CSCI-GA.3033-004
Graphics Processing Units (GPUs): Architecture and Programming

Lecture : OpenACC

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides for this lecture are adopted (and slightly edited) from
• David Kirk and Wei-mei W. Hwu
What is 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.

http://www.openacc.org/
What is OpenACC?

- Initially developed by Portland Group (PGI), CRAY, NVIDIA with support from CAPS enterprise
- Announced at the Supercomputing Conference (SC11), Nov 2011.
The Main Strategy

OpenACC is based on programmers inserting hints into their programs on how the code is to be parallelized.

The compiler runs the code on the hardware platform that is specified at the time of compilation.
In C and C++: the #pragma directive is: the method to provide, to the compiler, information that is not specified in the standard language.
MatrixMultiplication

```c
1  void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw) {
2  
3 
4 
5   for (int i=0; i<Mh; i++) {
6     
7       for (int j=0; j<Nw; j++) {
8         float sum = 0;
9         for (int k=0; k<Mw; k++) {
10            float a = M[i*Mw+k];
11            float b = N[k*Nw+j];
12            sum += a*b;
13         }
14         P[i*Nw+j] = sum;
15     }
16   }
17 }
```
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;
        }
    }
}
```
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;
        }
    }
}
```

The `copyin` clause and the `copyout` clause specify how the matrix data should be transferred between the host and the accelerator. The `parallel loop` means the ‘i’ loop is mapped to the 1st level of parallelism on the accelerator. The inner ‘j’ loop is mapped to the 2nd 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 Directives

`#pragma acc <directive> [clause ...]`

Directive is applied to the immediately following statement

Directive + statement = OpenACC Construct
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
Currently OpenACC does not allow synchronization across threads.
OpenACC Execution Model
OpenACC has two main directives

• **Compute directives**
  – Marks block of code to be accelerated

• **Data management directives**
  – For data movement
  – Can be used within compute directives
Compute Directives:
Two main constructs

Parallel Construct
The programmer does most of the work.

Kernels Construct
• Tells the compiler that this region should be placed on the accelerator.
• The compiler does most of the heavy lifting.

Loop Construct
Tells the compiler that loop iterations are independent.
Parallel Construct

```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.
• Parallel directive by itself is relatively useless.
• One can specify the number of gangs and number of workers in each gang.

```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.
Gangs Loop

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

One worker within each gang will execute the parallel region. So, the 2048 iterations will be executed redundantly and sequentially by 1024 gang leaders.

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

The 2048 loop iterations will be distributed among the 1024 gangs. Each gang leader will execute 2 iterations.
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;
}
#pragma acc parallel num_gangs(1)
num_workers(32)
{
    Statement 1;
    Statement 2;
    #pragma acc loop worker
    for (int i=0; i<n; i++) {
        Statement 3;
        Statement 4;
    }
    Statement 5;
    Statement 6;
    #pragma acc loop worker
    for (int i=0; i<m; i++) {
        Statement 7;
        Statement 8;
    }
    Statement 9;
    if (condition)
        Statement 10;
}
Multiple level of Parallelism

```c
{
    for(int i =0; i < 2048; i++){
        for( j = 0; j < 512; j++){
            for( k = 0; k < 1024; k++) {
                foo(i, j, k);
            }
        }
    }
}
```
Multiple level of Parallelism

```c
#pragma acc parallel num_gangs(1024) num_workers(32) vector_length(32)
{
    #pragma acc loop gang
    for(int i =0; i < 2048; i++){
        #pragma acc loop worker
        for( j = 0; j < 512; j++){
            #pragma acc loop vector
            for( k = 0; k < 1024; k++) {
                foo(i, j, k);
            }
        }
    }
}
```
Kernel Constructs

```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 tells the compiler to execute the code in the accelerator if possible.
- Kernel region may be broken into a series of kernels, each of which executed on the accelerator.
Example

```c
int a[n][m], b[n][m], c[n][m];
...

#pragma acc kernels
for(int j = 0; j < n; j++) {
    for(int k = 0; k < m; k++) {
        c[j][k] = a[j][k];
        a[j][k] = c[j][k] + n[j][k];
    }
}

