GPU Programming with PGI CUDA Fortran and the PGI Accelerator Programming Model

Douglas Miles
douglas.miles@pgroup.com
http://www.pgroup.com

January 2010

Part 2: Introduction to CUDA Fortran
Agenda

- Introduction
  - GPU parallel programming vs. Host parallel programming
- CUDA Fortran programming
  - CUDA Programming model
  - Programming with CUDA Fortran
  - Building CUDA Fortran Programs
GPU Architecture Features

- Optimized for high degree of regular parallelism
- Optimized for low precision
- High bandwidth memory
- Highly multithreaded (slack parallelism)
- Hardware thread scheduling
- Non-coherent software-managed data caches
- No multiprocessor memory model guarantees
NVIDIA GPU Features Summary

- Massively parallel thread processors
  - Organized into multiprocessors (up to 30 on current Tesla, see deviceQuery or pgaccelinfo)
  - Physically: 8 thread processors per multiprocessor
  - Logically: 32 threads per warp
- Memory hierarchy
  - host memory, device memory, constant memory, shared memory, register
- Queue of operations (kernels) on device
Identifying your GPU

- pgaccelinfo (for NVIDIA / AMD)
- deviceQuery (CUDA)

Peculiarities of note

- On Linux, NVIDIA driver powers down inactive devices
- On Windows, you must be at the console to access the GPU
- On MacOSX you may have two GPUs, but will power one down
Parallel Programming on CPUs

- Instruction level parallelism (ILP)
  - Loop unrolling, instruction scheduling
- Thread level / Multiprocessor / multicore parallelism
  - Parallel loops, parallel tasks
  - Posix threads, OpenMP, Cilk, TBB, ..... 
- Large scale cluster / multicomputer parallelism
  - MPI (& HPF, co-array Fortran, UPC, Titanium, X10, Fortress, Chapel)
pthreads main routine

call pthread_create( t1, NULL, jacobi, 1 )
call pthread_create( t2, NULL, jacobi, 2 )
call pthread_create( t3, NULL, jacobi, 3 )
call jacobi( 4 )
call pthread_join( t1, NULL )
call pthread_join( t2, NULL )
call pthread_join( t3, NULL )
subroutine jacobi( threadnum )
  lchange = 0
  do j = threadnum+1, n-1, numthreads
    do i = 2, m-1
      newa(i,j) = w0*a(i,j) + &
        w1 * (a(i-1,j) + a(i,j-1) + &
        a(i+1,j) + a(i,j+1)) + &
        w2 * (a(i-1,j-1) + a(i-1,j+1) + &
        a(i+1,j-1) + a(i+1,j+1))
      lchange = max(lchange,abs(newa(i,j)-a(i,j)))
    enddo
  enddo
  call pthread_mutex_lock( lck )
  change = max(change,lchange)
  call pthread_mutex_unlock( lck )
end subroutine
Jacobi Relaxation with OpenMP directives

\[
\text{change} = 0 \\
!$\text{omp parallel private}(i,j) \\
!$\text{omp do reduction(max:change)} \\
do \ j = 2, \ n-1 \\
do \ i = 2, \ m-1 \\
\text{newa}(i,j) = w0*\text{a}(i,j) + & \\
\quad w1 * (\text{a}(i-1,j) + \text{a}(i,j-1) + & \\
\quad \quad \quad \text{a}(i+1,j) + \text{a}(i,j+1)) + & \\
\quad w2 * (\text{a}(i-1,j-1) + \text{a}(i-1,j+1) + & \\
\quad \quad \text{a}(i+1,j-1) + \text{a}(i+1,j+1)) \\
\text{change} = \max(\text{change}, \text{abs(newa}(i,j)-\text{a}(i,j))) \\
\text{enddo} \\
\text{enddo} \\
!$\text{omp end parallel}
\]
Behind the Scenes

