groups can’t sync with each other, except by terminating the kernel

讲授内容：Related Programming Models: OpenCL

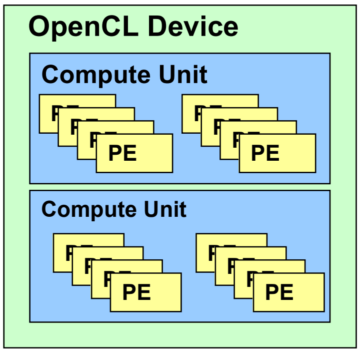
① OpenCL Data Parallelism Model

② OpenCL Device Architecture

③ OpenCL Host Code

OpenCL Hardware Abstraction

- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each device contains one or more “compute units”, i.e. cores, Streaming Multicprocessors, etc...
- Each compute unit contains one or more SIMD “processing elements”, (i.e. SP in CUDA)



OpenCL Device

Compute Unit

PE

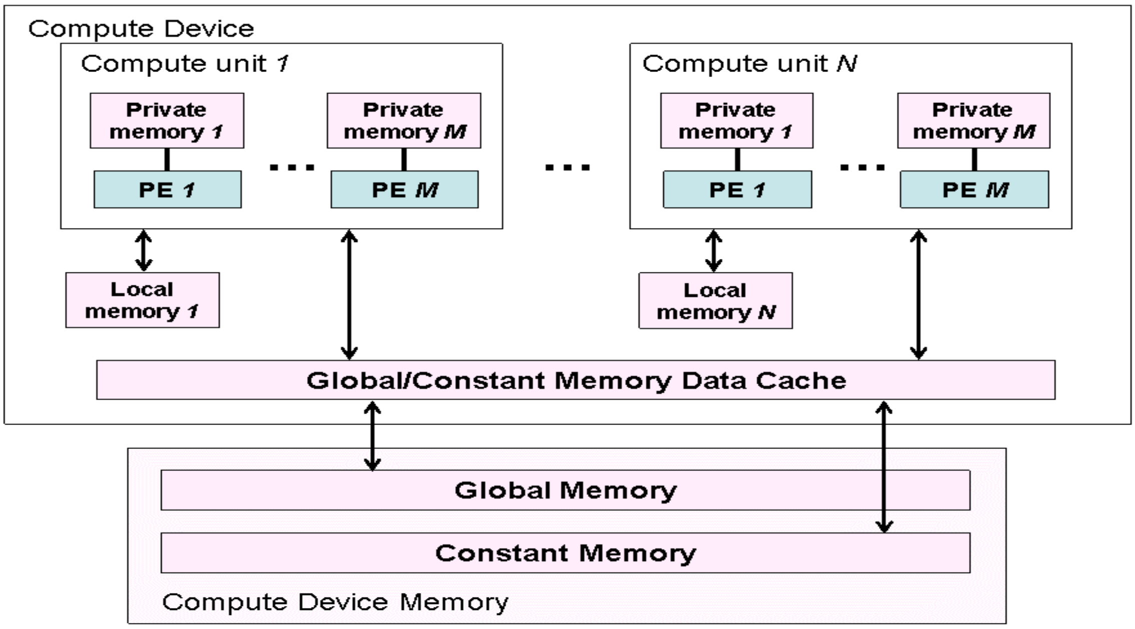
PE

Compute Unit

PE

PE

OpenCL Device Architecture



Compute Device

Compute unit *1*

Private memory *1*

PE *1*

...

Private memory *M*

PE *M*

...

Compute unit *N*

Private memory *1*

PE *1*

...

Private memory *M*

PE *M*

Local memory *1*

Local memory *N*

Global/Constant Memory Data Cache

Global Memory

Constant Memory

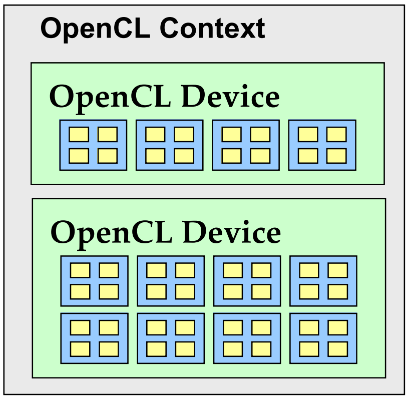
Compute Device Memory

OpenCL Device Memory Types

Memory Type	Host access	Device access	CUDA Equivalent
global memory	Dynamic allocation; Read/write access	No allocation; Read/write access by all work items in all work groups, large and slow but may be cached in some devices.	global memory
constant memory	Dynamic allocation; read/write access	Static allocation; read-only access by all work items.	constant memory
local memory	Dynamic allocation; no access	Static allocation; shared read-write access by all work items in a work group.	shared memory
private memory	No allocation; no access	Static allocation; Read/write access by a single work item.	registers and local memory

OpenCL Context

- Contains one or more devices
- OpenCL device memory objects are associated with a context, not a specific device



讲授内容：Related Programming Models: OpenCL

① OpenCL Data Parallelism Model

② OpenCL Device Architecture

③ OpenCL Host Code

OpenCL Context

- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- clCreateBuffer() is the main data object allocation function
  - error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory copy transfers are associated with a command queue (thus a specific device)

OpenCL Context Setup Code (simple)

```
cl_int clerr = CL_SUCCESS;

cl_context clctx = clCreateContextFromType(0,
CL_DEVICE_TYPE_ALL, NULL, NULL, &clerr);


size_t parmsz;

clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL,
&parmsz);


cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);

clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz,
cldevs, NULL);


cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0],
0, &clerr);
```

OpenCL Kernel Compilation: vadd

OpenCL kernel source code as a big string

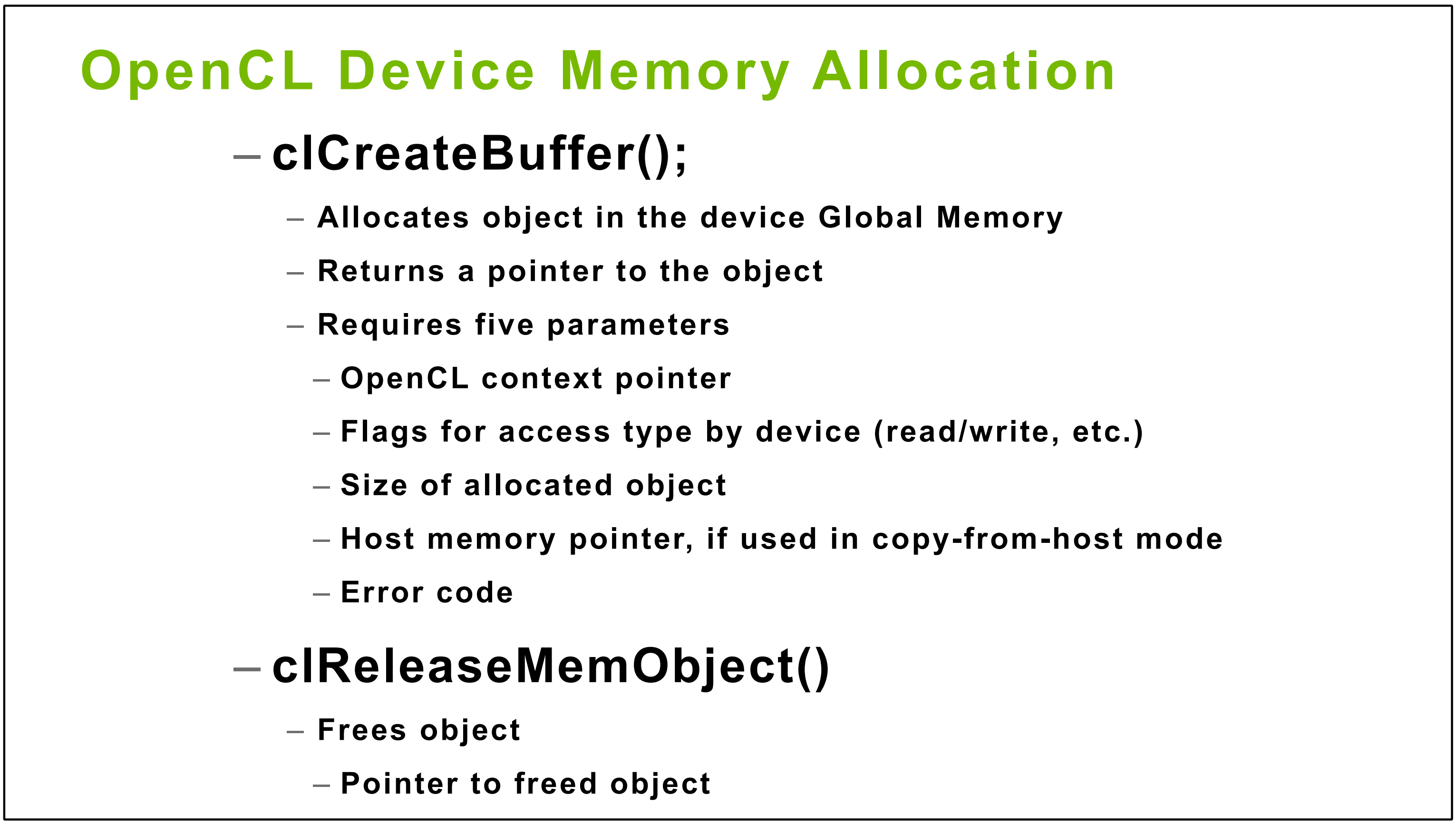
```
const char* vaddsrc =
   “__kernel void vadd(__global float *d_A, __global float *d_B,
__global float *d_C, int N) { \n“   […etc and so forth…]
```

Gives raw source code string(s) to OpenCL

```
cl_program clpgm;
clpgm = clCreateProgramWithSource(clctx, 1, &vaddsrc, NULL,
&clerr);
```

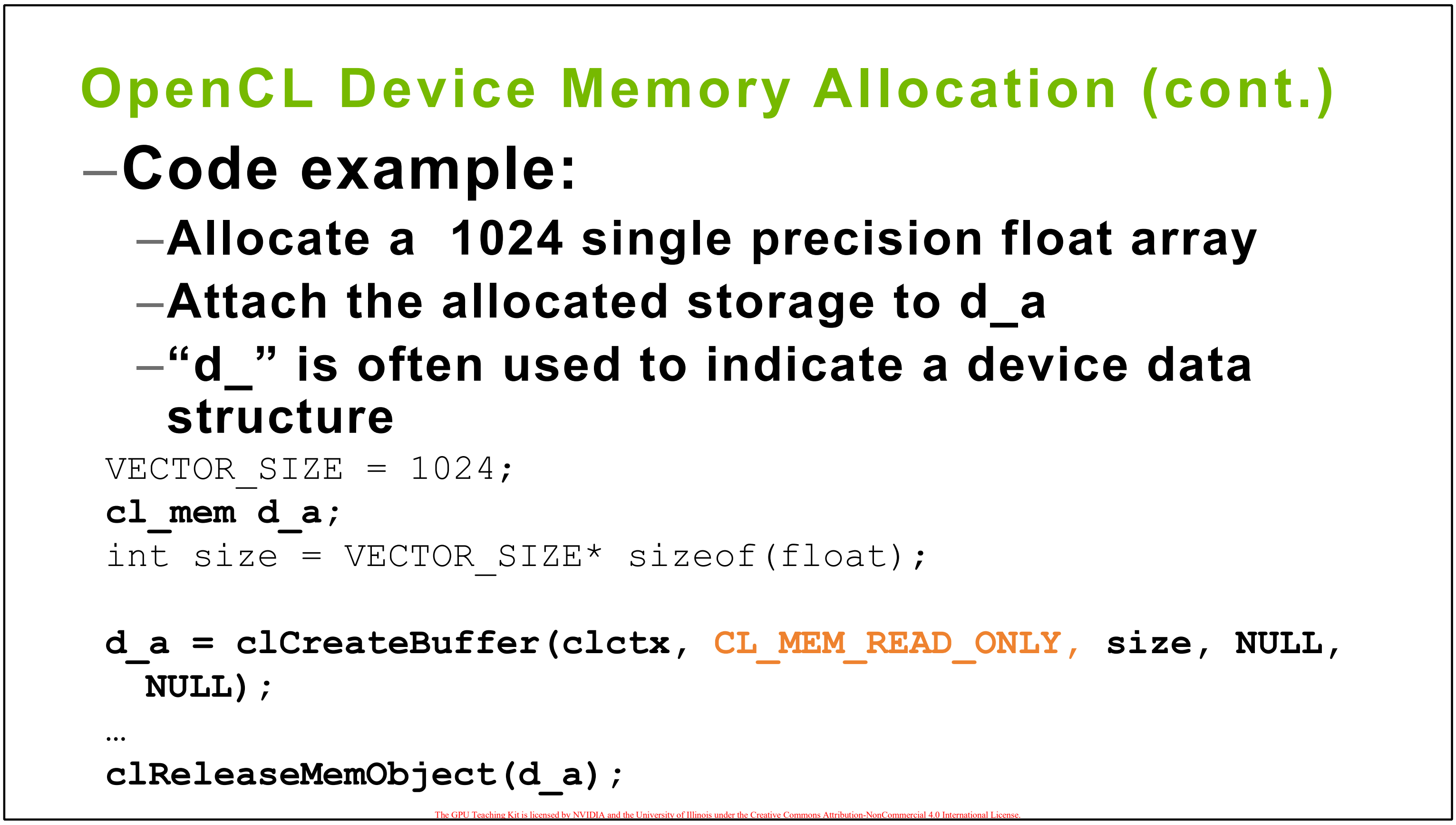
Set compiler flags, compile source, and retrieve a handle to the “vadd” kernel

```
char clcompileflags[4096];
sprintf(clcompileflags, “-cl-mad-enable”);
clerr = clBuildProgram(clpgm, 0, NULL, clcompileflags, NULL,
NULL);
cl_kernel clkern = clCreateKernel(clpgm, “vadd", &clerr);
```



OpenCL Device Memory Allocation

- clCreateBuffer();
  - Allocates object in the device Global Memory
  - Returns a pointer to the object
  - Requires five parameters
    - OpenCL context pointer
    - Flags for access type by device (read/write, etc.)
    - Size of allocated object
    - Host memory pointer, if used in copy-from-host mode
    - Error code
- clReleaseMemObject()
  - Frees object
    - Pointer to freed object

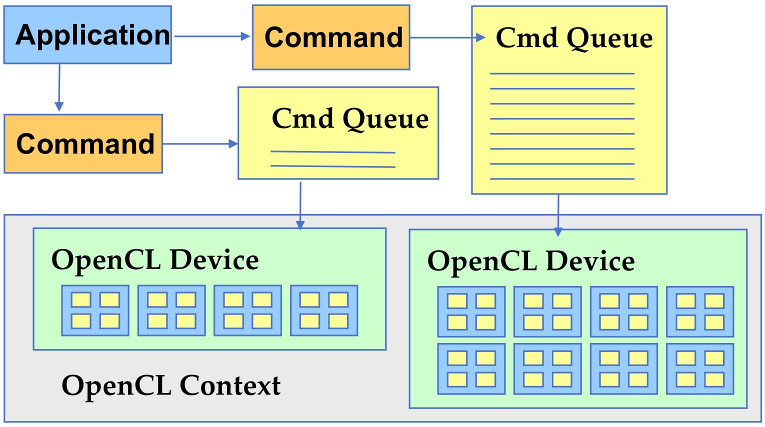


OpenCL Device Memory Allocation (cont.)

- **Code example:**
  - Allocate a 1024 single precision float array
  - Attach the allocated storage to d_a
  - “d_” is often used to indicate a device data structure

```
VECTOR_SIZE = 1024;
cl_mem d_a;
int size = VECTOR_SIZE* sizeof(float);

d_a = clCreateBuffer(clctx, CL_MEM_READ_ONLY, size, NULL,
  NULL);
…
clReleaseMemObject(d_a);
```

OpenCL Device Command Execution



OpenCL Host-to-Device Data Transfer

- `clEnqueueWriteBuffer();`
  - Memory data transfer to device
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size (in bytes) of written data
    - Source host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command

OpenCL Device-to-Host Data Transfer

- `clEnqueueReadBuffer();`
  - Memory data transfer to host
  - requires nine parameters
    - OpenCL command queue pointer
    - Source OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size of bytes of read data
    - Destination host memory pointer
    - List of events to be completed before execution of this command
    - Event object tied to this command

OpenCL Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - a is in host memory and d_a is in device memory

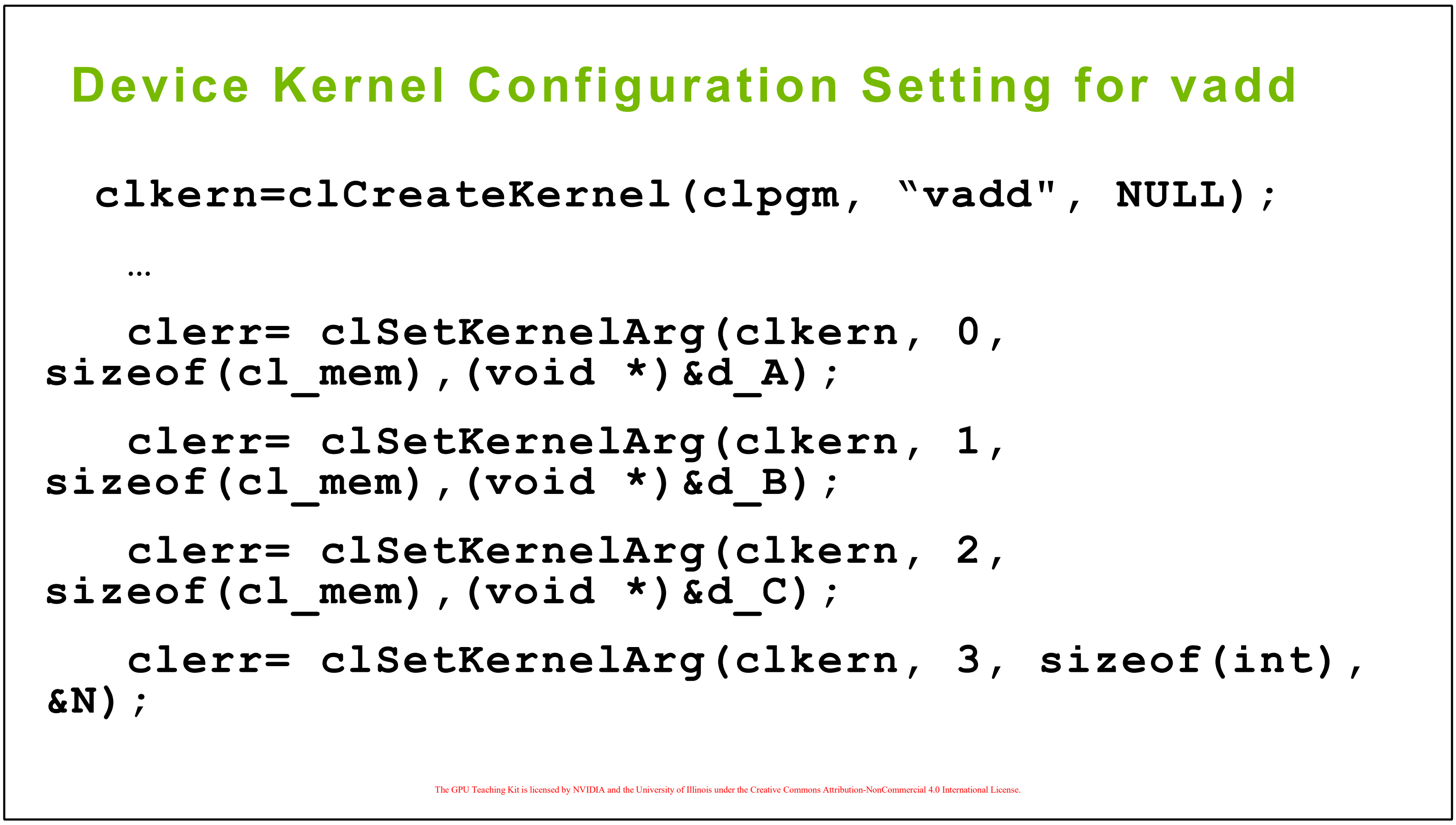
```
clEnqueueWriteBuffer(clcmdq, d_a, CL_FALSE, 0,
       mem_size, (const void * )a, 0, 0, NULL);

clEnqueueReadBuffer(clcmdq, d_result, CL_FALSE, 0,
       mem_size, (void * ) host_result, 0, 0, NULL);
```

OpenCL Host-Device Data Transfer (cont.)

- `clCreateBuffer` and `clEnqueueWriteBuffer` can be combined into a single command using special flags.
- Eg:
- `d_A=clCreateBuffer(clctxt,`
- `CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, mem_size, h_A, NULL);`
  - Combination of 2 flags here. `CL_MEM_COPY_HOST_PTR` to be used only if a valid host pointer is specified.
  - This creates a memory buffer on the device, and copies data from h_A into d_A.
  - Includes an implicit `clEnqueueWriteBuffer` operation, for all devices/command queues tied to the context clctxt.

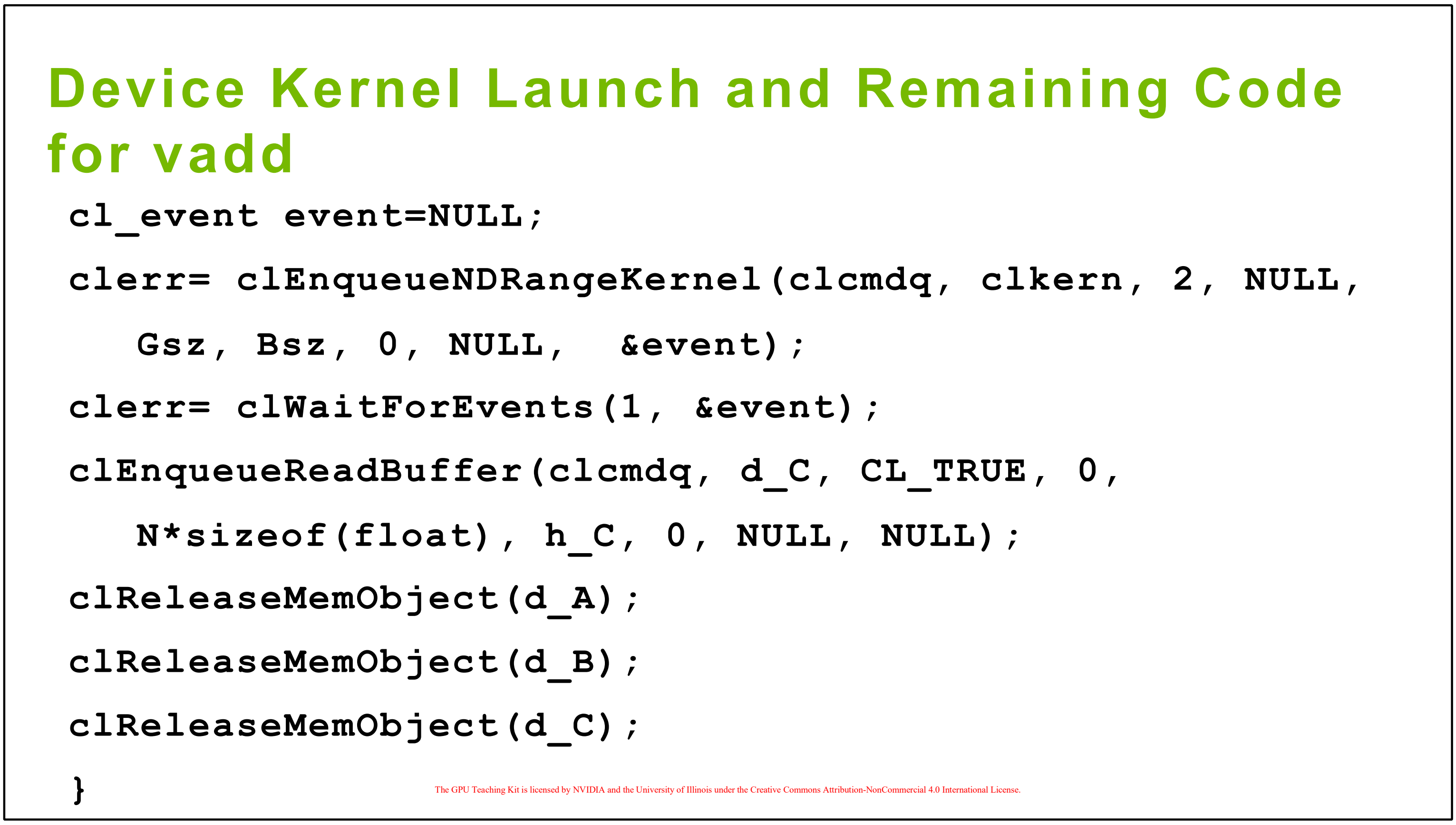
Device Memory Allocation and Data Transfer for vadd

```
float *h_A = …,   *h_B = …;
    // allocate device (GPU) memory
   cl_mem d_A, d_B, d_C;
   d_A = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_A,
NULL);
   d_B = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_B,
NULL);
   d_C = clCreateBuffer(clctx, CL_MEM_WRITE_ONLY,
 N *sizeof(float), NULL, NULL);
```



Device Kernel Configuration Setting for vadd

```
 clkern=clCreateKernel(clpgm, “vadd", NULL);
   …
   clerr= clSetKernelArg(clkern, 0,
sizeof(cl_mem),(void *)&d_A);
   clerr= clSetKernelArg(clkern, 1,
sizeof(cl_mem),(void *)&d_B);
   clerr= clSetKernelArg(clkern, 2,
sizeof(cl_mem),(void *)&d_C);
   clerr= clSetKernelArg(clkern, 3, sizeof(int),
&N);
```



Device Kernel Launch and Remaining Code for vadd

```
cl_event event=NULL;
clerr= clEnqueueNDRangeKernel(clcmdq, clkern, 2, NULL,
   Gsz, Bsz, 0, NULL,  &event);
clerr= clWaitForEvents(1, &event);
clEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0,
   N*sizeof(float), h_C, 0, NULL, NULL);
clReleaseMemObject(d_A);
clReleaseMemObject(d_B);
clReleaseMemObject(d_C);
}
```

讲授内容

- ➢Related Programming Models: OpenCL
- ➢Related Programming Models: OpenACC
- ➢Multi-GPU: OpenMP
- ➢Related Programming Models: MPI

