Introduction to Directive Based Programming on GPU

Helen He
Feb 28, 2020
Acknowledgement

• **Used materials and examples from:**
  - Jeff Larkin, Eric Wright: Slides and code examples from NVidia OpenACC training materials.
  - Tim Mattson, Simon McIntosh-Smith: SC19 Programming your GPUs with OpenMP tutorial
  - Michael Klemm and Bronis de Supinski: What’s new in OpenMP 5.0
  - Oscar Hernandez, et al.: ECP 2020 OpenMP 5.0/5.1 tutorial
  - Chris Daley: slides contents, OpenMP and OpenACC performance results
  - Nvidia OpenACC Boot Camp slides.
  - Many others: for OpenMP updates and performance results presented at the DOE ECP 2020 OpenMP BoF, etc.

• **Thank you all!**
CPU vs. GPUs

- CPUs generally have a small number of very fast physical cores.
- GPUs have thousands of simple cores able to achieve high performance in aggregate.
- Mostly CPU and GPU do not share memory. Data move between CPU and GPU is expensive. Keep data on GPU as long as possible.
- We need to keep GPU busy, and only offload computational intensive kernels to GPU.

Image from NVidia
Sample OpenMP and OpenACC Codes

```c
#define N 128
double x[N*N];
int i, j, k;
for (k=0; k<N*N; ++k) x[k] = k;

#pragma omp target
#pragma omp teams distribute
for (i=0; i<N; ++i) {
#pragma omp parallel for simd
    for (j=0; j<N; ++j) {
        x[j+N*i] *= 2.0;
    }
}
```

```c
#define N 128
double x[N*N];
int i, j, k;
for (k=0; k<N*N; ++k) x[k] = k;

#pragma acc parallel
#pragma acc gang worker
for (i=0; i<N; ++i) {
#pragma acc vector
    for (j=0; j<N; ++j) {
        x[j+N*i] *= 2.0;
    }
}
```

directives
Advantages of Directive Based Parallelism

- **Incremental parallel programming**
  - Find hotspot, parallelize, check correctness, repeat

- **Single source code for sequential and parallel programs**
  - Use compiler flag to enable or disable
  - No major overwrite of the serial code

- **Works for both CPU and GPU**

- **Low learning curve, familiar C/C++/Fortran program environment**
  - Do not need to worry about lower level hardware details

- **Simple programming model than lower level programming models**

- **Portable implementation:**
  - different architectures, different compilers handle the hardware differences
Device Execution Model

- Device: An implementation-defined logical execution unit.
- Can have a single host and one or more target devices (accelerators).
- Host and Device have separate data environment (except with managed memory or unified shared memory).
- The execution model is host-centric
  - Host creates/destroys data environment on the device(s)
  - Host maps data to the device data environment.
  - Host then offloads accelerator regions to the device for execution
  - Host updates the data between the host and the device.
  - Host destroys data environment on device.
Host/Device Platform Model and OpenMP

Parallel for `simd` to run each block of loop iterations on the processing elements.

Teams construct to create a league of teams with one team of threads on each compute unit.

Distribute construct to assign blocks of loop iterations to teams.

Target construct to get onto a device.

No synchronization among teams.

Courtesy of Tim Mattson and Simon McIntosh-Smith.
Host/Device Platform Model and OpenMP

Typical usage ... let the compiler do what's best for the device:

```
#pragma omp target
to get on the device
```

```
#pragma omp teams distribute parallel for simd
to assign work to the device processing elements
```

```
#pragma omp teams
to get onto a device
```

```
#pragma omp target
to get on the device
```

Courtesy of Tim Mattson and Simon McIntosh-Smith
Multi-level Device Parallelism

```c
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

    #pragma omp target data map(to:x[0:n])
    {
        #pragma omp target map(tofrom:y)
        #pragma omp teams num_teams(num_blocks) num_threads(bsize)
        all do the same

        #pragma omp distribute
        for (int i = 0; i < n; i += num_blocks) {
            workshare (w/o barrier)
        }

        #pragma omp parallel for simd
        for (int j = i; j < i + num_blocks; j++) {
            y[j] = a*x[j] + y[j];
            workshare (w/ barrier)
        }
    }
}
```

All do the same

workshare (w/o barrier)

workshare (w/ barrier)
Host/Device Platform Model and OpenACC

Worker Vector to run each block of loop iterations on the processing elements

Gang directive to create a league of gangs with one gang of threads on each compute unit, and assign blocks of loop iterations to gangs.

Parallel construct to get onto a device

No synchronization among gangs

Courtesy of Tim Mattson and Simon McIntosh-Smith
Host/Device Platform Model and OpenACC

Typical usage ... let the compiler do what’s best for the device: use “acc loop”

#pragma acc parallel to get on the device

Worker Vector to run each block of loop iterations on the processing elements

Gang directive to create a league of teams with one team of threads

#pragma acc loop to assign work to the device processing elements

Courtesy of Tim Mattson and Simon McIntosh-Smith
Sample OpenACC Codes

#pragma acc parallel
{
  for (i =0; i<n; i++) {
    c[i] = a[i] + b[i];
    ...
  }
}

Without #pragma acc loop, the loop is not distributed, and all threads will execute the entire loop redundantly.

