Directive based programming is (for all practical purposes)

- OpenMP
- OpenACC

Some of you just got an intro to classic OpenMP and some of you chose the advanced track - presumably because you already know OpenMP.

Problem: how do I not bore most of you without leaving some of you behind?
Solution!

- Start with OpenACC.
- It is simpler (more focused) than OpenMP.
- Finish with OpenMP 4.0
  - New and interesting
  - Accelerator oriented
Your original Fortran or C code

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

Directives

Simple compiler hints from coder.

Compiler generates parallel code.
OpenACC similar to OpenMP

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);
}

More on this later!
How Else Would We Accelerate GPU Applications?

Libraries

“Drop-in” Acceleration

OpenACC Directives

Incrementally Accelerate Applications

Programming Languages (CUDA)

Maximum Flexibility
Key Advantages Of This Approach

- High-level. No involvement of OpenCL, CUDA, etc.

- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial, non-GPU programmers can play along.

- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms.

- Performance portable. Supports GPU accelerators and co-processors from multiple vendors, current and future versions.

- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be quick.
A Few Cases

1. Reading DNA nucleotide sequences
   - Shanghai JiaoTong University
   - 4 directives
   - 16x faster

2. Designing circuits for quantum computing
   - UIST, Macedonia
   - 1 week
   - 40x faster

3. Extracting image features in real-time
   - Aselsan
   - 3 directives
   - 4.1x faster

4. HydroC- Galaxy Formation
   - PRACE Benchmark Code, CAPS
   - 1 week
   - 3x faster

5. Real-time Derivative Valuation
   - Opel Blue, Ltd
   - Few hours
   - 70x faster

6. Matrix Matrix Multiply
   - Independent Research Scientist
   - 4 directives
   - 6.4x faster
A Champion Case

4x Faster

Jaguar  Titan
42 days  10 days

Modified <1%
Lines of Code

Expected to be fastest scientific simulation ever done. 15 PF!

Design alternative fuels with up to 50% higher efficiency

S3D: Fuel Combustion
Comparison to Alternatives
Lattice-Boltzmann Example

![Comparison Graph]

Lines of code (main loops only)

<table>
<thead>
<tr>
<th></th>
<th>Ref</th>
<th>Opt</th>
<th>OpenMP</th>
<th>OpenACC</th>
<th>SSE</th>
<th>AVX</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time in seconds (lower is better)

<table>
<thead>
<tr>
<th></th>
<th>Optimised+OMP</th>
<th>SSE+OMP</th>
<th>AVX+OMP</th>
<th>OpenACC</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon 5650 x2 (12 cores) 2.66GHz</td>
<td>3.99</td>
<td>1.56</td>
<td>1.92</td>
<td>1.62</td>
<td></td>
</tr>
<tr>
<td>Core i5 IVB (4 cores) 3.3GHz</td>
<td>7.13</td>
<td>1.62</td>
<td>0.67</td>
<td>0.42</td>
<td></td>
</tr>
<tr>
<td>Tesla M2050</td>
<td>1.16</td>
<td>0.67</td>
<td></td>
<td></td>
<td>0.49</td>
</tr>
<tr>
<td>Nvidia GTX680</td>
<td>0.49</td>
<td>0.42</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Accelerator Support

- Xeon Phi support is planned and already undergoing testing and benchmarking. OpenMP (which we will discuss) is already there.

- AMD line of accelerated processing units (APUs) as well as the AMD line of discrete GPUs for preliminary PGI support in mid-2013.

- Carma - a hybrid platform based on ARM Cortex-A9 quad core and an NVIDIA Quadro® 1000M GPU.

- NVIDIA
A Simple Example: 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];
}
```

... // Somewhere in main
// call 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
```

... $ From main program
$ call SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
We request that each loop execute as a separate *kernel* on the GPU.

