CS/EE 217

GPU Architecture and Parallel Programming

Lecture 23: Introduction to OpenACC

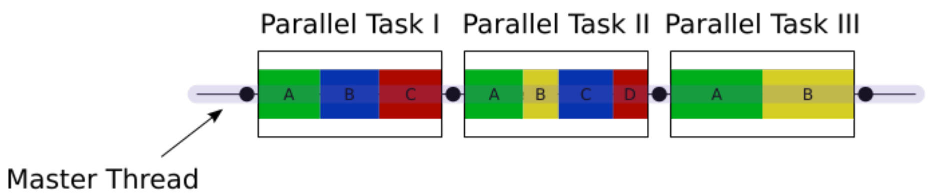
Objective

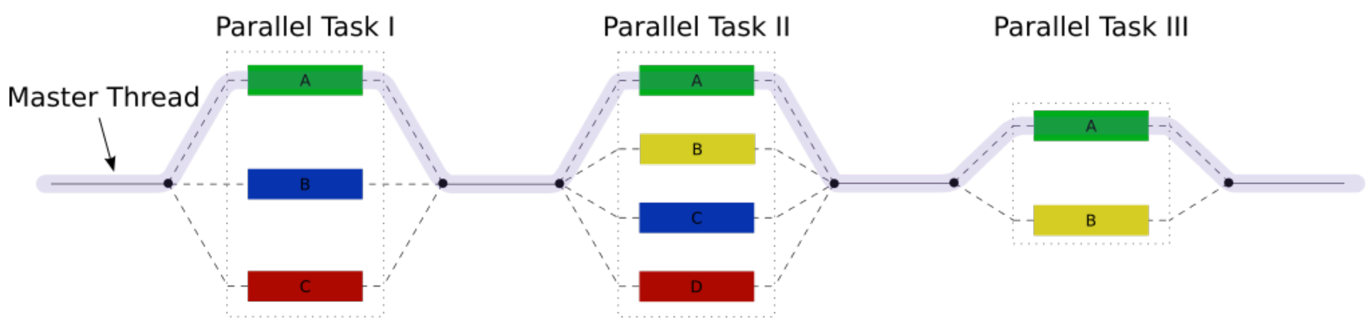
- To Understand the OpenACC programming model
  - basic concepts and pragmatypes
  - Simple examples to illustrate basic concepts and functionalities

OpenMP and 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.
- A sequential compiler can just ignore the pragmas to produce sequential code
  - If you are careful

OpenACC extends OpenMP





- OpenMP is a shared memory parallel programming API
- It uses pragmas (or compiler directives) to specify parallel regions within a program
- OpenACC extends openMP to allow some of the code to run on GPUs/acclerators
  - Also using pragmas

