II. Design and optimization approach

**II.A. Brute force method**

The brute force method for calculating the wall distance for a cell element is shown in Figure 1. For each cell element, its distance from each of the solid faces is calculated (2-D distance is shown and used for this project). Wall distance for the cell is the minimum of these distances.

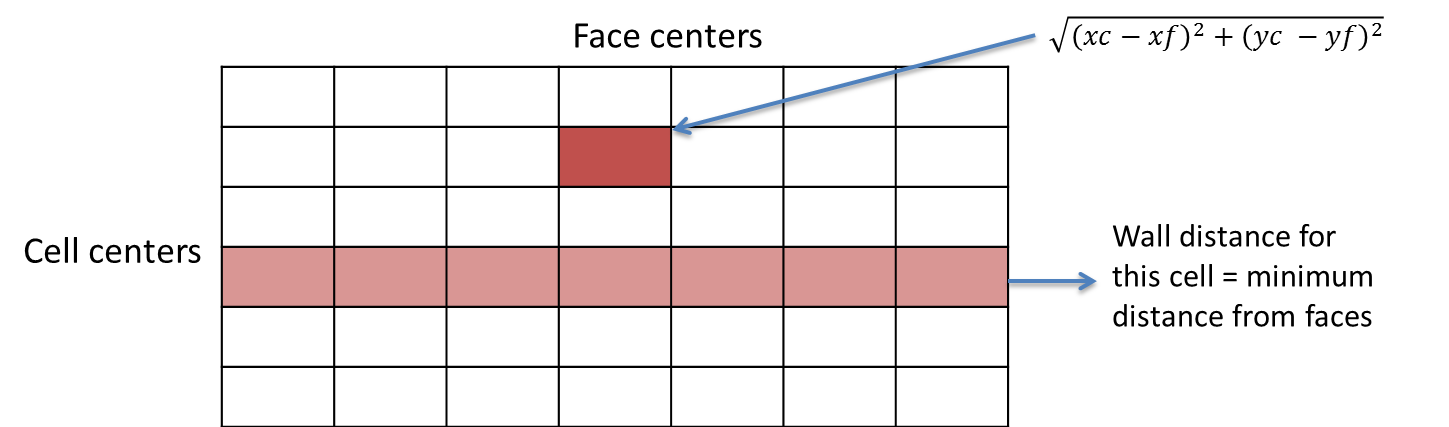


Figure 1: Brute force method

The serial implementation for the brute force method is shown in Figure 2. For each cell element, the distance from the cell to a large number of faces is calculated, and the wall distance is updated if this face has the least distance.

