Journal of Computer and Communications, 2017, 5, 21-51
http://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Adaptive Region Construction for Efficient Use of Radio Propagation Maps

Vinay B. Ramakrishnaiah, Suresh S. Muknahallipatna*, Robert F. Kubichek

Department of Electrical & Computer Engineering, University of Wyoming, Laramie, WY, USA
Email: *sureshm@uwyo.edu

How to cite this paper: Ramakrishnaiah, V.B., Muknahallipatna, S.S. and Kubichek, R.F. (2017) Adaptive Region Construction for Efficient Use of Radio Propagation Maps. *Journal of Computer and Communications*, **5**, 21-51.
https://doi.org/10.4236/jcc.2017.58003

Received: May 4, 2017
Accepted: June 12, 2017
Published: June 15, 2017

Abstract

The efficient construction of contours of a radio propagation map is crucial in using radio propagation maps in a number of real-time communication and network applications. In this research work, we first propose an adaptive region construction (ARC) technique capable of constructing contours of different resolutions of a radio propagation map. Next, the process of implementing the ARC technique for real-time execution on a GPU is presented. The drawbacks of the first implementation using only the global memory are discussed, and optimization techniques to improve the performance are discussed and implemented. Simulations are performed with varying sizes of radio propagation maps, and the suitability of the ARC technique for real-time operation is presented. A speedup of 25× is achieved with the shared version of the GPU compared to the sequential CPU implementation. Also, the contour constructed using the ARC technique is compared to that constructed using the convex hull approach demonstrating the higher accuracy of the contour from the ARC technique.

Keywords

Radio Propagation, Sampling, Parallel, Adaptive, Accelerator, Contours

1. Introduction

A mobile ad-hoc network (MANET) of nodes (equipped with sensors) can be deployed rapidly in an environment to provide a communication infrastructure for a number of applications like environmental monitoring, rescue and defense operations to mention a few. However, the successful deployment of a MANET is dependent on the ability of neighboring nodes establishing a wireless communication link and provides connectivity across the network. Establishing a communication link is dependent on the availability of radio spectrum, transmission power, interference from neighboring nodes, and the effect of terrain on radio propagation. Depending on the deployment environment, the effect of terrain on radio propagation can be modeled with free space (Omni-directional) or "two-ray" propagation models if the terrain does not have any hills, buildings, and foliage affecting the propagation of radio signals minimally. However, in a realistic environment where MANETs are deployed, the terrain consists of hills, buildings, foliage, etc., causing reflections, diffraction, blocking and unreliable path loss estimates which render free space or two-ray propagation models ineffective in modeling the effect of terrain on radio propagation. Hence, more complex propagation modeling techniques like the Walfisch-Ikegami model (WIM), and 3D ray tracing have to be used to determine the effect of terrain on radio propagation. The effect of terrain on the radio propagation by WIM or 3D ray tracing is quantified as radio propagation map.

Radio propagation maps specify the path loss (or received signal strength) at various distances and directions from the transmitter taking into account the effect of the terrain. The path loss [1] is computed using WIM, 3D ray tracing, and other models. Each modeling offers trade-offs in terms of accuracy and computational complexity. A radio propagation map consists of received signal strengths from a transmitter over a geographic region show neither as a heat map or radio contours. Heat maps represent varying signal strengths from a transmitter using distinct colors while in radio contours equal signal strength locations are connected by lines known as contour lines. A heat map can be stored as a pixel image, with pixels representing the signal strength of a location from the transmitter. Radio propagation maps find many applications like the optimum placement of mobile phone base stations for maximum coverage, adjusting the antenna beam patterns for efficient communication based on the terrain, localization of mobile nodes in MANETs, etc.

The application of radio propagation maps in localizing nodes, and antenna beam forming [2] [3] [4] [5] has been demonstrated by the authors of this paper. The first application addressing efficient localization of nodes of a MANET under free space and terrain effects is discussed in references [2] [3] [4]. In reference [3], the optimal trajectory of a single moving beacon (beacon mounted on a drone or UAV) to localize nodes under free space was presented. Next, the use of high-resolution radio propagation maps [2] to localize nodes in a MANET using a single moving beacon like a drone or unmanned aerial vehicle has been presented.

In the research work [2], the use of regular geometric shapes known as convex hulls to represent irregular radio propagation shapes constructed using the WIM model has been presented. The paramount reason for representing irregular radio propagation shapes with convex hulls is associated with the localizing algorithm [3] developed for nodes of a MANET. The localization algorithm requires determination of the area of intersection (approximate node position) of two radio propagation maps received by the node. Determining the area of intersection of two radio propagation maps using a simple bit-wise AND operation [4] was demonstrated and found to be impractical due to computational complexity, bandwidth, and storage requirements of each node to use the radio propagation maps. Therefore, to address the issues of bandwidth and storage requirements, the contour of radio propagation maps was constructed as convex hulls using Andrew's Monotone Chain algorithm [6]. Next, the algorithm developed by O'Rourke [7] was used to determine the area of intersection of two radio propagation maps using the constructed convex hulls. The second application consists of using low-resolution radio propagation map for computing antenna beam patterns [5] to reduce the interference by neighboring nodes.

The success rate of the localization algorithm using convex hulls representing the contour of a radio propagation map of the small geographical area was 80% with an accuracy of 1 m. However, with an increase in the size of the geographical area, the performance of the algorithm deteriorated first due to the convex hulls not capturing the contour of the radio propagation map accurately. The second issue was with the nonlinear increase of the computational time for constructing a convex hull of a radio propagation map of the large geographical area (a suburb of a city) with high resolution rendering the localization algorithm not suitable for real-time application.

Hence in this work, we propose an adaptive region construction technique to capture the contour of a radio propagation map of large geographical area accurately and suitable for real-time implementation. We propose the use of a general purpose graphical processing unit (GPGPU) based adaptive region construction (ARC) for constructing multiple convex hulls of a radio propagation map of a large geographical area and combine multiple convex hulls to form regions representing the contour of radio propagation maps accurately.

This paper is organized as follows: In Section 2, a brief discussion of the localization algorithm [2] developed previously is discussed. The error in the intersection area determined using two convex hulls in comparison to the direct use of radio propagation maps is shown. Also, the increase in computational time for constructing convex hulls of large radio propagation maps with high resolution is demonstrated. Section 3 presents a discussion on the related work of capturing contour of radio propagation maps. In Section 4, first, a brief discussion of the sequential Andrew's monotone chain algorithm and divide and conquer approach for constructing convex hulls is presented. Section 5 presents the adaptive region construction technique that merges intermediate convex hulls to form regions. The GPGPU implementation of the ARC using only the global memory of the GPU is presented in Section 6. Section 7, presents the optimizations using the shared memory of the GPU to improve the performance of the algorithms is presented. Results and analysis of the results are presented in Section 8. In Section 9, conclusion and future work are presented.

2. Localization in Mobile Ad Hoc Networks

Miles *et al.* envisioned a single moving beacon mounted on a vehicle capable of moving and broadcasting its position periodically to nodes in its transmission range based on radio propagation models [2] representing a particular urban environment. Using *a-priori* knowledge of local terrain including average building height and separation, average street width and orientation, etc., the WIM model is used to estimate path loss and transmission range in any direction. The single moving beacon will acquire its geographical position through GPS and broadcast its changing position. Having a single moving beacon broadcasting its changing position is equivalent to multiple stationary beacons broadcasting their different positions. An exchange of a few position messages and acknowledgments between an un-localized node and the moving beacon will allow the moving beacon to compute an approximate area to confine the un-localized node location.

Figure 1 shows an approximate propagation map generated using WIM techniques. The white area in Figure 1 shows the estimated area in which an antenna is capable of receiving a transmission from a transmitter (beacon node) positioned at the centroid of the white area. The possible position of a sensor in a wireless network using WIM transmission maps can be approximated as the centroid of the set given by the intersection of several beacon transmissions as shown in Figure 2. A WIM localization technique [4] suggested by Muralidhara and Kubichek relies on bitwise AND operations between two radio propagation maps for beacon transmissions near an unknown sensor. Depending on the size of the coverage area spanned by the MANET, these maps can be quite large.

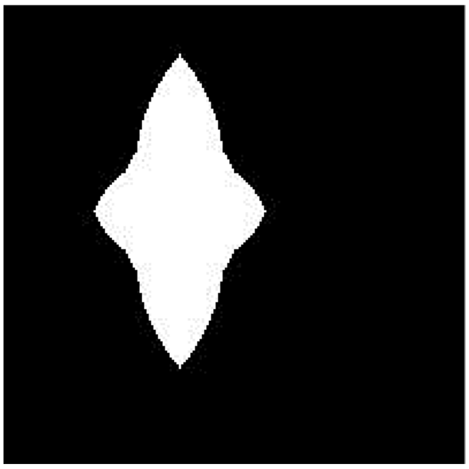


Figure 1. WIM propagation approximation.

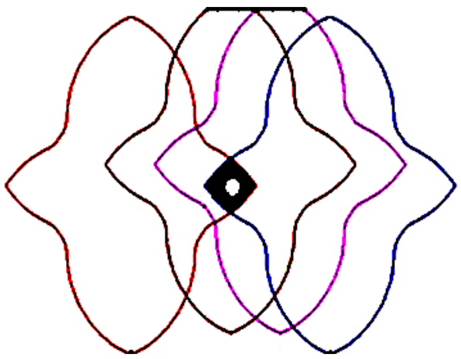
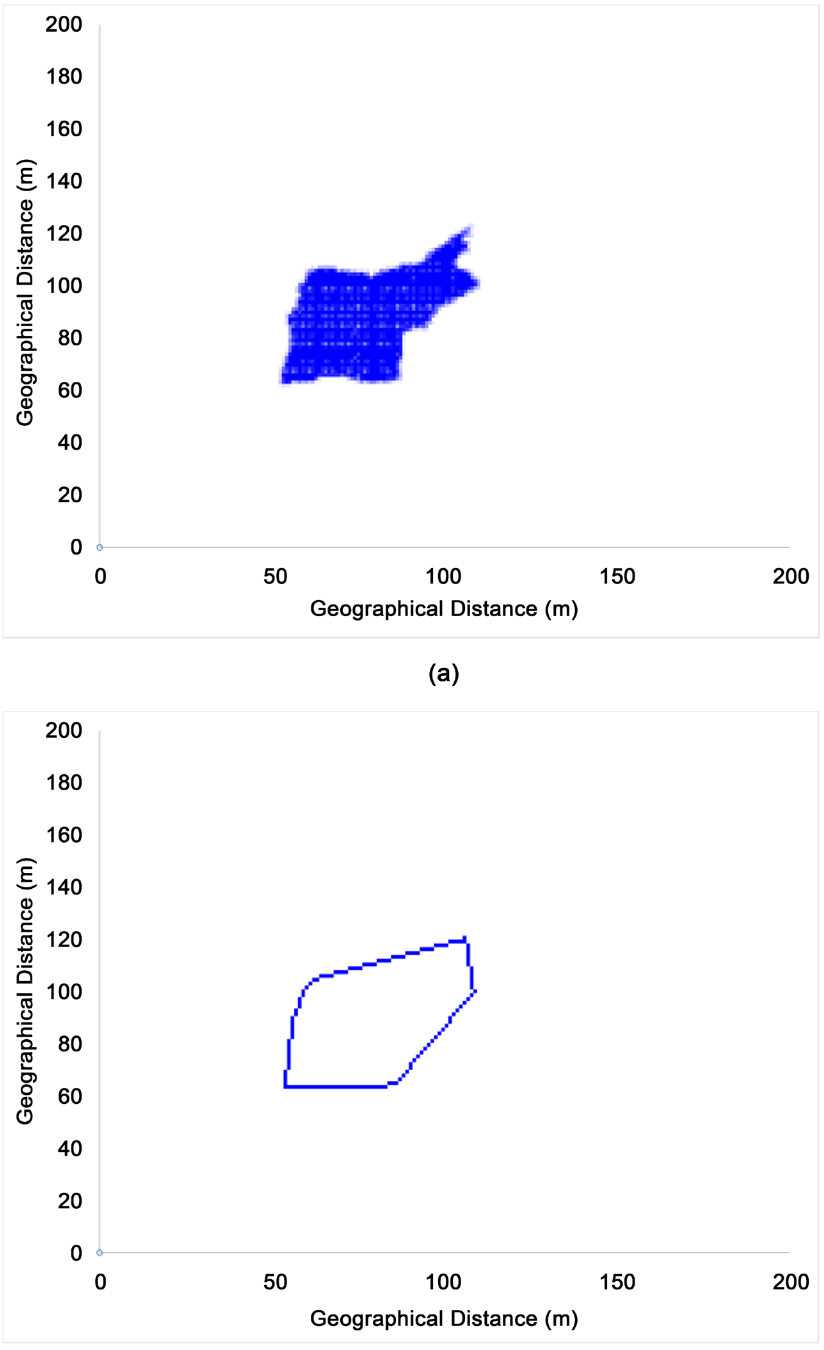


Figure 2. Possible position of a node in MANET using WIM.



