Performance and Accuracy Assessment of Line Marching Algorithm Computations Utilizing GPUs Within a Predictive GNSS Quality Service

Julian Gutierrez*
NASA Langley Research Center, Hampton, Virginia, 23666

David Kaeli†
Northeastern University, Boston, Massachusetts, 02115

Evan Dill‡
NASA Langley Research Center, Hampton, Virginia, 23666

Pau Closas§
Northeastern University, Boston, Massachusetts, 02115

This paper presents a detailed analysis of the accuracy and performance of line marching algorithms executing on a GPU. In the context of an accurate Global Navigation Satellite System (GNSS) quality of service simulation, horizon sky-plots are a useful tool to determine satellite visibility in the presence of obstructions from objects, such as buildings or dense foliage. In order to accurately model satellite visibility at a point of interest on a map, a horizon plot can identify the viewing angles at which objects are blocking the sky. This computation requires traversing a line starting at the point of interest on a 2D altitude map, moving outward for every azimuth angle. To explore the performance of this computation, we propose a new dynamic stopping condition for the traversal of the line, benefiting from objects close to the point of interest. We compare the accuracy of common line marching algorithms, and consider their parallel performance when developed in CUDA. We find that our proposed stopping condition for line marching provides a significant improvement in performance in urban canyon sky-plots, as compared to previous work. Additionally, these results show that simpler algorithms, such as the digital differential analyzer line algorithm, are better suited for GPUs than more sophisticated schemes such as Bresenham's algorithm, specifically in the context of sky-plot horizon computations. The trade-off between accuracy and performance is analyzed and guidelines are provided that depend on the targeted goal of the GNSS application.

I. Nomenclature

DDA	=	Digital Differential Analyzer
DSM	=	Digital Surface Model
DOP	=	Dilution Of Precision
DSC	=	Direct Scan Conversion
$GNSS$	=	Global Navigation Satellite Systems
GPS	=	Global Position System
GPU	=	Graphics Processing Unit
$LIDAR$	=	Light Detection and Ranging
LOS	=	Line Of Sight
POI	=	Point Of Interest
SV	=	Space Vehicle

*Research Engineer, Safety-Critical Avionics Systems Branch
†Distinguished Professor, Electrical and Computer Engineering Department
‡Research Engineer, Safety-Critical Avionics Systems Branch
§Assistant Professor, Electrical and Computer Engineering Department

II. Introduction

GLOBAL navigation satellite systems (GNSSs) play a key role in a variety of applications including vehicle navigation, surveying, environmental positioning, and monitoring systems [1–3]. Thanks to services such as Google Maps, Uber, Doordash and many more, our dependency on GNSS systems in our lives is increasing each day [4]. The effectiveness of this technology is strongly dependent on the proper reception of data from at least four satellites, as well as the position of those satellites in the sky. However, GNSS techniques have limitations [5, 6]. To ensure good quality of the solution from these systems, an unobstructed view of the sky is desirable. Relying on multiple satellites in line-of-sight (LOS), in order to compute an accurate location of the receiver, can become difficult to achieve when terrain or man-made obstacles (e.g., buildings in urban scenarios), obstruct large regions of the sky [7].

A simple solution to address this problem is to combine multiple GNSS systems, such as GPS and GLONASS (Globalnaya Navigazionnaya Sputnikovaya Sistema: a GNSS owned by the Russian Federation). Increasing the number of satellites in the visible sky increases the chances of providing an accurate solution. Despite this, availability can still be limited in urban canyons inside dense cities. Understanding the implications of these limitations is key to improving the reliability of this technology. The use of simulation environments to better understand these scenarios can help predict locations that suffer from low quality signals and provide solutions such as Shadow Matching [8].

Sky-plots provide a useful tool in diagnosing problems encountered when processing GPS data [9]. Sky-plots are simple visualization tools that display GPS satellite trajectories, referenced to a specific position on the ground. They provide an intuitive feel for satellite movement patterns, as well as reveal the impact of obstructions on satellite visibility, if computations can be completed to determine LOS obstructions from known Digital Surface Models (DSM). For this reason, we explore the accuracy and performance needed to adequately compute sky-plots representative of the observability of GNSS signals in a given environment.

As part of this work, we have explored application optimizations by deploying Graphics Processing Units (GPUs) wherever possible. GPUs can provide for significant acceleration of applications and so are becoming deployed across an ever-growing range of applications [10]. GPUs provide massive parallelism by leveraging a large number of compute cores and high memory bandwidth. Given the inherent parallelism in the horizon plot computation, GPUs are an effective path to achieving real-time performance whenever it is needed by the application. In this paper we explore how accurate horizon sky-plots can be computed using line marching algorithms in conjunction with Digital Surface Models (DSMs). We evaluate various popular line marching algorithms, including the digital differential analyzer (DDA), direct scan conversion (DSC) and Bresenham algorithm. We consider their accuracy as a function of the angle used to traverse the map, as well as the performance observed when used within sky-plot generation. We additionally provide two variations of the DDA algorithm: i) max-pooling and ii) a dynamic DDA.

Section III discusses the algorithmic details necessary to generate horizon sky-plots. Section IV describe the flow through each of the algorithms considered in this work. We consider implementation details that will impact our use of GPUs in Section V. Section VI provides results on the accuracy of each algorithm and the resulting performance observed on a GPU. Finally, we summarize our findings in Section VII.

III. Horizon Sky-plots

In the realm of GNSS systems, sky-plots are a tool used to indicate the location of satellites in the sky, commonly referred to as space vehicles (SVs). Sky-plots are also used to map the visibility of the sky, as defined by objects in the surroundings of a location. This trace or line, which defines the visibility of the sky, is commonly referenced as the horizon. Natural and anthropogenic obstacles such as buildings and trees can dramatically impact sky visibility. These obstructions, in turn, can reduce the accuracy of the GNSS solution, especially if satellites are obscured by these obstacles. Despite the widespread availability of GNSS systems, this issue is prevalent in many low altitude environments, such as dense urban areas where the view of the sky is often blocked by buildings.

