|epcc|

HPC Architectures: Accelerated Sharpen

David Henty

1 Introduction

The purpose of this practical session is to investigate running programs on the GPU nodes of Cirrus. You will be given a GPU-enabled version of the parallel CPU sharpen program that you ran at the start of the course. You can now compare the performance of the new GPU code to the old CPU version. If you want to find out the details of the hardware on Cirrus (Intel CPUs and NVIDIA GPUs) then see `https://www.cirrus.ac.uk/about/hardware.html`

2 Compiling and Running

First clone the repository:

```
git clone https://github.com/davidhenty/sharpen.git
```

(if you have already cloned sharpen you should be able to update using `git pull` within the repository).

3 CPU Performance

We will first run the CPU version parallelised using OpenMP. To make a fair comparison we will run on the host CPUs on the GPU nodes as these are more modern than the CPUs in the standard compute nodes. Note that these newer CPUs have 40 CPU-cores rather than 36.

Compile and run as follows:

```
module load gcc
module load nvidia/nvhpc
module load intel-20.4/compilers
cd sharpen/src/C-OMP
make
sbatch sharpengpuhost.job
```

The batch script is set up to run in a reserved GPU queue that is active for the practical session from 11-12 on Monday 11th November. Outside of that time (or if you are a student from an external programme, not the MSc in HPC) you will need to edit the batch script appropriately. We are **only interested in the calculation time** which excludes initialisation, IO etc. which are not parallelised.

The script is set up to run the sharpen code on a range of thread counts from 1 to 40. How well does it scale? How much faster is it than using the CPUs on the standard compute nodes?

4 GPU version

The GPU version is parallelised using CUDA which is a low-level programming model specific to NVIDA GPUs. In CUDA, the user writes a small kernel function to be executed by a single thread on the GPU. The program runs on the host CPU but there is special syntax which launches the kernel in parallel across large numbers of GPU threads (remember a single GPU has thousands of simple cores compared to a CPU which will have tens of general-purpose cores). Here, the kernel is the filter function that is applied to every pixel in the fuzzy image.

Since the CPU and GPU have separate memory spaces there are also routines to allocate GPU memory and transfer data from/to the CPU. CUDA does not have good support for multidimensional arrays so the CUDA filter kernel `dosharpenpixel.cu` looks horrible compared to the equivalent CPU version!

4.1 Default setup

To compile and run the program on a single GPU:

```
cd ../C-GPU
make
sbatch sharpengpu.job
```

You should always check that the output image `sharpened.pgm` is identical to the CPU version: runtime errors on the GPU are not always trapped properly so a program can run to completion but produce incorrect output. This will give unreliable performance results.

How much faster is the GPU version than a single CPU-core? How much faster is it than using all 40 CPU-cores? Do you think this additional performance is worth the cost (note that you may find it hard to find reliable price information on CPUs and GPUs for HPC!).

The performance increase we get for `sharpen` is unusually high as it is almost entirely CPU-bound rather than memory-bound: it computes a very large number of exponential functions. Although GPUs also have much better memory bandwidth than CPUs, the performance increase for a real program is unlikely to be as large as you observe here.

4.2 Varying the number of threads

The code that launches the kernel is in `dosharpen.cu` - the relevant parts are:

```
dim3 nthread = {16, 16, 1}; // 256 in a 16x16 grid
...
dosharpenpixel<<<nblock, nthread>>>(nx, ny, d, d_conv, d_fuzzyp);
```

The «`<nblock, nthread>`» syntax launches the kernel function `dosharpenpixel()` multiple times, each running on a block of `nthread` GPU threads. Each thread operates on a single pixel: the number of blocks is automatically chosen to cover the whole image. For a 2D image it is convenient to think of each thread block as a 2D grid: here we have 256 threads in each block arranged in a 16x16 grid.

GPUs use a Single Instruction Multiple Thread (SIMT) architecture with threads arranged in "warps" where all threads in a warp must run the same instruction. If our thread block is smaller than the warp size then some of the threads will be idle leading to reduced performance.

Vary the number of threads in a block by editing `dim3 nthread = 16,16,1` to different values `{1,1,1}`, `{2,2,1}`, ..., `{32,32,1}`. Each time you will need to recompile and resubmit: make sure you wait until each job completes before each recompilation, otherwise you may not be running the program you expect!

From how the performance varies as a function of the the number of threads in a block, how many threads do you think there are in each warp?