CSE 5441 (Fall 2019, Dr. Jones)
CUDA (Lab 4)

Caleb Lehman
lehman.346@osu.edu

November 16, 2019

Overview

For this lab, I parallelized my serial C program to perform Adaptive Mesh Refinement (AMR)[1] using the CUDA API. I created 2 programs, cuda-disposable and cuda-persistent, which mirror the disposable and persistent threads models used in the previous labs. In particular, the disposable code runs a new kernel for each iteration, while the persistent code has a single kernel which synchronizes the threads between iterations[2].

Tests

Environment

The program was developed and tested on the Owens cluster at the Ohio Supercomputer Center. Note that all my other labs were developed and tested on the Pitzer cluster, which has different specifications.

For development and testing, I loaded the cuda/9.2.88 module, which allowed the program to be compiled with version 9.2.88 of the nvcc compiler, as well as setting some environment variables pointing to CUDA-related headers and libraries.

For testing, I loaded the python/3.6-conda5.2 module, which loads a python environment with the NumPy, SciPy, and Matplotlib packages, amoung others. Python is only necessary for collecting and plotting the data from testing, not for the actual exectuion of the program.

Timing

I collected timing data using the same 4 methods as the first several labs: time, clock, and clock_gettime from the "time.h" header, and the UNIX utility time.

As with the second and third labs, I didn't use the results from the clock function from the "time.h" header, since it reports CPU time, not wall time. The other methods all returned values within 1 second of each other. The time function declared in "time.h" returns an integer number of seconds, but the other two methods return with sub-second precision. *For consistency, I used the clock_gettime function for all results in this report.*

Test Files

Dr. Jones provided the testgrid_400_12206 test file. As part of lab 1, I reduced the α (affect rate) and ε parameters until the serial runtime increased into the 3 to 6 minute range. In particular, I selected $\alpha = 0.01$ and $\varepsilon = 0.02$, for which the serial program completed in 261 seconds. However, my CUDA implementations yielded very poor performance, so I ended up using parameters of $\alpha = 0.1$ and $\varepsilon = 0.1$[3]. For this reason, and due to using the Owens cluster instead of the Pitzer cluster, it isn't particularly useful to compare the results from this lab to the results from labs 2 and 3.

[1] See lab 1 or project descriptions for details about AMR computation.

[2] Because the CUDA API doesn't directly provide a simple way to synchronize blocks, the cuda-persistent version is limited to running with a single block of threads.

[3] In any case, these parameters really only affect the final DSV values and number of iterations, not the actual amount of work done per iteration, so most of the relative results found in this report should be the same for different values of α, ε.

Results

The output was consistent across both programs and was as follows:

- testgrid_400_12206: 75197 iterations, $(max, min) = (0.077873, 0.086525)$

The relevant runtimes for the serial, cuda-persistent, and cuda-disposable versions on the test case are visualized in the figures below. I found that the cuda-persistent version had optimal performance around 320 threads (and 1 block) and the cuda-disposable version had optimal performance with 32 threads and 10 blocks. Those values are now hardcoded into the program, as requested. With the optimal values for threads and blocks, cuda-persistent ran in 106 seconds and cuda-disposable ran in 54 seconds.

The two biggest implications I see in this expirement are:

- My serial program (with the same $\alpha = 0.1, \varepsilon = 0.1$ parameters) runs in 14 seconds, so the CUDA versions are significantly slower. Since other parallelization methods (OpenMP and pthreads) improved performance, this indicates that my CUDA implementation was probably poorly done. I wasn't able to determine why the performance was so bad, but I suspect that it has something to do with cache performance, since the cache on the GPU is much smaller than that on the CPU.

- When comparing the cuda-persistent and cuda-disposable version with the same thread/block parameters, the cuda-disposable version consistently performed better. For example, when both programs used 256 threads and 1 block, cuda-disposable ran in 110 seconds, while cuda-persistent ran in 124 seconds. This shows that it is faster to kill and restart the entire kernel (cuda-disposable) than to just synchronize the threads (which do slightly uneven amounts of work in cuda-persistent). This is the opposite of the behaviour I noticed in the pthreads lab, confirming that CUDA threads are much more lightweight than even the relatively lightweight pthreads.