Example of OpenMP

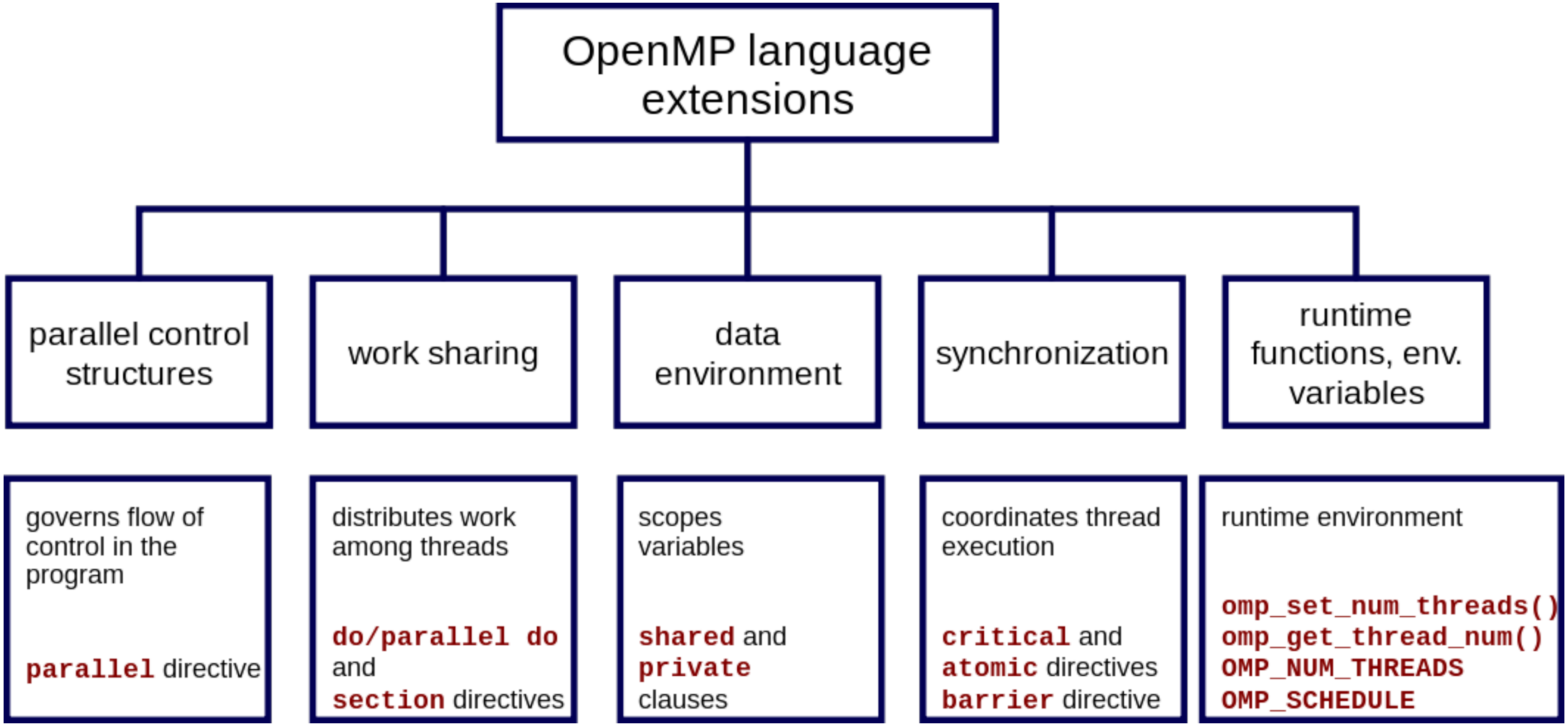
```c
#include <stdio.h>

int main(void) {

    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

- To compile: gcc -fopenmp hello.c -o hello
- What will the output be?

More interesting example

```
int main(int argc, char **argv)
   {
   int a[100000];
   #pragma omp parallel for
   int i;
   for (i = 0; i < 100000; i++)
         a[i] = 2 * i;
   return 0;
   }
```

Summary of OpenACC directives



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

Simple Matrix-Matrix Multiplication in OpenACC

```
void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
{

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

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 the compiler directive mechanism 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

