Introduction to OpenACC Directives

Duncan Poole, NVIDIA
GPUs Reaching Broader Set of Developers

1,000,000’s

Early Adopters

100,000’s

Research

Supercomputing Centers

Oil & Gas

Universities

Time

2004

Present

CAE

CFD

Finance

Rendering

Data Analytics

Life Sciences

Defense

Weather

Climate

Plasma Physics

100,000’s
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 ...
enddo
enddo
!
$acc end kernels
...
End Program myscience
Familiar to OpenMP Programmers

OpenMP

```c
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

```c
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);
}
```
OpenACC
Open Programming Standard for Parallel Computing

“OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan.”

--Buddy Bland, Titan Project Director, Oak Ridge National Lab

“OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP.”

--Michael Wong, CEO OpenMP Directives Board
OpenACC
The Standard for GPU Directives

- **Easy:** 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

- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU
High-level, with low-level access

- Compiler directives to specify parallel regions in C & Fortran
  - OpenACC compilers offload parallel regions from host to accelerator
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogeneous programs
  - Without explicit accelerator initialization,
  - Without explicit data or program transfers between host and accelerator
- Programming model allows programmers to start simple
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.
Directives: Easy & Powerful

Real-Time Object Detection
Global Manufacturer of Navigation Systems

Valuation of Stock Portfolios using Monte Carlo
Global Technology Consulting Company

Interaction of Solvents and Biomolecules
University of Texas at San Antonio

5x in 40 Hours
2x in 4 Hours
5x in 8 Hours

"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%
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online
  - www.openacc.org
- Quick reference card also available
- Beta implementations available now from PGI, Cray, and CAPS
Start Now with OpenACC Directives

Sign up for a free trial of the directives compiler now!

Free trial license to PGI Accelerator
Tools for quick ramp

www.nvidia.com/gpudirectives

Accelerate Your Scientific Code with OpenACC
The Open Standard for GPU Accelerator Directives

Thousands of cores working for you.
Based on the OpenACC standard, GPU directives are the easy, proven way to accelerate your scientific or industrial code. With GPU directives, you can accelerate your code by simply inserting compiler hints into your code and the compiler will automatically map compute-intensive portions of your code to the GPU. Here's an example of how easy a single directive hint can accelerate the calculations of pi. With GPU directives, you can get started and see results in the same afternoon.

By starting with a free, 30-day trial of PGI Directives today, you are working on the technology that is the foundation of the OpenACC directives standard. OpenACC is:

Dr. Kerry Black
University of Melbourne

"The PGI compiler is just how powerful it is. It's software we are written to do one thing: make the future happen. It's like a supercomputer."
Getting Started with OpenACC

Mark Harris, NVIDIA
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] ...
```

...often paired with a matching end directive surrounding a structured code block:

```
!$acc end directive
```

**C**

```
#pragma acc directive [clause [,] clause] ...
```

...often followed by a structured code block
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

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

**Kernel:** A parallel function that runs on the GPU
Kernels Construct

Fortran

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

C

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

Clauses

- if( condition )
- async( expression )

Also, any data clause (more later)
C tip: the restrict keyword

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g.
    
    ```c
    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

- **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:

```sh
pgcc -acc [-Minfo=accel] -o saxpy_acc saxpy.c
```

Fortran:

```sh
pgf90 -acc [-Minfo=accel] -o saxpy_acc saxpy.f90
```

Compiler output:

```sh
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc 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
```
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 ( 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++;
}
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++) {
            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 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++;
}
```c
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc kernels 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 kernels
    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

```fortran
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
Jacobi Iteration: OpenMP Fortran Code

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 end parallel do

!$omp parallel do shared(m,n,Anew,A)
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$omp end parallel do

iter = iter +1
end do
Jacobi Iteration: OpenACC Fortran Code

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

!$acc kernels reduction(max:err)
  do j=1,m
    do i=1,n
      Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                A(i , j-1) + A(i , j+1))
      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$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
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 err
  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
## 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. 1 CPU core**  
**Speedup vs. 6 CPU cores**
What went wrong?

Set `PGI_ACC_TIME` environment variable to ‘1’

Accelerator Kernel Timing data

```c
./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=6646916
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
```

57: region entered 1000 times
time(us): total=82135902 init=216 region=82135686
kernels=8346306 data=6677571
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
```

Accelerator init.c

acc_init
29: region entered 1 time
time(us): init=158248
```
Basic Concepts

For efficiency, decouple data movement and compute off-load.
Excessive Data Transfers

```
while ( err > tol && iter < iter_max ) {
    err=0.0;
    #pragma acc kernels 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  Copy  A, Anew resident on accelerator

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

A, Anew resident on host  Copy  A, Anew resident on accelerator

And note that there are two #pragma acc kernels, so there are 4 copies per while loop iteration!
Jacobi Iteration: OpenACC Fortran Code

```fortran
!$acc kernels reduction(max:err)
   do j=1,m
      do i=1,n
          Anew(i,j) = .25_fp_kind * (A(i+1, j ) + A(i-1, j ) + &
                                   A(i , j-1) + A(i , j+1))
          err = max(err, Anew(i,j) - A(i,j))
      end do
   end do
!$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
```

Jacobi Iteration: OpenACC Fortran Code

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

  !$acc kernels 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

  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
```
DATA MANAGEMENT
Data Construct

Fortran

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

C

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

General Clauses

```c
if( condition )
async( expression )
```

Manage data movement. Data regions may be nested.
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.
Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”

- 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: data clauses can be used on data, kernels or parallel
Update Construct

Fortran

!$acc update [clause ...]

