GPU acceleration of wall distance calculation for computational fluid dynamics codes

Nathan A. Wukie*
University of Cincinnati, Cincinnati, OH, 45221, USA

Chris Park†, Vasanth Ganapathy‡
GE Aviation, Cincinnati, OH, 45215 USA

I. Introduction

Computational fluid dynamics(CFD) is a branch of simulation sciences that focuses on predicting fluid flows based on a set of governing physical equations. There are many different methods for accomplishing this that span both, different sets of governing equations, in addition to different numerical methods for computing a solution. Some different sets of equations that can be solved include the potential flow equation, streamfunction-vorticity equations, Euler equations, and Navier-Stokes equations. Analytical solutions to these sets of equations are often not available except in some extremely simple cases. Interesting geometries require a discretized approach. Some common discretization methods include Finite Difference, Finite Volume, and Finite Element methods.

This research effort is focused on a problem for a subset of those methods. In particular, we are interested in efficient wall distance calculation algorithms to support Reynolds-Averaged Navier-Stokes(RANS) solvers using a cell-centered, Finite Volume discretization with a moving mesh capability.

I.A. Background

I.A.1. Reynolds-Averaged Navier-Stokes Equations

The Reynolds-Averaged Navier-Stokes(RANS) equations govern viscous, laminar and turbulent flows. For turbulent flows, the turbulence is accounted for via a turbulence model, which usually takes the form of an additional partial differential equation or sometimes a set of partial differential equations that solve for turbulent working variables.

One general method for formulating these turbulence models is to include "production" and "destruction" terms for the turbulent working variables. In this way, phenomena that typically increase turbulence, such as vorticity, will "produce" a turbulence effect by increasing the turbulent working variable. One of these variables for turbulence production/destruction is the distance of a grid cell to the nearest solid wall; simply known as the wall distance, d_{wall}.

I.A.2. Cell-Centered, Finite Volume Discretization

The numerical method that this investigation directly supports is a cell-centered, finite volume discretization, although it could be readily extended to other methods with some modifications. The finite volume method relies on an integral form of the governing equations, and the method we are interested in stores all variables at the center of each computational cell. The values in each cell can then be updated, based on boundary fluxes computed at the interface of each computational cell. The value of interest here is the d_{wall} variable required by the turbulence model.

*Graduate student researcher, School of Aerospace Systems, ML 70, Cincinnati, OH, AIAA Student Member.
†Software Engineer, GE Aviation
‡Lead Engineer, GE Aviation

I.A.3. Moving Mesh Calculation

Some investigations that use fluid simulations are interested in moving body problems such as store separation for aircraft, and rotating machinery, among others. One way to accomplish this is to use body-local grids that overlap a background grid as shown in Figure 1. In this way, the body grid can move ontop of the background mesh to facilitate body motion. This usually works by computing an iteration for the CFD code, computing the force on the body of interest, computing the distance moved, and updating coordinates of the body-local grid.

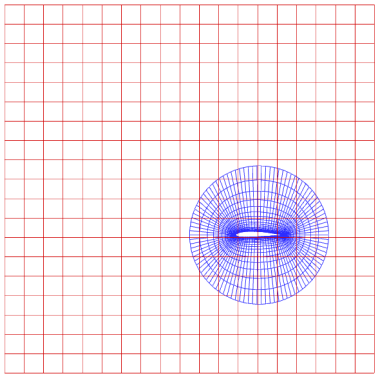

Figure 1. Chimera, overlapping grid for moving airfoil simulation

I.A.4. Motivation for efficient wall distance computation

Traditionally, the wall distance calculation would be of little importance from an efficiency perspective. In stationary grid simulations, the wall distance does not change, and so it would only be performed once as a pre-processing step, which would represent a trivial amount of time compared to the overall calculation. For moving body simulations however, the wall distance can change after each update of the body position and thus has to be recomputed for every iteration. It becomes extremely important then, to have an efficient method for recomuting the wall distance in order to maintain reasonable computation times. Using graphics processing units(GPU's) to accelerate the wall distance computation is seen as a promising method for improving computational efficiency for moving body problems.

II. Application-level objectives

The goal for this project is to implement a parallel algorithm for computing the wall distance for a computational grid on a GPU. The benefit of implementing this portion of the CFD code will be improved efficiency for moving body CFD calculations.

