Porting Applications with CUDA Fortran and OpenACC

Jeff Larkin
Cray Center of Excellence
<larkin@cray.com>

CUDA Fortran

- **CUDA Fortran is an extension of Fortran developed in cooperation between Nvidia and PGI that provides similar capabilities and limitations to CUDA for C.**
- **Pros**
  - It's FORTRAN - No reason to port parts of an existing FORTRAN code to C
  - Close to the metal performance
  - CUDA for C experience and best practices apply
  - IT'S FORTRAN!!!!
- **Cons**
  - Not Portable - Only available from PGI and Nvidia
  - Requires separate CPU and GPU code paths
- **One more Pro - Mixes well with OpenACC and CUDA Libraries**

What does it look like? - CUDA Fortran Kernel

```
real(kind=real_kind) ::  gv(nv,nv,2),vvtemp(nv,nv
do q=1,qsize
 do k=1,nlev
   do j=1,nv
     do i=1,nv
       gv(i,j,1) = &
        elem%metdet(i,j)*(elem%Dinv(1,1,i,j)*v5d(i,j,k,q,1) + &
        elem%Dinv(1,2,i,j)*v5d(i,j,k,q,2))
       gv(i,j,2) = &
        elem%metdet(i,j)*(elem%Dinv(2,1,i,j)*v5d(i,j,k,q,1) + &
        elem%Dinv(2,2,i,j)*v5d(i,j,k,q,2))
     enddo
   enddo
   do j=1,nv
     do l=1,nv
       dudx00=0.0d0
       dvdy00=0.0d0
       do i=1,nv
         dudx00 = dudx00 + deriv%Dvv(i,l  ) * gv(i,j,1)
         dvdy00 = dvdy00 + deriv%Dvv(i,l  ) * gv(j,i,2)
       end do
       div4d(l  ,j,k,q  ) = dudx00
       vvtemp(j  ,l  ) = dvdy00
     end do
   end do
   do j=1,nv
     do i=1,nv
       div4d(i,j,k,q)= elem%rmetdetp(i,j) * &
                       (rdx*div4d(i,j,k,q)+rdy*vvtemp(i,j))
     end do
   end do
 end do
end do
```

What does it look like? - CUDA Fortran Launcher

```
do q=1,qsize
 do k=1,nlev
   do j=1,nv
     do i=1,nv
       gv(i,j,1) = &
        elem%metdet(i,j)*(elem%Dinv(1,1,i,j)*v5d(i,j,k,q,1) + &
        elem%Dinv(1,2,i,j)*v5d(i,j,k,q,2))
       gv(i,j,2) = &
        elem%metdet(i,j)*(elem%Dinv(2,1,i,j)*v5d(i,j,k,q,1) + &
        elem%Dinv(2,2,i,j)*v5d(i,j,k,q,2))
     enddo
   enddo
   do j=1,nv
     do l=1,nv
       dudx00=0.0d0
       dvdy00=0.0d0
       do i=1,nv
         dudx00 = dudx00 + deriv%Dvv(i,l  ) * gv(i,j,1)
         dvdy00 = dvdy00 + deriv%Dvv(i,l  ) * gv(j,i,2)
       end do
       div4d(l  ,j,k,q  ) = dudx00
       vvtemp(j  ,l  ) = dvdy00
     end do
   end do
   do j=1,nv
     do i=1,nv
       div4d(i,j,k,q)= elem%rmetdetp(i,j) * &
                       (rdx*div4d(i,j,k,q)+rdy*vvtemp(i,j))
     end do
   end do
 end do
end do
```

CUDA Fortran Optimization

- **CUDA Fortran is just CUDA, so the same tools and techniques apply**
  - Use CUDA profiler by setting `COMPUTE_PROFILE=1` at runtime.
  - Use -Mcuda=ptxinfo for register and memory usage, useful with the Nvidia occupancy calculator
- **Shared memory, constant memory, coalesced memory operations, warp-divergence, etc. work just like CUDA for C**
- **Data transfer is key**
  - It doesn't matter how fast your kernel is if you're copying data inefficiently.
- **"Pin" your buffers to gain PCIe bandwidth and asynchronous transfer**

CUDA FORTRAN Best Practices

Use Interface Blocks

- Fortran Interface blocks allow overloading procedure name dependent on input types
- The “device” attribute can be used to specialize input arguments.
- So… by creating a generic interface, CPU and GPU routines can have the same calling sequence and will be picked at runtime according to be local to memory

```
module addone_mod
 interface addone
  module procedure &
       addone_host,addone_dev
 end interface
 contains
 subroutine addone_host(B,N)
   integer :: N
   real(8) :: B(N)
 end subroutine
 subroutine addone_dev(B,N)
   integer :: N
   real(8),device :: B(N)
   type(dim3) :: griddim,&
                 blockdim
 end subroutine
end module
```

Use CudaMemCpy rather than ‘=‘

- CUDA Fortran has an awesome feature that allows copying data (syncrhonously) using the standard assign operator (=)
- In several cases, we saw assignment result in many, small copies, rather than 1 large.
- Replacing with a call to cudaMemCpy or cudaMemCpyAsync gave much better results

- Aside: In theory one could overload the ‘=‘ operator within a module and implement this shortcut oneself, but we did not try this.