讲授内容：Related Programming Models: OpenACC

① Introduction to OpenACC

② OpenACC Subtleties

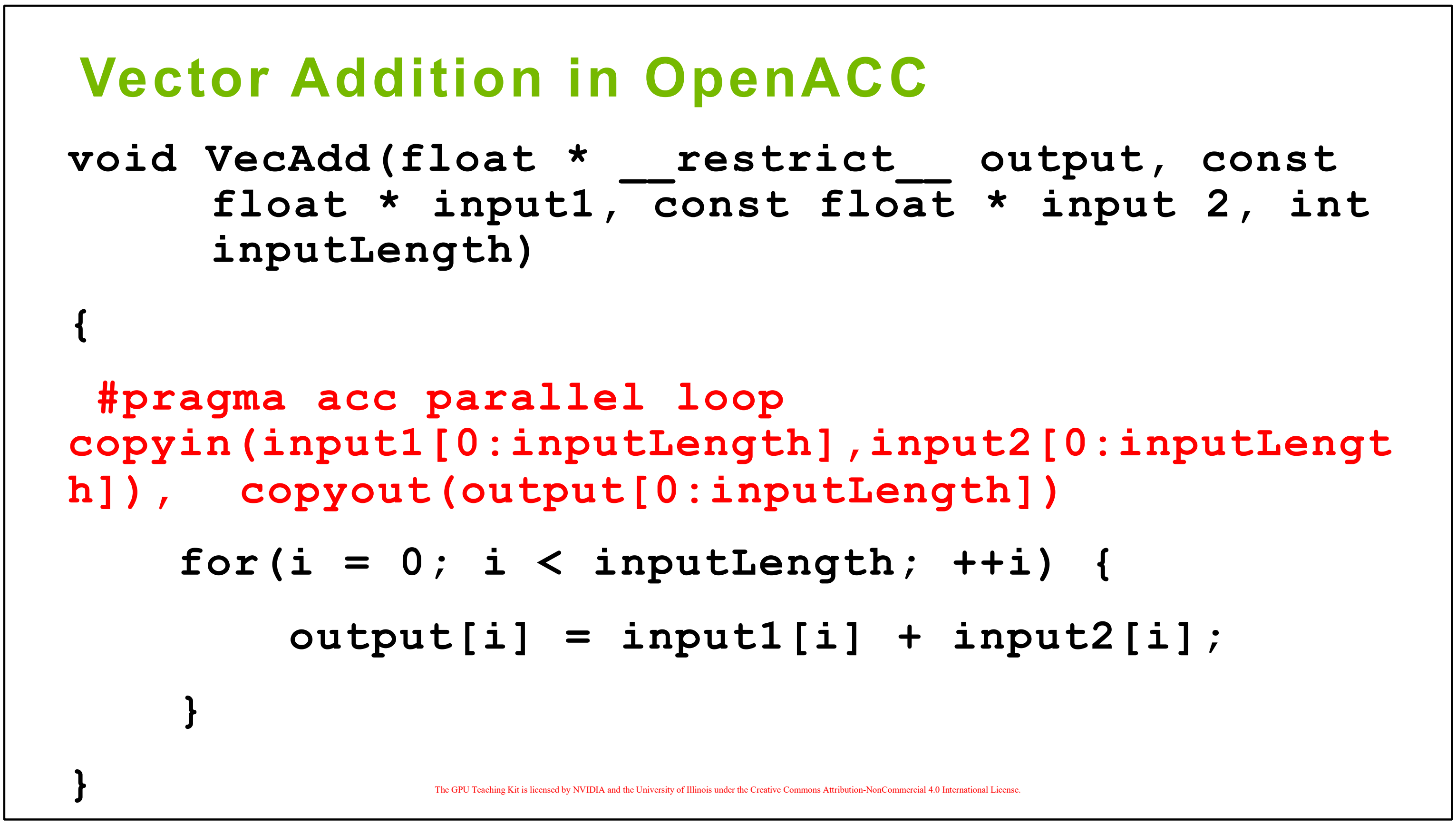
OpenACC

- The OpenACC Application Programming Interface provides a set of
  - compiler directives (pragmas)
  - library routines and
  - environment variables

that can be used to write data parallel Fortran, C and C++ programs that run on accelerator devices including GPUs and CPUs

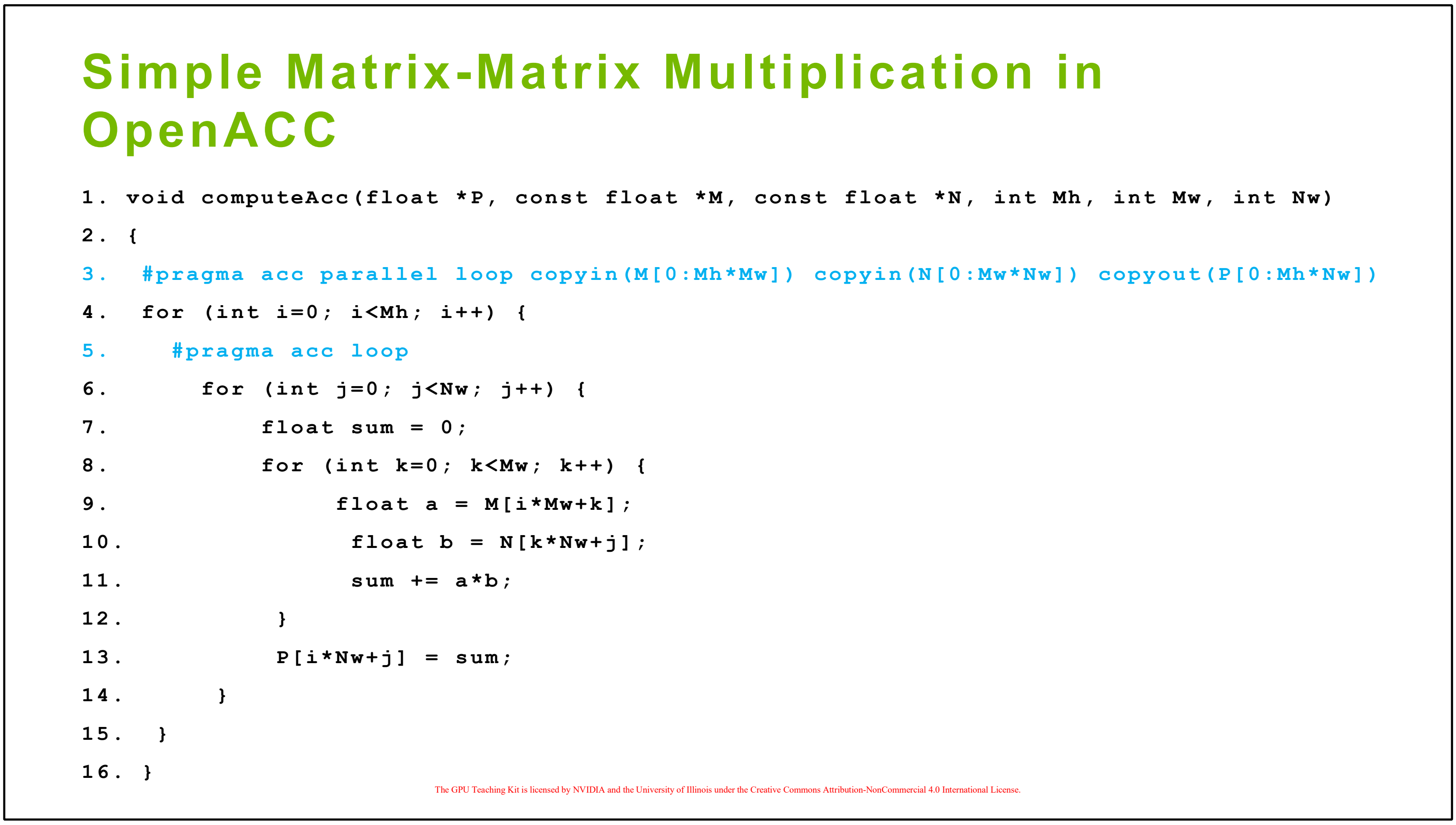
OpenACC Pragmas

- In C and C++, the #pragma directive is the method to provide to the compiler information that is not specified in the standard language.
  - These pragmas extend the base language



Vector Addition in OpenACC

```
void VecAdd(float * __restrict__ output, const
      float * input1, const float * input 2, int
      inputLength)
{
 #pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLengt
h]),  copyout(output[0:inputLength])
    for(i = 0; i < inputLength; ++i) {
        output[i] = input1[i] + input2[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:Mw*Nw]) 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;
      }
  }
}
```

Some Observations (1)

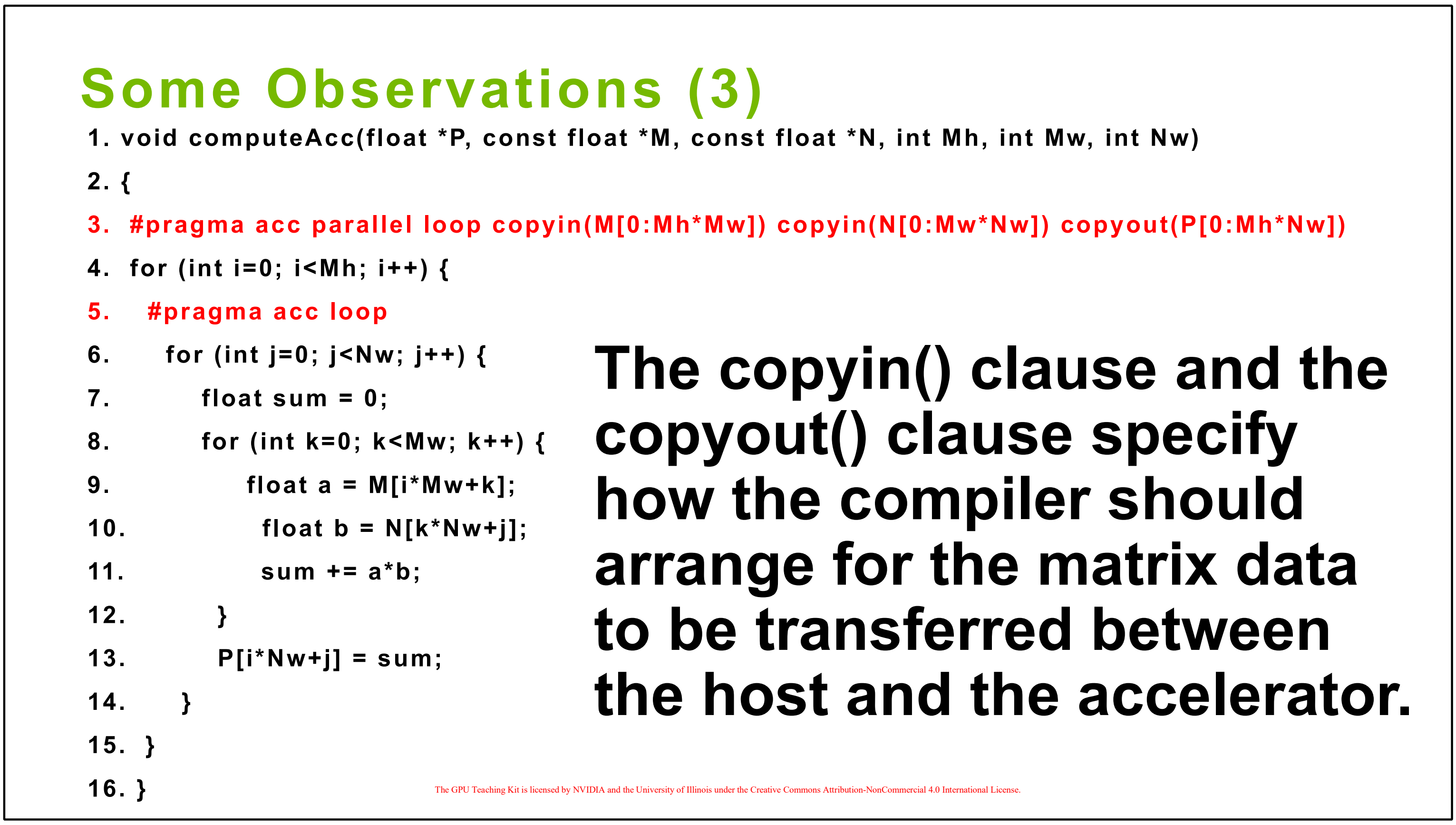
```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.     for (int j=0; j<Nw; j++) {
7.         float sum = 0;
8.  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

The code is almost identical to the sequential version, except for the two lines with #pragma at line 3 and line 5.

Some Observations (2)

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

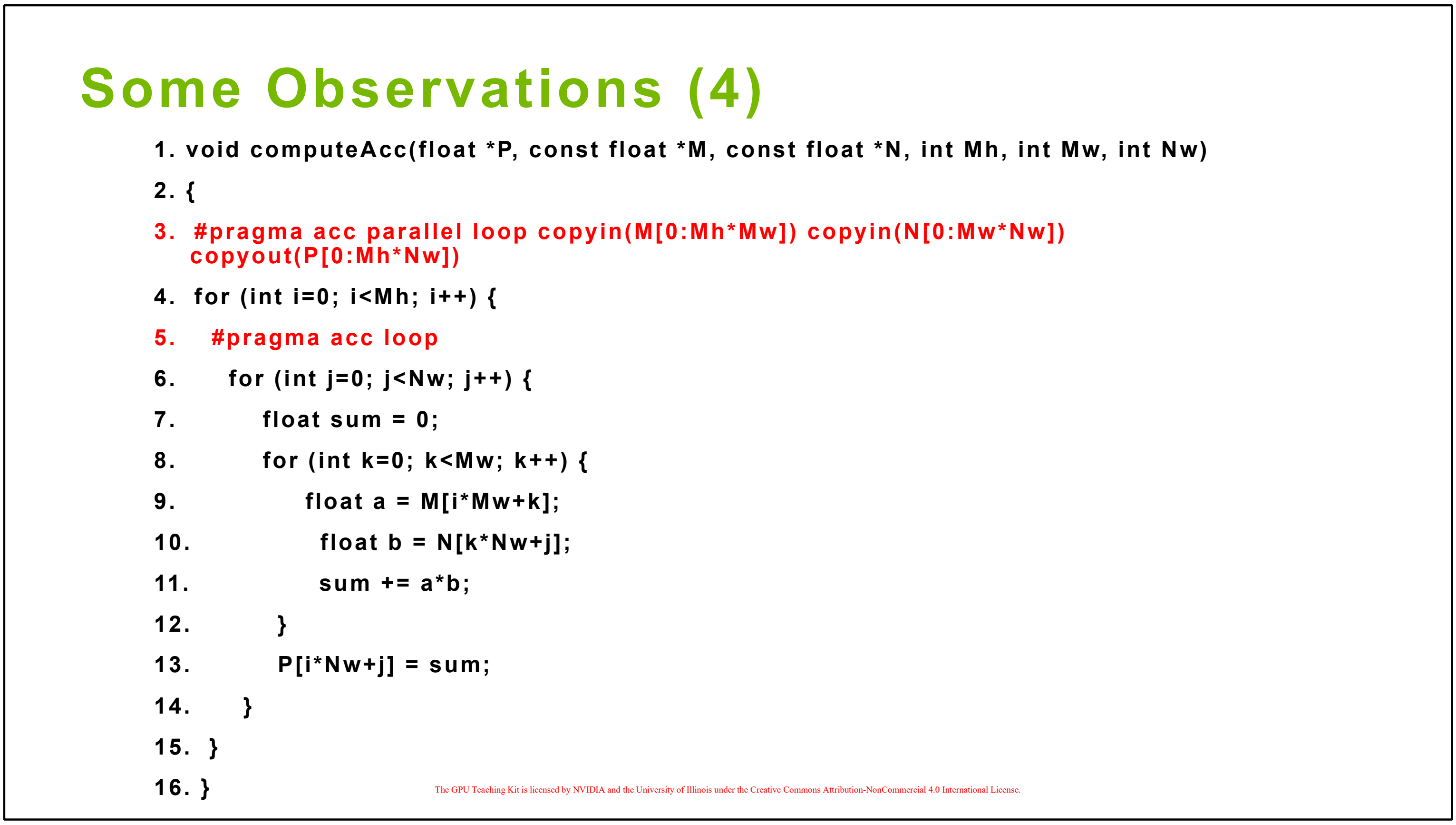
The #pragma at line 3 tells the compiler to generate code for the 'i' loop at line 4 through 15 so that the loop iterations are executed at the first level of parallelism on the accelerator.



Some Observations (3)

```
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:Mw*Nw]) 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;
    }
 }
}
```

The copyin() clause and the copyout() clause specify how the compiler should arrange for the matrix data to be transferred between the host and the accelerator.



Some Observations (4)

```
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:Mw*Nw])
 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;
    }
 }
}
```

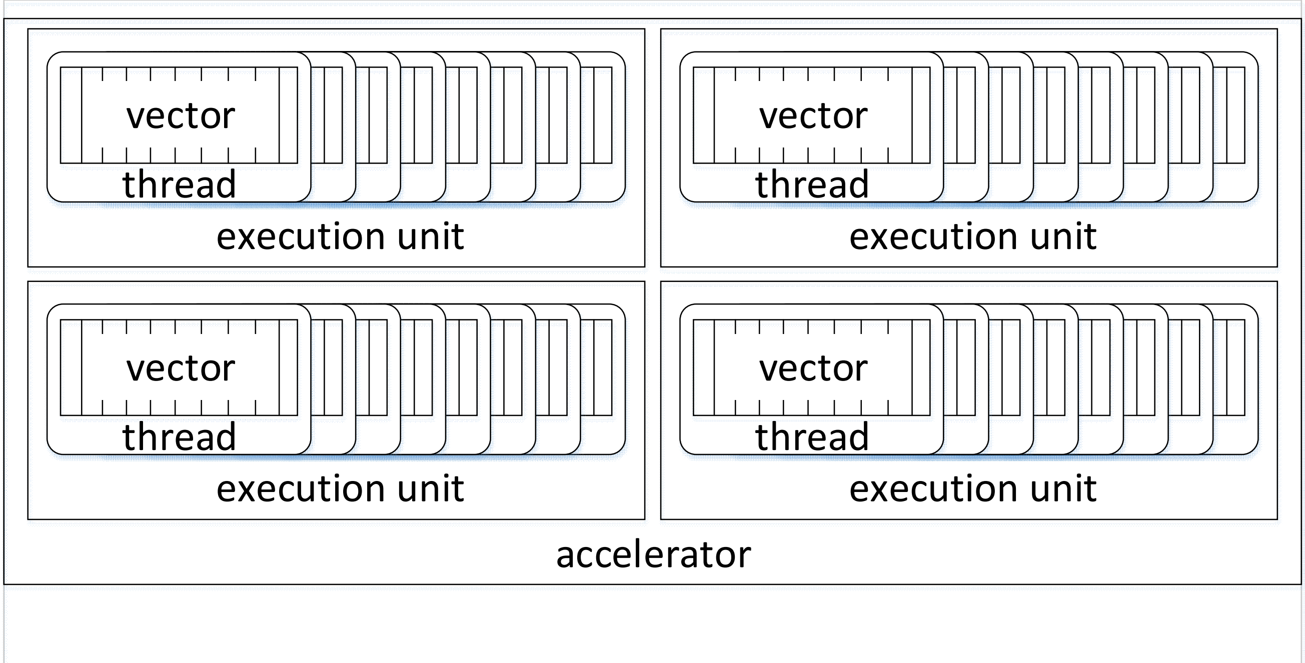
Motivation

- 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, memory allocation, and data transfers to the OpenACC compiler.
- OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

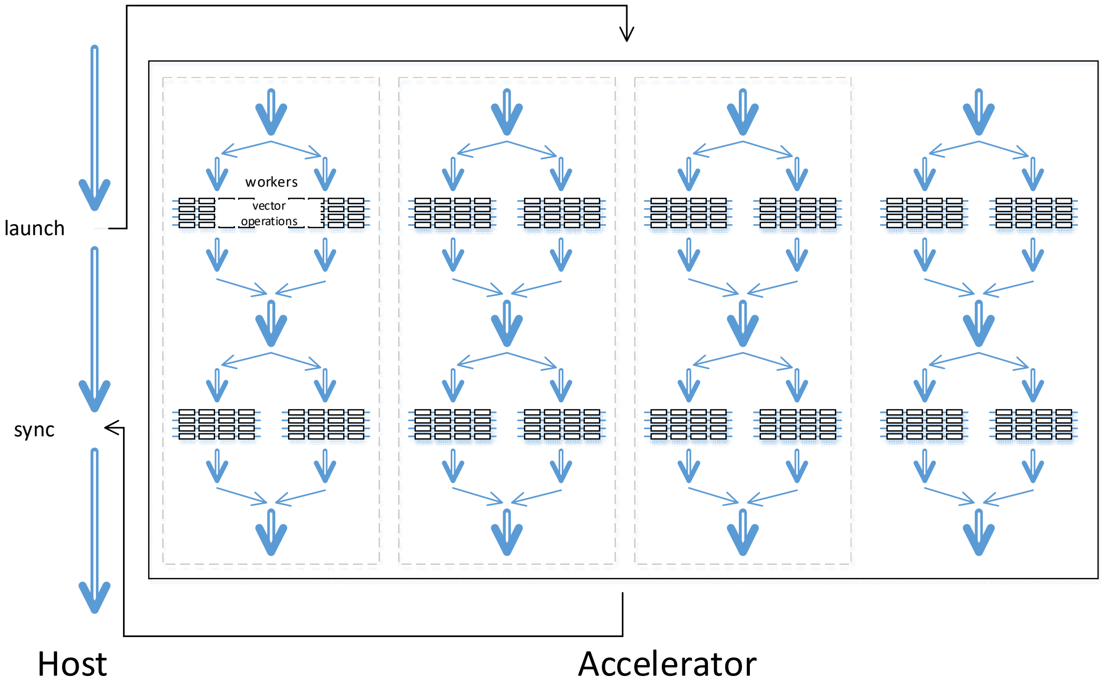
Frequently Encountered Issues

- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC program depends heavily on the quality of the compiler.
  - It may be hard to figure out why the compiler cannot act according to your hints
  - The uncertainty is much less so for CUDA or OpenCL programs

OpenACC Device Model



OpenACC Execution Model



讲授内容：Related Programming Models: OpenACC

① Introduction to OpenACC

② OpenACC Subtleties

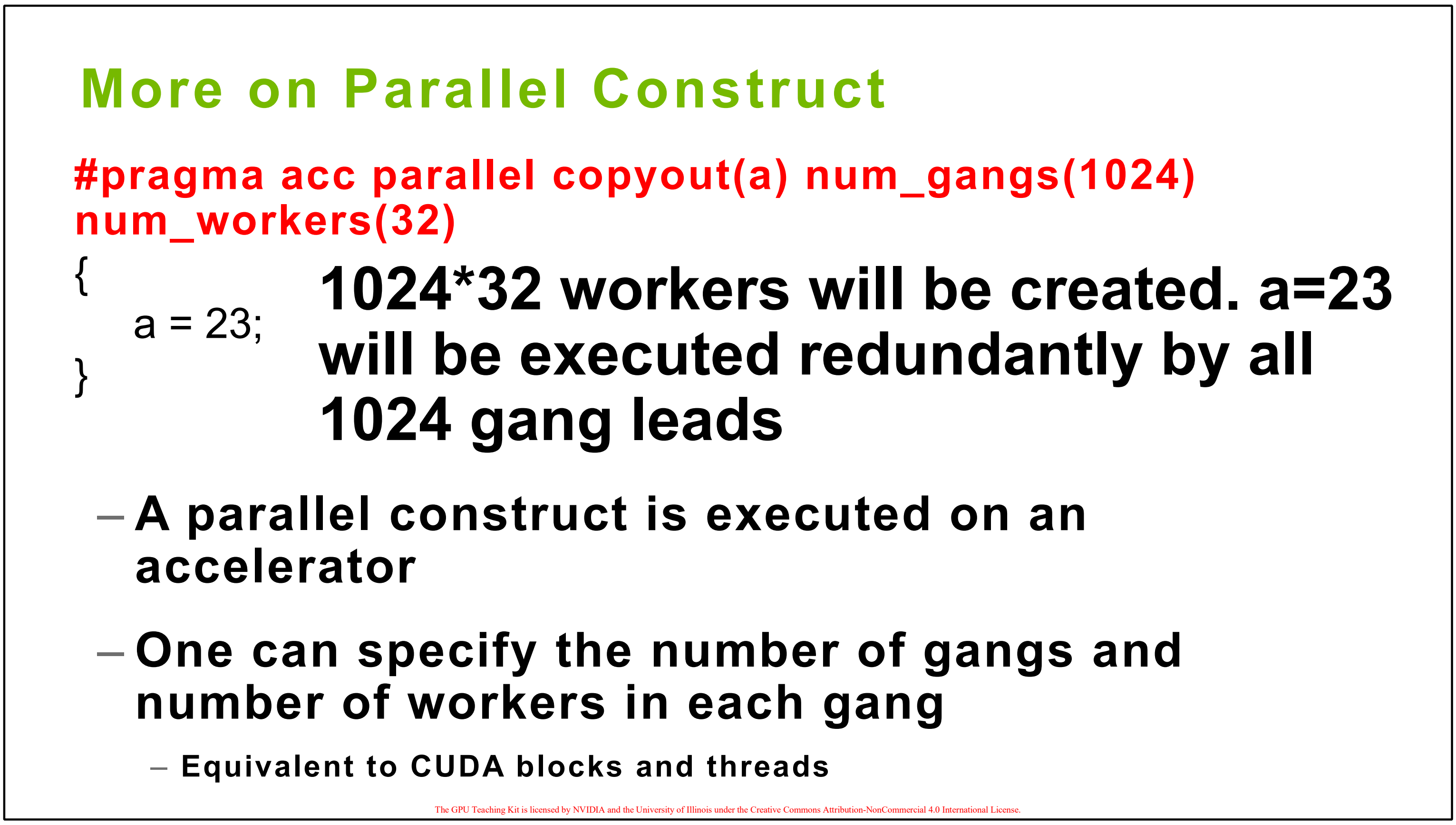
Parallel vs. Loop Constructs

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

(a parallel region that consists of a single loop)

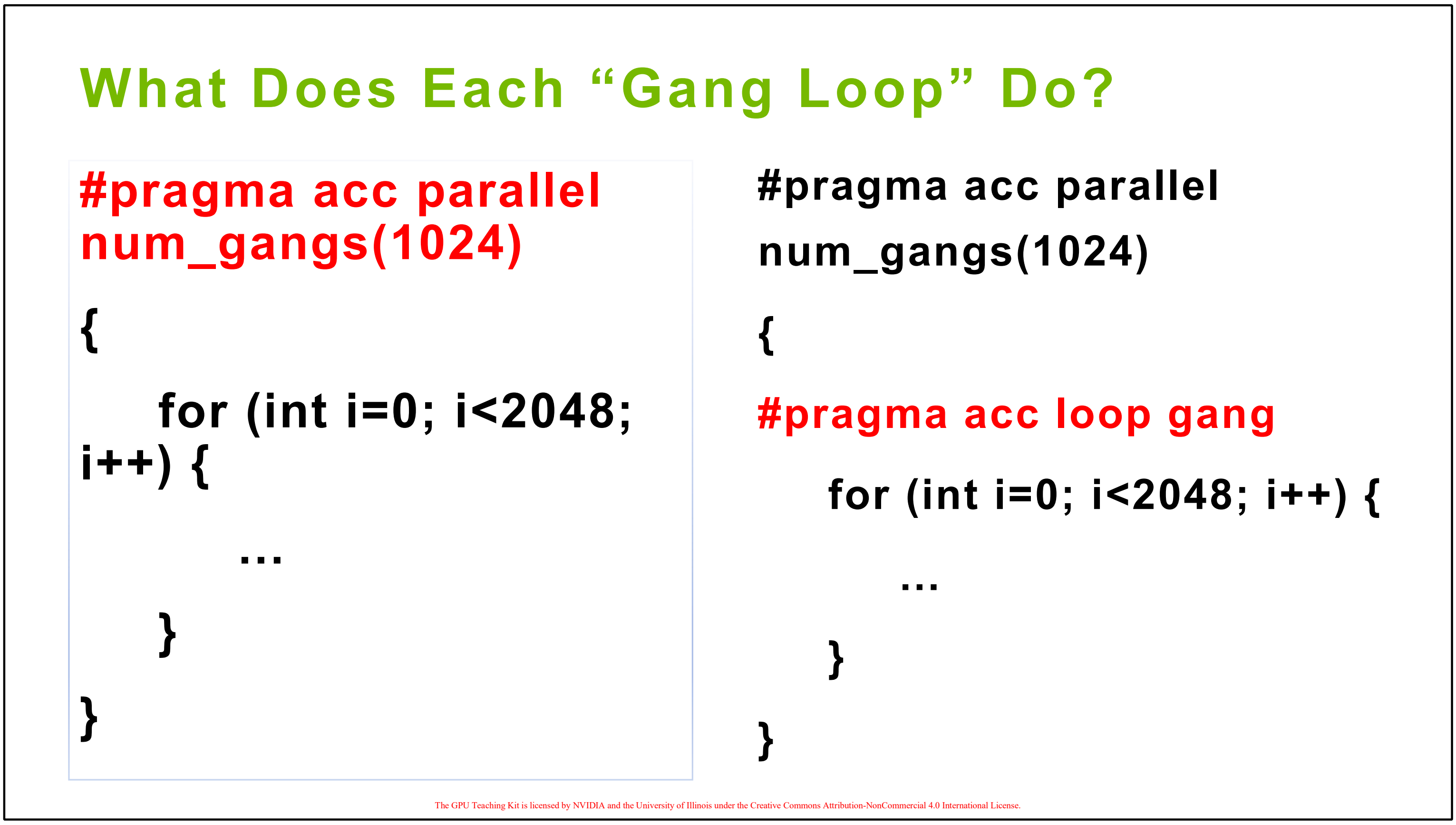


More on Parallel Construct

```
#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

- A parallel construct is executed on an accelerator
- One can specify the number of gangs and number of workers in each gang
  - Equivalent to CUDA blocks and threads



What Does Each "Gang Loop" Do?

