Introduction to Compiler Directives with OpenACC
Agenda

- Fundamentals of Heterogeneous & GPU Computing
- What are Compiler Directives?
- Accelerating Applications with OpenACC
  - Identifying Available Parallelism
  - Exposing Parallelism
  - Optimizing Data Locality
- Misc. Tips
- Next Steps
Heterogeneous Computing Basics
What is Heterogeneous Computing?

Application Execution

- High Serial Performance
- High Data Parallelism

CPU

GPU
Low Latency or High Throughput?
Latency vs. Throughput

**F-22 Raptor**
- 1500 mph
- Knoxville to San Jose in 1:25
- Seats 1

**Boeing 737**
- 485 mph
- Knoxville to San Jose in 4:20
- Seats 200
Latency vs. Throughput

**F-22 Raptor**
- Latency – 1:25
- Throughput – 1 / 1.42 hours = 0.7 people/hr.

**Boeing 737**
- Latency – 4:20
- Throughput – 200 / 4.33 hours = 46.2 people/hr.
Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other threads
Accelerator Fundamentals

- We must expose enough parallelism to fill the device
  - Accelerator threads are slower than CPU threads
  - Accelerators have orders of magnitude more threads
  - Accelerators tolerate resource latencies by cheaply context switching threads
- Fine-grained parallelism is good
  - Generates a significant amount of parallelism to fill hardware resources
- Coarse-grained parallelism is bad
  - Lots of legacy apps have only exposed coarse grain parallelism
3 Approaches to Heterogeneous Programming

- Libraries
  - Easy to use
  - Most Performance

- Compiler Directives
  - Easy to use
  - Portable code

- Programming Languages
  - Most Performance
  - Most Flexibility
Simplicity & Performance

- **Accelerated Libraries**
  - Little or no code change for standard libraries, high performance.
  - Limited by what libraries are available

- **Compiler Directives**
  - Based on existing programming languages, so they are simple and familiar.
  - Performance may not be optimal because directives often do not expose low level architectural details

- **Parallel Programming languages**
  - Expose low-level details for maximum performance
  - Often more difficult to learn and more time consuming to implement.
What are Compiler Directives?
What are Compiler Directives?

Programmer inserts compiler hints.
Execution Begins on the CPU.

Compiler Generates GPU Code

Data and Execution moves to the GPU.

Data and Execution returns to the CPU.

```c
int main() {
    do_serial_stuff()

    for(int i=0; i < BIGN; i++)
    {
        ...compute intensive work
    }

    do_more_serial_stuff();
}
```
OpenACC: The Standard for GPU Directives

- Simple: Directives are the easy path to accelerate compute intensive applications
- Open: OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- Portable: GPU Directives represent parallelism at a high level, allowing portability to a wide range of architectures with the same code.
OpenACC Members and Partners
Focus on Parallelism and locality

Example: Application tuning work using directives for Titan system at ORNL

**S3D**
Research more efficient combustion with next-generation fuels

- Tuning top 3 kernels (90% of runtime)
- 3 to 6x faster on CPU+GPU vs. CPU+CPU
- But also improved all-CPU version by 50%

**CAM-SE**
Answer questions about specific climate change adaptation and mitigation scenarios

- Tuning top key kernel (50% of runtime)
- 6.5x faster on CPU+GPU vs. CPU+CPU
- Improved performance of CPU version by 100%
- Work was done in CUDA Fortran (not OpenACC)
Accelerating Applications with OpenACC
Identify Available Parallelism
Parallelize Loops with OpenACC
Optimize Data Locality
Optimize Loop Performance
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:

\[
A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}
\]
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++;
}
Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Loop Performance