Additionally, SVs close to the natural horizon (usually at elevation angles $\leq 10^o$) are less helpful in resolving the GNSS position due to additional atmospheric distortions, high blockage probabilities, or higher chances of being affected by multipath propagation. These are effects that, essentially, reduce the signal-to-noise ratio of the corresponding SVs, ultimately impacting the achievable positioning error. This angle is commonly referenced as elevation mask angle. SVs under this elevation mask angle are usually discarded from computations to reduce the errors while computing the final position. If there are too few satellites present, the receiver will be unable to resolve its position. Sky-plots can be an invaluable tool to help monitor the current satellite configuration and predict how the environment might interfere with the solution provided by the receiver in the scenario. This is especially true whenever obstacles are blocking the line of

sight (LOS) to a satellite.

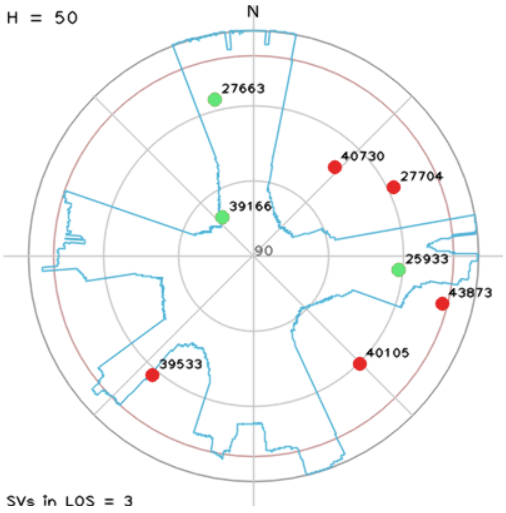


Fig. 1 Example of a sky-plot showing obstructed (in red) and unobstructed (in green) satellites in the sky.

Figure 1 shows an example of a sky-plot taken from the center of a busy intersection in Boston, MA, at a height of 50 meters from the ground. The outer red circle represents the elevation mask angle (10 degrees). The blue line represents the horizon, a boundary that includes the surrounding buildings that are blocking the view of the sky. All of the SVs marked in red are outside the horizon, which means they are blocked by an object or below the elevation mask angle. The SVs marked in green are inside the horizon, which indicates we have a direct line of sight to these satellites.

Many applications require an accurate representation of the horizon boundary. For example, simulation frameworks used to predict satellite trajectories and quality of the signal observed by a receiver are based on a metric, commonly referred to as the dilution of precision (DOP) [11]. Dilution of precision is a common calculation used in GNSS applications. It serves as a means to quantify the errors present in a calculated position solution, as a function of satellite geometry. A slight variation in the horizon mapped by the tool (e.g., a border along the side of a building) can impact the DOP predicted by the tool and result in a large difference observed by the receiver in the location.

To achieve an accurate representation of the horizon sky-plot, we require the following:

1) An accurate (and up to date) digital surface model (DSM) representing the terrain altitude (e.g., buildings, trees, mountains, etc.) observed at different locations within an area or map,
2) an accurate prediction of the trajectories and locations of the SVs in the sky, and
3) an accurate way to traverse the map to obtain the elevation angles at which the sky is visible.

DSM models can be obtained starting from light detection and ranging (LIDAR) [12] data sets. For this work, we convert the LIDAR point clouds into a GeoTIFF raster file, which can then be used in common GIS tools. When generating these types of files, it is beneficial to have a common procedure to determine the highest point in each grid cell. One possible way to accomplish this is to perform max-pooling on the points present in each defined cell. This measurement provides a worst-case scenario for the blocking capability of the measured surface (e.g., a building or other obstacles) [6]. These models can result in accurate representations of the buildings/trees in the map, but are commonly dated and may result in inaccurate predictions if changes have occurred within the environment. This issue with databases is particularly common, especially with publicly available databases. As a result, the assumption is that the DSM used for the analysis is sufficiently accurate to adequately represent the environment of interest.

On the other hand, there is a wealth of open source software and APIs available to accurately predict the trajectory and location of satellites in the sky, when a latitude-longitude coordinate is provided. We consider the software used in the prediction of the SV locations to be sufficiently precise, and therefore, potential satellite location errors are not considered within the scope of this work. Our work aims to analyze the accuracy of the elevation angles for objects within the DSM, allowing us to compute the horizon boundary as described above in (3).

A. Algorithm Description

The algorithm used to create accurate sky-plots is a result of: 1) selecting a location for the analysis, commonly referred to as the point of interest (POI) on the map, 2) computing the SV locations in the sky, and 3) dividing the

azimuth space into small angles (with sufficient precision) and tracing a line starting at the POI in the direction of each angle. At each point along the line, we compute the maximum elevation angle formed between the POI and the current location along the line, based on the altitude where the grid cell intersects the line.

B. Stopping Condition

A common way to improve the performance of this algorithm is to limit the distance needed to travel along the line. One way to accomplish this is to implement a stopping condition that limits the total distance traveled, while also reducing the accumulation of errors in terms of floating point precision. Given a two dimensional DSM and a maximum height (h_{max}), Gandolfi et al. [2] suggests setting a limit on the maximum distance needed to travel from the point of interest on a surface, based on the elevation mask angle. The underlying concept of this technique is that any object past this distance should not affect the SV's availability, as satellites in this range will already be under the elevation mask angle, and therefore not considered in LOS. This distance is given by the following equation:

$$d_{max} = (h_{max} - h_p)/\tan(\phi_m) \tag{1}$$

where h_{max} is the maximum height of the DSM, h_p is the height of the point of interest and ϕ_m is the elevation mask angle. This gives us a limit to the maximum distance needed to travel to ensure SVs are not blocked by objects described within the DSM. This equation gives a finite circumference around the point of interest, after which any additional line traversal will result in unnecessary computations. This stopping condition will be referenced as the static stopping condition.