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

These two are equivalent. With #pragma acc loop, it will choose the best gang/worker/vector values and parallelize the loop.

#pragma acc parallel
{
  #pragma acc loop gang worker vector
  for (i =0; i<n; i++) {
    c[i] = a[i] + b[i];
    ...
  }
}
The loop Directive/Construct

- OpenACC: “#pragma acc loop” lets the compiler to decide what’s best gang, worker, vector values to use to parallelize.
- OpenMP 5.0: the loop construct asserts to the compiler that the iterations of a loop are free of dependencies and may be run concurrently in any order.
- It lets the OpenMP implementation to choose the right parallelization scheme.
- Each iteration execute exactly once.

```c
#pragma omp target map(to:x[0:n]) map(tofrom:y)
{
    #pragma omp loop
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}
```
OpenACC and OpenMP Syntax

C/C++:

OpenACC:  
  `#pragma acc` directive clauses
  `<code>`

OpenMP:  
  `#pragma omp` directive clauses
  `<code>`

Fortran:

OpenACC:  
  `!$acc` directive clauses
  `<code>`
  `!$acc end directive`

OpenMP:  
  `!$omp` directive clauses
  `<code>`
  `!$omp end directive`

- A `pragma` in C/C++ or `$` in Fortran gives instructions to the compiler on how to compile the code.  `$acc end` or `$omp end` sometimes is optional (depends on what the directive is).

- “`acc`” or “`omp`” informs the compiler that this is an OpenACC or OpenMP directive.

- Directives are ignored by a compiler that does not understand a particular pragma (such as when the compiler flag is not turned on to enable OpenACC or OpenMP support).

- **Clauses** are specifiers or additions to directives.
OPENACC parallel Directive

Expressing parallelism

#pragma acc parallel
{
    When encountering the \textit{parallel} directive, the compiler will generate \textit{1 or more parallel gangs}, which execute redundantly.
}

Slide courtesy of Jeff Larkin, Eric Wright, NVidia
OPENACC parallel Directive

Expressing parallelism

```c
#pragma acc parallel
{
  for(int i = 0; i < N; i++)
  {
    // Do Something
  }
}
```

This loop will be executed redundantly on each gang

Slide courtesy of Jeff Larkin, Eric Wright, NVidia
**OPENACC PARALLEL DIRECTIVE**

*Expressing parallelism*

```c
#pragma acc parallel
{
    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }
    The *loop* directive informs the compiler which loops to parallelize.
}
```

*OpenACC*

---

Slide courtesy of Jeff Larkin, Eric Wright, NVidia
GANG WORKER VECTOR

- Gang / Worker / Vector defines the various levels of parallelism we can achieve with OpenACC
- This parallelism is most useful when parallelizing multi-dimensional loop nests
- OpenACC allows us to define a generic Gang / Worker / Vector model that will be applicable to a variety of hardware, but we will focus a little bit on a GPU specific implementation

```
#pragma acc parallel loop gang worker
for ( i = 0; i < N; i++ )
#pragma acc loop vector
for ( j = 0; j < M; j++ )
<loop code>
```

Slide courtesy of Jeff Larkin, Eric Wright, NVidia
Use combined Directives

• Each compiler supports different levels of parallelism
  – LLVM/clang 10, AMD, CCE9, IBM, PGI: teams, parallel
  – (Planned) LLVM/Clang 11, Intel: teams, parallel, simd
  – CCE8: teams, parallel or teams, simd

• Caveats:
  – Real applications will have algorithms that are structured such that they can’t immediately use the combined construct.
  – It may also make collapse hard to do
  – Performance can be achieved without combined directives, but likely won’t be portable

```c
#pragma omp target teams distribute parallel for simd
for (int i = 0; i < n; i++) {
    F(i) = G(i);
}
```
### Hardware and Software Mapping

<table>
<thead>
<tr>
<th><strong>GPU Hardware</strong></th>
<th><strong>OpenACC</strong></th>
<th><strong>OpenMP</strong></th>
<th><strong>CUDA</strong></th>
<th><strong>OpenCL</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Group</td>
<td>Gang</td>
<td>Team</td>
<td>Thread Block</td>
<td>Work Group</td>
</tr>
<tr>
<td>EU Thread</td>
<td>Worker</td>
<td>Thread</td>
<td>Worker Thread</td>
<td>Wave Front</td>
</tr>
<tr>
<td>SIMD Lane</td>
<td>Vector</td>
<td>SIMD</td>
<td>Warp</td>
<td>Thread</td>
</tr>
</tbody>
</table>