(b)

Figure 3. (a) Range map and its corresponding (b) convex hull.

Transmission and comparison of these maps require large bandwidth and computational capability. Also, it is difficult to implement an intelligent localization algorithm based on the shapes of the range maps. The above drawbacks are addressed by using convex hulls to represent the irregular radio propagation shapes with regular geometric shapes [2]. By using the convex hull approach, each radio propagation map can be stored using only a small number of boundary points as shown in **Figure 3**. Next, the intersection of two convex hulls representing the approximate location of an un-localized node as shown in **Fig-**

ure 4 is determined using the algorithm proposed by O'Rourke [7].

Even though the use of convex hulls to represent radio propagation maps had reduced the storage requirement and transmission bandwidth, it introduced significant errors in localization and increased the computational burden. In Figure 5, a number of examples of the artifacts introduced in representing radio propagation maps by convex hulls are shown. The first column (a - c) in Figure 5 depicts the radio propagation maps constructed using the WIM model, the second column (d - f) shows the corresponding convex hull, and the third column (g - i) shows the artifacts introduced in representing radio propagation maps by convex hulls. The difference in the area of the convex hull and the area enclosed by the radio propagation contour is the artifact introduced by the convex hull approximation. While using the technique of localization using the intersection of the convex hulls, the artifacts introduce superfluous area, which in turn introduces error in the computed localized position compared to the localization using the intersection of radio propagation contours as shown in Figure 6. In Figure 6, with the actual radio propagation maps, the position is close to the real position, whereas for the convex hulls it is the centroid of the intersection area of the two hulls, which is far away from the real position. As an example, a convex hull was constructed to represent the radio propagation map of size

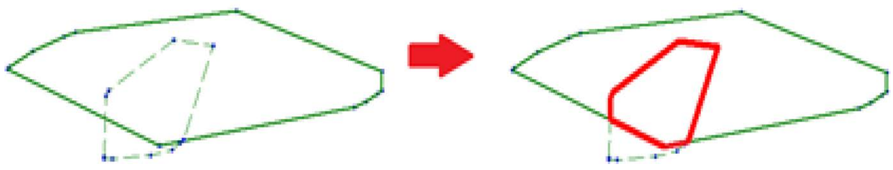


Figure 4. Intersection of two convex hulls.

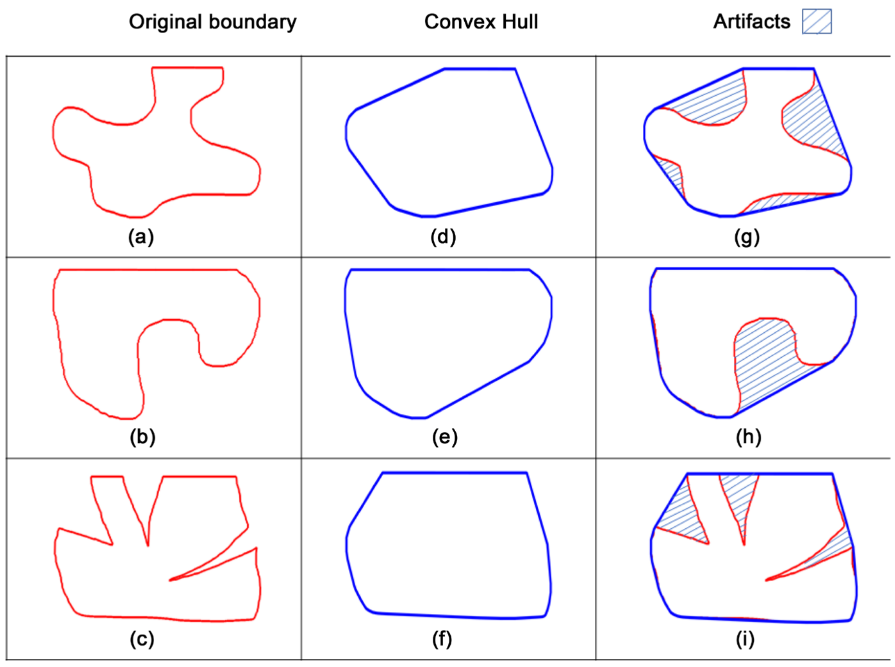


Figure 5. The original boundary, convex hull, and the artifacts introduced by convex hulls.

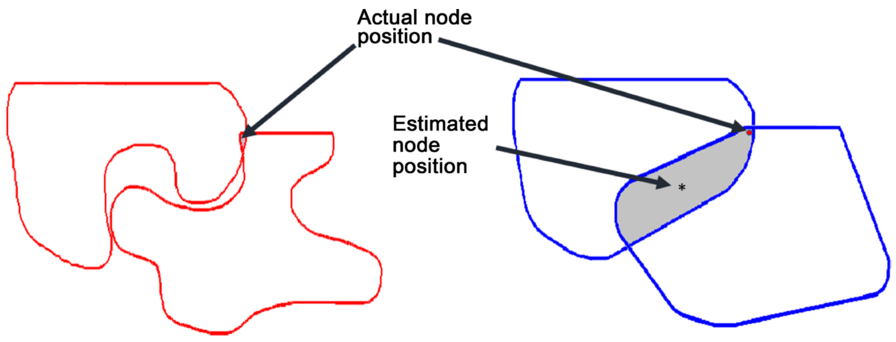


Figure 6. Estimated node positions using the boundary of radio propagation map compared to the estimated position using convex hulls.

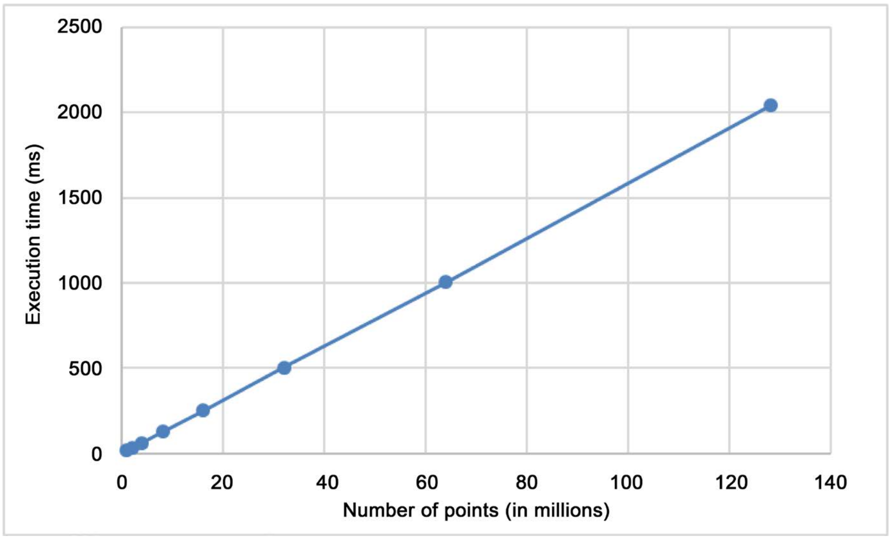


Figure 7. Time taken to compute convex hulls using Andrew's monotone chain algorithm.

2.5 km by 2.5 km. The difference in area due to the artifact introduced between the contour of the radio propagation map and the corresponding convex hull was approximately 0.4 km^2, which is about 1/15th of the original area. This difference in the area can be even more significant when considering radio propagation maps of large urban areas. Along with the introduction of artifacts, the computation of convex hulls of very large geographic areas is a computationally intensive problem. Even though efficient algorithms [7] are available to compute convex hulls with the computational complexity of ($O(n\log(n))$), the computational complexity is an issue when computing convex hulls of large cities sequentially. The time taken to compute a convex hull sequentially on an Intel Xeon Sandy Bridge processor using the best sequential algorithm for different resolutions is shown in Figure 7. From Figure 7 we can see that the execution time required to compute convex hulls using sequential algorithms is not suitable for real time applications, especially when high resolution radio propagation maps are used.

In this research work, we propose a new method to assist computation of contours of a radio propagation map known as the Adaptive Region Construction (ARC) technique. The ARC technique first reduces the superfluous area introduced by the use of convex hulls representing radio propagation maps and thereby reduces the localization error. Second, the ARC technique is implemented using the general-purpose computing on graphic processing units (GPGPU) to reduce the computational time and make use of radio propagation maps in real-time applications feasible.

3. Previous Work

Curve simplification algorithms like the Ramer-Douglas-Peucker algorithm [8] [9] are used to approximate radio contours by representing the sequence of points as line segments. The simplified curve consists of a subset of the original set of points, and the algorithm tries to minimize the distance between the original curve and the simplified curve. This algorithm does a crude simplification of joining the first and the last vertices of a polyline with a single edge, which can result in over-simplification of the details. Furthermore, using these algorithms directly on large data sets is not suitable for real time applications due to increase in computational complexity.

Many computer vision applications make use of convex hulls to approximate blobs and shapes in images. The authors of [10] use convex hulls to categorize shapes of leaves. Convex hulls are constructed to represent the contours of leaves and are categorized by finding the variations between the interior angles at each control point. However, if two distinct leaves have the same contour constructed as convex hulls, the method fails to categorize distinct leaves. This idea has been extended to represent radio contours using convex hulls in [2].

The authors of [11] propose using convex hulls for a simplified representation of "building footprints" on radio propagation which is a crucial step in wireless communication. Footprint reduction is crucial for reducing prediction time and controlling prediction accuracy in radio communication. They identify that such reductions often affect the accuracy of results as simplification error constrains the efficiency that can be achieved. In other words, the prediction accuracy can be improved by better footprint reduction techniques. This analysis helps in better understanding of the trade-off between the precision of the building database and the accuracy of predictions generated by ray-tracing based radio propagation prediction systems.

The authors of [12] construct convex ray paths to simplify radio propagation ray path calculations. Multiple reflections are modeled by their equivalent convex hull. This is a good approximation for calculating signal strengths at a given location from the transmitter but can result in errors if the propagation path of radio waves through the topography is complex, which cannot be modelled using simple convex hulls. They also propose using the ordinary graphics card and specialized algorithms to achieve extremely fast radio wave propagation predictions. They show that their implementation of the COST-Walfisch-Ikegami model can efficiently calculate 200 predictions per second, whereas a CPU implementation of the same COST-Walfisch-Ikegami models needs slightly less than a minute for a single prediction.

Cheng *et al.* [13] proposes a method to improve wireless connectivity in ad hoc networks using a partitioning technique based on the conductance of a network. They use convex optimization to maximize the number of connections between the communicating nodes on different sides of the partition. The optimization is used to find the precise location of the relay node, which is within the convex hull defined by the radio transmission ranges of all the nodes that can connect the relay node. This process can be improved by accurately defining the radio propagation maps for each transmitter positions.

Liu *et al.* [14] focus on identification of non-line-of-sight (NLOS) signal propagation, which is the dominant source of localization error in wireless network nodes. They present a theoretical analysis of mobile user localization involving one or more NLOS beacons and show that if the mobile user is within the convex hull region formed by the underlying beacons, localization involving NLOS is likely to be largely inconsistent. However, if the mobile user is outside the convex hull region, localization involving NLOS could be performed consistently. They argue that relying on existing methods to identify NLOS would lead to a great chance of underestimating the potentially serious errors in localization involving NLOS.

Most of the research work discussed above requires contour and points internal to the contour to represent the radio propagation maps. However, for localization and other applications, accurate representations of the radio propagation map to reduce the storages and bandwidth requirements, which are suitable for real time applications that are required. Hence in our work, we present the adaptive region construction (ARC) technique capable of aiding the construction of an accurate contour representing the shape of the radio propagation map. The ARC technique described in this paper can define the given radio propagation map contour more accurately compared to a convex hull. Even though our algorithm is developed for approximating radio propagation maps, it can improve the accuracy of many applications that make use of sample approximations and demand real-time/near real-time performance. Our algorithm is computationally less complexity and is parallelizable, which makes it suited for real-time applications.

4. Review of Convex Hulls

To review, by definition, a set, C, is convex [15] if and only if for any x_1, x_2 $\in$ C and any θ where $0 \le \theta \le 1$ the following condition holds: $\theta x_1 + (1-\theta) x_2 \in C$.

