GPU COMPUTING WITH OPENACC
3 WAYS TO ACCELERATE APPLICATIONS

Applications

Libraries

“Drop-in” Acceleration

OpenACC Directives

Easily Accelerate Applications

Programming Languages

Maximum Flexibility
OPENACC DIRECTIVES

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original
Fortran or C code

Program myscience
... serial code ...
!$acc kernels
do k = 1,n1
do i = 1,n2
... parallel code ...
endo
dendo
!$acc end kernels
...
End Program myscience

CPU

GPU
FAMILIAR TO OPENMP PROGRAMMERS

OpenMP

```
main() {
    double pi = 0.0; long i;
    #pragma omp parallel for reduction(+:pi)
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n", pi/N);
}
```

OpenACC

```
main() {
    double pi = 0.0; long i;
    #pragma acc kernels
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi = %f\n", pi/N);
}
```
Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications.

"..." - Developer at the Global Manufacturer of Navigation Systems
FOCUS ON EXPOSING PARALLELISM

With Directives, tuning work focuses on exposing parallelism, which makes codes inherently better.

Example: Application tuning work using directives for new 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%
A VERY SIMPLE EXERCISE: SAXPY

**SAXPY in C**

```c
void saxpy(int n,
            float a,
            float *x,
            float *restrict y)
{
    #pragma acc kernels
    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(:), y(:), a
    integer :: n, i
    !$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
    !$acc end kernels
end subroutine saxpy

... $ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...```
DIRECTIVE SYNTAX

- Fortran
  \$acc directive [clause [,] clause] \ldots\$
  Often paired with a matching end directive surrounding a structured code block
  \$acc end directive

- C
  \#pragma acc directive [clause [,] clause] \ldots\$
  Often followed by a structured code block
KERNELS: YOUR FIRST OPENACC DIRECTIVE

Each loop executed as a separate kernel on the GPU.

```c
!$acc kernels
do i=1,n
  a(i) = 0.0
  b(i) = 1.0
  c(i) = 2.0
end do

Kernel 1

Kernel: A parallel function that runs on the GPU

Kernel 2

do i=1,n
  a(i) = b(i) + c(i)
end do

!$acc end kernels
```
KERNELS CONSTRUCT

Fortran

```
!$acc kernels [clause ...]
   structured block
!$acc end kernels
```

C

```
#pragma acc kernels [clause ...]
   { structured block }
```

Clauses

- `if( condition )`
- `async( expression )`

Also, any data clause (more later)
COMPLETE SAXPY EXAMPLE CODE

- Trivial first example
  - Apply a loop directive
  - Learn compiler commands

```c
#include <stdlib.h>

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

int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats

    if (argc > 1)
        N = atoi(argv[1]);

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));

    for (int i = 0; i < N; ++i)
    {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }

    saxpy(N, 3.0f, x, y);

    return 0;
}
```

*restrict: “I promise y does not alias x”
COMPILe AND RUN

- C: `pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c`
- Fortran: `pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90`
- Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
  8, Generating copyin(x[:n-1])
  Generating copyin(y[:n-1])
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  9, Loop is parallelizable
  Accelerator kernel generated
  9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
    CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
    CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```
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 ( error > tol && iter < iter_max ) {
    error=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]);

            error = max(error, 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++;
}
JACOBI ITERATION FORTRAN CODE

do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind
  do j=1,m
    do i=1,n
      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                              A(i , j-1) + A(i , j+1))
      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
  iter = iter +1
end do

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

General instructions (compiling)

- Exercises are in “exercises/openacc” directory
  - Solutions in “exercise_solutions/openacc” directory
- module load pgi/14.6
- To compile, use one of the provided makefiles
  - C: > make
  - Fortran: > make -f Makefile_f90
- Remember these flags
  - -acc -ta=nvidia -Minfo=accel
To run, use `sbatch` with one of the provided job files:

