Your first CUDA program

Rupert Nash, Kevin Stratford, Alan Gray

Introduction

Credits

Purpose

In this lab, you will learn how to adapt a simple code such that it uses the GPU.

It has the purpose of negating an array of integers. We introduce the important concepts of device-memory management and kernel invocation. The final version should copy an array of integers from the host to device, multiply each element by -1 on the device, and then copy the array back to the host.

Choose the C or Fortran version.

Source code

You can get this from GitHub:

```
git clone https://github.com/EPCCed/APT-CUDA.git
cd APT-CUDA/exercises/cuda-intro/
```

Recall that on Cirrus, you need to use the `/work` filesystem for files that need to be accessed from compute node, such as executables and data files. Therefore I suggest you clone this to your directory under `/work`.

Note

The template source file is clearly marked with the sections to be edited, e.g.

```
/* Part 1A: allocate device memory */
```

Where necessary, you should refer to the CUDA C Programming Guide and Reference Manual documents available from http://developer.nvidia.com/nvidia-gpu-computing-documentation.

Part 1

Copying Between Host and Device

C

Start from the `intro.cu` template.

1A

Allocate memory for the array on the device: use the existing pointer `d_a` and the variable `sz` (which has already been assigned the size of the array in bytes).

1B

Copy the array `h_a` on the host to `d_a` on the device.

1C

Copy `d_a` on the device back to `h_out` on the host.

1D

Free `d_a`.

Fortran

Start from the intro.cuf template.

1A

Allocate memory for the array on the device: use the existing pointer `d_a` and `ARRAY_SIZE` (which has already been assigned the size of the array in elements)

1B

Copy the array `h_a` on the host to `d_a` on the device, using an appropriate assignment operation.

1C

Copy `d_a` on the device back to `h_out` on the host, using another assignment operation.

1D

Deallocate `d_a`.

Compilation

Load modules

First, we need to load a number of modules to allow compilation.

```
module load gcc nvidia/nvhpc/22.11
```

Use make

Compile the code using `make`. Note that the compute capability of the CUDA device is specified with the `-arch` flag for C and with `-March=` for Fortran.

Running

On Cirrus

You can only run on the backend nodes, so must submit the job to the batch system. To do this, you need to know your budget code - you can check by logging into SAFE, navigating to the relevant Cirrus login account and checking which budgets it can access.

Submit the job with

```
sbatch --account <YOUR BUDGET CODE> submit.sh
```

During on campus tutorials we have reserved one node (4 GPUs) for the use of the class. You can access this by editing the SLURM script or adding extra options to the `sbatch` command:

```
sbatch --account m22oc --qos=reservation
--reservation=<reservation ID>
```

(The reservation ID will be given on the day)

The output (the contents of the `h_out` array) or any error messages will be printed. So far the code simply copies from `h_a` on the host to `d_a` on the device,

then copies `d_a` back to `h_out`, so the output should be the initial content of `h_a` - the numbers 0 to 255.

Part 2

Launching Kernels

Now we will actually run a kernel on the GPU device.

C

2A

Configure and launch the kernel using a 1D grid and a single thread block (`NUM_BLOCKS` and `THREADS_PER_BLOCK` are already defined for this case).

2B

Implement the actual kernel function to negate an array element as follows:

```
int idx = threadIdx.x;
d_a[idx] = -1 * d_a[idx];
```

Compile and run the code as before. This time the output should contain the result of negating each element of the input array. Because the array is initialised to the numbers 0 to 255, you should see the numbers 0 down to -255 printed.

This kernel works, but since it only uses one thread block, it will only be utilising one of the multiple SMs available on the GPU. Multiple thread blocks are needed to fully utilize the available resources.

2C

Implement the kernel again, this time allowing multiple thread blocks. It will be very similar to the previous kernel implementation except that the array index will be computed differently:

```
int idx = threadIdx.x + (blockIdx.x * blockDim.x);
```

Remember to also change the kernel invocation to invoke `negate_multiblock` this time. With this version you can change `NUM_BLOCKS` and `THREADS_PER_BLOCK` to have different values - so long as they still multiply to give the array size.

Fortran

2A

Configure and launch the kernel using a 1D grid and a single thread block (`NUM_BLOCKS` and `THREADS_PER_BLOCK` are already defined for this case).

2B

Implement the actual kernel function to negate an array element as follows:

```
integer :: idx
idx = threadidx%x
aa(idx) = -1*aa(idx)
```

Compile and run the code as before. This time the output should contain the result of negating each element of the input array. Because the array is initialised to the numbers 0 to 255, you should see the numbers 0 down to -255 printed.

This kernel works, but since it only uses one thread block, it will only be utilising one of the multiple SMs available on the GPU. Multiple thread blocks are needed to fully utilize the available resources.

2C

Implement the kernel again, this time allowing multiple thread blocks. It will be very similar to the previous kernel implementation except that the array index will be computed differently:

```
idx = threadidx%x + ((blockidx%x-1) * blockdim%x)
```

Remember to also change the kernel invocation to invoke g_negate_multiblock this time. With this version you can change `NUM_BLOCKS` and `THREADS_PER_BLOCK` to have different values - so long as they still multiply to give the array size.

Part 3

Handling any size of array

Currently we are insisting that the array size be an exact multiple of the block size. In general we should handle any size that will fit in GPU memory.

Let the total number of elements be N and the block size be B.

Recall that in integer division we discard the fractional part so we can write:

$$N = k * B + r$$

i.e. N can divided into k (an integer) number of blocks, plus a remainder, r. If r is zero, then we need k blocks, or else we need $k + 1$.

This can be expressed in a simple formula:

$$nBlocks = \frac{N - 1}{B} + 1$$

Convince yourself this is correct.

3A

Update the kernel launch code to compute the number of blocks using this formula.

What will happen in the last block with the current kernel?

3B

Implement a condition in the kernel to protect against any problem which may arise.

Try changing `ARRAY_SIZE` to a non-multiple of 256 (e.g. 500).