- OpenMP compiler generates code for N threads:
  - split up the iterations across N threads
  - accumulate N partial sums (no synchronization)
  - accumulate final sum as threads complete

- Assumptions
  - uniform memory access costs
  - coherent cache mechanism
GPU Programming

- Allocate data on the GPU
- Move data from host, or initialize data on GPU
- Launch kernel(s)
  - GPU driver generates ISA code at runtime
  - preserves forward compatibility without requiring ISA compatibility
- Gather results from GPU
- Deallocate data
Appropriate GPU programs

- Characterized by nested parallel loops
- High compute intensity
- Regular data access
- Isolated host/GPU data movement
Jacobi Relaxation

change = 0;
for( i = 1; i < m-1; ++i ){
    for( j = 1; j < n-1; ++j ){
        newa[j][i] = w0*a[j][i] +
                     w1 * (a[j][i-1] + a[j-1][i] +
                           a[j][i+1] + a[j+1][i]) +
                     w2 * (a[j-1][i-1] + a[j+1][i-1] +
                           a[j-1][i+1] + a[j+1][i+1]);
        change = fmaxf(change,fabsf(newa[j][i]-a[j][i]));
    }
}

__device__ float change;

memsize = sizeof(float)*n*m
cudaMalloc( &da, memsize );
cudaMalloc( &dnewa, memsize );
cudaMalloc( &lchange, (n/16)*(m/16) );

cudaMemcpy( da, a, memsize, cudaMemcpyHostToDevice );

dim3 threads( 16, 16 );
dim3 blocks( n/16, m/16 );
jacobikernel<<<blocks,threads>>>( da, dnewa, lchange, n, m );
reduction<<<1,256>>>( lchange, (n/16)*(m/16) );

cudaMemcpy( a, dnewa, memsize, cudaMemcpyDeviceToHost );

cudaFree( da );
cudaFree( dnewa );
cudaFree( lchange );
extern "C" __global__ void jacobikernel( float* a, float* anew, float* lchange, int n, int m )
{
    int ti = threadIdx.x, tj = threadIdx.y; /* local indices */
    int i = blockIdx.x*16+ti+1;             /* global indices */
    int j = blockIdx.y*16+tj+1;
    __shared__ float mychange[16*16];

    mya = w0 * a[j*m+i] +
        w1 * (a[j*m+i-1] + a[(j-1)*m+i] +
              a[j*m+i+1] + a[(j+1)*m+i] +
              w2 * (a[(j-1)*m+i-1] + a[(j+1)*m+i-1] +
                     a[(j-1)*m+i+1] + a[(j+1)*m+i+1]));

    anew[j*m+i] = mya;
    /* this thread's "change" */
    mychange[ti+16*tj] = fabs(mya,a[j*m+i]);
    __syncthreads();

    /* reduce all "change" values for this thread block
     * to a single value */
extern "C" __global__ void
jacobikernel( float* a, float* anew, float* lchange, int n, int m )
{
  int ti = threadIdx.x, tj = threadIdx.y; /* local indices */
  int i = blockIdx.x*16+ti;               /* global indices */
  int j = blockIdx.y*16+tj;
  __shared__ float mychange[16*16], b[18][18];

  b[tj][ti] = a[(j-1)*m+i-1];
  if(ti<2) b[tj][ti+16] = a[(j-1)*m+i+15];
  if(tj<2) b[tj+16][ti] = a[(j+15)*m+i-1];
  if(ti<2&&tj<2) b[tj+16][ti+16] = a[(j+15)*m+i+15];
  __syncthreads();

  mya = w0 * b[tj+1][ti+1] +
       w1 * (b[tj+1][ti] + b[tj][ti+1]) +
       b[tj+1][ti+2] + b[tj+2][ti+1]) +
       w2 * (b[tj][ti] + b[tj+2][ti]) +
       b[tj][ti+2] + b[tj+2][ti+2]);
  newa[j][i] = mya;
 /* this thread's "change" */
Behind the Scenes

