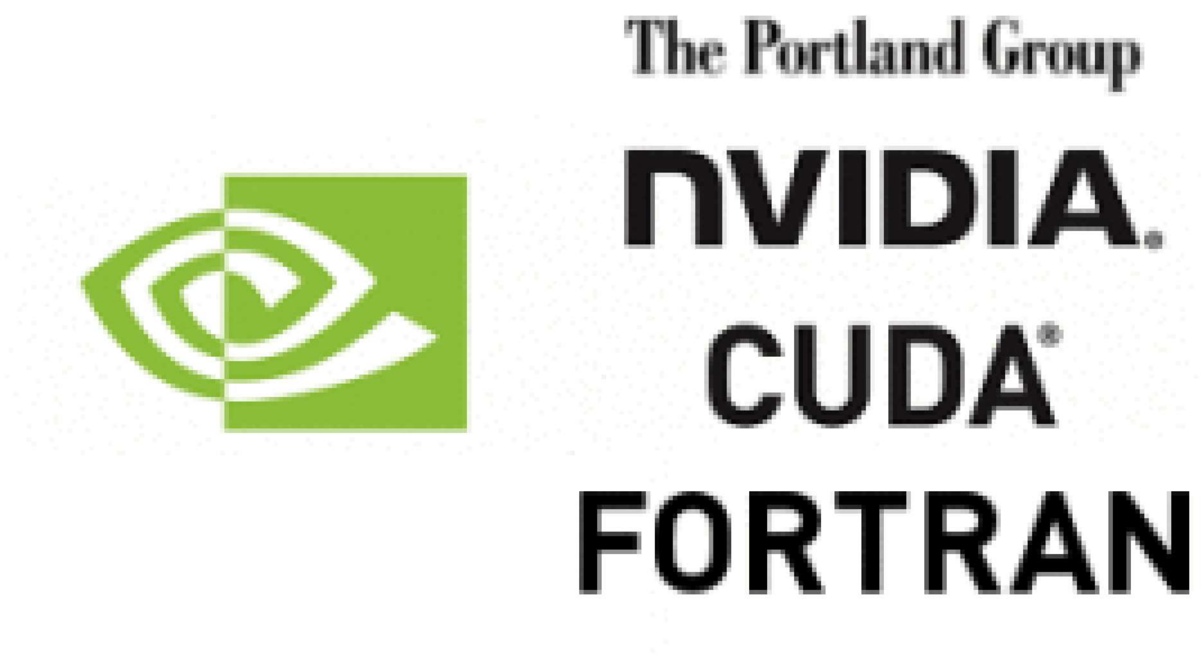
Simulation / Modeling / Design

An Easy Introduction to CUDA Fortran

Oct 29, 2012

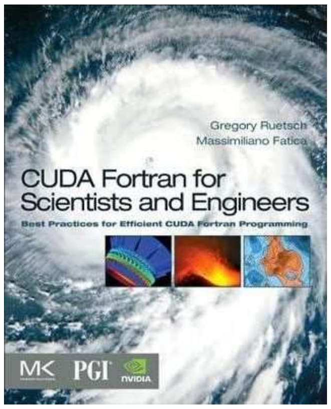
By Greg Ruetsch

+3 Like Discuss (7)



This post is the first in a series on CUDA Fortran, which is the Fortran interface to the CUDA parallel computing platform. If you are familiar with CUDA C, then you are already well on your way to using CUDA Fortran as it is based on the CUDA C runtime API. There are a few differences in how CUDA concepts are expressed using Fortran 90 constructs, but the programming model for both CUDA Fortran and CUDA C is the same.

If you are familiar with Fortran but new to CUDA, this series will cover the basic concepts of parallel computing on the CUDA platform. CUDA Fortran is essentially Fortran with a few extensions that allow one to execute subroutines on the GPU by many threads in parallel.


Gregory Ruetsch
Massimiliano Fatica
CUDA Fortran for Scientists and Engineers
Best Practices for Efficient CUDA Fortran Programming
MK PGI NVIDIA

CUDA Fortran for Scientists and Engineers shows how high-performance application developers can leverage the power of GPUs using Fortran.

CUDA Programming Model Basics

Before we jump into CUDA Fortran code, those new to CUDA will benefit from a basic description of the CUDA programming model and some of the terminology used. (Those familiar with CUDA C or another interface to CUDA can jump to the next section).

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the *host* refers to the CPU and its memory, while the *device* refers to the GPU and its memory. Code running on the host manages the memory on both the host and device, and also launches *kernels* which are subroutines executed on the device. These kernels are executed by many GPU threads in parallel.