- For OpenACC, recommend to use “acc loop” to let the compiler choose number of gangs, workers, and vectors.
- If you choose manually, recommend to use gang for most outer loop, and vector for most inner loop. Also to use the vector length a multiple of 32, which is the warp size.
- For OpenMP, recommend to use the combined contracts syntax and let the compiler choose number of teams, threads, etc.
Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$

Use Jacobi solver to iteratively update the value (e.g. Temperature) at each point from the average of neighboring points, until it converges

Common, useful algorithm

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$
Laplace Equation: C Code

```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++;
}
```

Iterate until converged
Iterate across matrix elements
Calculate new value from neighbors
Compute max error for convergence
Swap input/output arrays

Slide and example code courtesy of Jeff Larkin, Eric Wright, NVidia
reduction Clause

- Common situation: combine values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can’t be trivially removed
- We can use “reduction” clause in both OpenMP and OpenACC

```c
double ave=0.0, A[MAX];  int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```

- Syntax: Reduction (operator : list).
- Reduces list of variables into one, using operator.
- Reduced variables must be shared variables.
- Allowed Operators:
  - Arithmic: + - * / # add, substract, multiply, divide
  - Fortran intrinsic: max min
  - Bitwise: & | ^ # and, or, xor
  - Logical: && || # and, or
collapse clause

**FORTRAN example:**

```fortran
!$acc loop collapse (2)
do i = 1, 1000
do j = 1, 100
   a(i,j) = b(i,j) + c(i,j)
enddo
dddo
!$acc end loop
```

**FORTRAN example:**

```fortran
!$omp do collapse(2)
do i = 1, 1000
   do j = 1, 100
      a(i,j) = b(i,j) + c(i,j)
   enddo
enddo
!$omp end do
```

- **collapse** \((n)\) collapses the \(n\) nested loops into 1 large loop, then schedule work for each thread accordingly.
OpenACC Parallel

while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop reduction(max:err) collapse(2)
    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 parallel loop collapse(2)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}

Example code courtesy of Jeff Larkin, Eric Wright, NVidia
CUDA Managed Memory (Unified Shared Memory)

- Single address space over CPU and GPU memories
- Compiler manages data migration between CPU and GPU memories - no need to explicitly copy data
- Usually slower than explicitly memory management
- OpenACC: Currently only available from PGI on NVIDIA GPUs. Enabled via a compiler flag
- OpenMP 4.5: allocate use `cudaMallocManaged`
- OpenMP 5.0: language feature

```c
#pragma omp requires unified_shared_memory
for (k=0; k < NTIMES; k++) {
    #pragma omp target teams distribute parallel for simd
    for (j=0; j < N; j++) {
        a[j] = b[j] + scalar * c[j];
    }
}
```
% pgcc -fast -ta=tesla:cc70 -Minfo=accel laplace2d.c jacobi.c
PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol - A (laplace2d.c: 37)
calcNext:
  37, Generating Tesla code
  38, #pragma acc loop gang /* blockIdx.x */
      Generating reduction(max:error)
  41, #pragma acc loop vector(128) /* threadIdx.x */
  38, Accelerator restriction: size of the GPU copy of Anew,A is unknown
  41, Loop is parallelizable
PGC-F-0704-Compilation aborted due to previous errors. (laplace2d.c)
PGC/x86-64 Linux 19.10-0: compilation aborted

% pgcc -fast -ta=tesla:cc70,managed -Minfo=accel laplace2d.c jacobi.c
Almost no memory copy cost, with managed memory
### Data Clauses

#### Multi-dimensional Array shaping

<table>
<thead>
<tr>
<th>OpenACC</th>
<th>C/C++</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>copy(array[0:N][0:M])</code></td>
<td><code>map(tofrom:array[0:N][0:M])</code></td>
<td><code>map(tofrom:copy(array(1:N,1:M)))</code></td>
</tr>
<tr>
<td><code>copy(array(1:N,1:M))</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C/C++: `starting_index : length`

Fortran: `starting_index : ending_index`

- Data clauses allow the programmer to tell the compiler which data to move and when between the host and device.
- Data clauses may be added to kernels or parallel regions, but also data, enter data, and exit data.
### OpenACC and OpenMP data Clauses

<table>
<thead>
<tr>
<th>OpenACC</th>
<th>OpenMP</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy (var)</td>
<td>map (tofrom: var)</td>
<td>Allocate on GPU, copy from host to GPU when enter and copy from GPU to host when exit</td>
</tr>
<tr>
<td>copyin (var)</td>
<td>map (to: var)</td>
<td>Allocate on GPU, copy from host to GPU when enter</td>
</tr>
<tr>
<td>copyout (var)</td>
<td>map (from: var)</td>
<td>Allocate on GPU, copy from GPU to host when exit</td>
</tr>
<tr>
<td>create (var)</td>
<td>map (alloc: var)</td>
<td>Allocate on GPU but does not copy</td>
</tr>
<tr>
<td>delete (var)</td>
<td>map (delete: var)</td>
<td>Delete from GPU</td>
</tr>
<tr>
<td>present (var)</td>
<td>present (var)</td>
<td>Data is present on GPU</td>
</tr>
</tbody>
</table>
#pragma acc parallel loop copy(A[:m*n],Anew[:m*n]) reduction(max:err) collapse(2)
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 collapse(2)
for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ )
    {
        A[j][i] = Anew[j][i];
    }
}

Example code courtesy of Jeff Larkin, Eric Wright, NVidia
OpenACC Parallel, Basic Data

% pgcc -acc -fast -ta=tesla:cc70 -Minfo=accel,opt laplace2d.c jacobi.c
calcNext:
   37, Generating copy(A[:n*m]) [if not already present]
       Generating Tesla code
   38, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
   40, /* blockIdx.x threadIdx.x collapsed */
       Generating reduction(max:error)
   37, Generating implicit copy(error) [if not already present]
       Generating copy(Anew[:n*m]) [if not already present]
swap:
  52, Generating copy(Anew[:n*m],A[:n*m]) [if not already present]
     Generating Tesla code
  53, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
  55, /* blockIdx.x threadIdx.x collapsed */
jacobi.c:

Data copy at each loop. 98% time spent in data move

<table>
<thead>
<tr>
<th>Name</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min.</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>[CUDA memcpy HtoD]</td>
<td>46.1974s</td>
<td></td>
<td>1.3999ms</td>
<td>1.2480us</td>
<td>5.9465ms</td>
</tr>
<tr>
<td>[CUDA memcpyDtoH]</td>
<td>41.2126s</td>
<td></td>
<td>1.1139ms</td>
<td>1.8240us</td>
<td>1.5382ms</td>
</tr>
<tr>
<td>calcNext_36_gpu</td>
<td>572.06ms</td>
<td></td>
<td>572.06us</td>
<td>556.35us</td>
<td>590.68us</td>
</tr>
<tr>
<td>swap_52_gpu</td>
<td>452.57ms</td>
<td></td>
<td>452.57us</td>
<td>444.99us</td>
<td>464.09us</td>
</tr>
<tr>
<td>calcNext_44_gpu__red</td>
<td>7.8517ms</td>
<td></td>
<td>7.8510us</td>
<td>7.4550us</td>
<td>9.2160us</td>
</tr>
</tbody>
</table>
Structured Data Directive

**OpenACC**

