Getting started with GPUs and the DAS-4

For information about the DAS-4 supercomputer, please go to:
`http://www.cs.vu.nl/das4/`

For information about the special GPU node hardware in the DAS-4 go to the DAS-4 site, then select "Users → Special Nodes"

For more information on the software for programming GPUs navigate to
"Users → GPUs"

The host name of the VU cluster we are using is: `fs0.das4.cs.vu.nl`. Thus, use `ssh` to connect to the cluster:

```
ssh -Y username@fs0.das4.cs.vu.nl
```

Introduce the password. If correct, you are now logged in.

If you are a MacOS or Linux user, ssh is already available to you in the terminal.
If you are a Windows user, you need to use a ssh client for Windows. The easiest option is to use putty:
http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

For using the GPU nodes, we need a bit of configuration. Type the following line at the prompt:

```
module load cuda55/toolkit prun
```

You need to do that every time after you log in.

Alternatively, add the same line to your `.bashrc` file, which is found in your home directory (doing so makes this change permanent and automatically loaded at the start of any ssh session; in other words, you don't need to type it every time). If you use the `.bashrc` option, log out (use exit in you ssh session) and log in again. If this step succeeded, you should be able to run the CUDA compiler now, so please try:

```
nvcc --version
```

This should print:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2013 NVIDIA Corporation
Built on Wed_Jul_17_18:36:13_PDT_2013
Cuda compilation tools, release 5.5, V5.5.0
```

For running jobs, we use `prun` with different parameters. `prun` instructs the system that a job is ready to run and hat are its parameters. We typically use this command:

```
prun -v -np 1 -native '-l gpu=GTX480' <EXECUTABLE>
```

To simplify this execution, we can create an alias. In your terminal, write:

```
alias gpurun="prun -v -np 1 -native '-l gpu=GTX480'"
```

Try, for instance:

```
gpurun $CUDA_SDK/bin/x86_64/linux/release/deviceQuery
```

(the above is written on a single line!)

If your job doesn't start immediately, it means the system is busy running other jobs. Thus, you can check the queue status with:

```
preserve -long-list
```

Your job is probably listed there, waiting for its turn. You can cancel it and try using another type of GPU, or let it wait for its turn.

Folders and applications

The documentation for CUDA is in:
`/cm/shared/apps/cuda55/sdk/5.5.22/doc/`

Especially the CUDA programming guide (`CUDA_C_Programming_Guide.pdf`) is a good starting point and reference for learning and using CUDA. In general, the CUDA documentation is excellent, so use it! You can view it with the "evince" program.

```
evince /cm/shared/apps/cuda55/sdk/5.5.22/doc/
CUDA_C_Programming_Guide.pdf
```

In principle, everything you need for the GPU hands-on session is in:
`/var/scratch/alvarban/HPC_GPU_2k17`
You MUST copy this code to your own account, and work with it.

```
cp -r /var/scratch/alvarban/HPC_GPU_2k17 $HOME
```

DO NOT EDIT the code in the folder `/var/scratch/alvarban`

There are several interesting folders for this practical: `vector-*`, `crypto`, `conv`, and `difficult`.

1. Vector addition

We start with `vector-add`, which contains most of the code for a simple vector addition.

Exercise 1.1:
Identify the kernel, and add the necessary operation to implement the vector addition. Change the operation and compare the performance of vector addition, subtraction, multiplication, division.

Compile and run the code. You can compile the code with `make`.

To run it:
```
gpurun ./vector-add
```
or
```
prun -v -np 1 -native '-l gpu=GTX480' ./vector-add
```

The result should be something like:
```
vector-add (Sequential): 0.000155 seconds.
vector-add (kernel): 0.000055 seconds.
vector-add (memory): 0.000411 seconds.
results OK!
```

Run the code for at 5 different sizes of the array: 256, 1024, 65536, 100000, and 1000000.
Report the execution times for each timer. What do you observe?

Exercise 1.2 (optional):
Take a look at the vector-add_events.cu. Notice the difference in measuring performance by comparing the code with the code in vector-add.cu.

Exercise 1.3:
We now work on the `vector-transform` folder.
Please write the kernel that implements the vector transformation from the sequential version (as seen in function `vectorTransformSeq`.
Run the code for at 5 different sizes of the array: 256, 1024, 65536, 100000, and 1000000.
What do you observe, performance-wise?
Compare against the performance of vector-add. What do you observe?

Exercise 1.4 (optional):
Take a look at the `vector-add-streams` version of the code.
What is the difference with vector-add? What is the value of nStreams that delivers the best performance for the overall code?

2. A Cryptography example

There are many cryptography examples that can be accelerated using parallel processing. The simplest of them is Cesar's code.
In this symmetric encryption/decryption algorithm, one needs to set a numerical key (1 number) that will be added to every character in the text to be encoded.

For example:
Input : ABCDE
Key: 1
Encrypted output: BCDEF

In this assignment you are requested to build a parallel encryption/decryption of a given text file.

The starting code for this example can be found in the folder `crypto`.

Exercise 2.1:
Please implement a correct encryption and decryption and test it on at least 5 different files. Make a correlation between the size of the files and the performance of the application for both the sequential and the GPU versions. Report speed-up.

Note that the file names are fixed: `original.data` is the file to be encrypted, `sequential.data` is the reference result for the CPU encryption, and `cuda.data` is the result of the GPU encryption. You are recommended to use `recovered.data` for the decryption, which should be identical with `original.data`.

To test whether the files are identical, use the `diff` command:
```
diff file1 file2
```

If no output is produced the files are identical. If there is a list of differences printed on the screen, these are marked by position in the original file(s).

Exercise 2.2:
A very interesting extension of this encryption algorithm is to use a larger key - i.e., a set of values, applied to consecutive values.

For example:

Input: ABCDE
Key : [1,2]
Output: BDDFF

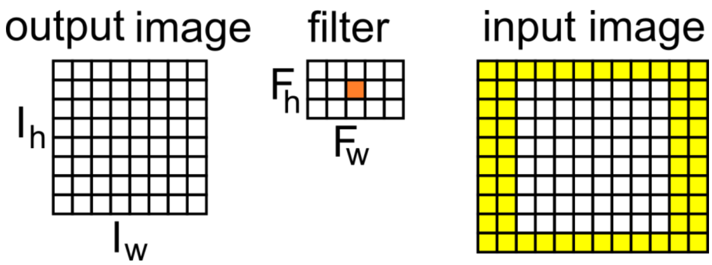
Please implement this encryption/decryption algorithm as an extension to the original version. You can assume the key is already known (fixed).
Test this extended version for the same few files as in the previous case, and compare again the results against the sequential version. Report speed-up per file.

3. Convolution

2D convolution is an application that finds its use in many domains. The most common is, most likely, image processing, where many filters are based on convolution.

The idea is as follows: given an input image and a filter, the output image is computed such that every pixel is a convolution of the filter and input image. The convolution is a dot product operation.

For example:



A sequential implementation of the convolution is:

```
//for each pixel in the output image
for (y=0; y < image_height; y++) {
  for (x=0; x < image_width; x++) {
     //for each filter weight
      for (i=0; i< filter_height; i++) {
        for (j=0; j < filter_width; j++) {
          output[y][x] += input[y+i][x+j] * filter[i][j];
}}}}
```

The code for convolution is to be found in the folder conv.

Exercise 3.1:
Please implement a naive kernel for the convolution. Report speed-up over the sequential implementation.

Exercise 3.2:
Propose and implement different optimizations for this code (e.g., consider shared memory). Report speed-up over the previous implementation.

4. A more difficult assignment

You are requested to implement an image processing pipeline: the pipeline takes a color image as its input, convert it to grayscale, use the image's histogram to contrast enhance the grayscale image and, eventually, returns a smoothed grayscale version of the input image. For simplicity and accura-cy all operations are done in floating point. The program must be benchmarked on the NVIDIA GTX480 GPUs on the DAS-4. A brief description of the four algorithms follows.

Converting a color image to grayscale

Our input images are RGB images; this means that every color is rendered adding together the three components representing Red, Green and Blue. The gray value of a pixel is given by weighting this three values and then summing them together. The formula is:

```
gray = 0.3*R + 0.59*G + 0.11*B.
```

Histogram Computation

The histogram measures how often a value of gray is used in an image. To compute the histogram, simply count the value of every pixel and increment the corresponding counter. There are 256 pos-sible values of gray.

Contrast Enhancement

The computed histogram is used in this phase to determine which are the darkest and lightest gray values actually used in an image - i.e., the lowest (min) and highest (max) gray values that have "scored" in the histogram above a certain threshold. Thus, pixels whose values are lower than min are set to black, pixels whose values are higher than max are set to white, and pixels whose values are inside the interval are scaled.

Smoothing

Smoothing is the process of removing noise from an image. To remove the noise, each point is re-placed by a weighted average of its neighbors. This way, small-scale structures are removed from the image. We are using a triangular smoothing algorithm, i.e. the maximum weight is for the point in the middle and decreases linearly moving from the center. As an example, a 5-point triangular smooth filter in one dimension will use the following weights: 1, 2, 3, 2, 1. In this assignment you will use a two-dimensional 5-point triangular smooth filter.

A sequential version of the application is provided, for your convenience, in the directory “sequential”. Please read the code carefully, and try to understand it. A template for the parallel version is also provided, in the “cuda” directory. We recommend that you to start from the template implementation that is provided. You need to parallelize and offload the previously described algorithms to the GPU. The "Kernel" comment in the code indicates the part that must be parallelized.

There are no assumptions about the size of the input images, thus the code must be capable of running with color images of any size. The output must match the sequential version; a compare utility is provided to test for this. The output that you should verify is the final output image, named **smooth.bmp**. You are free to also save the intermediate images, e.g. for debugging, but do not include the time to write this images in the performance measurements.

In the directory “images”, 16 different images are provided for testing. You can measure the total execution time of the application, the execution time of the four kernels and the (introduced) memory transfer overheads. This way, you can compute the speedup over the sequential implementation, the achieved GFLOP/s and the utilization.

Exercise 4:

Try to optimize (at least) the histogram computation and the smoothing filter (hint: use shared memory). As an indication, the execution times (in milliseconds) measured for the computation of the largest image in the set (gpu/images/image09.bmp), can be below the following thresholds:

- Grayscale Conversion < 5.5 ms
- Histogram < 68 ms
- Contrast Enhancement < 8.7 ms
- Smoothing < 84 ms
- Total execution < 1023 ms

Compiling and Running Your Application

Please use the provided Makefiles for compiling. Now, you can run your parallel Cuda application with, for example:

```
prun -v -np 1 -native '-l gpu=GTX480' ../bin/cuda
../images/image02.jpg
```

You can look at the results (e.g., smooth.jpg) with the “display” command.
Enjoy!