Solving PDEs with PGI CUDA Fortran
Part 2: Introduction to PGI CUDA Fortran

Outline

Why Fortran, why CUDA Fortran. Compilers for NVIDIA GPUs. Hierarchy of CUDA Fortran, CUDA C and CUDA Runtime API. Kernel and device subroutines. GPU memory specification and allocation. Host-device data transfers. Launching kernels. Porting source codes: to-do list. Compiler switches. Source-code examples.

Why Fortran

– a well-established programming language for scientific and engineering applications
– supports high performance computing and parallelization (OpenMP, MPI, ...)
– ready-to-link optimized numerical libraries (LAPACK, NAG, IMSL, ...)
– hides some technicalities ("pointers and asterisks are not for everybody")
– standardized interoperability with C
– and of course, http://www.pbm.com/~lindahl/real.programmers.html

Why PGI CUDA Fortran and PGI Accelerator

– CUDA Fortran: a small set of extensions to Fortran that supports and is built upon the CUDA computing architecture
– PGI Accelerator: directives similar to OpenMP style to define accelerated regions and corresponding data
– a Fortran programmer can keep living in the Fortran world:
    device data can be declared and allocated by Fortran specification and allocation statements
    host-device data transfer can be done by Fortran assignment statements (including array syntax)
    kernels can be written and launched using extended Fortran syntax
– CUDA Fortran is higher-level programming model relative to CUDA C
– PGI Accelerator is higher-level programming model relative to CUDA Fortran

Compilers for NVIDIA GPUs

nvcc: a free C/C++ proprietary compiler by NVIDIA, included in the CUDA Toolkit
    a command-line tool on top of a standard C/C++ compiler
    provides calls to (higher-level) CUDA Runtime API and (lower-level) CUDA Driver API
    June 2011: release version 4.0; previous versions 3.2, 3.1, 2.3

PGI Workstation: a commercial compiler suite by PGI (the Portland Group)
    Fortran 95/2003 and C/C++ compilers
    for Linux, Windows, MacOS, 32 and 64 bits, with OpenMP and MPI support
    the command-line interface, for Windows: the Microsoft Visual Studio Express environment
    GPU support: CUDA Fortran, CUDA C/C++, PGI Accelerator directives
    selected parts of CUDA Toolkit in the suite: nvopencc, CUDA Runtime API, CUBLAS, CUFFT
    June 2011: release version 11.6 with CUDA Toolkits 3.2 and 4.0

a working set: NVIDIA graphics driver (with support of CUDA Toolkit used by PGI)
    PGI Workstation or PGI Server (Linux, Windows, MacOS) or PGI Visual Fortran (Windows)
    optional: NVIDIA CUDA Toolkit (requires gcc on Linux, MS Visual C++ Express on Windows)

CUDA Fortran mission

writing kernel subroutines and device procedures

```
ATTRIBUTES(KERNEL) SUBROUTINE MyKernel(arguments)
```

declaring and allocating data in the device or on-chip memory

```
INTEGER,ALLOCATABLE,DEVICE :: ad(:) ! dmem
REAL :: b                           ! registers or lmem
REAL,CONSTANT :: pi                 ! cmem
COMPLEX,SHARED :: c(nmax)           ! smem
```

transferring data between host and device

```
ad=a ; ... ; a=ad
```

launching kernels

```
call MyKernel<<<gridsize,blocksize>>>(arguments)
```

calling CUDA Runtime API routines

```
istatus=cudaThreadSynchronize()
```

accessing definitions of CUDA types and interfaces

```
use cudafor
```

Kernel subroutine and device procedures

Kernel subroutine
– launched by the host for execution on GPU
– specified by ATTRIBUTES(GLOBAL)
– written for a single thread, executed by each thread (the total number of threads is assigned by the kernel call)
– typically updates an array passed as an argument, often one array element by one thread

Supported datatypes
– INTEGER(1,2,4,8), REAL(4,8), COMPLEX(4,8), LOGICAL(1,2,4,8), CHARACTER(1) and derived types

Supported statements
– assignment statements, including the array language
– conditional statements and constructs IF and SELECT CASE
– loops DO with index variable, DO WHILE and unbounded DO, along with CYCLE and EXIT statements
– statements CONTINUE, GOTO, CALL and RETURN