This paper presents a new dynamic stopping condition that can drastically improve performance in most scenarios by dynamically adjusting the maximum distance traversed along the line, based on the maximum elevation angle observed at the POI. Figure 2 shows examples of how the dynamic stopping condition affects the maximum distance needed to travel. In this figure, lines are traversed starting at the POI, and moving along the line towards the right, while measuring the height of the different obstacles intersecting the projection line. In figure 2(a) we can observe the angle a_{B1} formed by the difference in elevation between B_1 and the POI, as well as the distance d_{B1} between the POI and B_1. The ϕ_e angle is projected to provide us with a maximum distance at which the height of the tallest building in the DSM (marked in green) would cause this same angle, given by the following equations:

$$\phi_e = \max(\phi_c, \phi_e) \tag{2}$$

$$d_{max} = (h_{max} - h_p)/\tan(\phi_e) \tag{3}$$

where ϕ_e is the maximum elevation angle observed across the line and ϕ_c is the elevation angle observed at the current position on the line. As figure 2(b) suggests, if we traverse past this d_{max}, even the tallest building on the map would be unable to generate a higher elevation angle. This means it would be unable to block the sky view at the POI more than B_1. As is the case with B_3 in figure 2(b), where it is at a larger distance than d_{max}, which creates an angle a_{B3} smaller than a_{B1}.

A different scenario and possible building configuration is presented in Figure 2(c). Through analysis of the projected elevation angle formed by B_1, we find a d_{max}, as shown in figure 2(c). In this scenario, B_3 is at a closer distance than d_{max}. We continue the traversal of the line as shown in figure 2(d). Through the analysis of B_3, we can see that the elevation angle a_{B3} at the POI is larger than a_{B1}, so this is the new elevation angle we store. We then update d_{max}, where we can observe that this distance is now the same as the current traversed distance. Note that since B_3 is the tallest building on the map, we can therefore conclude the processing across this line.

IV. Line Marching Algorithms

In this section, we describe common line marching algorithms used in a diverse set of applications, including: i) ray tracing [13], ii) path planning [14], iii) occupancy grid maps for autonomous vehicles [15], etc. These algorithms are chosen as our baseline for this study. They were selected based on their popularity and their ability to perform grid traversals within a discretized DSM.

An important factor from the sky-plot horizon algorithm is the accurate computation of the distance between the POI and the current location along a projected line. This is required to correctly compute the elevation angle formed between the altitude measured from the DSM and the distance traveled across the line.

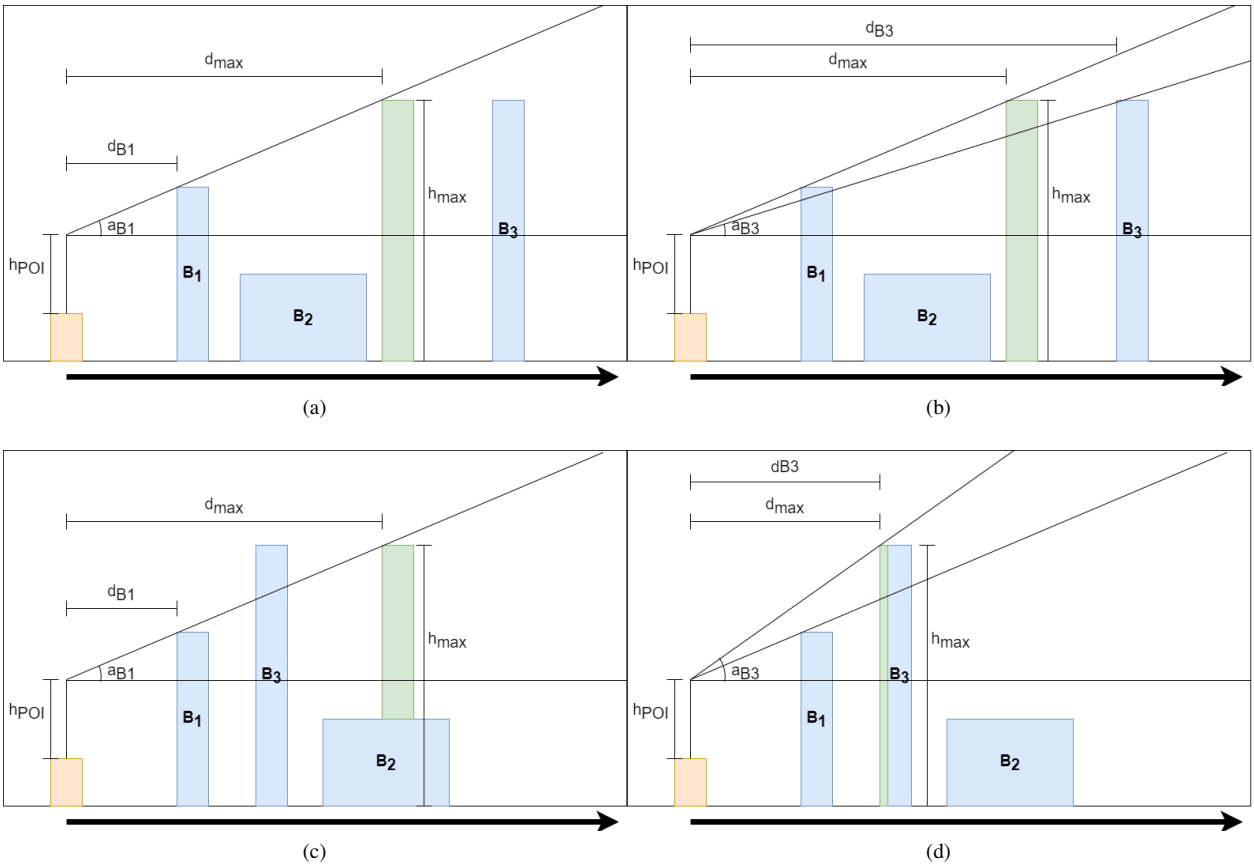


Fig. 2 Diagram of the maximum distance needed, based on the dynamic stopping condition.

A. Direct Scan Conversion (DSC)

DSC is one the most computationally simple line algorithms. Starting at the POI, a line is traversed by using the linear equation of a line at each step: $\hat{y} = m * \hat{x} + b$. Steps are taken along the $\hat{x}$ axis, and the equation is evaluated for each iterative step. This involves floating point multiplication operations, which can be expensive to compute. As such, this algorithm is usually not used in practice. Additionally, we can pre-compute the distance traveled at each step and update the total distance traveled by using a single floating point addition inside the loop.

