Programming GPUs in Fortran

Accelerator and Cuda Fortran
Open Accelerator Standard

What is OpenACC API?
- OpenACC API allows parallel programmers to provide simple hints, known as “directives,” to the compiler, identifying which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself. By exposing parallelism to the compiler, directives allow the compiler to do the detailed work of mapping the computation onto the accelerator.

Quote from Michael Wong, CEO of the OpenMP Architecture Review Board:
"I am enthusiastic about the future of accelerator technologies. The OpenACC announcement highlights the technically impressive initiative undertaken by members of the OpenMP Working Group on Accelerators. I look forward to working with all four companies within the OpenMP organization to merge OpenACC with other ideas to create a common specification which extends OpenMP to support accelerators. We look forward to incorporating accelerator support with the full support of all OpenMP members in a future version of the OpenMP specification."
GPU considerations: Architecture

**GPU architecture**

- Large number of cores working in SIMD mode
- Slow global memory access, high bandwidth
- CPU communication over PCI bus
- Warp scheduling and fast switching queue model
GPU considerations: Programming

- Allocate data on the GPU
- Move data from host, or initialize data on GPU
- Launch kernel(s)
- Gather results from GPU
- Deallocate data
Programming example

```c
int main( void ) {
    int a[N], b[N], c[N];
    int *a_d, *b_d, *c_d;

    // allocate the memory on the GPU
    cudaMemcpy( a_d, a, N* sizeof(int) );
    cudaMemcpy( b_d, b, N* sizeof(int) );
    cudaMemcpy( c_d, c, N* sizeof(int) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( a_d, a, N* sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy( b_d, b, N* sizeof(int), cudaMemcpyHostToDevice);

    // launch kernel
    add<<<N,1>>>( a_d, b_d, c_d );

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, c_d, N* sizeof(int), cudaMemcpyDeviceToHost);

    // free the memory allocated on the GPU
    cudaFree( a_d );
    cudaFree( b_d );
    cudaFree( c_d );

    return 0;
}
```
int main( void ) {
    int a[N], b[N], c[N];
    int *a_d, *b_d, *c_d;

    // allocate the memory on the GPU
    cudaMalloc( (void**)&a_d, N * sizeof(int) );
    cudaMalloc( (void**)&b_d, N * sizeof(int) );
    cudaMalloc( (void**)&c_d, N * sizeof(int) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( a_d, a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy( b_d, b, N*sizeof(int), cudaMemcpyHostToDevice);

    // Launch Kernel
    add<<<N,1>>>( a_d, b_d, c_d );

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, c_d, N*sizeof(int), cudaMemcpyDeviceToHost);

    // free the memory allocated on the GPU
    cudaFree( a_d );
    cudaFree( b_d );
    cudaFree( c_d );
    return 0;}

__global__ void add( int *a, int *b, int *c) {
    int tid = blockIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
Know your Accelerator

CUDA device query

- deviceQuery

CUDA Device Query (Runtime API) version (CUDART static linking)

Found 2 CUDA Capable device(s)

Device 0: "Tesla M2050"

(14) Multiprocessors x (32) CUDA Cores/MP: 448 CUDA Cores

Warp size: 32

Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535

CUDA Driver Version: 4000

PGI pgaccelinfo

- pgaccelinfo

CUDA Driver Version: 4000

Device Number: 0
Device Name: Tesla M2050
Device Revision Number: 2.0
Global Memory Size: 2817982464
Number of Multiprocessors: 14
Number of Cores: 448

Warp Size: 32
The goal is to identify parallel regions which could be exploited using OpenMP like directives.

OpenMP like directives

• acc region

Compiler handles parallelization

• pgfortran source.f90 – ta=nvidia

http://www.pgroup.com/resources/articles.htm
PGI Accelerator directives: Examples

```c
#pragma acc region
{
    for( i = 0; i < n; ++i )
        c[i] = a[i] + b[i];
}

!$acc region
do i = 1, n
    c(i) = a(i) + b(i)
enddo

!$acc end region
```
PGI Accelerator vs OpenMP

```c
#pragma acc region
{
    for( i = 0; i < n; ++i )
    c[i] = a[i]+b[i];
}

#pragma omp parallel
{
    #pragma omp for
    for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
}

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

!$omp parallel
   !$omp do
       do i = 1, n
           c(i) = a(i) + b(i)
       enddo
   !$omp end do
!$omp end parallel
```
PGI Accelerator Model