In simple terms, this means that a set is convex if the direct path between any two points in the set is entirely included in the set. Figure 8 shows a convex set. Note that the line between two elements within the set is, itself, completely encompassed in the set. If a set bounded by the edges of the white area in Figure 1 is used, it is easy to see that there are direct paths between elements of the set that do not lie completely within the set as in Figure 9(a). A convex hull of a set, C, is the minimum convex set that contains the set C. Figure 9(b) represents the

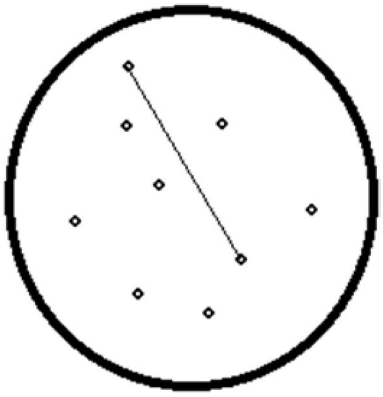


Figure 8. A convex set of points.

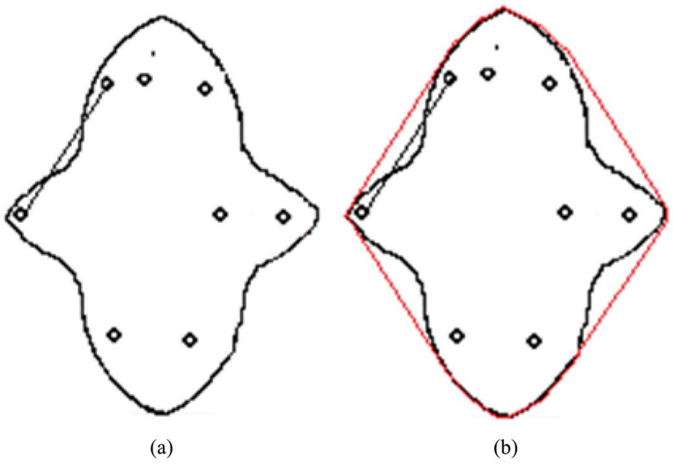


Figure 9. Convex hull of a non-convex set of points.

convex hull of the set shown in Figure 9(a). As the figure indicates, all paths between points within the convex hull are now completely encompassed in the set.

By constructing a convex hull of a range map, the storage and transmission bandwidth requirements can be greatly reduced. This results from the fact that only a small number of boundary points are required to represent the convex hull, which approximates the actual radio propagation map. This will serve as a lossy compression technique for the localization method.

Another benefit of the convex hull is that it can be used to make intelligent movement decisions more easily as the computation of the intersecting area only requires the use of the boundary points of the intersecting convex hulls instead of the entire radio propagation maps.

4.1. Overview of Andrew's Monotone Chain Convex Hull Algorithm

The Andrew's monotone chain convex hull algorithm [6] can find the convex hull of a set of points in $O(n \log n)$ time. One particular advantage of this algorithm is that it can find the convex hull in $O(n)$ time if the points are already sorted in ascending order from left to right and top to bottom, which is the case for the data in the radio propagation maps used in this work. This algorithm computes the upper and lower convex hulls of a monotone chain of points. The flowchart in Figure 10 illustrates the mechanism by which this algorithm computes the convex hull.

The upper hull is computed in a similar fashion, and the two hull sets are joined to find the final convex hull. Essentially, the algorithm works by comparing points to lines formed between previous points starting from left to right to make the upper hull, and then from right to left to make the lower hull. The algorithm makes its decision on which point belongs in the hull by computing the curl between the vector composed of the previously selected point and the second to last point in the hull and vector between the current point and the second to last point in the hull. Figure 11 shows a set of points in the early stage of hull construction. The curl, *Crl*, between the vectors $P_{minmax,1}$ and $P_{minmax,2}$, computed from Equation (1) will result in a positive number, indicating that the point P_2 lies to the relative interior if the line between P_{minmax} and P_2.

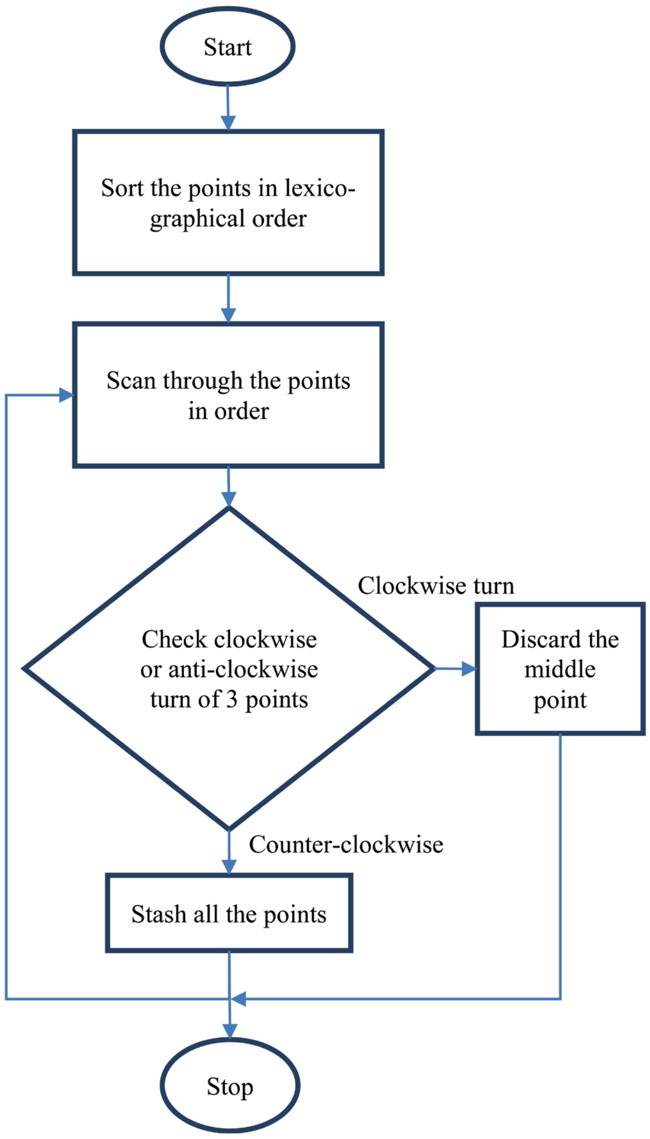

Start
Sort the points in lexico-graphical order
Scan through the points in order
Check clockwise or anti-clockwise turn of 3 points
Clockwise turn
Discard the middle point
Counter-clockwise
Stash all the points
Stop

Figure 10. Flow chart of the sequential Andrew's monotone chain convex hull algorithm.

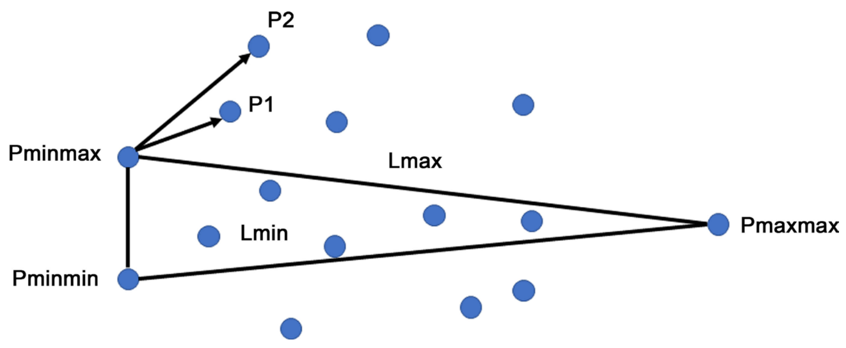


Figure 11. Set of points in the early stage of hull computation.

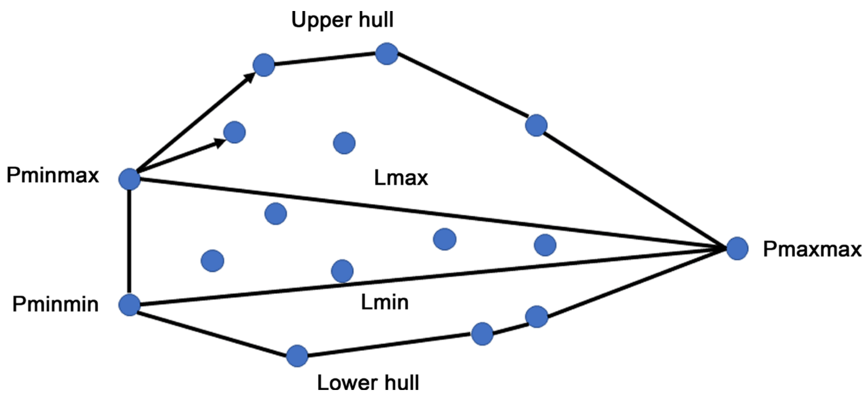


Figure 12. Resulting convex hull.

$$Crl = P_{\min\max,1} \times P_{\min\max,2} \tag{1}$$

In order to satisfy Equation (1), the points included in the hull must be located to the relative exterior of all points included in the hull, and in line with all points included in the hull. In the case illustrated in Figure 11, the point, P_1 will be discarded from the hull and replaced, by P_2. Then the algorithm proceeds by checking points to the right until it reaches the right-most point. Then it begins moving back to the left computing the lower hull in a similar fashion. Figure 12 shows the result of the algorithm for a set of points.

4.2. The Divide and Conquer Approach for Constructing Convex Hulls

The divide and conquer approach was developed by [16] as an efficient algorithm for computing convex hulls in three dimensions if the points are sorted in lexicographical order. The points are divided into two sets A and B, containing the left half and the right half of the points respectively as shown in Figure 13. Convex hulls are computed recursively on these two sets, and the sets are merged by computing the union of the convex hulls. The division of points into left and right sets is continued recursively until the number of points, n, in each set is less than or equal to three. This algorithm assumes non-collinear points, which makes the smallest convex hull either a triangle (if n = 3) or a straight line (if n = 2). Therefore, majority of the computational effort involved in this algorithm is with the merge step.

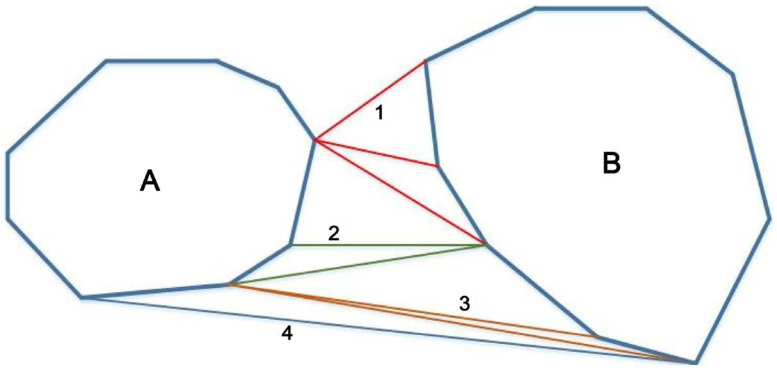


Figure 13. Constructing the lower tangent to merge two consecutive hulls.

To merge the convex hulls, common tangents are constructed between two consecutive convex hulls and the convex hulls are merged hierarchically. In Figure 13, the numbers represent the steps of the tangent determination process. In order to construct the tangents, a line is drawn between the rightmost point of the left hull and the leftmost point of the right hull. The left end of the line is fixed, and the right end is moved on the convex hull until it becomes a tangent to the right hull (Step 1 in Figure 13). Now the right end of the line is fixed, and the left end is moved until it becomes a tangent to the left hull (Step 2 in Figure 13). This process is repeated until a common tangent is attained as shown in Figure 13.