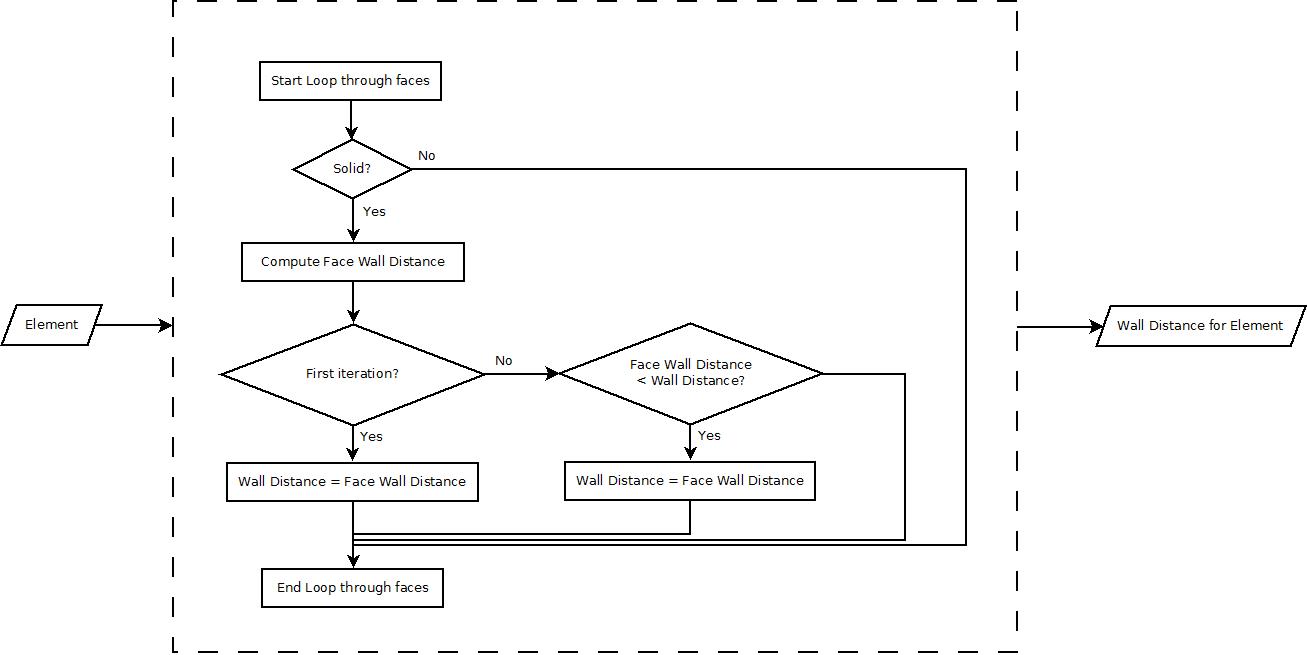


Figure 2: Brute force serial implementation

**II.A.1. Complexity**

The work and step complexities for the serial approach with *n* cell elements and *m* solid faces can be approximated as:

Work: O(*n* x *m*) – for each cell element, each solid face will have to be evaluated. So, total number of operations is in the order of *n* x *m*.

Step: O(*m*) – the cell elements can be evaluated in parallel, but the distance from each solid face to the element is evaluated in sequence. So, the longest chain of sequential dependencies is in the order of *m*.

**II.A.2. Parallelization**

The opportunity for parallelism arises in being able to perform the distance calculations and comparisons for each element and/or for each face separately. 2 parallel approaches were implemented for the brute force method:

1. Block-per-cell (Figure 3): Each block (CUDA block) calculates the wall distance for a cell. Each thread within a block calculates distance from a face; after all the threads have calculated the distance, a reduce minimum operation is performed to get the cell wall distance.

