University of Amsterdam
BSc Informatica

Concurrency and Parallel Programming

Lab 4: CUDA

Many-Core Programming with CUDA

dr. Robert Belleman dr. Ana Varbanescu

December 3, 2018

Getting started with CUDA on DAS

For these assignments you will be using the GPUs installed in the DAS4 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`.

To use the GPU nodes, we need a little bit of configuration, to address two things: the gcc compiler needs to be 4.8.5 or 4.8.3, and the CUDA version needs to be 8.0.

For gcc, please check which version of gcc is preloaded. If higher than needed, unload the module (example below), check the version again, and load a new module if needed.

For example:

```
gcc --version
gcc (GCC) 6.3.0
module unload gcc/6.3.0

gcc --version
gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-16)
```

Next, please make sure you load the CUDA module:

```
module load cuda80
```

You can either run these commands every time you login, or you can add the lines to your .bashrc, then 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

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2016 NVIDIA Corporation
Built on Tue_Jan_10_13:22:03_CST_2017
Cuda compilation tools, release 8.0, V8.0.61
```

If you receive output similar to the above, you are ready to execute CUDA programs as follows:

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

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:

```
$ make
$ prun -v -np 1 -native '-C GTX480 --gres=gpu:1' ./vector-add
```

When running make, you might get a couple of warnings about deprecated flags and/or functions without returning values - you can ignore those.

If you get an error about `prun`, you need to make sure the module is loaded, too:

```
module load prun
```

If the compilation and execution have succeded, you should see something like this (excerpt only:)

```
Adding two vectors of 65536 integer elements.
vector-add (sequential):                   = 0.000339496 seconds
...
```

If you reach this point, your environment is setup correctly. You are ready to start working on the assignments.

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 (especially GTX680). In case you choose for other cards, please make sure you change the Makefiles to reflect the correct compute capability of your card. Specifically, you have to replace one line from your Makefile or Makefile.inc as follows. In any case: make sure you clearly state in your report which card(s) and flag(s) you have used for your experiments.

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

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

Assignment 4.1: Wave equation simulation with CUDA

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. If you want, you can follow these steps:

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 Caesar'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, `Caesar('ABCDE', 1) = '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. Execute the following steps using the framework:

1. Implement sequential encryption and decryption versions and CUDA kernels (replacing the dummy ones already there) and test them on at least 5 different files of different sizes from small (a couple of kilobytes) to very large (many megabytes). 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 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 while `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. If this command produces non-empty output, the input files are not identical.

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, `Caesar('ABCDE', [1, 2]) = 'BDDFF'`. Implement this encryption/decryption algorithm as an extension to the original version. 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 the same way you did before. 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.