B. Digital Differential Analyzer (DDA)

The DDA algorithm is a simple implementation that is used in a wide range of applications [16]. It is considered the line marching algorithm, as defined by Perlin et al. [17], and follows the equation:

$$l(k) = l_0 + \Delta t k l_d \tag{4}$$

Where l_0 is the starting point of the line, Δt is the predefined step size taken along the line, k is the step number, and l_d is the slope of the line. This can be translated into a two-dimensional grid, where the algorithm computes the size of the step taken along the x and y axes. In the main loop, computation is started at the POI and updated along the x and y values by increasing the step size for each axis. Similar to DSC, DDA pre-computes the distance traveled along the line at each step, so it only requires floating point addition. One of the drawbacks of this algorithm is that it suffers from rounding operations, and is therefore subject to cumulative errors due to using a floating point representation. This issue is not relevant in this study, as the magnitude remains well within the boundaries of the precision of single-precision floating point.

C. Bresenham

Developed by Bresenham in 1965 [18], this technique is a simplified implementation of the DSC algorithm, approximated by only integer arithmetic. The algorithm steps along the $\hat{x}$ axis and only increases in $\hat{y}$ when the distance between the actual line and the nearest grid intersection reaches the inflection point. This inflection point is easily pre-computed, requiring the loop to only perform simple integer arithmetic, resulting in fast computations. The basic implementation (for slopes between 0 and 1) requires minimal control flow operations. However, in order to achieve a robust implementation that can traverse lines in any direction, the algorithm's control flow complexity increases. It uses only integer addition, subtraction and bit shifting. All of these operations are computationally inexpensive operations when run on today's computer architectures. Another limiting factor is the inability to maintain an accurate distance approximation. For the purposes of this work, an additional costly computation at each iteration is required to compute the current distance between the point along the line and the POI. Finally, this algorithm can produce visually acceptable lines, however it does not traverse all the grid cells touched by the line [18]. Therefore, the Bresenham algorithm can result in incorrect computations for our sky-plot algorithm.

V. Application Performance Considerations

GPUs have many attractive computational characteristics and continue to exploit technology advances to provide more cores per chip, larger memory capacities and increased memory bandwidth. These improvements have fueled dramatic increases in GPU usage across a wide range of fields. Despite these technological advances, in order to exploit the full capabilities of a GPU, the programmer faces some challenges [19]. Given their Single Instruction Multiple Data (SIMD) programming model, and given their unique memory design (significantly different than on a CPU), a subset of algorithms can be easily mapped to GPUs. However, given the inherent nature of a number of applications, the task of mapping serial code to the parallel architecture of the GPU is challenging, and can result in lower performance versus execution on a CPU.

Applications that can naturally be divided in a number of independent parallel tasks are commonly referred to as *task parallel* or *embarrassingly parallel* applications. This class of applications lends itself to attain large performance benefits through the use of GPUs, as each parallel task can be handled by a thread on the GPU, and threads do not need to communicate or synchronize. The sky-plot application referenced in this paper is embarrassingly parallel, as the analysis of each angle under study is completely independent from the computations of other angles. Thus, we can divide the sky into fractions of an angle and analyze a line defined from the point of interest towards the azimuth angle in question. The analysis of each line can be easily executed in parallel threads on a GPU. Even though this results in multiple reads from the same point, the elevation angle observed at each azimuth angle can be accurately portrayed.

We use an Nvidia GTX 1080 in this study. CUDA [20] is used as the programming framework for the implementation, and the output data is displayed using OpenCV [21]. The data from the DSM is stored on a texture surface in the main memory of the GPU. This enables us to utilize a more efficient memory pipeline present in the 1080 GPU, resulting in improved 2D spatial locality due to the nature of the specialized texture cache. Our application exhibits high spatial locality at the beginning of the application, when threads retrieve grid cell values which are close to each other in the address space. This happens when threads are reading from points along the line which are close to the start point. The following table lists the algorithms that we analyze.

Table 1 Algorithm implementation acronyms.

Acronym	Algorithm
BRE	Bresenham
DDA	Digital Differential Analyzer
DDA-D	Dynamic Digital Differential Analyzer
DSC	Direct Scan Conversion
MP	Digital Differential Analyzer using Max-pooling

A. Dynamic DDA Implementation

The dynamic DDA implementation computes the necessary distance to reach the boundary of each cell that the line touches. This results in more computations at each step along the line. The main benefit of this implementation is that

the algorithm reads each cell that the line touches only once, and it is guaranteed to read all of the cells that the line touches.

B. Max-pooling Implementation

The Max-pooling implementation follows the same logical flow as the DDA algorithm, while using a step size of 1. In contrast to the other techniques explored, all of the four neighboring grid cells are used in the calculation, as opposed to just using the one closest to the current point along the line. This results in four reads during each step. The maximum difference between the values that are read is computed and used as the height observed at the current point along the line. Taking the largest difference found among the neighboring grid cells ensures a worst-case-scenario when analyzing the maximum elevation angle along the line. A significant performance impact would be observed using this method due to the added reads. One way to counter this impact is to leverage the GPU's texture memory, using the $tex2Dgather$ API call to retrieve the needed data in an efficient way. This can significantly reduce the overhead associated with the additional reads.

C. DSC and Bresenham Adaptation

Bresenham and DSC follow the standard line equation and step along the $\hat{x}$ axis of the equation. This means that x must have a slope less than 1, or else each step would increase y by a larger proportion, reducing the resolution of the line, and ultimately skipping the grid cells during analysis. There are two approaches to avoid this problem: 1) design specific control flow operations within the loop for each case (leveraging if/then/else statements), or 2) pre-processing the starting conditions of the algorithm and retaining the algorithm's original logic, without adding expensive control flow operations. This paper proposes an innovative solution to (2), which allows the technique to keep the integrity of the original algorithms intact. A pre-processing step is applied to the starting conditions of the algorithm. The standard line equation, $\hat{y} = m\hat{x} + b$, is mapped to either the x or y axis, with the axis selected based on the value of the azimuth angle. Angles closer to the x axis will use the standard form of the equation $\hat{y} = m\hat{x} + b$, while angles closer to the y axis will invert the axes in the equation, resulting in $\hat{x} = m\hat{y} + b$. For example, a line drawn at an azimuth angle of 10 degrees (with North as reference) is closest to the y axis, which means, the line equation $\hat{y} = m\hat{x} + b$ will use the y axis as $\hat{x}$. This modification results in relatively little computational overhead (as compared to the line traversal), while reducing complexity and avoiding branch divergence as the algorithms traverse the line.