Primary objectives:

- Serial brute force wall distance calculation
- Parallel implementation of brute force method

Secondary objectives:

- Serial implementation of advancing boundary algorithm
- Parallel implementation of advancing boundary algorithm

III. Design overview

III.A. Preprocessor

A preprocessor will be developed in order handle reading the computationsl grid and formatting the data into an efficient manner for use by the wall distance calculation algorithms. This module can be seen in

block diagram form in Figure 2.

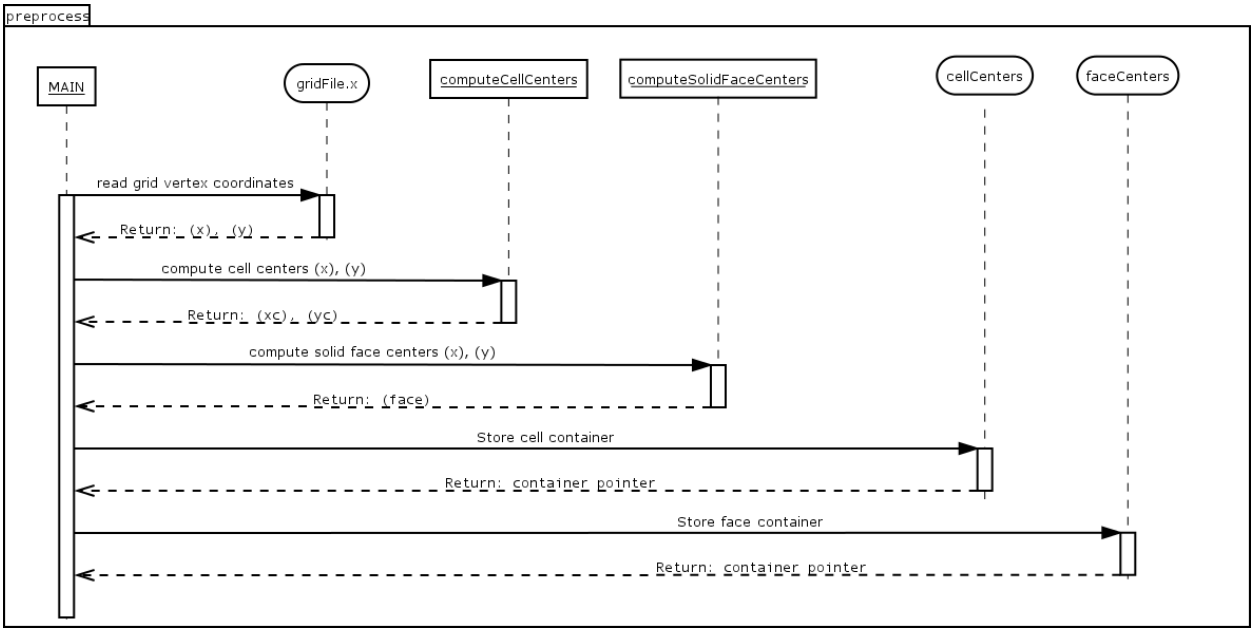

Figure 2. Block diagram for data preprocessor

III.B. Brute force algorithm

III.B.1. Serial

A serial brute-force implementation for calculating the wall distance for an element is shown in Figure 3. For each element, a large number of faces would have to be evaluated to check if the face is solid, to calculate its distance from the wall, and then to update the wall distance if this face has the least distance. The opportunity for parallelism arises in being able to perform the checks and wall distance calculation for each face separately.

For each face:

- Wall distance from the face will be calculated
- b. The minimum wall distance will be updated if the face wall distance is lower than the current minimum wall distance

In the worst case, work and step complexities for the serial approach with n faces would be:

- W n*m = O(mn) for each element, each solid face will have to be evaluated. So, total number of operations is O(mn)
- S m = O(m) all the 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 O(m).

III.B.2. Parallel

Two simple parallel implementations for the brute-force method will be explored. The first parallel implementation for calculating the wall distance for an element is shown in Figure 4. For each element, the host sets up block and thread sizes, and then launches a device kernel. The device kernel will assign a thread for each face, and then compute the wall distance for that face. The minimum wall distance operation will be computed using atomic min so that only one thread is reading and updating the overall wall distance at a given time.