- What you write is what you get
- Implicitly parallel
  - threads into warps
  - warps into thread groups
  - thread groups into a grid
- Hardware thread scheduler
- Highly multithreaded
CUDA Fortran

- Simple introductory program
- Programming model
- Low-level Programming with CUDA Fortran
- Building CUDA Fortran programs
- Performance Tuning
subroutine host_vadd(A,B,C,N)
  real(4) :: A(N), B(N), C(N)
  integer :: N
  integer :: i
  do i = 1,N
    C(i) = A(i) + B(i)
  enddo
end subroutine
CUDA Fortran Programming

- **Host code**
  - Optional: select a GPU
  - Allocate device memory
  - Copy data to device memory
  - Launch kernel(s)
  - Copy data from device memory
  - Deallocate device memory

- **Device code**
  - Scalar thread code, limited operations
  - Implicitly parallel
subroutine vadd( A, B, C )
  use kmod
  real(4), dimension(:) :: A, B, C
  real(4), device, allocatable, dimension(:):: &
      Ad, Bd, Cd
  integer :: N
  N = size( A, 1 )
  allocate( Ad(N), Bd(N), Cd(N) )
  Ad = A(1:N)
  Bd = B(1:N)
  call kernel<<< (N+31)/32, 32 >>>( Ad, Bd, Cd, N )
  C(1:N) = Cd
  deallocate( Ad, Bd, Cd )
end subroutine
Launching Kernels

- Subroutine call with chevron syntax for launch configuration
  - call kernel <<< grid, block >>> ( ... )
  - Launch configuration: <<< grid, block >>>
  - grid, block may be scalar integer expressions, or type(dim3) variables

- The launch is asynchronous
  - host program continues, may issue other launches
Launching Kernels

- Using scalar expressions to define grid & thread blocks:

  \[
  \text{call kernel <<< (N+31)/32, 32 >>> ( A, B, C, N )}
  \]

- Using variables defined using built in \texttt{dim3} derived type:

  \[
  \begin{align*}
  \text{type(dim3) :: g, b} \\
  g &= \text{dim3}((N+31)/32, 1, 1) \\
  b &= \text{dim3}( 32, 1, 1 ) \\
  \text{call kernel <<< g, b >>> ( A, B, C, N )}
  \end{align*}
  \]

- Interface must be explicit
  - In the same module as the host subprogram
  - In a module that the host subprogram uses
  - Declared in an interface block
CUDA Programming: the GPU

- A scalar program, runs on one thread
  - All threads run the same code
  - Executed in thread groups
  - grid may be 1D or 2D (max 65535x65535)
  - thread block may be 1D, 2D, or 3D (max size 512)
  - blockIdx pre-defined as block index in grid ( %x , %y )
  - gridDim pre-defined as extent of grid dimension ( %x , %y )
  - threadIdx pre-defined as thread index within block ( %x , %y , %z )
  - blockDim pre-defined as extent of block dimension ( %x , %y , %z )

- Kernel runs implicitly in parallel
  - thread blocks scheduled by hardware on any multiprocessor
  - runs to completion before next kernel
module kmod
  use cudafor
contains
  attributes(global) subroutine kernel(A,B,C,N)
    real(4), device :: A(N), B(N), C(N)
    integer, value :: N
    integer :: i
    i = (blockidx%x-1)*32 + threadIdx%x
    if( i <= N ) C(i) = A(i) + B(I)
  end subroutine
end module
Time for a Live Demo (demo1, demo2)
CUDA Fortran Vector Add
Common CUDA Errors

- Out of memory
- Launch failure
- No device found
- Invalid device code

Test for error:

```c
ir = cudaGetLastError()
if( ir ) print *, cudaGetErrorString( ir )
```
CUDA Fortran Language

- **Host code**
  - Declaring and allocating device memory
  - Moving data to and from device memory
  - Pinned memory
  - Launching kernels