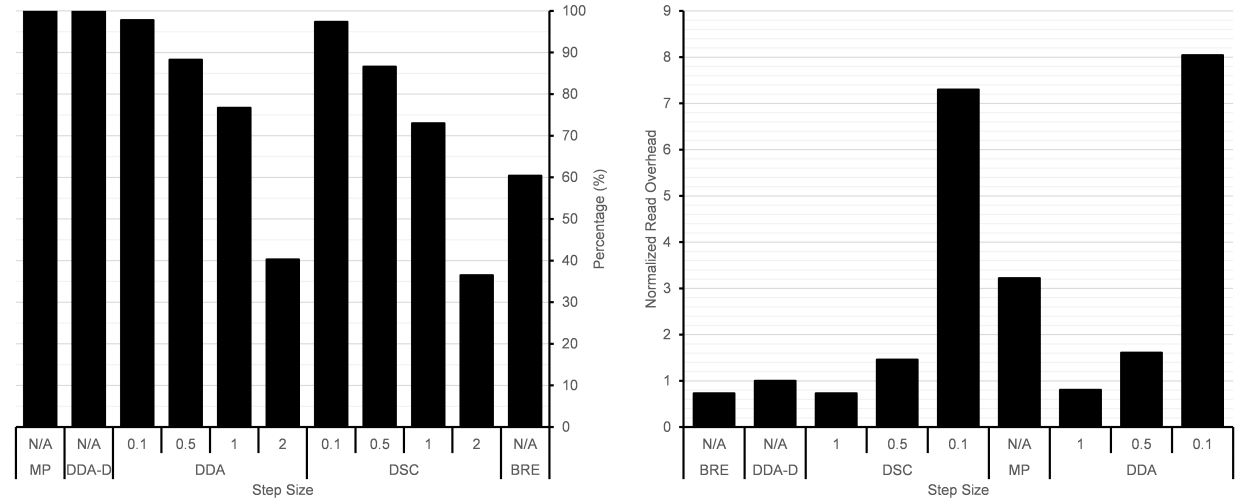
VI. Results

Next, we discuss the performance of the line marching algorithms when using our proposed optimizations executing on a GPU. We first analyze the accuracy of the algorithms using line coverage as the main metric. Second, we then compare the average execution time of the algorithm running on a GPU. Third, we compare our proposed dynamic stopping condition to the static stopping condition under various scenarios to provide evidence on the effectiveness of this solution for the skyplot application.

A. Line Coverage and Read Overhead

We define line coverage as the percentage of grid cells the algorithm touches, divided by the total number of grid cells the line touches (we will refer to this value as the ground truth). Line coverage is important here because it demonstrates how close to the actual line each algorithm is able to produce. We analyze each algorithm across different angles, ranging from 0 degrees to 90 degrees. Figure 3 (a) shows the average line coverage across all angles. As figure 3 (a) suggests, BRE has the lowest average line coverage at 61%. This is a result of taking the same number of steps, regardless of the angle. When the angle is close to 45 degrees, the total number of grid cells the ground truth touches increases, but BRE does not. DDA-D and MP both reached 100% coverage, regardless of the angle. This says that both solutions provide optimal coverage in terms of accurately portraying the values across the line, though MP does result in a significant increase in the number of grid cell values considered that are not part of the ground truth.

DDA and DSC both can vary the step size to improve line coverage. By decreasing the step size to 0.1 for both algorithms, we can improve the line coverage to over 96%. This comes with an additional overhead, as a number of unnecessary reads are now performed. Figure 3 (b) shows the average read overhead (number of reads done by the algorithm over the total number of pixels the ground truth intersects) across all angles. The step size can be reduced further, but the benefits are outweighed by the performance degradation from the additional reads, which will be discussed in the next section.

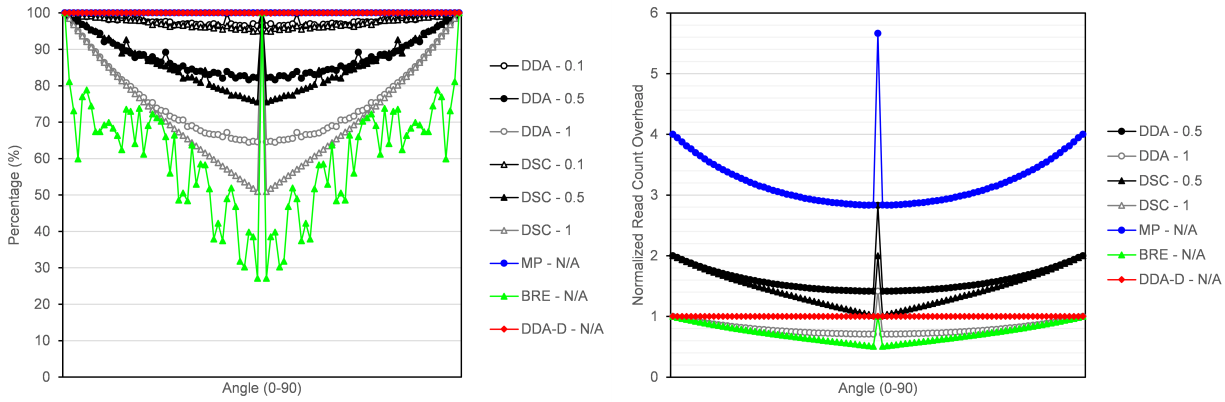


(a) Line Coverage

(b) Read Overhead

Fig. 3 Average line coverage (a) and read overhead (b) for the line algorithms, as compared to the ground truth.

