Objective

• To understand the OpenACC, a high-level programming model for CPU/GPU
  – basic concepts and pragma types
  – Simple examples to illustrate basic concepts and functionalities
  – A valuable tool to quickly adopt GPU code in large, existing C/C++/FORTRAN applications

OpenACC

• The OpenACC Application Programming Interface (API) 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 means to provide, to the compiler, information that is not specified in the standard language.
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 user-specified synchronization across threads.

OpenACC Execution Model

(Terminology: Gangs and Works)

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
• Programmer’s directive

```
#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++) {

  …
  }

}
```

The for-loop will be redundantly executed by 1024 gangs
The 2048 iterations of the for-loop will be divided among 1024 gangs for execution

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 Complex Example

```
#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

```c
#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 (Suggestions)

C/C++ vs. FORTRAN

```c
// C or C++
#pragma acc <directive> <clauses>
{ … }

! Fortran
$acc <directive> <clauses>
…
$acc end <directive>
```

ANY MORE QUESTIONS?
READ CHAPTER 15