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

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

*Kernel:*
A parallel routine to run on the GPU.
General Directive Syntax and Scope

Fortran

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

C

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

clause:

```plaintext
if( condition )  
async( expression )  
data clause (more later)
```
C Detail: the `restrict` keyword

- Promise given by the programmer to the compiler for a pointer
  ```
  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”

- Limits the effects of pointer aliasing

- OpenACC 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
Complete SAXPY Example Code

```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 saxpy.c

Fortran:
pgf90 -acc -ta=nvidia -Minfo=accel saxpy.f90

Compiler Output

pgcc -acc -Minfo=accel -ta=nvidia saxpy.c
saxpy:
8, Generating copyin(x[:n-1])
  Generating copy(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
__global__ void saxpy_kernel( float a, float* x, float* y, int n ){
    int i;
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i <= n ) x[i] = a*x[i] + y[i];
}

void saxpy( float a, float* x, float* y, int n ){
    float *xd, *yd;
    cudaMalloc( (void**)&xd, n*sizeof(float) );
    cudaMalloc( (void**)&yd, n*sizeof(float) );
    cudaMemcpy( xd, x, n*sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( yd, y, n*sizeof(float), cudaMemcpyHostToDevice );
    saxpy_kernel<<< (n+31)/32, 32 >>>( a, xd, yd, n );
    cudaMemcpy( x, xd, n*sizeof(float), cudaMemcpyDeviceToHost );
    cudaMemcpy( y, yd, n*sizeof(float), cudaMemcpyDeviceToHost );
    cudaFree( xd );
    cudaFree( yd );
}
Compare: Partial CUDA Fortran SAXPY Code
Just the subroutine

module kmod
use cudafor
contains
attributes(global) subroutine saxpy_kernel(A,X,Y,N)
real(4), device :: A, X(N), Y(N)
integer, value :: N
integer :: i
i = (blockidx%x-1)*blockdim%x + threadIdx%x
if( i <= N ) X(i) = A*X(i) + Y(i)
end subroutine
end module

subroutine saxpy( A, X, Y, N )
use kmod
real(4) :: A, X(N), Y(N)
integer :: N
real(4), device, allocatable, dimension(:):: &
   Xd, Yd
allocate( Xd(N), Yd(N) )
Xd = X(1:N)
Yd = Y(1:N)
call saxpy_kernel<<<(N+31)/32,32>>>(A, Xd, Yd, N)
X(1:N) = Xd
deallocate( Xd, Yd )
end subroutine
```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;
}
```

Entire Subroutine

#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];
}
Big Difference!

- With CUDA, we changed the structure of the old code. Non-CUDA programmers can’t understand new code. It is not even ANSI standard code.

- We have separate sections for the host code, and the GPU code. Different flow of code. Serial path now gone forever.

- Where did these “32’s” and other mystery variables come from? This is a clue that we have some hardware details to deal with here.
This looks easy! Too easy...

- If it is this simple, why don’t we just throw *kernel* in front of every loop?
- Better yet, why doesn’t the compiler do this for me?

The answer is that there are two general issues that prevent the compiler from being able to just automatically parallelize every loop.

- Data Dependencies in Loops
- Data Movement

The compiler needs your higher level perspective (in the form of directive hints) to get correct results, and reasonable performance.
Most directive based parallelization consists of splitting up big do/for loops into independent chunks that the many processors can work on simultaneously.

Take, for example, a simple for loop like this:

```c
for(index=0, index<1000000,index++)
    Array[index] = 4 * Array[index];
```

When run on 1000 processors, it will execute something like this...
No Data Dependency

for(index=0, index<999, index++)
Array[index] = 4*Array[index];

for(index=1000, index<1999, index++)
Array[index] = 4*Array[index];

for(index=2000, index<2999, index++)
Array[index] = 4*Array[index];

for(index=3000, index<3999, index++)
Array[index] = 4*Array[index];

for(index=4000, index<4999, index++)
Array[index] = 4*Array[index];

....
Data Dependency

But what if the loops are not entirely independent?

