Objective

- to understand OpenACC (see openacc.org)
  - a directive-based programming model for heterogeneous platforms
  - a valuable tool to quickly adapt existing C/C++/FORTRAN applications to GPUs
- basic concepts and pragma types
- simple examples to illustrate basic concepts and functionalities

OpenACC

The OpenACC Application Programming Interface (API) provides a set of
- compiler directives (pragmas),
- library routines, and
- environment variables
that enable
- FORTRAN, C and C++ programs
- to execute on accelerator devices
- including GPUs and CPUs.

Pragmas Provide Extra Information

In C and C++,
- the `#pragma` directive
- provides the compiler with
- information not specified in the language.

For OpenACC, they look like this:
`#pragma acc [ the information goes here ]`
The OpenACC Abstract Machine Model

Simple Matrix-Matrix Multiplication in OpenACC

Add Pragmas to Sequential Code

The code is
• identical to the sequential version
• except for the two pragmas
• at lines 4 and 6.

OpenACC uses the compiler directive mechanism to extend the base language.
Simple Matrix-Matrix Multiplication in OpenACC

```c
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;
        }
    }
}
```

Motivating Goal: One Version of Code

OpenACC programmers
- can often start with a sequential version,
- then annotate their program with directives,
- leaving most kernel details and data transfers
  - to the OpenACC compiler.

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

Reality is More Complicated

Reality check:
- can be difficult to write code
- that works correctly and well
  - with and without pragmas.

Some OpenACC programs
- behave differently or even incorrectly
  - if pragmas are ignored.

Pitfall: Strong Dependence on Compiler

Some OpenACC pragmas
- are hints to the OpenACC compiler,
  - which may or may not be able to act accordingly

Performance depends heavily
- on the quality of the compiler
  - (more so than with CUDA or OpenCL).
Currently OpenACC does not allow user-specified synchronization across threads.

OpenACC Execution Model (Terminology: Gangs and Works)

Parallel vs. Loop Constructs

```c
#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:

```c
#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

```c
#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++) {
        ...
    }
}
```

The for-loop will be redundantly executed by 1024 gangs.

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

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 instances 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 j=0; j<m; j++) {
        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] = e[k];
    }
}
```

- Kernel constructs are descriptive of programmer intentions (suggestions).
C/C++ vs. FORTRAN

// C or C++
#pragma acc <directive> <clauses>
{ ... }

! Fortran
!$acc <directive> <clauses>

!$acc end <directive>

ANY MORE QUESTIONS?
READ CHAPTER 15