building-gpu-code

March 16, 2022

1 Building and Monitoring GPU Programs

1.1 NCAR GPU Workshop Lab

Presenter: Brian Vanderwende
Date: March 17, 2022

1.2 Configuring your environment

The default compute environment on Casper provides the Intel compilers for CPU workflows. We want to use the NVIDIA HPC SDK, and so we need to switch our environment configuration. On most supercomputers, you will use environment modules to do so.

[1]:
```
# See the available versions of the NVIDIA HPC SDK
module avail nvhpc
```

```
--------------- /glade/u/apps/dav/modulefiles/default/compilers ----------------
   nvhpc/20.9     nvhpc/21.3    nvhpc/21.9  (D)    nvhpc/22.1
   nvhpc/20.11    nvhpc/21.7    nvhpc/21.11        nvhpc/22.2

  Where:
   D:  Default Module

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 …" to search for all possible modules matching
any of the "keys".
```

[2]:
```
# Remove any loaded modules and load the latest NVIDIA HPC SDK
module purge
module load nvhpc/22.2
module list
```

```
Currently Loaded Modules:
```

```
1) nvhpc/22.2
```

1.3 Compiling a basic OpenACC Fortran code

For this demonstration, we will use a basic Fortran code from the set of OpenACC examples provided by the NVIDIA HPC SDK. We need to make a copy of the source file at a location in which we have write permissions.

[3]:
```
# Prepare a directory to contain the case
mkdir -p openacc_f1
cd openacc_f1
cp $NVHPC/Linux_x86_64/22.2/examples/OpenACC/samples/acc_f1/acc_f1.f90 .
ls
```

```
acc_f1.f90
```

This code contains a few OpenACC directives to offload scalar multiplication operations to a GPU.

The details of OpenACC programming will be taught in future sessions - for now we will only focus on how to compile the code.

As this is a Fortran code, we will use the `nvfortran` compiler. The `-acc` flag must be given to `nvfortran` in order to enable OpenACC directives (and the same to nvc++ for C++ pragmas). Without this flag, only CPU instructions will be generated.

[4]:
```
# Compile the fortran code and output into a binary called acc_f1
nvfortran -o acc_f1 -acc acc_f1.f90
ls
```

```
acc_f1  acc_f1.f90
```

We can verify that OpenACC was used in a number of ways - here via the `strings` utility, which can be used to extract human-readible text strings from binary files. We search the `strings` output using `grep`, and instruct it to only report the first match with `-m1`.

[5]:
```
# Use strings to look for "libacc" OpenACC libraries in our binary
echo "OpenACC libraries:"
strings acc_f1 | grep -m1 libacc
```

```
OpenACC libraries:
libacchost.so
```

The above output indicates that OpenACC libraries have been used by `nvfortran` when compiling our binary. Meanwhile, if we compile without OpenACC support, we should see that grep returns no match.

[6]:
```
nvfortran -o no_acc_f1 acc_f1.f90
echo "OpenACC libraries:"
strings no_acc_f1 | grep -m1 libacc || echo "None found"
```

```
OpenACC libraries:
None found
```

1.3.1 Compiling for OpenMP GPU offload

Many of the concepts shown here extend to compiling OpenMP GPU code as well. However, the flags for activating GPU offload are slightly different:

```
nvfortran -o omp_gpu -mp=gpu omp.f90
```

1.4 Getting acceleration information from the compiler

The NVIDIA compilers themselves provide diagnostic options - the like the powerful flag `-Minfo` - which allow us to learn about compile-time decisions including GPU offloading. The `accel` argument to `-Minfo` will give us information specifically pertaining to OpenACC (*or OpenMP*) GPU acceleration at compile time.

[7]:
```
nvfortran -o acc_f1 -acc -Minfo=accel acc_f1.f90
```

```
main:
     28, Generating implicit copyin(a(1:n)) [if not already present]
         Generating implicit copyout(r(1:n)) [if not already present]
     29, Loop is parallelizable
         Generating NVIDIA GPU code
         29, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

Alternatively, you can specify `-Minfo` by itself to get all available information about compile-time decisions. Some of the information includes: * accel - information about accelerator region targeting * loop - information about loop optimizations * par - information about loop parallelization * vect - information about automatic loop vectorization

Note that using `-Minfo` without any arguments will produce both CPU and GPU diagnostic information!

1.5 Customizing target offload capabilities

New GPU generations almost always provide new features and capabilities. The NVIDIA compilers allow you to generate code for one or more specific GPU *compute capabilities*. For example, GPUs at NCAR fall into three capabilities: * Quadro GP100 - cc60 * Volta V100 - cc70 * Ampere A100 - cc80

If you include more compute capabilities when compiling, the compile time and size of your binary file will grow, but you will have an executable that better matches the optimizations of each target GPU. All GPU compute capabilites are provided at https://developer.nvidia.com/cuda-gpus

[8]:
```
# Here, we can compile our binary for GP100 and V100 execution
echo "Compile time for cc60, cc70:"
time nvfortran -o cc_ncar -acc -gpu=cc60,cc70 acc_f1.f90
```