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

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

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 Substantial Example

- Statements 1, 3, 5, 6 are redundantly executed by 32 gangs

```
#pragma acc parallel num_gangs(32)
{
   Statement 1;
   #pragma acc loop gang
   for (int i=0; i<n; i++) {
      Statement 2;
   }
。。。
}
```

```
#pragma acc parallel num_gangs(32)
{
   。。。
   Statement 3;
   #pragma acc loop gang
   for (int i=0; i<m; i++) {
      Statement 4;
   }
   Statement 5;
   if (condition) Statement 6;
}
```

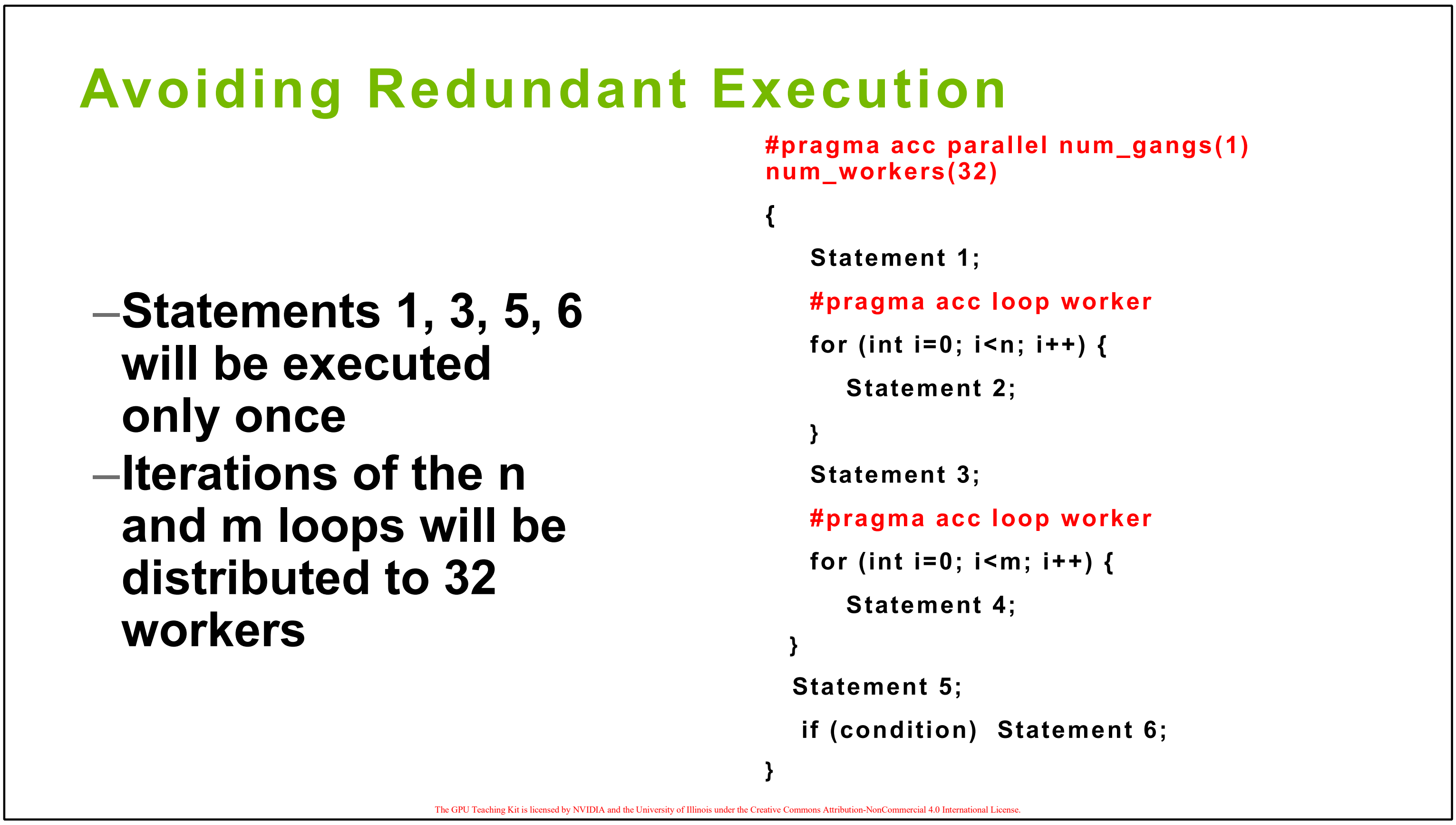
A More Substantial Example

- The iterations of the n and m for-loop iterations are distributed to 32 gangs
- Each gang could further distribute the iterations to its workers
  - The number of workers in each gang will be determined by the compiler/runtime

A More Substantial Example

```
#pragma acc parallel
num_gangs(32)
{
    Statement 1;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
        Statement 2;
    }
。 。 。
}
```

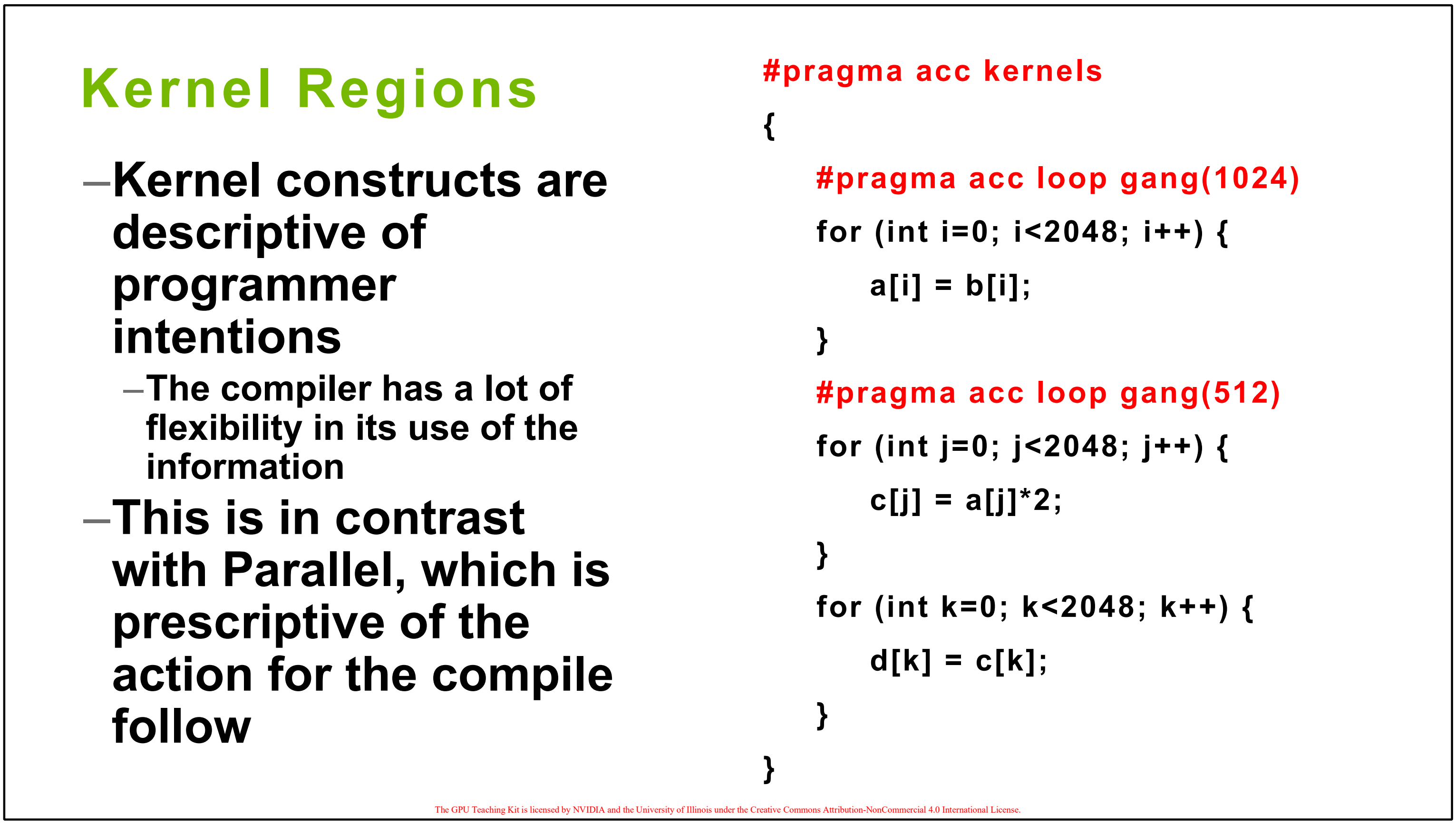
```
#pragma acc parallel num_gangs(32)
{
。 。 。
    Statement 3;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
        Statement 4;
    }
    Statement 5;
    if (condition) Statement 6;
}
```



Avoiding Redundant Execution

- Statements 1, 3, 5, 6 will be executed only once
- Iterations of the n and m loops will be distributed to 32 workers

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



Kernel Regions

- Kernel constructs are descriptive of programmer intentions
  - The compiler has a lot of flexibility in its use of the information
- This is in contrast with Parallel, which is prescriptive of the action for the compile follow

```
#pragma acc kernels
{
    #pragma acc loop gang(1024)
    for (int i=0; i<2048; i++) {
        a[i] = b[i];
    }
    #pragma acc loop gang(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 Regions

- Code in a kernel region can be broken into multiple CUDA/OpenCL kernels
- The i, j, k loops can each become a kernel
  - The k-loop may even remain as host code
- Each kernel can have a different gang/worker configuration

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

讲授内容

- ➢Related Programming Models: OpenCL
- ➢Related Programming Models: OpenACC
- ➢Multi-GPU: OpenMP
- ➢Related Programming Models: MPI

讲授内容：Multi-GPU: OpenMP

① OpenMP

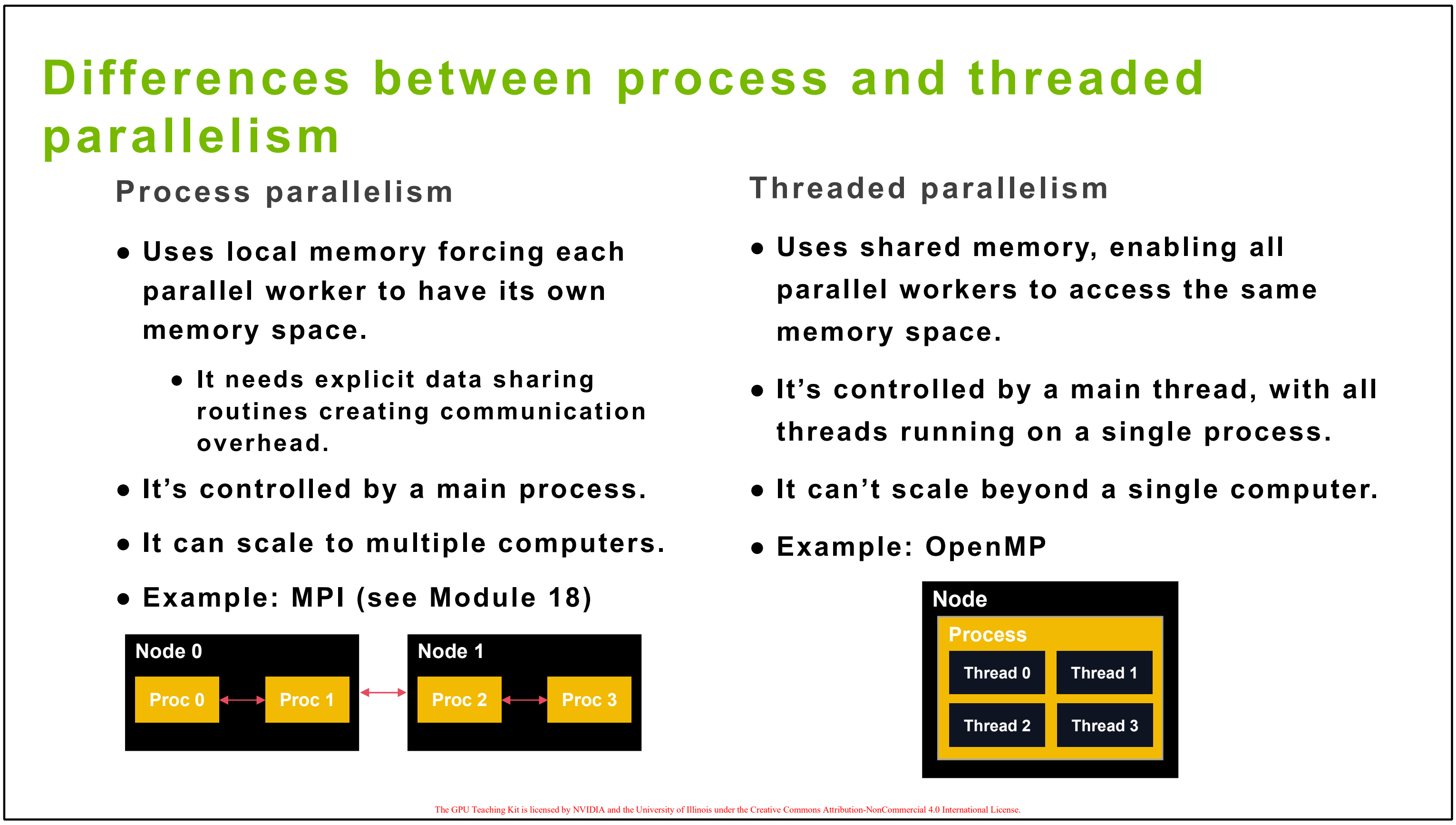
② Multi GPU Introduction I

③ Multi GPU Introduction II

④ Multi GPU patterns with OpenMP & Cooperative Groups

OpenMP

- A parallel programming model based on the concepts of multithreading and shared memory
- OpenMP programs consists on annotations written into the serial code, called directives
- Needs a compiler with OpenMP support, which will translate the directives to the actual parallel code
- Highly portable across systems
- Limited to a single computer, however it can handle several processors



Differences between process and threaded parallelism

Process parallelism

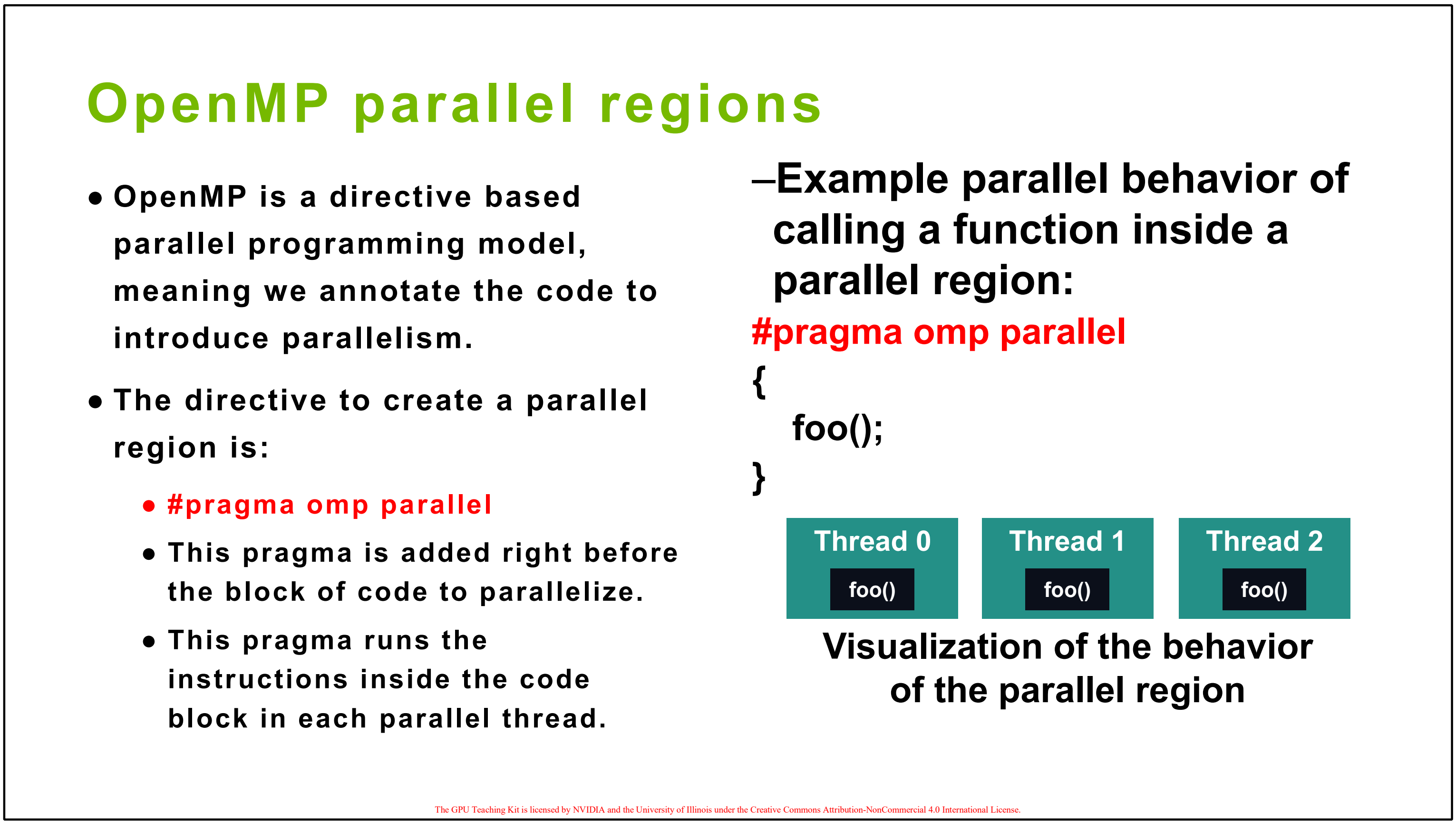
- Uses local memory forcing each parallel worker to have its own memory space.
  - It needs explicit data sharing routines creating communication overhead.
- It's controlled by a main process.
- It can scale to multiple computers.
- Example: MPI (see Module 18)

Node 0
Proc 0 ↔ Proc 1
↔
Node 1
Proc 2 ↔ Proc 3

Threaded parallelism

- Uses shared memory, enabling all parallel workers to access the same memory space.
- It's controlled by a main thread, with all threads running on a single process.
- It can't scale beyond a single computer.
- Example: OpenMP

Node
Process
Thread 0 Thread 1
Thread 2 Thread 3



OpenMP parallel regions

- OpenMP is a directive based parallel programming model, meaning we annotate the code to introduce parallelism.
- The directive to create a parallel region is:
  - #pragma omp parallel
  - This pragma is added right before the block of code to parallelize.
  - This pragma runs the instructions inside the code block in each parallel thread.

–Example parallel behavior of calling a function inside a parallel region:

```
#pragma omp parallel
{
    foo();
}
```

Thread 0 foo()
Thread 1 foo()
Thread 2 foo()

Visualization of the behavior of the parallel region

Parallelizing a “for” loop with OpenMP

- OpenMP is aware that in many cases “for” loops are a source for introducing parallelism, that is why it has special directives for loops
- To parallelize a for loop exists the directive:
  - #pragma omp parallel for
  - This pragma is added right before the for loop to parallelize.
  - An OpenMP compiler will identify the pragma and create a program where the loop is executed in parallel.

–Serial version of the vector sum algorithm:

```
for(int i = 0; i < n; ++i)  {
  c[i] = a[i] + b[i];
}
```

–Parallel OpenMP version of the vector sum algorithm:

```
#pragma omp parallel for
for(int i = 0; i < n; ++i)  {
  c[i] = a[i] + b[i];
}
```

#pragma omp parallel for

- Each variable used inside the loop has a kind determining if it’s accessible by other threads: shared, private, firstprivate, lastprivate
- Shared variables are accessible by every thread in the region
- Private variables are local to each thread and are not initialized
- Firstprivate behave similarly to private variables but are initialized with the value before the parallel region
- Unless otherwise stated, every variable outside the loop will be considered shared and every inside the loop will be considered private

OpenMP Synchronization

- By default every parallel region will wait until the region is finished before executing other instructions. To change this behavior use the nowait option when declaring the parallel region
- #pragma omp barrier specifies a synchronization point equivalent to __syncthreads() in CUDA, the pragma is not associated to a block of code

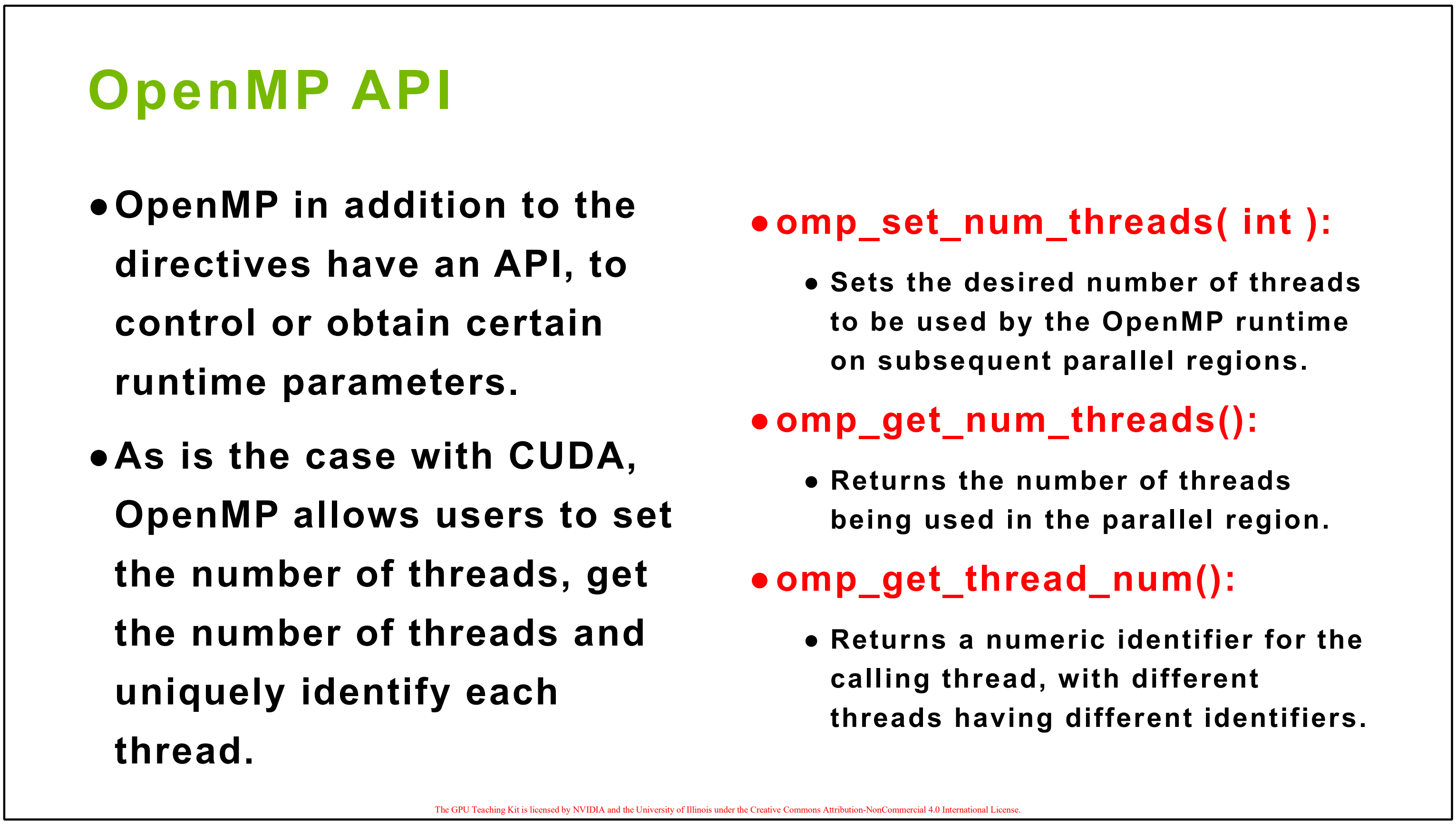
```
#pragma omp parallel for
for(int i = 0; i < n; ++i) {
  c[i] = a[i] + b[i];
  ...
#pragma omp barrier
  c[i] += i >= 1 ? c[i - 1] : 0;
}
```

OpenMP Sections

- Critical sections:
  - Are executed by only one thread at a time
  - Are associated to a block of code
  - #pragma omp critical
- Single sections:
  - Are executed by only one thread of the team
  - Are associated to a block of code
  - #pragma omp single