Supported Fortran and CUDA intrinsics
Fortran: abs, aimag, aint, ..., min, max, ..., acos, asin, atan, ..., all, any, count, maxloc, maxval, sum, ...
CUDA: fma_rn, fma_rz, fma_ru, fma_rd, fmaf_rn, ... (many others)

Constraints
– cannot contain STOP and PAUSE
– cannot contain input/output commands (PRINT *,scalar possible since ver. 11.6)
– cannot contain ALLOCATABLE or POINTER data
– cannot be RECURSIVE, PURE or ELEMENTAL
– cannot contain procedures, cannot be contained in a procedure (can appear in a module)

Arguments
– arrays must be ready in device memory already, scalars can be in host memory
– actual arguments in device memory are passed by reference (arrays always, scalar optionally)
and are shared by all threads – they are in global memory (residing in device memory),
corresponding dummy arguments have the DEVICE attribute by default
– actual arguments in host memory (scalars only) are passed by value
and are private to each thread – they are in registers or in local memory (residing in device memory),
corresponding dummies must have the VALUE attribute

Device procedures
– subroutines and functions that can be called from a kernel or another device procedure (not from a host procedure)
– specified by ATTRIBUTES(DEVICE)

Thread indexing
– a unique thread index that can (must) be found from built-in variables...
a) threadidx for the index of a thread in a block (1 is the lowest)
b) blockidx for the index of a block in a grid (also one-based)
c) blockdim for the size and shape of a block
d) griddim for the size and shape of a grid
– the variables are structures of type dim3: type dim3 ; integer x,y,z ; end type
i.e., they allow for 1D (via x component), 2D (x,y) and 3D (x,y,z) block and grid shapes

Correspondence between array indexes and thread and block indexes can be done in many ways,
but the threadidx%x should always correspond to consecutive array elements (for memory coalescing)
– 1D array, 1D block, 1D grid: i=threadidx%x+blockdim%x*(blockidx%x-1)
– 1D array, 2D block, 1D grid: i=threadidx%x+blockdim%x*((threadidx%y-1)+blockdim%y*(blockidx%x-1))
– 2D array, 1D block, 1D grid: i=threadidx%x, j=blockidx%x etc.
– Fortran-matrices have the column-major order ("1st index is changing fastest"),
i.e., threadidx%x should correspond to the array index for the first dimension

Synchronization
– threads running in parallel occasionally need synchronization at a certain point (a barrier)
– a barrier for all threads of a block is a call to a CUDA routine: call syncthreads() and variants
– a barrier for all threads of a grid is the end of the kernel: end subroutine

GPU memory specification and allocation

Device memory (dmem)

– resides on the device, accessible by both the host and the device
– bandwidth to the host side slower (via PCI Express slot, ~< 6 GB/s), to multiprocessors faster (~< 100 GB/s)
– used primarily for global memory (public data shared by all threads),
    also for local memory (private data local in each thread if registers are exhausted and spilled),
    also for constant memory (read-only public data, cached on multiprocessors),
    also for texture memory (similar, but not accessible by CUDA Fortran)
– provides a two-way bridge between the host and the device:
    a host procedure declares, allocates and (optionally) initializes data in dmem,
    the dmem variables are passed as kernel arguments by reference,
    if updated in the kernel, the host procedure can copy the dmem data back to host memory
– variables in dmem are declared with the DEVICE attribute:
    if in a host procedure, the variable can appear only in allocations and deallocations, in assignment
        statements (as the source and/or the destination) and as an actual or dummy argument
    if in a kernel as a dummy argument, the variable is public and shared by all threads
        (the DEVICE attribute of dummy arguments is implicit for actual arguments from dmem)
    if in a kernel as a local variable, the variable is private in each thread
– device arrays in a host procedure can be both static and allocatable, local arrays in kernels can be static only
– if threads of a warp access dmem arrays by consecutive elements, they will all be served simultaneously
    because of the device memory coalescing
– the ordering of multidimensional arrays in linear memory must be taken into account:
    the column-major order used by Fortran (as opposite to C) implies that the thread index threadidx%x
    should be used as the array index for the first dimension
– fully fledged dmem coalescing also requires correct alignment of accessed array elements but failing to do so
    will result in two memory accesses at worst