The work complexity for this parallel implementation is similar to the serial implementation:

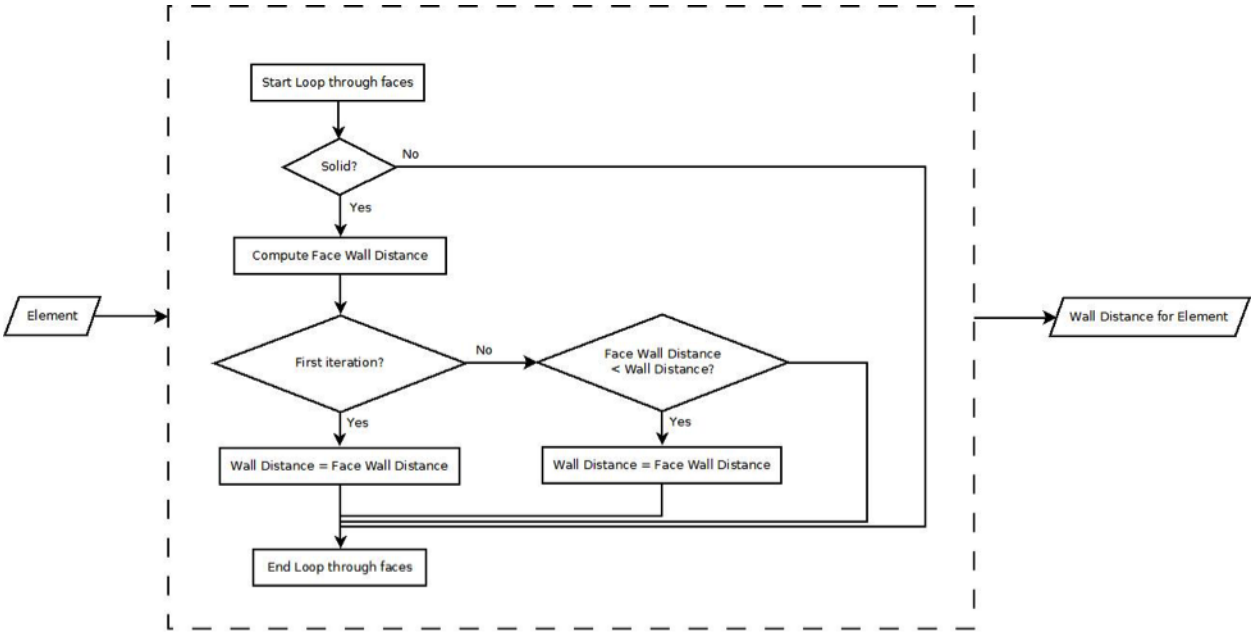

Figure 3. Serial, brute force algorithm

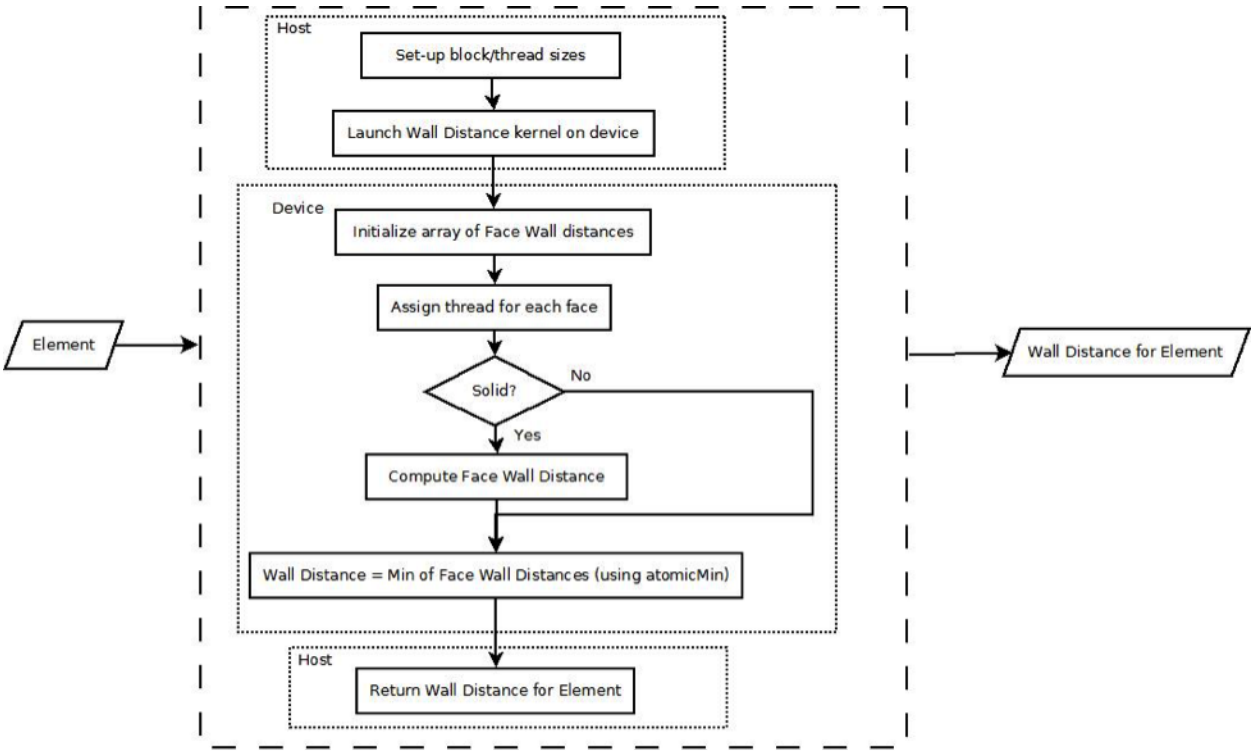

Figure 4. Parallel, brute force algorithm #1

- W n*m = O(mn).

It is slightly higher than the serial as there is additional overhead for assigning threads and initializing face wall distance in the parallel implementation. Assuming infinite threads, the wall distances for each solid face can be calculated in parallel in one step. However, the atomic min operation will limit the amount of parallelism that can be performed, as each thread may have to wait a long time before updating the minimum wall distance. Though the step complexity for this parallel implementation can be improved by dividing the atomic operations to be within blocks, and then combining the results from the blocks, it will not be significantly better than the serial implementation:

- S m = O(m)