```
int a = 0;
int b = 0;
#pragma omp parallel num_threads(4)
{
#pragma omp critical
  a += 1 ;
#pragma omp single
  b += 1;
}
```

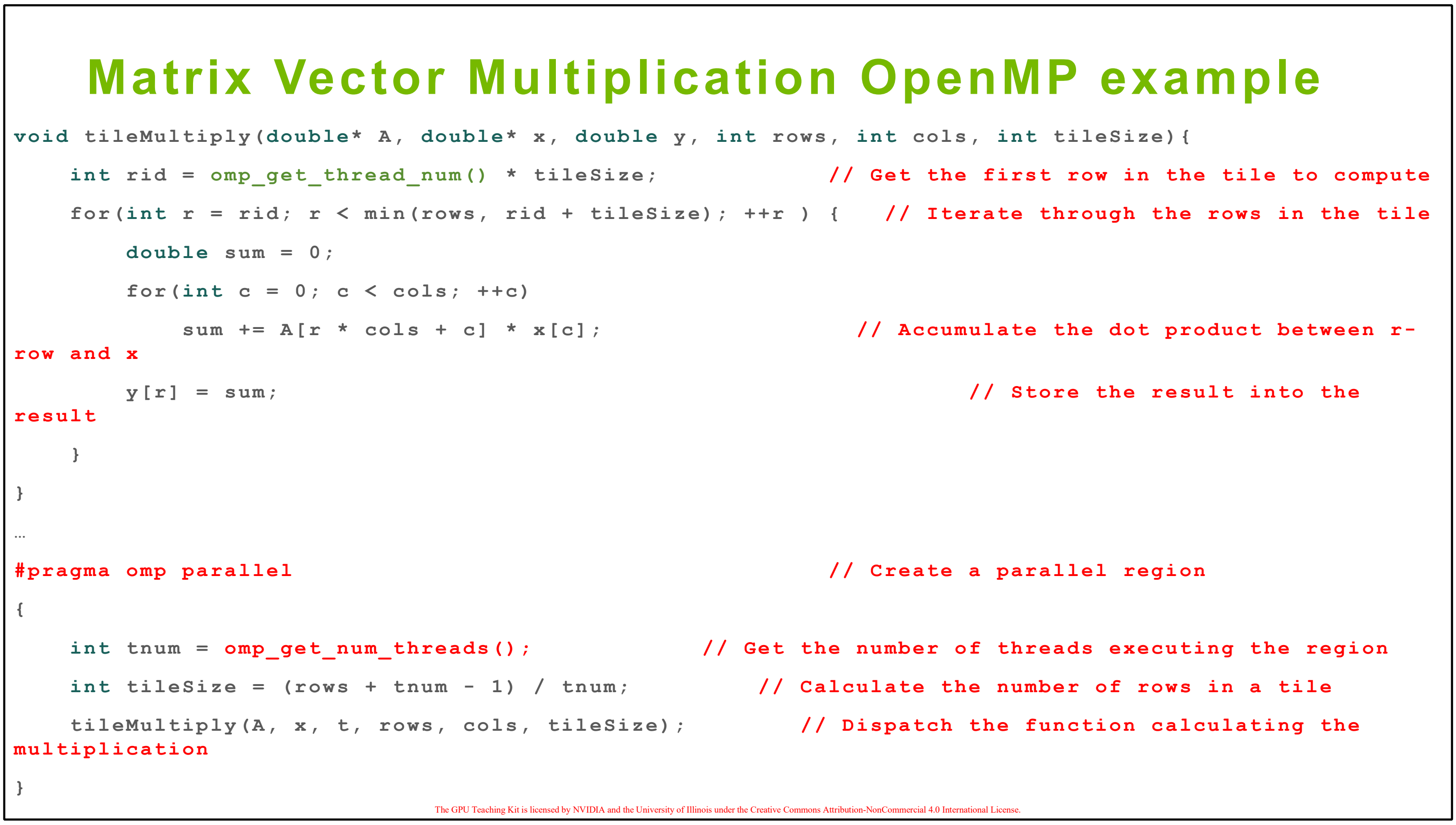
- a final value is 4, because each thread executed the section one at a time
- b final value is 1, because only one thread executed the section



OpenMP API

- OpenMP in addition to the directives have an API, to control or obtain certain runtime parameters.
- As is the case with CUDA, OpenMP allows users to set the number of threads, get the number of threads and uniquely identify each thread.

- **omp_set_num_threads(int):**
  - Sets the desired number of threads to be used by the OpenMP runtime on subsequent parallel regions.
- **omp_get_num_threads():**
  - Returns the number of threads being used in the parallel region.
- **omp_get_thread_num():**
  - Returns a numeric identifier for the calling thread, with different threads having different identifiers.



Matrix Vector Multiplication OpenMP example

```
void tileMultiply(double* A, double* x, double y, int rows, int cols, int tileSize){
    int rid = omp_get_thread_num() * tileSize;              // Get the first row in the tile to compute
    for(int r = rid; r < min(rows, rid + tileSize); ++r ) {    // Iterate through the rows in the tile
        double sum = 0;
        for(int c = 0; c < cols; ++c)
            sum += A[r * cols + c] * x[c];                     // Accumulate the dot product between r-
row and x
        y[r] = sum;                                                    // Store the result into the
result
    }
}
…
#pragma omp parallel                                        // Create a parallel region
{
    int tnum = omp_get_num_threads();                // Get the number of threads executing the region
    int tileSize = (rows + tnum - 1) / tnum;           // Calculate the number of rows in a tile
    tileMultiply(A, x, t, rows, cols, tileSize);          // Dispatch the function calculating the
multiplication
}
```

Compiling an OpenMP program

- OpenMP compilers typically will ignore the OpenMP pragmas unless we specify the OpenMP flag.
- To use the API functions you need to include in the appropriate file:
  - #include <omp.h>
- Examples:
  - GCC & Clang: to compile an OpenMP annotated source file we use:
    - gcc <file to compile> -fopenmp
    - clang <file to compile> -fopenmp
    - Additionally if there are multiple OpenMP libraries, you may specify which version to use, for example –lgomp uses the GNU implementation.
  - NVCC: to compile an OpenMP annotated source file we use:
    - nvcc <options> -Xcompiler -fopenmp

讲授内容：Multi-GPU: OpenMP

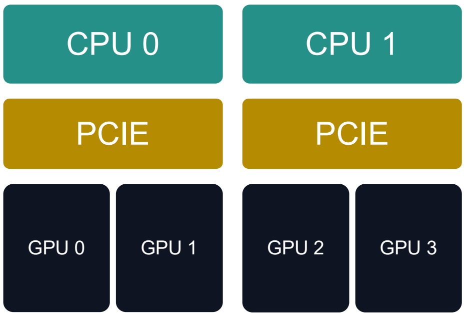
① OpenMP

② Multi GPU Introduction I

③ Multi GPU Introduction II

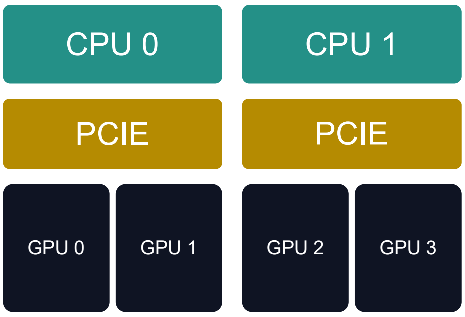
④ Multi GPU patterns with OpenMP & Cooperative Groups

Schematic of a Multi GPU system



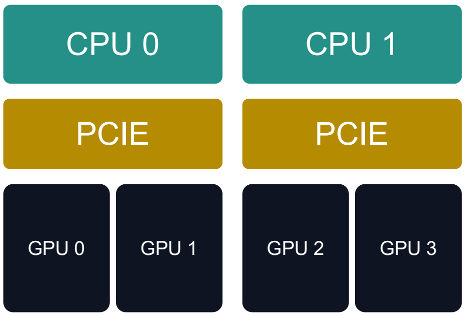
- The figure represents a system with 2 CPU's and 4 GPU's.
- GPU's are numbered from 0 to n-1, where n is the number of GPU’s.
- The CUDA driver always starts with a default active device.
- There are two broad types of Multi GPU communication:
  - Through the PCIE bus
  - Through NVLINK

CUDA host API calls for Multi GPU's



- cudaSetDevice()
  - Set GPU device to use for device code execution on the active host thread.
  - Requires one parameter:
    - An int with the device id number
  - This function doesn’t affect other host threads, meaning that setting the device on one thread will not set the device in other host threads. Also doesn’t affect previous async calls.

CUDA host API calls for Multi GPU's


CPU 0
CPU 1
PCIE
PCIE
GPU 0
GPU 1
GPU 2
GPU 3

- cudaGetDevice()
  - Get GPU device being currently used by the active host thread.
  - Requires one parameter:
    - An int pointer to store the device id
- cudaGetDeviceCount()
  - Get the number of CUDA-capable devices in the system.
  - Requires one parameter:
    - An int pointer to store the device count

CUDA host API calls for Memory allocation with Multiple GPU's

To allocate or associate memory with a specific device using non-Managed CUDA-API calls, it's necessary to call cudaSetDevice() before doing the allocation call.

- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object in terms of bytes

- cudaHostAlloc()
  - Allocates pinned memory on the host
  - Three parameters
    - Address of pointer to the allocated memory
    - Size of the allocated memory in bytes
    - Host Alloc flags

CUDA host API calls for Memory allocation with Multiple GPU's Unified Memory

- If the flag cudaDevAttrConcurrentManagedAccess is set in all devices, then it's not necessary to call cudaSetDevice before the cudaMallocManaged call.
- If the flag is not set but devices can access each others memory, then calling cudaSetDevice before the cudaMallocManaged call will establish the context for the managed memory on the active device.
  - With other devices accessing the data via PCIE at reduced bandwidth.

CUDA runtime calls affected by cudaSetDevice

- If cudaSetDevice() was called before a kernel launching call, the kernel will execute in the active device.
  - It's crucial that every non managed memory being used in the kernel resides in the active device, otherwise an error will occur.
- If cudaSetDevice() was called before a cudaStreamCreate(), then the stream will be associated with the active device.
- The synchronization functions: cudaDeviceSynchronize(), cudaStreamSynchronize() are also affected by cudaSetDevice(), synchronizing tasks only for the active device on the active host thread

Putting it all together, vecAdd

```
    float *m_A0, float *m_B0, *m_A1, float *m_B1, int n;
    int size = n * sizeof(float);
    cudaSetDevice(0);                                            // Will set the active device to 0
    cudaMalloc((void**) &m_A0, size);                  // Will allocate memory on device 0
    cudaMalloc((void**) &m_B0, size);                  // Will allocate memory on device 0
    cudaSetDevice(1);                                            // Will set the active device to 1
    cudaMalloc((void**) &m_A1, size);                  // Will allocate memory on device 1
    cudaMalloc((void**) &m_B1, size);                  // Will allocate memory on device 1


    // Memory initialization on the Host and memory transfers
   cudaSetDevice(0);                                             // Set the device for kernel
execution

   vecAdd<<<gridDim, blockDim>>>(m_A0,m_B0);


   cudaSetDevice(1);                                             // Set the device for kernel
execution

   vecAdd<<< gridDim, blockDim>>>(m_A1,m_B1);


    cudaFree(m_A0); cudaFree(m_B0);

    cudaFree(m_A1); cudaFree(m_B1);
```

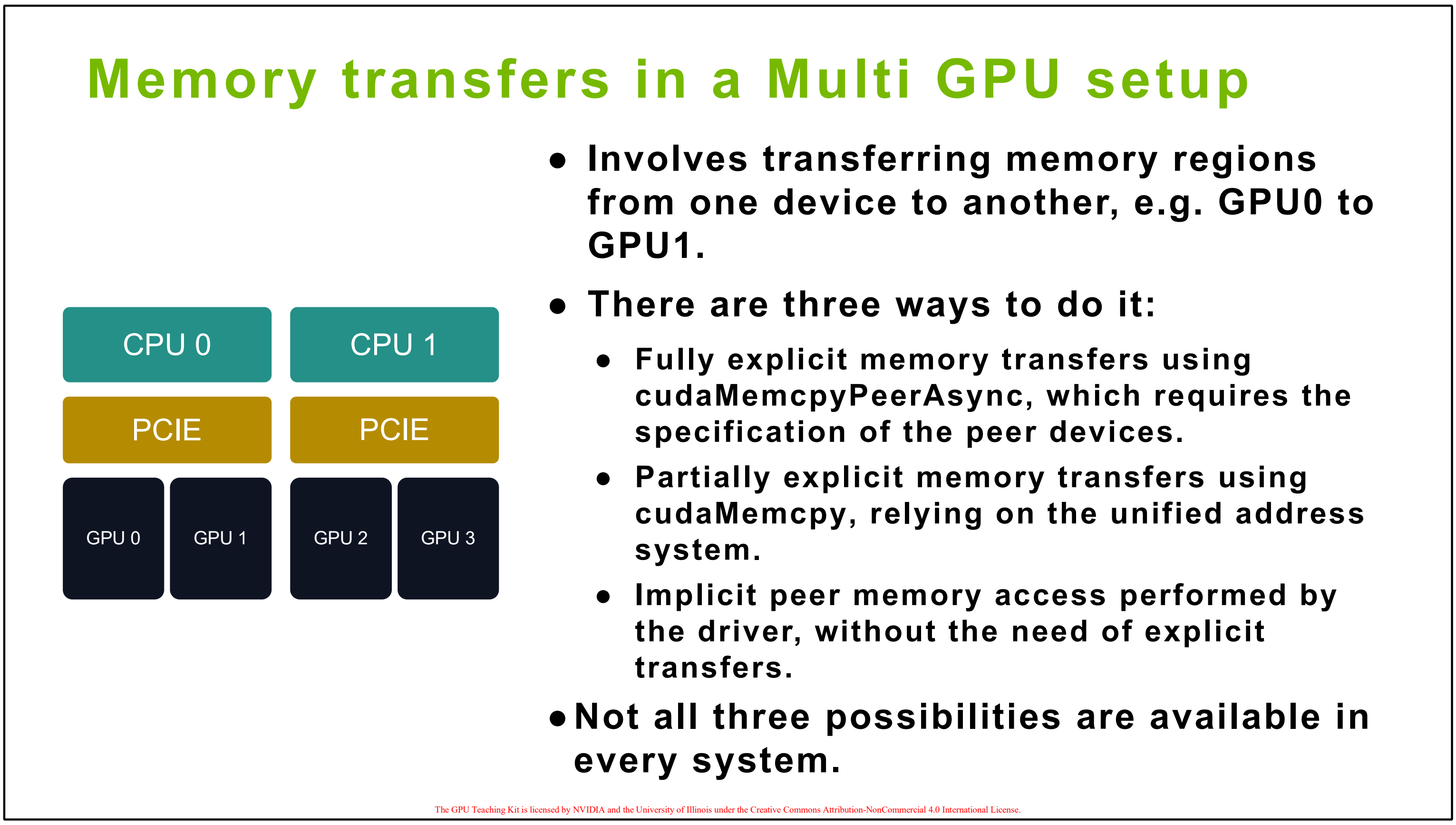
讲授内容：Multi-GPU: OpenMP

① OpenMP

② Multi GPU Introduction I

③ Multi GPU Introduction II

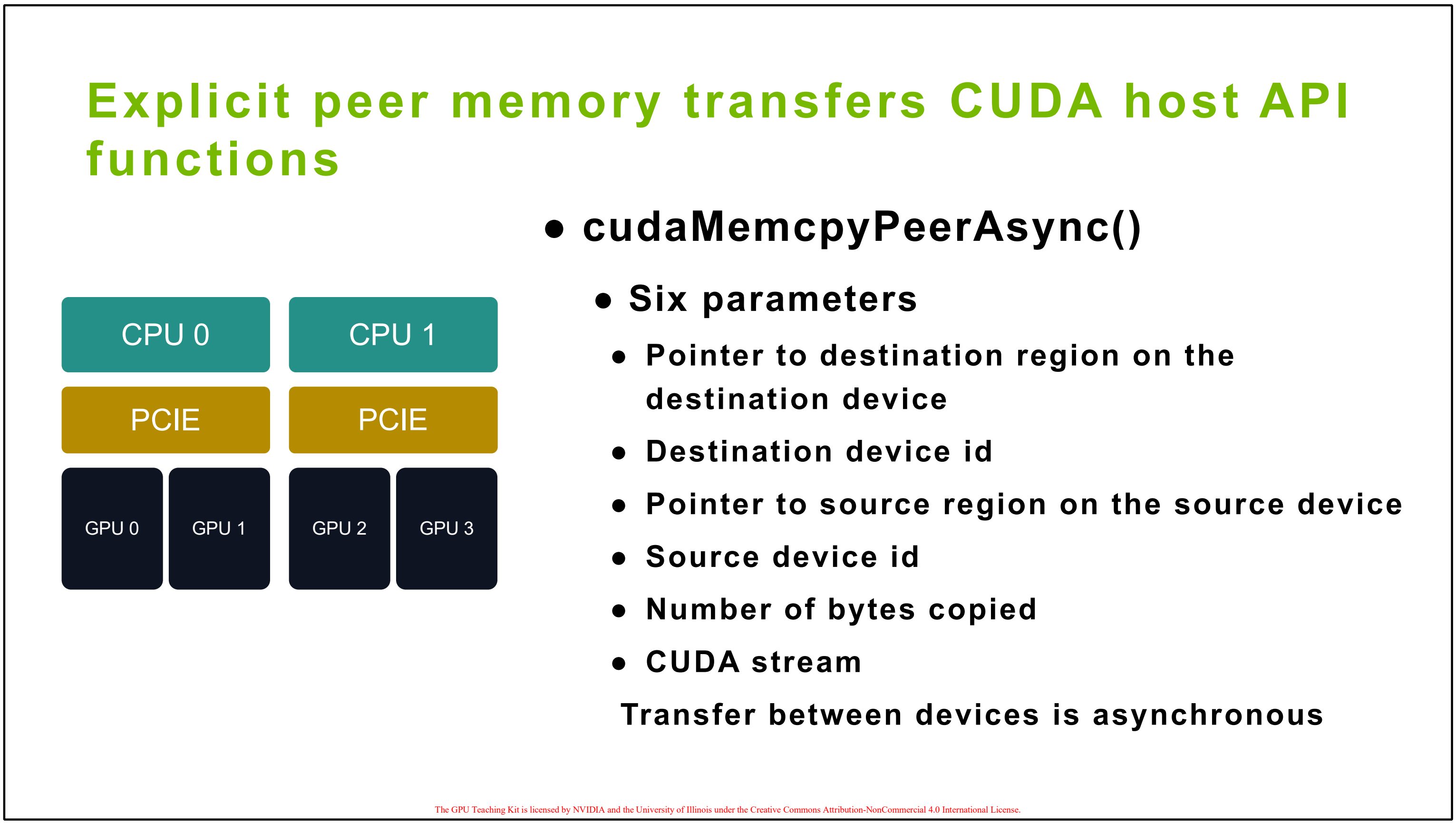
④ Multi GPU patterns with OpenMP & Cooperative Groups



Memory transfers in a Multi GPU setup

CPU 0
CPU 1
PCIE
PCIE
GPU 0
GPU 1
GPU 2
GPU 3

- Involves transferring memory regions from one device to another, e.g. GPU0 to GPU1.
- There are three ways to do it:
  - Fully explicit memory transfers using cudaMemcpyPeerAsync, which requires the specification of the peer devices.
  - Partially explicit memory transfers using cudaMemcpy, relying on the unified address system.
  - Implicit peer memory access performed by the driver, without the need of explicit transfers.
- Not all three possibilities are available in every system.



Explicit peer memory transfers CUDA host API functions

CPU 0
CPU 1
PCIE
PCIE
GPU 0
GPU 1
GPU 2
GPU 3

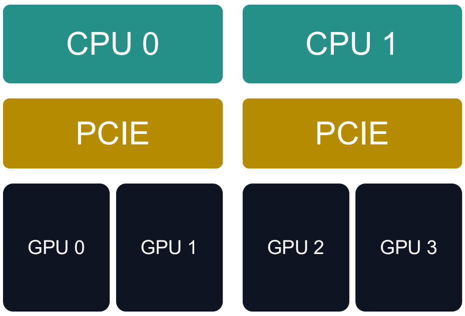
- cudaMemcpyPeerAsync()
  - Six parameters
    - Pointer to destination region on the destination device
    - Destination device id
    - Pointer to source region on the source device
    - Source device id
    - Number of bytes copied
    - CUDA stream

  Transfer between devices is asynchronous

Example: peer transfer cudaMemcpyPeerAsync

```
float *A0, *A1;

int size;


cudaSetDevice(0);  // Set active device to 0
cudaMalloc((void**) &A0, size);// Allocate memory on device 0
cudaSetDevice(1); // Set active device to 1
cudaMalloc((void**) &A1, size);  // Allocate memory on device 1


 // Initialize region A0 on device 0

cudaMemcpyPeerAsync(A1, 1, A0, 0, size, stream); // Copy the data on A0 on device 0 to the region
A1 on device 1


cudaSetDevice(1);  // Set the device for kernel execution

kernel<<<gridDim, blockDim, 0, stream>>>(A1); // Perform computations on A1


cudaFree(A0);  // Free A0 region
cudaFree(A1);  // Free A1 region
```

Explicit peer memory transfers CUDA host API functions



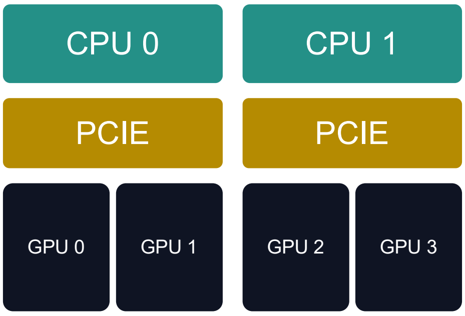
- If the flag cudaDevAttrUnifiedAddressing is set to 1, then you may copy regions between devices using the traditional cudaMemcpy API function, setting the copy kind to cudaMemcpyDefault.
- To check if the flag is set you can use the API function:
  - cudaDeviceGetAttribute()

Example: peer transfer cudaMemcpy

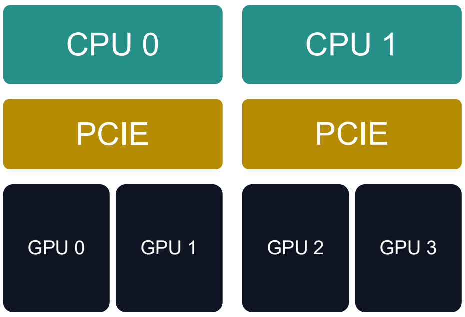
```
float *A0, *A1;

int size;

int unifiedAddr_flag0 = 0;

int unifiedAddr_flag1 = 0;
cudaSetDevice(0);// Set active device to 0
cudaMalloc((void**) &A0, size);// Allocate memory on device 0
cudaSetDevice(1);// Set active device to 1

cudaMalloc((void**) &A1, size);// Allocate memory on device 1

// Initialize region A0 on device 0

cudaDeviceGetAttribute(unifiedAddr_flag0, cudaDevAttrUnifiedAddressing, 0); // Check if unified
addressing is available on dev 0

cudaDeviceGetAttribute(unifiedAddr_flag1, cudaDevAttrUnifiedAddressing, 1); // Check if unified
addressing is available on dev 0

if( unified_addressing_flag0 == 1 && unified_addressing_flag1 == 1 )

 cudaMemcpy(A1, A0, size, cudaMemcpyDefault);  // Copy the data on A0 on device 0 to the region A1 on
device 1

else

 // Throw error indicating the copy couldn’t be performed
```

Implicit peer memory access

- To query if implicit peer memory access are enabled use:
- cudaDeviceCanAccessPeer:
  - Three parameters:
    - Int pointer to place to store the flag.
    - Device id of device trying to access peer
    - Device id of peer device
  - This call is not symmetric, meaning that if the canAccess flag is set to 1 for deviceA and deviceB, it may not be set to 1 for deviceB and deviceA.



Enabling and disableing implicit peer memory access



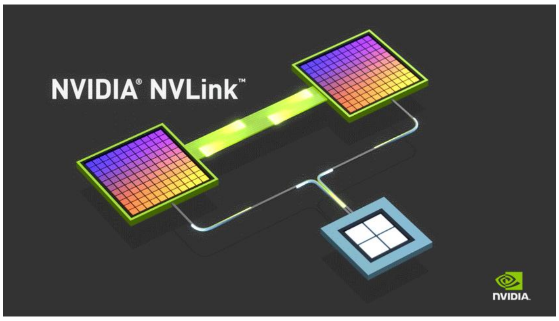
- **cudaDeviceEnablePeerAccess:**
  - **Two parameters:**
    - **Device id to enable access to from the current active device.**
    - **Int flag set to 0.**
  - **This call is not symmetric.**
  - **Returns error cudaErrorInvalidDevice if not possible.**
- **cudaDeviceDisablePeerAccess:**
  - **One parameter:**
    - **Device id to disable access to from the current device.**
  - **This call is not symmetric.**

Example: implicit peer access