- **Kernel code**
  - Attributes clause
  - Kernel subroutines, device subprograms
  - Shared memory
  - What is and what is not allowed in a kernel
  - CUDA Runtime API
Declaring Device Data

- Variables / arrays with device attribute are allocated in device memory
  - `real, device, allocatable :: a(:)`
  - `real, allocatable :: a(:)`
    - `attributes(device) :: a`

- In a host subroutine or function
  - device allocatables and automatics may be declared
  - device variables and arrays may be passed to other host subroutines or functions (explicit interface)
  - device variables and arrays may be passed to kernel subroutines
Declaring Device Data

- Variables / arrays with device attribute are allocated in device memory, with constant attribute in GPU “constant” memory
  - module mm
    - real, device, allocatable :: a(:)
    - real, device :: x, y(10)
    - real, constant :: c1, c2(10)
    - integer, device :: n
  - contains
    - attributes(global) subroutine s(b)
    - ...

- Module data must be fixed size, or allocatable
Declaring Device Data

- Data declared in a Fortran module
  - Device variables, arrays, allocatables allowed
  - Device variables, arrays, allocatables are accessible to device subprograms within that module
  - Also accessible to host subprograms in that module or which use that module
  - Constant attribute (not to be confused with parameter) puts variable or array in constant memory
Allocating Device Data

- Fortran allocate / deallocate statement
  ```fortran
  real, device, allocatable :: a(:,,:), b
  allocate( a(1:n,1:m), b )
  ....
  deallocate( a, b )
  ```
- Arrays or variables with device attribute are allocated in device memory
  - Allocate is done by the host subprogram
  - Memory is not virtual, you can run out
  - Device memory is shared among users / processes, you can have deadlock
  - `STAT=ivar` clause to catch and test for errors
Copying Data to / from Device

Assignment statements

- `real, device, allocatable :: a(:,,:), b`
  - `allocate( a(1:n,1:m), b )`
  - `a(1:n,1:m) = x(1:n,1:m) ! copies to device`
  - `b = 99.0`
  - `....`
  - `x(1:n,1:m) = a(1:n,1:m) ! copies from device`
  - `y = b`
  - `deallocate( a, b )`

- Data copy may be noncontiguous, but will then be slower (multiple DMAs)

- Data copy to / from pinned memory will be faster
Using the API

use cudafor
real, allocatable, device :: a(:)
real :: b(10), b2(2), c(10)
...
istat = cudaMalloc( a, 10 )
istat = cudaMemcpy( a, b, 10 )
istat = cudaMemcpy( a(2), b2, 2 )

istat = cudaMemcpy( c, a, 10 )
istat = cudaFree( a )
Pinned Memory

- Pinned memory is linear non-paged host memory; can do
  DMAs directly from pinned memory to the GPU device memory

- Pinned attribute for host data
  ```
  real, pinned, allocatable :: x(:, :)
  real, device, allocatable :: a(:, :)
  allocate( a(1:n, 1:m), x(1:n, 1:m) )
  ...
  a(1:n, 1:m) = x(1:n, 1:m) ! copies to device
  ....
  x(1:n, 1:m) = a(1:n, 1:m) ! copies from device
  deallocate( a, b )
  ```

- Downsides
  - Limited amount of pinned memory on the host
  - May not succeed in getting pinned memory
Behind the Scenes

- allocates/deallocates turn into cudaMalloc and cudaFree calls

- Assignments turn into cudaMemcpy calls
  - Non-contiguous data may be many many calls
  - Scalar assignments as well!