Take, for example, a similar loop like this:

```plaintext
for(index=1, index<1000000,index++)
    Array[index] = 4 * Array[index] - Array[index-1];
```

This is perfectly valid serial code.
Data Dependency

Now Processor 2, in trying to calculate its first iteration,

\[
\text{for}(\text{index}=1000, \text{index}<1999, \text{index}++)
\]

\[
\text{Array}[1000] = 4 \times \text{Array}[1000] - \text{Array}[999];
\]

needs the result of Processor 1’s last iteration. If we want the correct (“same as serial”) result, we need to wait until processor 1 finishes. Likewise for processors 3, 4, ...
Data Dependencies

That is a data dependency. If the compiler even suspects that there is a data dependency, it will, for the sake of correctness, refuse to parallelize that loop.

As large, complex loops are quite common in HPC, especially around the most important parts of your code, the compiler will often balk most when you most need a kernel to be generated. What can you do?
Data Dependencies

• Rearrange your code to make it more obvious to the compiler that there is not really a data dependency.

• Eliminate a real dependency by changing your code.
  • There is a common bag of tricks developed for this as this issue goes back 40 years in HPC. Many are quite trivial to apply.
  • The compilers have gradually been learning these themselves.

• Override the compiler’s judgment (independent clause) at the risk of invalid results. Misuse of restrict has similar consequences.
Exercise Foundation: 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++) {


            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++;
}

Iterate until converged
Iterate across matrix elements
Calculate new value from neighbors
Compute max error for convergence
Swap old/current arrays
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 old/current arrays
OpenMP

Why are we involving OpenMP here?

- We can compare our GPU results to the best the multi-core XEON CPUs can do. It is also a free chance to see how OpenMP scales there.

- You already know some OpenMP from yesterday, or you can learn a little by accident as you learn OpenACC. They are very similar, and we will discuss the differences later.

- Because it is directive based, you can just totally ignore or delete the OpenMP statements and view these as strictly old-school serial codes. *This is exactly what other developers can do with your OpenACC directives!*
OpenMP C Code

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

#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++) {
            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 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++;
}
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
  do j=1,m
    do i=1,n
      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                 A(i , j-1) + A(i , j+1))
      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$omp parallel do shared(m,n,Anew,A)
  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
Exercises: General Instructions for Compiling

- Exercises are in the “exercises” directory in your home directory

- Solutions are in the “solutions” directory

- To compile, use one of the provided makefiles
  
  > cd exercises/001-laplace2D-kernels
  
  For C:
  
  > make

  For Fortran:
  
  > make -f Makefile_f90

- Remember these compiler flags:
  
  -acc -ta=nvidia -Minfo=accel
Exercises: General Instructions for Running

To run, use `qsub` with one of the provided job files

> qsub laplace_acc.job
> qstat # prints qsub status

Output is placed in `laplace_acc.job.o<job#>` when finished.

The OpenACC job file looks like this

```
#!/bin/csh
PBS -l walltime=3:00
./laplace2d_acc
```

The OpenMP job file specifies the number of cores to use

```
#!/bin/csh
PBS -l walltime=3:00
setenv OMP_NUM_THREADS 6
./laplace2d_omp
```
Exercise 1: Use kernels to parallelize the Jacobi loops
(About 30 minutes)

1) Edit laplace2D.c or laplace2D.f90 (your choice)
   Add directives where it helps

2) Edit Makefile or Makefile_f90 (Fortran)
   Add proper compilation flags (similar to SAXPY example)
   Compile with OpenACC parallelization
   Already compiles with OpenMP (original code has OpenMP directives)

3) Run
   OpenACC version with laplace_acc.job
   OpenMP version with laplace_omp.job

Q: Can you get a speedup with just kernels directives? Versus 1 CPU core? Versus 6 CPU cores?
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

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

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

iter = iter +1
end do
Exercise 1 Solution: C Makefile

CC       = pgcc
CCFLAGS  =
ACCFLAGS = -acc -ta=nvidia, -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN = laplace2d_omp laplace2d_acc

all: $(BIN)

laplace2d_acc: laplace2d.c
      $(CC) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_omp: laplace2d.c
      $(CC) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
      $(RM) $(BIN)
Exercise 1 Solution: Fortran Makefile

