GPU Teaching Kit

Accelerated Computing

Lecture 21.1 - Related Programming Models: OpenACC

Introduction to OpenACC

Objective

- To understand the OpenACC programming model
  - basic concepts and pragma types
  - simple examples

NVIDIA ILLINOIS

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

NVIDIA ILLINOIS

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

NVIDIA ILLINOIS

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:inputLength]),
      copyout(output[0:inputLength])
   for(i = 0; i < inputLength; ++i) {
      output[i] = input1[i] + input2[i];
   }
}
```


NVIDIA ILLINOIS

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.  #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. }
```


NVIDIA ILLINOIS

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.

NVIDIA ILLINOIS

Some Observations (3)

```
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 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.

NVIDIA ILLINOIS

Some Observations (4)

```
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 5 instructs the compiler to map the inner 'j' loop to the second level of parallelism on the accelerator.

NVIDIA ILLINOIS

Motivation

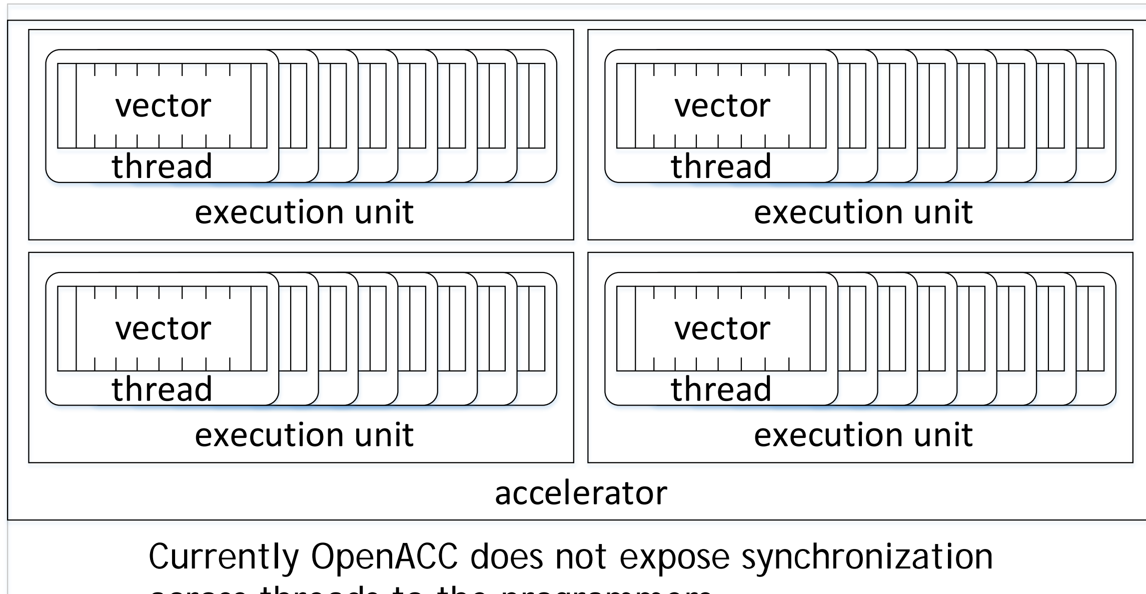
- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
  - leave most of the details in generating a kernel, memory allocation, and data transfers to the OpenACC compiler.

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

Frequently Encountered Issues

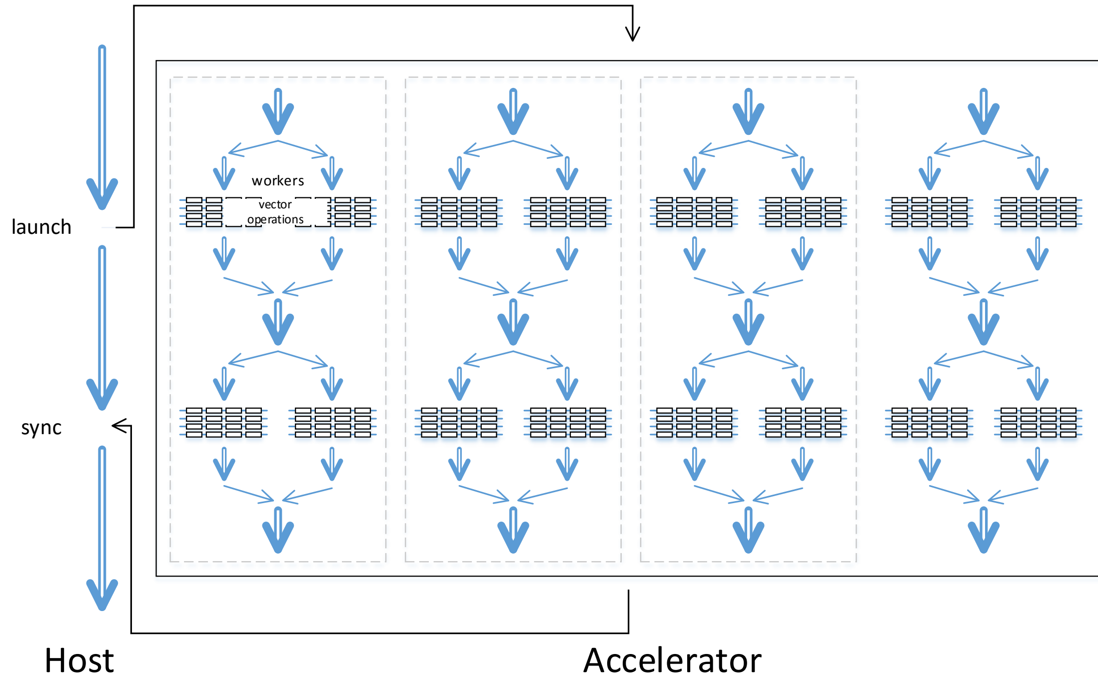
- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC program depends heavily on the quality of the compiler.
  - It may be hard to figure out why the compiler cannot act according to your hints
  - The uncertainty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

OpenACC Device Model



NVIDIA ILLINOIS

OpenACC Execution Model



NVIDIA
ILLINOIS

GPU Teaching Kit

The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.