GPU Programming with PGI CUDA Fortran

Michael Wolfe
Michael.Wolfe@pgroup.com
http://www.pgroup.com

March 2010

PGI Workstation / Server / CDK
Linux, Windows, MacOS, 32-bit, 64-bit, AMD64, Intel 64
UNIX-heritage Command-level Compilers + Graphical Tools

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Language</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGF95™</td>
<td>Fortran 95 w/some F2003</td>
<td>pgf95</td>
</tr>
<tr>
<td>PGCC®</td>
<td>ANSI C99, K&amp;R C and GNU gcc Extensions</td>
<td>pgcc</td>
</tr>
<tr>
<td>PGC++®</td>
<td>ANSI/ISO C++</td>
<td>pgCC</td>
</tr>
<tr>
<td>PGDBG®</td>
<td>MPI/OpenMP debugger</td>
<td>pgdbg</td>
</tr>
<tr>
<td>PGPROF®</td>
<td>MPI/OpenMP profiler</td>
<td>pgprof</td>
</tr>
</tbody>
</table>

Self-contained OpenMP/MPI Development Solution

© Copyright 2009-2010, The Portland Group, Inc.
CUDA Fortran

- Simple introductory program
- Programming model
- Low-level Programming with CUDA Fortran
- Building CUDA Fortran programs
- Performance Tuning
Fortran VADD on Host

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 VADD Device Code

module kmod
    use cudafor
    contains
        attributes(global) subroutine vaddkernel(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
CUDA Fortran VADD Host Code

```fortran
subroutine vadd( A, B, C )
use kmod
real(4), dimension(:) :: A, B, C
real(4), device, allocatable:: 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 vaddkernel<<<(N+31)/32,32>>>( Ad, Bd, Cd, N )
C(1:N) = Cd
deallocate( Ad, Bd, Cd )
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
Elements of CUDA Fortran - Host

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 vaddkernel<<<(N+31)/32,32>>>( Ad, Bd, Cd, N )
C(1:N) = Cd
deallocate( Ad, Bd, Cd )
end subroutine

Allocate device memory
Copy data to device
Launch a kernel
Copy data back from device
Deallocate device memory

CUDA Programming: the GPU

- A scalar program, runs on one thread
  - All threads run the same code
  - Executed in a grid of thread blocks
  - grid may be 1D or 2D (max 65535x65535)
  - thread block may be 1D, 2D, or 3D (max size 512)
  - blockidx gives block index in grid (%x,%y)
  - threadidx gives thread index within block (%x,%y,%z)

- Kernel runs implicitly in parallel
  - thread blocks scheduled by hardware on any multiprocessor
  - runs to completion before next kernel
Elements of CUDA Fortran - Kernel