5. Adaptive Region Construction

This section describes the process of adaptive region construction (ARC). This approach is developed to represent the radio propagation characteristics like signal strength at a spatial location in an efficient way. The procedure described here combines the ideas from the Andrews's monotone chain convex hull algorithm [6] and the divide and conquer approach [17] to implement ARC. ARC provides more details about a contour of a radio propagation map when compared to a convex hull approximation. ARC constructs the contour of a radio propagation map by constructing intermediate convex hulls to fit the given radio propagation map contour and combining them consecutively. This process also requires points to be sorted in lexicographical order. A number of parallel sorting algorithms like the CUDA Dynamic Parallelism (CDP) quicksort, CUDA quicksort [18], etc., can be used. Furthermore, if the radio propagation map is constructed using WIM, the data points are sorted in the Cartesian coordinate system, thereby eliminating the need to sort the data set. The given data is divided into smaller segments and operated in parallel on individual segments of data to exploit data level parallelism. Andrew's monotone chain convex hull algorithm is used on each segment of data to construct intermediate convex hulls. Later, the individual convex hulls are combined by constructing common tangents to the consecutive convex hulls in parallel. The resulting set of points tries to preserve the shape of the radio propagation map in general. The boundary of ARC compared with the convex hull of the corresponding radio propagation

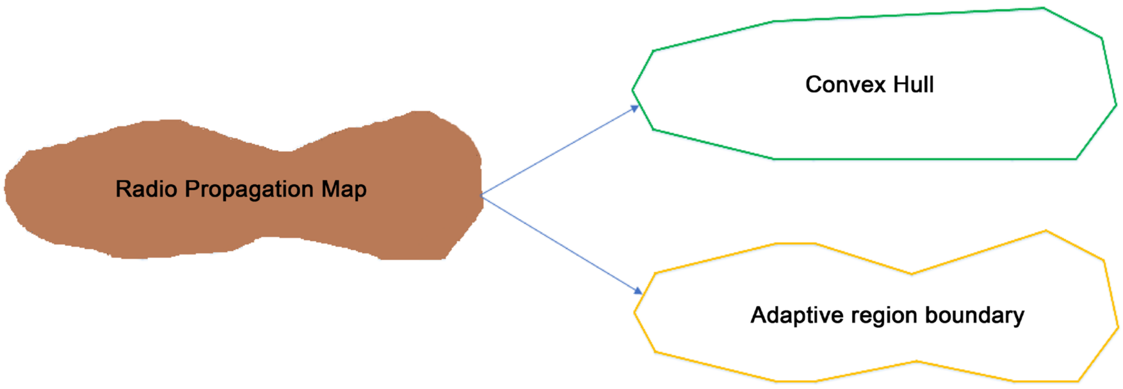


Figure 14. Radio propagation map represented using convex hull and adaptive region construction.

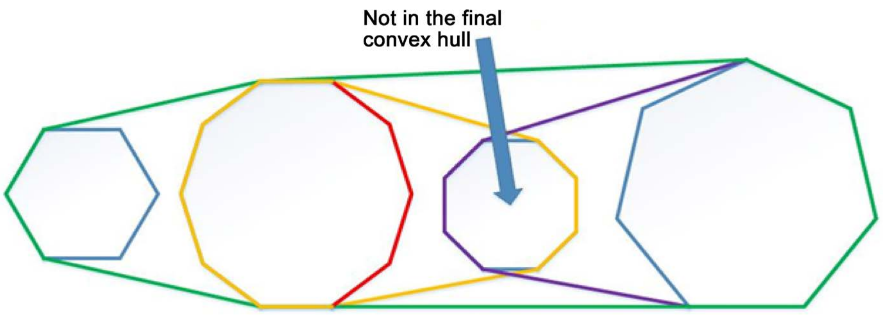


Figure 15. Illustration of adaptive region construction. Adaptive region construction includes the octagon which is not a part of the final convex hull.

map is exemplified in Figure 14. This set obtained using the process described above includes points that are not in the final convex hull for the given set of points. The result at the end of this step is the adaptively constructed region as illustrated in Figure 15. The ARC process is a combination of Andrew's monotone chain algorithm and the divide and conquer approach of computing convex hulls, and both have a worst case computational complexity of $O(n\log(n))$. Therefore, the inherently parallel process of constructing adaptive regions has a computational complexity of the order

$$O\left(\underbrace{\frac{n}{m}\log\left(\frac{n}{m}\right)}_{\text{Andrew's monotone chain}}+\underbrace{\frac{n}{m}\log\left(\frac{n}{m}\right)}_{\text{Divide \& Conquer}}\right)=O\left(2\frac{n}{m}\log\left(\frac{n}{m}\right)\right),$$

where n is the total number of points in the dataset, m is the number of processes or threads that can execute simultaneously, and $n/m \geq 3$ as at least three points are required to compute a convex hull. This shows that having many processes running in parallel reduces the computational complexity of the algorithm.

6. General Purpose GPU Implementation

Heterogeneous computing is the approach of using accelerators/co-processors in conjunction with Central Processing Units (CPUs) to solve computationally intensive problems. Accelerators can be vector processors; many core processors like Graphics processing units (GPUs) and Intel Xeon Ph is that improve the performance of applications by utilizing parallelism. GPUs are specialized hardware designed to handle the intensive operation of the rendering of image frames for output to a display device. With the emergence of programmable shaders, researchers started using GPUs to solve problems involving matrices and vectors to achieve performance improvement by making use of parallelism. When GPUs are used for computations in non-graphics related problems, it is known as general purpose GPU (GPGPU) computing. Initial efforts of programming GPUs involved refactoring the problems to use graphics primitives provided by the graphics application programming interfaces. NVIDIA's Compute Unified Device Architecture (CUDA) [19] is an attempt to ameliorate the cumbersome process of programming GPUs. CUDA provides simple language extensions to programming languages like C/C++, FORTRAN, and Python to expose fine and coarse grained parallelism in applications. NVIDIA has introduced several hardware architectures with CUDA support to improve the performance of parallelized programs.

The generalized hardware hierarchy in NVIDIA GPUs consists of multiple arithmetic and logic units (ALUs), and they are called CUDA cores as shown in the right-half of Figure 16. A fixed number of these cores are grouped along with control hardware and memory to form units known as Streaming Multi-processors (SMs). The entire device constitutes of several SMs, providing a large number of processing cores. This hierarchy in hardware is matched in the software hierarchy shown in the left-half of Figure 16 by the CUDA programming model. It provides a software hierarchy of threads, thread blocks, and grids, which have an affinity to CUDA cores, SMs, and the device correspondingly

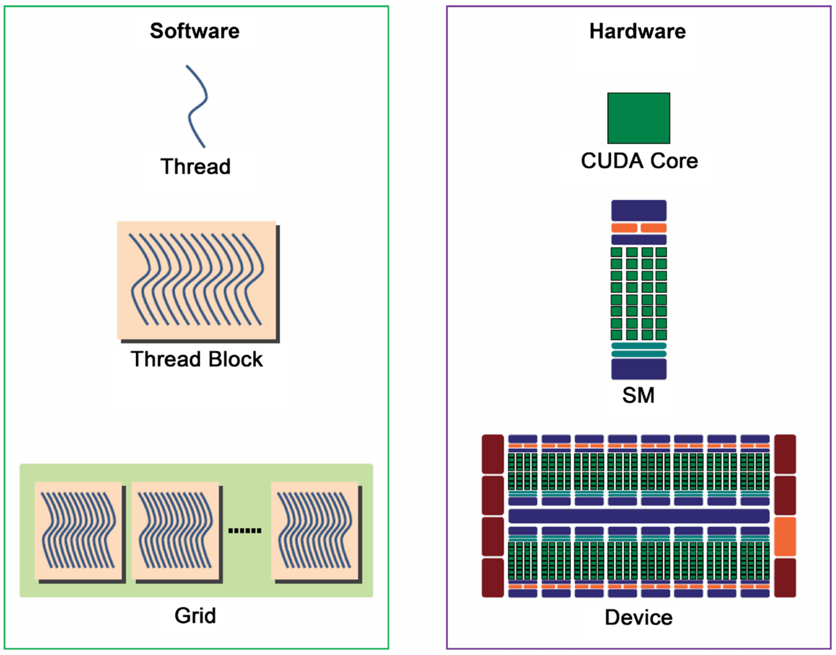


Figure 16. The software (left-half) and hardware (right-half) hierarchy correspondence in NVIDIA GPUs (Image source [21]).

as shown in the left half of Figure 16. The threads in a thread block can be arranged in 1D, 2D, or 3D fashion and in a similar fashion, the thread blocks can be arranged in a grid. The total number of threads spawned in a thread block is fixed, hence is the load handled by each SM.

GPU memory can be classified into 3 categories namely the registers, shared memory, and global memory as shown in Figure 17. Some GPU architectures, in addition to the shared, global, and register memory also have texture and constant memory that are read-only memories for GPUs with optimized cache access. Each CUDA thread has limited private registers which are the fastest form of memory available on a GPU. Threads within a block have access to the shared memory through which they can exchange data, while all the threads in the device have access to the global memory. The memory access latency increases exponentially from registers to global memory as we move away from the processing core and so does the size of memory. In other words, GPUs have a memory hierarchy similar to any modern-day vector processor. Like any other modern computer, GPUs are also benefited by the efficient use of memory bandwidth.

NVIDIA GPUs follow the single program, multiple threads (SPMT) execution model of parallel computing. This means a group of threads execute the same set of instructions in lock-step, though conditional branches in the algorithm can violate the lock-step execution of instructions contributing to an increase in computational time. The SMs in NVIDIA GPUs always execute instructions

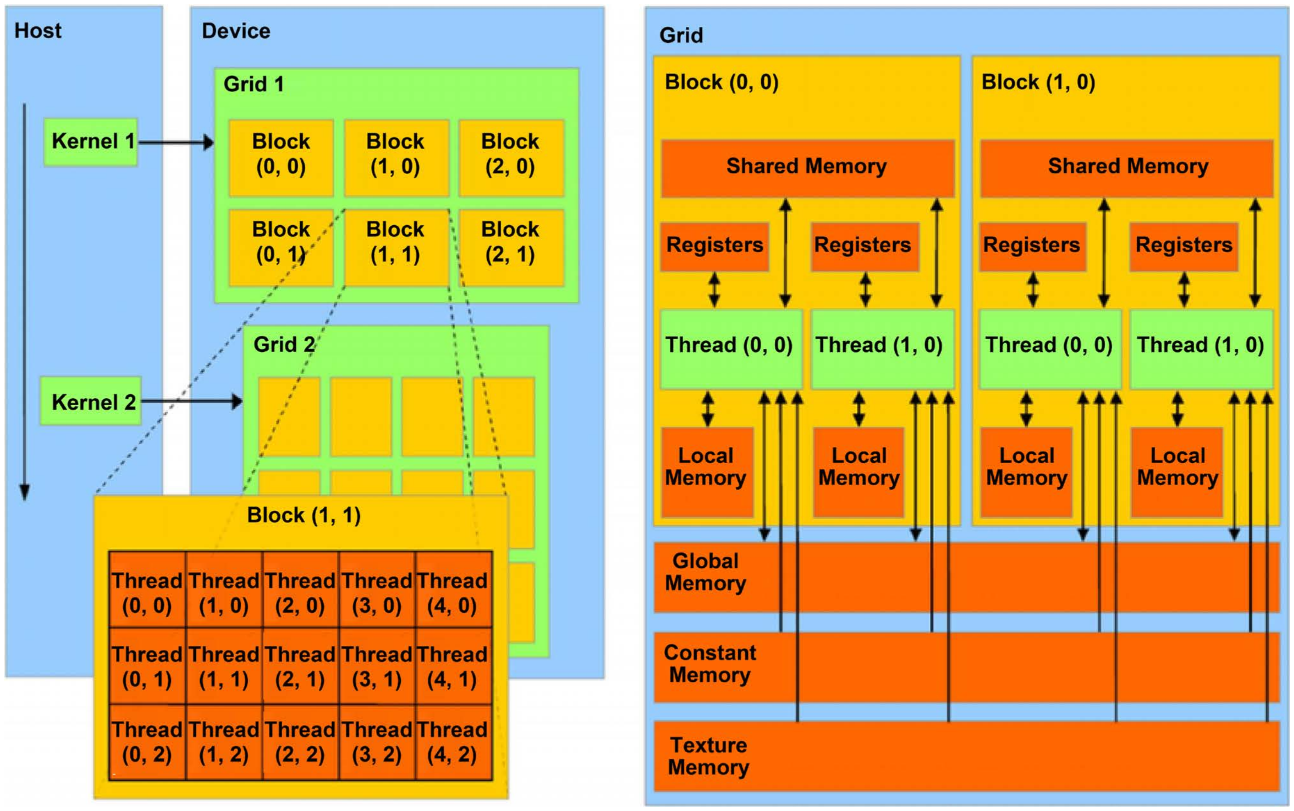


Figure 17. NVIDIA's representation of CUDA execution model and the memory hierarchy in GPUs.

with a granularity of 32 threads known as a warp. A SM has multiple warp schedulers allocating hardware resources to each thread/warp and scheduling the concurrent execution of multiple warps based on the requested shared resources per thread.