Figure 3 (b) shows that BRE always achieves a lower read overhead versus the ground truth. BRE takes steps in integer values, and only takes the same finite number of steps each time. DDA-D reads the same number of pixels that the line touches, as it is supposed to capture the exact grid cells the line touches. Thus, DDA-D comes with no added overhead as compared to the ground truth. MP has the overhead of reading neighboring grid cells. This can have a severe impact on performance, but as stated before, it could serve as a worst-case scenario, given it covers a wider range of grid cells.



(a) Line Coverage

(b) Read Overhead

Fig. 4 Line coverage (a) and read overhead (b), as a function of the angle of the line.

Figure 4 (a) shows the line coverage as a function of the angle. This data is important, since we would like to analyze 360 degrees when running the sky-plot application. It was confirmed through analysis that the line coverage of the algorithms is cyclical in behavior, repeating every 90 degrees. As we can observe in figure 4 (a), we can also see the same pattern mirrored for half of a 90 degree section (i.e., the pattern repeats at 45 degrees). The total number of cells touched by the line increases when approaching 45 degrees. At exactly 45, the line no longer touches other cells, except the cells along the diagonal. Thus, the total number of cells required is lower. BRE always reads the same number of cells, regardless of the angle. Hence, BRE misses more cells the closer the angle is to 45. DDA and DSC have a similar

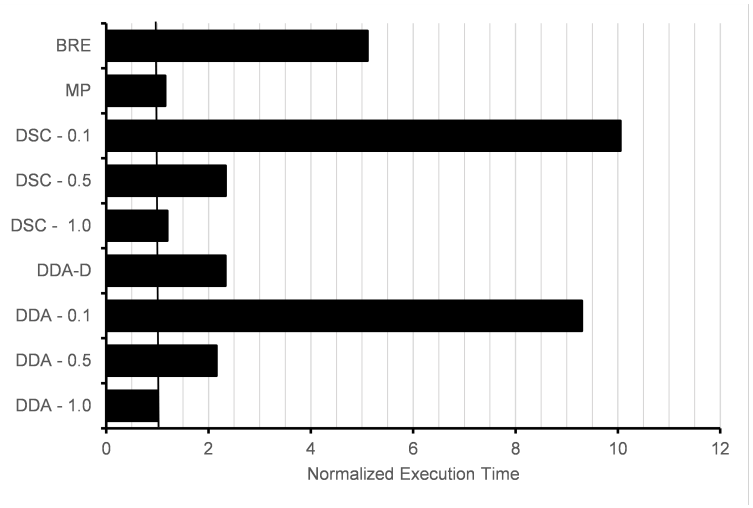


Fig. 5 Average normalized execution time of the line algorithms used for sky-plots on a GTX 1080 GPU (lower is better).

behavior, but the increase is lower than the ground truth. If we decrease the step size for these algorithms, they are able to achieve 95% the line coverage for all angles using a step size of 0.1.

As figure 4 (b) suggests, DDA-D has no read count overhead as compared to the ground truth, since DDA-D reads the exact cells the line touches. BRE's overhead is less than the overhead experienced with the ground truth as the angle approaches 45 degrees. This results in an equivalent loss of line coverage. Therefore, BRE does not provide an accurate representation of the line for this range of angles. All algorithms demonstrated a spike in the read count overhead for a 45 degree angle. This impulse response is a result of the total number of grid cells touched by the line. When approaching 45 degrees, the number of cells touched by the line increases. When we reach 45 degrees, the slope of the line is exactly 1, and the number of cells touched by the line decreases dramatically since it only touches those on the diagonal. For example, DDA-1.0 will continue to advance along the line at a step size of 1. Because the angle is 45 degrees, the total steps taken will increase by $\sqrt{2}$ times, which is the value observed at that spike.

B. Performance Analysis on GPUs

Figure 5 shows the normalized execution time of each algorithm compared to DDA-1.0 when running the sky-plot algorithm. The azimuth search space is discretized into 10,000 steps per degree, resulting in 3,600,000 angle measurements (and an equal number of GPU threads). The performance of the CUDA kernel is reported. A line is drawn for every step within the azimuth space, starting at the POI and ending when the stopping condition is met. A DSM of the Boston City is used as the reference. The coordinates are **42°21'32.2"N 71°03'24.4"W**, specifying the location for the analysis. Each performance test is executed 10 times and the average execution time reported.

In this scenario, the Bresenham technique does not perform well due to the distance computation at each step. The algorithm results in an execution time of 5 times the duration of DDA-1.0. Additionally, when we replace the distance computation with an approximation of the distance, this results in significant errors and incorrect projections of the sky-plot. For this reason, the distance computation is required and BRE cannot avoid the associated computational burden. 80% of BRE's compute utilization involve arithmetic operations (i.e., BRE is not memory bound). The instructions used to compute the distance traversed at each step to control the flow of the algorithm resulted in higher single-precision usage versus integer arithmetic.

The observed performance difference between MP and DDA is minimal. A much higher texture read count was observed with MP, but performance was not affected given our use of an optimized text2dgather operation. This also resulted in a proportional increase in memory bandwidth usage/efficiency.

The DSC algorithm increases the use of fused multiply-add (FMA) operations on the GPU. This results in a highly efficient solution to the line equation using SP operations on GPUs. Using DSC, the difference between DDA and DSC is due to SP additions (performed by DDA), which are faster than FMA operations. The additional operations needed to map the DSC algorithm to the axis (depending on the value of the azimuth angle) have a near negligible impact on the performance of DSC.

The DDA-D implementation resulted in the most registers required per thread, using 37 as opposed to the standard DDA version which uses 26. This results in a decrease in the occupancy achieved on the GPU from 94% to 49%. This reduces the overall performance of the application, as half of the GPU is not being used due to the added contention for registers by each block. An improvement in the occupancy can be observed by reducing the number of threads per block from 1024 to 512, but this would only increase the occupancy to a maximum of 75%. Additional changes to the logic would be required in order to fully reap the benefits of using a GPU with this algorithm.