```c
#pragma acc data map(to:A, B) map(from: C)
{
  #pragma acc parallel
  {do lots of stuff with A, B and C}

  {do something on the host}

  #pragma acc parallel
  {do lots of stuff with A, B, and C}
}
```

**OpenMP**

```c
#pragma omp target data map(to:A, B) map(from: C)
{
  #pragma omp target
  {do lots of stuff with A, B and C}

  {do something on the host}

  #pragma omp target
  {do lots of stuff with A, B, and C}
}
```
#pragma acc data copyin(A[:n*m]) create(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop copy(A[:n*m],Anew[:n*m]) reduction(max:err) collapse(2)
    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 collapse(2)
    for( int j = 1; j < n-1; j++ ) {

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

    iter++;
}

Copy A to/from the accelerator only when needed. Copy initial condition of Anew, but not final value.

Example code courtesy of Jeff Larkin, Eric Wright, NVidia
% pgcc -acc -fast -ta=tesla:cc70 -Minfo=accel,opt laplace2d.c jacobi.c

laplace2d.c:
calcNext:
  37, Generating copy(A[:n*m]) [if not already present]
  Generating Tesla code
  38, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
  40, /* blockIdx.x threadIdx.x collapsed */
  Generating reduction(max:error)
  37, Generating implicit copy(error) [if not already present]
  Generating copy(Anew[:n*m]) [if not already present]

swap:
  52, Generating copy(Anew[:n*m],A[:n*m]) [if not already present]
  Generating Tesla code
  53, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
  55, /* blockIdx.x threadIdx.x collapsed */

jacobi.c:
main:
  59, Generating create(Anew[:m*n]) [if not already present]
  Generating copyin(A[:m*n]) [if not already present]

Now < 2% time spent in data move

Total 1.071 sec

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>44.06%</td>
<td>369.08ms</td>
<td>1000</td>
<td>369.08us</td>
<td>367.32us</td>
<td>372.12us</td>
<td>calcNext_37_gpu</td>
</tr>
<tr>
<td>43.01%</td>
<td>360.23ms</td>
<td>1000</td>
<td>360.23us</td>
<td>358.40us</td>
<td>362.04us</td>
<td>swap_52_gpu</td>
</tr>
<tr>
<td>11.29%</td>
<td>94.543ms</td>
<td>1000</td>
<td>94.542us</td>
<td>89.503us</td>
<td>102.72us</td>
<td>calcNext_37_gpu__red</td>
</tr>
<tr>
<td>1.30%</td>
<td>10.877ms</td>
<td>8</td>
<td>1.3596ms</td>
<td>1.3534ms</td>
<td>1.3708ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>0.18%</td>
<td>1.5005ms</td>
<td>1000</td>
<td>1.5000us</td>
<td>1.4080us</td>
<td>2.8160us</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>0.16%</td>
<td>1.3699ms</td>
<td>1000</td>
<td>1.3690us</td>
<td>1.3110us</td>
<td>1.9840us</td>
<td>[CUDA memset]</td>
</tr>
</tbody>
</table>
OpenMP Target, Fortran, Structured Data

