Using CUDA on Crescent

Logging in and environment To run the exercises, you'll need to log into the `Crescent2` systems, with a command like

```
ssh -X s12345@crescent2.central.cranfield.ac.uk
```

where you need to substitute your account in place of `s12345`.

For additional information, please go to the Intranet and search for Services/IT Services/Research Computing/User Guides/Crescent user guide.

To use CUDA you have to set up the appropriate environment variables with the commands

```
module use /apps/modules/all/
module load GCC/11.3.0 CUDA/11.7.0 CMake/3.24.3-GCCcore-11.3.0
export CC=$(which gcc)
export CXX=$(which g++)
```

These commands must be issued every time you log into the system (i.e. for every session).

Compilation You can then unzip the exercises file which will produce a directory called `exercises`. Go to to this directory.

You now have to download the CUDA samples:

- `wget https://github.com/NVIDIA/cuda-samples/archive/v11.6.tar.gz`;
- `tar xzf v11.6.tar.gz`.

Then you will have to prepare for the build process with

```
cmake .
```

You can now go to the `introduction` directory and compile the first example, taken from the CUDA SDK samples, with:

```
make deviceQuery
```

As you progress through the exercises, you may need to add new targets to the makefile; to do this you have to edit the file

```
CMakeLists.txt
```

and uncomment the appropriate lines (i.e. take out the initial `#` character); then you have to re-run `cmake` to regenerate the Makefiles.

If you want to build your own makefile, the most important options of the `nvcc` compiler you should be aware of are the following:

1. `-g -G` Generate debug info for host code `-g` or device code `-G`; `-lineinfo` also adds line number info to device code;

2. `-O -O3` optimize host code;

3. `--x {c|c++|cu}` Specify explicitly the input language, overriding the file suffix;

4. `-gencode arch=XXX,code=YYY` Control code generation behaviour; for example, `-gencode arch=compute_70,code=sm_70` generates code suitable for a device with compute capability 7.0. You can specify multiple instances, and the compiler will generate multiple code paths, which will be chosen at runtime according to the executing device.

For full details please consult the compiler documentation; all documentation about the CUDA SDK is available from `http://docs.nvidia.com/cuda/`

Job submission Copy the file `exercises/cuda.sub` into the `introduction`, go to the `introduction` subdirectory, change the email address at line 32 to match your Cranfield email; then submit a batch job with the command

```
qsub -V cuda.sub
```

If everything has been set up properly, the system will respond with a message like

```
35291.mgmt01
```

(with a new number assigned to each invocation), and at the end of the run you will have a file called `cudatest.o35291` which will contain something like:

```
CUDA_VISIBLE_DEVICES=GPU-2b6c369f-e343-b816-fe19-1200242394f1
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla V100-PCIE-32GB"
  CUDA Driver Version / Runtime Version          12.1 / 11.7
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 32501 MBytes (34079899648 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1380 MHz (1.38 GHz)
  Memory Clock rate:                             877 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(163
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
```

```
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 7 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 9 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneous

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.1, CUDA Runtime Version = 11
Result = PASS
```

You have successfully run your first CUDA application.

You will now need to edit the other exercise source files filling in the missing part to get them to compile and execute. Don't forget to adapt `cuda.sub` script, e.g. updating the name of the executable program. The script uses the `gpu_V100` queue.

Profiling The Cuda Toolkit provides a nice tool for analyzing performance, the Visual Profiler; by default, however, it assumes to be running on the same computer as the xomputational experiments take place, and this is clearly not the case with the lab setup.

To profile your applications you will then need to split the process into two steps:

1. Collect the profile statistics (on Delta);
2. Visualize the profiles (on the local machine).

To collect the profile statistics, you will need to run your CUDA application under the NVIDIA profiler; you will need to do it twice, once to get the timeline, and once (if desired) to collect the detailed per-kernel statistics. To do so, you will need to substitute the run command in `cuda.sub`, e.g.

```
./MyApplication my_command_args
```

with the following commands:

```
nvprof -f --export-profile MyApplication_timeline.nvprof \
 ./MyApplication my_command_args
nvprof -f --metrics sm_efficiency,gld_throughput,gld_requested_throughput,\
  shared_efficiency,gld_efficiency,warp_execution_efficiency,flop_count_dp,\
  flop_dp_efficiency,achieved_occupancy,inst_per_warp \
  -o MyApplication_metrics.nvprof ./MyApplication my_command_args
```

The metrics listed above give fairly complete information for most applications. To get a complete list of available metrics use the command

```
nvprof --query-metrics
```

or consult the help with

```
nvprof --help
```

After having run the application with the profiler, you can download the timeline and metrics files onto the local computer, and examine them with the `nvvp` application by using the `Import` dialog.