- Declarations of device data become
  - Static data is static on host + GPU
  - Non-static data is dynamically allocated on GPU
Time for a Live Demo (demo3, demo4)

CUDA Fortran Vector Add

Using the CUDA API

Pinned vs Non-pinned Data Bandwidth

Timing a Kernel
Writing a CUDA Fortran Kernel (1)

- global attribute on the subroutine statement
  - attributes(global) subroutine kernel ( A, B, C, N )
- May declare scalars, fixed size arrays in local memory
- May declare shared memory arrays
  - real, shared :: sm(16,16)
  - Limited amount of shared memory available
  - shared among all threads in the same thread block
- Data types allowed
  - integer(1,2,4,8), logical(1,2,4,8), real(4,8), complex(4,8), character(len=1)
  - Derived types
Writing a CUDA Fortran Kernel (2)

- **Predefined variables**
  - `blockidx, threadidx, griddim, blockdim, warpsize`

- **Executable statements in a kernel**
  - assignment
  - do, if, goto, case
  - call (to device subprogram, must be inlined)
  - intrinsic function call, device subprogram call (inlined)
  - where, forall
Disallowed statements include

- read, write, print, open, close, inquire, format, any IO at all
- allocate, deallocate, adjustable-sized arrays
- pointer assignment
- recursive procedure calls, direct or indirect
- ENTRY statement, optional arguments, alternate return
- data initialization, SAVEd data
- assigned goto, ASSIGN statement
- stop, pause
Supported Intrinsic Functions

- **Fortran numeric intrinsics**
  - abs
  - aimag
  - aint
  - anint
  - ceiling
  - cmplx
  - conjg
  - dim
  - floor
  - int
  - logical
  - max
  - min
  - mod
  - modulo
  - nint
  - real
  - sign

- **Fortran mathematical intrinsics**
  - acos
  - asin
  - atan
  - atan2
  - cos
  - cosh
  - exp
  - log
  - log10
  - sin
  - sinh
  - sqrt
  - tan
  - tanh
New Intrinsic Procedures

- call syncthreads()
  - synchronizes all threads in the same thread block at a barrier
- call gpu_time(clock)
  - returns clock cycle counter
- warp vote functions
  - allthreads(a(i)<0.0), anythread(a(i)<0.0)
- atomic functions, integer(4) only
  - = atomicadd( a[j], 1 ), atomicsub, etc.
  - = atomicinc( a[i] ), atomicdec
  - = atomiccas( a[i], compare, val )
Modules and Scoping

- **attributes(global) subroutine kernel** in a module
  - can directly access device data in the same module
  - can call device subroutines / functions in the same module

- **attributes(device) subroutine / function** in a module
  - can directly access device data in the same module
  - can call device subroutines / functions in the same module
  - implicitly private

- **attributes(global) subroutine kernel** outside of a module
  - cannot directly access any global device data (just arguments)

- **host subprograms**
  - can call any kernel in any module or outside module
  - can access module data in any module
  - can call CUDA C kernels as well (explicit interface)
Building a CUDA Fortran Program

- `pgfortran -Mcuda a.f90`
  - `pgfortran -Mcuda=[emu|cc10|cc11|cc13]`
  - `pgfortran a.cuf`
    - `.cuf` suffix implies CUDA Fortran (free form)
    - `.CUF` suffix runs preprocessor
    - `-Mfixed` for F77-style fixed format

- Debug code compiled with `-g -Mcuda=emu` on the host!
- Must use `-Mcuda` when linking from object files
- Must have appropriate gcc for preprocessor (Linux, Mac OSX)
  - CL, NVCC tools bundled with compiler
The –Mcuda Compiler Option

-Mcuda[=emu|cc10|cc11|cc13|keepbin|keepptx|maxregcount:<n>]

emu          Enable emulation mode
cc10         Compile for compute capability 1.0
cc11         Compile for compute capability 1.1
cc13         Compile for compute capability 1.3
keepbin      Keep CUDA binary files
keepptx      Keep PTX portable assembly files
maxregcount:<n> Set maximum number of registers to use on the GPU
Time for a Live Demo (demo5)