```bash
> sbatch runit.acc
> qstat -u <username>  # prints qsub status
```

Output is placed in `slurm.*` when finished.
EXERCISE 1

Jacobi kernels

- Task: use `acc kernels` to parallelize the Jacobi loop nests
- Edit `laplace2D.c` or `laplace2D.f90` (your choice)
  - In the `001-laplace2D-kernels` directory
  - Add directives where it helps
  - Figure out the proper compilation flags to use
  - Optionally: Run OpenACC version with `laplace_acc`
- Q: can you get a speedup with just kernels directives?
  - Versus 1 CPU core? Versus 6 CPU cores?
EXERCISE 1 SOLUTION: OPENACC C

```c
while ( error > tol && iter < iter_max ) {
    error=0.0;

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```
EXERCISE 1 SOLUTION: OPENACC FORTRAN

```fortran
do while ( error > tol .and. iter < iter_max )
    err=0._fp_kind

$!acc kernels
    do j=1,m
        do i=1,n
            Anew(i,j) = 0.25 * (A(i+1,j) + A(i-1,j) + &
                             A(i,j-1) + A(i,j+1))
            err = max(err, abs(Anew(i,j) - A(i,j));
        enddo
    enddo
$!acc end kernels

$!acc kernels
    do j=1, m-2
        do i=1, n-2
            A(i,j) = Anew(i,j)
        enddo
    enddo
$!acc end kernels

iter = iter+1
endo
```

Execute GPU kernel for loop nest

Execute GPU kernel for loop nest
EXERCISE 1: COMPILER OUTPUT (C)

pgcc -tp sandybridge-64 -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c
main:

56, Generating present_or_copyout(Anew[1:4094][1:4094])
    Generating present_or_copyin(A[:][:])
    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
68, Generating present_or_copyin(Anew[1:4094][1:4094])
    Generating present_or_copyout(A[1:4094][1:4094])
    Generating Tesla code
69, Loop is parallelizable
71, Loop is parallelizable
   Accelerator kernel generated
   69, #pragma acc loop gang /* blockIdx.y */
   71, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
## EXERCISE 1: PERFORMANCE

**CPU:** Intel E5-2680 v2  
10 Cores @ 2.80 GHz

**GPU:** NVIDIA Tesla K20m

<table>
<thead>
<tr>
<th>Execution (4096x4096)</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>108.7</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>58.6</td>
<td>1.85x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>31.9</td>
<td>3.41x</td>
</tr>
<tr>
<td>CPU 8 OpenMP threads</td>
<td>20.3</td>
<td>5.35x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>176.1</td>
<td>0.12x FAIL</td>
</tr>
</tbody>
</table>

**Speedup vs. 1 CPU core**

**Speedup vs. 8 CPU cores**
WHAT WENT WRONG?

PGI_ACC_TIME=1

time(us): 101,104,174
56: data region reached 1000 times
56: data copyin reached 8000 times
  device time(us): total=22,030,081 max=2,858 min=2,746 avg=2,753
68: data copyout reached 8000 times
  device time(us): total=23,018,701 max=6,552 min=2,855 avg=2,877
56: compute region reached 1000 times
59: kernel launched 1000 times
  grid: [32x4094]  block: [128]
  device time(us): total=6,456,517 max=6,516 min=6,447 avg=6,456
  elapsed time(us): total=6,471,110 max=7,066 min=6,460 avg=6,471
59: reduction kernel launched 1000 times
  grid: [1]  block: [256]
  device time(us): total=270,280 max=276 min=268 avg=270
  elapsed time(us): total=283,763 max=353 min=282 avg=283
68: data region reached 1000 times
68: data copyin reached 8000 times
  device time(us): total=23,271,701 max=2,946 min=2,891 avg=2,908
77: data copyout reached 8000 times
  device time(us): total=23,016,095 max=2,993 min=2,854 avg=2,877
68: compute region reached 1000 times
71: kernel launched 1000 times
  grid: [32x4094]  block: [128]
  device time(us): total=3,040,799 max=3,050 min=3,037 avg=3,046
  elapsed time(us): total=3,056,315 max=3,131 min=3,052 avg=3,056

Huge Data Transfer Bottleneck!