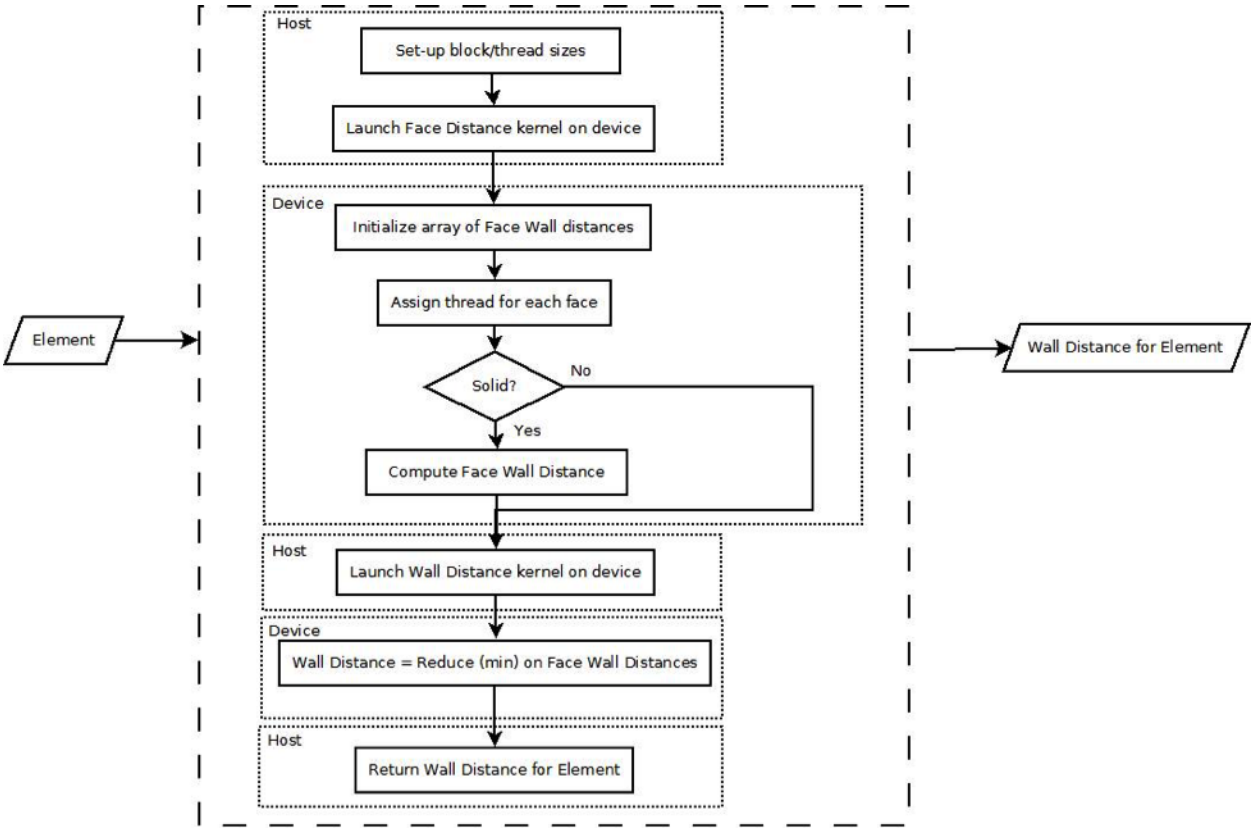

Figure 5. Parallel, brute force algorithm #2

The second parallel implementation for calculating the wall distance for an element is shown in Figure 5. For each element, the host sets up block and thread sizes, and then launches device kernels. The first kernel will assign a thread for each face, and then compute the wall distance for that face (same as the first parallel implementation). However, after this kernel is executed, a reduce (minimum) operation will be performed to calculate the overall wall distance for the element, instead of using an atomic operation.

The work complexity for this parallel implementation is similar to the first parallel (and serial) implementation:

- W n*m = O(mn)

However, the step complexity is much lower. Assuming infinite threads, 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 log m. Therefore, step complexity for this parallel implementation is:

- S(n) log m = O(log m).

For example, if we have 65,536 solid faces, the second parallel implementation can compute the wall distance in 16 steps, compared to 65,536 steps in the serial implementation.

III.C. Advancing boundary algorithm

The second method for computing the wall distance field for this investigation is an advancing boundary algorithm that relies on creating an auxiliary grid, which is used to identify a smaller subset of the original set of solid faces that can be sorted more efficiently.

This method consists of two parts. First, a preprocessing step will be used to compute an auxiliary grid overtop of the solid geometry. The auxiliary grid will then be compacted to include only those cells which intersect the solid boundary. The resulting auxiliary grid cells should contain multiple geometry faces per cell in order to improve efficiency. This auxiliary grid must would only be computed once at the beginning of a CFD calculation, and so we are not concerned with the efficiency of this step. A diagram of this process can be seen in Figure 6 but we are not concerned about parallelizing this module.

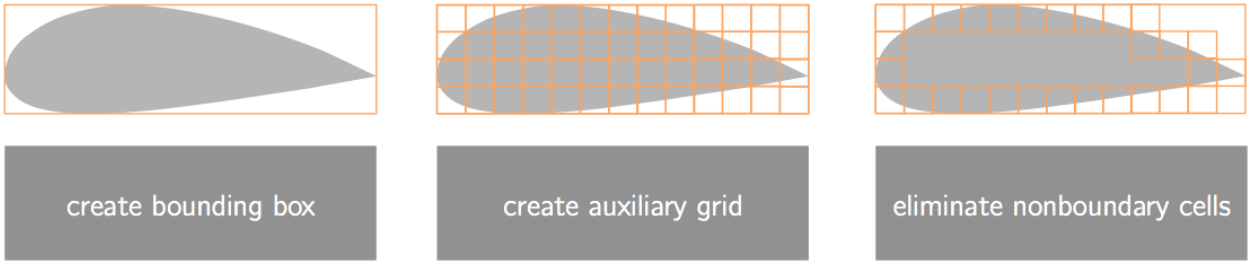

Figure 6. Advancing boundary method: preprocessing step

Once the preprocessing step is completed, the reduced auxiliary grid is used as the input to the wall distance calculation. This process, which was described above, can be seen in diagram form in Figure 7 and is used for each compuational cell in the grid.