Optimize Data Locality
Identify Available Parallelism

- A variety of profiling tools are available:
  - gprof, pgprof, Vampir, Score-p, HPCToolkit, CrayPAT, …
- Using the tool of your choice, obtain an application profile to identify hotspots
- Since we’re using PGI, I’ll use pgprof

```bash
$ pgcc -fast -Minfo=all -Mprof=ccff laplace2d.c
main:
  40, Loop not fused: function call before adjacent loop
      Generated vector sse code for the loop
  57, Generated an alternate version of the loop
      Generated vector sse code for the loop
      Generated 3 prefetch instructions for the loop
  67, Memory copy idiom, loop replaced by call to __c_mcopy8

$ pgcollect ./a.out
$ pgprof -exe ./a.out
```
Identify Parallelism With PGPROF

PGPROF informs us:
1. A significant amount of time is spent in the loops at line 56/57.
2. The computational intensity (Calculations/Loads&Stores) is high enough to warrant OpenACC or CUDA.
3. How the code is currently optimized.

NOTE: the compiler recognized the swapping loop as data movement and replaced it with a memcpy, but we know it’s expensive too.
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 Available Parallelism

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Loop Performance
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
  ```

Don’t forget `acc`
OpenACC parallel loop Directive

parallel - Programmer identifies a block of code containing parallelism. Compiler generates a kernel.

loop - Programmer identifies a loop that can be parallelized within the kernel.

NOTE: parallel & loop are often placed together

```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
Parallelize with OpenACC

```c
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++;
}
```

* A reduction means that all of the N*M values for err will be reduced to just one, the max.
OpenACC loop directive: private & reduction

- 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
Building the Code

$ 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, Loop not vectorized/parallelized: potential early exits
55, Accelerator kernel generated
   55, Max reduction generated for error
   56, #pragma acc loop gang /* blockIdx.x */
   58, #pragma acc loop vector(256) /* threadIdx.x */
55, Generating copyout(Anew[1:4094][1:4094])
   Generating copyin(A[:, :])
   Generating Tesla code
58, Loop is parallelizable
66, Accelerator kernel generated
   67, #pragma acc loop gang /* blockIdx.x */
   69, #pragma acc loop vector(256) /* threadIdx.x */
66, Generating copyin(Anew[1:4094][1:4094])
   Generating copyout(A[1:4094][1:4094])
   Generating Tesla code
69, Loop is parallelizable
OpenACC kernels Directive

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++)
{
    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.
Parallelize with OpenACC 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 -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, 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 */
OpenACC parallel loop vs. kernels

**PARALLEL LOOP**

- Requires analysis by programmer to ensure safe parallelism
- Will parallelize what a compiler may miss
- 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 to optimize

Both approaches are equally valid and can perform equally well.
Why did OpenACC slow down here?

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

- Any tool that supports CUDA can likewise obtain performance information about OpenACC.
- NVIDIA Visual Profiler (nvvp) comes with the CUDA Toolkit, so it will be available on any machine with CUDA installed.
Very low Compute/Memcpy ratio

Compute: 4.7s
Memory Copy: 84.3s
1. Copy input data from CPU memory/NIC to GPU memory
Processing Flow

1. Copy input data from CPU memory/NIC to GPU memory
2. Execute GPU Kernel
1. Copy input data from CPU memory/NIC to GPU memory
2. Execute GPU Kernel
3. Copy results from GPU memory to CPU memory/NIC
One step of the convergence loop

Iteration 1

Iteration 2
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

  These copies happen every iteration of the outer while loop!

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

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++;
}

Does the CPU need the data between these loop nests?

Does the CPU need the data between iterations of the convergence loop?
Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Loop Performance
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.

```
#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**.

The next OpenACC makes **present_or_*** the default behavior.
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:size]), 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
Optimize Data Locality

```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.
Rebuilding the Code

$ 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

Was 128ms
Intel Xeon E5-2698 v3 @ 2.30GHz (Haswell) vs. NVIDIA Tesla K40

Socket/Socket: 6.24X

Speed-Up (Higher is Better)
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.
Unstructured Data Directives

Used to define data regions when scoping doesn’t allow the use of normal data regions (e.g. The constructor/destructor of a class).