F90    = pgf90
CCFLAGS =
ACCFLAGS = -acc -ta=nvidia, -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN = laplace2d_f90_omp laplace2d_f90_acc

all: $(BIN)

laplace2d_f90_acc: laplace2d.f90
    $(F90) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_f90_omp: laplace2d.f90
    $(F90) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
    $(RM) $(BIN)
Exercise 1: Compiler output (C)

pgcc -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c

main:
  57, Generating copyin(A[:4095][:4095])
  Generating copyout(Anew[1:4094][1:4094])
  Generating compute capability 1.3 binary
  Generating compute capability 2.0 binary
  58, Loop is parallelizable
  60, Loop is parallelizable
  Accelerator kernel generated
  58, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
  60, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
  Cached references to size [18x18] block of 'A'
  CC 1.3 : 17 registers; 2656 shared, 40 constant, 0 local memory bytes; 75% occupancy
  CC 2.0 : 18 registers; 2600 shared, 80 constant, 0 local memory bytes; 100% occupancy
  64, Max reduction generated for error
  69, Generating copyout(A[1:4094][1:4094])
  Generating copyin(Anew[1:4094][1:4094])
  Generating compute capability 1.3 binary
  Generating compute capability 2.0 binary
  70, Loop is parallelizable
  72, Loop is parallelizable
  Accelerator kernel generated
  70, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
  72, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
  CC 1.3 : 8 registers; 48 shared, 8 constant, 0 local memory bytes; 100% occupancy
  CC 2.0 : 10 registers; 8 shared, 56 constant, 0 local memory bytes; 100% occupancy

Compiler was able to parallelize

Compiler was able to parallelize
Exercise 1: Performance

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz
GPU: NVIDIA Tesla M2070

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>162.16</td>
<td>0.24x FAIL</td>
</tr>
</tbody>
</table>

Speedup vs. 6 CPU cores

Speedup vs. 1 CPU core
What went wrong?

Add -ta=nvidia,time to compiler command line

Accelerator Kernel Timing data
/usr/users/6/harrism/openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c

main
   69: region entered 1000 times
time(us): total=77524918 init=240 region=77524678
   kernels=4422961 data=66464916
w/o init: total=77524678 max=83398 min=72025 avg=77524
72: kernel launched 1000 times
   grid: [256x256] block: [16x16]
time(us): total=4422961 max=4543 min=4345 avg=4422

66.5 seconds

Huge Data Transfer Bottleneck!
Computation: 12.7 seconds
Data movement: 133.3 seconds

66.8 seconds

8.3 seconds

/usr/users/6/harrism/openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c

main
   57: region entered 1000 times
time(us): total=82135902 init=216 region=82135686
   kernels=8346306 data=66775717
w/o init: total=82135686 max=159083 min=76575 avg=82135
60: kernel launched 1000 times
   grid: [256x256] block: [16x16]
time(us): total=8201000 max=8297 min=8187 avg=8201
64: kernel launched 1000 times
   grid: [1] block: [256]
time(us): total=145306 max=242 min=143 avg=145

acc_init.c
acc_init
   29: region entered 1 time
time(us): init=158248

4.4 seconds
Basic Concept
Simplified, but sadly true
Multiple Times Each Iteration

CPU Memory

A(i-1,j)  A(i,j)  A(i+1,j)  A(i,j-1)

GPU Memory

A(i-1,j)  A(i,j)  A(i+1,j)  A(i,j-1)
Excessive Data Transfers

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

#pragma acc kernels

```c
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]));
    }
}
```

4 copies happen every iteration of the outer while loop! (2 for each kernel)
Data Management

The First, Most Important, and possibly Only OpenACC Optimization
Data Construct Syntax and Scope

Fortran

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

C

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

General Clauses

- if( condition )
- async( expression )
- data copy clauses
## Data Clauses

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>copy(list)</code></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. Principal use: For many important data structures in your code, this is a logical default to input, modify and return the data.</td>
</tr>
<tr>
<td><code>copyin(list)</code></td>
<td>Allocates memory on GPU and copies data from host to GPU when entering region. Principal use: Think of this like an array that you would use as just an input to a subroutine.</td>
</tr>
<tr>
<td><code>copyout(list)</code></td>
<td>Allocates memory on GPU and copies data to the host when exiting region. Principal use: A result that isn’t overwriting the input data structure.</td>
</tr>
<tr>
<td><code>create(list)</code></td>
<td>Allocates memory on GPU but does not copy. Principal use: Temporary arrays.</td>
</tr>
</tbody>
</table>
Present Data Clauses

The “present” data clauses are used when the data is already present because of a containing data region.

