Accelerate Your Science: An Introduction to High Performance Computing
Lecture 4: Introduction to OpenACC

Guido Juckeland
Visiting Scholar
Technische Universität Dresden, Germany

July 8, 2014
CPUs – flexible, low latency

http://www.nvidia.com/content/nvision2008/art_science/index.html
GPUs – massively parallel

http://www.nvidia.com/content/nvision2008/art_science/index.html
Accelerator system setup

- CPU
- Main Memory
- Accelerator
- Local Memory
- Application
- Accelerator Library
- Subprogram

Connections:
- System Bus
- DMA Transfers
- Calls to Accelerator Library
- Invokes Subprogram
Defining terms: HOST and DEVICE

HOST

DEVICE
The five results of parallel programming

- Compiler error
- Program crashes
- Program produces wrong results
- Program runs very slow
- Program runs fast and correct
Five tricks to make it easier

- Use comments (it’s all logical now, but how about a week from now?)
- Use error handling (it will tell you what went wrong)
- Test for correctness
- Use external libraries
- Use building blocks where possible
Motivation

- GPU programming so far
  
  ```
  __kernel void foo () { ... }
  
  ... 
  
  cudaMalloc...
  cudaMemcpy...
  foo<<<?,?,?>>>();
  cudaMemcpy...
  ```

- Isn’t there a simpler way?
3 ways to accelerate applications on GPUs

- **Applications**
- **Libraries**
  - "Drop-in" Acceleration
- **Compiler Directives**
  - Easily Accelerate Applications
- **Programming Languages**
  - Maximum Flexibility

What are compiler directives?

When a compiler directive is encountered the compiler/runtime will...

1. Generate parallel code for GPU
2. Allocate GPU memory and copy input data
3. Execute parallel code on GPU
4. Copy output data to CPU and deallocate GPU memory
Why use compiler directives?

- **Single Source Code**
  - No need to maintain multiple code paths
- **High Level**
  - Abstract away device details, focus on expressing the parallelism and data locality
- **Low Learning Curve**
  - Programmer remains in same language and adds directives to existing code
- **Rapid Development**
  - Fewer code changes means faster development
History of OpenACC

• OpenACC is a specification for high-level, compiler directives for expressing parallelism for accelerators.
  – Aims to be performance portable to a wide range of accelerators.
  – Multiple Vendors, Multiple Devices, One Specification
• The OpenACC specification was first released in November 2011.
  – Original members: CAPS, Cray, NVIDIA, Portland Group
• OpenACC 2.0 was released in June 2013, expanding functionality and improving portability
• At the end of 2013, OpenACC had more than 10 member organizations
OpenACC directive syntax

- **C/C++**
  
  ```
  #pragma acc directive [clause [,] clause] ...
  ```

  ...often followed by a structured code block

- **Fortran**
  
  ```
  !$acc directive [clause [,] clause] ...
  ```

  ...often paired with a matching end directive surrounding a structured code block:

  ```
  !$acc end directive
  ```
OpenACC example: SAXPY

SAXPY in C

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc parallel loop
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

...// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...```

SAXPY in Fortran

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(n), y(n), a
    integer :: n, i

    !$acc parallel loop
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$acc end parallel loop
end subroutine saxpy

... // Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...```
OpenMP 4.0

- OpenMP has existed since 1997 as a specification for compiler directives for shared memory parallelism.
- In 2013, OpenMP 4.0 was released, expanding the focus beyond shared memory parallel computers, including accelerators.
- The OpenMP 4.0 `target` construct provides the means to offload data and computation to accelerators.
- Additional directives were added to support multiple thread teams and simd parallelism.
- OpenMP continues to improve upon its support for offloading to accelerators.
OpenMP directive syntax

- **C/C++**
  
  ```c
  #pragma omp target directive [clause [,] clause]…
  ```

  ...often followed by a structured code block

- **Fortran**
  
  ```fortran
  !$omp target directive [clause [,] clause] …
  ```

  ...often paired with a matching end directive surrounding a structured code block:

  ```fortran
  !$omp end target directive
  ```
OPENACC IS NOT GPU PROGRAMMING, IT IS EXPRESSING PARALLELISM IN YOUR CODE
Compiler directives by example

Identify Parallelism > Express Parallelism > Express Data Locality > Optimize
Example: Jacobi iteration (stencil code)

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
  - Common, useful algorithm
  - Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$

$$A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}$$
while ( err > tol && iter < iter_max ) {
    err=0.0;

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
Step 1: Identify parallelism
while ( err > tol && iter < iter_max ) {
    err=0.0;

    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                 A[j-1][i] + A[j+1][i]);
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
Step 2: Express parallelism

Identify Parallelism → Express Parallelism → Express Data Locality → Optimize

© Jeff Larkin, NVIDIA, S4167_JeffLarkin_Introduction_Complier_Directives.pdf
Parallel loop directive

Programmer identifies a block of code as having parallelism, compiler generates a parallel **kernel** for that loop.

```c
#pragma acc parallel loop
for(int i=0; i<N; i++)
{
    y[i] = a*x[i]+y[i];
}
```