```fortran
!$omp target data map(to:A) map(alloc:Anew)
do while ( error .gt. tol .and. iter .lt. iter_max )
  !$omp target teams distribute parallel do simd collapse(2) map(to:A) map(from:Anew)
  map(tofrom:error) reduction(max:error)
  do j=1,m-2
    do i=1,n-2
      Anew(i,j) = 0.25_fp_kind * ( A(i+1,j ) + A(i-1,j ) + &
                                      A(i ,j-1) + A(i ,j+1) )
      error = max( error, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
  !$omp end target teams distribute parallel do simd
  !$omp target teams distribute parallel do simd collapse(2) map(to:Anew) map(from:A)
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
  !$omp end target teams distribute parallel do
  iter = iter + 1
end do
!$omp end target data
```

Total 0.782 sec. Use CCE9

<table>
<thead>
<tr>
<th>Name</th>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>jacobi_$ck_L47_1</td>
<td>60.41%</td>
<td>595.91ms</td>
<td>1000</td>
<td>595.91us</td>
<td>586.91us</td>
<td>607.67us</td>
</tr>
<tr>
<td>jacobi_$ck_L48_4</td>
<td>36.49%</td>
<td>359.97ms</td>
<td>1000</td>
<td>359.97us</td>
<td>359.07us</td>
<td>361.24us</td>
</tr>
<tr>
<td>[CUDA memcpy HtoD]</td>
<td>2.94%</td>
<td>29.016ms</td>
<td>1002</td>
<td>28.958us</td>
<td>1.3120us</td>
<td>27.652ms</td>
</tr>
<tr>
<td>[CUDA memcpyDtoH]</td>
<td>0.15%</td>
<td>1.4800ms</td>
<td>1000</td>
<td>1.4800us</td>
<td>1.4390us</td>
<td>1.8880us</td>
</tr>
</tbody>
</table>
void init_array(int *A, int N) {
  for (int i = 0; i < N; ++i)
    A[i] = i;
  #pragma acc enter data copyin (A[0:N])
}

int main(void) {
  int N = 1024;
  int *A = malloc(sizeof(int) * N);
  init_array(A, N);

  #pragma acc loop
  for (int i = 0; i < N; ++i)

  #pragma acc exit data copyout (A[0:N])
}
Structured vs. Unstructured Data Directive

**Structured**
- Has explicit start and end points
- Within a single function
- Memory exist within the data region

**Unstructured**
- Can have multiple start and end points
- Can branch across multiple functions
- Memory exists until explicitly deallocated
void initialize( ...) {
    
    #pragma acc enter data copyin(A[:m*n],Anew[:m*n])
}

while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc parallel loop copy(A[:n*m],Anew[:n*m]) reduction(max:err) collapse(2)
    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]));
        }
    }
    ..
    iter++;
}

void deallocate(..) {
    #pragma acc exit data delete(A,Anew)
    ..
}

Example code courtesy of Jeff Larkin, Eric Wright, NVidia
% pgcc -acc -fast -ta=tesla:cc70 -Minfo=accel,opt laplace2d.c jacobi.c

laplace2d.c:
initialize:
  33, Generating enter data copyin(Anew[:m*n],A[:m*n])
calcNext:
  39, Generating copy(A[:n*m]) [if not already present]
       Generating Tesla code
  40, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
       /* blockIdx.x threadIdx.x collapsed */
       Generating reduction(max:error)
  39, Generating implicit copy(error) [if not already present]
       Generating copy(Anew[:n*m]) [if not already present]
swap:
  54, Generating copy(Anew[:n*m],A[:n*m]) [if not already present]
       Generating Tesla code
  55, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
       /* blockIdx.x threadIdx.x collapsed */
deallocate:
  66, Generating exit data delete(Anew[:1],A[:1])
jacobi.c:

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>43.52%</td>
<td>369.11ms</td>
<td>1000</td>
<td>369.11us</td>
<td>367.64us</td>
<td>372.70us</td>
<td>calcNext_39_gpu</td>
</tr>
<tr>
<td>42.47%</td>
<td>360.21ms</td>
<td>1000</td>
<td>360.21us</td>
<td>358.78us</td>
<td>362.14us</td>
<td>swap_54_gpu</td>
</tr>
<tr>
<td>11.12%</td>
<td>94.307ms</td>
<td>1000</td>
<td>94.307us</td>
<td>89.791us</td>
<td>103.17us</td>
<td>calcNext_39_gpu__red</td>
</tr>
<tr>
<td>2.56%</td>
<td>21.678ms</td>
<td>16</td>
<td>1.3549ms</td>
<td>1.3535ms</td>
<td>1.3627ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>0.18%</td>
<td>1.4986ms</td>
<td>1000</td>
<td>1.4980us</td>
<td>1.4080us</td>
<td>2.8480us</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>0.16%</td>
<td>1.3682ms</td>
<td>1000</td>
<td>1.3680us</td>
<td>1.3110us</td>
<td>1.9840us</td>
<td>[CUDA memset]</td>
</tr>
</tbody>
</table>
OpenMP Target, Fortran, Unstructured Data

!$omp enter data map(to:A) map(alloc:Anew)
  do while ( error .gt. tol .and. iter .lt. iter_max )

  !$omp target teams distribute parallel do simd collapse(2) map(to:A) map(from:Anew)
    map(tofrom:error) reduction(max:error)
      do j=1,m-2
        do i=1,n-2
          Anew(i,j) = 0.25_fp_kind * ( A(i+1,j ) + A(i-1,j ) + &
                                      A(i  ,j-1) + A(i  ,j+1) )
          error = max( error, abs(Anew(i,j)-A(i,j)) )
        end do
      end do

  !$omp end target teams distribute parallel do simd

  !$omp target teams distribute parallel do simd collapse(2) map(to:Anew) map(from:A)
    do j=1,m-2
      do i=1,n-2
        A(i,j) = Anew(i,j)
      end do
    end do

  !$omp end target teams distribute parallel do simd

  !$omp exit data map(from:A) map(delete:Anew)

Total 0.777 sec. Use CCE9

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>58.85%</td>
<td>595.69ms</td>
<td>1000</td>
<td>595.69us</td>
<td>586.23us</td>
<td>607.67us</td>
<td>jacobi_$ck_L46_1</td>
</tr>
<tr>
<td>35.53%</td>
<td>359.68ms</td>
<td>1000</td>
<td>359.68us</td>
<td>357.88us</td>
<td>361.08us</td>
<td>jacobi_$ck_L47_4</td>
</tr>
<tr>
<td>2.86%</td>
<td>28.984ms</td>
<td>1002</td>
<td>28.925us</td>
<td>1.3120us</td>
<td>27.601ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>2.75%</td>
<td>27.839ms</td>
<td>1001</td>
<td>27.811us</td>
<td>1.4400us</td>
<td>26.336ms</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
</tbody>
</table>
Laplace Performance on Cori GPU (and Summit)

- Code optimized with explicit data management, but no further tuning.
- OpenMP XL is from Summit
- OpenMP CCE 9 Fortran, CCE 8 C/Fortran, and OpenACC PGI C are among the best.
- OpenACC GCC8 Fortran and OpenMP XL C are closer up.
OpenACC kernels vs. parallel

• The kernels directive instructs the compiler to search for parallel loops in the code
• The compiler will analyze the loops and parallelize those it finds safe and profitable to do so. Correctness is guaranteed.
• The kernels directive can be applied to regions containing multiple loop nests. The compiler will attempt to parallelize all loops within the kernels region. Each loop can be parallelized/optimized in a different way

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

#pragma acc parallel
#pragma acc loop
for( int j=1; j<n-1; j++ ) {
    for(int i=1; i< m-1; i++){
        #pragma acc loop
        for(int i=1; i< m-1; i++){
            Anew[j][i] = ...(A[j][i+1]) ...
        }
    }
```
Laplace, OpenACC Kernels. Not Managed, Runtime Error