As the step size for DSC and DDA is increased (indicated by the number following the algorithm name in Figure 5), the execution time decreases as a result of a decrease in the number of total steps. The relationship between the total number of steps taken and execution time is almost 1 : 1. This occurs because we are able to effectively use the texture memory cache. When we use smaller steps for the line algorithm, we continuously read from grid cells which are already in the cache, which reduces the penalty for subsequent reads once that grid cell value is in the cache. Interestingly, DDA-D shows a performance degradation that tracks with DDA-0.5, even though the number of reads done by this method is the minimum required to cover each line. This is a result of two factors. First, threads processing angles close to multiples of 45 (135, 225, and 315) degrees will take the longest due to their higher number of reads, increasing the total execution time of the algorithm. Second, the calculations to compute the size of the step after each read results in additional execution overhead.

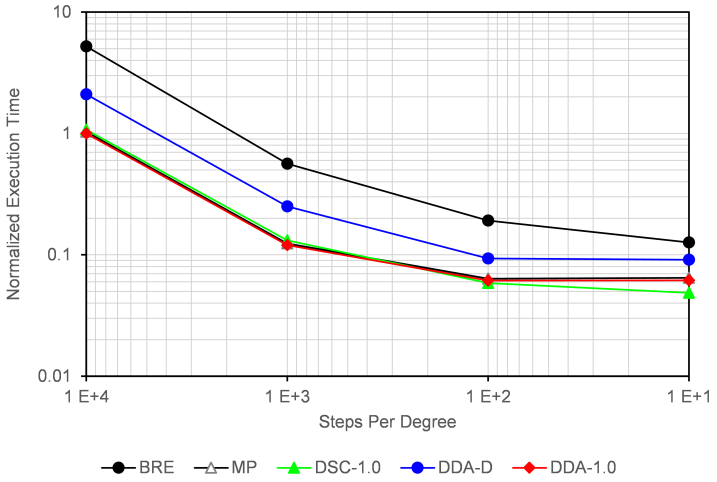


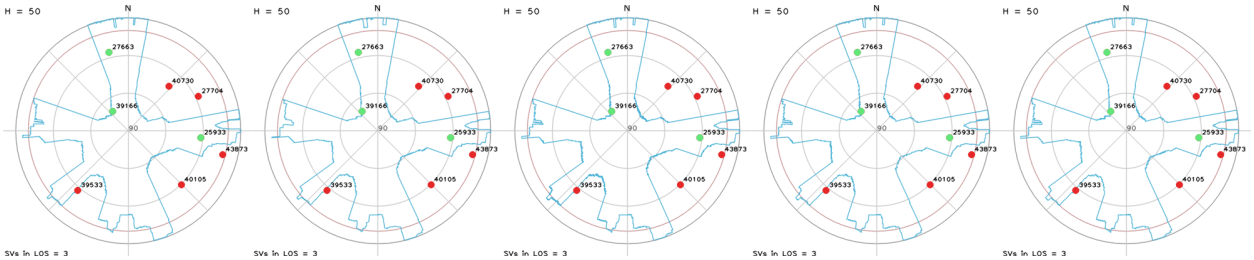
Fig. 6 Performance of the algorithms as a function of the steps per degree.

Decreasing the steps per degree (which in turn decreases the number of executing threads on the GPU) results in minimal differences in the output sky-plot for each algorithm. Figure 6 shows the normalized execution time of the line algorithms as the steps per degree are decreased. The execution time is normalized to the execution time of DDA-1.0 with 10,000 steps per degree, as shown in Figure 6. The performance of the algorithms scale proportionally with the number of threads, as we decrease the steps per degree from 10,000 to 1,000. Decreasing the steps per degree to 100 resulted in less performance improvement, and a small shift in the resulting sky-plot. Steps per degree of 100 or less are not recommended, since we are losing angle resolution for the sky-plot.

Figure 7 shows the output sky-plot from the different algorithms at 10,000 steps per degree. Close inspection shows minor differences between the plots, only occurring near the edge of tall buildings close to the POI. In this scenario, all algorithms resulted in the same satellites in line-of-sight. During real-time testing, we observed cases where BRE resulted in incorrectly categorizing a satellite as not visible. This only happened when satellites were close to the edge of a building and the line algorithm had shifted the angle of the building edge enough for this behavior to occur.

C. Stopping Condition

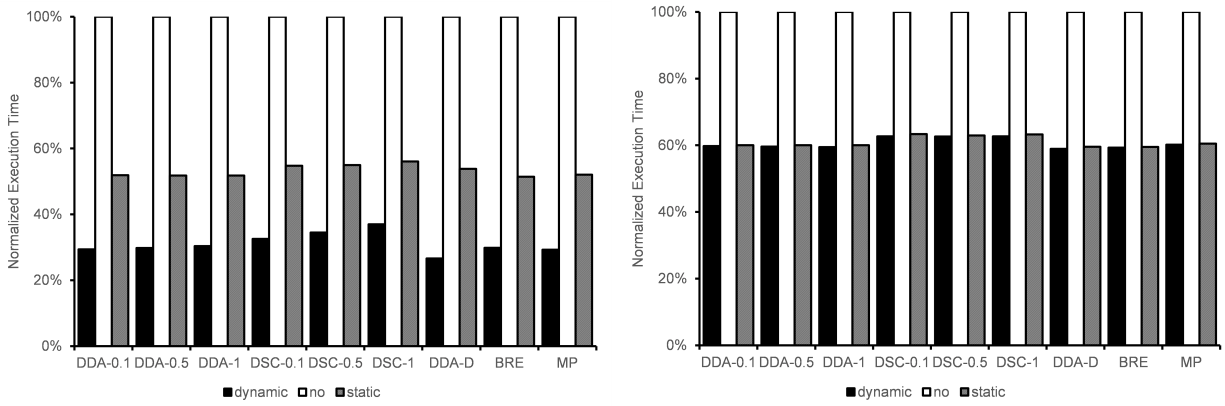
Finally, we compare the proposed dynamic stopping condition with a static stopping condition [2], as well as no stopping condition to exit the algorithm earlier. The various stopping conditions can have a different impact on performance. The degree of degradation depends on the variability and closeness of objects relative to the POI. For example, having multiple buildings close to the POI would result in a much faster termination of the loop, but if the POI is near open spaces, the benefits are significantly reduced.