**enter data** Defines the start of an unstructured data lifetime clauses: **copyin(list), create(list)**

**exit data** Defines the end of an unstructured data lifetime clauses: **copyout(list), delete(list)**

```plaintext
#pragma acc enter data copyin(a)
...
#pragma acc exit data delete(a)
```
Unstructured Data Regions: C++ Classes

- Unstructured Data Regions enable OpenACC to be used in C++ classes
- Unstructured data regions can be used whenever data is allocated and initialized in a different scope than where it is freed.

```cpp
class Matrix {
    Matrix(int n) {
        len = n;
        v = new double[len];
        #pragma acc enter data create(v[0:len])
    }

    ~Matrix() {
        #pragma acc exit data delete(v[0:len])
        delete[] v;
    }

    private:
    double* v;
    int len;
};
```
Identify Available Parallelism

Parallelize Loops with OpenACC

Optimize Data Locality

Optimize Loop Performance
Aliasing Can Prevent Parallelization

23, Loop is parallelizable

Accelerator kernel generated
23, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */

25, Complex loop carried dependence of 'b->' prevents parallelization
Loop carried dependence of 'a->' prevents parallelization
Loop carried backward dependence of 'a->' prevents vectorization
Accelerator scalar kernel generated

27, Complex loop carried dependence of 'a->' prevents parallelization
Loop carried dependence of 'b->' prevents parallelization
Loop carried backward dependence of 'b->' prevents vectorization
Accelerator scalar kernel generated
C99: restrict Keyword

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g.
  - `float *restrict ptr`
  - Meaning: “for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points”*

- Parallelizing compilers often require restrict to determine independence
  - Otherwise the compiler can’t parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined

OpenACC independent clause

Specifies that loop iterations are data independent. This overrides any compiler dependency analysis. This is implied for parallel loop.

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

Informs the compiler that both loops are safe to parallelize so it will generate both kernels.
Write Parallelizable Loops

- Use countable loops
  - C99: while->for
  - Fortran: while->do

- Avoid pointer arithmetic

- Write rectangular loops
  (compiler cannot parallelize triangular lops)

```c
bool found=false;
while(!found && i<N) {
    if(a[i]==val) {
        found=true
        loc=i;
    }
    i++;
}
```

```c
for(int i=0;i<N;i++) {
    if(a[i]==val) {
        found=true
        loc=i;
    }
}
```

```c
for(int i=0;i<N;i++) {
    for(int j=i;j<N;j++) {
        sum+=A[i][j];
    }
}
```

```c
for(int i=0;i<N;i++) {
    for(int j=0;j<N;j++) {
        if(j>=i)
            sum+=A[i][j];
    }
}
```
OpenACC Routine Directive

The routine directive specifies that the compiler should generate a device copy of the function/subroutine in addition to the host copy and what type of parallelism the routine contains.

Clauses:

- **gang/worker/vector/seq**
  - Specifies the level of parallelism contained in the routine.
- **bind**
  - Specifies an optional name for the routine, also supplied at call-site
- **no_host**
  - The routine will only be used on the device
- **device_type**
  - Specialize this routine for a particular device type.
OpenACC Debugging

- Most OpenACC directives accept an if(condition) clause

```c
#pragma acc update self(A) if(debug)
#pragma acc parallel loop if(!debug)
```

- Use default(none) to force explicit data directives

```c
#pragma acc data copy(...) create(...) default(none)
```
Next Steps
1. Identify Available Parallelism
   - What important parts of the code have available parallelism?

2. Parallelize Loops
   - 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.

3. Optimize Data Locality
   - The programmer will *always* know better than the compiler what data movement is unnecessary.

4. Optimize Loop Performance
   - 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*. 
Typical Porting Experience with OpenACC Directives

- Step 1: Identify Available Parallelism
- Step 2: Parallelize Loops with OpenACC
- Step 3: Optimize Data Locality
- Step 4: Optimize Loops

Graph showing application speed-up over development time.