INTRODUCTION TO ACCELERATED COMPUTING USING COMPILER DIRECTIVES

Jeff Larkin
OUTLINE

▪ What are compiler directives?
▪ OpenACC 2.0 and OpenMP 4.0
▪ Compiler directives by example
  – OpenACC `parallel` and `kernels`
  – Profiling an application
  – OpenACC data optimization
▪ What’s next
WHAT ARE COMPILER DIRECTIVES?
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

Your original Fortran, C, or C++ code

#pragma acc parallel loop
for (int i = 0; i<n; ++i) {
    ... parallel code ...
}
... serial code ...
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
WHY SUPPORT COMPILER DIRECTIVES?

Increased Usability, Broader Adoption

1,000,000’s

100,000’s

Early Adopters

- 1,000,000’s
- 100,000’s
- Early Adopters

- Universities
- Supercomputing Centers
- Oil & Gas
- Research

- CAE
- CFD
- Finance
- Rendering
- Data Analytics
- Life Sciences
- Defense
- Weather
- Climate
- Plasma Physics

Present

Increased Usability, Broader Adoption
3 WAYS TO ACCELERATE APPLICATIONS

- Libraries: “Drop-in” Acceleration
- Compiler Directives: Easily Accelerate Applications
- Programming Languages: Maximum Flexibility
OPENACC 2.0 AND OPENMP 4.0
OPENACC 2.0

- 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`
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(2**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
  ```
SAXPY in C

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma omp target teams \
    distribute parallel for
    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
    !$omp target teams &
    !$omp& distribute parallel do
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$omp end target teams &
    !$omp& distribute parallel do
end subroutine saxpy
...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x, y)
...`
COMPILER DIRECTIVES BY EXAMPLE
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: $\nabla^2 f(x, y) = 0$

\[
A_{k+1}(i, j) = \frac{A_k(i - 1, j) + A_k(i + 1, j) + A_k(i, j - 1) + A_k(i, j + 1)}{4}
\]
JACOBI ITERATION: C CODE

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++;
}
IDENTIFY PARALLELISM
JACOBI ITERATION: 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++ ) {
            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

Independent loop iterations

Data dependency between iterations.

Express Parallelism

Express Data Locality

Optimize
EXPRESS PARALLELISM

- Identify Parallelism
- Express Parallelism
- Express Data Locality
- Optimize
while (err > tol && iter < iter_max) {
  err=0.0;

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

  iter++;
}
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
JACOBI ITERATION: OPENACC C CODE

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

```c
#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++;
OPENACC 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.
$ 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
Why is OpenACC so much slower?
$ nvprof ./laplace2d_acc
Jacobi relaxation Calculation: 4096 x 4096 mesh

==10619== NVPROF is profiling process 10619, command: ./laplace2d_acc
  0,  0.250000
 100,  0.002397
 200,  0.001204
 300,  0.000804
 400,  0.000603
 500,  0.000483
 600,  0.000403
 700,  0.000345
 800,  0.000302
 900,  0.000269

Total: 134.259326 s

==10619== Profiling application: ./laplace2d_acc

==10619== Profiling result:

<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>49.59%</td>
<td>44.0095s</td>
<td>17000</td>
<td>2.5888ms</td>
<td>864ns</td>
<td>2.9822ms</td>
<td>[CUDA memcpy HtoD]</td>
</tr>
<tr>
<td>45.06%</td>
<td>39.9921s</td>
<td>17000</td>
<td>2.3525ms</td>
<td>2.4960us</td>
<td>2.7687ms</td>
<td>[CUDA memcpyDtoH]</td>
</tr>
<tr>
<td>2.95%</td>
<td>2.61622s</td>
<td>1000</td>
<td>2.6162ms</td>
<td>2.6044ms</td>
<td>2.6319ms</td>
<td>main_56_gpu</td>
</tr>
<tr>
<td>2.39%</td>
<td>2.11884s</td>
<td>1000</td>
<td>2.1188ms</td>
<td>2.1023ms</td>
<td>2.1374ms</td>
<td>main_68_gpu</td>
</tr>
<tr>
<td>0.01%</td>
<td>12.431ms</td>
<td>1000</td>
<td>12.430us</td>
<td>12.192us</td>
<td>12.736us</td>
<td>main_63_gpu_red</td>
</tr>
</tbody>
</table>
PROFILING AN OPENACC APPLICATION

```
$ PGI_ACC_TIME=1 ./laplace2d_acc
Accelerator Kernel Timing data
/home/jlarkin/openacc-workshop/exercises/001-laplace2D-parallel/laplace2d.c
  main NVIDIA devicenum=0
time(us): 89,242,926
  56: compute region reached 1000 times
  56: data copyin reached 8000 times
     device time(us): total=22,334,806 max=3,022 min=2,747 avg=2,791
  56: kernel launched 1000 times
     grid: [4094]  block: [256]
     device time(us): total=2,643,298 max=2,841 min=2,629 avg=2,643
     elapsed time(us): total=2,654,729 max=2,855 min=2,640 avg=2,654
  56: reduction kernel launched 1000 times
     grid: [1]  block: [256]
     device time(us): total=19,182 max=75 min=17 avg=19
     elapsed time(us): total=29,669 max=87 min=28 avg=29
  68: data copyout reached 8000 times
     device time(us): total=20,100,500 max=2,797 min=2,494 avg=2,512
  68: compute region reached 1000 times
  68: data copyin reached 8000 times
     device time(us): total=21,902,474 max=2,939 min=2,724 avg=2,737
  68: kernel launched 1000 times
     grid: [4094]  block: [256]
     device time(us): total=2,144,960 max=2,202 min=2,130 avg=2,144
     elapsed time(us): total=2,157,152 max=2,215 min=2,143 avg=2,157
  77: data copyout reached 8000 times
     device time(us): total=20,097,706 max=2,809 min=2,494 avg=2,512
```
NVIDIA VISUAL PROFILER: NEW SESSION

1. New Session
   Icon or File -> New Session

2. Select Executable

3. Next
NVIDIA VISUAL PROFILER: NEW SESSION (CONT.)

4. Finish
NVIDIA VISUAL PROFILER: GUIDED ANALYSIS

2. CUDA Application Analysis

The guided analytic system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application, it is important to fully utilize the complete and daily movement of your GPU usage (as well as the performance of individual names).

- Examine GPU Usage
  - Determine your application’s overall GPU usage. This analysis requires an application profile, so your application will be run once to collect data if it is not already available.

- Examine Individual Elements
  - Determine the best elements in the application are the most performance critical and have the most opportunities for improvement. This analysis requires application data that has run once, on your application will be run once to collect data if it is not already available.

- Delete Existing Analysis Information
  - If the application has changed since the last analysis then the existing analysis information may be invalid and should be deleted before continuing.

- Switch to unguided analysis
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]));
        }
    }
}

A, Anew resident on host

A, Anew resident on accelerator

These copies happen every iteration of the outer while loop!*

A, Anew resident on host

A, Anew resident on accelerator

... 

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

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>copy</strong> <em>(list)</em></td>
<td>Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.</td>
</tr>
<tr>
<td><strong>copyin</strong> <em>(list)</em></td>
<td>Allocates memory on GPU and copies data from host to GPU when entering region.</td>
</tr>
<tr>
<td><strong>copyout</strong> <em>(list)</em></td>
<td>Allocates memory on GPU and copies data to the host when exiting region.</td>
</tr>
<tr>
<td><strong>create</strong> <em>(list)</em></td>
<td>Allocates memory on GPU but does not copy.</td>
</tr>
<tr>
<td><strong>present</strong> <em>(list)</em></td>
<td>Data is already present on GPU from another containing data region.</td>
</tr>
</tbody>
</table>

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 be used on data, parallel, or kernels
JACOBI ITERATION: OPENACC C CODE

```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++ ) {
        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
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.
VISUAL PROFILER: REDUCED DATA MOVEMENT

Was 132ms
Speed-Up (Higher is Better)

- Single Thread
- 2 Threads
- 4 Threads
- 6 Threads
- OpenACC
OPENACC **PRESENT** CLAUSE

It’s sometimes necessary for a data region to be in a different scope than the compute region.

When this occurs, the **present** clause can be used to tell the compiler data is already on the device.

Since the declaration of A is now in a higher scope, it’s necessary to shape A in the present clause.

High-level data regions and the present clause are often critical to good performance.

