

Fortran on GPUs

STAR Working Session: GPGPUs

02/25/11

Lars Koesterke, Ph.D.

High Performance Computing
Texas Advanced Computing Center



THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Fortran on GPUs

Answers to the most important questions in 15 minutes!

- Fortran for HPC, not Vis!
- Who provides and supports it?
- What is it?
- How does it work?
- Why is it important?
- What are the capabilities and benefits?
- **Why is it so exciting?**

Who provides and supports Fortran on GPUs?

- Available on NVIDIAs GPUs
- Developed by “The Portland Group” (PGI)
 - Partnership with NVIDIA
 - Strong interest by NVIDIA to keep the compiler up-to-date as the evolution of CUDA continues
- Name: PGI CUDA Fortran Compiler
- Two usage models
 - CUDA Fortran, allows also for automatic kernel generation
 - CUDA Fortran Accelerator Directives (similar to OpenMP)

What is CUDA Fortran?

- Extension to Fortran that enables access to NVIDIA GPUs through “native” Fortran
- No C code necessary
 - Host code written in Fortran
 - Device code (Kernels) written in Fortran
 - No need to call C-kernels from Fortran
- Complete CUDA C API available in Fortran

How does CUDA Fortran work?

- PGI CUDA Fortran leverages:
 - Existing Fortran constructs
 - Variable/function/subroutine attributes, allocate/deallocate
 - Array syntax for data transfer
 - CUDA C language elements (translated to native Fortran!) and naming conventions
 - Host, global, device, kernel, etc.
 - thread grids, thread blocks
- Compile with flag: *pgf90 -Mcuda* (or *-Mcuda=emu*)
- Generally: Kernels are simple by nature
 - Number of instructions, number of arguments in calls, etc.
 - Limited complexity of code-logic (IF-THEN-ELSE constructs)
 - Limited complexity of language constructs (OO, Polymorph.)
- Source-to-source translation of the kernel
 - Details hidden under the covers

Why is CUDA Fortran important?

- One half of the users of our large clusters at TACC (Ranger, Lonestar, Longhorn) code in Fortran
 - Half of all source code is written in Fortran
 - Half of the service units (SUs) are consumed by Fortran executables
 - The other half is C (including some C++)
 - ... and there are users that need to move on to one of these languages
- The same applies to other HPC communities
 - The fraction of Fortran users and Fortran code in HPC is large
- Vendors that embrace Fortran users have a huge advantage in the “Battle of the Accelerators”

What are the capabilities and benefits?

- CUDA Fortran provides full* access to the GPU
 - Kernel launch
 - GPU memory
 - Main memory (GDDR5)
 - Shared and constant memory
 - Registers
 - *no easy access to texture memory yet
 - “Pinned” memory on CPU
 - Synchronous data transfer
 - Asynchronous data transfer and streams
 - Grids and blocks of threads
 - Thread synchronization

What are the capabilities and benefits?

- CUDA Fortran looks and feels like Fortran
- Existing Fortran language elements are expanded
 - Attributes for variables and subroutines/functions
 - Declare where a variable/array is stored: host or device
 - Declare where a subroutine/function is executed: host or device
 - Array syntax
 - Transfer of data between host and device
 - Predefined structures with thread grid & block information
- No “clunky” API calls to declare/allocate variables and to transfer data
 - Full API is exposed,
 - all major API calls are integrated into the Fortran language

Why is CUDA Fortran so exciting?

- CUDA C exposes the API to programmers
- CUDA Fortran does the same, but goes much farther ...
- CUDA Fortran is an extension to the Fortran language

Why is access to the API not enough?

- Compilers are here to help!
 - Integration allows compiler to optimize for performance
 - HPC languages (Fortran, C++) evolve to fit our needs
 - Code becomes more readable if API is integrated
 - Automatic kernel generation
 - Better code can be written faster ...
- We are already “stuck” with one “bare-metal” API
 - All attempts to integrate MPI have given us bad performance
 - We can’t afford to pile on API after API

A CUDA Fortran example in a nutshell

New attributes Predefined structures
Data/Subroutine on device
Replacement for API call
Chevron syntax for Kernel launch

```
program cuda_fortran

real, allocatable, dimension(:,:,:) :: h      ! h on Host
real, allocatable, dimension(:,:,:), device :: d  ! d on Device
real :: a = 2.1 ! variable

allocate (h(512,5,20), d(512,5,20)) ! Allocate on Host and Device

h = ... ! Preset h
d = h ! Transfer 3-dim array from host to device (Array syntax)

dimblock = dim3(512,1,1) ! Block of Threads (dim3: derived type)
dimgrid = dim3(5, 20,1) ! Grid of Threads

call sub_gpu<<<dimgrid,dimblock>>>(d,a)

h = d ! Transfer 3-dim array back

end
```

```
k = griddim%y
d(i,j,k) = d(i,j,k) + a
end
```

A CUDA Fortran example in a nutshell

New attributes Predefined structures
Data/Subroutine on device
Replacement for API call
Chevron syntax for Kernel launch

```
program cuda_fortran

real, allocatable, dimension(:,:,:) :: h      ! h on Host
real, allocatable, dimension(:,:,:), device :: d    ! d on Device
real :: a = 2.1 ! variable

allocate (h(512,5,20), d(512,5,20)) ! Allocate on Host and Device

h = ... ! Preset h
d = h ! Transfer 3-dim array from host to device (Array syntax)

dimblock = dim3(512,1,1) ! Block of The
dimgrid = dim3(5, 20,1) ! Grid of The

call sub_gpu<<<dimgrid,dimblock>>>(d,a)

h = d ! Transfer 3-dim array back

end

attributes(global) sub_gpu(d,a)

real, dimension(:,:,:) :: d
real, value :: a
integer :: i, j, k

i = blockdim%x
j = griddim%x
k = griddim%y

d(i,j,k) = d(i,j,k) + a

end
```

CUDA Fortran “Kernel Loop Directives”

```
module madd_device_module
  use cudafor
contains
  subroutine madd_dev(a,b,c,n1,n2)
    real, dimension(:, :), device :: a,b,c
    integer :: n1,n2
    real :: sum = 0.
    !$cuf kernel do(2) <<<(*,*) , (32,4)>>>
    do j=1, n2
      do i=1, n1
        a(i,j) = b(i,j) + c(i,j)
        sum = sum + a(i,j)
      enddo
    enddo
  end subroutine
end module
```

New attributes Thread parameters
Data/Subroutine on device
Compiler directive creating kernel
Chevron syntax for Kernel launch

“Kernelize” the two (2) loops
!\$cuf similar to OpenMP !\$omp
Use 2D blocks: x=32 y=4
Grid shape automatic (*,*)
Compiler can handle a
“global” reduction

No Synchronization beyond a thread block!

A global reduction requires:

- Reduction on block level (Step I) within kernel
- Exiting kernel for global synchronization
- Second kernel for reduction on grid level (Step II)
- Both reductions are operations on a tree

CUDA code becomes very convoluted very quickly!

CUDA Fortran is forward-looking ...

- CUDA Fortran empowers 50% of the HPC community
 - Exposure of the API
 - Integration into language, way ahead of C and C++
 - Expect to see integration of CUDA into C++
 - GPU-language extension is easier to learn than “bare” API
- Other vendors? AMD, Intel
- OpenMP accelerator directives: pushed by Cray, etc.
- Allow the compiler to:
 - help us code more efficiently
 - optimize for execution speed
 - minimize and group data transfer
- APIs can be the way to go (e.g. MPI), but ...

Getting to Exascale means developing new languages/language paradigms, not using more bare APIs