Registers and local memory (lmem)

– registers reside in fast on-chip memory, lmem in slower device memory, but it can be stored in L1 or L2 cache
– registers readable with no latency, dmem latency is hundreds of clock cycles, cache latency in-between
– used for private data local in each thread, not accessible by the host
– variables in registers and lmem can appear in kernels and device procedures, not in host procedures
– variables in registers and lmem are declared without any CUDA attribute
– kernel dummy arguments with the VALUE attribute are stored into registers or lmem

Constant memory (cmem)

– resides in device memory (64 KB) and can be cached on each multiprocessor (8 KB)
– must be initialized in a host procedure and is read-only in a kernel and device procedures
– variables in cmem are declared with the CONSTANT attribute which is illegal in host procedures, thus,
    cmem data should be declared in a module with a kernel and made visible to a host procedure
– cmem broadcasting: when more threads of a warp read the same word from cmem cache, they will all
    be served simultaneously

Shared memory (smem)

– resides in fast on-chip memory of 16 KB (CC 1.x/2.x) or 48 KB (CC 2.x) on each multiprocessor
– used for data shared by all threads in a block (but not among different blocks)
– variables in smem are declared with the SHARED attribute in a kernel or a device procedure
    and must be initialized by threads
– the amount of smem required by a block poses a limit on a number of resident blocks in a multiprocessor,
    e.g., real(8) shared array of 1024 elements occupies 8 KB and, with 48 KB smem per multiprocessor,
    max. 6 blocks can be resident on a multiprocessor (but 8 blocks is the upper limit anyway)
– smem allows fast access as is (an order of magnitude faster than uncached access to dmem)
– moreover, when threads of a warp access consecutive 4B words in smem, they will all be served simultaneously
    because of the ability of smem to access smem banks simultaneously
    (consecutive words are assigned to consecutive smem banks)
– smem broadcasting: when more threads of a warp read the same word from smem, they will all be served
    simultaneously
– a smem bank conflict occurs when two or more threads of a warp access a different word in the same smem bank

Host-device data transfer

– simple assignment statements in host procedures
– statements provided for: host-to-device transfer, `ad=a`
device-to-host transfer, `a=ad`
device-to-device transfer, `bd=ad`
– for arrays, one contiguous block transfer shoud be preferred
– similarly, host-to-constant-memory transfer can be issued
– on systems with integrated host and device memory, the mapped page-locked memory should be used,
data transfer would then be superfluous
– CUDA memory management routines are also provided

Launching kernels

– a kernel call statement is expected to set the number of threads that will execute the kernel by assigning
the execution configuration, i.e., actual grid and block sizes
– grid and block sizes are either scalar integers or dim3 structures,
`grid=dim3(1024,1024,1), block=256`
– extended form of the CALL statement:
`CALL Kernel<<<grid,size>>>(arguments)`
– the chevrons can optionally specify amount of dynamically assigned shared memory (a scalar integer bytes)
and the stream identification (0 or a scalar integer returned by the cudaStreamCreate function)
`<<<grid,size,bytes,stream>>>`
– grid and block sizes have to satisfy compute-capability limits, dynamically assigned smem must be available

Porting source codes from CPU Fortran to CUDA Fortran: To-do list

1. extract parallelizable portions of the code (most often with one or more loops)
into subroutines contained in a module
2. edit the kernel:
set the global attribute
set the value attributes of kernel arguments passed by value
substitute outer loops with thread indexing
3. edit the host procedure:
attach the cudafor module
set grid and block shape and sizes
allocate device memory data
transfer data from host memory to device memory
set the execution configuration by chevrons
pass kernel arguments:
arrays in device memory by reference
scalars in host memory by value
transfer data from device memory back to host memory

Compiler switches

pgfortran -help [option]	or pgf77, pgf90, pgf95
pgfortran -V	version information
pgfortran file.f90	no optimization
pgfortran -fast file.f90	common local optimization set, see pgfortran -help -fast
pgfortran -fastsse file.f90	more local optimizations on 32bit systems (equivalent to -fast on 64bit systems)
pgfortran -fast -Mipa=fast file.f90	global optimization
pgfortran -g file.f90	debugging information
pgfortran -Mcuda file.f90	enable CUDA Fortran