Given the heterogeneous nature of the CUDA programming model, a typical sequence of operations for a CUDA Fortran code is:

1. Declare and allocate host and device memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

```
    n = size(x)
    i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
    if (i <= n) y(i) = y(i) + a*x(i)
  end subroutine saxpy
end module mathOps

program testSaxpy
  use mathOps
  use cudafor
  implicit none
  integer, parameter :: N = 40000
  real :: x(N), y(N), a
  real, device :: x_d(N), y_d(N)
  type(dim3) :: grid, tBlock

  tBlock = dim3(256,1,1)
  grid = dim3(ceiling(real(N)/tBlock%x),1,1)

  x = 1.0; y = 2.0; a = 2.0
  x_d = x
  y_d = y
  call saxpy<<<grid, tBlock>>>(x_d, y_d, a)
  y = y_d
  write(*,*) 'Max error: ', maxval(abs(y-4.0))
end program testSaxpy
```

The module `mathOps` above contains the subroutine `saxpy`, which is the kernel that is performed on the GPU, and the program `testSaxpy` is the host code. Let's begin our discussion of this program with the host code.

Host Code

The program `testSaxpy` uses two modules:

```
  use mathOps
  use cudafor
```

The first is the user-defined module `mathOps` which contains the `saxpy` kernel, and the second is the `cudafor` module which contains the CUDA Fortran definitions. In the variable declaration section of the code, two sets of arrays are defined:

```
  real :: x(N), y(N), a
  real, device :: x_d(N), y_d(N)
```

The real arrays `x` and `y` are the *host* arrays, declared in the typical fashion, and the `x_d` and `y_d` arrays are *device* arrays declared with the `device` variable attribute. As with CUDA C, the host and device in CUDA Fortran have separate memory spaces, both of which are managed from host code. But while CUDA C declares variables that reside in device memory in a conventional manner and uses CUDA-specific routines to allocate data on the GPU and transfer data between the CPU and GPU, CUDA Fortran uses the `device` variable attribute to indicate which data reside in device memory and uses conventional means to allocate and transfer data. The arrays `x_d` and `y_d` could have been declared with the `allocatable` in addition to `the device` attribute and allocated with the F90 `allocate` statement.

One consequence of the strong typing in Fortran coupled with the presence of the `device` attribute is that transfers between the host and device can be performed simply by assignment statements. The host-to-device transfers prior to the kernel launch are done by:

```
  x_d = x
  y_d = y
```

while the device-to-host transfer of the result is done by:

```
  y = y_d
```

The `saxpy` kernel is launched by the statement:

```
  call saxpy<<<grid,tBlock>>>(x_d, y_d, a)
```

The information between the triple chevrons is the *execution configuration*, which dictates how many device threads execute the kernel in parallel. In CUDA there is a hierarchy of threads in software which mimics how thread processors are grouped on the GPU. In CUDA we speak of launching a kernel with a *grid* of *thread blocks*. The second argument in the execution configuration specifies the number of threads in a thread block, and the first specifies the number of thread blocks in the grid. Threads in a thread block can be arranged in a multidimensional manner to accommodate the multidimensional nature of the underlying problem, and likewise thread blocks can be arranged as such in a grid. It is for this reason that the derived type `dim3`, which contains *x*, *y*, and *z* components, is used for these two execution configuration parameters. In a one-dimensional case such as this, these two execution configuration parameters could also have been specified as integers. In this case we launch the kernel with thread blocks containing 256 threads, and use the ceiling function to determine the number of thread blocks required to process all `N` elements of the arrays:

```
  tBlock = dim3(256,1,1)
  grid = dim3(ceiling(real(N)/tBlock%x),1,1)
```

For cases where the number of elements in the arrays is not evenly divisible by the thread block size, the kernel code must check for out-of-bounds memory accesses.

```
    n = size(x)
    i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
    if (i <= n) y(i) = y(i) + a*x(i)
  end subroutine saxpy
```

The `saxpy` kernel is differentiated from host subroutines via the `attributes(global)` qualifier. In the variable declarations, the `device` attribute is not needed as it is assumed that in device code all arguments reside on the device. This is the case for the first two arguments of this kernel, which correspond to the device arrays `x_d` and `y_d` in host code. However, the last argument, the parameter `a`, was not transferred to the device in host code. Because Fortran passes arguments by reference rather than value, accessing a host variable from the device would cause an error unless the `value` variable attribute is used in such cases, which instructs the compiler to pass such arguments by value.