% pgcc -acc -fast -ta=tesla:cc70 -Minfo=accel,opt laplace2d.c jacobi.c -o jacobi_kernels

37, Accelerator restriction: size of the GPU copy of Anew,A is unknown

Loop carried dependence of Anew-> prevents parallelization
Loop carried dependence of Anew-> prevents vectorization
Loop carried backward dependence of Anew-> prevents vectorization
Generating implicit copyout(Anew[:]) [if not already present]
Generating implicit copyin(A[:]) [if not already present]

39, Loop is parallelizable

Generating Tesla code
37, #pragma acc loop seq
39, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
43, Generating implicit reduction(max:error)

Runtime error:
call to cuMemcpyDtoHAsync returned error 700: Illegal address during kernel execution
call to cuMemFreeHost returned error 700: Illegal address during kernel execution
Laplace OpenACC Kernels, Managed

% pgcc -acc -fast -ta=tesla:cc70,managed -Minfo=accel,opt laplace2d.c jacobi.c -o jacobi_kernels_managed

36, Generating implicit copyin(A[:]) [if not already present]
Generating implicit copyout(Anew[:]) [if not already present]

37, Loop carried dependence of Anew-> prevents parallelization
Loop carried dependence of Anew-> prevents vectorization
Loop carried backward dependence of Anew-> prevents vectorization

39, Loop is parallelizable

Generating Tesla code

37, #pragma acc loop seq
39, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */

43, Generating implicit reduction(max:error)

