Programming Accelerators using Directives

Credits: Introduction to OpenACC and toolkit – Jeff Larkin, Nvidia
Basic Programming Models

- **Offload model**
  - idea: offload computational kernels
    - send data (or use unified memory)
    - call kernel(s)
    - retrieve data
  - accelerator-specific compiler support
    - Cuda compiler (nvcc)
  - accelerator-neutral OpenCL
    - Cuda-like notation
    - OpenCL compiler can target Nvidia or Intel Xeon Phi
      - no access to Nvidia or Intel Xeon Phi specifics
Emerging Programming Models

- **directive model**
  - idea: identify sections of code to be compiled for accelerator(s)
    - data transfer and kernel invocation generated by compiler

- **accelerator-neutral efforts**
  - **OpenACC**
    - `#pragma acc parallel loop`
      ```c
      for (...) {
          ...
      }
      ```
    - gang, worker, vector (threadblock, thread, warp in SIMT lockstep)
  - PGI, Cray, CAPS, Nvidia compilers
  - **OpenMP 4.0**
    - similar directives to (but more general than) OpenACC
    - implemented by gcc 4.9 and icc compiler
Introduction to OpenACC

Jeff Larkin, NVIDIA Developer Technologies
Why OpenACC?
OpenACC
Simple | Powerful | Portable

Fueling the Next Wave of Scientific Discoveries in HPC

main()
{
    <serial code>
    #pragma acc kernels
    // automatically runs on GPU
    {
        <parallel code>
    }
}

University of Illinois
PowerGrid- MRI Reconstruction
70x Speed-Up
2 Days of Effort

RIKEN Japan
NICAM- Climate Modeling
7-8x Speed-Up
5% of Code Modified

8000+ Developers
using OpenACC

http://www.cray.com/sites/default/files/resources/OpenACC_213462.12_OpenACC_Cosmo_CS_FNL.pdf
http://www.openacc.org/content/experiences-porting-molecular-dynamics-code-gpus-cray-xk7
OpenACC Directives

Manage Data Movement
- #pragma acc data copyin(a,b) copyout(c)
  
  { ... }  

Initiate Parallel Execution
- #pragma acc parallel
  
  { ...
     #pragma acc loop gang vector
     for (i = 0; i < n; ++i) {
       z[i] = x[i] + y[i];
     ... 
     } 
  
  } ...

Optimize Loop Mappings

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, MIC
Accelerated Computing Fundamentals
Accelerated Computing
10x Performance & 5x Energy Efficiency for HPC

CPU
Optimized for Serial Tasks

+

GPU Accelerator
Optimized for Parallel Tasks
What is Heterogeneous Programming?

- **GPU**: A few % of Code, A large % of Time
  - Compute-Intensive Functions
- **CPU**: Rest of Sequential CPU Code

Application Code + GPU

A few % of Code
A large % of Time

Rest of Sequential CPU Code
Portability & Performance

Accelerated Libraries
- High performance with little or no code change
- Limited by what libraries are available

Compiler Directives
- High Level: Based on existing languages; simple, familiar, portable
- High Level: Performance may not be optimal

Parallel Language Extensions
- Greater flexibility and control for maximum performance
- Often less portable and more time consuming to implement
Code for Portability & Performance

- **Libraries**
  - Implement as much as possible using portable libraries

- **Directives**
  - Use directives for rapid and portable development

- **Languages**
  - Use lower level languages for important kernels
OpenACC Programming Cycle
Identify Available Parallelism

Optimize Loop Performance

Express Data Movement

Express Parallelism
Example: Jacobi Iteration

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++) {
            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++;
}
Identify Parallelism

```c
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++;
}
```

Data dependency between iterations.

Independent loop iterations
OpenACC kernels Directive

The kernels directive identifies a region that may contain loops that the compiler can turn into parallel kernels.

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

    for(int i=0; i<N; i++) {
        y[i] = a*x[i] + y[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++) {


                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++;
}
Building the code

$ pgcc -fast -ta=tesla -Minfo=all laplace2d.c

main:

40, Loop not fused: function call before adjacent loop
   Generated vector sse code for the loop
51, Loop not vectorized/parallelized: potential early exits
55, Generating copyout(Anew[1:4094][1:4094])
   Generating copyin(A[:][:])
   Generating copyout(A[1:4094][1:4094])
   Generating Tesla code
57, Loop is parallelizable
59, Loop is parallelizable
   Accelerator kernel generated
57, #pragma acc loop gang /* blockIdx.y */
59, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
63, Max reduction generated for error
67, Loop is parallelizable
69, Loop is parallelizable
   Accelerator kernel generated
67, #pragma acc loop gang /* blockIdx.y */
69, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
Why did OpenACC slow down here?

Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40
Very low Compute/Memcpy ratio

- **Compute**: 5 seconds
- **Memory Copy**: 62 seconds

The amount of time performing compute is low relative to the amount of time required for memcpy.
PCIe Copies

104ms/iteration
while ( err > tol && iter < iter_max )
{
nerr=0.0;
}

#pragma acc kernels
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]));
    }
}

These copies happen every iteration of the outer while loop!
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++) {
                err = max(err, abs(Anew[j][i] - A[j][i]));
            }
        }
    }
    iter++;
}
Identify Available Parallelism

Express Parallelism

Express Data Movement

Optimize Loop Performance
Data regions

The `data` directive 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 kernels
...
#pragma acc kernels
...
}
```

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.

- **deviceptr** *(list)*: The variable is a device pointer (e.g. CUDA) and can be used directly on the device.
Array Shaping

Compiler sometimes cannot determine size of arrays

Must specify explicitly using data clauses and array “shape”

**C/C++**

```c
#pragma acc data copyin(a[0:nelem]) copyout(b[s/4:3*s/4])
```

**Fortran**

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

Note: data clauses can be used on *data, parallel, or kernels*
Express Data Locality

```c
#pragma acc data copy(A) create(Anew)
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++) {
                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++;
}
```

Copy A to/from the accelerator only when needed.
Create Anew as a device temporary.
Rebuilding the code

```bash
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c

main:
40, Loop not fused: function call before adjacent loop
   Generated vector sse code for the loop
51, Generating copy(A[:][:])
      Generating create(Anew[:][:])
   Loop not vectorized/parallelized: potential early exits
56, Accelerator kernel generated
   56, Max reduction generated for error
   57, #pragma acc loop gang /* blockIdx.x */
      59, #pragma acc loop vector(256) /* threadIdx.x */
56, Generating Tesla code
59, Loop is parallelizable
67, Accelerator kernel generated
   68, #pragma acc loop gang /* blockIdx.x */
       70, #pragma acc loop vector(256) /* threadIdx.x */
67, Generating Tesla code
70, Loop is parallelizable
```
Visual Profiler: Data Region

Iteration 1

Iteration 2

Was 104 ms
Speed-Up (Higher is Better)

Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40

- Single Thread: 0.00X
- 2 Threads: 1.90X
- 4 Threads: 3.20X
- 6 Threads: 3.74X
- 8 Threads: 3.83X
- OpenACC: 19.89X

Socket/Socket: 5.2X
The loop Directive

The `loop` directive gives the compiler additional information about the next loop in the source code through several clauses.

- **independent** - all iterations of the loop are independent
- **collapse(N)** - turn the next N loops into one, flattened loop
- **tile(N[,M,...])** - break the next 1 or more loops into tiles based on the provided dimensions.

These clauses and more will be discussed in greater detail in a later class.
Optimize Loop Performance

```c
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;
    #pragma acc kernels
    {
        #pragma acc loop device_type(nvidia) tile(32,4)
        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]));
            }
        }
        #pragma acc loop device_type(nvidia) tile(32,4)
        for( int j = 1; j < n-1; j++ ) {
            for( int i = 1; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
        }
    }
    iter++;
}
```

“Tile” the next two loops into 32x4 blocks, but only on NVIDIA GPUs.
Speed-Up (Higher is Better)

Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40

- Single Thread: 1.90X
- 2 Threads: 3.20X
- 4 Threads: 3.74X
- 6 Threads: 3.83X
- 8 Threads: 19.89X
- OpenACC: 21.22X
- OpenACC Tuned: 21.22X
Portability of OpenMP Offload Directives
Jeff Larkin, OpenMP Booth Talk SC17
Background

Many developers choose OpenMP in hopes of having a single source code that runs effectively anywhere (performance portable).

As of November 2017, OpenMP compilers deliver on performance, portability, and performance portability?

- Will OpenMP Target code be portable between compilers?
- Will OpenMP Target code be portable with the host?

I will compare results using 6 compilers: CLANG, Cray, GCC, Intel, PGI, and XL
Goal of this study

1. For the GPU-enabled compilers, compare performance on a simple benchmark code.
   Metric = Execution time on GPU

2. Quantify each compiler’s ability to performantly fallback to the host CPU
   In other words, if I write offloading code, will it still perform well on the host?
   Metric = Time Native OpenMP / Time Host Fallback
Compiler Versions & Flags

CLANG (IBM Power8 + NVIDIA Tesla P100)
- clang/20170629
- -O2 -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda --cuda-path=$CUDA_HOME

Cray (Cray XC50)
- 8.5.5

GCC (IBM Power8 + NVIDIA Tesla P100)
- 7.1.1 20170718 (experimental)
- -O3 -fopenmp -foffload="-lm"
Compiler Versions & Flags

Intel (Intel “Haswell”)
- 17.0
- -Ofast -qopenmp -std=c99 -qopenmp-offload=host

XL (IBM Power8 + NVIDIA Tesla P100)
- xl/20170727-beta
- -O3 -qsmp -qoffload

PGI (Intel CPU)
- 17.10 (llvm)
- -fast -mp -Minfo -Mllvm
Case Study: Jacobi Iteration
Example: Jacobi Iteration

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: $\nabla^2 f(x, y) = 0$

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$
Teams & Distribute
OpenMP Teams

- **TEAMS Directive**
  
  - To better utilize the GPU resources, use many thread teams via the TEAMS directive.
  
  - Spawns 1 or more thread teams with the same number of threads
  
  - Execution continues on the master threads of each team (redundantly)
  
  - No synchronization between teams
OpenMP Teams

- **DISTRIBUTE Directive**

  - Distributes the iterations of the next loop to the master threads of the teams.
  
    - Iterations are distributed statically.
    - There’s no guarantees about the order teams will execute.
    - No guarantee that all teams will execute simultaneously
    - Does not generate parallelism/worksharing within the thread teams.
Teaming Up

