CS 758: Programming Multicore Processors (Fall 2016)

Homework 7

You should do this assignment alone. No late assignments.

Filelist for the assignment:

- Template Files
- Intro to using Euler cluster

The purpose of this assignment is for you to familiarize yourself with GPGPU computing platforms (CUDA) and to gain experience with GPGPU specific optimizations. For this assignment you will be given a basic implementation of an algorithm which runs on the GPU and you will procedurally improve it applying GPGPU optimization principals.

IMPORTANT

CUDA can be tricky, especially if you make a mistake. Error messages are often cryptic and uninformative. Start this assignment early! If you run into any problems post on the email list.

The problem

For this assignment you will again be implementing the Ocean algorithm. For more information see (Homework 1). You will be comparing the performance of your GPU-optimized algorithm to your solution from Homework 1. A simple solution to homework 1 is also included in the template files feel free to use it if you want.

The hardware

You will be using the Euler cluster. You should have or soon will receive an email with a username and temporary password. (MAKE SURE YOU RESET YOUR PASSWORD!) Read the above tutorial that describes the hardware configuration.

Job submission

You should do this assignment using the SLURM *sbatch* submission mechanism. Notes on using SLURM

To get started:

```
local $ ssh user@euler.wacc.wisc.edu
euler $ ssh euler01
euler01 $ scp <username>@ale-01.cs.wisc.edu:/p/course/cs758-
david/public/html/Fall2016/handouts/homeworks/hw7-dist.tgz .
euler01 $ tar -x -f hw7-dist.tgz
euler01 $ mv hw4-dist hw7
euler01 $ cd hw7
euler01 $ make
euler01 $ ./serial_ocean.sh
```

You shouldn't have any problems as long as your code finishes quickly and you don't leave cuda-gdb open for long periods of time (they have come across a few bugs where cuda-gdb sometimes blocks access to all other GPUs).

Information on the hardware provided in the Euler cluster is available here. You will use one of the Fermi cards (Tesla 2070/2050 or GTX 480). Each of which as 448 CUDA cores (14 SMs).

Distributed with CUDA 5.5 is an application called *computeprof* which does a good job of concisely representing the performance counters available on the NVidia GPUs. To use this program, you will need to use "`ssh -X`" to login to the Euler cluster in order to forward the X server. You can then run it using "`/usr/local/cuda/5.5.22/cuda/bin/computeprof`" I recommend sitting on campus while doing this since there is much higher bandwidth. You can use *computeprof* to diagnose the bottlenecks in each implementation of the algorithm.

Additional Information

Dan Negrut taught a GPU Computing course (ME964) last fall. If you need additional info for your homework, you may find what you need at his course web page:
http://sbel.wisc.edu/Courses/ME964/2015/
There is also a forum where students in that class posted questions/answers. You may find useful information here:
http://sbel.wisc.edu/Forum/viewforum.php?f=15

Step 0: Pre-porting Exploration

GPU programming is a challenging and time-consuming process. You only want to do it if you are absolutely sure that you can get reasonable speedup compared to your serial implementation. Also, GPU optimization seems like a never-ending process, with no clear stopping point.
To this end, we introduce you to a GPU performance prediction framework which predicts the potential speedup on GPU based on the serial implementation of your program. You will use the prediction as a rough guideline into tuning the amount of effort you put into porting.
The first step in using a GPU performance model is to construct the model for your target GPU. We have already done this step and embedded the model in the tar file.
The second step is to collect program properties from your program, serial ocean, and feed it into the model to get the final speedup prediction.
This is as easy as adding region-markers to your program and running:
./push-button.sh [benchmark_suite] [benchmark_name] [input].
For instance, you can predict speedup for ocean, with the input of `128 128 10`, using the following command:
./push-button hw7 ocean 1
The second argument can take values between 1 to 18. Check run.py to find the mapping between the input variable and arguments.

Under the hood, R and Pintools will be installed, and program properties will be collected and fed into the model to make the speedup prediction.

Note 0. Make sure to use the new tarfile. I have modified the underlying Makefile and added region markers to the serial code.[Updated Saturday Oct29, 2:30 pm].
Note 1. Make sure to run the tool on euler machines. The model is specific to euler machines.
Note 2. Modify the variable "here" in push-button.sh to point to the path where you uncompressed hw7-XAPP.tar.gz. When uncompressed, there are two directories and one file, hw7, XAPP and ./push-button.sh.
Note 3. Let the feature collection phase finish to the end. If you kill the process early, the result would be unreliable.

