APA - Exercise 2

Tuoxing Liu, Moritz Gutfleisch

1 Exercise 2.1

- To set the boundary to an arbitrary float value:

add `boundary= some float`when executing the file FDTD3d. For example: `./FDTD3d boundary=0.5`

- The following values are defined in `FDTD3d.h`:
  - The default value of boundary is `0.0f`.
  - The minimum value of boundary is `0.0f`.
  - The maximum value of boundary is `1.0f`.

2 Exercise 2.2

Is the placement of both cg::sync(cta) statements (L110, L135) necessary for correctness?

Yes, both are necessary to ensure no conflicts can arise. The first one is necessary to ensure the WAR (write after read) dependency between reading the tile at the end of the loop (L43-L47) and updating it at the start (L25-L36). Without any synchronization points between these points, it could not be guaranteed that all threads read the correct tiles to calculate the output. The synchronization point is placed at the last possible place before writing to the tile again to minimize the time that is spent waiting for other threads.

The second call to cg::sync(cta) is required to ensure the tile is not read from before all cells have been updated (read after write). If it were not there, some threads might access either invalid memory or memory from the last iteration.

Could the load of the halo into shared memory (L121 - L132) be performed with higher parallelism?

Yes, at the moment the threads at the with indexes less than the radius update the halo on both sides (above and below / left and right), while this could be split: The threads with indexes less than the radius fetch the halo from above/left while the threads with indexes greater than blockDim.(x/y)-RADIUS fetch the halo from below/right.

Is it necessary to constantly rotate the values through infront and behind (L96, L103)?

By utilizing this circular approach, the required memory per thread for storing these values is kept pretty low, while still being able to use registers instead of directly accessing shared or global memory. Using direct access would further decrease required memory, but likely decrease execution times, as no data reuse would be performed (except through GPU caching). When doing input caching, rotating around this buffer seems to be the most memory-efficient way of doing things. Simply using a larger buffer and using it to store all values behind and infront of the thread (or just adding to the existing buffer while traversing the xy planes), might allow for better performance as long as memory is not a problem.

3 Exercise 2.3

When executing the program for different radii, the overall speed decreases (the amount of points per second), while the amount of flops per second increase. These graphs don't fully match our expectations. While it is to be expected that overall execution time increases, as the more operations have to be performed per point. We expected the amount of flops per second to also decrease, but I guess it's possible that since more accesses are made to the current tile stored in shared memory, more time is spent using this shared memory compared to the time spent initializing it than before.

Unfortunately we did not manage to get the implementation to work for radii greater than 7, for some reason the resulting values did not match the ones generated by the reference implementation. It seems probable that this has something to do with the block dimensions used, but we did not manage to find the source of the problem.

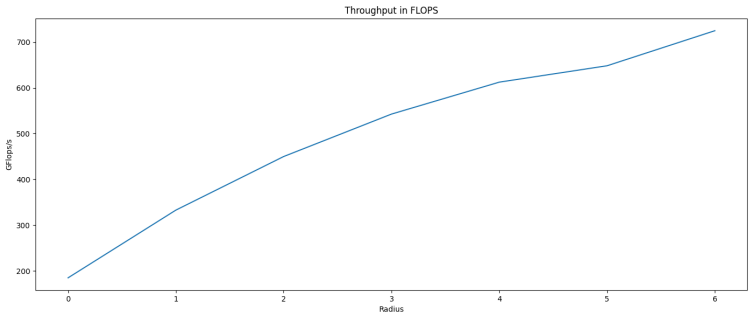

Figure 1: GFlops/s for varying Radii

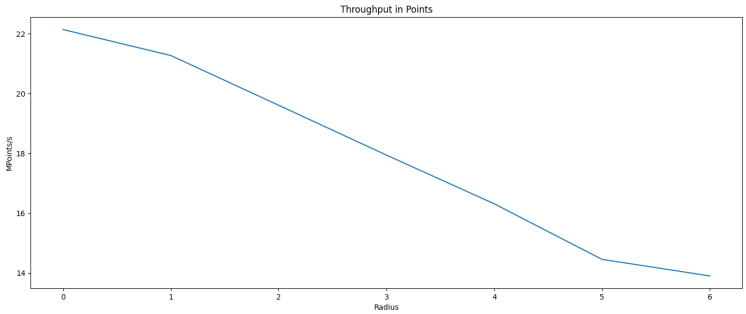

Figure 2: GPoints/s for varying Radii