\textbf{present( list )}

Data is already present on GPU from another containing data region. Principal use: You are calling this routine from inside a routine that already has a data clause.

\textbf{present_or_copy}
\textbf{present_or_copyin}
\textbf{present_or_copyout}
\textbf{present_or_create}

You can’t be positive that the data is present from a surrounding data region. Principal use: A subroutine that may or may not be called from within a data region.
Array Shaping

- Compilers sometimes cannot determine the size of arrays, so we must specify explicitly using data clauses with an array “shape”. The compiler will let you know if you need to do this. Sometimes, you will want to for your own efficiency reasons.

- C

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

- Fortran

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

- Note that for both Fortran and C the format is start:length.
- Data clauses can be used on data, kernels or parallel
Data Regions Have Real Consequences

**Simplest Kernel**

```c
int main(int argc, char** argv){
    float A[1000];
    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++){
        A[iter] = 1.0;
    }
    A[10] = 2.0;
}
```

**With Global Data Region**

```c
int main(int argc, char** argv){
    float A[1000];
    #pragma acc data copy(A)
    {
        #pragma acc kernels
        for( int iter = 1; iter < 1000 ; iter++){
            A[iter] = 1.0;
        }
        A[10] = 2.0;
    }
}
```

**Output:**


With Global Data Region: `A[10] = 1.0`
Data Movement Decisions

Much like loop data dependencies, sometime the compiler needs your human intelligence to make high-level decisions about data movement. Otherwise, it must remain conservative - sometimes at great cost.

You must think about when data truly needs to migrate, and see if that is better than the default.

Besides the scope based data clauses, there are a few OpenACC options to let us manage data movement more intensely or asynchronously. We could (poorly) override the above behavior with the update construct:

Fortran:

```
!$acc update [clause …]
```

C:

```
#pragma acc update [clause …]
```
Exercise 2: Use acc data to minimize transfers
(about 20 minutes)

1) Start with the C or Fortran code in exercises/002-laplace2D-data. This is just the solution of the last exercise. Add data directives where it helps. Think: when should I move data between host and GPU?

2) Run:
   OpenACC version with laplace_acc.job

Q: What speedup can you get with data + kernels directives?
#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++) {
        }
        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.

```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

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz
GPU: NVIDIA Tesla M2070

<table>
<thead>
<tr>
<th>Execution</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1 OpenMP thread</td>
<td>69.80</td>
<td>--</td>
</tr>
<tr>
<td>CPU 2 OpenMP threads</td>
<td>44.76</td>
<td>1.56x</td>
</tr>
<tr>
<td>CPU 4 OpenMP threads</td>
<td>39.59</td>
<td>1.76x</td>
</tr>
<tr>
<td>CPU 6 OpenMP threads</td>
<td>39.71</td>
<td>1.76x</td>
</tr>
<tr>
<td>OpenACC GPU</td>
<td>13.65</td>
<td>2.9x</td>
</tr>
</tbody>
</table>

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU
Further speedups

- OpenACC gives us even 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

- We tackle these in other talks, but you have already gained most of an impressive speedup, and you did it with a few lines of directives!
Is OpenACC Living Up To My Claims?

- High-level. No involvement of OpenCL, CUDA, etc.

- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial, non-GPU programmers can play along.

- Efficient. Experience show very favorable comparison to low-level implementations of same algorithms.

- Performance portable. Supports GPU accelerators and co-processors from multiple vendors, current and future versions.

- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be quick.
OpenMP has a very similar directive philosophy. This is no surprise as OpenACC was started by OpenMP members as an “accelerator development branch” with the idea of merging back in to OpenMP.

So, what are the differences?

- OpenMP had no data migration control.
- OpenMP is too flexible in its thread model and will not catch data dependencies or recognize the limitations of GPU type hardware.
OpenMP Data Migration

OpenMP comes from an SMP multi-core background. The original idea was to avoid the pain of using Unix/Posix pthreads directly. As SMPs have no concept of different memory spaces, OpenMP has not been concerned with that until now. With OpenMP 4.0 (to be approved in July), that changes. We now have data migration control and related capability like data shaping.

