PGI insider

Technical News from The Portland Group

PGI Home Page

October 2009

Introduction to PGI CUDA Fortran

NVIDIA CUDA™ is a general purpose parallel programming architecture with compilers and libraries to support programming of NVIDIA GPUs. The CUDA SDK includes an extended C compiler, here called CUDA C, allowing GPU programming from a high level language. The CUDA programming model supports four key abstractions: cooperating threads organized into thread groups, shared memory and barrier synchronization within thread groups, and coordinated independent thread groups organized into a grid.

PGI and NVIDIA defined CUDA Fortran, which is supported in the upcoming PGI 2010 release, to enable CUDA programming directly in Fortran. CUDA Fortran is a small set of extensions to Fortran that supports and is built upon the CUDA computing architecture. The extensions allow the following actions in a Fortran program:

- Declaration of variables that reside in GPU device memory
- Dynamic allocation of data in GPU device memory
- Copying of data from host memory to GPU memory, and back
- Invocation of GPU subroutines from the host

A CUDA programmer partitions a program into coarse grain blocks that can be executed in parallel. Each block is partitioned into fine grain threads, which can cooperate using shared memory and barrier synchronization. A properly designed CUDA program will run on any CUDA-enabled GPU, regardless of the number of available processor cores. This article will teach you the basics of CUDA Fortran programming and enable you to quickly begin writing your own CUDA Fortran programs.

CUDA Fortran Basics

Those of you familiar with CUDA programming know the following basic steps for creating a CUDA C program:

- Insert CUDA function calls to allocate data in GPU device memory
- Insert CUDA function calls to copy host data to the GPU, and back
- Isolate innermost loop code into GPU kernel functions
- Convert loop nests into implicitly parallel grids/thread blocks
- Use CUDA grids/thread blocks to launch GPU kernel functions
- Insert CUDA function calls to free GPU device memory

Three features of Fortran 90 were leveraged to enable use of more natural language constructs in CUDA Fortran: attributed declaration syntax, the allocate statement, and array assignments.

Rather than using a special data allocation function for GPU device data allocation, we can use a combination of F90 attributed declarations and allocate statements to create device-resident data:

```
real,device :: A(M,N)              ! A instantiated in GPU memory
real,device,allocatable :: B(:,:)  ! B is allocatable in GPU memory
...
allocate(B(size(A,1),size(A,2)),stat=istat) ! Allocate B in GPU memory
```

In addition to using the DEVICE attribute to specify data that should reside on the GPU device, SHARED and CONSTANT attributes can be used to place data in CUDA shared or constant memory respectively. A PINNED attribute can be used to specify host arrays to be placed in CUDA pinned memory.

Similarly, we can leverage Fortran 90 array assignment syntax to effect data copies from host memory to GPU device memory, and back. For example:

```
real :: A(M,N)            ! A instantiated in host memory
real,device :: Adev(M,N)  ! Adev instantiated in GPU memory
...
Adev = A                  ! Copy data from A (host) to Adev (GPU)
...
A = Adev                  ! Copy data from Adev (GPU) to A (host)
```

As a result of these features, CUDA Fortran programs can be written mostly in natural Fortran without many runtime library calls. As we'll see in the following sections, many other aspects of CUDA Fortran will be familiar to CUDA C programmers, including the pre-defined variables for initialization of grids and thread blocks and the chevron syntax for launching kernels for implicitly parallel execution on an NVIDIA GPU.

For consistency with CUDA C, and to enable the full range of CUDA software development features, CUDA Fortran also implements most of the CUDA Runtme API routines for device, thread, memory, stream and event management.

A Simple Program in CUDA Fortran

This example shows a program to compute the product C of two matrices A and B, as follows:

- Each thread block computes one 16x16 submatrix of C
- Each thread within the block computes one element of the submatrix

The submatrix size is chosen so the number of threads in a block is a multiple of the CUDA warp size (32) and is less than the maximum number of threads per thread block (512).

Each element of the result is the product of one row of A by one column of B. The program computes the products by accumulating submatrix products; it reads a block submatrix of A and a block submatrix of B, accumulates the submatrix product, then moves to the next submatrix of A row-wise and of B column-wise. The program caches the submatrices of A and B in the fast CUDA shared memory .

For simplicity, the program assumes the matrix sizes are a multiple of 16, and has not been highly optimized for execution time.