```
float *ptrA;  // Pointer to memory region on device devA
int devA;
int devB;
int BcanAccessA = 0;
cudaError_t error;
cudaDeviceCanAccessPeer(&BcanAccessA, devB, devA);  // Check if devB can access
devA memory.
cudaSetDevice(devB);  // Set the current active device to devB
if(BcanAccessA == 0)
 error = cudaDeviceEnablePeerAccess(devA, 0);  // Enable peer accesses to devA
memory
if(error == cudaSuccess) {
  kernel<<<gridDim, blockDim, 0, stream>>>(ptrA);  // Access ptrA on device devA
from device devB
}
cudaDeviceDisablePeerAccess(devA); // Disable peer access to devA, this call is
not needed.
```

NVLINK

- Is a proprietary interconnect technology developed by NVIDIA
- Provides higher memory bandwidth communication between GPUS than PCIE communication
- NVLINK on the Tesla V100 delivers a 300 GB/s communication data rate, whereas the typical PCIE 3.0 link delivers only 32 GB/s



讲授内容：Multi-GPU: OpenMP

① OpenMP

② Multi GPU Introduction I

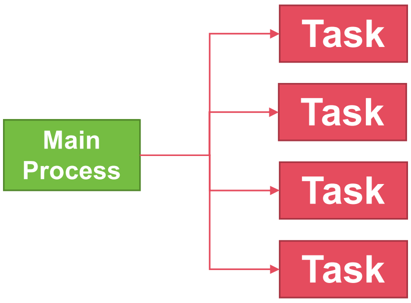
③ Multi GPU Introduction II

④ Multi GPU patterns with OpenMP & Cooperative Groups

Common parallel patterns in a Multi-GPU environment

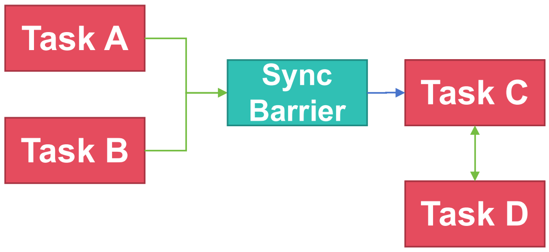
Batch processing:

- Execute the same independent task multiple times with different data.



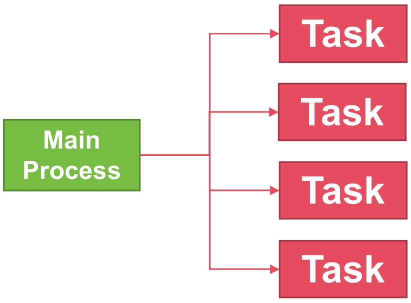
Cooperative patterns:

- Tasks need to cooperate between each other to collectively reach a goal.



Batch processing

- Is an embarrassingly parallel pattern.
- With enough data it is usual we can achieve 100% usage of the compute resources.
- It’s common on video and image processing applications, where we need to apply the same operation to lots of different data.

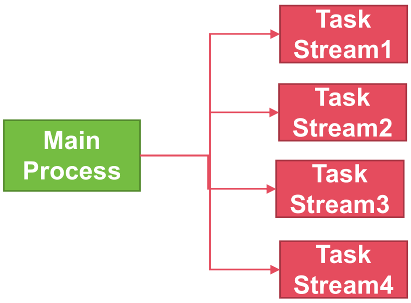


Batch processing

Typical operations to accomplish batch processing:

1. Get number of available devices.
2. Considering the number of devices and number of desired tasks allocate, initialize and copy the memory need it by the algorithm.
3. Create CUDA streams for each of the tasks to be executed concurrently.
4. Launch in parallel the kernel.

* Remember to set the device at the beginning of each group of operations.


Main Process

Task Stream1

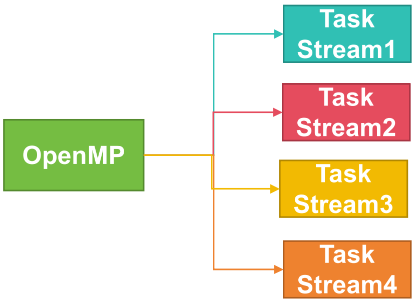
Task Stream2

Task Stream3

Task Stream4

Batch processing example with OpenMP

```
    int deviceCount;
    cudaGetDeviceCount(& deviceCount);
    std::vector<cudaStream_t> streams(deviceCount);
#pragma omp parallel for num_threads(deviceCount)
    for(int dev = 0; dev < deviceCount; ++dev) {
        cudaSetDevice(dev);
        cudaStreamCreate(&streams[i]);
        // Allocate, initialize and transfer memory
    }
#pragma omp parallel for num_threads(deviceCount)
    for(int dev = 0; dev < deviceCount; ++dev) {
        cudaSetDevice(dev);
        kernel<<<gridDim, blockDim,
streams[i]>>>(…);
    }
```

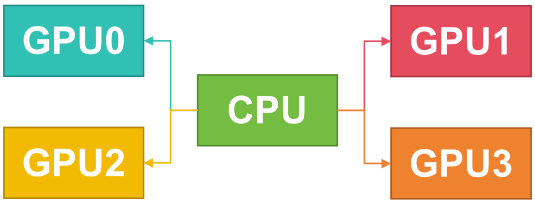

OpenMP

Task Stream1

Task Stream2

Task Stream3

Task Stream4


GPU0

GPU1

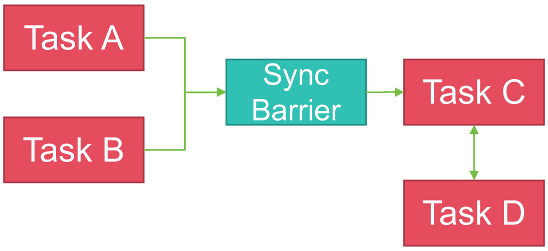
CPU

GPU2

GPU3

Cooperative patterns

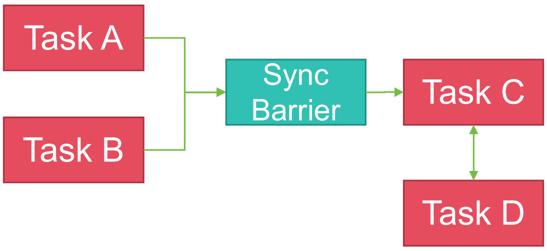
- Might have unavoidable syncing points causing tasks to wait and thus wasting compute resources.
- In some cases even when massive amounts of input data it might not reach 100% resource usage.
- It’s common on applications with steps to reach a goal like iterative algorithms.

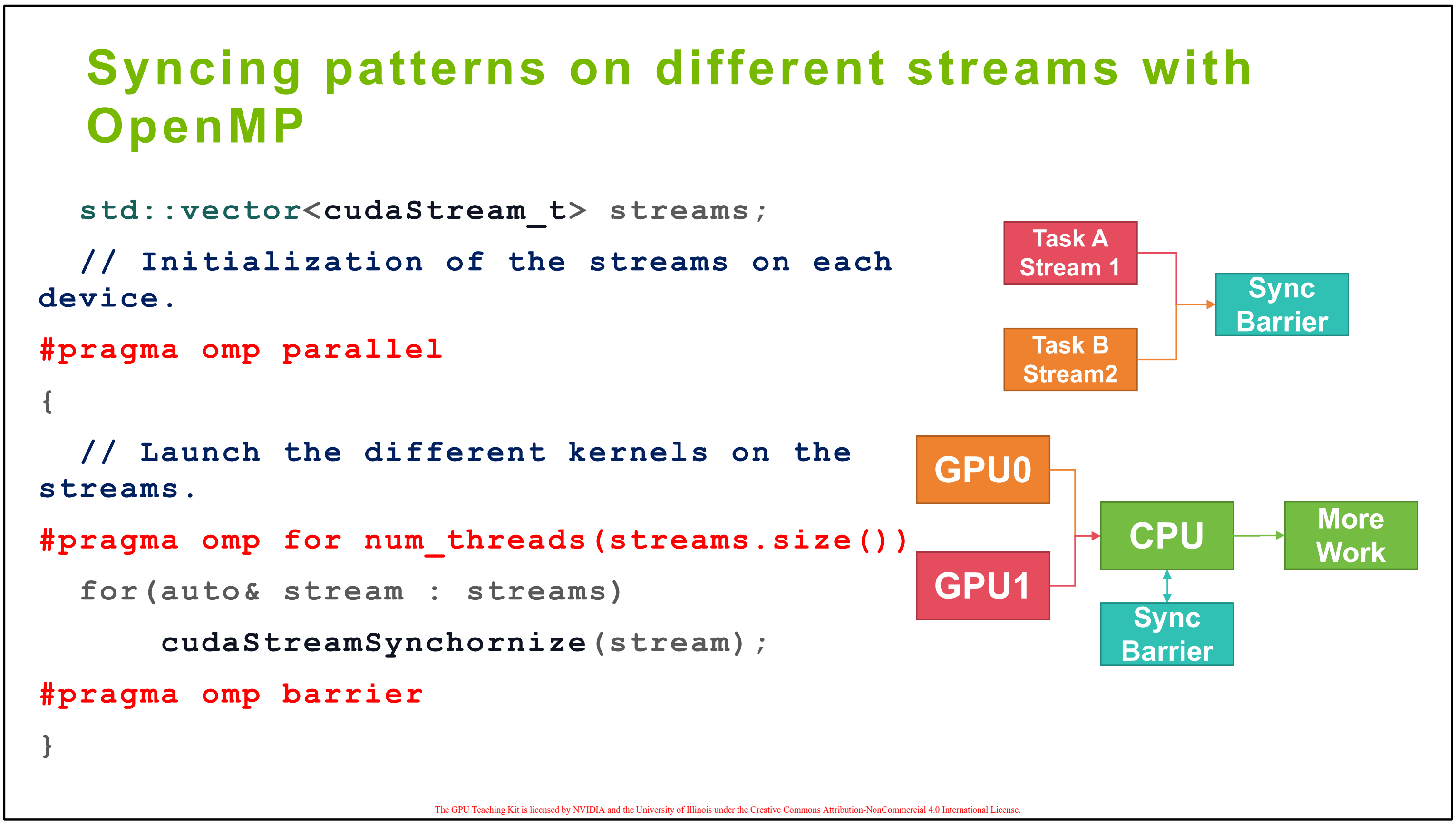


Cooperative patterns

- With cooperative patterns there is no single fit solution like with batch processing.
- Thus the process consists in a loop of:

1. Launching the code in parallel.
2. Profiling it.
3. Analyzing and removing bottlenecks.

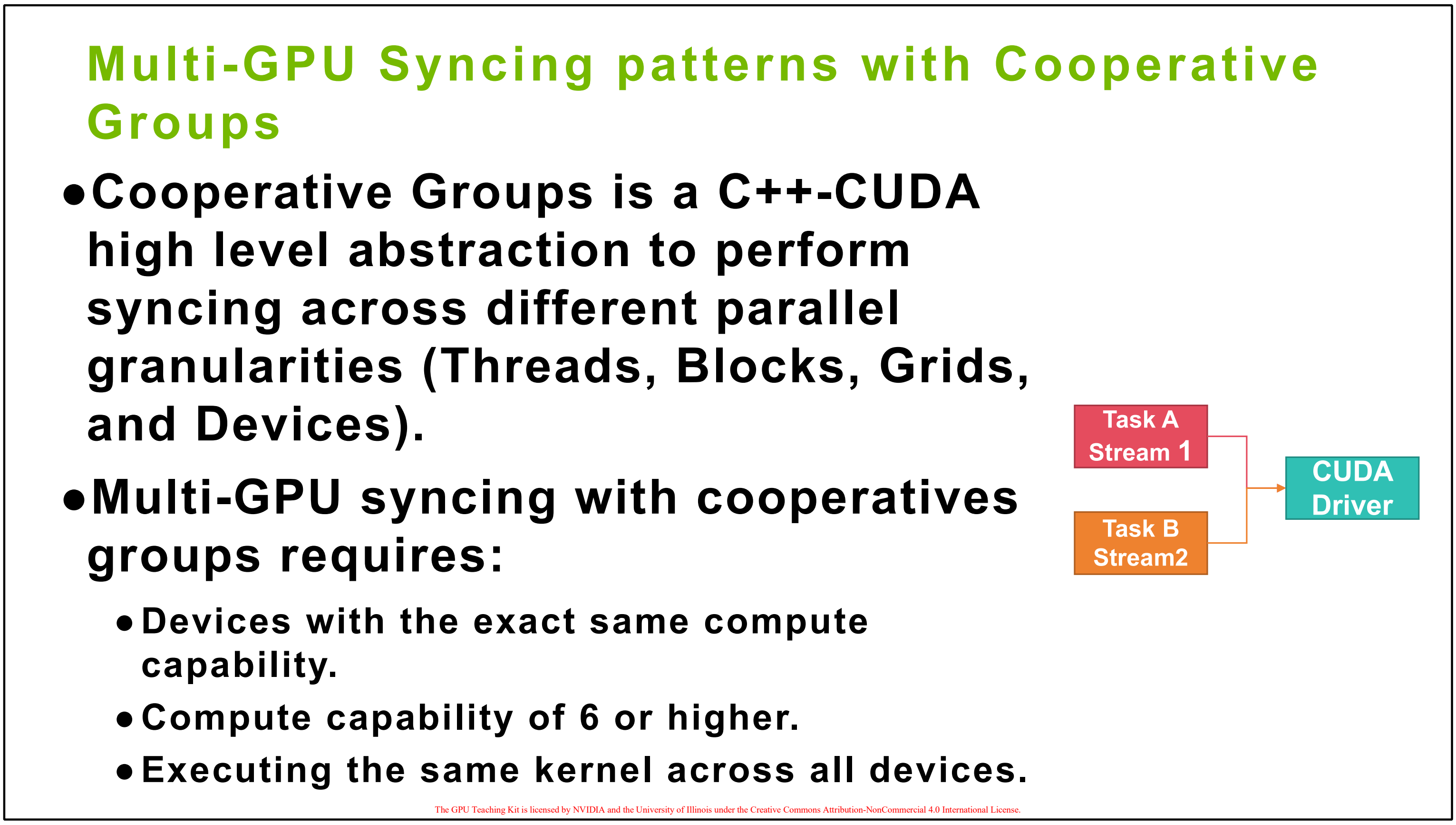




Syncing patterns on different streams with OpenMP

```
  std::vector<cudaStream_t> streams;
  // Initialization of the streams on each
device.
#pragma omp parallel
{
  // Launch the different kernels on the
streams.
#pragma omp for num_threads(streams.size())
  for(auto& stream : streams)
      cudaStreamSynchornize(stream);
#pragma omp barrier
}
```

Task A Stream 1
Task B Stream2
Sync Barrier
GPU0
GPU1
CPU
More Work
Sync Barrier

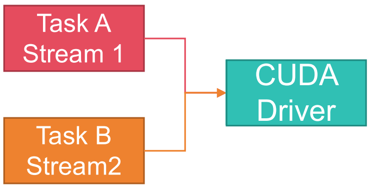


Multi-GPU Syncing patterns with Cooperative Groups

- Cooperative Groups is a C++-CUDA high level abstraction to perform syncing across different parallel granularities (Threads, Blocks, Grids, and Devices).
- Multi-GPU syncing with cooperatives groups requires:
  - Devices with the exact same compute capability.
  - Compute capability of 6 or higher.
  - Executing the same kernel across all devices.

Multi-GPU Syncing patterns with Cooperative Groups

- To enable the use of Cooperative Groups we need to include the file **cooperative_groups.h** and use the namespace **cooperative_groups**.
- The kernels needs to be compiled using separate compilation and then linked with the **–rdc=true** flag.
- You also need to ensure that MPS is disabled and CU_DEVICE_ATTRIBUTE_COOPERATIVE_MULTI_DEVICE_LAUNCH is set in the device properties using **cuDeviceGetAttribute** the API function.



Multi-GPU Syncing patterns with Cooperative Groups

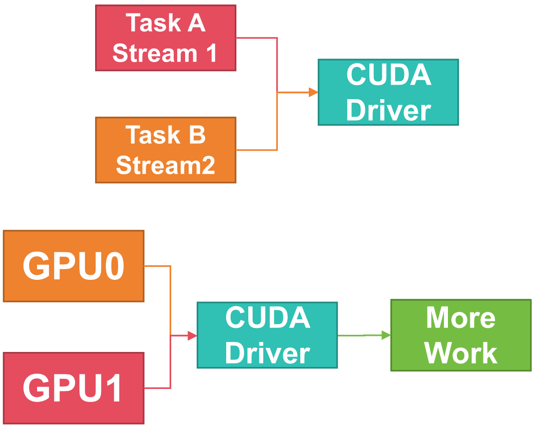
Launching a kernel with Multi-GPU syncing and Cooperative Groups requires using the API function:

- cudaLaunchCooperativeKernelMultiDevice:
  - The first parameter is an array of CUDA_LAUNCH_PARAMS, where except for kernel params and the stream, all other fields needs be the same.
  - The number of devices to use.

```
typedef struct CUDA_LAUNCH_PARAMS_st {
  CUfunction function;       // Kernel to launch.
  unsigned int gridDimX;    // Grid dimensions.
  unsigned int gridDimY;
  unsigned int gridDimZ;
  unsigned int blockDimX; // Block dimensions.
  unsigned int blockDimY;
  unsigned int blockDimZ;
  unsigned int sharedMemBytes; // Shared memory
size.
  CUstream hStream;          // Stream to perform
the work
  void **kernelParams;      // Kernel parameters
} CUDA_LAUNCH_PARAMS;
```

Multi-GPU Syncing patterns with Cooperative Groups

```
#include <cooperative_groups.h>
using namespace cooperative_groups;
void __global__ kernel(...) {
    // Work
    multi_grid_group multi_grid =
this_multi_grid();
    multi_grid.sync();
   // Work
}
//  Work
cudaLaunchCooperativeKernelMultiDevice
(...);
```



讲授内容

- ➢Related Programming Models: OpenCL
- ➢Related Programming Models: OpenACC
- ➢Multi-GPU: OpenMP
- ➢Related Programming Models: MPI

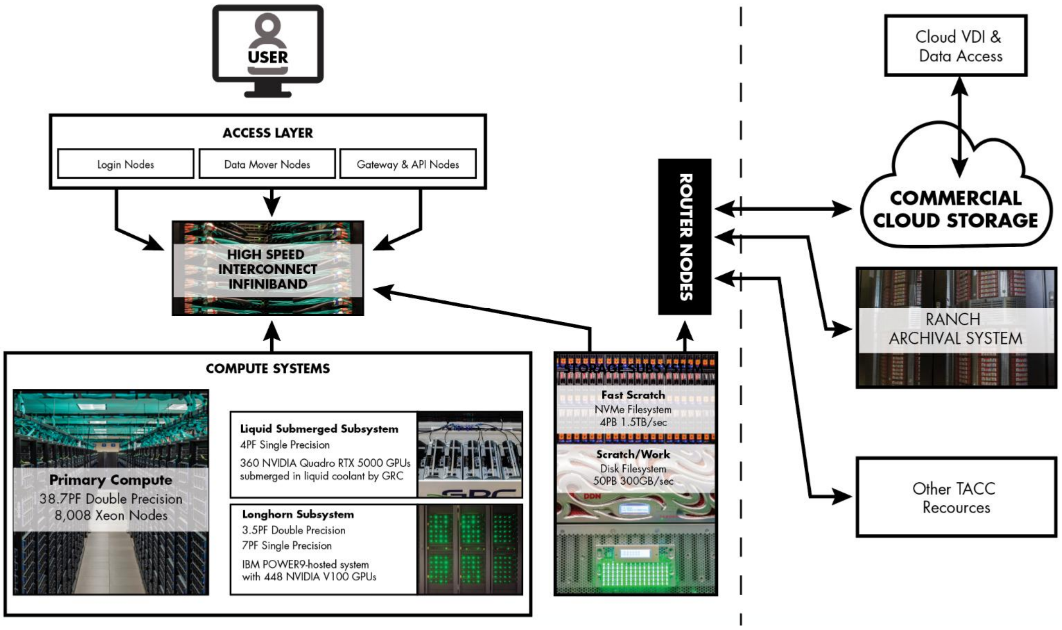
讲授内容：Related Programming Models: MPI

① Warps and SIMD Hardware

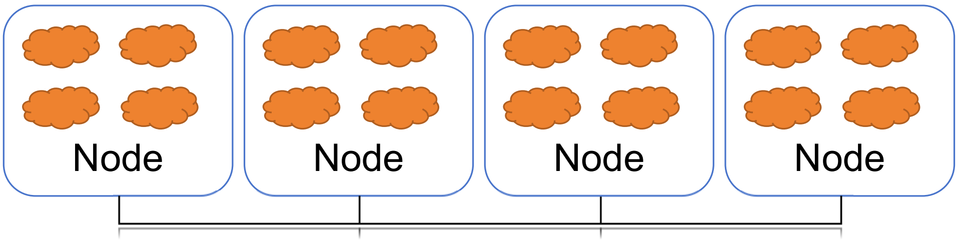
② Performance Impact of Control Divergence

③ Overlapping Computation with Communication

Frontera – Installed at TACC 9/2019



MPI – Programming and Execution Model



Many processes distributed in a cluster

Each process computes part of the output

Processes communicate with each other

Processes can synchronize

MPI Initialization, Info and Sync

```
int MPI_Init(int *argc, char ***argv)
```

Initialize MPI

```
MPI_COMM_WORLD
```

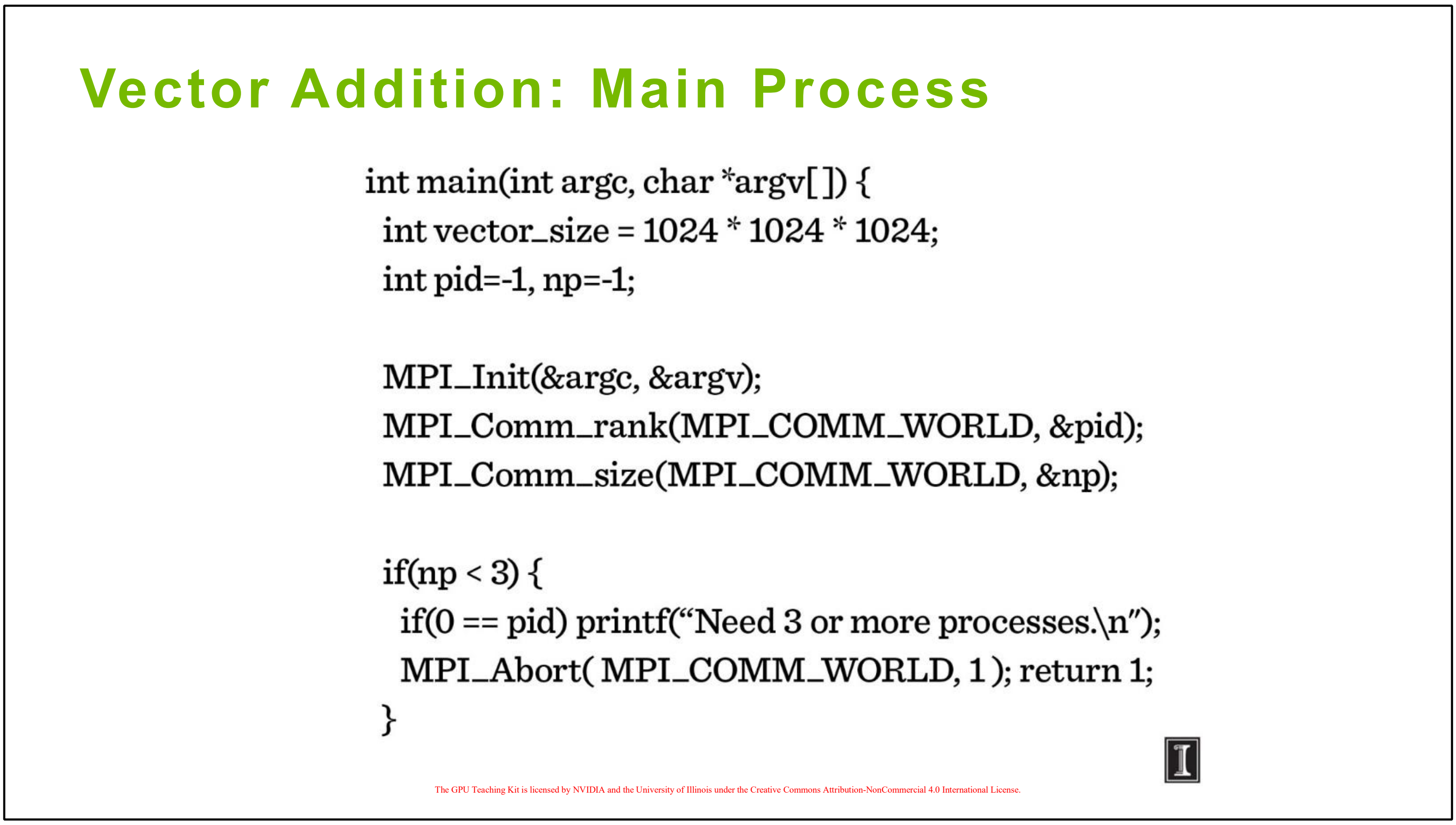
MPI group with all allocated nodes