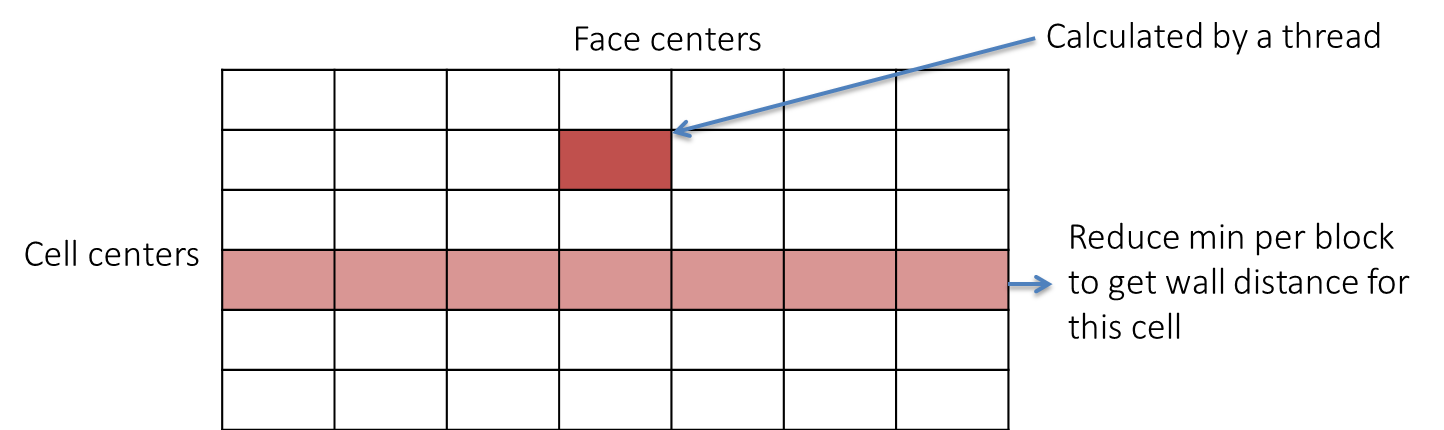


Figure 3: Brute force Block-per-cell implementation

1. Thread-per-cell (Figure 4): Each thread calculates wall distance for a cell. Each thread calculates distance from the cell to each of the faces, and keeps a “running” minimum distance value as each faces’ distance gets calculated.

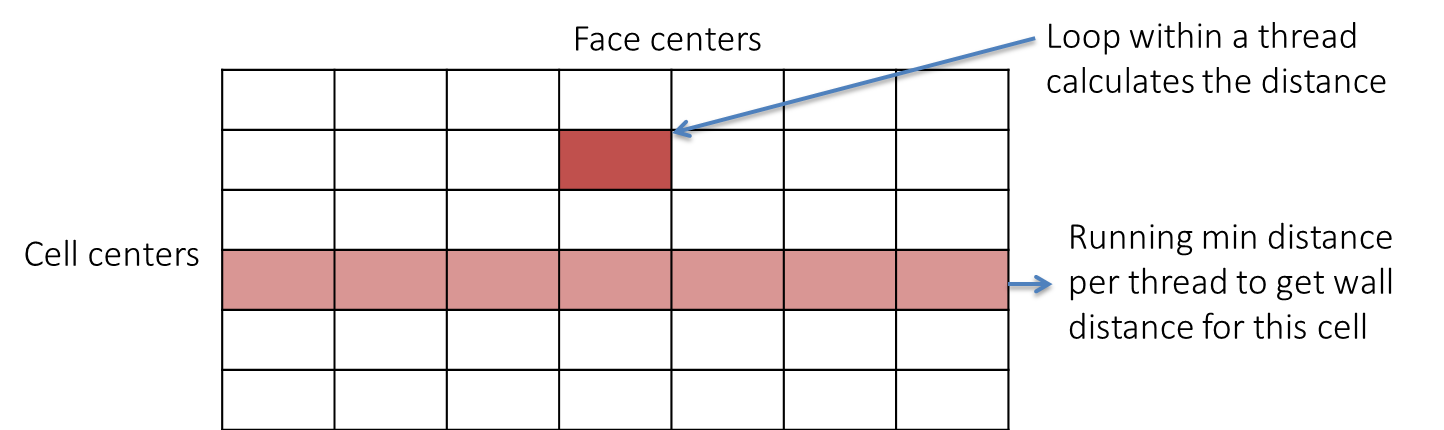


Figure 4: Brute force Thread-per-cell implementation

The work and step complexities for the parallel approaches with *n* cell elements and *m* solid faces can be approximated as:

Work: O(*n* x *m*) – work complexity for the 2 parallel implementations is similar to the serial implementation, as for each cell element, distance from each solid face will have to be evaluated. So, total number of operations is in the order of *n* x *m*.

Step: O(log2 *m*) for Block-per-cell, and O(*m*) for Thread-per-cell – In the Block-per-cell implementation, the wall distances for each face can be calculated in parallel in one step, and the reduce operation to compute the minimum wall distance has a step complexity of log2 m. Therefore, step complexity for this parallel implementation is in order of log2 m. In the Thread-per-cell implementation, each of the cell elements can be evaluated in parallel in a separate thread, but the distance from each solid face to the element is evaluated in sequence inside each thread. So, the longest chain of sequential dependencies is in the order of *m*.

III. Application performance and results

**III.4. Brute-force algorithm**

Several implementations of the 2 brute force parallel methods were tested in order to evaluate efficient memory usage such as coalesced memory access, use of available shared memory and thread local memory.