- Question a) What is the predicted speedup for the input of `4098 4098 100`?
- Question b) Based on the predicted speedup, is it worth to port ocean to GPU? Why?
- Question c) How does the predicted speedup vary with the grid size? How does it vary with the time-stamp size?

(Regardless of your answer to the question b, you need to proceed to the next step, for pedagogical purpose!)

Step 1: Porting the CPU algorithm

I have included this implementation of the `ocean_kernel` in the template files. You can find it in *cuda_ocean_kernels.cu* after `#ifdef VERSION1`. Although considerably more verbose, this is a mostly literal translation of the algorithm in *omp_ocean.c* with OpenMP static partitioning. Each thread gets a chunk of locations within the red/black ocean grid and updates those locations. Study this code and be sure to understand how it works.

- Question a) Describe *memory* divergence and why it leads to poorly performing code in the SIMT model.
- Question b) Describe the *memory* divergence behavior of `VERSION1` of `ocean_kernel`.
- Question c) Vary the block size / grid size. What is the optimal block / grid size for this implementation of ocean? What is the speedup over 1 block and 1 thread ("single threaded")? Run with an input of `4098 4098 100`.
- Question d) What is the speedup over the single threaded CPU version? Run with an input of `4098 4098 100`.
- Question e) How close did you get to the predicted speedup? Did you surpass the prediction? Report the relative error (|predicted speedup - measured speedup|/measured speedup) for different input sizes.

Step 2: Reduce memory divergence (Convert algorithm to "SIMD")

Implement `VERSION2` of `ocean_kernel`. This version of the kernel will take a step towards reducing the memory divergence. Instead of giving each thread a chunk of the array to work on, re-write the algorithm so that the threads in each block work on adjacent elements. (I.e. for a red iteration, thread 0 will work on element 0, thread 1 will work on element 2, thread 3 will work on element 6, etc).

- Question a) Describe where *memory* divergence still exists in this implementation of ocean.
- Question b) Vary the block size / grid size. What is the optimal block / grid size for this implementation of ocean?
- Question c) How does this version compare to VERSION1? Run with the optimal block sizes for each respectively and an input of `4098 4098 100`.

Step 3: Further reduce memory divergence (Modify data structure to be GPU-centric).

Implement `VERSION3` of `ocean_kernel`. Instead of using one flat array to represent the ocean grid, split it into two arrays, one for the red cells and one for the black cells. You should start by writing two other kernels which will split the grid object into red_grid and black_grid and take red/black_grid and put them back into the grid object.

If you're feeling adventurous, feel free to add any other optimizations to this implementation. Just describe them in your write-up.

- Question a) How does the performance of this version compare to VERSION2? Is this what you expected?
- Question b) Time each kernel and the memory copies separately (ocean_kernel, and (un)split_array_kernel). Which actions are taking the most execution time? How does this affect the overall execution time of the algorithm? (*computeprof* does a good job summarizing this data)
- Question c) Vary the block size / grid size. What is the optimal block / grid size for this implementation of ocean? Does it change when you change the problem size?
- Question d) Describe *branch* divergence and why it leads to poorly performing code in the SIMT model. Does your code exhibit any branch divergence? If so, where?
- Question e) Given each node in the Euler cluster has 2 Intel Xeon E5520 processors and the GPUs have 448 CUDA cores (GTX480/C2050/C2070) how do you think the performance of

your GPU version will compare to the CPU version?
- Question f) Run either your OpenMP version of ocean or the one in the template files. How does the performance of the CPU version of Ocean compare to the GPU version, better or worse? Why do you think this is? Use omp_ocean.sh to submit the OpenMP version. Run with problem sizes 1026, 2050, 4098, and 8194 with 100 timesteps.
- Question g) What do you think of CUDA? SIMT programming in general?

Tips and Tricks

- **Start early.**
- Be mindful that Professor Dan Negrut has been gracious to allow us to use his computing resources for this assignment.

What to Hand In

Please turn this homework in on **paper** at the beginning of lecture. You must include:
- A printout of your GPU kernels
- Answers to all of the questions and supporting graphs.

Important: Include your name on EVERY page.

Page last modified on October 29, 2016, at 02:56 PM, visited times