Clauses

  host( list )
  device( list )

C

#pragma acc update [clause ...]

if( expression )
async( expression )

Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

Move data from GPU to host, or host to GPU.
Data movement can be conditional, and asynchronous.
Jacobi Iteration: Data Directives

Task: use acc data to minimize transfers in the Jacobi example
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;

    #pragma acc kernels 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 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.
Jacobi Iteration: OpenACC Fortran Code

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

\$acc kernels 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
\$acc end kernels

...
### Performance

<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>

**CPU:** Intel Xeon X5680  
6 Cores @ 3.33GHz

**GPU:** NVIDIA Tesla M2070

**Note:** same code runs in 9.78s on NVIDIA Tesla M2090 GPU
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
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
  - PGI_ACC_TIME = 1 (environment variable)
- Eliminate pointer arithmetic
- Inline function calls in directives regions
  - (PGI): -Minline or -Minline=levels:<N>
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with _OPENACC macro
Using CUDA Libraries with OpenACC

Mark Harris, NVIDIA
3 Ways to Accelerate Applications

- Libraries
  - “Drop-in” Acceleration
- OpenACC Directives
  - Easily Accelerate Applications
- Programming Languages
  - Maximum Flexibility

CUDA Libraries are interoperable with OpenACC
3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in” Acceleration

OpenACC Directives

Easily Accelerate Applications

Programming Languages

CUDA Languages are interoperable with OpenACC, too!
CUDA Libraries
Overview
GPU Accelerated Libraries
“Drop-in” Acceleration for Your Applications
CUDA Math Libraries

High performance math routines for your applications:

- cuFFT - Fast Fourier Transforms Library
- cuBLAS - Complete BLAS Library
- cuSPARSE - Sparse Matrix Library
- cuRAND - Random Number Generation (RNG) Library
- NPP - Performance Primitives for Image & Video Processing
- Thrust - Templated C++ Parallel Algorithms & Data Structures
- math.h - C99 floating-point Library

Included in the CUDA Toolkit  Free download @ www.nvidia.com/getcuda

More information on CUDA libraries:

http://www.nvidia.com/object/gtc2010-presentation-archive.html#session2216
FFTs up to 10x Faster than MKL

1D used in audio processing and as a foundation for 2D and 3D FFTs

- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

Performance may vary based on OS version and motherboard configuration
CUDA 4.1 optimizes 3D transforms

Single Precision All Sizes 2x2x2 to 128x128x128

Consistently faster than MKL

>3x faster than 4.0 on average

Performance may vary based on OS version and motherboard configuration

- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuBLAS: Dense Linear Algebra on GPUs

- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex

- New in CUDA 4.1
  - New batched GEMM API provides >4x speedup over MKL
    - Useful for batches of 100+ small matrices from 4x4 to 128x128
    - 5%-10% performance improvement to large GEMMs
cuBLAS Level 3 Performance

Up to 1 TFLOPS sustained performance and >6x speedup over Intel MKL

Performance may vary based on OS version and motherboard configuration
cuBLAS Batched GEMM API improves performance on batches of small matrices

![Graph showing performance comparison between cuBLAS 100 matrices, cuBLAS 10,000 matrices, and MKL 10,000 matrices.](image_url)

- cuBLAS 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

Performance may vary based on OS version and motherboard configuration.
**cuSPARSE: Sparse linear algebra routines**

- Sparse matrix-vector multiplication & triangular solve
  - APIs optimized for iterative methods
- **New in 4.1**
  - Tri-diagonal solver with speedups up to 10x over Intel MKL
  - ELL-HYB format offers 2x faster matrix-vector multiplication

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
\end{bmatrix} = \alpha \begin{bmatrix}
  1.0 & \cdots & \cdots & \cdots \\
  2.0 & 3.0 & \cdots & \cdots \\
  \cdots & \cdots & 4.0 & \cdots \\
  5.0 & \cdots & 6.0 & 7.0 \\
\end{bmatrix} \begin{bmatrix}
  1.0 \\
  2.0 \\
  3.0 \\
  4.0 \\
\end{bmatrix} + \beta \begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
\end{bmatrix}
\]
cuSPARSE is >6x Faster than Intel MKL

Performance may vary based on OS version and motherboard configuration

• cuSPARSE 4.1, Tesla M2090 (Fermi), ECC on
  • MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

*Average speedup over single, double, single complex & double-complex
cuRAND: Random Number Generation

- Pseudo- and Quasi-RNGs
- Supports several output distributions
- Statistical test results reported in documentation

New commonly used RNGs in CUDA 4.1
- MRG32k3a RNG
- MTGP11213 Mersenne Twister RNG

Monte Carlo Integration
1000+ New Imaging Functions in NPP 4.1

Up to 40x speedups

- NVIDIA Performance Primitives (NPP) library includes over 2200 GPU-accelerated functions for image & signal processing
  - Arithmetic, Logic, Conversions, Filters, Statistics, etc.