The source code for our matrix multiply example has two components, a host subroutine MMUL that allocates necessary data structures and initializes for GPU execution, and an MMUL_KERNEL kernel subroutine containing the code to be launched and executed on the GPU.

The host MMUL subroutine

The host subroutine has two input arrays, `A` and `B`, and one output array, `C`, passed as assumed-shape arrays. Notice in the declarations section that the allocatable arrays `Adev`, `Bdev` and `Cdev` are defined with the device attribute. These will be allocated in GPU device memory and used as copies of the input and output argument arrays. Notice also the use of the dim3 derived type, defined in the system cudafor module, to declare the `dimGrid` and `dimBlock` variables; these will be used to define the CUDA grid and thread-block geometry used in the GPU kernel launch.

```
   subroutine mmul( A, B, C )
!
   use cudafor
   real, dimension(:,:) :: A, B, C
   integer :: N, M, L
   real, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev
   type(dim3) :: dimGrid, dimBlock
!
   N = size(A,1) ; M = size(A,2) ; L = size(B,2)
   allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )
   Adev = A(1:N,1:M)
   Bdev = B(1:M,1:L)
   dimGrid = dim3( N/16, L/16, 1 )
   dimBlock = dim3( 16, 16, 1 )
   call mmul_kernel<<<dimGrid,dimBlock>>>( Adev,Bdev,Cdev,N,M,L )
   C(1:N,1:M) = Cdev
   deallocate( Adev, Bdev, Cdev )
!
   end subroutine
```

The routine performs the following operations:

1. Determines the dimensions of the matrices - `N`, `M`, and `L`
2. Allocates device memory arrays `Adev`, `Bdev`, and `Cdev`
3. Copies host arrays `A` and `B` to `Adev` and `Bdev` using array assignments;
4. Initializes `dimGrid` and `dimBlock` with the grid and thread block sizes
5. Launches the `mmul_kernel` GPU kernel to compute `Cdev` on the GPU
6. Copies `Cdev` back from device memory to `C` using an array assignment
7. Frees the device memory arrays

Because the data copy operations are synchronous, no extra synchronization is needed between the copy operations and the kernel launch. This host routine is a little simplistic in assuming `A`, `B` and `C` are all appropriately dimensioned for a legal matrix multiply. Other than that, it will work correctly regardless of the dimensions of the matrix arguments.

The GPU device MMUL_KERNEL subroutine

The GPU device kernel subroutine has six input arguments, the three input/output arrays and their dimensions. The Fortran extension `attributes(global)` is used to specify that this is a kernel to be compiled for execution on the GPU.

The argument arrays `A`, `B`, and `C` are declared with the device attribute, signifying that the actual arguments being passed in are device arrays (it they are not, it is a coding error). Local 16x16 sub-block arrays `Ab` and `Cb` have the shared attribute. This means they will be instantiated in CUDA shared memory rather than in device main memory. CUDA shared memory is a very fast memory local to each GPU multi-processor, which can be accessed and shared by all of the threads in a given thread block. A local `Cij` scalar variable is defined to hold dot product accumulations; this scalar can be placed in a register, rather than computing directly into the corresponding `C(i,j)` element in device main memory.

```
   attributes(global) subroutine MMUL_KERNEL( A,B,C,N,M,L)
!
   real,device :: A(N,M),B(M,L),C(N,L)
   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 kb = 1, M, 16
      ! Fetch one element each into Ab and Bb;  note that 16x16 = 256
      ! threads in this thread-block are fetching separate elements
      ! of Ab and Bb
      Ab(tx,ty) = A(i,kb+ty-1)
      Bb(tx,ty) = B(kb+tx-1,j)
      ! Wait until all elements of Ab and Bb are filled
      call syncthreads()
      do k = 1, 16
         Cij = Cij + Ab(tx,k) * Bb(k,ty)
      enddo
      ! Wait until all threads in the thread-block finish with
      ! this iteration's Ab and Bb
      call syncthreads()
   enddo
   C(i,j) = Cij
!
   end subroutine
```

Recall that when this kernel is launched, it is launched for simultaneous execution on a large number of GPU threads as defined by the CUDA grid and thread-block dimensions in the host caller. A GPU thread executing this routine is one of 16x16 threads cooperating in a thread block. This routine computes the dot product of `A(i,:)*B(:,j)` for a particular value of `i` and `j`, depending on the block and thread index. It performs the following operations:

1. Determines the thread indices for this thread within its thread-block using the pre-defined `threadidx` derived type variable
2. Determines the `i` and `j` indices for the element `C(i,j)` which this thread will compute using the pre-defined blockidx derived type variable
3. Initializes a temporary scalar `Cij` into which this thread will accumulate its dot product
4. Steps through the arrays `A` and `B` in blocks of size 16x16; for each block, it does the following steps:
    a. Loads one element of the submatrices of `A` and `B` into shared memory
    b. Synchronizes to ensure both submatrices are fully loaded by all threads in the block
    c. Accumulates the dot product of its row and column of the submatrices
    d. Synchronizes again to make sure all threads are done reading the submatrices before starting the next block
5. Finally, it stores the computed value into the correct element of `C`

The result is that one thread of execution computes one element `C(i,j)` of the result matrix `C` by doing a dot product of the `i`th row of `A` with the `j`th row of `B`. The thread cooperates with other threads in its thread block to effect high-bandwidth transfers from device main memory into the CUDA shared memory associated with the multi-processor on which the thread-block is executing. Once the `Ab` and `Bb` sub-blocks are loaded, the threads execute in parallel and completely independently to compute a partial dot product.

Note that MMUL_KERNEL is not completely general. It assumes that the input device arrays have dimensions that are valid for a matrix multiply, and that they are multiples of 16. To make this kernel completely general, we would need to insert clean-up code to handle partial blocks.

Using the PGI CUDA Fortran compiler

You can download a version of this matrix multiply example to compile and execute using the PGI 9.0-4 or higher Fortran compiler. Note that PGI CUDA Fortran does not require a new or separate compiler; the new language features and CUDA Fortran programming model are supported by the same PGI Fortran compiler you have probably used previously. It functions either as a standard F95/03 compiler or as a CUDA Fortran compiler, allowing you to incrementally add CUDA Fortran to large existing Fortran programs you already build with the PGI compilers.

When you download the example you will see a single matmul.CUF file. Files with a .cuf extension are treated by the PGI compiler as Fortran source files that may include CUDA Fortran extensions; .CUF files are the same, but are pre-processed. Alternatively, you can use CUDA Fortran extensions in standard .f or .F files, but in that case you must use the `-Mcuda` option on the PGI Fortran command line.

Presuming you have a suitable release of NVIDIA CUDA installed, and have installed the PGI 9.0-4 or higher PGI Fortran 95/03 compiler, compiling and executing on a system with a CUDA-enabled GPU is as simple as:

```
% pgfortran -fast matmul.CUF
% a.out
  arrays sized           512   by          1024   by           512
 calling mmul
 Kernel time excluding data xfer:    3128.000      microseconds
 Megaflops excluding data xfer:      85816.96
 Total time including data xfer:     84462.00      microseconds
 Megaflops including data xfer:      3178.180
 No errors found
%
```

Obviously there is a driver program and some timing calls inserted in the downloadable example, but the source code for both the host subroutine and the kernel are identical to those listed above.

Summary

Together, MMUL and MMUL_KERNEL implement a complete matrix multiplication algorithm in CUDA Fortran. Note that there is only one blocked loop, in the kernel. Normally a simple matrix multiplication involves three nested loops, something like this:

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

In creating the CUDA Fortran implementation, the `i` loop and the `j` loop become implicit in the MMUL_KERNEL launch. Logically, the CUDA Fortran programmer thinks as if `N * L` instances of the inner loop will execute simultaneously. They will in fact be executed by different threads, scheduled in groups (thread-blocks) on each CUDA multi-processor. `N` and `L` are used to calculate the number of thread-blocks to be used, and the number of threads that will exist in each thread-block.

To ensure efficient use of memory bandwidth and CUDA shared memory, the MMUL_KERNEL itself is designed so that threads within an executing thread-block used shared local data structures and perform main memory accesses of data that will be used by the given thread and the other threads in its thread-block. This eliminates redundant accesses to device memory by the threads in a thread-block, and allows the memory accesses to be grouped by the NVIDIA hardware into efficient multi-word transfers from device memory into shared memory.

This basic introduction to PGI CUDA Fortran should give you a flavor of how the programming model works, and how to compile and execute a program. See the CUDA Fortran Programming Guide and Reference for complete details on the CUDA Fortran programming language and PGI CUDA Fortran features and capabilities.