**Kernel:** A function that runs in parallel on the GPU

© Jeff Larkin, NVIDIA, S4167_JeffLarkin_Introduction_Compiler_Directives.pdf
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    #pragma acc parallel loop
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
Kernels directive

The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```
#pragma acc kernels
{
  for(int i=0; i<N; i++)
  {
    a[i] = 0.0;
    b[i] = 1.0;
    c[i] = 2.0;
  }

  for(int i=0; i<N; i++)
  {
    a(i) = b(i) + c(i)
  }
}
```

The compiler identifies 2 parallel loops and generates 2 kernels.
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc kernels
{
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                               A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
Parallel loop vs. kernels

**Parallel loop**
- Requires analysis by programmer to ensure safe parallelism
- Straightforward path from OpenMP

**Kernels**
- Compiler performs parallel analysis and parallelizes what it believes safe
- Can cover larger area of code with single directive
- Gives compiler additional leeway.

Both approaches are equally valid and can perform equally well.
Building the code (using PGI compilers)

```
$ pgcc -acc -ta=nvidia:5.5,kepler -Minfo=accel -o laplace2d_acc laplace2d.c

main:
  56, Accelerator kernel generated
  57, #pragma acc loop gang /* blockIdx.x */
  59, #pragma acc loop vector(256) /* threadIdx.x */
  56, Generating present_or_copyout(Anew[1:4094][1:4094])
  Generating present_or_copyin(A[0:][0:])
  Generating NVIDIA code
  Generating compute capability 3.0 binary
  59, Loop is parallelizable
  63, Max reduction generated for error
  68, Accelerator kernel generated
  69, #pragma acc loop gang /* blockIdx.x */
  71, #pragma acc loop vector(256) /* threadIdx.x */
  68, Generating present_or_copyin(Anew[1:4094][1:4094])
  Generating present_or_copyout(A[1:4094][1:4094])
  Generating NVIDIA code
  Generating compute capability 3.0 binary
  71, Loop is parallelizable
```
Speed-up (higher is better)

Why is OpenACC so much slower?
Excessive data transfers

while ( err > tol && iter < iter_max ) {
  err=0.0;
  #pragma acc parallel loop reduction(max:err)
  for( int j = 1; j < n-1; j++ ) {
    for(int i = 1; i < m-1; i++) {
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  ...
}

And note that there are two #pragma acc parallel, so there are 4 copies per while loop iteration!

© Jeff Larkin, NVIDIA, S4167_JeffLarkin_Introduction_Compiler_Directives.pdf
Step 3: Express data locality
Defining data regions

- The `data` construct defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```c
#pragma acc data
{
#pragma acc parallel loop
...
#pragma acc parallel loop
...
}
```

Arrays used within the data region will remain on the GPU until the end of the data region.
Data clauses

- **copy (list)**: Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
- **copyin (list)**: Allocates memory on GPU and copies data from host to GPU when entering region.
- **copyout (list)**: Allocates memory on GPU and copies data to the host when exiting region.
- **create (list)**: Allocates memory on GPU but does not copy.
- **present (list)**: Data is already present on GPU from another containing data region.

And **present_or_copy[in|out]**, **present_or_create**, **deviceptr**.
Array shaping

• Compiler sometimes cannot determine size of arrays
  – Must specify explicitly using data clauses and array “shape”

C
  #pragma acc data copyin(a[0:size]),
  copyout(b[s/4:3*s/4])

Fortran
  !$acc data copyin(a(1:end)), copyout(b(s/4:3*s/4))

• Note: data clauses can also be used on data, parallel, or kernels
  e.g.: #pragma acc parallel loop copyin(a)
Jacobi iteration:
OpenACC C code (parallel loop + data)

```c
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc parallel loop
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Copy A to/from the accelerator only when needed. Create Anew as a device temporary.
A WORD OF CAUTION
General accelerator setup
Rationalize speedups

<table>
<thead>
<tr>
<th>Category</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single CPU core (reference)</td>
<td>1</td>
</tr>
<tr>
<td>First GPU port</td>
<td>84</td>
</tr>
<tr>
<td>Optimized GPU version</td>
<td>128</td>
</tr>
<tr>
<td>New GPU generation</td>
<td>233</td>
</tr>
<tr>
<td>Backporting to CPU</td>
<td>9.6</td>
</tr>
</tbody>
</table>
Reasonable speedups

• CPU in BigRed II has a peak performance of 160 GFLOP/s (DP) (320 GFLOP/s for CPU only nodes)
• GPU in BigRed II has a peak performance of 1,317 GFLOP/s (DP)
  \[ \rightarrow \text{factor 8.2 (or 4.1 vs. CPU nodes)} \]

• CPU in BigRed II has a peak memory bandwidth of 51.2 GB/s
• GPU in BigRed II has a peak memory bandwidth of 200 GB/s
  \[ \rightarrow \text{factor 3.9} \]

• Any speedup over 8 is unrealistic (or does not use multithreading on CPU, or ignores time for data transfers)
• Typical speedups are 2-4
QUESTIONS?