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.
• In C and C++: the #pragma directive is:

the method to provide, to the compiler, information that is not specified in the standard language.
Matrix Multiplication

```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     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      }
14     P[i*Nw+j] = sum;
15   }
16 }
17 }
```
Matrix Multiplication in OpenACC

```c
1 void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw) {
2  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw])
3    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 ```
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 1\textsuperscript{st} level of parallelism on the accelerator.

Instructs the compiler to map the inner ‘j’ loop on the 2\textsuperscript{nd} 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
Currently OpenACC does not allow synchronization across threads.
OpenACC Execution Model
OpenACC has two main constructs

- Parallel Construct
- Kernels Construct
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

• 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
define num_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
define num_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

```c
#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;
}

- Statements 1 and 2 are redundantly executed by 32 gang leaders (32)
- The n for-loop iterations are distributed to 32 gangs, each gang will distribute its share to a number of workers.
- Number of workers is determined by compiler/runtime.

What if statements: 1, 2, 5, 6, 9, and 10 must be executed only once for the correctness of the program?
#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 are descriptive of programmer intentions
- Kernel region may be broken into a series of kernels, each of which executed on the accelerator.
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];

    // no data dependence
    #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;
        }

        // 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
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.