```
Compile time for cc60, cc70:

real    0m1.407s
user    0m1.199s
sys     0m0.141s
```

[9]:
```
# We can also compile for all available compute capabilities
echo "Compile time for ccall:"
time nvfortran -o cc_all -acc -gpu=ccall acc_f1.f90
```

```
Compile time for ccall:

real    0m5.203s
user    0m4.487s
sys     0m0.417s
```

[10]:
```
# Let's compare the sizes of each
echo -e "File sizes:\n"
ls -l -h cc_*
```

```
File sizes:

-rwxr-xr-x 1 vanderwb csgteam 84K Mar 16 21:54 cc_all
-rwxr-xr-x 1 vanderwb csgteam 37K Mar 16 21:53 cc_ncar
```

*The default compute capability depends on whether you are compiling on a system with a detectable GPU: * If a GPU is found (e.g., Casper's GPU nodes), that GPU's compute capability will be used * If no GPU is found (e.g., Casper's login nodes), the binary will be compiled with* `-gpu=ccall`

1.5.1 More details about compiler options

As with many Linux programs, one of the best ways to learn about all of the features and configuration options of each compiler is to examine its *man* (manual) page.

```
man nvfortran
```

For example, here is an excerpt from the man page entry describing the -acc flag to nvfortran:

```
Target-specific Options
       -acc   Enable OpenACC pragmas and directives to explicitly
              parallelize regions of code for execution by accelerator
              devices.  See the -gpu flag to select options specific to
              NVIDIA GPUs.  The options are:

              autopar (default) noautopar
```

```
                Enable loop autoparallelization within parallel
                constructs.

        gpu (default)
                Compile OpenACC directives for parallel execution
                on the GPU.

        host    Compile OpenACC directives for serial execution
                on the host CPU.
        ...
```

1.6 Compiling an CUDA Fortran program

Next, let's shift from building a code with OpenACC offloading to a Fortran program written with CUDA instructions. Again, we will forgo analysis of the code itself and simply focus on compilation tasks. This program utilizes accelerated CUDA FFT routines, which we must link to via arguments to the compiler.

The program consists of three *.cuf* source files. First, let's copy the source files from the NVIDIA HPC SDK examples directory to our own working space.

[11]:
```
# Make a directory for the source files and copy from the examples
mkdir -p ../cuf_fft
cd ../cuf_fft
cp $NVHPC/Linux_x86_64/22.2/examples/CUDA-Fortran/SDK/cufftTest/*.cuf .
ls
```

```
cufftTest.cuf  cufft_m.cuf  precision_m.cuf
```

1.6.1 Using Makefiles

These three files will need to be compiled and then linked into a binary. We could do this interactively on the command line. We could also write a shell script to contain these commands. However, the standard way to build many open source applications in a Linux environment is to use a *Makefile*.

A *Makefile* simply defines a set of targets (rules) which are then interpreted by the `make` program to execute commands.

Before creating our compilation rules, it is helpful to define settings - in the form of variables - which can then be used by the rules to affect compiler and linker behavior. Keep in mind that while some syntax may appear similar, variable definitions (and other code structures) differ in Makefiles from that of your shell (e.g., bash/tcsh).

[12]:
```
cat > Makefile << 'EOF'
# Specify the Fortran compiler
# The ?= syntax tells Make to only set if currently undefined
FC ?= nvfortran
```

```
# Define some compiler flags
# -fast -> let the compiler choose ideal optimizations for the target platform
# -Mpreprocess -> force the compiler to preprocess specified files instead of␣
 ↪guessing from extension
FCFLAGS = -fast -Mpreprocess

# Tell the compiler to link to the cuFFT library
CULIBS = -cudalib=cufft

EOF
```

Now we can define our make rules, and use variables and shortcuts to generalize them. These generalizations can prove very powerful in more complex Makefiles.

[13]:
```
cat >> Makefile << 'EOF'
# Define another variable that lists all source files
SRCS = precision_m.cuf cufft_m.cuf cufftTest.cuf

# String replacement on SRCS to get list of object files (e.g., cufftTest.o)
OBJS := $(SRCS:.cuf=.o)

# target: prerequisites
build: $(OBJS)
    $(FC) -o cuFFTTest $(OBJS) $(CULIBS)

# Fancy rule to generalize (%) to any .o file
# $< variable references the specific prerequisite file
%.o: %.cuf
    $(FC) $(FCFLAGS) -c $<

clean:
    rm -f $(OBJS) *.mod cuFFTTest
EOF
```

JupyterLab replaces tabs with spaces, but Make requires tab indentation (Make, like Python, is very picky about white-space). The following command replaces spaces at the beginning of lines with a tab. In normal editing, this step should not be necessary!

[14]:
```
sed -i 's/^ \+/\t/g' Makefile
```

[15]:
```
# Finally, let's run our Makefile. By not specifying a target, we implicitly␣
 ↪choose the first rule (build)
make
```