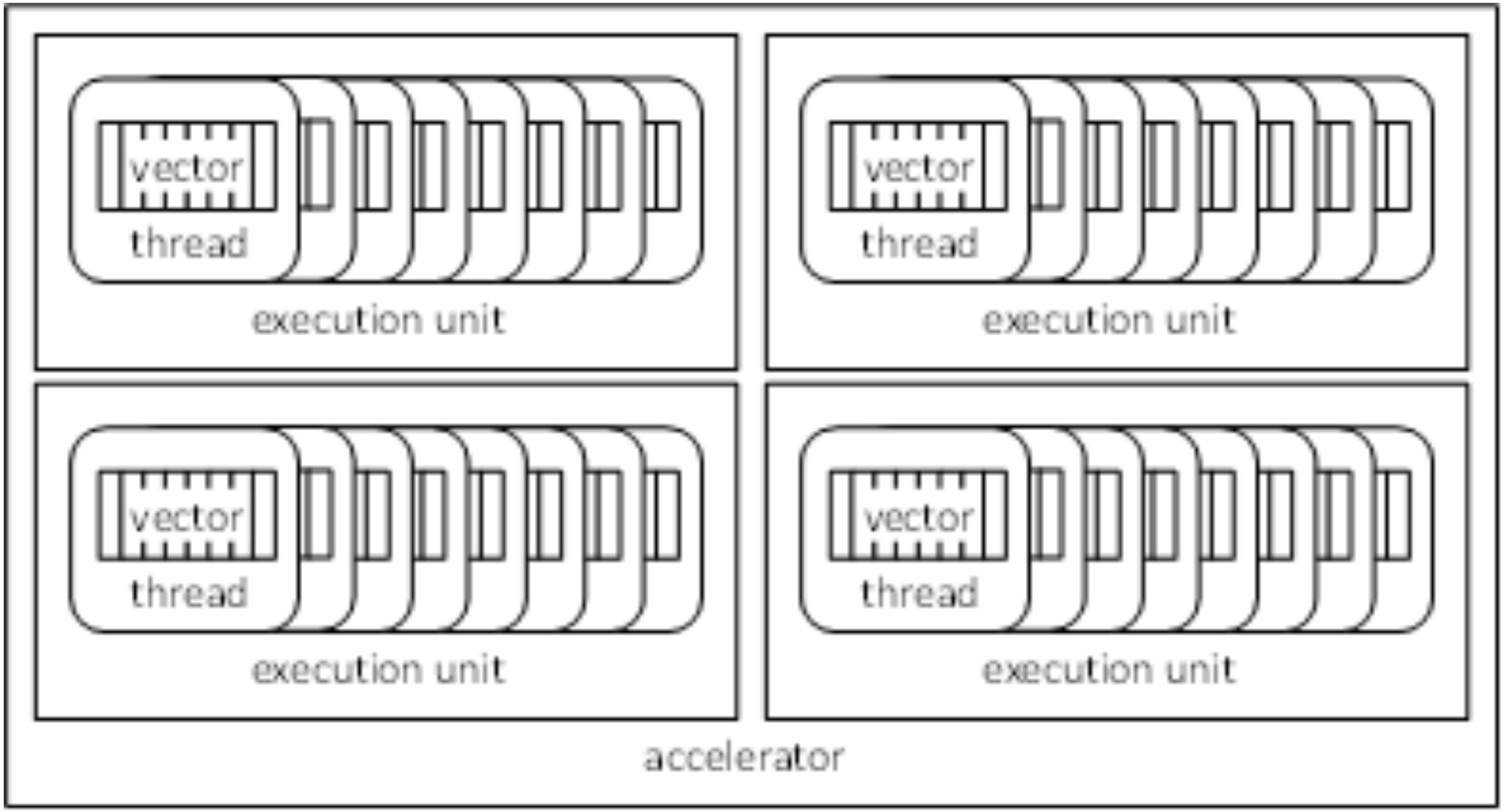
- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
  - leave most of the details in generating a kernel and data transfers to the OpenACC compiler.