Slower than the unmanaged version of unstructured data with 0.863 sec

<table>
<thead>
<tr>
<th>Time(%)</th>
<th>Time</th>
<th>Calls</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Nam</th>
</tr>
</thead>
<tbody>
<tr>
<td>58.34%</td>
<td>975.07ms</td>
<td>1000</td>
<td>975.07us</td>
<td>847.70us</td>
<td>95.594ms</td>
<td>calcNext_39_gpu</td>
</tr>
<tr>
<td>41.37%</td>
<td>691.33ms</td>
<td>1000</td>
<td>691.33us</td>
<td>682.30us</td>
<td>702.04us</td>
<td>swap_54_gpu</td>
</tr>
<tr>
<td>0.13%</td>
<td>2.1075ms</td>
<td>1000</td>
<td>2.1070us</td>
<td>2.0470us</td>
<td>12.448us</td>
<td>calcNext_43_gpu__red</td>
</tr>
<tr>
<td>0.08%</td>
<td>1.4173ms</td>
<td>1000</td>
<td>1.4170us</td>
<td>1.3760us</td>
<td>2.5280us</td>
<td>[CUDA memcpy DtoH]</td>
</tr>
<tr>
<td>0.08%</td>
<td>1.2969ms</td>
<td>1000</td>
<td>1.2960us</td>
<td>1.2480us</td>
<td>1.9520us</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
</tbody>
</table>

Total: 1.68 sec
update Directive

- You can update data between host and device memory.
- Useful when you want to synchronize data in the middle of a data region

```c
#include<omp.h>

#pragma acc data map(to: A,B) map(from: C) 
{ 
    #pragma acc parallel
    {do lots of stuff with A, B and C}

    #pragma acc update self(A)

    host_do_something_with(A)

    #pragma acc update device(A)

    #pragma acc parallel
    {do lots of stuff with A, B, and C}
} 

#pragma omp target data map(to: A,B) map(from: C) 
{ 
    #pragma omp target
    {do lots of stuff with A, B and C}

    #pragma omp target update from(A)

    host_do_something_with(A)

    #pragma omp target update to(A)

    #pragma omp target
    {do lots of stuff with A, B, and C}
} 
```
OpenACC and OpenMP Progress and Timeline

- OpenACC started initially to focus on accelerator performance and quicker specification turnaround
- OpenMP target offload adopts important features from OpenACC

<table>
<thead>
<tr>
<th>Year</th>
<th>OpenMP</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>OpenMP 3.0</td>
<td>OpenACC 1.0</td>
</tr>
<tr>
<td>2009</td>
<td>OpenMP 3.1</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>OpenMP 4.0</td>
<td>OpenACC 2.0</td>
</tr>
<tr>
<td>2014</td>
<td>OpenMP 4.5</td>
<td>OpenACC 2.5</td>
</tr>
<tr>
<td>2015</td>
<td>OpenMP 5.0</td>
<td>OpenACC 2.6</td>
</tr>
<tr>
<td>2016</td>
<td></td>
<td>OpenACC 2.7</td>
</tr>
<tr>
<td>2017</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2018</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OpenACC 3.0 (minor updates) released in Nov 2019

Table courtesy of Oscar Hernandez
OpenACC Resources and Compilers

Available on Perlmutter:
-- PGI, GCC, Cray (deprecated since CCE9)

Available on other DOE systems:
-- PGI, GCC, Cray (deprecated)
OpenMP Resources and Compilers

Available on Perlmutter:
-- PGI, Cray, Clang, GCC, Flang

Available on other DOE systems:
-- Cray, Clang, GCC, PGI, Flang, IBM, Intel, AMD

The complete list is available at:
https://www.openmp.org/resources/openmp-compilers-tools/
• Critical for application developers to consider portable (and even better performance portable) solutions which can target different platforms across vendors.
• OpenMP is an open standard supported by nearly every vendor. Multiple Compilers will support a common set of OpenMP directives on GPUs
  – LLVM/Clang 10
  – AMD (mostly tracks LLVM)
  – Cray (CCE 10)
  – IBM (XL V16.1.6)
  – Intel (Approximately 2021 timeframe),
  – Nvidia/PGI (Early 2021 for a production release through the NERSC/PGI NRE)
## OpenACC <=> OpenMP Conversion

<table>
<thead>
<tr>
<th>OpenACC Syntax</th>
<th>OpenMP Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc parallel</td>
<td>omp [target] teams</td>
</tr>
<tr>
<td>acc loop independent</td>
<td>omp loop</td>
</tr>
<tr>
<td>acc loop gang</td>
<td>omp distribute order(concurrent)</td>
</tr>
<tr>
<td>acc loop worker</td>
<td>omp parallel for order(concurrent)</td>
</tr>
<tr>
<td>acc loop vector</td>
<td>omp simd order(concurrent)</td>
</tr>
<tr>
<td>acc parallel loop</td>
<td>omp [target] teams loop</td>
</tr>
<tr>
<td>acc copyin(), copyout(), copy()</td>
<td>omp map(to:), map(from:), map(tofrom:)</td>
</tr>
<tr>
<td>acc data, acc end data</td>
<td>omp target data, omp end target data</td>
</tr>
<tr>
<td>acc enter data, acc exit data</td>
<td>omp target enter data, omp target exit data</td>
</tr>
<tr>
<td>acc update host(), acc update device()</td>
<td>omp target update to(), omp target update from()</td>
</tr>
</tbody>
</table>
GPP mini-app from BerkeleyGW