(a) BRE (b) MP (c) DSC-1.0 (d) DDA-D (e) DDA-1.0

Fig. 7 Sky-plot results from the different algorithms implemented on CUDA.

Figure 8 shows the normalized execution time of DDA-1.0 at two different locations on the Boston DSM. Figure 8 (a) shows the performance observed at the coordinates **42°21'32.2"N 71°03'24.4"W**, a busy intersection surrounded by tall buildings, and (b) shows the performance observed at the coordinates **42°21'41.8"N 71°04'34.6"W**, the middle of a bridge near the center of the map, far from tall structures. The static stopping condition limits the search space for our DSM of Boston to a radius of 1207 m, which still encompasses a large portion of the map. For this reason, figure 8 (a) shows the dynamic stopping condition performs 3.2X faster than having no stopping condition, compared to 1.9X faster using the static stopping condition. Figure 8 (b) shows virtually no difference between both stopping condition for a scenario with limited blockage of the sky, both reaching 1.6X better performance as compared to the no stopping condition baseline.



(a) Urban canyon (b) Open area

Fig. 8 Stopping condition performance comparison when computing the sky-plot horizon using DDA-1.0.

VII. Closing Remarks

In this paper we explored the performance and accuracy characteristics of 5 different line marching algorithms used in the context of a sky-plot application. We found that DDA and DSC deliver similar performance, but the DDA implementation is straightforward and simpler to implement. Decreasing the step size for these two algorithms is needed to obtain good line coverage, especially with angles close to the diagonals in the sky-plot. DDA-D is a very effective algorithm if correct line coverage is required, without impacting the performance significantly. If performance is important and a worst-case scenario is useful, MP is the best option, as it guarantees full line coverage and negligible performance impact compared to DDA. Finally the correctness of the implementation can be critically important, dependent on the application. There is a clear trade-off between performance and accuracy.

References

[1] Dardari, D., Falletti, E., and Luise, M., *Satellite and terrestrial radio positioning techniques: a signal processing perspective*, Academic Press, 2011.

[2] Gandolfi, S., and La Via, L., "SKYPLOT_DEM: a tool for GNSS planning and simulations," *Applied Geomatics*, Vol. 3, No. 1, 2011, pp. 35–48.

[3] Dardari, D., Closas, P., and Djurić, P. M., "Indoor tracking: Theory, methods, and technologies," *IEEE Transactions on Vehicular Technology*, Vol. 64, No. 4, 2015, pp. 1263–1278.

[4] Kassas, Z. M., Closas, P., and Gross, J., "Navigation systems panel report navigation systems for autonomous and semi-autonomous vehicles: current trends and future challenges," *IEEE Aerospace and Electronic Systems Magazine*, Vol. 34, No. 5, 2019.

[5] Amin, M. G., Closas, P., Broumandan, A., and Volakis, J. L., "Vulnerabilities, threats, and authentication in satellite-based navigation systems [scanning the issue]," *Proceedings of the IEEE*, Vol. 104, No. 6, 2016, pp. 1169–1173.

[6] Pelc-Mieczkowska, R., Tomaszewski, D., and Bednarczyk, M., "GNSS obstacle mapping as a data preprocessing tool for positioning in a multipath environment," *Measurement Science and Technology*, Vol. 31, No. 1, 2019, p. 015017.

[7] Morton, Y. J., van Diggelen, F., Spilker Jr, J. J., Parkinson, B. W., Lo, S., and Gao, G., *Position, Navigation, and Timing Technologies in the 21st Century, Volumes 1 and 2: Integrated Satellite Navigation, Sensor Systems, and Civil Applications, Set*, John Wiley & Sons, 2020.

[8] Groves, P. D., "Shadow matching: A new GNSS positioning technique for urban canyons," *Journal of Navigation*, Vol. 64, No. 3, 2011, pp. 417–430.

[9] Marshall, J., "Creating and viewing skyplots," *GPS solutions*, Vol. 6, No. 1, 2002, pp. 118–120.

[10] Pratx, G., Chinn, G., Olcott, P. D., and Levin, C. S., "Fast, accurate and shift-varying line projections for iterative reconstruction using the GPU," *IEEE transactions on medical imaging*, Vol. 28, No. 3, 2008, pp. 435–445.

[11] Hofmann-Wellenhof, B., Lichtenegger, H., and Collins, J., *Global positioning system: theory and practice*, Springer Science & Business Media, 2012.

[12] Taylor, G., Li, J., Kidner, D., Brunsdon, C., and Ware, M., "Modelling and prediction of GPS availability with digital photogrammetry and LiDAR," *International Journal of Geographical Information Science*, Vol. 21, No. 1, 2007, pp. 1–20.

[13] Wang, H., and Chang, X., "3-D ray tracing method based on graphic structure," *Chinese Journal of Geophysics*, Vol. 43, No. 4, 2000, pp. 568–575.

[14] Tordesillas, J., Lopez, B. T., and How, J. P., "Faster: Fast and safe trajectory planner for flights in unknown environments," *2019 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, IEEE, 2019, pp. 1934–1940.

[15] Fickenscher, J., Schlumberger, J., Hannig, F., Teich, J., and Bouzouraa, M. E., "Cell-based update algorithm for occupancy grid maps and hybrid map for ADAS on embedded GPUs," *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2018, pp. 443–448.

[16] Xiao, K., Chen, D. Z., Hu, X. S., and Zhou, B., "Efficient implementation of the 3D-DDA ray traversal algorithm on GPU and its application in radiation dose calculation," *Medical physics*, Vol. 39, No. 12, 2012, pp. 7619–7625.

[17] Perlin, K., and Hoffert, E. M., "Hypertexture," *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, 1989, pp. 253–262.

[18] Bresenham, J. E., "Algorithm for computer control of a digital plotter," *IBM Systems journal*, Vol. 4, No. 1, 1965, pp. 25–30.

[19] Jia, X., Ziegenhein, P., and Jiang, S. B., "GPU-based high-performance computing for radiation therapy," *Physics in Medicine & Biology*, Vol. 59, No. 4, 2014, p. R151.

[20] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide," , 2021. Version 11.2.

[21] Bradski, G., "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.