3 variations of the Block-per-cell method were implemented:

1. 1A: Shared memory writes for face distances – since a reduce operation per block is performed to get the minimum distance from an array of distances calculated by the threads, the distance array is stored in shared memory.
2. 1B: (1A) + Shared memory reads to face (x, y) arrays – storing the face (x, y) arrays in shared memory, so that every thread does not have to read all of the face (x, y) arrays from global memory.
3. 1C: Same as (1B) with (x, y) arrays converted to an interspersed array for coalesced shared memory reads, i.e. x0, y0, x1, y1, ....

3 variations of the Thread-per-cell method were implemented:

1. 2A: Each thread calculates wall distance for a cell
2. 2B: Shared memory reads to face (x, y) arrays
3. 2C: Same as (2A) with (x, y) arrays converted to an interspersed array for coalesced global memory reads, i.e. x0, y0, x1, y1, ....

The performance of the different brute force parallel methods is shown in Table 1. The times shown are based on several sample runs of the algorithm on the Ohio Supercomputer Center (OSC) Oakley cluster.

Table 1: Brute force algorithm performance

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Small Grid**  **(msec)** | **Fine Grid**  **(msec)** | **Extra Fine Grid**  **(msec)** | **Speedup (extra fine grid)**  **vs. Serial** |
| Serial | 33 | 595 | 13636 | Not applicable |
| Block-per-cell (1A)  - Shared memory writes | 1 | 28 | - | - |
| Block-per-cell (1B)  - Shared memory reads | 1 | 30 | - | - |
| Block-per-cell (1C)  - Coalesced shared memory reads | 1 | 31 | - | - |
| Thread-per-cell(2A) | <1 | 7 | 173 | 79 |
| Thread-per-cell(2B)  - Shared memory reads | 1 | 9 | 293 | 46 |
| Thread-per-cell(2C)  - Coalesced global memory reads | <1 | 7 | 166 | 82 |

The Block-per-cell methods do not support grid sizes with a large number of faces, as these methods require the number of faces to be less than the maximum number of threads available per block (1024 for the OSC cluster). Therefore, performance could not be measured for the extra fine grid. Also, the overhead of setting up shared memory and coalesced reads outweighs the performance gains for the small/fine grid sizes (1B and 1C). Therefore, performance for all three Block-per-cell methods was similar for the small/fine grid sizes.

The Thread-per-cell methods performed better than the Block-per-cell methods. Also, the Thread-per-cell methods do not have limitations on the grid size, and can accommodate any grid size. Thread-per-cell is not a good algorithm for utilizing shared memory, since each thread has to calculate the wall distance for a cell. However, this method (2B) was implemented in order to access if there is a performance gain from one thread in a block reading the face values and storing them in shared memory, instead of all threads reading them from shared memory. As expected, this caused a negative impact on performance due to thread divergence, as one thread had to perform significantly large amount of work compared to all the other threads in the block. The coalesced global memory reads made a slight improvement in performance, since reading the blocks in a coalesced manner reduces the memory access time. Therefore, the Thread-per-cell method with coalesced global memory reads (2C) was the optimal brute force algorithm implemented in this project, with a speed-up of 82 compared to the serial method.

For future improvements, a combination of the Thread-per-cell and Block-per-cell methods can be used in order to achieve the best possible performance (as close to O(log2 *m*) as possible). This can be done by separating out the 2 main calculations into 2 steps: calculating the face distances from the cell centers in one kernel, with each thread calculating a distance (i.e. “thread-per-cell face distance”); and then assigning blocks so that each block performs a reduce minimum on the face distances related to a cell (i.e. “block-per-cell reduce). This will require a block/thread sizing algorithm that can handle the different grid sizes. Another future improvement that should be explored is to break up the grid calculations into tiles, as that helps with efficient memory reads.

**IV.B. Self-assessment: Vasanth Ganapathy**

I was responsible for the “Brute Force” algorithm development, implementation, and testing. This includes the serial and parallel methods for the “Brute Force” algorithm.