```fortran
module kmod
  use cudafor
  contains
  attributes(global) subroutine vaddkernel(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
```

global means kernel

device attribute implied

value vs. Fortran default

blockidx from 1..(N+31)/32

threadidx from 1..32

array bounds test

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
  - 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 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
  - real, device, allocatable :: a(:,,:), b
  - allocate( a(1:n,1:m), b )
  - dealloc( 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**
  
  ```fortran
  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

```fortran
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 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)` \hspace{1cm} copies to device
  - `x(1:n,1:m) = a(1:n,1:m)` \hspace{1cm} copies from device
  - `deallocate( a, b )`

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

Launching Kernels

- **Subroutine call with chevron syntax for launch configuration**
  - `call vaddkernel <<< (N+31)/32, 32 >>> ( A, B, C, N )`
  - `type(dim3) :: g, b`
  - `g = dim3((N+31)/32, 1, 1)`
  - `b = dim3( 32, 1, 1 )`
  - `call vaddkernel <<< g, b >>> ( A, B, C, N )`

- **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
Launching Kernels

- Subroutine call with chevron syntax for launch configuration
  - call vaddkernel <<< (N+31)/32, 32 >>> (A, B, C, N)
  - type(dim3) :: g, b
    g = dim3((N+31)/32, 1, 1)
    b = dim3(32, 1, 1)
    call vaddkernel <<< g, b >>> (A, B, C, N)

- launch configuration
  - <<< grid, block >>>
  - grid, block may be scalar integer expression, or type(dim3) variable

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

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

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|cc12|cc13|cc20]`
  - `pgfortran a.cuf`
    - .cuf suffix implies CUDA Fortran (free form)
    - .CUF suffix runs preprocessor
    - `-Mfixed` for F77-style fixed format
- Must use `-Mcuda` when linking from object files
- Must have appropriate gcc for preprocessor (Linux, Mac OSX)
  - CL, NVCC tools bundled with compiler

CUDA C vs CUDA Fortran

**CUDA C**
- supports texture memory
- supports Runtime API
- supports Driver API
- cudaMalloc, cudaMemcpy
- directMemcp
- OpenGL interoperability
- Direct3D interoperability
- textures
- arrays zero-based
- threadidx/blockidx 0-based
- unbound pointers
- pinned allocate routines

**CUDA Fortran**
- no texture memory
- supports Runtime API
- no support for Driver API
- allocate, deallocate
- assignments
- no OpenGL interoperability
- no Direct3D interoperability
- no textures
- arrays one-based
- threadidx/blockidx 1-based
- allocatable are device/host
- pinned attribute
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
  - CUDA 3.0 will be an option when it becomes available

CUDA Fortran Matrix Multiplication Code Walkthrough

- do i = 1, N
  - do j = 1, M
    - C(i,j) = 0.0
  - do k = 1, L
    - C(i,j) = C(i,j) + A(i,k)*B(k,j)

- Kernel computes a 16x16 submatrix
  - Initially, assume matrix sizes are divisible by 16

- thread block is (16,16), grid is (N/16,M/16)
  - Each thread accumulates one element of the 16x16 block of C
  - K loop is strip mined in strips of size 16
  - Threads cooperatively load a 16x16 block of A and B
module mmulmod
contains
  attributes(global) subroutine mmul( A,B,C,N,M,L)
  real,device :: A(N,L),B(L,M),C(N,M)
  integer,value :: N,M,L
  integer :: i,j,kb,k,tx,ty

  real,shared :: Ab(16,16), Bb(16,16)
  real :: Cij
  tx = threadidx%x ; ty = threadidx%y
  i = (blockidx%x-1) * 16 + tx
  j = (blockidx%y-1) * 16 + ty
  Cij = 0.0
  do k = 1, L
    Cij = Cij + A(i,k) * B(k,j)
  enddo
  C(i,j) = Cij
end subroutine
end module
subroutine mmul( A, B, C )
use cudafor
use mmulmod
real, dimension(:,:) :: A, B, C
real, device, allocatable, dimension(:,:,:) :: Ad,Bd,Cd
type(dim3) :: dimGrid, dimBlock
integer :: N, M, L
N = size(C,1) ; M = size(C,2) ; L = size(A,2)
allocate( Ad(N,L), Bd(L,M), Cd(N,M) )
Ad = A(1:N,1:L)
Bd = B(1:L,1:M)
dimGrid = dim3( N/16, M/16 )
dimBlock = dim3( 16, 16, 1 )
call mmul<<<dimGrid,dimBlock>>>( Ad,Bd,Cd,N,M,L )
C(1:N,1:M) = Cd
deallocate( Ad, Bd, Cd )
end subroutine

end module

end subroutine
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
Occupancy

- How many simultaneously active warps / maximum (maximum is 24 or 32)
- Limits
  - threads per multiprocessor
  - thread blocks per multiprocessor
  - register usage
  - shared memory usage
- Low occupancy leads to low performance
- High occupancy does not guarantee high performance

Execution Configuration

- Execution configuration affects occupancy
- Want many threads per thread block
  - multiple of 32
  - 64, 128, 256
- Want many many thread blocks
Divergence

- Scalar threads executing in SIMD mode
  - if( threadIdx%x <= 10 )then
    foo = foo * 2
  else
    foo = 0
  endif

- Each path taken
  - do i = 1, threadIdx%x
    a(threadIdx%x,i) = 0
  enddo

- Only matters within a warp

Divergence

- Pad arrays to multiples of block size
  - i = (blockIdx%x-1)*64 + threadIdx%x
  - if( i <= N ) A(i) = ...
Global Memory

- **Stride-1, aligned accesses**
  - address is aligned to mod(threadidx%x, 16)
  - threadidx%x and threadidx%x+1 access consecutive addresses
  - alignment critical for Compute Capability 1.0, 1.1

- **Using shared memory as data cache**
  - Redundant data access within a thread
  - Redundant data access across threads
  - Stride-1 data access within a thread

Redundant access within a GPU Thread

```fortran
! threadidx%x from 1:64
! this thread block does 256 'i' iterations
ilo = (blockidx%x-1)*256
ihi = blockidx*256 - 1
...
do j = jlo, jhi
   do i = ilo+threadidx%x, ihi, 64
      A(i,j) = A(i,j) * B(i)
   enddo
endo
do
do```

© Copyright 2009-2010, The Portland Group, Inc.
Redundant access within a GPU Thread

```fortran
real,shared :: BB(256)
...
do ii = 0, 255, 64
   BB(threadidx%x+ii) = B(ilo+ii)
enddo
call syncthreads()
do j = jlo, jhi
   do i = ilo+threadidx%x, ihi, 64
      A(i,j) = A(i,j) * BB(i-ilo)
   enddo
enddo
```

Redundant access across GPU Threads

```fortran
! threadidx%x from 1:64
i = (blockidx%x-1)*64 + threadidx%x
...
do j = jlo, jhi
   A(i,j) = A(i,j) * B(j)
enddo
```
Redundant access across GPU Threads

```fortran
real, shared :: BB(64)

i = (blockidx%x-1)*64 + threadidx%x
...
do jb = jlo, jhi, 64
    BB(threadidx%x) = B(jb+threadidx%x)
    call syncthreads()
    do j = jb, min(jhi,jb+63)
        A(i,j) = A(i,j) * BB(j-jb+1)
    enddo
enddo
```