Mcuda suboptions: -Mcuda=cc11,cc13,cc20,3.1,3.2,4.0,emu,keepgpu,ptxinfo etc.

cc11, cc13, cc20	specific compute capability (default: all; cc21 not yet available)
3.1, 3.2, 4.0	specific CUDA Toolkit compatibility (default in ver. 11.6: 3.2, in 11.5: 3.1)
emu	emulation mode
keepgpu	keeping kernel CUDA C source files
ptxinfo	messages from ptxas about register and lmem/smem/cmem usage
	(ptxas = PTX-language assembler; PTX = Parallel Thread Execution)

pgaccelinfo utility
CUDA Fortran source coudes can have .cuf extension, then the -Mcuda option is default

CUDA Fortran source-code examples

Example 1 (template): addition of 1D array and a scalar
a(:)=a(:)+z
goals: setting the execution configuration – the grid and block sizes
passing arguments to a kernel – arrays by reference, scalars by value
correspondence of 1D arrays and thread indexing
Example 2: addition of 2D arrays by a device function
a(:,:)=a(:,:)+b(:,:)
goals: correspondence of 2D arrays and thread indexing
device procedures
Example 3: accessing 2D arrays in nested loops
a(:,:)=a(:,:)+b(:,:) once again
goals: efficient access to 2D arrays with column-major order on CPU and GPU
device memory coalescing
Example 4: avoiding divergent warps
each thread summing the harmonic series $\sum_{n=1}^{N} 1/n$
goals: execution time for various grid and block sizes
execution time for various amounts of diverging execution paths in a warp
Example 5: using shared memory (the 3-point moving average filter)
the moving average = a finite-impulse response filter that creates a series of averages of the original data set
$\text{for } n = 0, 1, 2, \ldots: a_0^{n+1} = (2a_0^n + a_1^n)/3, \text{ for } j = 1, \ldots, J: a_j^{n+1} = (a_{j-1}^n + a_j^n + a_{j+1}^n)/3, a_J^{n+1} = (a_J^n + a_{J+1}^n)/3$
goals: transfer of device memory data to shared memory
synchronization of threads in a block
and...

A final example: Mandelbrot set

wiki: a set of points, whose boundary generates a two-dimensional fractal shape
a set M of complex numbers c for which the $lim_{n\to\infty}|z_n|$ of the sequence $z_{n+1} = z_n^2 + c,\ z_0 = 0$, remains bounded
c is not in M, if $|z_n| > 2$ for any n
a source-code snippet:

```
complex cc,z ;
z=0.; do n=1,nmax ; z=z*z+cc ; if (abs(z)>2.) record_cc_and_exit ; enddo
```

note: abs(z)>2. may be rather slow, real(z)**2+imag(z)**2>4. is expected to evaluate faster
vizualization of Mandelbrot-set approximations:
for each c, the highest n, if any, for which abs(z_n)<=2, is recorded
all c corresponding to a fixed n form a set M_n, the n-th approximation of M
all M_n are vizualized, each with a different color

Links and references

PGI resources
CUDA Fortran Programming Guide and References, Release 2011
PGI Compiler User's Guide, Release 2011 (chapter Using an Accelerator)
PGI Compiler Reference Manual, Release 2011 (chapter PGI Accelerator Compilers Reference)
Articles, PGInsider newsletters, White papers and specifications, Technical papers and presentations
http://www.pgroup.com/resources/articles.htm
NVIDIA GPU Computing Documentation
NVIDIA CUDA C Programming Guide (esp., Chap. 5: Performance guidelines)
NVIDIA CUDA C Best Practices Guide
NVIDIA Tuning CUDA applications for Fermi
http://developer.nvidia.com/nvidia-gpu-computing-documentation
Source-code examples
CUDA Fortran SDK: C:\Program Files\PGI\win64\2011\cuda\CUDA Fortran SDK
Wolfe M., CUDA Fortran: The next level, PGInsider, 2010
http://www.pgroup.com/lit/articles/insider/v2n3a1.htm