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

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:inputLength]),
       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)

```
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.

Motivation

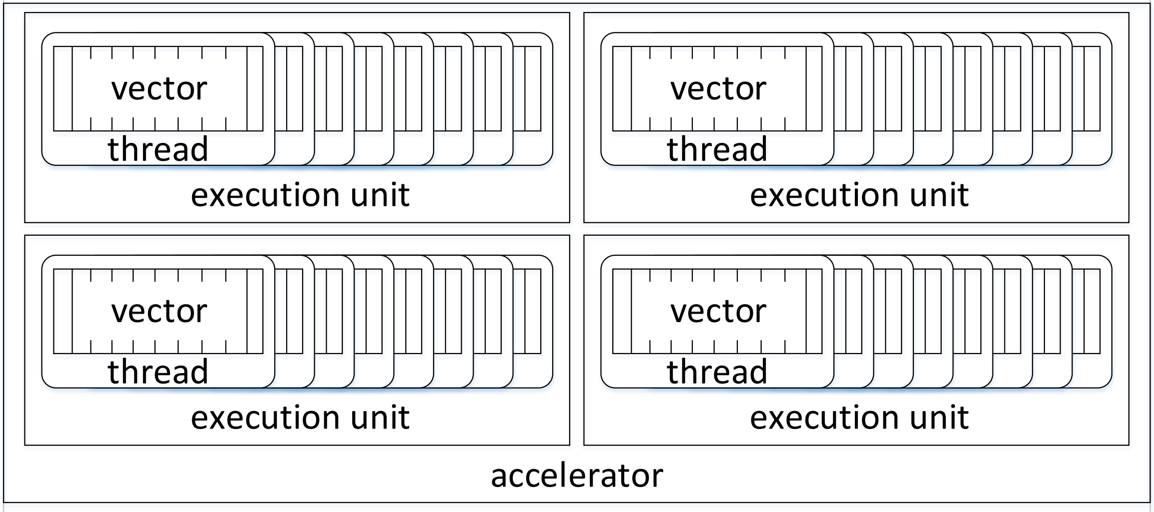
– OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
  – leave most of the details in generating a kernel, memory allocation, and data transfers to the OpenACC compiler.

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

Frequently Encountered Issues

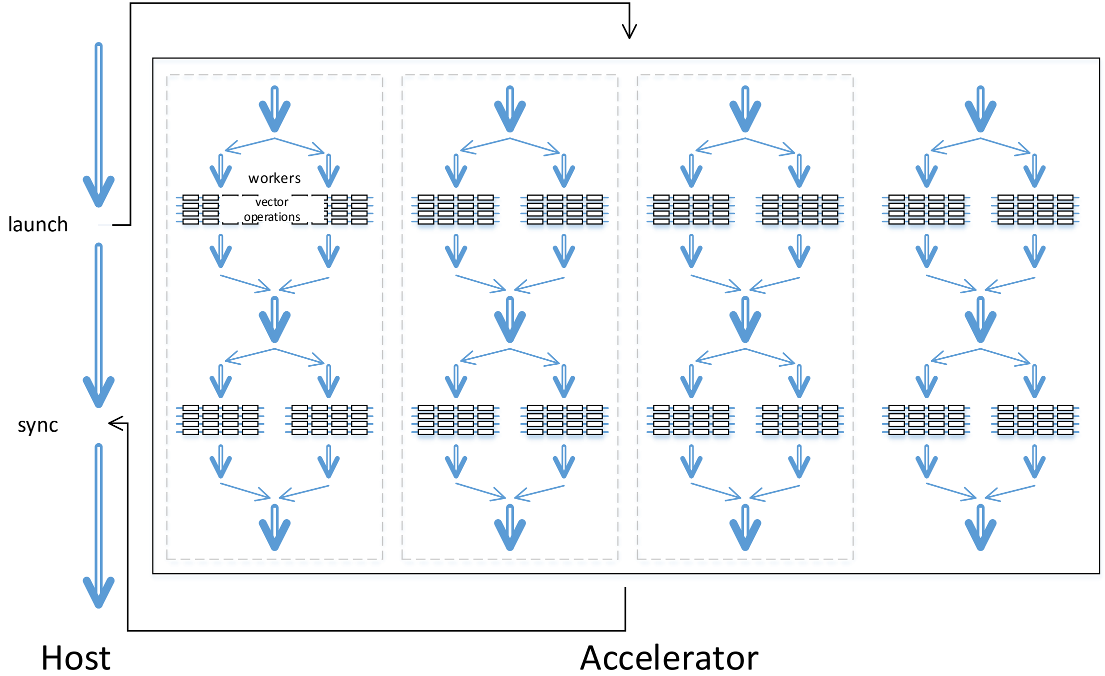
- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC program depends heavily on the quality of the compiler.
  - It may be hard to figure out why the compiler cannot act according to your hints
  - The uncertainty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

OpenACC Device Model



Currently OpenACC does not expose synchronization across threads to the programmers.

OpenACC Execution Model



GPU Teaching Kit

GPU Teaching Kit

Accelerated Computing

Lecture 21.2 - Related Programming Models: OpenACC

OpenACC Subtleties

Objective

- To understand some important and sometimes subtle details in OpenACC programming
  - parallel loops
  - simple examples to illustrate basic concepts and functionalities

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;
    }
    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

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

GPU Teaching Kit