```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

Rank of the calling process in group of comm

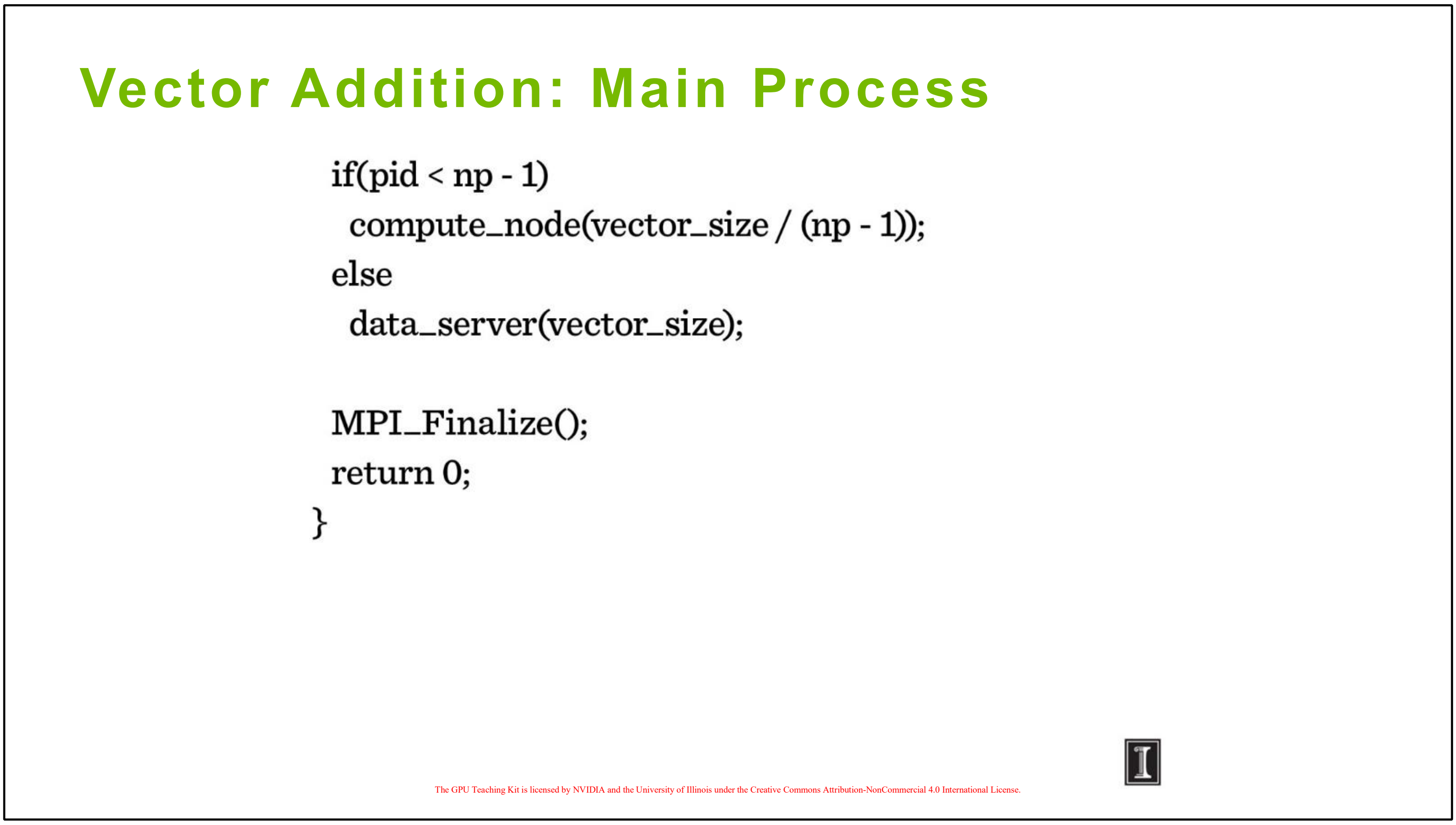
```
int MPI_Comm_size (MPI_Comm comm, int *size)
```

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("Need 3 or more processes.\n");
  MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
 }
```



Vector Addition: Main Process

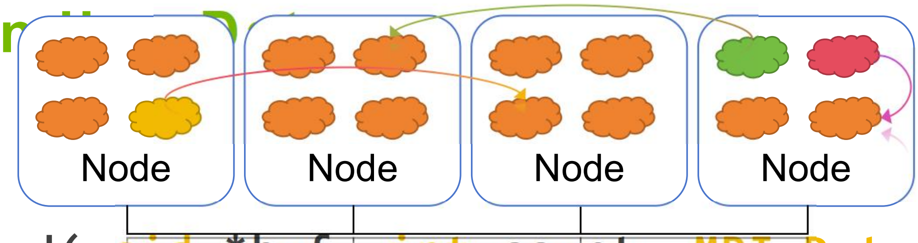
```
 if(pid < np - 1)
  compute_node(vector_size / (np - 1));
 else
  data_server(vector_size);

 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**: Initial address of send buffer (choice)
- **Count**: Number of elements in send buffer (nonnegative integer)
- **Datatype**: Datatype of each send buffer element (handle)
- **Dest**: Rank of destination (integer)
- **Tag**: Message tag (integer)
- **Comm**: Communicator (handle)

MPI Sen



```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

- **Buf**: Initial address of send buffer (choice)
- **Count**: Number of elements in send buffer (nonnegative integer)
- **Datatype**: Datatype of each send buffer element (handle)
- **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: Initial address of receive buffer (choice)

Count: Maximum number of elements in receive buffer (integer)

Datatype: Datatype of each receive buffer element (handle)

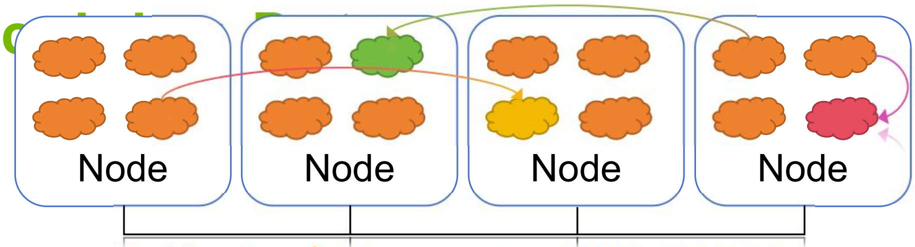
Source: Rank of source (integer)

Tag: Message tag (integer)

Comm: Communicator (handle)

Status: Status object (Status)

MPI Rec



```
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 (choice)

Count: Maximum number of elements in receive buffer (integer)

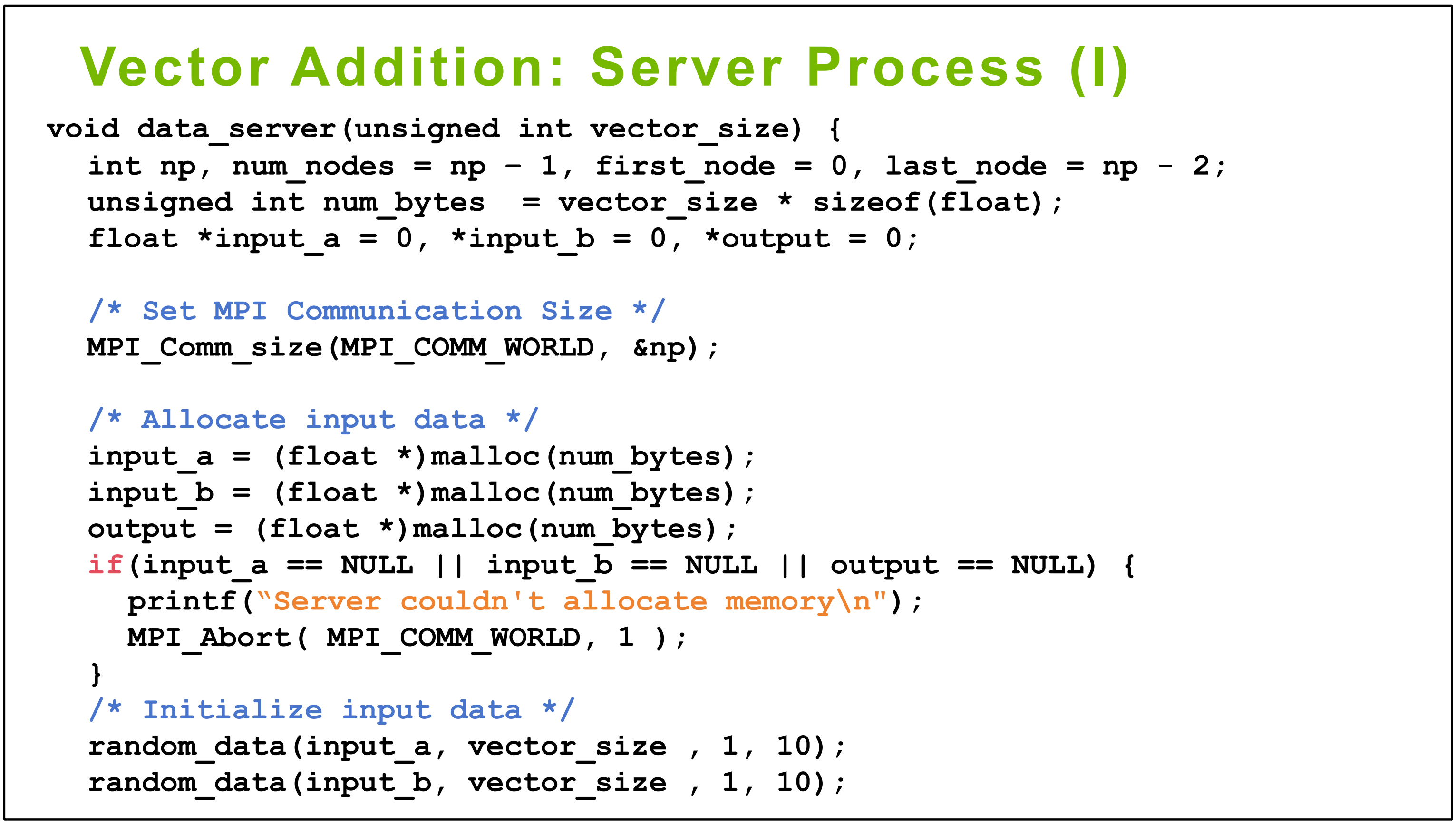
Datatype: Datatype of each receive buffer element (handle)

Source: Rank of source (integer)

Tag: Message tag (integer)

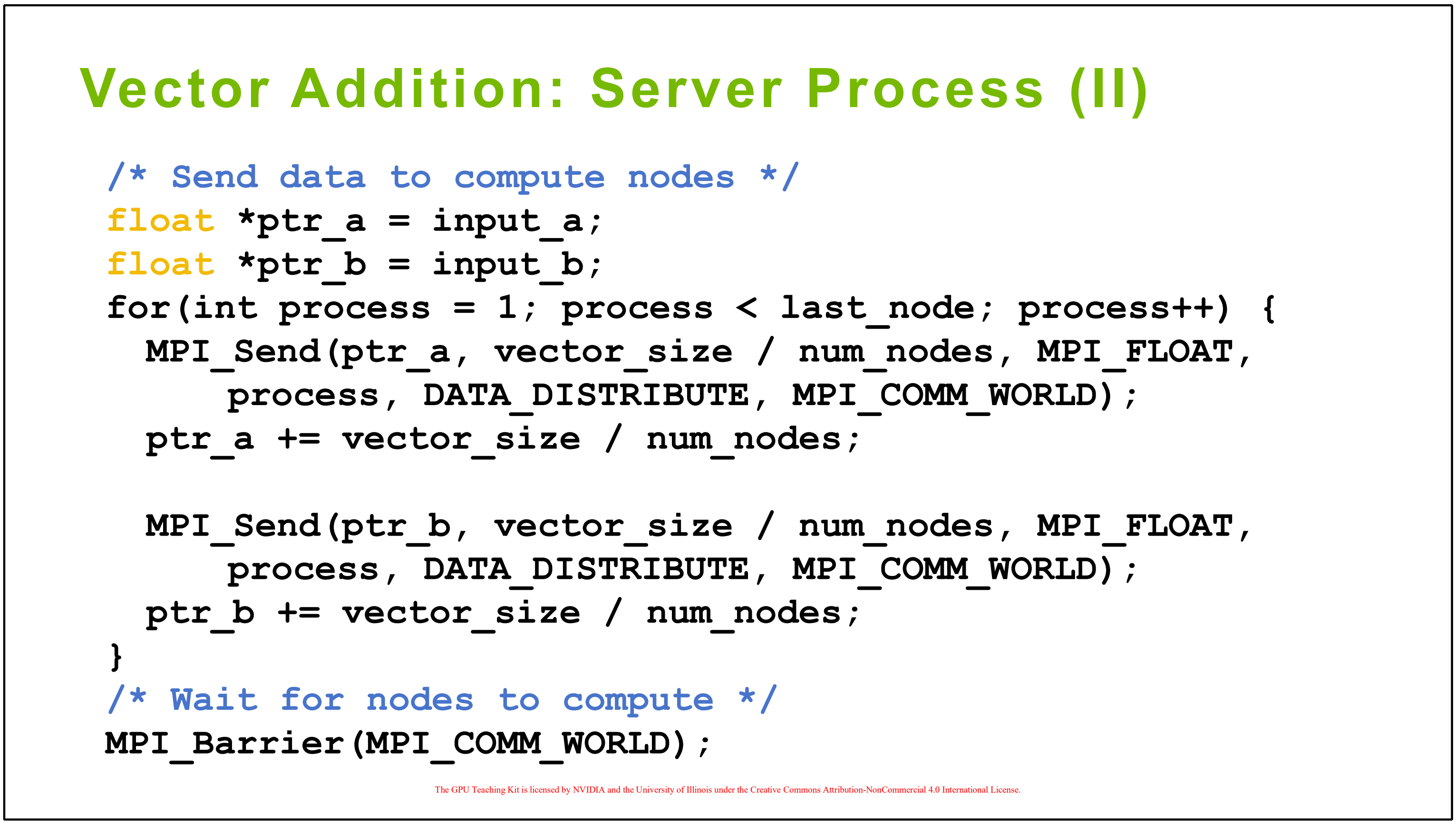
Comm: Communicator (handle)

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 , 1, 10);
```



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

Vector Addition: Server Process (III)

```
  /* Wait for previous communications */
  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_a);
  free(input_b);
  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);
```

MPI Barriers

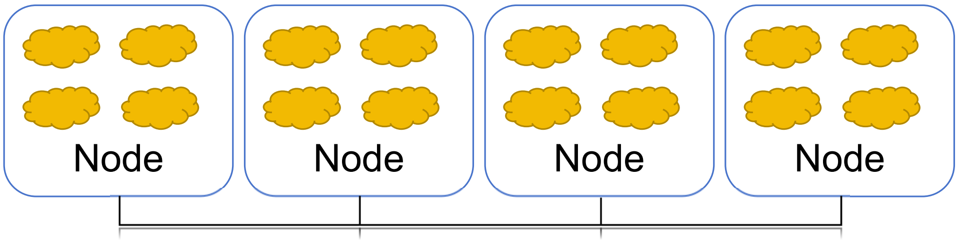
- int MPI_Barrier (MPI_Comm comm)
  - Comm: Communicator (handle)
- Blocks the caller until all group members have called it; the call returns at any process only after all group members have entered the call.

MPI Barriers

- Wait until all other processes in the MPI group reach the same barrier
  - All processes are executing Do_Stuff()
  - Some processes reach the barrier and the wait in the barrier until all reach the barrier

Example Code

```
Do_stuff();

MPI_Barrier();

Do_more_stuff();
```



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];
    }
    /* Report to barrier after computation is done*/
    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);
  }
```

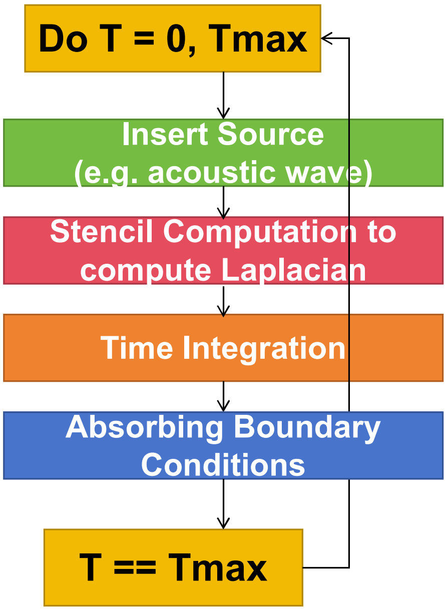
讲授内容：Related Programming Models: MPI

① Warps and SIMD Hardware

② Performance Impact of Control Divergence

③ Overlapping Computation with Communication

A Typical Wave Propagation Application



Do T = 0, Tmax

Insert Source (e.g. acoustic wave)

Stencil Computation to compute Laplacian

Time Integration

Absorbing Boundary Conditions

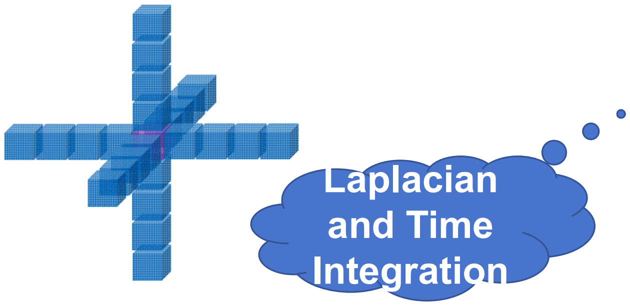
T == Tmax

Review of Stencil Computations

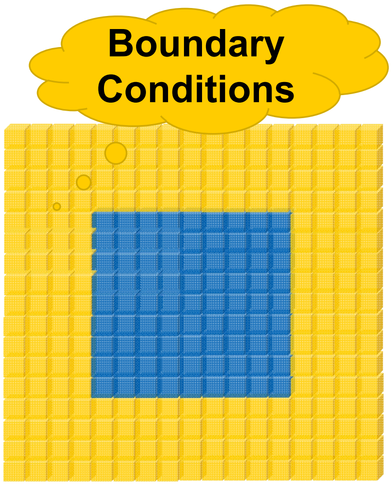
– Example: wave propagation modeling

– $\nabla^2 U - \frac{1}{v^2}\frac{\partial U}{\partial t} = 0$

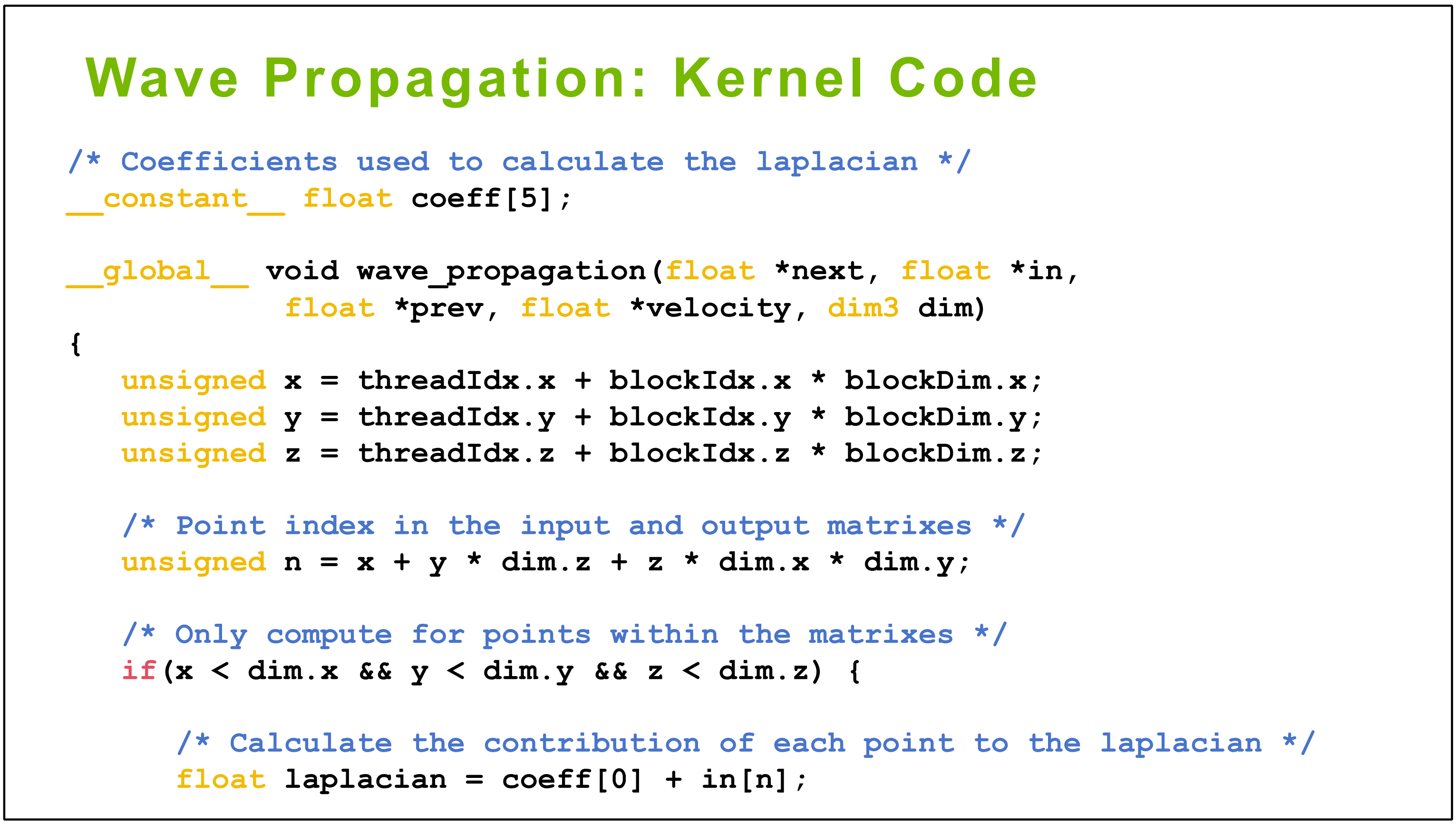
– Approximate Laplacian using finite differences



Laplacian and Time Integration

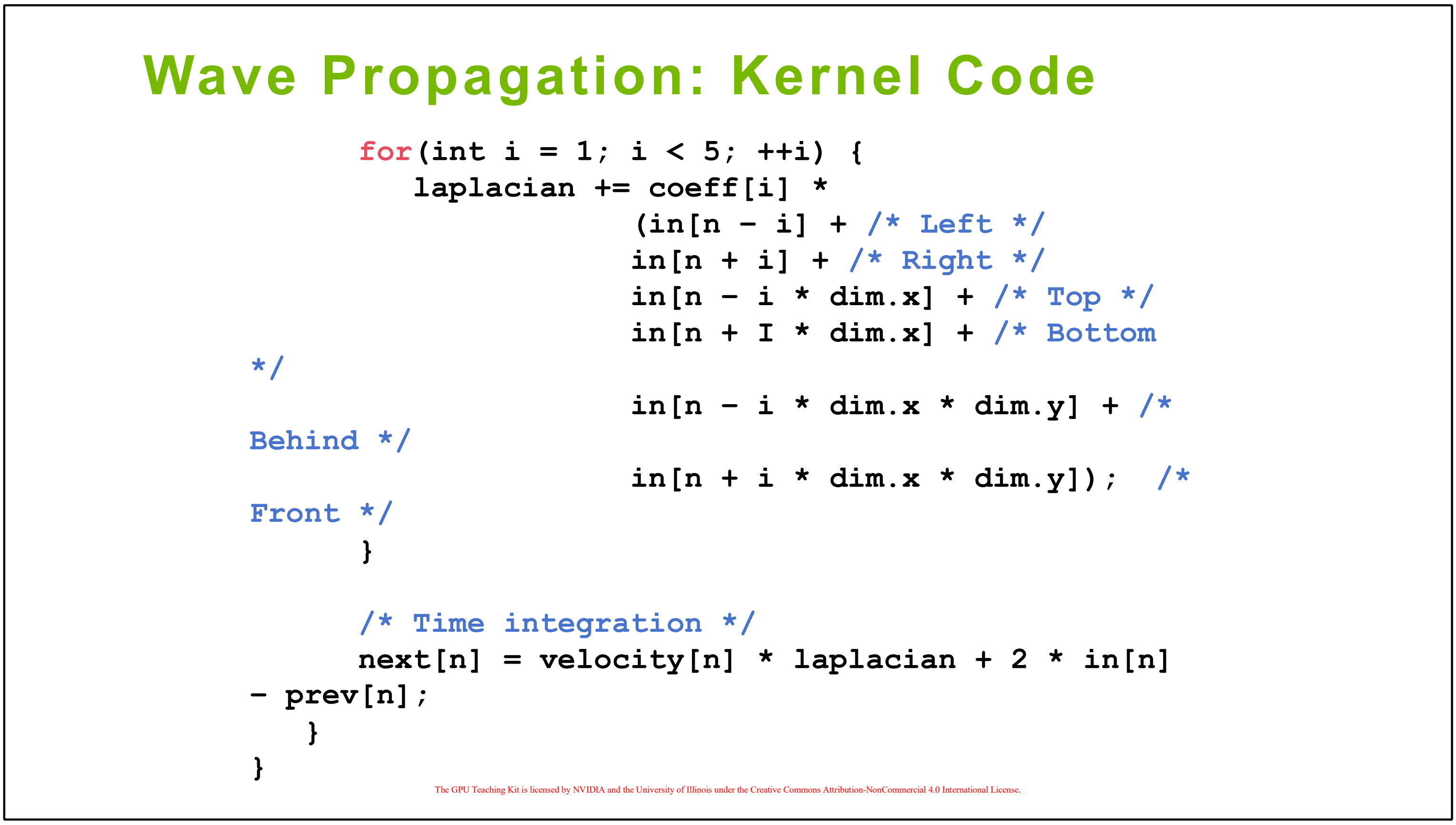


Boundary Conditions



Wave Propagation: Kernel Code

```
/* Coefficients used to calculate the laplacian */
__constant__ float coeff[5];

__global__ void wave_propagation(float *next, float *in,
            float *prev, float *velocity, dim3 dim)
{
   unsigned x = threadIdx.x + blockIdx.x * blockDim.x;
   unsigned y = threadIdx.y + blockIdx.y * blockDim.y;
   unsigned z = threadIdx.z + blockIdx.z * blockDim.z;

   /* Point index in the input and output matrixes */
   unsigned n = x + y * dim.z + z * dim.x * dim.y;

   /* Only compute for points within the matrixes */
   if(x < dim.x && y < dim.y && z < dim.z) {

      /* Calculate the contribution of each point to the laplacian */
      float laplacian = coeff[0] + in[n];
```