- 3 OpenMP compilers obtain similar performance to CUDA
- GPP has 10x higher performance when using a NVIDIA V100 GPU
- GCC GPU-offload not shown

Time = 28.4 seconds

Courtesy of Chris Daley, Rahul Gayatri
## ACCEL Benchmarks 1.2 (offload)

<table>
<thead>
<tr>
<th>SPEC ACCEL 1.2 Benchmark</th>
<th>Summit</th>
<th>Titan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencil (C)</td>
<td>1.12</td>
<td>0.67</td>
</tr>
<tr>
<td>LBM (C)</td>
<td>1.77</td>
<td>1.96</td>
</tr>
<tr>
<td>MRI-Q (C)</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>MD (Fortran)</td>
<td>0.14</td>
<td></td>
</tr>
<tr>
<td>EP (Fortran)</td>
<td>0.27</td>
<td>1.07</td>
</tr>
<tr>
<td>CLVRLEAF (C, Fortran)</td>
<td>0.88</td>
<td></td>
</tr>
<tr>
<td>CG (C)</td>
<td>0.32</td>
<td>1.12</td>
</tr>
<tr>
<td>SEISMIC (C)</td>
<td>1.49</td>
<td></td>
</tr>
<tr>
<td>SP (Fortran)</td>
<td>1.27</td>
<td></td>
</tr>
<tr>
<td>SP (C)</td>
<td>0.21</td>
<td>0.54</td>
</tr>
<tr>
<td>MinGhost (C, Fortran)</td>
<td>2.18</td>
<td></td>
</tr>
<tr>
<td>LBDC (Fortran)</td>
<td>2.45</td>
<td></td>
</tr>
<tr>
<td>Swim (Fortran)</td>
<td>1.18</td>
<td></td>
</tr>
<tr>
<td>BT (C)</td>
<td>0.12</td>
<td>0.7</td>
</tr>
</tbody>
</table>

HPGMG-FV

- Initial OpenMP offload performance with IBM compiler is encouraging
- Easy to translate from CUDA launch configuration to OpenACC/OpenMP loops
- Several compiler bugs found with strict correctness tests
Monte Carlo Particle Transport Case Study: XSBench

- OpenMP offloading relatively easy to implement
- Available compilers worked well – no bugs.
- IBM XL compiler generated fast code (better than naive CUDA implementation!)
- However, clang-ykt was 2.5x slower than IBM XL

XSBench FOM Performance on V100
(Higher is Better)

- CUDA
- CUDA (Optimized)
- OpenCL
- hipSYCL
- OpenMP Offload (Xeon host, clang-ykt)
- OpenMP Offload (Power9 host, xlc_r)

Courtesy of John Tramm
Best Practices

- Profile the sequential code, choose hot spots, give enough work for GPU to do. Offload computational intensive work, such as large loops.
- Use compiler hint, and remove loop carried dependencies to help parallelizing loops.
- Reorder loops or transpose arrays if needed to expose SIMD/SIMT in outer most loops.
- Keeping data resident on the device for the greatest possible time.
- Collapsing loops with the collapse clause, so there is a large enough iteration space to saturate the device.
- To utilize warps in OpenACC, always make sure vector length is a multiple of 32.
- Try different compilers, such as PGI for OpenACC, XL/CCE/Clang/GCC for OpenMP.

- 55 -
Summary

- OpenACC has been used widely in many applications (since Titan). Mature implementation for Nvidia GPU.
- More OpenMP compilers available than OpenACC.
- OpenMP 5.0 compilers will have more mature support for accelerators, expect to see great improvements in quality and more devices to use.
- Performance with OpenMP 4.5 is on par or catching up with OpenACC and native CUDA.
- PGI OpenMP upcoming, leverage OpenACC implementation expertise.
- Transition from OpenACC to OpenMP is relatively straightforward.
- OpenMP 5.0 improvement provides more convenient porting from OpenACC to OpenMP.
Recommendation

- OpenACC is good for Nvidia GPUs, on Summit and Perlmutter, for example.
- OpenMP is good for all DOE Exascale computers, such as Perlmutter, ORNL Frontier, and ANL Aurora.
- NERSC users have used OpenMP for past systems, easier to use OpenMP for continuity. MPI + OpenMP is a successful programming model for Cori.
- Recommend to use OpenMP on Perlmutter, especially with PGI now on board (timing lines up with when Perlmutter is arriving).
Thank you.