- Most are 5x-10x faster than analogous routines in Intel IPP

http://developer.nvidia.com/content/graphcuts-using-npp

* NPP 4.1, NVIDIA C2050 (Fermi)
* IPP 6.1, Dual Socket Core™ i7 920 @ 2.67GHz
Using CUDA Libraries with OpenACC
Sharing data with libraries

- CUDA libraries and OpenACC both operate on device arrays

- OpenACC provides mechanisms for interop with library calls
  - `deviceptr` data clause
  - `host_data` construct

- Note: same mechanisms useful for interop with custom CUDA C/C++/Fortran code
deviceptr Data Clause

deviceptr( list ) declares that the pointers in list refer to device pointers that need not be allocated or moved between the host and device for this pointer.

Example:

C
#pragma acc data deviceptr(d_input)

Fortran
$!acc data deviceptr(d_input)
host_data Construct

Makes the address of device data available on the host.

deviceptr( list ) Tells the compiler to use the device address for any variable in list. Variables in the list must be present in device memory due to data regions that contain this construct

Example

C
    #pragma acc host_data use_device(d_input)

Fortran
    !$acc host_data use_device(d_input)
Example: 1D convolution using CUFFT

Perform convolution in frequency space
1. Use CUFFT to transform input signal and filter kernel into the frequency domain
2. Perform point-wise complex multiply and scale on transformed signal
3. Use CUFFT to transform result back into the time domain

We will perform step 2 using OpenACC

Code walk-through follows. Code available with exercises.

In exercises/cufft-acc
// Transform signal and kernel
error = cufftExecC2C(plan, (cufftComplex *)d_signal,
                    (cufftComplex *)d_signal, CUFFT_FORWARD);
error = cufftExecC2C(plan, (cufftComplex *)d_filter_kernel,
                    (cufftComplex *)d_filter_kernel, CUFFT_FORWARD);

// Multiply the coefficients together and normalize the result
printf("Performing point-wise complex multiply and scale.\n");
complexPointwiseMulAndScale(new_size,
                    (float *restrict)d_signal,
                    (float *restrict)d_filter_kernel);

// Transform signal back
error = cufftExecC2C(plan, (cufftComplex *)d_signal,
                    (cufftComplex *)d_signal, CUFFT_INVERSE);
void complexPointwiseMulAndScale(int n, float *restrict signal,  
    float *restrict filter_kernel)  
{
    // Multiply the coefficients together and normalize the result
    #pragma acc data deviceptr(signal, filter_kernel)  
    {
        #pragma acc kernels loop independent  
        for (int i = 0; i < n; i++) {
            float ax = signal[2*i];
            float ay = signal[2*i+1];
            float bx = filter_kernel[2*i];
            float by = filter_kernel[2*i+1];
            float s = 1.0f / n;
            float cx = s * (ax * bx - ay * by);
            float cy = s * (ax * by + ay * bx);
            signal[2*i] = cx;
            signal[2*i+1] = cy;
        }
    }
}  

Note: The PGI C compiler does not currently support structs in OpenACC loops, so we cast the Complex* pointers to float* pointers and use interleaved indexing.
#include “cufft.h”

Compiler command line options:

CUDA_PATH = /usr/local/pgi/linux86-64/2012/cuda/4.0
CCFLAGS = -I$(CUDA_PATH)/include -L$(CUDA_PATH)/lib64
-lcudart -lcufft

Must link libcudart and libcufft

Must use PGI-provided CUDA toolkit paths
Results

[harrism@computer cufft-acc]$ ./cufft_acc
Transforming signal cufftExecC2C
Performing point-wise complex multiply and scale.
Transforming signal back cufftExecC2C
Performing Convolution on the host and checking correctness

Signal size: 500000, filter size: 33
Total Device Convolution Time: 11.461152 ms (0.242624 for point-wise convolution)
Test PASSED
Summary

- Use `deviceptr` data clause to pass pre-allocated device data to OpenACC regions and loops.
- Use `host_data` to get device address for pointers inside `acc` data regions.

The same techniques shown here can be used to share device data between OpenACC loops and:

- Your custom CUDA C/C++/Fortran/etc. device code.
- Any CUDA Library that uses CUDA device pointers.
Thank you

mharris@nvidia.com
Thank you