The CUDA programming model can be exploited to implement both data level and task level parallelism in the implementation of ARC. The given data is divided into smaller segments and Andrew's monotone chain convex hull algorithm is used on individual segments to construct intermediate convex hulls. Each CUDA thread operates on a segment of data and computes one convex hull. CUDA has the capability to spawn a large number of threads to compute several convex hulls in parallel. Once the intermediate convex hulls are constructed by individual threads, each thread next considers two consecutive convex hulls at a time and constructs common tangents to merge the two hulls. This process is shown in Figure 18. If the number of intermediate convex hulls is N, then N-1 threads are required for to merging these hulls in parallel. The resulting set of points may not form a convex hull as we do not combine the intermediate convex hulls hierarchically but consecutively as explained in Section 5. One of the important observations is that the computations on the upper half are independent of the lower half computations, thereby allowing concurrent compu-

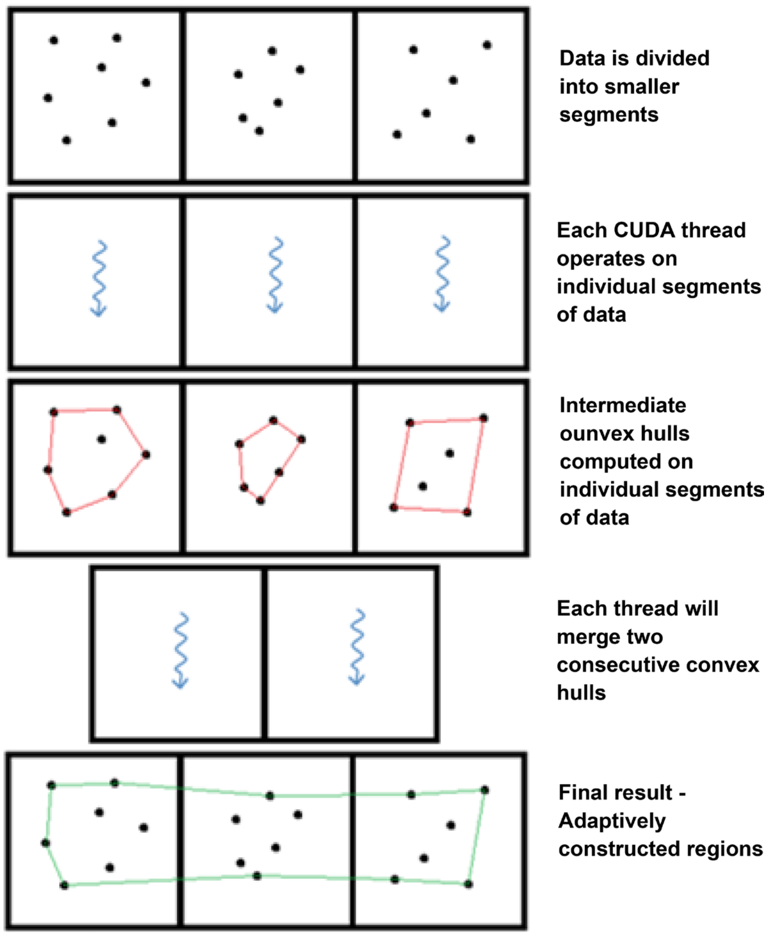

Data is divided into smaller segments
Each CUDA thread operates on individual segments of data
Intermediate ounvex hulls computed on individual segments of data
Each thread will merge two consecutive convex hulls
Final result - Adaptively constructed regions

Figure 18. Scaled down illustration of data level parallelism in our implementation.

tations on the upper and lower half. The CUDA streams approach is used to compute the upper and lower hull in parallel and exploit task level parallelism.

An initial version of the algorithm utilizing both data level and task level parallelism was implemented on a NVIDIA Tesla K40c accelerator and hence forth known as the naïve version. For the naïve version, the number of points processed by each thread was fixed at a value of 4 and the kernel execution time of the naive version is shown in **Figure 19**. Comparing the execution time of the naïve and the sequential versions of Andrew's monotone chain convex hull algorithm (**Figure 7**) shows a 60% improvement in performance. Even though 60% improvement in performance is significant for many applications, the naïve version is still not suitable for real time applications, especially when handling large data points (64 million points take 650 ms for computation of ARC). To improve the performance of the naive version, refactoring the program to the GPU hardware and software architectures is necessary. Therefore, as a first step, the naïve version code is profiled using the NVIDIA visual profiler [19].

6.1. Profiling Analysis of Naive Version

The NVIDIA Visual Profiler [20] is a cross-platform performance analysis tool that provides guidance for optimizing CUDA applications. It helps in the identification of performance bottlenecks and delivers a graphical visualization of the bottlenecks. The profiling results of the naive version are shown in **Figure 20**. The naïve version consists of 4 kernel functions: 2 kernels to construct upper/lower intermediate convex hulls and 2 kernels to merge the intermediate convex hulls consecutively. As we can see in **Figure 20** timeline, the memory transfer time from the host (CPU) to the device (GPU) and vice versa is significantly greater than the computational time, and also the memory transfers are not contiguous. This indicates that the program performance is bottlenecked by the bandwidth of the Peripheral Component Interconnect Express (PCIe) bus. Also, the computations of the kernel functions that operate on the upper/lower halves

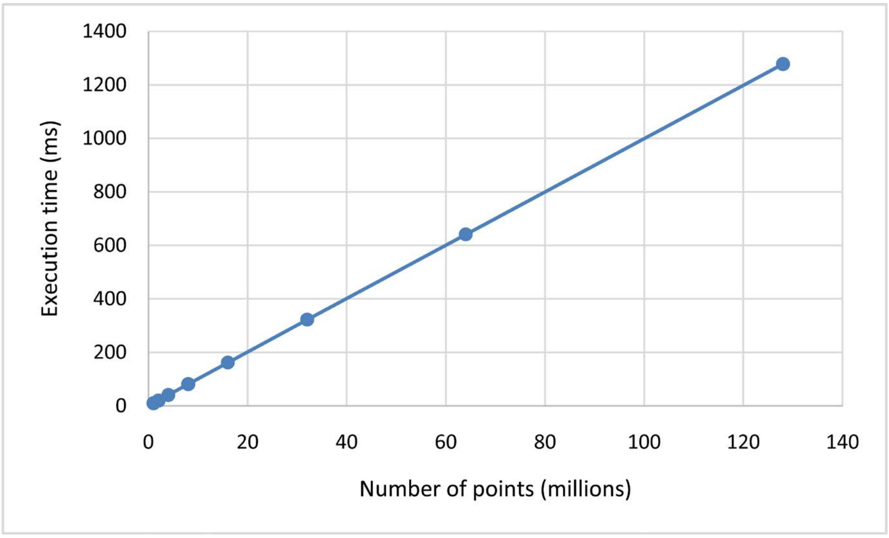


Figure 19. Execution time of the naive version of ARC.

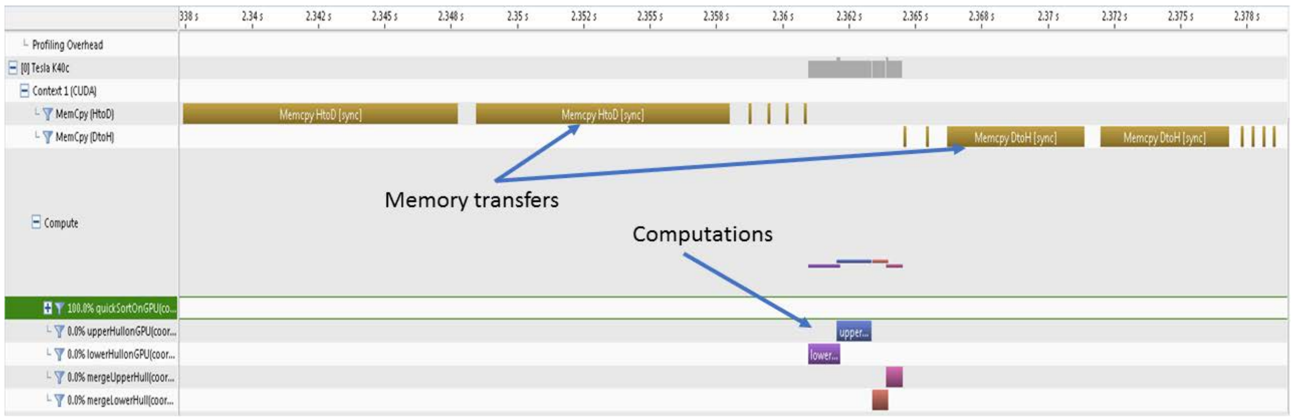


Figure 20. Execution timeline generated by the NVIDIA visual profiler.

of data are not perfectly overlapped. This is mainly due to the GPU being stalled as it waits for all the data required by the kernel to be transferred before starting the computations.

Furthermore, the profiler also identifies additional performance bottlenecks which are summarized below:

- Low warp execution efficiency due to divergent branches

The profiler indicates low warp execution efficiency for the kernel functions signifying the inefficient use of GPUs for computation. The compute resources are best utilized when all the threads in a warp are active. The algorithm is implemented with different control statements that result in branching, and the profiler recognizes 33.2% and 93% divergence in the kernel function that computes intermediate convex hulls and the kernel function that merges the hulls respectively. The number of active threads in an SPMD execution model can be improved by having less divergent branches executing different instructions within the same warp.

- Global memory alignment and access pattern

The profiler identifies inefficient use of memory bandwidth due to misaligned global memory access patterns. As the instructions are issued per warp in an SPMD execution model, 32 threads in a warp cooperatively request a single memory access, which is serviced by one or more memory transactions. Unaligned and non-coalesced memory access due to warp divergence or the pattern of memory addresses requested by each thread can result in inefficient memory accesses. For uncached global memory accesses, the data always flows through the L2 cache, and it performs four 32-byte transactions in a single memory cycle. In ARC, redundant loads of data occur if the threads in a warp access data points such that N mod (128) ≠ 0, where N is the total number of data points accessed by the threads in a warp as shown in Figure 21. Redundant loads could be avoided by making N an integer multiple of 32, however, for efficient utilization of memory bandwidth N must be an integer multiple of 128.

- L2 cache access latency

The profiler records 2.7 million global memory loads performed at a rate of 155.852 GB/s and 5.3 million reads from the L2 cache. The L2 cache reads are

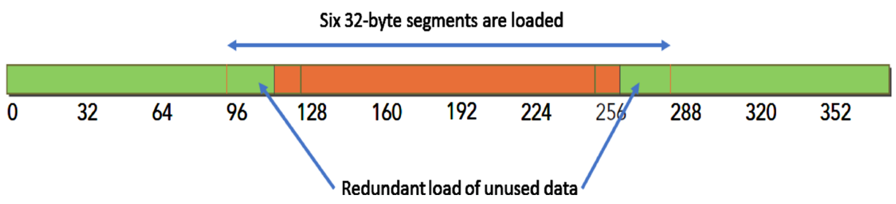

Six 32-byte segments are loaded
0
32
64
96
128
160
192
224
256
288
320
352
Redundant load of unused data

Figure 21. Inefficient use of memory bandwidth due to redundant loads.

Table 1. CGMA for different kernel functions.

Kernel functions	CGMA
lower Hull On GPU	1/2
upper Hull On GPU	1/2
merge Lower Hull	8/15
merge Upper Hull	8/15

higher because the algorithm reuses spatially adjacent data in computations, benefitting by both temporal and spatial locality of data. As an example, we have the arrays that store the size of intermediate convex hulls and the convex hulls themselves accessed repeatedly within the same kernel function and therefore are cached. However, the L2 cache located outside the SMs has significant memory access latency of 100 clock cycles, and this latency can be reduced by moving data that is reused to a cache closer to the SMs. The cache closer to the SMs which can be programmatically controlled in the GPUs is known as the shared memory which has a latency of 12 to 32 clock cycles.

GPUs use DDR5 memory, which is a high bandwidth memory but has latency [21] of 400 - 800 cycles resulting in large memory access latency compared to 10 - 20 cycle latency for arithmetic operations. The memory access latency is hidden due to the multiple threads executing the job at the same time, but it is still necessary to access the memory efficiently. To analyze the memory access efficiency, the compute-to-global memory access ratio (CGMA) for the naive version is determined. CGMA is defined as:

$$\text{CGMA} = \frac{\text{Number off loating point operations}}{\text{Number of global memory accesses}}. \tag{2}$$

If CGMA is significantly greater than 1, the GPU spends more time performing computations rather than fetching data from memory. These types of problems are called compute bound problems. On the other hand, if the CGMA is less or close to 1, the problem is memory bound indicating that the GPU spends most of the time fetching data from the memory rather than computing. Table 1 shows the CGMA of different kernel functions of the naïve version. The CGMA of all the four kernels is significantly less than 1, making the naïve version memory bound. The performance of memory bound problems is limited by the memory bus bandwidth and memory clock speed, which makes it difficult to improve the performance.