```c
#pragma omp target data map(to:Anew) map(A)
while ( error > tol && iter < iter_max )
{
    error = 0.0;

#pragma omp target teams distribute parallel for reduction(max:error) map(error)
    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] );
            error = fmax( error, fabs( Anew[j][i] - A[j][i] ) );
        }
    }

#pragma omp target teams distribute parallel for
    for( int j = 1; j < n-1; j++ )
    {
        for( int i = 1; i < m-1; i++ )
        {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
    iter++;
}
```

Explicitly maps arrays for the entire while loop.

- Spawns thread teams
- Distributes iterations to those teams
- Workshares within those teams.
Execution Time (Smaller is Better)

CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100
Execution Time (Smaller is Better)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Data</th>
<th>Kernels</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLANG</td>
<td>7.50616</td>
<td></td>
<td></td>
<td>7.50616</td>
</tr>
<tr>
<td>Cray</td>
<td>1.47895</td>
<td></td>
<td></td>
<td>1.47895</td>
</tr>
<tr>
<td>GCC simd</td>
<td>4.109866</td>
<td></td>
<td></td>
<td>4.109866</td>
</tr>
<tr>
<td>XL</td>
<td>17.402229</td>
<td></td>
<td></td>
<td>17.402229</td>
</tr>
<tr>
<td>XL simd</td>
<td>11.046584</td>
<td></td>
<td></td>
<td>11.046584</td>
</tr>
</tbody>
</table>

CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100
Increasing Parallelism
Increasing Parallelism

Currently both our distributed and workshared parallelism comes from the same loop.

- We could collapse them together
- We could move the PARALLEL FOR to the inner loop

The COLLAPSE(N) clause

- Turns the next N loops into one, linearized loop.
- This will give us more parallelism to distribute, if we so choose.
Collapse the two loops into one and then parallelize this new loop across both teams and threads.
Execution Time (Smaller is Better)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Kernel Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLANG</td>
<td>1.490654</td>
</tr>
<tr>
<td>Cray</td>
<td>1.820148</td>
</tr>
<tr>
<td>XL</td>
<td>3.706288</td>
</tr>
<tr>
<td>GCC</td>
<td>41.812337</td>
</tr>
</tbody>
</table>

CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100
Execution Time (Smaller is Better)

CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100
Splitting Teams & Parallel

```c
#pragma omp target teams distribute map(error)
    for( int j = 1; j < n-1; j++)
    {
        #pragma omp parallel for reduction(max:error)
            for( int i = 1; i < m-1; i++ )
            {
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
    }

#pragma omp target teams distribute
    for( int j = 1; j < n-1; j++)
    {
        #pragma omp parallel for
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
    }
```

Distribute the “j” loop over teams.

Workshare the “i” loop over threads.
Execution Time (Smaller is Better)

CLANG, GCC, XL: IBM “Minsky”, NVIDIA Tesla P100, Cray: Cray XC-40, NVIDIA Tesla P100
Execution Time (Smaller is Better)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLANG</td>
<td>2.033891</td>
</tr>
<tr>
<td>Cray</td>
<td>1.47519</td>
</tr>
<tr>
<td>GCC simd</td>
<td>7.375151</td>
</tr>
<tr>
<td>XL</td>
<td>14.10688</td>
</tr>
</tbody>
</table>

CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100
Host Fallback
Fallback to the Host Processor

Most OpenMP users would like to write 1 set of directives for host and device, but is this really possible?

Using the “if” clause, offloading can be enabled/disabled at runtime.

```c
#pragma omp target teams distribute parallel for reduction(max:error) map(error) \
collapse(2) if(target:use_gpu)
  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] );
      error = fmax( error, fabs(Anew[j][i] - A[j][i]) );
    }
  }
```

Compiler must build CPU & GPU codes and select at runtime.
Host Fallback Comparison

“Native” OpenMP = Standard OMP PARALLEL FOR (SIMD)

“Host Fallback” = Device OpenMP, forced to run on host

Metric: Native Time / Host Fallback Time

In other words…

100% means the perform equally well

50% means the host fallback takes 2X longer
Host Fallback vs. Host Native OpenMP

CLANG, GCC, XL: IBM "Minsky", NVIDIA Tesla P100, Cray: Cray XC-50, NVIDIA Tesla P100, Intel, PGI: Intel "Haswell"
Conclusions
Conclusions

• Will OpenMP Target code be portable between compilers?

Maybe. Compilers are of various levels of maturity. SIMD support/requirement inconsistent.

• Will OpenMP Target code be portable with the host?

Highly compiler-dependent. Intel, PGI, and XL do this very well, CLANG somewhat well, and GCC and Cray did poorly.

Future Work: Revisit these experiments as compilers are updated.