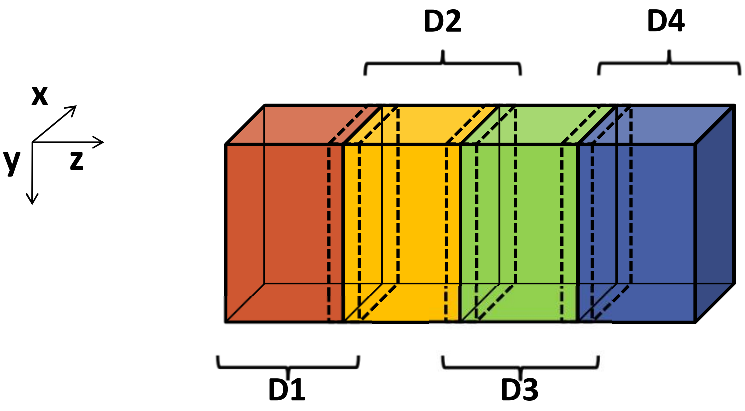


Wave Propagation: Kernel Code

```
      for(int i = 1; i < 5; ++i) {
         laplacian += coeff[i] *
                     (in[n - i] + /* Left */
                     in[n + i] + /* Right */
                     in[n - i * dim.x] + /* Top */
                     in[n + I * dim.x] + /* Bottom
*/
                     in[n - i * dim.x * dim.y] + /*
Behind */
                     in[n + i * dim.x * dim.y]);  /*
Front */
      }

      /* Time integration */
      next[n] = velocity[n] * laplacian + 2 * in[n]
- prev[n];
   }
}
```

Stencil Domain Decomposition

- Volumes are split into tiles (along the Z-axis)
  - 3D-Stencil introduces data dependencies



Wave Propagation: Main Process

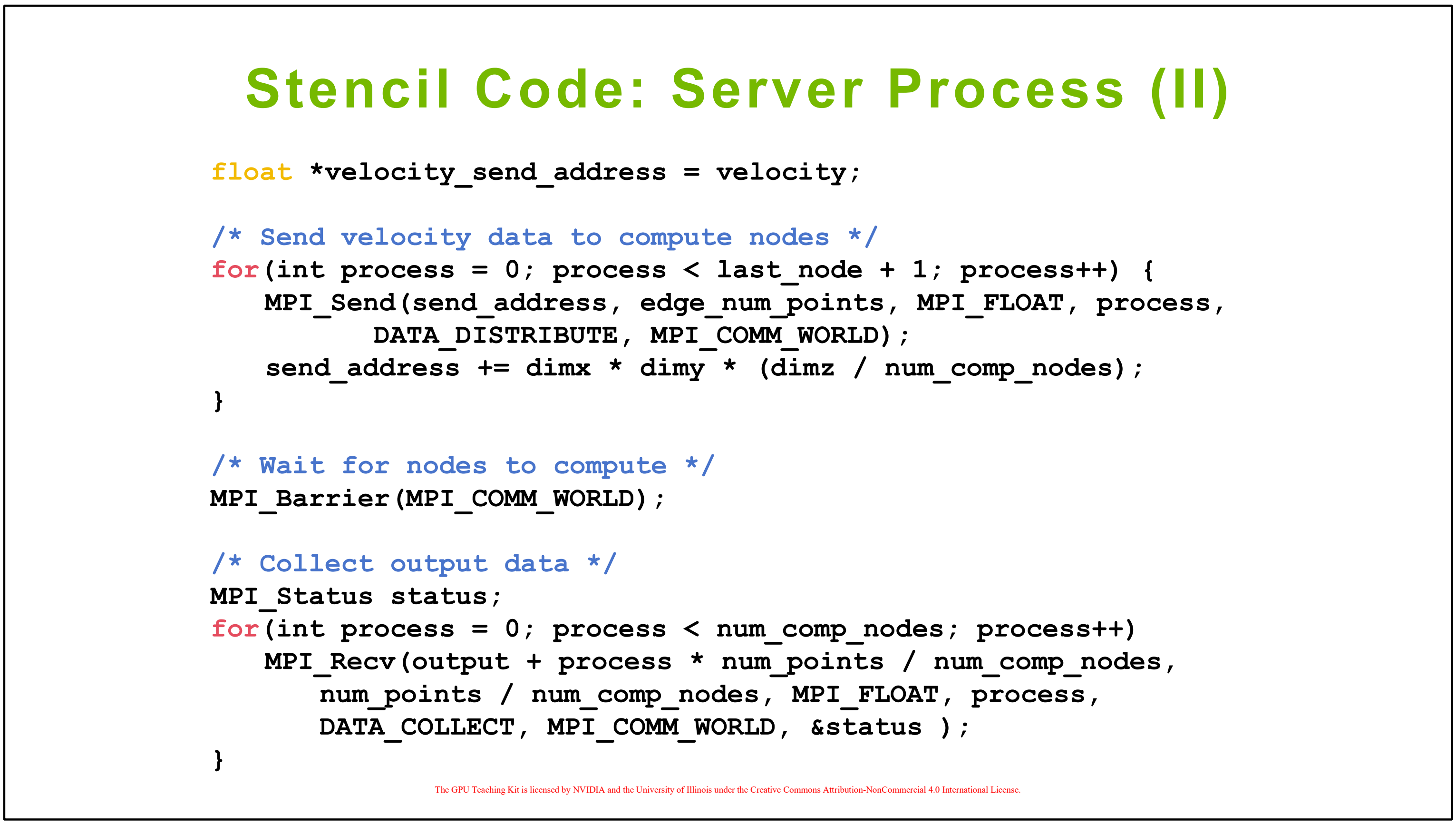
```
int main(int argc, char *argv[]) {
    int pad = 0, dimx  = 480+pad, dimy  = 480, dimz  = 400, nreps = 100;
    int pid=-1, np=-1;

    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(dimx, dimy, dimz / (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz, nreps );

    MPI_Finalize();
    return 0;
}
```

Stencil Code: Server Process (I)

```
void data_server(int dimx, int dimy, int dimz, int nreps)
{
    int np, num_comp_nodes = np – 1, first_node = 0,
last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes  = num_points * sizeof(float);
    float *input=0, *output = NULL, *velocity = NULL;
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    velocity = (float *)malloc(num_bytes);
    if(input == NULL || output == NULL || velocity == NULL)
{
        printf("Server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data and velocity */
    random_data(input, dimx, dimy ,dimz , 1, 10);
    random_data(velocity, dimx, dimy ,dimz , 1, 10);
```

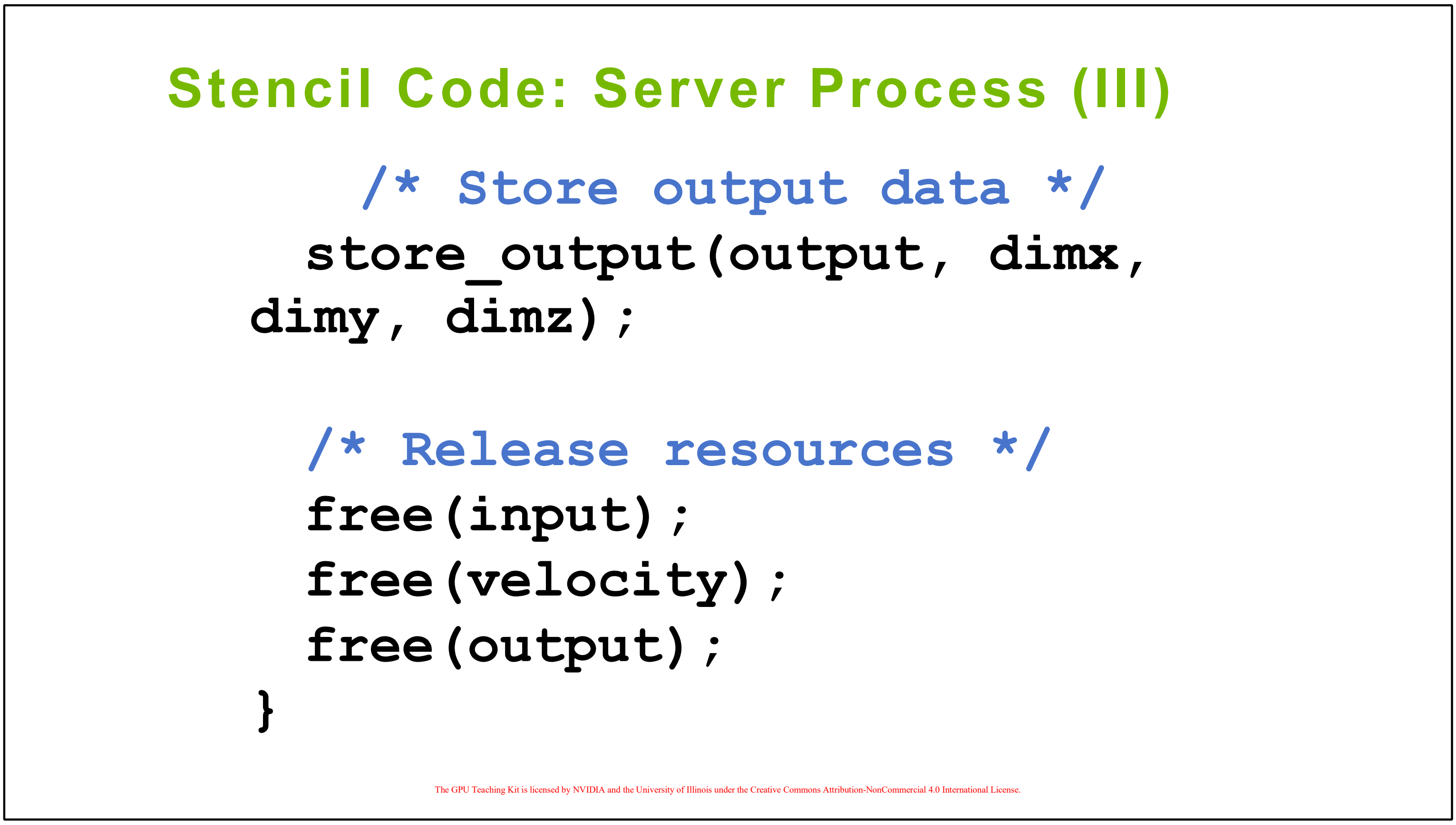
Stencil Code: Server Process (II)

```
/* Calculate number of shared points */
int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
float *input_send_address = input;

/* Send input data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, first_node,
        DATA_DISTRIBUTE, MPI_COMM_WORLD );
send_address += dimx * dimy * (dimz / num_comp_nodes - 4);

/* Send input data to "internal" compute nodes */
for(int process = 1; process < last_node; process++) {
    MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
            DATA_DISTRIBUTE, MPI_COMM_WORLD);
    send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Send input data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_REAL, last_node,
        DATA_DISTRIBUTE, MPI_COMM_WORLD);
```



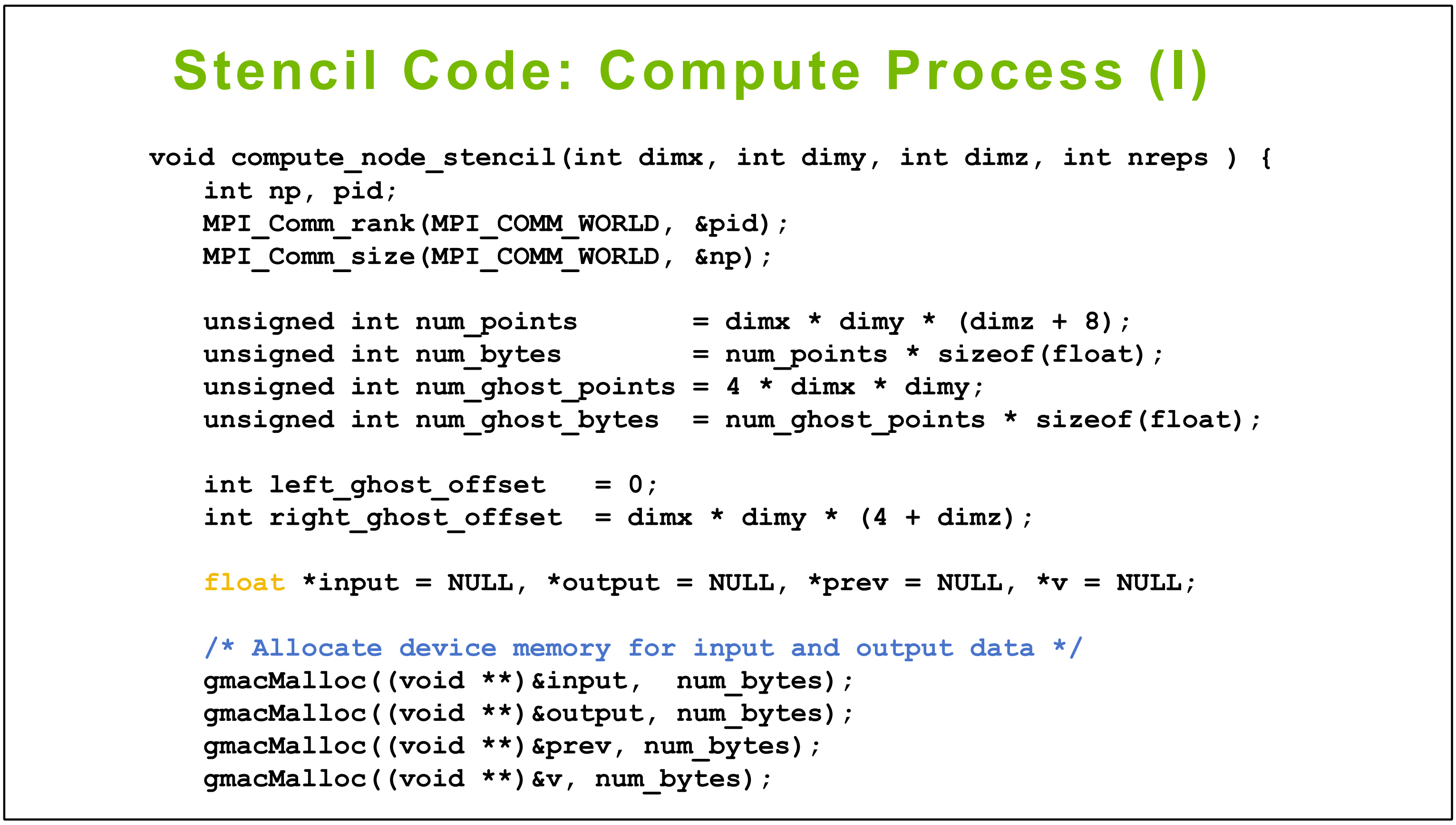
Stencil Code: Server Process (II)

```
float *velocity_send_address = velocity;

/* Send velocity data to compute nodes */
for(int process = 0; process < last_node + 1; process++) {
   MPI_Send(send_address, edge_num_points, MPI_FLOAT, process,
         DATA_DISTRIBUTE, MPI_COMM_WORLD);
   send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);

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



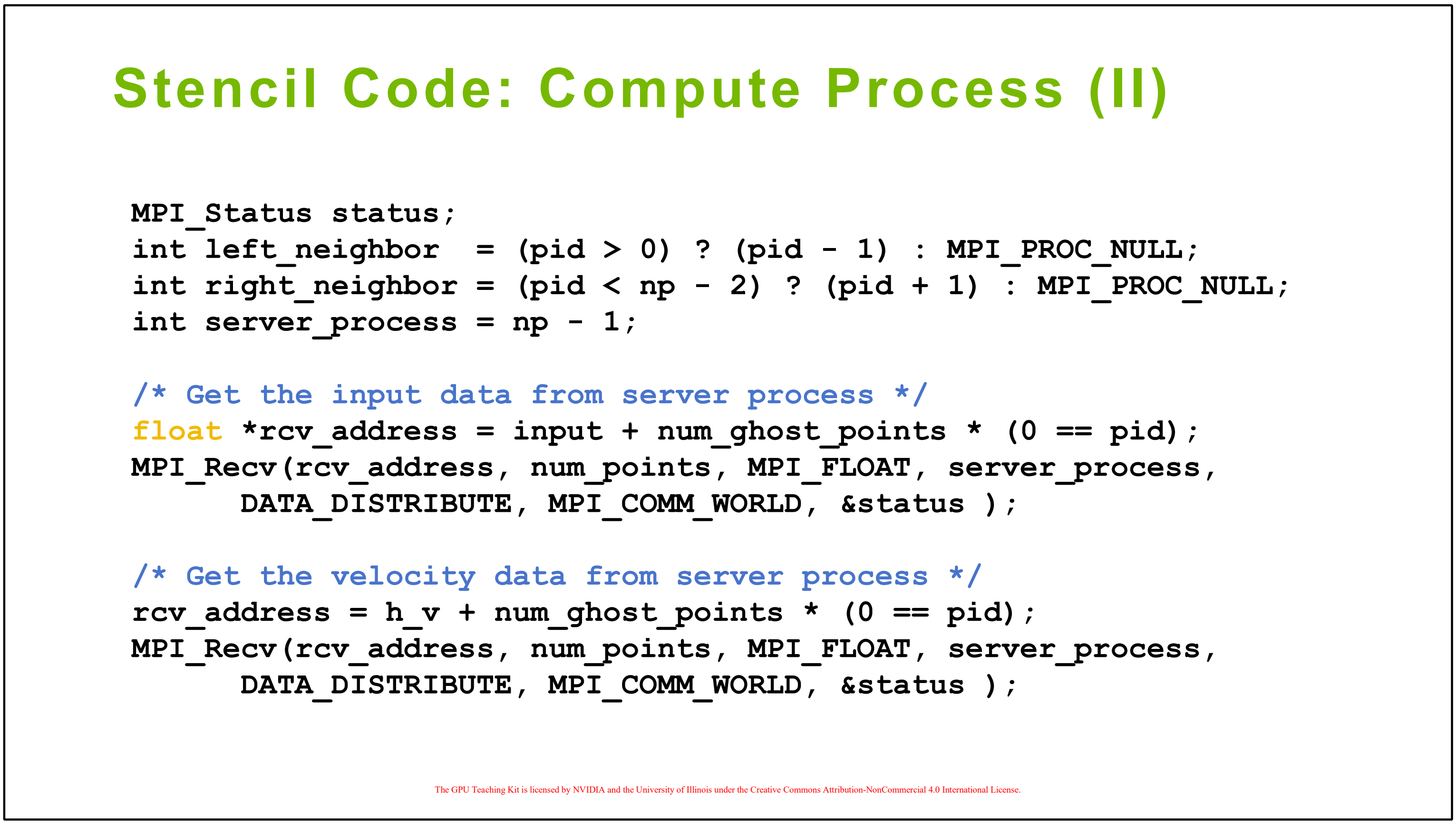
Stencil Code: Server Process (III)

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

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



Stencil Code: Compute Process (I)

```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    unsigned int num_points       = dimx * dimy * (dimz + 8);
    unsigned int num_bytes        = num_points * sizeof(float);
    unsigned int num_ghost_points = 4 * dimx * dimy;
    unsigned int num_ghost_bytes  = num_ghost_points * sizeof(float);

    int left_ghost_offset   = 0;
    int right_ghost_offset  = dimx * dimy * (4 + dimz);

    float *input = NULL, *output = NULL, *prev = NULL, *v = NULL;

    /* Allocate device memory for input and output data */
    gmacMalloc((void **)&input,  num_bytes);
    gmacMalloc((void **)&output, num_bytes);
    gmacMalloc((void **)&prev, num_bytes);
    gmacMalloc((void **)&v, num_bytes);
```



Stencil Code: Compute Process (II)

```
MPI_Status status;
int left_neighbor  = (pid > 0) ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;
int server_process = np - 1;

/* Get the input data from server process */
float *rcv_address = input + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
      DATA_DISTRIBUTE, MPI_COMM_WORLD, &status );

/* Get the velocity data from server process */
rcv_address = h_v + num_ghost_points * (0 == pid);
MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
      DATA_DISTRIBUTE, MPI_COMM_WORLD, &status );
```

讲授内容：Related Programming Models: MPI

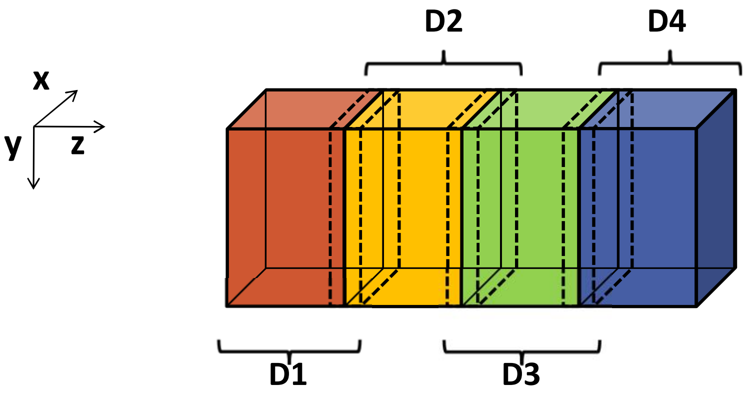
① Warps and SIMD Hardware

② Performance Impact of Control Divergence

③ Overlapping Computation with Communication

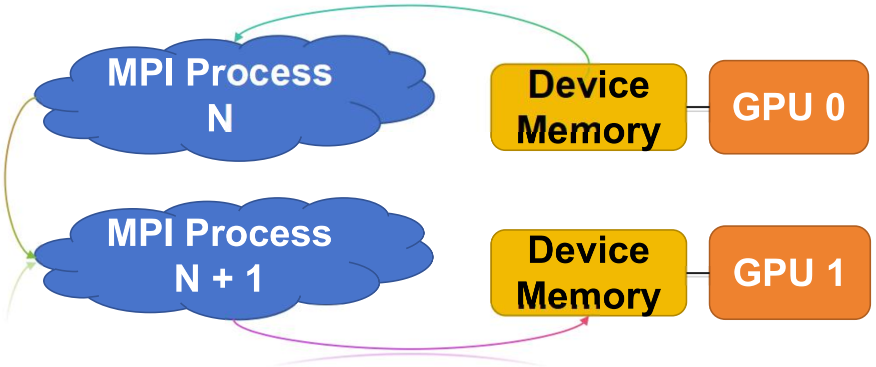
Stencil Domain Decomposition

- Volumes are split into tiles (along the Z-axis)
  - 3D-Stencil introduces data dependencies



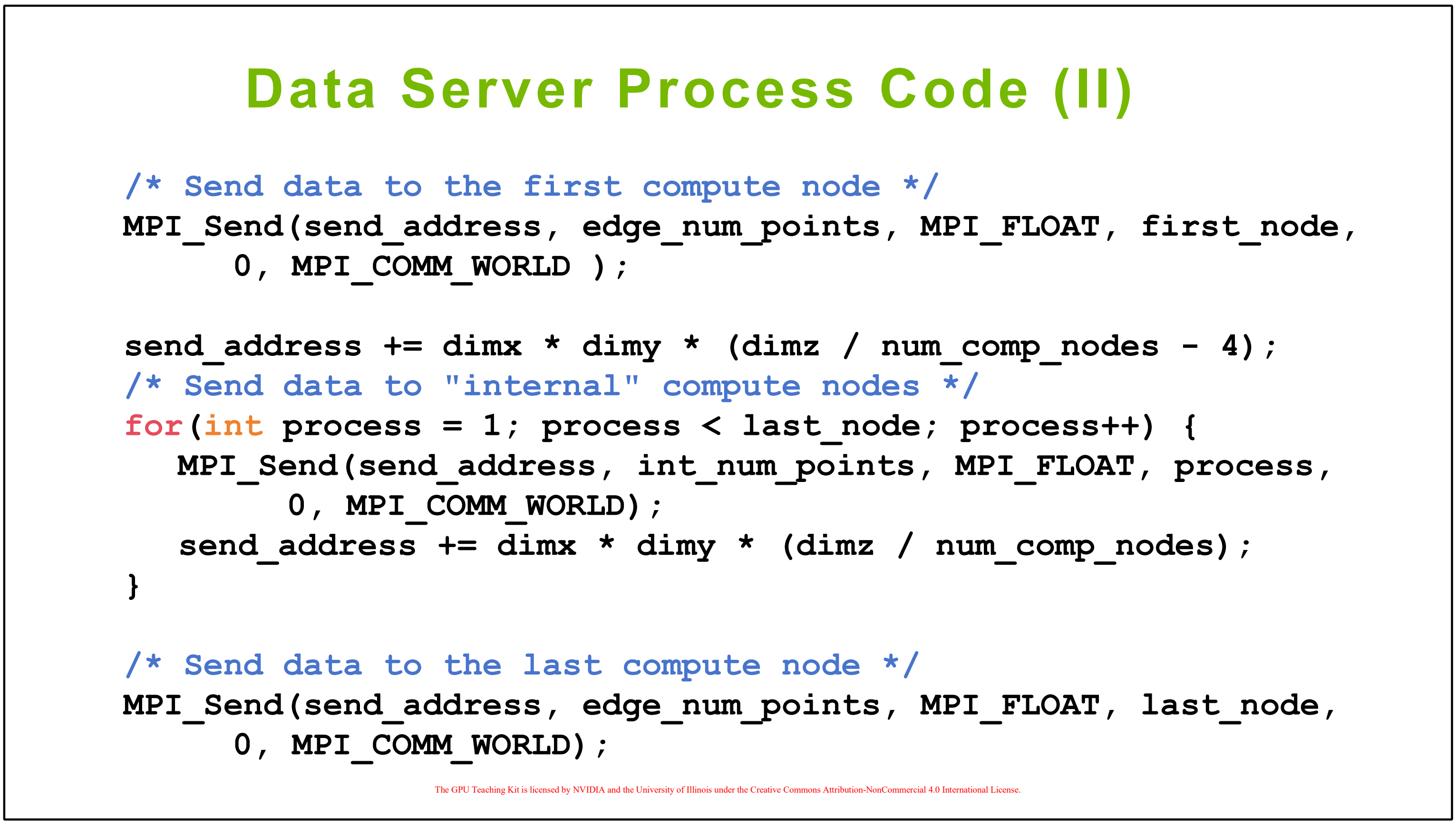
CUDA and MPI Communication

- Source MPI process:
  - cudaMemcpy(tmp,src, cudaMemcpyDeviceToHost)
  - MPI_Send()
- Destination MPI process:
  - MPI_Recv()
  - cudaMemcpy(dst, src, cudaMemcpyDeviceToDevice)



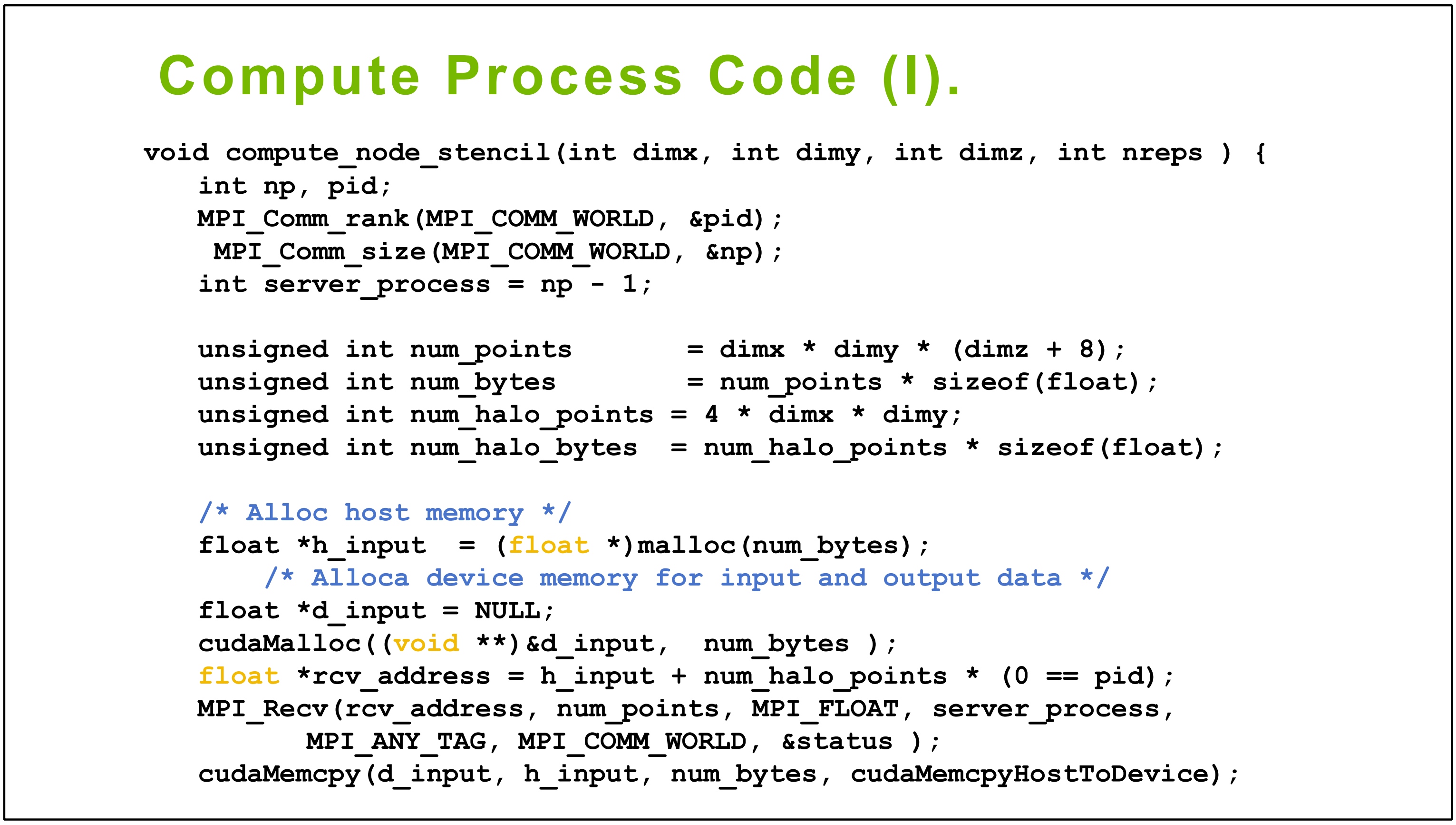
Data Server Process Code (I)