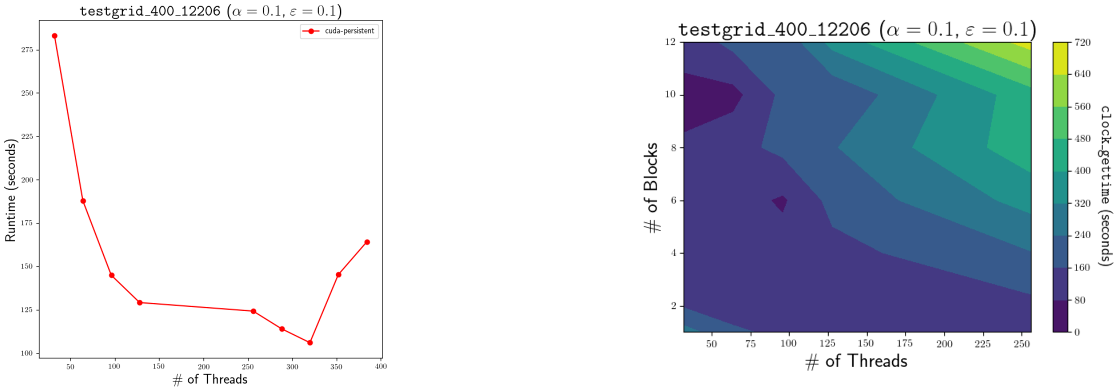

Figure 1: The runtimes of the cuda-persistent (left) and cuda-disposable (right) parallelizations. Note that, due to the poor performance of the CUDA parallelization, higher α and ε parameters were used compared to previous labs. For comparison, my serial program runs in 14 seconds with these parameters. **Important note:** the cuda-persistent plot displays the runtimes on a scale starting at 100 seconds.

The following is the series of computations performed for the ith box during each iteration, which make up basically all of the "useful" work done in the programs:

```
updated_vals[i] = box->self_overlap * current_vals[i];
for (int nhbr = 0; nhbr < box->num_nhbrs; ++nhbr) {
    updated_vals[i] += box->overlaps[nhbr] * current_vals[box->nhbr_ids[nhbr]];
}
updated_vals[i] /= box->perimeter;
updated_vals[i] = current_vals[i] * (1 - affect_rate)
    + updated_vals[i] * affect_rate;
```

The number of arithmetic operations in the above sample is $1 + 2 \cdot N_i + 1 + 4$, where N_i is the number of neighbors of the ith box. Letting n be the number of boxes, N be the total number of neighbors (counted with multiplicities), and I be the number of iterations, the total number of arithmetic operations is given by $I \cdot (6n + 2N)$. For the given test file (`testgrid_400_12206`) and choice of parameters ($\alpha = 0.1, \varepsilon = 0.1$), we have the values $I = 75197$, $n = 12206$, and $N = 71890$. Substituting in these values, I computed $GFlops = \frac{ops/10^9}{\#of seconds} = \frac{16.32}{\#of seconds}$ for each of the programs to get the following results[4]:

	serial	`cuda-disposable`	`cuda-persistent`
GFlops	1.166	0.302	0.154

Project Usage

Building

To build the `cude-disposable` and `cude-persistent` executables, navigate to the top level of the submitted directory and build as follows:

```
# Ensure that you have nvcc compiler

# Note that the provided makefile
# assumes the CUDA_HOME environment
# variable is set appropriately

# On the OSC clusters, this can be achieved
# with the command:
$ module load cuda

$ make
$ ls
... cuda-persistent cuda-disposable ...
```

Running

The syntax to run the program is:

```
$ ./[program] [affect-rate] [epsilon] <[test-file]
```

where `program` is one of {`cuda-persistent`, `cuda-disposable`}

[4] I used the optimal times from each of the `CUDA` programs for this computation.