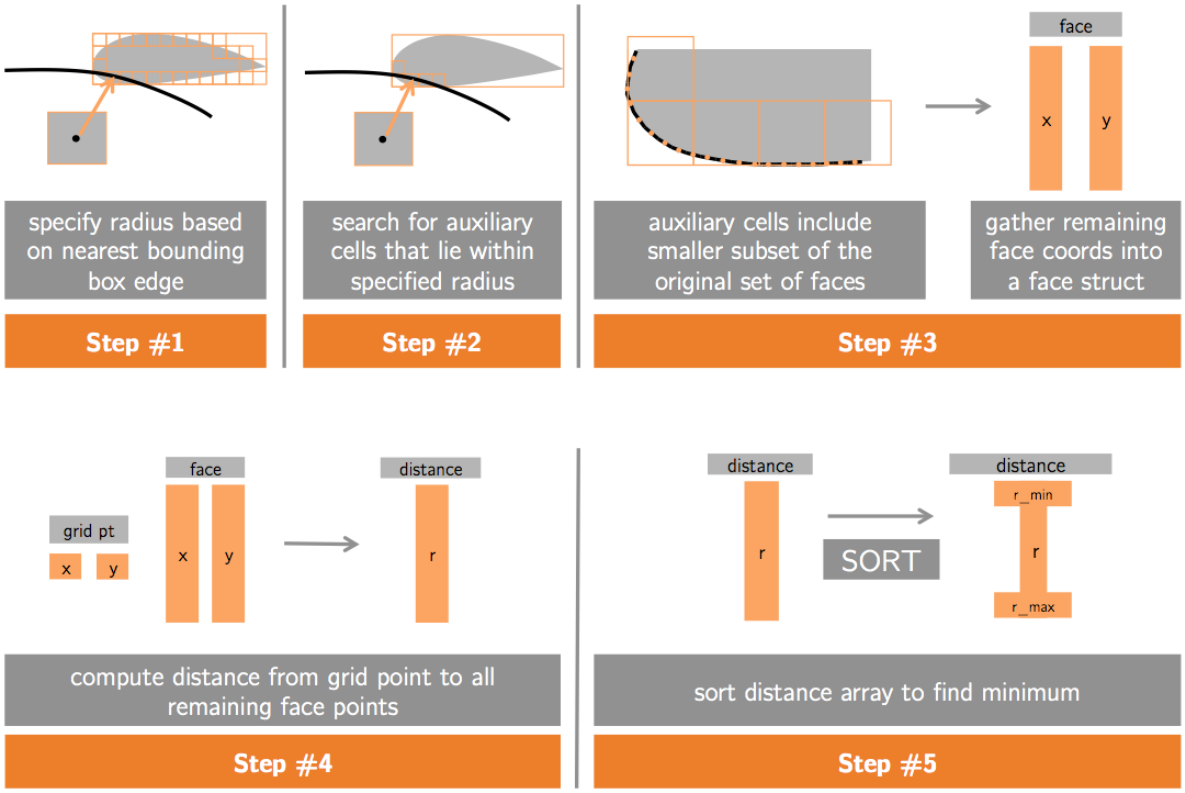

Figure 7. Advancing boundary method: wall distance calculation

III.C.1. Parallelization

There are several opportunities for potential gains in efficiency by parallelizing certain portions of this algorithm. These opportunities are outlined in the list below:

- Step #2: Implement compact kernel to search for auxiliary cells that lie within specified radius. Returns array of remaining auxiliary cells.
- Step #4: Implement map kernel to compute remaining distance values. Returns array of wall distances.
- Step #5: Implement sort kernel to find minimum wall distance. Returns minimum wall distance.

.

A block diagram of the algorithm is shown in Figure 8.