PGI Accelerator flags

- `-ta=nvidia,host`
  Run on GPU if available, else run on host

- `-Minfo=accel`
  Print out kernel information

- `--keepptx`
  Generate ptx code

```bash
pgfortran source.f90 \
  -ta=nvidia,host  -Minfo
```
program vector_add
    implicit none
    integer, parameter :: n=10
    real, allocatable :: a(:,), b(:,), c(:)

    allocate(a(1:n), b(1:n), c(1:n))
    a=1.0; b=2.0; c=0.0;

    !$acc region
    c = a + b
    !$acc end region

disable(a, b, c)
end program
program vector_add
  implicit none
  integer, parameter :: n=10
  real, allocatable :: a(:), b(:), c(:)

  ! +-----------------------+
  ! | Initialize arrays    |
  ! +-----------------------+
  allocate(a(1:n), b(1:n), c(1:n))
  a=1.0; b=2.0; c=0.0;
  ! +-----------------------+
  ! | Add arrays           |
  ! +-----------------------+
  !$acc region
  c = a + b
  !$acc end region

dallocate(a, b, c)
end program

$ pgfortran vecadd.f90 -ta=nvidia -Minfo
  vector_add:
  14, Generating copyin(b(1:10))
     Generating copyin(a(1:10))
     Generating compute capability 1.0 binary
     Generating compute capability 1.3 binary
     Generating compute capability 2.0 binary
  15, Loop is parallelizable
     Accelerator kernel generated
     14, !$acc do parallel, vector(10) ! blockidx%x threadidx%x
        CC 1.0 : 6 registers; 44 shared, 4 constant, 0 local
        memory bytes; 33% occupancy
        CC 1.3 : 6 registers; 44 shared, 4 constant, 0 local
        memory bytes; 25% occupancy
        CC 2.0 : 14 registers; 4 shared, 56 constant, 0 local
        memory bytes; 16% occupancy

  Data info
  Kernel info
# PGI Accelerator: Example

```fortran
program vector_add
  implicit none
  integer, parameter :: n=10
  real, allocatable :: a(:), b(:), c(:)

  ! +-----------------------------+
  ! | Initialize arrays          |
  ! +-----------------------------+
  allocate(a(1:n), b(1:n), c(1:n))
  a=1.0; b=2.0; c=0.0;
  ! +-----------------------------+
  ! | Add arrays                 |
  ! +-----------------------------+
  !$acc region
  c = a + b
  !$acc end region

disable(a, b, c)
end program
```

```
$ time ./a.out
Accelerator Kernel Timing data
/home/bthakur/vecadd.f90
vector_add
  14: region entered 1 time
time(us): total=1110418 init=1107179 region=3239
kernels=30 data=528
w/o init: total=3239 max=3239 min=3239 avg=3239
  15: kernel launched 1 times
grid: [1] block: [225]
time(us): total=30 max=30 min=30 avg=30

real 0m1.199s
user 0m0.003s
sys 0m1.192s
```

---

### Time in microseconds

<table>
<thead>
<tr>
<th>Component</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>0m1.199s</td>
</tr>
<tr>
<td>User</td>
<td>0m0.003s</td>
</tr>
<tr>
<td>System</td>
<td>0m1.192s</td>
</tr>
</tbody>
</table>
Time for a simple demo

What does compiler do?

- ?

How to increase efficiency

- ?

Timing and profiling

- ?
Time for a simple demo

What did compiler do?

- Analyze info on generated kernel

How to increase efficiency

- Initialize device (pgcudainit)
- Increase occupancy! More threadblocks and more threads

Timing and profiling

- Simple profiling: -ta=nvidia,time
PGI Accelerator: Reduction

**Primitive reductions**

```
sum=0.0
!$acc region
  do i=1,n
    sum = sum + a(i)
  end do
!$acc end region
```

```
$ pgfortran -ta=nvidia -Minfo vecsum.f90
vector_add:
17, Generating copyin(a(1:10000))
  ...
18, Loop is parallelizable
    Accelerator kernel generated
18, !$acc do parallel, vector(256) ! blockIdx%x threadIdx%x
  ...
19, Sum reduction generated for sum
```
### PGI Accelerator: Intrinsics

