Guidelines for CUDA Fortran

In this document, we are going to cover the very basic CUDA Fortran concepts in comparison with C. We ask you to follow the CUDA Laboratory 1 description in C and to use this document to understand what would be your changes in Fortran.

Compiling a CUDA Fortran Program

The compilation is very similar to CUDA C, but with slight variations. First, you need to load not only the CUDA module, but also the PGI compiler:

```
module load cuda/7.0 pgi
```

To compile a CUDA Fortran program, use `pgfortran` and include the architecture (i.e., `cc3x`):

```
pgfortran -Mcuda=cc3x your_cuda_file.cuf -o your_cuda_file.out
```

You can run a program as in the CUDA C version, allocating a node first with `salloc` and then running the code with `srun`:

```
srun -n 1 ./your_cuda_file.out
```

Kernel Management

The concept of `grid` and `block` is the same as in CUDA C. In this case, you need to declare both variables as `type(dim3)`. This is an example with a grid of 1 block of 32 threads in X:

```
type(dim3) :: grid
type(dim3) :: block
grid  = dim3(1, 1, 1)
block = dim3(32, 1, 1)
```

To launch CUDA Fortran kernel, use the same syntax as in CUDA C with the triple-brackets:

```
call your_kernel<<<grid, block>>>( ... )
```

Memory Management

To allocate memory on the GPU and release it afterwards, use the `cudaMalloc()` and the `cudaFree()` functions. You need to declare the variable with the `device` attribute:

```
real, allocatable, device :: d_x(:)
hr = cudaMalloc(d_x, 256)
```

Here, we declare an array `d_x` of type real, `allocatable` and to be used on the GPU. Then, we use `cudaMalloc()` to define the size with 256 elements. We have captured the status result of the operation in an integer `hr`, in case we would like to check if there were any errors.

To copy memory from the host to the GPU (or viceversa), use the `cudaMemcpy()` function:

```
hr = cudaMemcpy(d_x, x, ARRAY_SIZE) ! Copy the content from x to d_x
```

Compared to the CUDA C version, the main difference is that we no longer have to specify the direction of the copy. In this case, we are copying from x (on the CPU) to d_x (on the GPU). But we could revert the direction by simply swapping the variables:

```
hr = cudaMemcpy(x, d_x, ARRAY_SIZE) ! Copy the content from d_x to x
```

Kernel Implementation

The CUDA Fortran kernels are once again very similar to their CUDA C counterpart. In this case, you need to declare a new subroutine with the `global` attribute, such as:

```
attributes(global) subroutine your_kernel(n, d_x)
...
end subroutine your_kernel
```

The type definition of the constant arguments, such as “n” in the previous example, must contain the attribute `value`:

```
integer, value :: n
```

We also recommend you to specify the `intent` of the input parameter. In the case of `d_x`, we could declare it as (note that, inside the kernel, we do not specify the `device` attribute):

```
real, intent(inout) :: d_x(:)
```

Finally, the predefined constants `gridDim`, `blockDim`, `blockIdx` and `threadIdx`, are all available inside the CUDA Fortran kernel:

```
integer :: tid
tid = (blockIdx%x - 1) * blockDim%x + threadIdx%x
```