Stride-1 Access within a GPU thread

```fortran
! threadIdx%x from 1:32
i = (blockidx%x-1)*32 + threadIdx%x
...
ix = indx(i)
do j = jlo, jhi
    A(i,j) = A(i,j) * B(ix+j)
enddo
```
Stride-1 Access within a GPU thread

```
real, shared :: BB(33,32)
integer, shared :: IXX(32)
i = (blockidx%x-1)*32 + threadidx%x
...
ix = indx(i)
IXX(threadidx%x) = ix
call syncthreads()
do jb = jlo, jhi, 32
   do j = 1, 32
      BB(threadidx%x,j) = B(IXX(j)+threadidx%x)
   enddo
   call syncthreads()
do j = jb, min(jhi,jb+31)
   A(i,j) = A(i,j) * BB(j,threadidx%x)
   enddo
```

Shared Memory

- 16 memory banks
- Use threadidx%x in leading (stride-1) dimension
- Avoid stride of multiple of 16
- Shared memory also used to pass kernel arguments, affects occupancy
Unroll the Parallel Loop

- If thread ‘j’ and ‘j+1’ share data, where
  - j is a parallel index
  - j is not the stride-1 index

- Unroll two or more iterations of ‘j’ into the kernel

module mmulmod
contains
    attributes(global) subroutine mmul( A,B,C,N,M,L)
    real,device :: A(N,L),B(L,M),C(N,M)
    integer,value :: N,M,L
    integer :: i,j,kb,k,tx,ty
    real,shared :: Ab(16,16), Bb(16,16)
    real :: Cij
    tx = threadIdx%x ; ty = threadIdx%y
    i = (blockIdx%x-1) * 16 + tx
    j = (blockIdx%y-1) * 16 + ty
    Cij = 0.0
! continued
module mmulmod
contains
  attributes(global) subroutine mmul(A,B,C,N,M,L)
    real,device :: A(N,L),B(L,M),C(N,M)
    integer,value :: N,M,L
    integer :: i,j,kb,k,tx,ty
    real,shared :: Ab(16,16), Bb(16,16)
    real :: Cij
    tx = threadIdx%x ; ty = threadIdx%y
    i = (blockIdx%x-1) * 16 + tx
    j = (blockIdx%y-1) * 16 + ty
    Cij = 0.0
  end subroutine
end module
module mmulmod
contains
! unroll two ‘j’ iterations
attributes(global) subroutine mmul(A,B,C,N,M,L)
  real,device :: A(N,L),B(L,M),C(N,M)
  integer,value :: N,M,L
  integer :: i,j,kb,k,tx,ty
  real,shared :: Ab(16,16), Bb(16,16)
  real :: Cij1, Cij2
  tx = threadidx%x ; ty = threadidx%y
  i = (blockidx%x-1) * 16 + tx
  j1 = (blockidx%y-1) * 32 + ty
  j2 = j1+16
  Cij1 = 0.0
  Cij2 = 0.0
! continued

  do kb = 1, L, 16
    Ab(tx,ty) = A(i,kb+ty-1)
    Bb(tx,ty) = B(kb+tx-1,j)
    call syncthreads()
    do k = 1,16
      Cij = Cij + Ab(tx,k) * Bb(k,ty)
      enddo
    call syncthreads()
  enddo
  C(i,j) = Cij
end subroutine
end module
do kb = 1, L, 16
   Ab(tx,ty) = A(i,kb+ty-1)
   Bb(tx,ty) = B(kb+tx-1,j1)
   call syncthreads()
   do k = 1,16
      Cij1 = Cij1 + Ab(tx,k) * Bb(k,ty)
      call syncthreads()
   enddo
   Bb(tx,ty) = B(kb+tx-1,j2)
   call syncthreads()
   do k = 1,16
      Cij2 = Cij2 + Ab(tx,k) * Bb(k,ty)
      call syncthreads()
   enddo
enddo
C(i,j1) = Cij1
C(i,j2) = Cij2
end subroutine
end module

Constant Memory

- Small, read-only, written by the host
  - assignment or API
- Hardware cached
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”

Coming CUDA Fortran Features

- Module allocatable device arrays
  - directly accessible by kernel routines in that module
- Device array pointers
  - pointer assignment on the host
- Assumed-shape argument arrays
Fermi vs Tesla

- ECC
- double = float / 2
- two level hardware data cache
- constant memory cache
- 16/48KB shared memory
- <=16 kernels at a time
- 2*16TP * 32SM
- unified address space
- dynamic allocation (?)
- enhanced support for C++

- no ECC
- double = float / 8
- no user-visible hardware cache
- constant memory cache
- 16KB shared memory
- 1 kernel at a time
- 8TP * 30SM
- shared/local/global ptrs
- allocate from host only

Copyright Notice

© Contents copyright 2009-2010, The Portland Group, Inc. This material may not be reproduced in any manner without the expressed written permission of The Portland Group.