#### Fortran Intrinsics

```fortran
program intrinsic
  implicit none
  integer, parameter :: n=32
  real :: a(n,n), b(n,n), c(n,n)
  call random_number(a)
call random_number(b)
!
  !$acc region
c  = matmul(transpose(b), matmul(a,b))
!
  !$acc end region
end
```

```bash
$ pgfortran -ta=nvidia -Minfo intrinsic.f90
intrinsic:
11, Generating copyout(tmp$r(1:32,1:32))
Generating copyin(b(1:32,1:32))
...
13, Loop is parallelizable
  Accelerator kernel generated
13, !$acc do parallel, vector(16) ! blockIdx%x threadIdx%x
  !$acc do parallel, vector(16) ! blockIdx%y threadIdx%y
CC 1.0 : 10 registers; 40 shared, 8 constant, 0 local memory bytes; 100% occupancy
CC 1.3 : 10 registers; 40 shared, 8 constant, 0 local memory bytes; 100% occupancy
CC 2.0 : 12 registers; 8 shared, 48 constant, 0 local memory bytes; 100% occupancy
```
Accelerator restrictions

Loop dependency
• \( a(n) = a(n-1) + b(n) \times c(n) \)

Non-strided access
• \( vout( i_ptr(n) ) = vout( i_ptr(n) ) + vin( j_ptr(n) ) \times val_ptr(n) \)

Conditional assignments
• If \( a(i) < 0 \) \( x = b(i) \times w \)

Functions
• fatorial(n)

Memory restrictions
• Fixed size shared memory arrays
• No textures
Accelerator restrictions

No Triangular loops
- Do i=1, 100
- do

Conditional assignments
- If ( a(i) < 0 ) x = b(i)*w

Memory restrictions
- Fixed size shared memory arrays
- No textures
Demo: 2

- Memory layout
- Register, shared and constant memory

Avoiding restrictions

- Arrange data in SIMD compatible format
PGI Accelerator: Data clauses

- **Copyin/updatein** (into the GPU) and **copyout /updateout** (out of the GPU)
- Compiler moves the smallest part of each array needed to execute the loop
- Use **local** clause to leave unwanted data
- **Data region** allows boundaries for data movement
- **Mirrored allocatable data** allows a copy to array to be allocated on the device
PGI Accelerator: Schedule, feedback

Accelerator data clauses

• **Copyin/updatein** (into the GPU) and **copyout /updateout** (out of the GPU)

• Compiler moves the smallest part of each array needed to execute the loop

• Use **local** clause to leave unwanted data

• **Data region** allows boundaries for data movement

• **Mirrored allocatable data** allows a copy to array to be allocated on the device
PGI Accelerator: Schedule clauses

Accelerator data clauses

• **Copyin/updatein** (into the GPU) and **copyout /updateout** (out of the GPU)

• Compiler moves the smallest part of each array needed to execute the loop

• Use **local** clause to leave unwanted data

• **Data region** allows boundaries for data movement

• **Mirrored allocatable data** allows a copy to array to be allocated on the device
PGI Accelerator: Data clauses

Accelerator data clauses: *Mirrored allocatable data*

```fortran
module glob
  real, allocatable :: x(:)
  !$acc mirror( x )
end glob

subroutine sub( y )
  use glob
  real, dimension(:) :: y
  !$acc region
  do i = 1, ubound(y,1)
    y(i) = y(i) + x(i)
  enddo
  !$acc end region
end subroutine

No analogue in C
```
PGI CUDA Fortran

CUDA C like extension to Fortran

- If accelerator doesn’t work, CUDA Fortran might.
- Gives you greater flexibility.
- Attributes easier than CUDA C to remember.
- Of course, usual CUDA restrictions apply.
Recollect CUDA Model

**GPU programming**

- Allocate data on the GPU
- Move data from host, or initialize data on GPU
- Launch kernel(s)
- Gather results from GPU
- Deallocate data

Diagram:
- Data In
- Kernel Launch
- SIMD mode: Threads and blocks
- Gather, Deallocate
- Data Out
Recollect CUDA model

PGI CUDA Fortran Basics

- Seamless integration into modules
- Attributes define subroutine behavior
- Simple allocation and copy in/out
- Kernel launch configuration similar to CUDA C

- Use module cudafor
  use cudafor

- attributes(global/host) subroutine

- Host and Device arrays
  real, device, allocatable :: a_dev(:)
  allocate( a_dev(n))

- Easy copying
  a_dev = a_host

- Kernel configuration
  type(dim3) :: dimGrid, dimBlock
  dimGrid = dim3( N/16, L/16, 1 )
  dimBlock = dim3( 16, 16, 1 )
Attributes(host)

The host attribute, specified on the subroutine or function statement, declares that the subroutine or function is to be executed on the host. Such a subprogram can only be called from another host subprogram.

