Universiteit van Amsterdam
BSc Informatica
Dr. Robert G. Belleman & Dr. Ana Lucia Varbanescu

Concurrency en
Parallel Programmeren
2015/2016

Assignment 4

Many-Core Programming with CUDA

Getting started with CUDA on the DAS

For these assignments you will be using the GPUs installed in the DAS-4 cluster. Specifically, you can use the VU cluster (hostname: `fs0.das4.cs.vu.nl`) or the TUDelft cluster (hostname: `fs3.das4.tudelft.nl`).

The GPUs you can use are listed here: `http://www.cs.vu.nl/das4/special.shtml`.

For using the GPU nodes, we need a little bit of configuration. Add the following lines to your `.bashrc`:

```
module load cuda55
module load prun
```

Now, log out and log in again. If all is ok, 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, use this command:

```
prun -v -np 1 -native '-l gpu=GTX480' <EXECUTABLE> [arg1 arg2 ...]
```

Try, for instance:

```
prun -v -np 1 -native '-l gpu=GTX480' \
$CUDA_SDK/bin/x86_64/linux/release/deviceQuery
```

If your job doesn't start immediately, you can check the queue status with:

```
preserve -long-list
```

Testing your setup

The framework contains a directory `vector-add` with sample code for a simple vector addition. Compile and run that code first. Build the code with `make` and then run it with:

```
prun -v -np 1 -native '-l gpu=GTX480' ./vector-add
```

The result should be something like:

```
vector add timer: avg=541ms, total=541ms, count=1
results OK!
```

Study the code in `vector-add.cu`, and copy/paste parts of this code into your assignment later, so you don't have to start from scratch!

Note that you can use the GTX480 GPUs (most of the available cards are GTX480 cards), but also the other cards. In case you choose for other cards, please make sure you change the Makefiles to reflect the correct compute capability of your card (see https://en.wikipedia.org/wiki/CUDA). Specifically, you have to replace one line from Makefile or Makefile.inc (more details: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#abstract):

Original (GTX480): `-arch compute_20 -code sm_20`

Modified (e.g., GTX680): `-arch compute_30 -code sm_30`

In any case: make sure you clearly state in your report which card(s) and flag(s) you have used for your experiments.

Assignment 4.1: Wave equation simulation

Reconsider the 1-dimensional wave equation studied in the OpenMP and MPI assignments. Parallelize your sequential simulation code using CUDA. Use the source code for `vector-add` as a starting point.

TO DO List:

1. Time the performance of your implementation by running experiments with different problem sizes, i.e. number of amplitude points being 10^3, 10^4, 10^5, 10^6, 10^7. For the number of time steps, take the same as you did with your earlier experiments so that you can compare the results. For now, fix the number of threads per block to 512. Report your results, and compare them with those of your implementation from the OpenMP and MPI assignments.

2. Experiment with different thread block sizes, using 8, 16, 32, …, 512, 1024 threads per block (on the same card). Report your results and explain why some sizes perform better than others.

3. Compare the accuracy of the results generated by your CUDA implementation with those generated by your sequential implementation. Report your findings and explain the expected behavior, as well as any unexpected results.

4. Are there any optimizations you see possible for the CUDA solution you have implemented? If so, which ones? Please discuss and/or apply (some of) them. If not, please explain why no optimizations are possible.

Assignment 4.2: Parallel cryptography in CUDA

There are many cryptography algorithms that can be accelerated using parallel processing. The simplest one of them is Cesar's code. In this symmetric encryption/decryption algorithm, one needs to set a numerical key (1 number, typically between 1 and 255) 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 crypto.zip. To compile, use the provided Makefile. To execute the code, use the same structure as for the vector-add example:

```
prun -v -np 1 -native '-l gpu=GTX480' ./encrypt
```

TODO List:

1. Implement a correct encryption and decryption sequential versions and CUDA kernels (replacing the *dummy* ones already there) and test them on at least 5 different files of different sizes (from small (few KB) to very large (MB or tens of MB); you can use any (text) file as an input for the algorithm, and you can get very large text files online). Report speed-up per file per operation, and compare all these speed-ups in a graphical manner. Is there a correlation (to be seen) between the size of the files and the performance of the application for the sequential and the GPU versions? Explain.

Note that the file names are hardcoded: 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. Do not submit any input files with your solution, but make sure you state, in your report, the size (in bytes) of each input file.

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

If the file `differences` is empty, the files are identical. If the file is not empty, it will contain the differences between the two files, marked by their position in the original file(s).

2. An extension of this encryption algorithm is to use a larger key - i.e., a set of values, applied to consecutive characters.

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

Implement this encryption/decryption algorithm as an extension to the original version (1). You can assume the key is already known (fixed, constant), and choose its length (or, better, test with multiple lengths). Optimize your code as you see necessary for this extended version, and test the extended version with the same files as for point 4.2.1. Compare the results against the single-value key, report the comparison in a graphical form, and comment on your findings.

3. Implement a kernel to efficiently calculate the checksum of a file in CUDA (use a simple, additive checksum). For doing this efficiently in parallel, the information presented in class is sufficient. Report the performance of the kernel for each file (in a graphical form, too) and comment on the correlation (if any) between performance and the file size. Apply this kernel to both the original and encrypted files, and compare the output. Comment on your findings.

Assignment due date: CODE on December 3, 2015, 23:59 (midnight).

Assignment due date: REPORT on December 4, 2015, 23:59 (midnight).