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

Frequently Encountered Issues

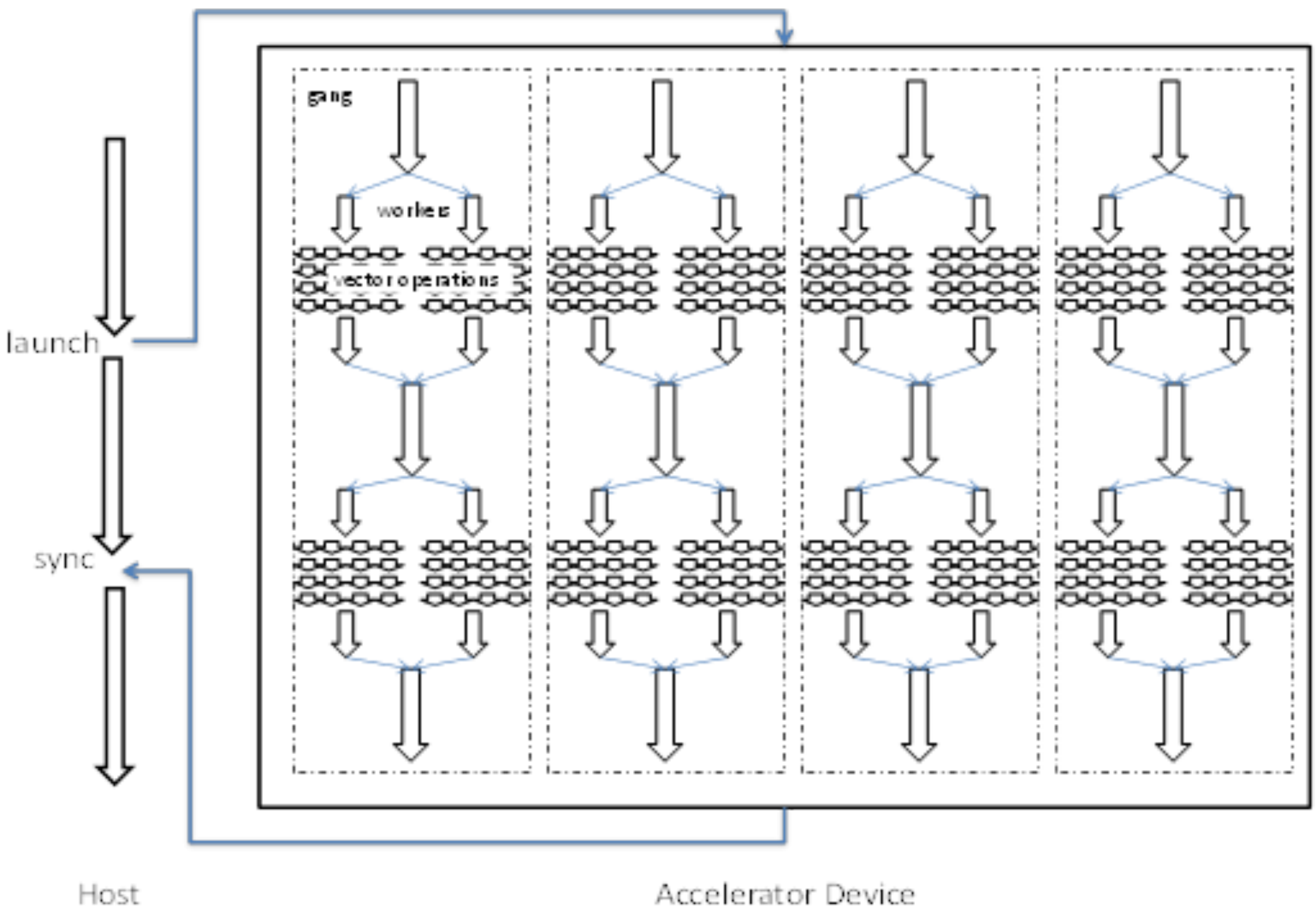
- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC depends heavily on the quality of the compiler.
  - Much less so in CUDA or OpenCL
- Some OpenACC programs may behave differently or even incorrectly if pragmas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

OpenACC Device Model



Currently OpenACC does not allow synchronization across threads.

OpenACC Execution Model



Parallel vs. Loop Constructs

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

is equivalent to:

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

(a parallel region that consists of just a loop)

Parallel Construct

- A parallel construct is executed on an accelerator
- One can specify the number of gangs and number of works in each gang

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

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

What does each “Gang Loop” do?

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

```
#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()

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

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

Kernel Regions

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

- Kernel constructs are descriptive of programmer intentions

ANY MORE QUESTIONS?
READ CHAPTER 15