After the variable declarations, there are only three lines in our `saxpy` kernel. As mentioned earlier, the kernel is executed by multiple threads in parallel. If we want each thread to process an element of the resultant array, then we need a means of distinguishing and identifying each thread. This is accomplished through the predefined variables `blockDim`, `blockIdx`, and `threadIdx`. These predefined variables are of type `dim3`, and are analogous to the execution configuration parameters in host code. The predefined variable `blockDim` in the kernel is equivalent to the thread block specified in host code by the second execution configuration parameter. The predefined variables `threadIdx` and `blockIdx` give the identity of the thread within the thread block and the thread block within the grid, respectively. The expression:

```
    i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
```

generates a global index that is used to access elements of the arrays. Note that in CUDA Fortran, the components of `threadIdx` and `blockIdx` have unit offset, so the first thread in a block has `threadIdx%x=1` and the first block in the grid has `blockIdx%x=1`. This differs from CUDA C which has zero offset for these built-in variables, where the equivalent expression for an index used to access C arrays would be:

```
    i = blockDim.x*blockIdx.x + threadIdx.x;
```

Before this index is used to access array elements, its value is checked against the number of elements, obtained from the `size()` intrinsic, to ensure there are no out-of-bounds memory accesses. This check is required for cases where the number of elements in an array is not evenly divisible by the thread block size, and as a result the number of threads launched by the kernel is larger than the array size.

Compiling and Running the Code

The CUDA Fortran compiler is a part of the PGI compilers which can be downloaded from PGI's web site, which offers a free 15-day trial license.

Any code in a file with a `.cuf` or `.CUF` extension is compiled with CUDA Fortran automatically enabled. Otherwise, the compiler flag `-Mcuda` can be used to compile CUDA Fortran code with other extensions. If this code is in a file named `saxpy.cuf`, compilation and execution of the code is as simple as:

```
% pgf90 -o saxpy saxpy.cuf
% ./saxpy
 Max error:  0.000000
```

Summary and Conclusions

Through a discussion of a CUDA Fortran implementation of SAXPY, this post explained the basic components of programming CUDA Fortran. Looking back at the full code, there are only a few extensions to Fortran required to “port” a Fortran code to CUDA Fortran: variable and function attributes used to distinguish device from host counterparts, the execution configuration when launching a kernel, and the built-in device variables used to identify and differentiate GPU threads that execute the kernel in parallel.

One advantage of having a heterogeneous programming model is that porting an existing code from Fortran to CUDA Fortran can be done incrementally, one kernel at a time. Contrast this to other parallel programming approaches, such as MPI, where porting is an all-or-nothing endeavor.

In the next post of this series, we will look at some performance measurements and metrics.

Related resources

- **DLI course:** An Even Easier Introduction to CUDA
- **GTC session:** Mastering CUDA C++: Modern Best Practices with the CUDA C++ Core Libraries
- **GTC session:** Introduction to CUDA Programming and Performance Optimization
- **GTC session:** How To Write A CUDA Program: The Ninja Edition
- **NGC Containers:** CUDA
- **SDK:** CUDA MATH API

Discuss (7)   +3 Like

Tags

Simulation / Modeling / Design | HPC / Scientific Computing | CUDA | Beginner | CUDA Fortran

View all posts by Greg Ruetsch >

Comments

Notable Replies

July 13, 2014

anon61998926

Hi dear all
I am a beginner to CUDA (CUDA FORTRAN) . I have installed PGI 13.9 but the problem is, when I try to debug even a very simple CUDA Fortran code I get plenty of errors such as below:
Error 1 unresolved external symbol cudaSetupArgument referenced in function mathops_saxpy_ saxp.obj
Error 2 unresolved external symbol cudaLaunch referenced in function mathops_saxpy_ saxp.obj
Error 3 unresolved external symbol __cudaRegisterFatBinary referenced in function mathops_saxpy_ saxp.obj
Error 4 unresolved external symbol __cudaRegisterFunction referenced in function mathops_saxpy_ saxp.obj
Error 5 unresolved external symbol __cudaUnregisterFatBinary referenced in function mathops_saxpy_ saxp.obj
Error 6 unresolved external symbol pgf90_dev_auto_alloc04 referenced in function MAIN_ saxp.obj
Error 7 unresolved external symbol pgf90_dev_copyin referenced in function MAIN_ saxp.obj
.
.
.
.Error 12 unresolved external symbol CUDAFOR saxp.obj
Furthermore, when I check the project properties I see that the CUDA FOR is not even enabled when I enable it and debug again I get the same errors. I will be appreciated if someone help me with this problem.
Thanks,
Reza

July 15, 2014

anon27139054

Hi Reza:
I sounds like you are using the free version of PVF where CUDA Fortran is not enabled. If so, information on how to enable CUDA Fortran can be found at https://www.pgroup.com/prod....

April 28, 2017

anon79155597

Hi,
i tried to run this script and it returned 'Max error: 2.0000'.
Where this error come from?
from the cudaDeviceProp in Fortran CUDA i got
Device Number: 0
Device name: GeForce GTX 1060 3GB
Memory Clock Rate (KHz): 4004000
Memory Bus Width (bits): 192
Peak Memory Bandwidth (GB/s): 192.19

April 28, 2017

anon79155597

this is work when I use pgf90 -Mcuda=cc60 -o saxpy saxpy.cuf to compile

September 26, 2017

anon42684293

Thanks! I have exact same problem with you. And it can be solved by your solution.

Technical Blog

Subscribe >

2 more replies

Participants