Attributes(global)

The global attribute may only be specified on a subroutine statement; it declares that the subroutine is a kernel subroutine, to be executed on the device, and may only be called from the host using a kernel call containing the chevron syntax and runtime mapping parameters.

Attributes(device)

The device attribute, specified on the subroutine or function statement, declares that the subprogram is to be executed on the device; such a routine must be called from a subprogram with the global or device attribute.
CUDA Fortran variable qualifiers

• **Attributes(device)**
  – Allocated in the device global memory. If declared in a module, the variable may be accessed by any subprogram in that module and by any subprogram that uses the module.
  – A device array may be an allocatable array, or an assumed-shape dummy array. An allocatable device variable has a dynamic lifetime, from when it is allocated until it is deallocated. Other device variables have a lifetime of the entire application.

• **Attributes(constant)**
  – Device constant variables are allocated in the device constant memory space. Device constant data may not be assigned or modified in any device subprogram, but may be modified in host subprograms. Device constant variables may not be allocatable, and have a lifetime of the entire application.

• **Attributes(shared)**
  – A shared variable may only be declared in a device subprogram.
  – A shared variable is allocated in the device shared memory for a thread block, and has a lifetime of the thread block. It can be read or written by all threads in the block, though a write in one thread is only guaranteed to be visible to other threads after the next call to the SYNCTHREADS().

• **Attributes(pinned)**
  – A pinned variable must be an allocatable array. It is allocated in host pagelocked memory. The advantage of using pinned variables is that copies from page-locked memory to device memory are faster.
program spmv

use mod_spmv

! Allocate and copy in
n=1024
allocate( a(n,n), u(n), v(n),&
a_d(n,n), u_d(n), v_d(n) )
a=1.0; u=0.0; v=1.0
v_d(1:n) = v(1:n)
u_d(1:n) = u(1:n)
a_d(1:n,1:n) = a(1:n,1:n)

! Kernel launch
thds_per_blk = 32
Grid = dim3( (n+thds_per_blk-1)/thds_per_blk, 1, 1 );
Block = dim3( thds_per_blk, 1, 1 )
call kernel<<<Grid, Block>>>( a_d, u_d, v_d, n )

! Copy out
u(1:n) = u_d(1:n)
deallocate( a, u, v)
deallocate( a_d, u_d, v_d )
end
module mod_spmv
use cudafor

integer :: n, thds_per_blk
real, allocatable :: a(:, :) , u(:,), v(:)
real, allocatable, device :: a_d(:, :) , u_d(:,), v_d(:)
type(dim3) :: Grid, Block

contains

attributes(global) subroutine kernel(a, u, v, n)
  real, device :: a(n,n), u(n), v(n)
  integer, value :: n
  integer i, j
  i = (blockidx%x-1)*blockdim%x + threadIdx%x
  if (i.le.n) then
    do j=1,n
      u(i) = u(i) + a(i,j)*v(j)
    end do
  end if
end subroutine kernel

end module mod_spmv
Demo: 3

Using CUDA Fortran

- Using shared memory to speedup Mat-vec
- Possibilities for sparse matrix formats
PGI Accelerator vs CUDA Fortran

Comparison of time between four Monte Carlo Integration implementations from PGI website. Source codes available for experimentation.

<table>
<thead>
<tr>
<th></th>
<th>Host Fortran with auto-parallelization</th>
<th>PGI Accelerator Model</th>
<th>CUDA Fortran with host RNG</th>
<th>CUDA Fortran with CUDA C RNG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>13.80947</td>
<td>13.52597</td>
<td>13.37639</td>
<td>Total Time 0.95668</td>
</tr>
<tr>
<td>RNG</td>
<td>12.27999</td>
<td>12.09849</td>
<td>11.91478</td>
<td>0.52054</td>
</tr>
<tr>
<td>Compute</td>
<td>1.43082</td>
<td>0.24569</td>
<td>0.24730</td>
<td>0.24725</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>0.00000</td>
<td>1.10420</td>
<td>1.02121</td>
<td>0.00038</td>
</tr>
</tbody>
</table>

Source: Tuning a Monte Carlo Algorithm on GPUs
http://www.pgroup.com/lit/articles/insider/v2n1a4.htm
Conclusion

• Not-so-steep learning curve for Fortran-only users.
• Easy to explore possibly speedup using accelerator
• Offers easy migration to GPU based supercomputers for large legacy Fortran codes.