In order to improve the performance, we have to increase the CGMA for our implementation. Considering Equation (2), we can either increase the numerator to improve CGMA or decrease the denominator. Increasing the numerator is not a feasible option because increasing the number of floating point operations translates to artificially introducing the computational complexity of the existing algorithm. Therefore, we consider the second option, which is to decrease the value of the denominator. This can be done by reducing the number of global memory accesses and specifically multiple accesses to the same data either on the global memory or L2 cache. We use shared memory, which is a user controlled cache to store chunks of data from global memory. Later, we use the data in the shared memory to perform the computations. This reduces the memory access latency due to multiple accesses of the data on global memory and L2 cache.

7. Optimizations

The profiler analysis of the naïve version along with the CGMA computations provides insights about the possible approaches that can improve performance. This section discusses the various optimization approaches used to improve the performance of the naïve version.

7.1. Shared Memory to Reduce Global Memory Access

To improve L2 cache access latency by reusing on-chip data, and reduce the global memory bandwidth required by the kernels we make use of shared memory. Assuming each thread performs only one iteration of the algorithm, the kernel function that computes convex hulls of one half of the given set of points have to access the global memory 16 times, and the kernel function that combines two consecutive convex hulls has to access the global memory 30 times in the naïve version. The shared memory latency being 12 to 32 cycles is about 50 times lower than the uncached global memory latency [22] and three times lower than the L2 cache. Therefore, a program accessing shared memory instead of global memory or L2 cache performs better. To reduce the number of global memory accesses, we load the data from global memory to shared memory and perform the computations. Later, the result is written back into global memory.

In the kernel function that computes either the upper half or lower half of intermediate convex hulls, the data seen by each thread block is loaded into the shared memory. Each thread computes convex hulls by considering a small number of elements. The number of elements processed by each thread is calculated as the ratio of the total number of points to the total number of threads. Copying the data from the global memory to the shared memory and performing computations using the copied data on the shared memory is shown in Step 1(a) and 1(b) of Figure 22. Once all the threads in a block have finished computing the convex hulls, the resulting points are written back into the global memory. This is illustrated in Step 2 of Figure 22. As discussed previously, two consecutive convex hulls are combined together by constructing common tangents between them. To reduce the number of global memory accesses while constructing tangents, the convex hulls seen by each thread block along with one

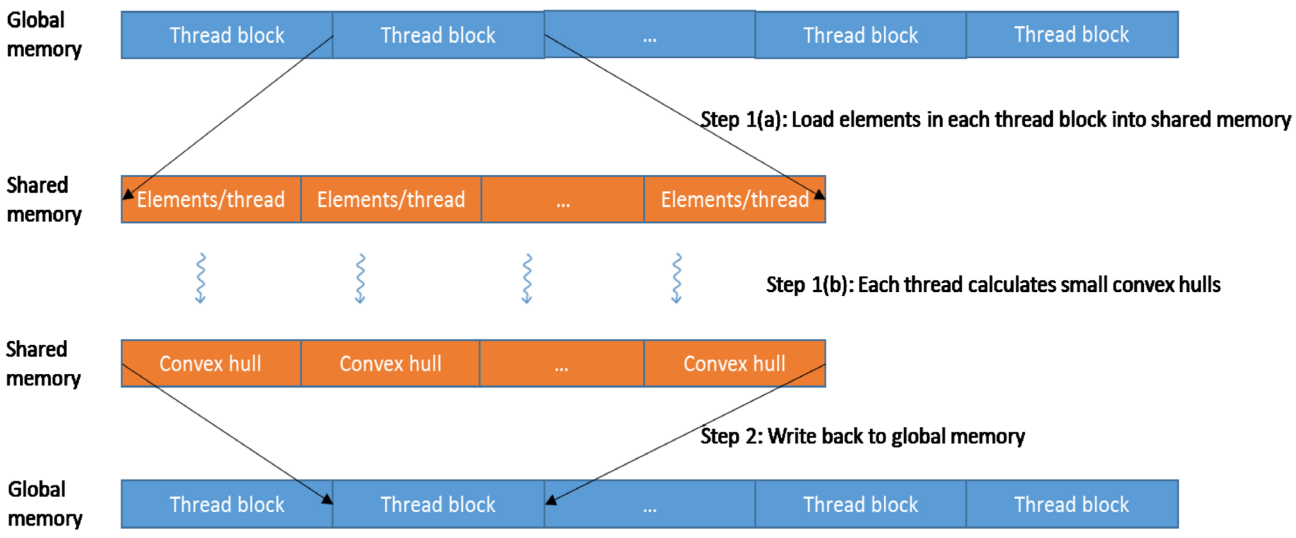


Figure 22. Shared memory implementation of constructing one half (upper/lower) of the convex hulls.

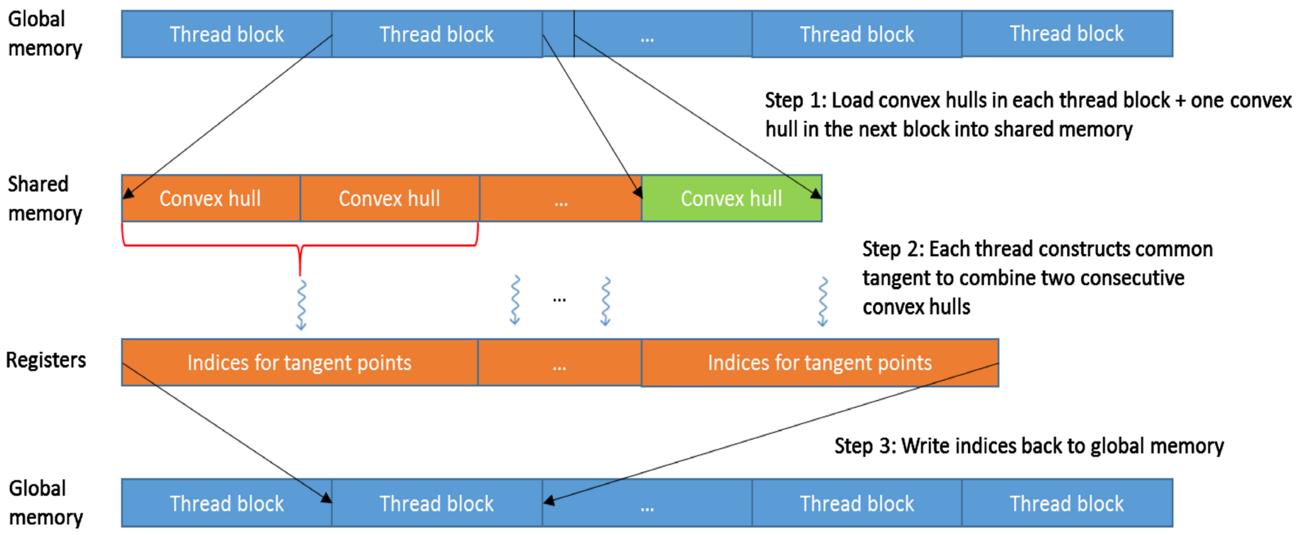

Global memory
Thread block
Thread block
...
Thread block
Thread block
Step 1: Load convex hulls in each thread block + one convex hull in the next block into shared memory
Shared memory
Convex hull
Convex hull
...
Convex hull
Step 2: Each thread constructs common tangent to combine two consecutive convex hulls
Registers
Indices for tangent points
...
Indices for tangent points
Step 3: Write indices back to global memory
Global memory
Thread block
Thread block
...
Thread block
Thread block

Figure 23. Shared memory implementation of combining two convex hulls by constructing common tangents.

convex hull from the next thread block is loaded into the shared memory (Figure 23: Step 1). For a given pair of consecutive convex hulls, only the indices to increment the right hull and the indices to decrement the left hull are stored in registers local to each thread (Figure 23: Step 2). These indices point to the data points that form a common tangent to the two consecutive convex hulls. The indices are written into the global memory after each thread completes its operation (Figure 23: Step 3).

We also use shared memory as a scratchpad memory to store the size of intermediate convex hulls and also enabled L1 caching (16 KB) along with the shared memory (48 KB) to cache global memory transactions.

7.2. Avoiding Warp Divergence

While loading the data into shared memory for combining two consecutive convex hulls, each thread loads one convex hull into the shared memory. But the threads at the end of each thread block (except for the last block) must load two convex hulls, one at the end of thread block and the other from the beginning of the next thread block. This can be easily achieved by using simple control statements on a traditional CPU based computing system. CPUs have complex hardware with advanced branch prediction mechanisms to implement control statements. On a CPU, there are pipelines for each program flow of the control statement. If the predicted branch is false, a CPU can quickly switch to the other pipeline and continue with the execution flow, eluding any significant performance penalty.

On the other hand, GPUs are simple devices with no branch prediction mechanisms requiring all the 32 threads in a warp execute in a synchronous fashion. If different threads in a warp execute different instructions, the GPU flushes the execution pipeline each time to load new instructions resulting in the sequential execution of each branch of the control statement. Also, since all threads in a warp execute in parallel, some of the threads in a warp will be idle and will become active during the upcoming sequence that will make the previously executing threads in that warp idle as shown in Figure 24.

To avoid warp divergence, we loaded both the flow paths of the control statements into the same branch by making use of multiple *if* statements instead of *if-else* chains as exemplified in Figure 25. In this way, both the conditional instructions are loaded into the execution pipeline, and only the statements with conditions resulting to true will be executed by the threads.

7.3. Optimized Memory Access

While accessing global memory, the data has to pass through L2 cache by default, and four 32-byte transactions are performed to fetch 128 bytes of single

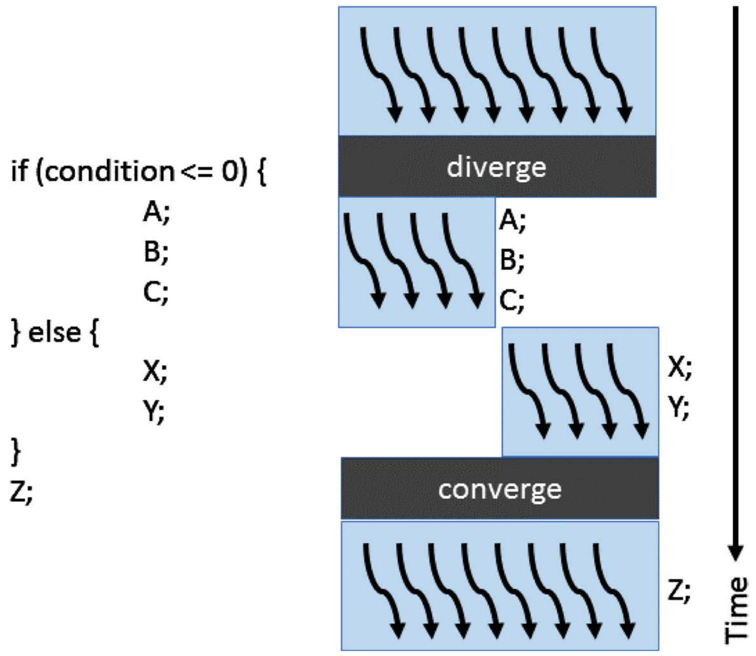


Figure 24. Warp divergence due to *if-else* statements.

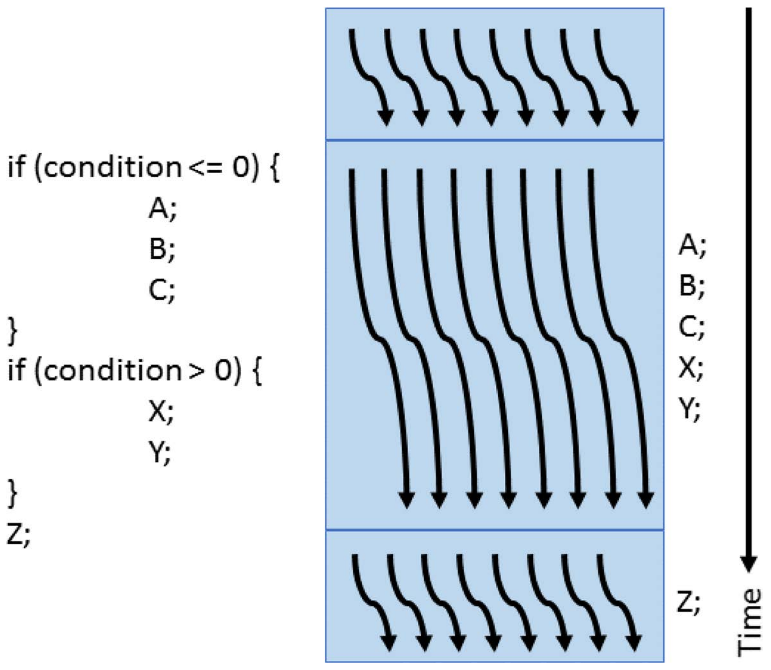


Figure 25. Warp divergence due to *if-else* statements.