Computation: 10 seconds
Data movement: 90 seconds
For efficiency, decouple data movement and compute off-load
EXCESSIVE DATA TRANSFERS

```c
while ( error > tol && iter < iter_max ) {
    error=0.0;
    #pragma acc kernels
    for ( int j = 1; j < n-1; j++ ) {
        for (int i = 1; i < m-1; i++) {
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }
}
```

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

*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!*
Data Management
**DATA CONSTRUCT**

Fortran

```fortran
!$acc data [clause ...]
  structured block
!$acc end data
```

C

```c
#pragma acc data [clause ...]
  { structured block }
```

**General Clauses**

```plaintext
if( condition )

async( expression )
```

Manage data movement. Data regions may be nested.
<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy (list)</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>copyin (list)</td>
<td>Allocates memory on GPU and copies data from host to GPU when entering region.</td>
</tr>
<tr>
<td>copyout (list)</td>
<td>Allocates memory on GPU and copies data to the host when exiting region.</td>
</tr>
<tr>
<td>create (list)</td>
<td>Allocates memory on GPU but does not copy.</td>
</tr>
<tr>
<td>present (list)</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-1]), copyout(b[s/4:3*s/4])
- Fortran
  - !$pragma acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
- Note: data clauses can be used on data, kernels or parallel
EXERCISE 2: JACOBI DATA DIRECTIVES

- Task: use `acc data` to minimize transfers in the Jacobi example
- Start from given `laplace2D.c` or `laplace2D.f90` (your choice)
  - In the `002-laplace2d-data` directory
  - Add directives where it helps (hint: `[do] while loop`)

- Q: What speedup can you get with data + kernels directives?
  - Versus 6 CPU cores?
    - `OMP_NUM_THREADS=6 ./laplace2d_omp`
Exercise 2 Solution: OpenACC C

```c
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=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]);
            error = max(error, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc kernels
    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 in at beginning of loop, out at end. Allocate Anew on accelerator.
Exercise 2 Solution: OpenACC Fortran

```fortran
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                               A(i , j-1) + A(i , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
$acc end kernels

...

iter = iter +1
end do
$acc end data
```
## EXERCISE 2: PERFORMANCE

<table>
<thead>
<tr>
<th>Execution (4096x4096)</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 8 OpenMP thread</td>
<td>20.3</td>
<td>--</td>
</tr>
<tr>
<td>OpenACC K20m</td>
<td>176.1</td>
<td>0.12x</td>
</tr>
<tr>
<td>OpenACC K20m-opt</td>
<td>8.34</td>
<td>2.43x</td>
</tr>
</tbody>
</table>

Speedup vs. 8 CPU cores
WHAT WENT RIGHT?

Transfer Bottleneck Eliminated!
Computation: 10 seconds
Data movement: negligible
FURTHER SPEEDUPS

- OpenACC gives us more detailed control over parallelization
  - Via gang, worker, and vector clauses
- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance
- Will tackle these in later exercises
FINDING PARALLELISM IN YOUR CODE

- (Nested) for loops are best for parallelization.
- Large loop counts needed to offset GPU/memcpy overhead.
- Iterations of loops must be independent of each other.
  - To help compiler: restrict keyword (C), independent clause.
- Compiler must be able to figure out sizes of data regions.
  - Can use directives to explicitly control sizes.
- Pointer arithmetic should be avoided if possible.
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.
TIPS AND TRICKS

- (PGI) Use time option to learn where time is being spent
  - -ta=nvidia,time

- Eliminate pointer arithmetic

- Inline function calls in directives regions
  - (PGI): -inline or -inline,levels(<N>)

- Use contiguous memory for multi-dimensional arrays

- Use data regions to avoid excessive memory transfers

- Conditional compilation with _OPENACC macro