Compiler is free to parallelize the code in any way it sees, or run it sequentially, to ensure correctness.
```
int a[n][m], b[n][m], c[n][m];
...
#pragma acc kernels
{
    for(int j = 0; j < n; j++) {
        for(int k = 0; k < m; k++) {
            c[j][k] = a[j][k];
            a[j][k] = c[j][k] + n[j][k];
        }
    }
}
for(int j = 0; j < n; j++) {
    for(int k = 0; k < m; k++) {
        d[j][k] = a[j][k] - 5;
    }
}
int a[n][m], b[n][m], c[n][m];
...
#pragma acc kernels
{
    for(int j = 0; j < n; j++) {
        for(int k = 0; k < m; k++) {
            c[j][k] = a[j][k];
            a[j][k] = c[j][k] + n[j][k];
        }
    }
}

for(int j = 0; j < n; j++) {
    for(int k = 0; k < m; k++) {
        d[j][k] = a[j][k] - 5;
    }
}
}

int a[n][m], b[n][m], c[n][m];
...
#pragma acc parallel loop
for(int j = 0; j < n; j++) {
    for(int k = 0; k < m; k++) {
        c[j][k] = a[j][k];
        a[j][k] = c[j][k] + n[j][k];
    }
}

#pragma acc parallel loop
for(int j = 0; j < n; j++) {
    for(int k = 0; k < m; k++) {
        d[j][k] = a[j][k] - 5;
    }
}
void foo(int * x, int * y, int n, int m){
    int a[2048], b[2048];

    #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
    {
        //no data dependence
        #pragma acc loop
        for( int i =0; i < 2047; i++){
            a[i] = b[i+1] + 1;
        }

        //data dependence
        #pragma acc loop
        for( int j =0; j < 2047; j++){
            a[j] = a[j+1] + 1;
        }

        // Data dependence if x[] is not aliased with y[]
        #pragma acc loop
        for( int k =0; k < 2047; k++){
            x[i] = y[i+1] + 1;
        }

        //no data dependence if n >= m
        #pragma acc loop
        for( int l =0; l < m; l++){
            x[l] = x[l+n] + 1;
        }
    }
void foo(int * x, int * y, int n, int m){
    int a[2048], b[2048];

    #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
    {
        //no data dependence
        #pragma acc loop
        for( int i =0; i < 2047; i++){
            a[i] = b[i+1] + 1;
        }

        //data dependence
        #pragma acc loop
        for( int j =0; j < 2047; j++){
            a[j] = a[j+1] + 1;
        }

        // Data dependence if x[] is not aliased with y[]
        #pragma acc loop
        for( int k =0; k < 2047; k++){
            x[i] = y[i+1] + 1;
        }

        //no data dependence if n >= m
        #pragma acc loop
        for( int l =0; l < m; l++){
            x[l] = x[l+n] + 1;
        }
    }
}

OpenACC compiler has no problem parallelizing this loop.
void foo(int * x, int * y, int n, int m){
    int a[2048], b[2048];

    #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
    {
        //no data dependence
        #pragma acc loop
        for( int i =0; i < 2047; i++){
            a[i] = b[i+1] + 1;
        }
    }

    //data dependence
    #pragma acc loop
    for( int j =0; j < 2047; j++){
        a[j] = a[j+1] + 1;
    }

    // Data dependence if x[] is not aliased with y[]
    #pragma acc loop
    for( int k =0; k < 2047; k++){
        x[i] = y[i+1] + 1;
    }

    //no data dependence if n >= m
    #pragma acc loop
    for( int l =0; l < m; l++){
        x[l] = x[l+n] + 1;
    }
}

OpenACC compiler has no problem deciding that this loop is not parallelizable.
void foo(int * x, int * y, int n, int m){
    int a[2048], b[2048];

    #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
    {
        //no data dependence
        #pragma acc loop
        for( int i =0; i < 2047; i++){
            a[i] = b[i+1] + 1;
        }

        //data dependence
        #pragma acc loop
        for( int j =0; j < 2047; j++)
        {
            a[j] = a[j+1] + 1;
        }

        // Data dependence if x[] is not aliased with y[]
        #pragma acc loop
        for( int k =0; k < 2047; k++){
            x[i] = y[i+1] + 1;
        }

        //no data dependence if n >= m
        #pragma acc loop
        for( int l =0; l < m; l++)
        {
            x[l] = x[l+n] + 1;
        }
    }
}

The compiler will take the conservative approach and not parallelize this loop. If you are sure that x[] and y[] are not aliased then use:

    foo( int * restricted x, int * restricted y, ....)
void foo(int * x, int * y, int n, int m){
    int a[2048], b[2048];

    #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
    {
        // no data dependence
        #pragma acc loop
        for( int i =0; i < 2047; i++){
            a[i] = b[i+1] + 1;
        }

        // data dependence
        #pragma acc loop
        for( int j =0; j < 2047; j++){
            a[j] = a[j+1] + 1;
        }

        // no data dependence if x[] is not aliased with y[]
        #pragma acc loop
        for( int k =0; k < 2047; k++){
            x[i] = y[i+1] + 1;
        }

        // no data dependence if n >= m
        #pragma acc loop
        for( int l =0; l < m; l++){
            x[l] = x[l+n] + 1;
        }
    }
}

The compiler will take the conservative approach and not parallelize this loop.

If you are sure that it can be parallelized, then use:

#pragma acc loop independent
Some Words About Data Constructs

- Loop control variable is **private** to each thread.
- **copyin**: to the device
- **copyout**: from the device
- **copy**: both and automatically
- **Example**: `#pragma acc data copy(array[i:n])`
  - `i`: the start index (can be anything)
  - `n`: number of elements to be copied
Some Words About Data Constructs

```c
#pragma acc loop
for(int j = 0; j < m; j++)
#pragma acc cache (b[j])
  b[j] = b[j] *c;
```

Tells the compiler that each iteration of the for-loop uses one element of array b. The compiler will try to move n elements to fast memory.
Is this Code Correct?

```c
#pragma acc parallel{
    #pragma acc loop
    for(int i = 0; i < 1000; i++){
        a[i] = b[i];
    }

    #pragma acc loop
    for(int i = 0; i < 1000; i++){
        b[i] = b[i] * 2;
        c[i] = n[i] + a[i];
    }
}
```
Is this Code Correct?

```c
#pragma acc kernels{
    #pragma acc loop
    for(int i = 0; i < 1000; i++){
        a[i] = b[i];
    }

    #pragma acc loop
    for(int i = 0; i < 1000; i++){
        b[i] = b[i] * 2;
        c[i] = n[i] + a[i];
    }
}
```
module load pgi

For GPU:
pgcc -acc -Minfo filename.c

For multicore:
pgcc -acc -ta=multicore -Minfo filename.c
Conclusions

• OpenACC is easy to learn and gets you to a fast start to use an accelerators.
• Directives on top of C, C++, and Fortran
• Compared with CUDA, OpenACC gives you less control of how the final code on the accelerator will be.
• OpenACC can be used fairly fine with CUDA and its libraries.