CUDA Fortran Vector Add

Debugging with Emulation Mode
## CUDA C vs CUDA Fortran

<table>
<thead>
<tr>
<th>CUDA C</th>
<th>CUDA Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>supports texture memory</td>
<td>no texture memory</td>
</tr>
<tr>
<td>supports Runtime API</td>
<td>supports Runtime API</td>
</tr>
<tr>
<td>supports Driver API</td>
<td>no support for Driver API</td>
</tr>
<tr>
<td>cudaMalloc, cudaFree</td>
<td>allocate, deallocate</td>
</tr>
<tr>
<td>cudaMemcpy</td>
<td>assignments</td>
</tr>
<tr>
<td>OpenGL interoperability</td>
<td>no OpenGL interoperability</td>
</tr>
<tr>
<td>Direct3D interoperability</td>
<td>no Direct3D interoperability</td>
</tr>
<tr>
<td>textures</td>
<td>no textures</td>
</tr>
<tr>
<td>arrays zero-based</td>
<td>arrays one-based</td>
</tr>
<tr>
<td>threadIdx/blockIdx 0-based</td>
<td>threadIdx/blockIdx 1-based</td>
</tr>
<tr>
<td>unbound pointers</td>
<td>allocatable are device/host</td>
</tr>
<tr>
<td>pinned allocate routines</td>
<td>pinned attribute</td>
</tr>
</tbody>
</table>
Interoperability with CUDA C

- CUDA Fortran uses the Runtime API
  - `use cudafor` gets interfaces to the runtime API routines
  - CUDA C can use Runtime API (cuda...) or Driver API (cu...)
- CUDA Fortran calling CUDA C kernels
  - explicit interface (interface block), add BIND(C)
  - interface
    - attributes(global) subroutine saxpy(a,x,y,n) bind(c)
    - real, device :: x(*), y(*)
    - real, value :: a
    - integer, value :: n
  - end subroutine
  - end interface
  - call saxpy<<<grid,block>>>( aa, xx, yy, nn )
Interoperability with CUDA C

- CUDA C calling CUDA Fortran kernels
  - Runtime API
  - make sure the name is right
    - module_subroutine_ or subroutine_
  - check value vs. reference arguments
  - extern __global__ void saxpy_( float a, float* x, float* y, int n );
    ...
    saxpy_( a, x, y, n );
  - attributes(global) subroutine saxpy(a,x,y,n)
    real, value :: a
    real :: x(*), y(*)
    integer, value :: n
Interoperability with CUDA C

- CUDA Fortran kernels can be linked with nvcc
  - The kernels look to nvcc just like CUDA C kernels

- CUDA C kernels can be linked with pgfortran
  - remember –Mcuda flag when linking object files
  - This CUDA Fortran release uses CUDA 2.3
Time for a Live Demo (demo7)

CUDA Fortran Vector Add

CUDA Fortran calling CUDA C kernels
CUDA C calling CUDA Fortran kernels
Backup Slides
Performance Tuning

- Performance Measurement
- Choose an appropriately parallel algorithm
- Optimize data movement between host and GPU
  - frequency, volume, regularity
- Optimize device memory accesses
  - strides, alignment
  - use shared memory, avoid bank conflicts
  - use constant memory
- Optimize kernel code
  - redundant code elimination
  - loop unrolling
  - Optimize compute intensity
    - unroll the parallel loop
Host-GPU Data Movement

- Avoid altogether
- Move outside of loops
- Better to move a whole array than subarray
- Update halo regions rather than whole array
  - use GPU to move halo region to contiguous area
- Use streams, overlap data / compute
  - requires pinned memory
Kernel Launch Configuration

- Kernel launch configuration affects occupancy
- Want many threads per thread block
  - multiple of 32
  - 64, 128, 256
- Want many many thread blocks
Low-level Optimizations

- instruction count optimizations
  - loop unrolling (watch memory access patterns)
  - loop fusion
- minimize global memory accesses
  - use scalar temps
  - scalarizing arrays
  - downsides:
    increased register usage
    spills to “local memory”