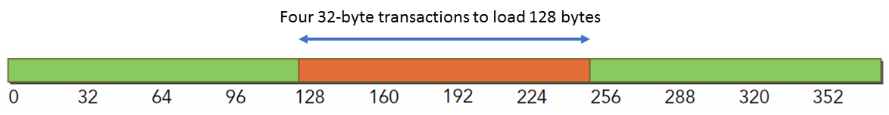


Figure 26. Optimized memory access with L1 caching disabled.

precision data for the threads in a warp. On enabling the L1 cache, a 128-byte transaction request is used to load single precision data for a warp. In other words, NVIDIA GPUs has a L1 cache line granularity of 128 bytes and an L2 cache line granularity of 32 bytes. The memory fetches from the global memory is a major performance bottleneck, and it is necessary to keep the number of load transactions to a minimum. One way to keep the load transactions to a minimum is to load only the required data by a warp and avoid redundant data loads. Figure 26 shows an example where a warp requests 128 bytes of data and the GPU performs four 32-byte transactions to load 128 bytes, and the data is accessed within the same 128-bytesegment (aligned to the 128-byte boundary). This is very efficient when compared to the unaligned redundant load shown in **Figure 21**, where the data requested by a warp requires six 32-byte transactions in three 128-byte segments, and all the data that is loaded is not used by the warp.

We adjust the number of points seen by each thread to construct intermediate convex hulls such that the data requested by a warp is a multiple of cache line granularity depending on the problem size, thereby minimizing redundant loads of data. NVIDIA also reports [23] that the effective bandwidth is poor for strided memory access with strides greater than 8 as the hardware cannot combine the accesses that are far apart in the physical memory. Therefore, adjacent threads in our program access contiguous data points to construct adjacent intermediate convex hulls and do not perform strided access.

7.4. Reducing Host to Device Data Transfer Latency

The transfer of data from the host to the device takes place over the PCIe bus. Even though it is not possible to increase the speed of data transfer due to hardware limitations, it is possible to reduce the time that the GPU spends waiting for data. Data is allocated on the CPU memory as pageable memory. Pageable memory can be swapped into the secondary storage by the operating system to give an illusion of additional main memory than available. Since the GPU does not have control over the paging operation, it takes more time for the data to be transferred from pageable memory to GPU memory. To decrease the data transfer time from CPU memory to GPU memory, we used pinned memory on the CPU. Pinned memory or page-locked memory is a non-swappable memory allocation on the CPU random access memory (RAM) preventing the operating system from swapping the allocated memory to secondary storage. This allows the data transfer between CPU and GPU through the PCIe bus at a higher bandwidth.

8. Results and Analysis

We implemented the ARC technique on a NVIDIA GPU using the CUDA C programming model. The hardware platform consists of an Intel Xeon E5-2620-0 (Sandy Bridge) processor for implementing the sequential Andrew's monotone chain convex hull algorithm and NVIDIA Tesla K40c for implementing the ARC technique.

Sets of random points with a normal distribution to test and compare the optimized implementation of ARC technique with the sequential Andrew's monotone chain convex hull algorithm were generated. Figure 27 shows the execution times of the sequential Andrew's monotone chain convex hull algorithm and the parallel ARC technique kernels of the naïve and advanced versions using the shared memory with the number of elements processed by each thread held constant. The results shown in Figure 27 correspond to the load being equally distributed among CUDA threads with each thread operating on 4 data points to construct the intermediate convex hulls and excludes the memory transfer time over the PCIe bus for the parallel implementations. We can see from Figure 27 that the computation of adaptive regions for 16 million data points takes place in about 14 ms in contrast to 306 ms for the sequential implementation, indicating near real time performance. Figure 28 shows the overall execution time for the same three cases indicated previously and the result includes the data transfer time over the PCIe bus. The naïve parallel version had a larger execution time in comparison to the sequential version due to significant time spent on memory transfers from host to device and vice versa. The considerable time for memory transfers is due to a redundant data transfers required in the naïve version. The memory transfer time was improved by eliminating redundant data transfers and making use of pinned memory in the advanced version, which shows

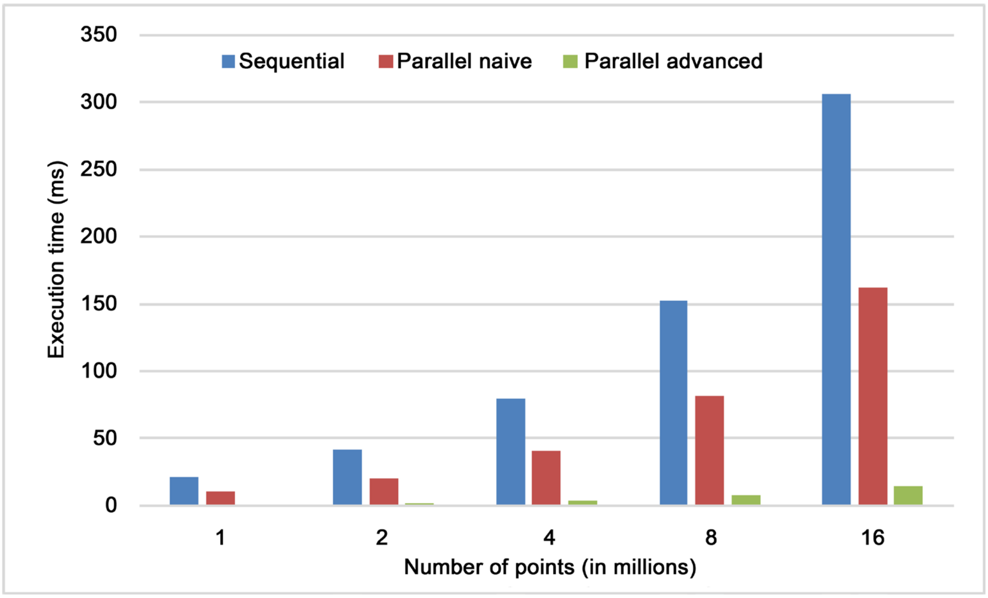


Figure 27. Execution times of sequential, naive parallel kernel, and advanced parallel kernel implementations for computing contours.

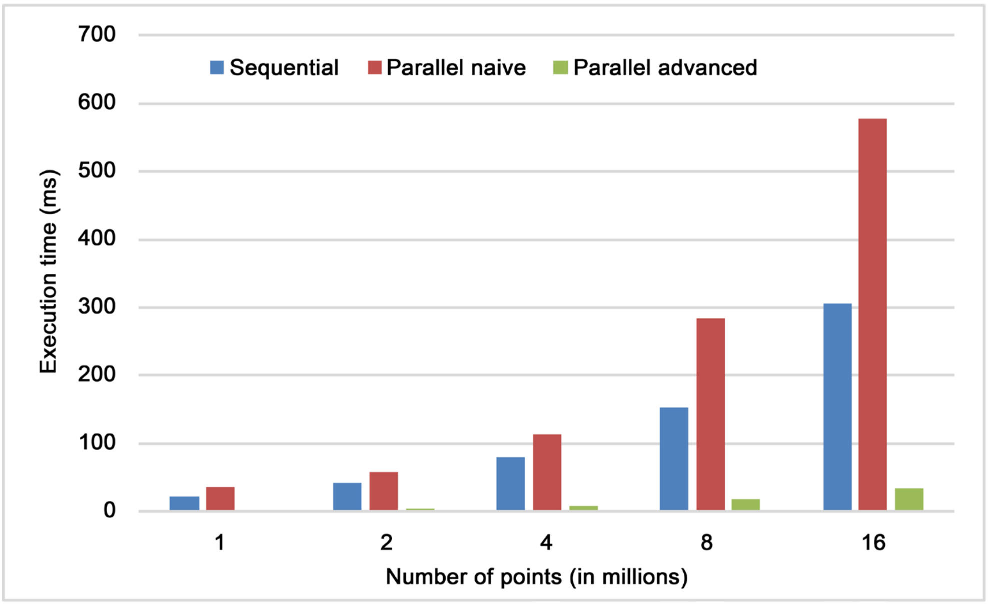


Figure 28. Overall execution times of sequential, naive parallel and advanced parallel implementations for computing contours.

significant improvement in performance. In Figure 29, the speedup between the naïve kernel and sequential version and the advanced kernel and sequential version is presented. The advanced version kernel on an average has a speedup of 21.6× while the naïve version kernel had only a speedup of 2×. The higher speedup of the advanced version clearly demonstrates the effectiveness of the optimizations applied to the naïve version. Taking into account the memory transfer time for the advanced versions, the average speedup achieved is 9.3× as shown in Figure 30. The overall improved speedup of the advanced version can be attributed to the use of pinned memory instead of the paged memory in the naïve

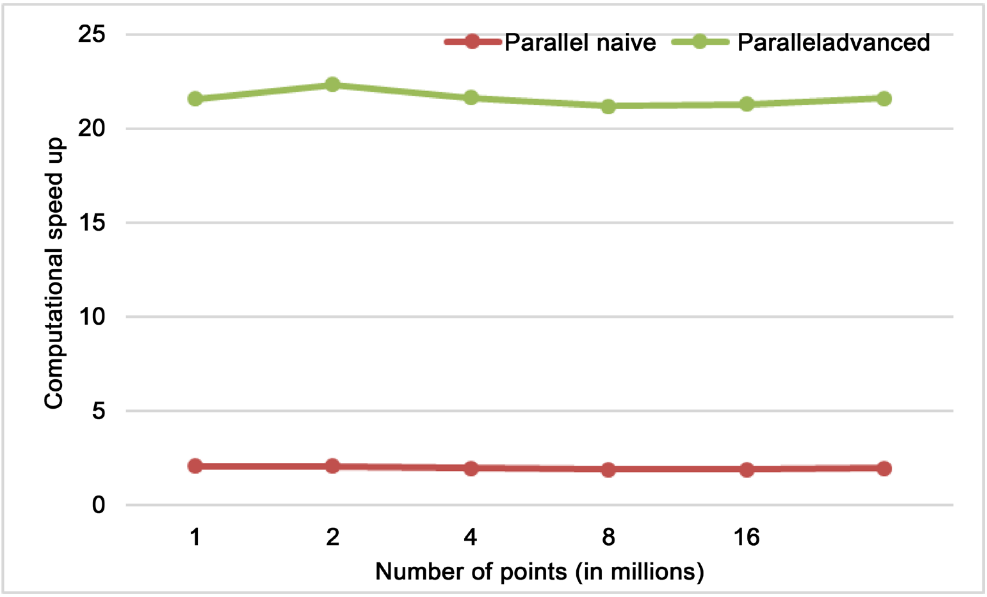


Figure 29. Computational speedup of naive and advanced parallel versions.

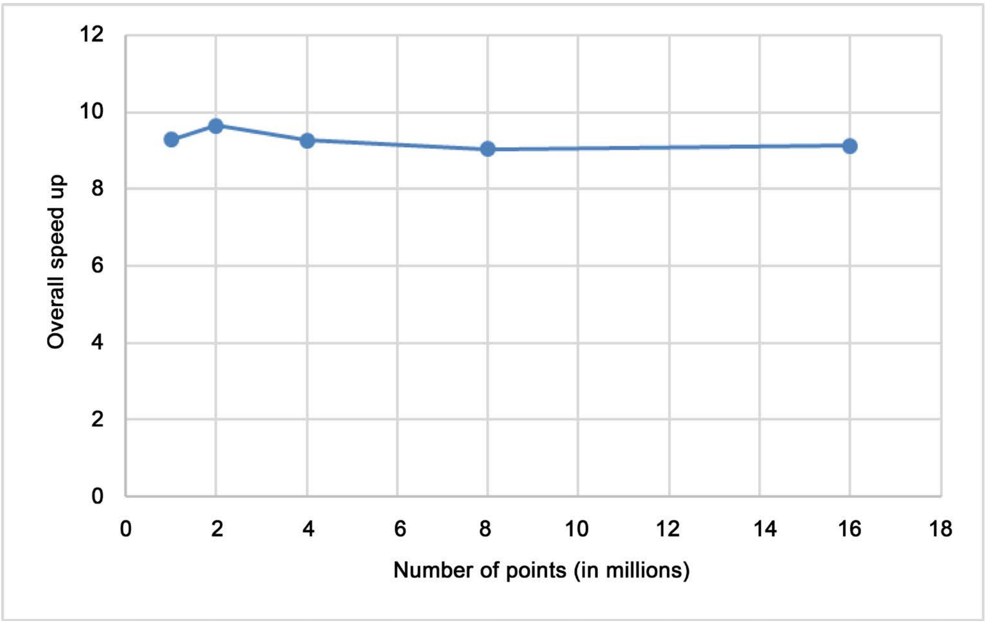


Figure 30. Overall speedup of the advanced parallel version.

version in addition to the use of shared memory. Even though an overall 9.3× speedup has been achieved, the speedup remains constant with increasing number of points as depicted in Figure 30 indicating weak scaling of the problem. As the number of points in increased, correspondingly, the number of threads is also increased as the number of points per thread is held constant. However, the number of cores or processors in a GPU is constant. According to Gustafson's law [24], the workload is scaled up to maintain a fixed execution time as the number of processor increases; the speedup increases linearly. Since the number of GPU cores is not increasing with increasing workload, the speedup has to remain constant or decrease with increasing workload.