```
#pragma omp target device(0) map(tofrom:B)
```
```c
int main(int argc, const char* argv[]) {
    int n = 10240; float a = 2.0f; float b = 3.0f;
    float* x = (float*) malloc(n * sizeof(float));
    float* y = (float*) malloc(n * sizeof(float));

    // Run SAXPY TWICE inside data region
    #pragma omp target data map(to:x)
    {
        #pragma omp target map(tofrom:y)
        #pragma omp teams
        #pragma omp distribute
        #pragma omp parallel for
        for(int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }
        #pragma omp target map(tofrom:y)
        #pragma omp teams
        #pragma omp distribute
        #pragma omp parallel for
        for(int i = 0; i < n; ++i){
            y[i] = b*x[i] + y[i];
        }
    }
}
```

Courtesy Christian Terboven
OpenMP was traditionally oriented towards controlling fully independent processors. In return for the flexibility to use those processors to their fullest extent, OpenMP assumes that you know what you are doing and does not recognize data dependencies in the same way as OpenACC.

While you override detected data dependencies in OpenACC (with the `independent` clause), there is no such thing in OpenMP. Everything is assumed to be independent. You must be the paranoid one, not the compiler.

OpenMP assumed that every thread is expensive and has its own synchronization control (barriers, locks). GPUs have very cheap threads without uniform control at all levels. For example, NVIDIA GPUs have synchronization at the Warp level, but not the Thread Block level. There are implications that I have not had to contemplate (yet, for reasons mentioned below). Others have been concerned.
Comparing OpenACC with OpenMP 4.0 on NVIDIA and Phi

First two examples
Courtesy Christian Terboven

OpenMP 4.0 for Intel Xeon Phi

```c
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

OpenMP 4.0 for NVIDIA GPU

```c
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
    #pragma omp parallel for
    for (b = i; b < i+num_blocks; b++)
        B[b] += sin(B[b]);
```

OpenACC for NVIDIA GPU

```c
#pragma acc kernels
for (i=0; i<N; ++i)
    B[i] += sin(B[i]);
```
Which way to go?

While this might be an interesting discussion of the finer distinctions between these two standards and the future merging thereof, it is not. At the moment, there is a simpler reality:

- OpenMP 4.0 will just be ratified in July, so it will be a while before it has the widespread support of OpenMP 3. It is currently implemented only on Intel compilers for Xeon Phi - and not fully compliant there yet.

- OpenACC will eventually support Phi, but does not as of yet. It does work great on NVIDIA GPUs.
So, at this time...

- If you are using Phi, you are going to be using the Intel OpenMP release.
  - We have been considering Offload mode
  - In Native mode you can just use OpenMP 3.0!

- If you are using NVIDIA GPU’s, you are going to be using OpenACC.

Of course, there are other ways of programming both of these devices. You might treat Phi as MPI cores and use CUDA on NVIDIA, for example. But if the directive based approach is for you, then your path is clear. I attempt to discuss the many other types of accelerators here (AMD, DSPs, FPGAs, ARM), but these techniques apply there as well.

And as you should now suspect, even if it takes a while to merge these two, it is not a big jump to move between them.
Availability

XSEDE has OpenMP and OpenACC compiler licenses on the appropriate platforms.

PGI has a 15 day trial license of their OpenACC compiler that is pretty easy to obtain.

Every major compiler (Intel, Gnu, IBM, PGI,...) supports the OpenMP standard and you can anticipate that these 4.0 features will eventually make their way into these distributions.
Credits

This lecture, particularly the exercises, were derived from the work of Mark Harris of NVIDIA. In addition, suggestions by attendees of earlier workshops have had a major influence. And credits as noted on individual slides reflect gracious contributions.