```c
function laplace2D(double[N][M] A,n,m)
{
  #pragma acc data present(A[n][m]) create(Anew)
  while ( err > tol && iter < iter_max ) {
    err=0.0;
    ...
  }
}
```

```c
function main(int argc, char **argv)
{
  #pragma acc data copy(A)
  {
    laplace2D(A,n,m);
  }
}
```

It’s sometimes necessary for a data region to be in a different scope than the compute region.

When this occurs, the **present** clause can be used to tell the compiler data is already on the device.

Since the declaration of A is now in a higher scope, it’s necessary to shape A in the present clause.

High-level data regions and the present clause are often critical to good performance.
OPTIMIZE
The **loop** directive provides the compiler with additional information for the next loop.

**Notable Clauses:**
- `private` & `reduction`
- `gang/worker/vector/seq`
- `collapse`
- `tile`
The private and reduction clauses are not optimization clauses, they may be required for correctness.

- **private** – A copy of the variable is made for each loop iteration
- **reduction** – A reduction is performed on the listed variables.
  - Supports +, *, max, min, and various logical operations
**OPENACC LOOP DIRECTIVE: LOOP PARALLELISM**

- OpenACC supports 3 levels of parallelism (gang, worker, vector).

**Parallel Loop**
- #pragma acc parallel loop num_gangs(X)
- num_workers(Y) vector_length(Z)
- #pragma acc loop seq

**Kernels**
- #pragma acc kernels
- #pragma acc loop gang(X)
- #pragma acc loop worker(Y)
- #pragma acc loop vector(Z)
- #pragma acc loop seq

---

**Identify Parallelism**

**Express Parallelism**

**Express Data Locality**

**Optimize**
OPENACC: 3 LEVELS OF PARALLELISM

- **Vector** threads work in lockstep (SIMD/SIMT parallelism)
- **Workers** have 1 or more vectors.
- **Gangs** have 1 or more workers and share resources (such as cache, the streaming multiprocessor, etc.)
- Multiple gangs work independently of each other

On NVIDIA GPUs, vector lengths should be a multiple of 32 and the worker dimension can usually be ignored.
The **collapse** directive instructs the compiler to convert the next $N$ loops into one flattened loop.

- This is especially useful when some loops lack enough iterations to make effective use of the GPU.

```c
#pragma acc parallel loop collapse(2)
for(int i=0; i<N; i++)
{
 for(int j=0; j<4; j++)
   ...
}
```

```c
#pragma acc parallel for
for(int ij=0; ij<4*N; ij++)
{
   ...
}
```
The **tile** directive instructs the compiler to block the following loops to better exploit locality and data reuse.

- The compiler will generate additional tile loops according to the tile dimensions provided.

```c
#pragma acc parallel loop
for(int i=0; i < n; i++)
  #pragma acc loop tile(8,8)
  for(int j=0; j < n; j++)
    for(int k=0; k < n; k++)
      c[i][j] = c[i][j] + a[i][j] * b[k][j];
```
### Identify Parallelism
- What important parts of the code have available parallelism?

### Express Parallelism
- Express as much parallelism as possible and ensure you still get correct results.
- Because the compiler *must* be cautious about data movement, the code will generally slow down.

### Express Locality
- The programmer will *always* know better than the compiler what data movement is unnecessary.

### Optimize
- Don’t try to optimize a kernel that runs in a few *us* or *ms* until you’ve eliminated the excess data motion that is taking *many seconds*.
NEXT STEPS: TRY OPENACC

- GTC14 has multiple hands-on labs for trying OpenACC
  - S4803 - Getting Started with OpenACC - Wednesday 5PM
  - S4796 - Parallel Programming: OpenACC Profiling - Thursday 2PM
- Get a PGI Trial license for your machine
- NVIDIA Developer Blogs
  - Parallel Forall Blog
  - CUDACasts Screencast Series
MORE AT GTC14

- S4200 - Advanced Accelerated Computing Using Directives
  - 1PM in this room
  - Continues where this talk has left off

- S4514 - Panel on Compiler Directives for Accelerated Computing
  - Wednesday @ 4PM
  - Wednesday afternoon in LL20C has all compiler-directives talks

- Check the sessions agenda for more talks on compiler directives

Please remember to fill out your surveys.