Remember that memcpy is faster than PCIe

- It's tempting to think that streaming PCIe copies over chunks is cheaper than packing/unpacking buffers
- Don't do this:

```
do i=1,nchunks
   cudaMemcpy chunk
end do
```

- DDR3 Memory is capable of > 12 GB/s
- PCIe is capable of < 6GB/s

- Do this instead:

```
pack_chunks_on_device(chunks,buffer)
cudaMemcpy buffer
unpack_chunks_on_host(buffer,chunks)
```

- Make sure your buffer is "pinned"

Stupid Programmer Tricks

CUDA Fortran and OpenACC can do that??

Start with something simple…

- Create Vectors A & B, both of length N.
- We don't need to initialize them on the CPU, so create and initialize each on the device
- Return the results to the CPU, where they'll be output.

- What does this really show?
  - OpenACC is great at the high-level
  - No need for multiple copies of each array (device/host)
  - When possible, populate device arrays on the device to avoid the cost of a copy

```
program main
  integer, parameter  :: &
    N = 1000
  real(8) :: A(N),B(N)
  integer :: i

  !$acc data create(A), &
    copyout(B)
  !$acc parallel
  A(:) = 1.0
  !$acc end parallel
  !$acc parallel
  B(:) = 2.0 * A(:)
  !$acc end parallel
  !$acc end data

  print *,B(1:6),"\n...\n",&
          B((N-5):N)
end program
```

Add some CUDA Fortran

```
module addone_mod
  use cudafor
  implicit none
  private
  public interface addone
    module procedure addone_host,addone_dev
  end interface
  contains
  subroutine addone_dev(B,N)
    integer :: N
    real(8),device :: B(N)
    type(dim3) :: griddim, blockdim
    griddim = dim3(ceiling(real(N)/real(512)),1,1)
    blockdim = dim3(512,1,1)
    call addone_kernel<<<griddim,blockdim>>>(B,N)
    print *, "device"
  end subroutine
  attributes(global)&
  subroutine addone_kernel(B,N)
    integer,value :: N
    real(8) :: B(N)
    integer i
    i = ((blockIdx%x - 1) * blockDim%x) + &
        threadIdx%x
    if (i.le.N) then
      B(i) = B(i) + 1
    endif
  end subroutine
end module
```

```
program main
  use addone_mod
  integer, parameter  :: N = 1000
  real(8) :: A(N),B(N)
  integer :: i

  !$acc data create(A), copyout(B)
  !$acc parallel
  A(:) = 1.0
  !$acc end parallel
  !$acc parallel
  B(:) = 2.0 * A(:)
  !$acc end parallel

  !DEVICE
  call addone(B,N)

  !$acc end data

  !HOST
  call addone(B,N)

  print *,B(1:6),"\n...\n",B((N-5):N)

end program
```

Add in a library or two…

```
program main
  use addone_mod
  use cublas
  integer, parameter  :: N = 1000
  real(8) :: A(N),B(N)
  integer :: i

  !$acc data create(A), copyout(B)
  !$acc parallel
  A(:) = 1.0
  !$acc end parallel
  !$acc parallel
  B(:) = 2.0 * A(:)
  !$acc end parallel

  !DEVICE
  call addone(B,N)

  call daxpy(N,alpha,A,1,B,1)
  !$acc end data

  !HOST
  call addone(B,N)

  print *,B(1:6),"\n...\n",B((N-5):N)

end program
```

- What's my point of this silly example?
- Just because you choose 1 programming model today, doesn't mean you're stuck with that choice.
- Mixing CUDA C, CUDA Fortran, Libraries, and OpenACC is both possible and reasonable.
- More on this in a moment…

My recommendations

If I were starting today, what would I do?

My Recommendations

- **Start with OpenACC**
  - OpenACC has matured to the point that it is useful for most applications.
  - If you do a lot of partial array updates and/or partial derived type updates, you may still have some trouble
  - It helps to have an efficient OpenMP code first.
- **Get the data movement figured out first.**
  - If it takes longer to copy the data back and forth than computing on the CPU, kernels can be infinitely fast and it won't matter.
- **If you find things that are hard or inefficient to do via directives, fall back to CUDA (C or Fortran)**
  - Don't forget to use accelerated libraries when available.
- **Report bugs!!**
  - Compilers won't get better if we don't know they're broken.
  - Poor performance is a bug too, if you can beat our performance, show us the code.