```
void data_server(int dimx, int dimy, int dimz, int nreps) {
    int np,
    /* Set MPI Communication Size */
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    num_comp_nodes = np – 1, first_node = 0, last_node = np - 2;
    unsigned int num_points = dimx * dimy * dimz;
    unsigned int num_bytes  = num_points * sizeof(float);
    float *input=0, *output=0;
        /* Allocate input data */
    input = (float *)malloc(num_bytes);
    output = (float *)malloc(num_bytes);
    if(input == NULL || output == NULL) {
        printf("server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data */
    random_data(input, dimx, dimy ,dimz , 1, 10);
    /* Calculate number of shared points */
    int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
    int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
    float *send_address = input;
```



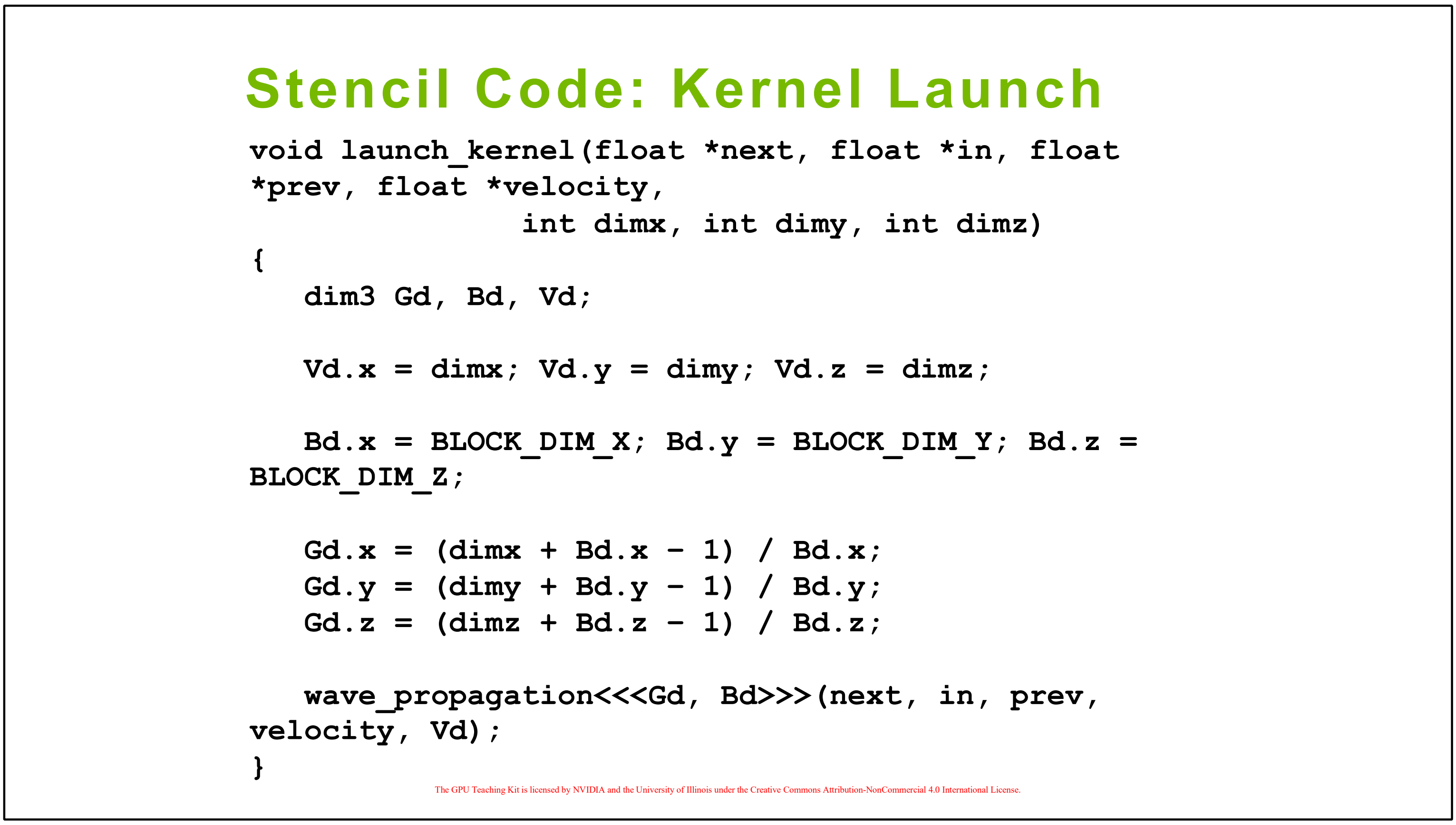
Data Server Process Code (II)

```
/* Send data to the first compute node */
MPI_Send(send_address, edge_num_points, MPI_FLOAT, first_node,
      0, MPI_COMM_WORLD );

send_address += dimx * dimy * (dimz / num_comp_nodes - 4);
/* Send data to "internal" compute nodes */
for(int process = 1; process < last_node; process++) {
   MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
         0, MPI_COMM_WORLD);
   send_address += dimx * dimy * (dimz / num_comp_nodes);
}

/* Send data to the last compute node */
MPI_Send(send_address, edge_num_points, MPI_FLOAT, last_node,
      0, MPI_COMM_WORLD);
```



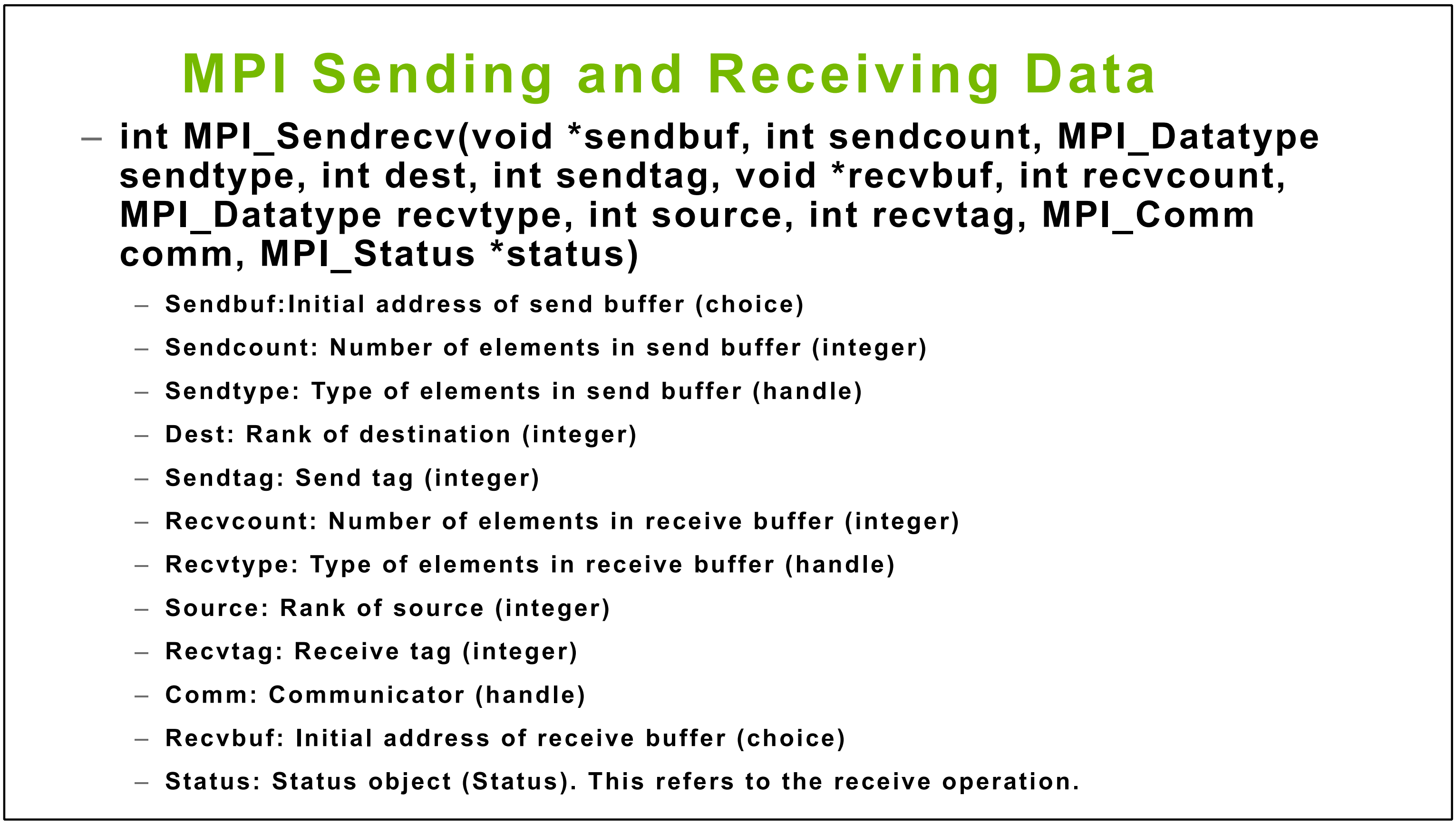
Compute Process Code (I).

```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
   int np, pid;
   MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
   int server_process = np - 1;

   unsigned int num_points       = dimx * dimy * (dimz + 8);
   unsigned int num_bytes        = num_points * sizeof(float);
   unsigned int num_halo_points = 4 * dimx * dimy;
   unsigned int num_halo_bytes  = num_halo_points * sizeof(float);

   /* Alloc host memory */
   float *h_input  = (float *)malloc(num_bytes);
       /* Alloca device memory for input and output data */
   float *d_input = NULL;
   cudaMalloc((void **)&d_input,  num_bytes );
   float *rcv_address = h_input + num_halo_points * (0 == pid);
   MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
         MPI_ANY_TAG, MPI_COMM_WORLD, &status );
   cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);
```



Stencil Code: Kernel Launch

```
void launch_kernel(float *next, float *in, float
*prev, float *velocity,
               int dimx, int dimy, int dimz)
{
   dim3 Gd, Bd, Vd;

   Vd.x = dimx; Vd.y = dimy; Vd.z = dimz;

   Bd.x = BLOCK_DIM_X; Bd.y = BLOCK_DIM_Y; Bd.z =
BLOCK_DIM_Z;

   Gd.x = (dimx + Bd.x – 1) / Bd.x;
   Gd.y = (dimy + Bd.y – 1) / Bd.y;
   Gd.z = (dimz + Bd.z – 1) / Bd.z;

   wave_propagation<<<Gd, Bd>>>(next, in, prev,
velocity, Vd);
}
```



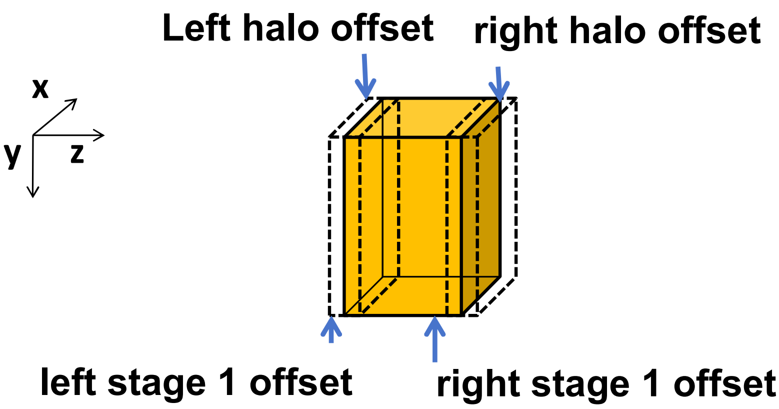
MPI Sending and Receiving Data

– int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
  - Sendbuf:Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object (Status). This refers to the receive operation.

Compute Process Code (II)

```
float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
float *h_output = (float *)malloc(num_bytes);
cudaMalloc((void **)&d_output, num_bytes );

float *h_left_boundary = NULL, *h_right_boundary = NULL;
float *h_left_halo = NULL, *h_right_halo = NULL;

/* Alloc host memory for halo data */
 cudaHostAlloc((void **)&h_left_boundary, num_halo_bytes, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_right_boundary,num_halo_bytes, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_left_halo,     num_halo_bytes, cudaHostAllocDefault);
cudaHostAlloc((void **)&h_right_halo,    num_halo_bytes, cudaHostAllocDefault);

 /* Create streams used for stencil computation */
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);
```

Device Memory Offsets Used for Data Exchange with Neighbors



Left halo offset

right halo offset

left stage 1 offset

right stage 1 offset

Compute Process Code (III)

```
MPI_Status status;
int left_neighbor  = (pid > 0)     ? (pid - 1) : MPI_PROC_NULL;
int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

/* Upload stencil cofficients */
upload_coefficients(coeff, 5);

int left_halo_offset   = 0;
int right_halo_offset  = dimx * dimy * (4 + dimz);
int left_stage1_offset  = 0;
int right_stage1_offset = dimx * dimy * (dimz - 4);
int stage2_offset       = num_halo_points;

MPI_Barrier( MPI_COMM_WORLD );
for(int i=0; i < nreps; i++) {
    /* Compute boundary values needed by other nodes first */
    launch_kernel(d_output + left_stage1_offset,
        d_input + left_stage1_offset, dimx, dimy, 12, stream0);
    launch_kernel(d_output + right_stage1_offset,
        d_input + right_stage1_offset, dimx, dimy, 12, stream0);

    /* Compute the remaining points */
    launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                 dimx, dimy, dimz, stream1);
```

Compute Process Code (IV)

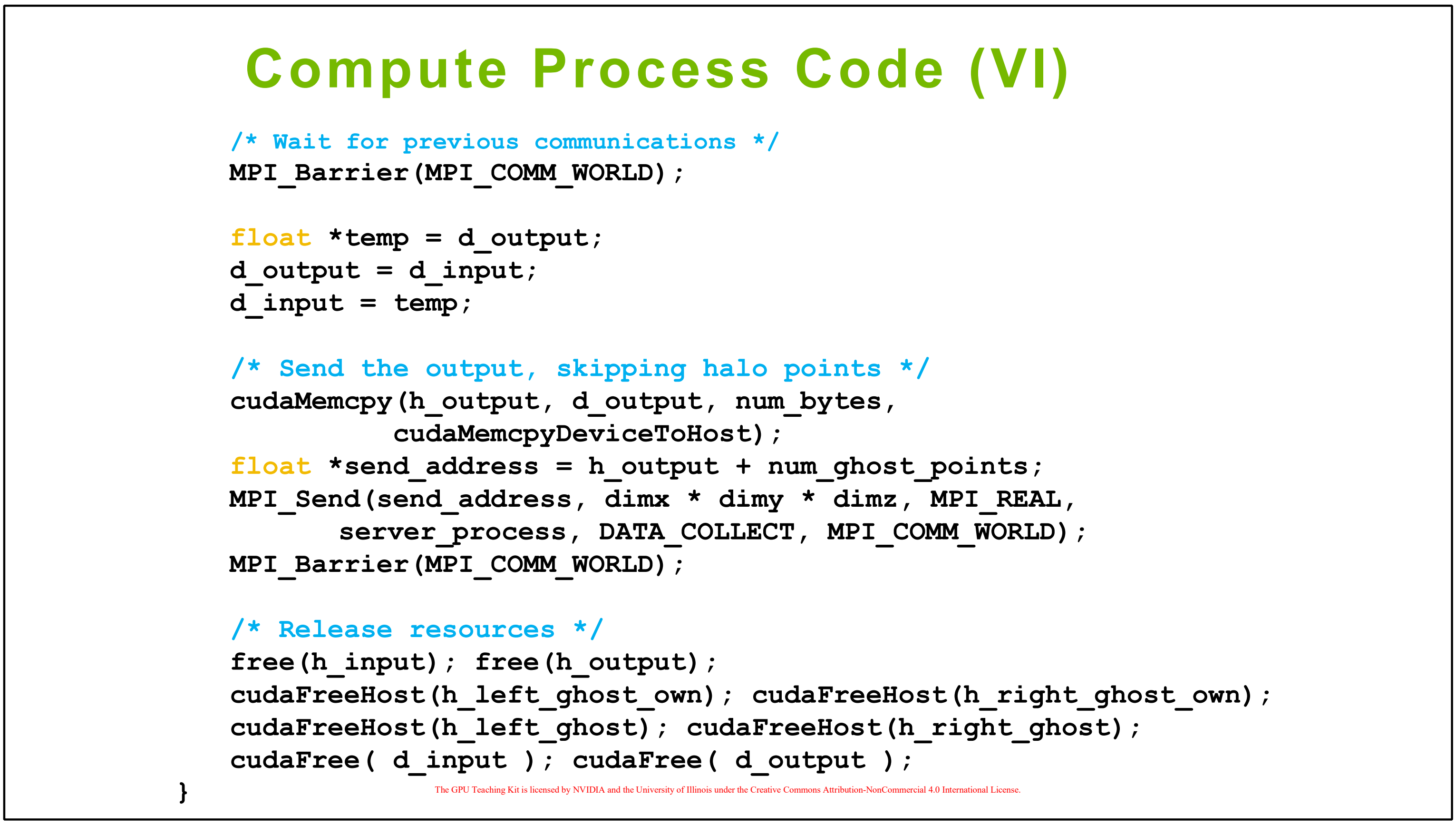
```
    /* Copy the data needed by other nodes to the 
host */
    cudaMemcpyAsync(h_left_boundary, d_output + 
num_halo_points,
            num_halo_bytes, cudaMemcpyDeviceToHost, 
stream0 );
    cudaMemcpyAsync(h_right_boundary,
              d_output + right_stage1_offset + 
num_halo_points,
              num_halo_bytes, cudaMemcpyDeviceToHost, 
stream0 );
    cudaStreamSynchronize(stream0);
```

Syntax for MPI_Sendrecv()

- int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
  - Sendbuf:Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object (Status). This refers to the receive operation.

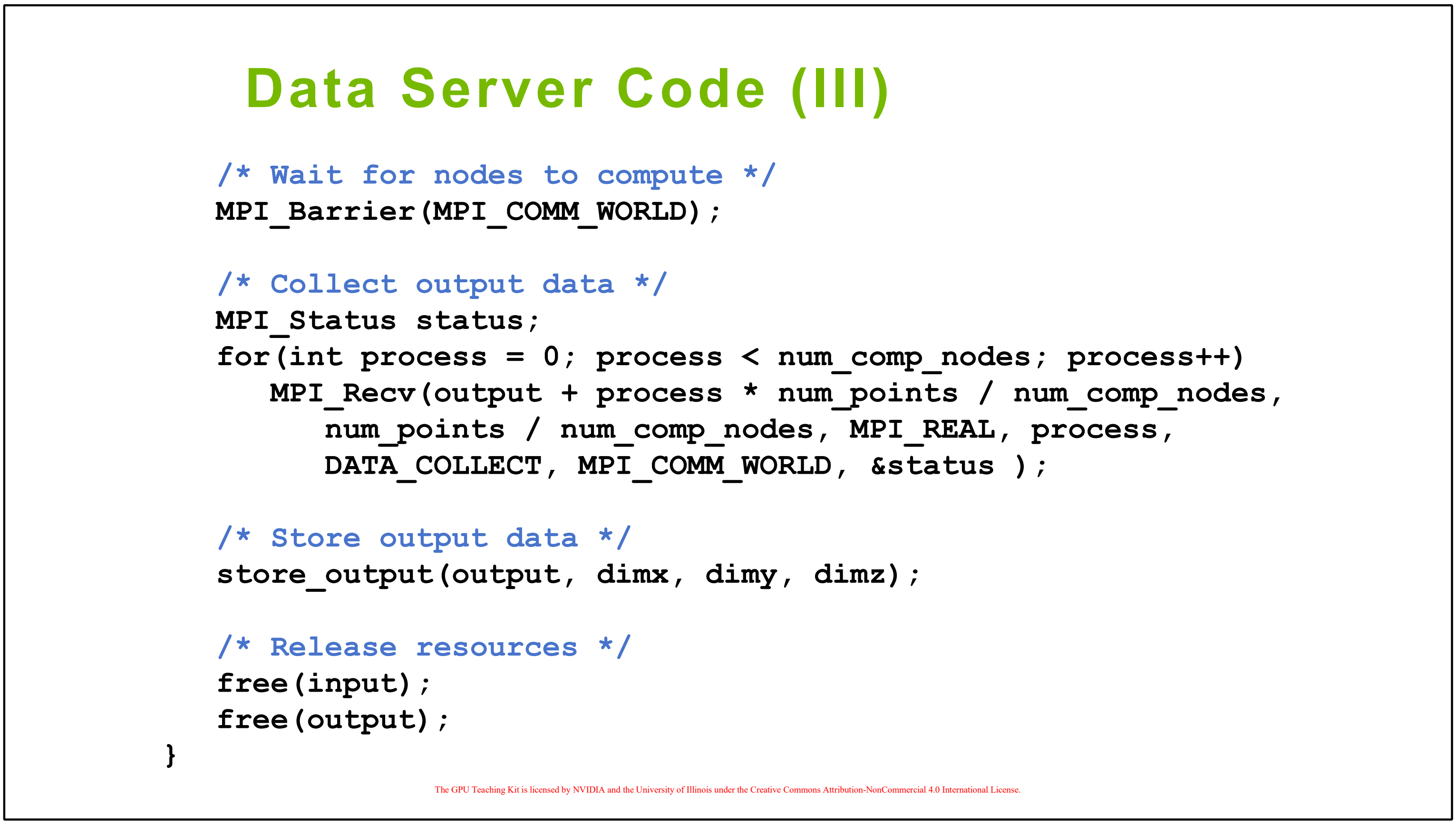
Compute Process Code (V)

```
    /* Send data to left, get data from right */
    MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
             left_neighbor,  i, h_right_halo,
             num_halo_points, MPI_FLOAT, right_neighbor, i,
             MPI_COMM_WORLD, &status );
    /* Send data to right, get data from left */
    MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
             right_neighbor, i, h_left_halo,
             num_halo_points, MPI_FLOAT, left_neighbor,  i,
             MPI_COMM_WORLD, &status );

    cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
             num_halo_bytes, cudaMemcpyHostToDevice, stream0);
    cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
             num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
    cudaDeviceSynchronize();
    float *temp = d_output;
    d_output = d_input; d_input = temp;
}
```



Compute Process Code (VI)

```
/* Wait for previous communications */
MPI_Barrier(MPI_COMM_WORLD);

float *temp = d_output;
d_output = d_input;
d_input = temp;

/* Send the output, skipping halo points */
cudaMemcpy(h_output, d_output, num_bytes,
          cudaMemcpyDeviceToHost);
float *send_address = h_output + num_ghost_points;
MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
       server_process, DATA_COLLECT, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

/* Release resources */
free(h_input); free(h_output);
cudaFreeHost(h_left_ghost_own); cudaFreeHost(h_right_ghost_own);
cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
cudaFree( d_input ); cudaFree( d_output );
}
```



Data Server Code (III)

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

   /* Collect output data */
   MPI_Status status;
   for(int process = 0; process < num_comp_nodes; process++)
      MPI_Recv(output + process * num_points / num_comp_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);
}
```

More on MPI Message Types

- Point-to-point communication
  - Send and Receive
- Collective communication
  - Barrier
  - Broadcast
  - Reduce
  - Gather and Scatter