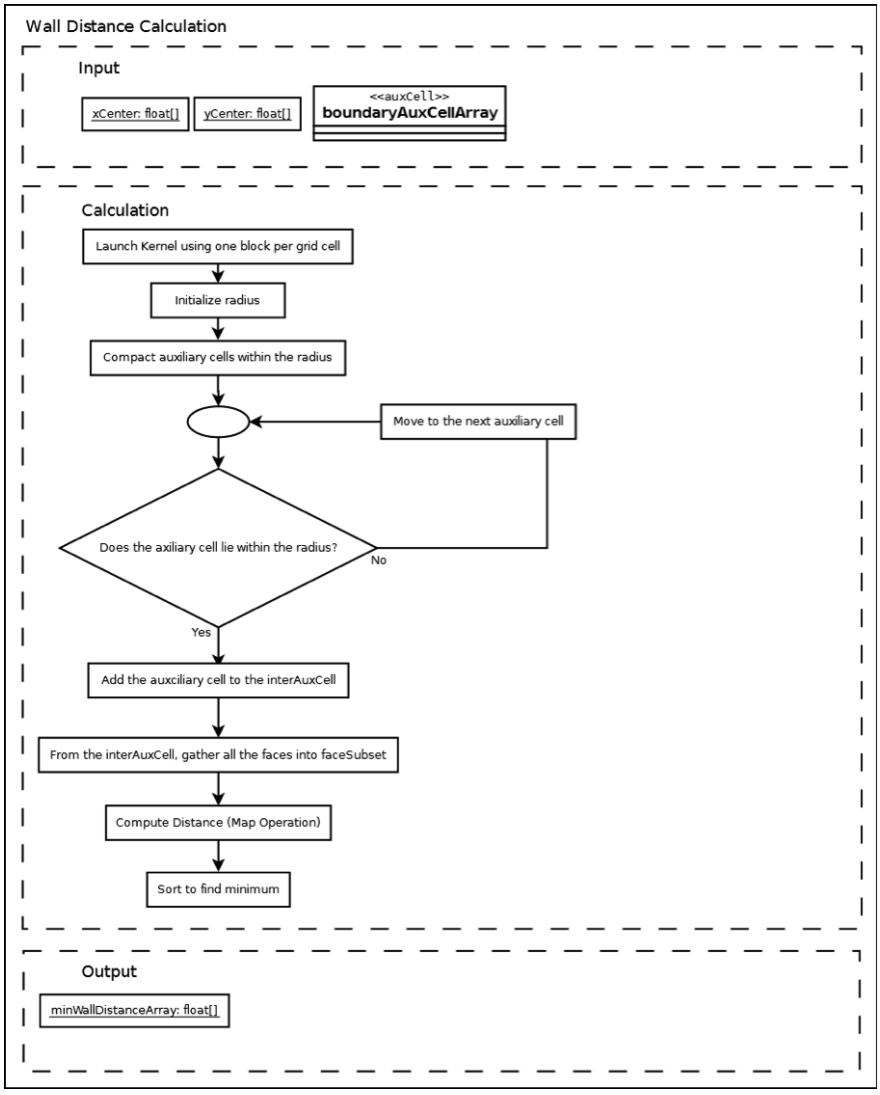

Figure 8. Auxiliary grid method: block diagram

IV. Performance goals

IV.A. Brute force method

IV.B. Advancing boundary method

The advancing boundary method is largely experimental and it is unclear at the moment if this method will improve efficiency due to more kernel calls that would have to be executed. The efficiency will also be dependent on the size of the problem and the ratio of solid faces to grid cells. A study will be performed to evaluate the trade-off between problem size and computational efficiency.

IV.C. Verification

The verification of each algorithm will follow a visual method, similar to the process used by the Udacity course. The wall distance field variables will be stored to a file that can be opened along with the computational grid to view the wall distance field. A visual inspection is expected to be sufficient verification as any errors will be easily detected.

V. Schedule and division of work

Figure 9 shows the proposed development schedule for this effort along with the tentative work split between the team contributors.

Algorithm development schedule					
	November				December
Task	W1	W2	W3	W4	W1
Pre-processor development	Nathan				
Brute force algorithm					
- Serial	Vasanth				
- Parallel		Vasanth	Vasanth		
Advancing boundary algorithm					
- Serial	Chris				
- Parallel		Chris/Nathan	Chris/Nathan	Chris/Nathan	
Final report					All

Figure 9. Development schedule and division of work