Gabriel Mancino-Ball
mancig@rpi.edu

Parallel Computing
Assignment 2

CSCI 6360
February 22, 2021

1 Executing the Code

To properly run `highlife.cu`, the following arguments are required:

1. the world configuration (0, 1, 2, 3, 4, 5),
2. the world size (a power of 2),
3. the number of world updates/iterations,
4. **and** the thread count.

Thus an example of proper execution of the code is

```
./highlife 5 1024 1024 8
```

where the only difference from the first assignment is the input of the number of threads as the last integer argument.

2 Numerical Results

An empirical analysis is done on the performance of the CUDA implementation of the *HighLife* program. We test the `highlife.cu` code on varying world sizes and varying thread sizes, i.e. the level of parallelization within each world size. The relevant quantitative results to display are (a) the wall clock time for the execution of the algorithm, (b) the optimal thread count for a given world size, and (c) the number of cells updated per second for each thread count and world size combination. Tables 1 and 2 answer parts (b) and (c), respectively. Figures 1 – 7 compare the wall clock time to the thread count for each given world size.

The trends to notice are that, with the exception of the two smallest worlds, as the number of thread increase, the wall clock time of the algorithm decreases until the ”warp” size is reached, then a slight increase in running time occurs before more efficiency takes place. I conjecture this is because of the trade off that occurs between the *serial* code that is ran inside of and outside of the kernel function. To see this, compare the following results from the `time` function on the medium sized world of 8192:

8192 world size, 8 threads:

```
real 0m0.353s
user 0m0.076s
sys 0m0.171s
```

8192 world size, 32 threads:

```
real 0m0.267s
user 0m0.020s
sys 0m0.196s
```

8192 world size, 128 threads:

```
real 0m0.285s
user 0m0.042s
sys 0m0.188s
```

Notice that in general, the `user` time decreases as the number of threads increases, thus the amount of time spent outside of the kernel (i.e. swapping the pointers and synchronizing the devices) decreases. Finally, as indicated in bold in table 2, the most amount of cell updates per second can be achieved in the world size of 32786 with a thread count of 32. The only general trend to comment on in this category of performance is that as the thread count increases, so does the number of cells updated per second.

World Size	1024	2048	4096	8192	16384	32786	65536
Optimal Thread Count	64	16	128	32	64	32	256

Table 1: Fastest thread count in terms of total execution time.

	1024	2048	4096	8192	16384	32786	65536
8	3.47E+09	1.33E+10	5.71E+10	1.95E+11	7.25E+11	2.02E+12	5.66E+09
16	3.21E+09	1.66E+10	6.16E+10	2.52E+11	8.30E+11	2.29E+12	1.13E+10
32	3.09E+09	1.63E+10	6.18E+10	2.57E+11	8.01E+11	**2.54E+12**	2.26E+10
64	4.05E+09	1.44E+10	6.25E+10	1.96E+11	8.48E+11	2.41E+12	4.49E+10
128	4.01E+09	1.58E+10	6.48E+10	2.41E+11	8.35E+11	2.10E+12	8.90E+10
256	2.97E+09	1.18E+10	4.77E+10	2.54E+11	8.23E+11	2.46E+12	1.75E+11

Table 2: Cells updated per second. Bold indicates the best performance.

