Tyler Petrochko

CPSC 524 Assignment 6

General

All the reported times were computed running on node compute-45-3 with device nvidia2. Along with my report, I've included the following source files:

kij.cu

tiled.cu

adjacent.cu

…along with their respectively named binaries:

kij

tiled

adjacent

The first source file, "kij.cu" contains the code for Task 1; namely, the naïve GPU computation along with the serial "kij" implementation. "tiled.cu" contains the source for the shared-memory computation required for Task 2, and "adjacent.cu" contains the source for the approach used in Task 3.

The data for the actual runs themselves is included as well, using the following naming convention for part (a):

task<task #>_run<run #>.txt

and

task<task #>b.txt

for part (b). As an example, the first column reported in Task 1 is generated by

$ ./kij 1024 1024 1024 16 64 `cat ../.cpsc424_gpu` > task1_run1.txt

To aid in building/running my code, I included a Makefile and a tests.sh script. To build everything, run

```
$ make
```

… And to generate all the .txt files, run

```
$ bash tests.sh
```

After changing the precision of kij.cu and tiled.cu to double, you can run the commands

```
$ ./kij 8192 8192 8192 32 256 `cat ../.cpsc424_gpu` > task1_b.txt
```

```
$ ./tiled 8192 8192 8192 16 512 `cat ../.cpsc424_gpu`  > task2_b.txt
```

… to generate the two remaining .txt files. All of the modules I used to compile my code can be generated with the command

```
$ module load Langs/Intel/15; module load Langs/Intel/15 MPI/OpenMPI/1.8.6-intel15
```

There were two runtimes I couldn't compute: the first was the serial CPU kernel for Task 1, and the second was Run 5 of Task 2. In the first case, I didn't have enough time available to run the entire test, and in the second there was an issue properly reporting the CUDA execution time (it would always give 0.0 ms regardless of how long it took to run).

Task 1

My times for part (a) were as following:

Times (ms)					
	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**
kij kernel	986	?	7595	60398	60386
GPU kernel	47	23850	386	2857	3325

And I used the following block and grid times:

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**
Block Size	16	32	32	16	16

Grid Size	64	256	32	512	512

For part (b), I recompiled the program using double precision and achieved a runtime of **35949 ms** for Run 2. This is about 50% longer than using single precision, which is actually surprising considering each individual arithmetic operation should take about twice as much time. This probably happens because, compared to Task 2, there's a good deal more non-arithmetic overhead due to non-shared memory accesses when copying matrices. This extra overhead takes up a good deal of the computation time, so the increase in arithmetic complexity is less evident

For part (c), the largest square matrix I could compute in a reasonable amount of time was of dimension 16,384. The runtime information is as follows:

Dimension	16384
Block Size	16
Grid Size	1024
Runtime (ms)	222089

Task 2

My times for part (a) were as following:

Times (ms)					
	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**
Tiled GPU kernel	25	12316	194	1545	?

And I used the following block and grid times:

	Run 1	**Run 2**	**Run 3**	**Run 4**	**Run 5**
Block Size	16	16	16	16	16
Grid Size	64	512	64	512	512

For part (b), I recompiled the program using double precision and achieved a time of **24924 ms** for Run 2. In this task, it's not surprising that using double precision took almost exactly twice as much time as using single precision. When memory is being shared locally, there's a great deal less memory overhead, so much more of the actual computation time is comprised of arithmetic operations.

For part (c), the largest square matrix I could compute in a reasonable amount of time was of dimension 16,384. The runtime information is as follows:

Dimension	16384
Block Size	16
Grid Size	1024
Runtime (ms)	98586

Task 3

Tilesize	**Time (ms)**
1	11263.5
2	12193.0
4	19166.1