The second goal for using the ARC technique was to eliminate the artifacts present in the contour of a radio propagation map determined using the convex hull approach. Given a set of points, the ARC technique can either construct a convex hull or a set of points, which is not a convex hull representing the contour of a radio propagation map accurately. If the ARC technique is forced to use a single thread, *i.e.*, a sequential construct, the set of points obtained using ARC will match the convex hull. However, by varying the number of threads, the result can be a non-convex hull with varying levels of granularity. The resulting set of points obtained using ARC is selected by computing a number of intermediate convex hulls to fit the given set of points. These intermediate convex hulls are merged consecutively in order to obtain the resulting set. In other words, the number of intermediate convex hulls constructed represents the "resolution" or detail with which the radio propagation map is approximated. In our implementation, since each thread constructs one intermediate convex hull, the resolution of approximation will depend on the number of threads. Decreasing the number of threads decreases the number of intermediate convex hulls, and degrades the application performance as the load handled by each thread increases. ARC does not result in the contour of a radio propagation map directly but also includes points inside the contour that are eliminated using simple techniques [25] if a contour is desired. In **Figure 31**, the original radio propagation map, convex hull based, and ARC based contours are shown for increasing number of points. We can see that the contours generated based on the ARC technique has eliminated the artifacts present in the convex hull based contour and also accurately represent the contour of the original radio propagation map.

9. Conclusions

The technique of adaptive region construction is a low complexity approach that can represent the given contour with varying degrees of details. Adaptive region construction technique provides the capability to construct the contour of a radio propagation map efficiently. The implementation of the adaptive region construction technique on a GPU using the CUDA programming model has been demonstrated. The GPU implementation provides good application performance (speedup) for high resolution representation of contours but is not suitable for low resolution representations. By applying optimization techniques to the naïve version, a 21× improvement in computational performance for large data sets was achieved. As most of the applications that use radio propagation maps are benefited by the detailed representation of radio propagation maps, the ARC technique fulfills the necessity for a fast algorithm. The ARC technique is not only suitable for real-time operation but also avoids artifacts in contrast to the contours determined using the convex hull approach.

In addition to using the ARC technique for determining the contour of a radio propagation map, it is also possible to approximate other spatial data. Using the ARC, multi-resolution representation of the spatial data is possible. The multi-resolution representation of large spatial data sets allows improved processing

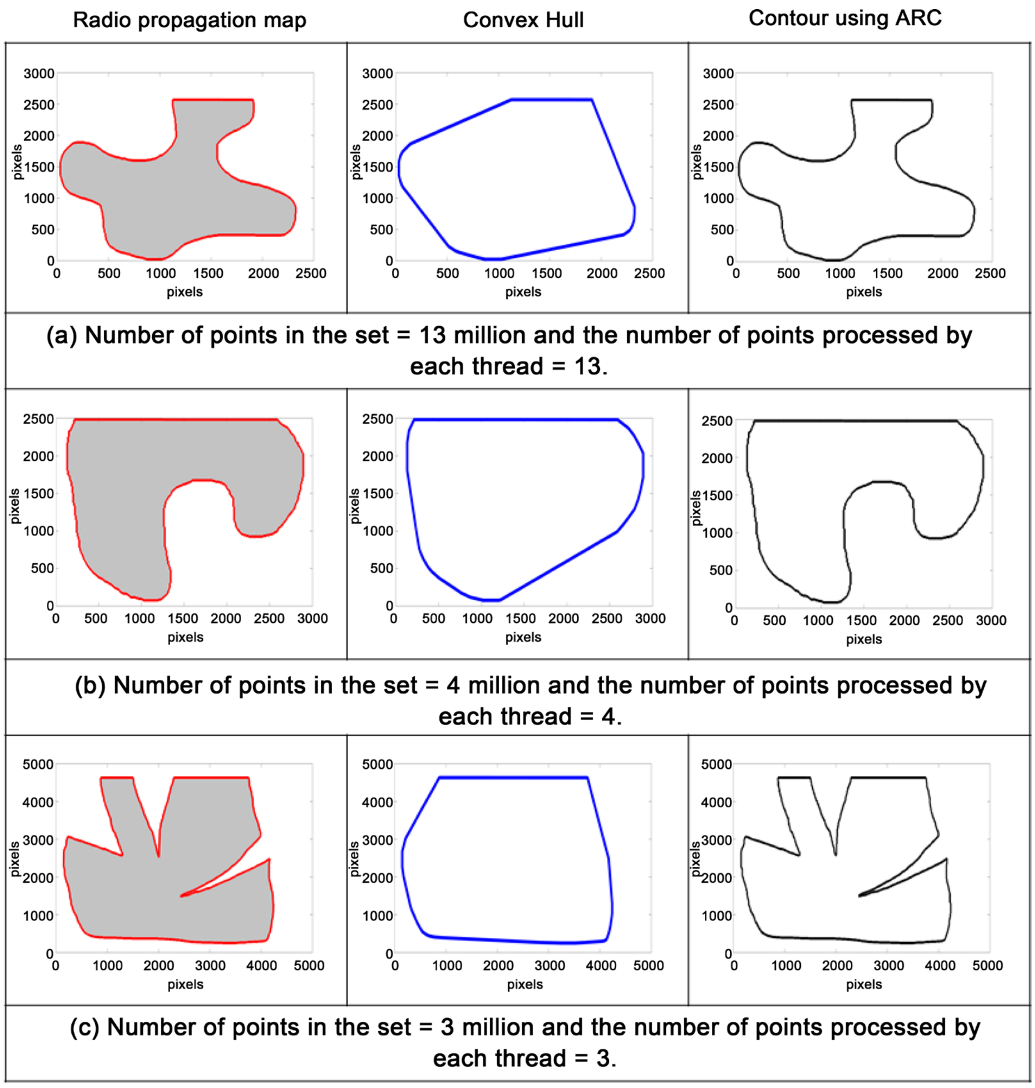


Figure 31. Contours of radio propagation map using ARC.

time and lower storage requirements.

The ARC technique as mentioned previously is inefficiently operating on low resolution radio propagation maps. Also, with large resolution, special attention has to be paid to the memory transfers between the CPU and the GPU. However, with the newer versions of the NVIDIA GPU equipped with the NVlink technology, the latency due to memory transfers is significantly reduced.

References

[1] Haslett, C. (2008) Essentials of Radio Wave Propagation. Cambridge University Press, Cambridge.

[2] Miles, J., Muknahallipatna, S., Kubichek, R.F., McInroy, J. and Muralidhara, H. (2014) Use of Radio Propagation Maps in a Single Moving Beacon Assisted Localization in MANETs. 2014 *International Conference on Computing, Networking and Communications* (*ICNC*), Honolulu, 3-6 February 2014, 871-877.

[3] Miles, J., Kamath, G., Muknahallipatna, S., Stefanovic, M. and Kubichek, R.F. (2013) Optimal Trajectory Determination of a Single Moving Beacon for Efficient Locali-

zation in a Mobile Ad-Hoc Network. *Ad Hoc Networks*, **11**, 238-256.
https://doi.org/10.1016/j.adhoc.2012.05.009

[4] Muralidhara, H. and Kubichek, R. (2011) MANET Localization Using Non-Circular Overlapping Range Maps. *International Conference on Wireless Networks*, Las Vegas, 18-21 July 2011, 8-12.

[5] Ramakrishnaiah, V.B., Kubichek, R.F. and Muknahallipatna, S.S. (2015) Optimization of Antenna Beam Pattern in Ad Hoc Networks for Optimal Global Performance. *2015 IEEE 58th International Midwest Symposium on Circuits and Systems* (*MWSCAS*), Fort Collins, 2-5 August 2015, 1-4.

[6] Andrew, A.M. (1979) Another Efficient Algorithm for Convex Hulls in Two Dimensions. *Information Processing Letters*, **9**, 216-219.

[7] O'Rourke, J. (1993) Computational Geometry in C. Cambridge University Press, Cambridge.

[8] Douglas, D.H. and Peucker, T.K. (1973) Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or Its Caricature. *Cartographical: The International Journal for Geographic Information and Geovisualization*, **10**, 112-122. https://doi.org/10.3138/FM57-6770-U75U-7727

[9] Ramer, U. (1972) An Iterative Procedure for the Polygonal Approximation of Plane Curves. *Computer Graphics and Image Processing*, **1**, 244-256.
https://doi.org/10.1016/S0146-664X(72)80017-0

[10] Prakash, N. and Sarkar, A. (2015) Development of Shape Based Leaf Categorization," *IOSR Journal of Computer Engineering* (*IOSR-JCE*), **17**, 48-53.

[11] Chen, Z., Delis, A. and Bertoni, H.L. (2004) Building Footprint Simplification Techniques and Their Effects on Radio Propagation Predictions. *The Computer Journal*, **47**, 103-133. https://doi.org/10.1093/comjnl/47.1.103

[12] Catrein, D., Reyer, M. and Rick, T. (2007) Accelerating Radio Wave Propagation Predictions by Implementation on Graphics Hardware. *IEEE Vehicular Technology Conference*, Dublin, 22-25 April 2007, 510-514.
https://doi.org/10.1109/vetecs.2007.116

[13] Cheng, M.X., Ling, Y. and Sadler, B.M. (2014) Wireless Ad Hoc Network Connectivity Assessment and Relay Node Deployment. 2014 *IEEE Global Communications Conference*, Austin, 8-12 December 2014, 399-404.
https://doi.org/10.1109/GLOCOM.2014.7036841

[14] Liu, D., Lee, M.-C., Pun, C.-M. and Liu, H. (2013) Analysis of Wireless Localization in Nonline-of-Sight Conditions. *IEEE Transactions on Vehicular Technology*, **62**, 1484-1492. https://doi.org/10.1109/TVT.2013.2244928

[15] Boyd, S. and Vandenberghe, L. (2004) Convex Optimization. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9780511804441

[16] Preparata, F.P. and Hong, S.J. (1977) Convex Hulls of Finite Sets of Points in Two and Three Dimensions. *Communications of the ACM*, **20**, 87-93.
https://doi.org/10.1145/359423.359430

[17] Preparata, F.P. and Shamos, M.I. (1985) Computational Geometry. Springer, Berlin.
https://doi.org/10.1007/978-1-4612-1098-6

[18] Manca, E., Manconi, A., Orro, A., Armano, G. and Milanesi, L. (2016) CUDA-Quicksort: An Improved GPU-Based Implementation of Quicksort. *Concurrency and Computation*: *Practice and Experience*, **28**, 21-43.
https://doi.org/10.1002/cpe.3611

[19] NVIDIA (2017) NVIDIA Developer Zone—CUDA Toolkit Documentation.
http://docs.nvidia.com/cuda/index.html#axzz4fZgKptTq

[20] NVIDIA (year) NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler

[21] Cheng, J., Grossman, M. and McKercher, T. (2014) Professional CUDA C Programming. John Wiley & Sons, Indiana.

[22] Harris, M. (2013) Using Shared Memory in CUDA C/C++. https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/

[23] Harris, M. (2013) How to Access Global Memory Efficiently in CUDA C/C++ Kernels. https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

[24] Gustafson, J.L. (1988) Reevaluating Amdahl's law. *Communications of the ACM*, **31**, 532-533. https://doi.org/10.1145/42411.42415

[25] Nitzberg, M., Mumford, D. and Shiota, T. (1993) Filtering, Segmentation and Depth. Springer-Verlag New York, Inc., Secaucus. https://doi.org/10.1007/3-540-56484-5