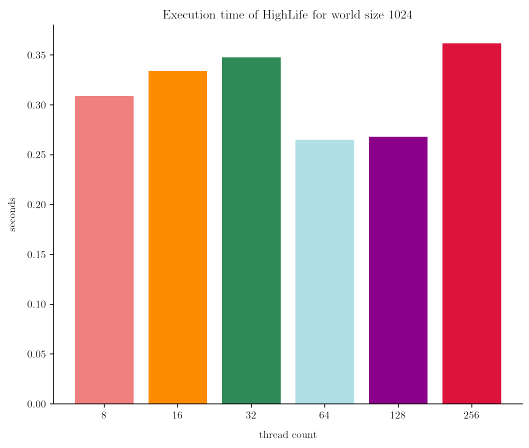

Figure 1: Execution time of various thread sizes for world size 1024

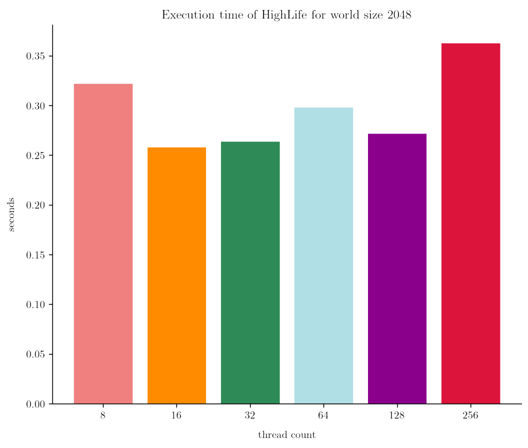

Figure 2: Execution time of various thread sizes for world size 2048

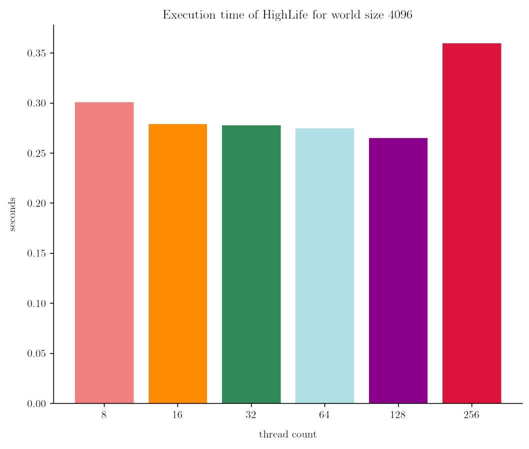

Figure 3: Execution time of various thread sizes for world size 4096

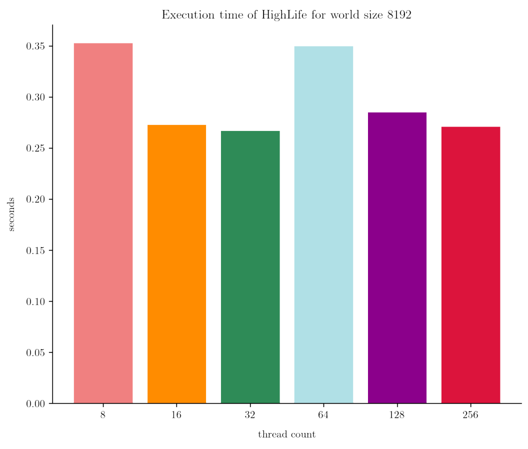

Figure 4: Execution time of various thread sizes for world size 8192

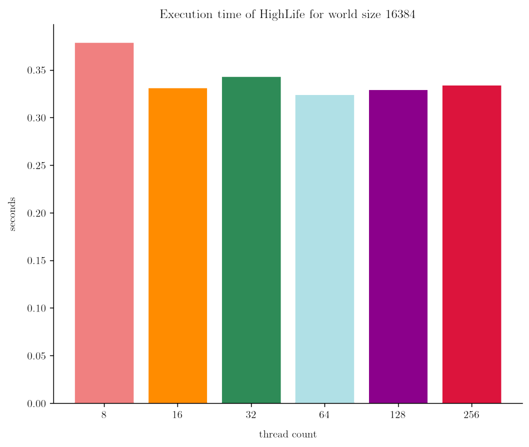

Figure 5: Execution time of various thread sizes for world size 16384

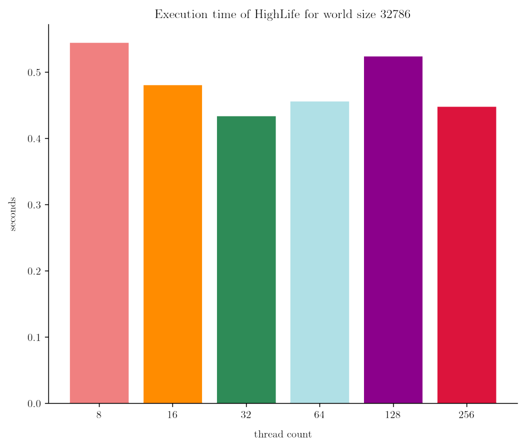

Figure 6: Execution time of various thread sizes for world size 32786

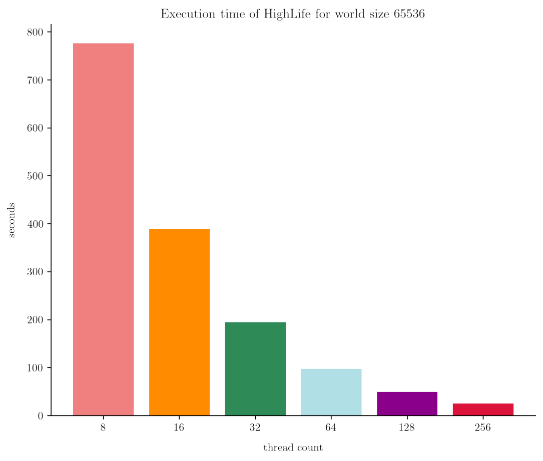

Figure 7: Execution time of various thread sizes for world size 65536