```
nvfortran -fast -Mpreprocess -c precision_m.cuf
nvfortran -fast -Mpreprocess -c cufft_m.cuf
nvfortran -fast -Mpreprocess -c cufftTest.cuf
```

```
nvfortran -o cuFFTTest precision_m.o cufft_m.o cufftTest.o -cudalib=cufft
```

[16]:
```
ls -l
```

```
total 19
-rw-r--r-- 1 vanderwb csgteam   884 Mar 16 21:54 Makefile
-rwxr-xr-x 1 vanderwb csgteam 22664 Mar 16 21:54 cuFFTTest
-rwxr-xr-x 1 vanderwb csgteam  2188 Mar 16 21:54 cufftTest.cuf
-rw-r--r-- 1 vanderwb csgteam 13672 Mar 16 21:54 cufftTest.o
-rwxr-xr-x 1 vanderwb csgteam  7920 Mar 16 21:54 cufft_m.cuf
-rw-r--r-- 1 vanderwb csgteam 15154 Mar 16 21:54 cufft_m.mod
-rw-r--r-- 1 vanderwb csgteam 20560 Mar 16 21:54 cufft_m.o
-rwxr-xr-x 1 vanderwb csgteam   891 Mar 16 21:54 precision_m.cuf
-rw-r--r-- 1 vanderwb csgteam   871 Mar 16 21:54 precision_m.mod
-rw-r--r-- 1 vanderwb csgteam  1648 Mar 16 21:54 precision_m.o
```

1.7 Monitoring your GPU application

Typical Linux utilities like `top` and `ps` will give you a CPU-centric view of what is running on the node. NVIDIA provides additional utilities to monitor GPU usage. One of the most basic, though powerful, tools is `nvidia-smi`.

`nvidia-smi` has multiple modes of operation, detailed in depth in its *man* page. The following cells demonstrate the default output, the *device monitoring* mode, and the *process monitoring* mode.

[17]:
```
# By default, nvidia-smi will provide an overview of the GPU states and running␣
↪processes
nvidia-smi
```

```
Wed Mar 16 21:54:59 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.57.02    Driver Version: 470.57.02    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Quadro GP100        On   | 00000000:18:00.0 Off |                  Off |
| 61%   82C    P0   182W / 235W |   8459MiB / 16278MiB |    100%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      5157      G   /usr/bin/X                        216MiB |
```

```
|    0   N/A  N/A      5845      G   /usr/bin/gnome-shell                  7MiB |
|    0   N/A  N/A     83097      G   /usr/bin/gnome-shell                 75MiB |
|    0   N/A  N/A    220753      C   ./gpu_burn                         8155MiB |
+-----------------------------------------------------------------------------+
```

Alternatively, we can use device monitoring to log a particular GPU's state over time:

[18]:
```
# Display a single "dmon" instance from GPU ID 0 with Time labels
nvidia-smi dmon -c 1 -o T -i 0
```

```
#Time        gpu    pwr gtemp mtemp    sm    mem    enc    dec  mclk  pclk
#HH:MM:SS    Idx      W     C     C     %     %      %      %   MHz   MHz
 21:55:02      0    180    82     -   100     9      0      0   715  1366
```

Similarly, a list of processes running on one or more GPUs can be monitored over time:

[19]:
```
nvidia-smi pmon -c 1 -o T -i 0
```

```
#Time        gpu        pid  type    sm   mem   enc   dec   command
#HH:MM:SS    Idx          #   C/G     %     %     %     %   name
 21:55:05      0       5157     G     -     -     -     -   X
 21:55:05      0       5845     G     -     -     -     -   gnome-shell
 21:55:05      0      83097     G     -     -     -     -   gnome-shell
 21:55:05      0     220753     C    99     9     -     -   gpu_burn
```

Finally, we can correlate GPU information to CPU tasks via the process ID (pid):

[20]:
```
# Get the process ID of the first listed GPU process and store in a bash␣
 ↪variable
GPUPID=$(nvidia-smi pmon -c 1 | awk 'FNR==3 {print $2}')

# Next, look up the process using the standard Linux utility "ps"
ps -o pid,uid,user,cmd,%mem,%cpu -p $GPUPID
```

```
  PID   UID USER     CMD                         %MEM %CPU
 5157     0 root     /usr/bin/X :0 -background n  0.0  1.0
```

You can perform live interrogation of running GPU code using `nvidia-smi` even in a batch job. First, identify the node(s) on which the job is running, and then `ssh` to that compute node from a Casper login node. Note that you may only `ssh` to a compute node if you have a job currently running on that node.

```
casper-login1$ qstat -n <jobid>
casper-login1$ ssh <compute-node>
compute-node$ nvidia-smi
```

You should now be able to compile basic OpenACC, OpenMP GPU, and CUDA Fortran codes both on the command line and using a Makefile, and get basic diagnostic information for running GPU-enabled programs. In the next couple of workshop sessions, we will begin the coding portion of the series with an overview of using OpenACC.