POINT IN POLYGON (PIP) SEARCH ACCELERATION USING GPU BASED QUADTREE FRAMEWORK

by

SREEVIDDYADEVI KARUNAGARAN

B.E., Anna University, India, 2008

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Electrical Engineering

2016

This thesis for the Master of Science degree by

SREEVIDDYADEVI KARUNAGARAN

has been approved for the

Department of Electrical Engineering

by

Dan Connors, Chair

Yiming Deng

Chao Liu

April 14, 2016

KARUNAGARAN, SREEVIDDYADEVI (M.S., Electrical Engineering)

POINT IN POLYGON (PIP) SEARCH ACCELERATION USING GPU BASED QUADTREE FRAMEWORK

Thesis directed by Professor Dan Connors

ABSTRACT

The point-in-polygon (PIP) problem defines for a given set of points in a plane whether those points are inside, outside or on the boundary of a polygon. The PIP for a plane is a special case of point range finding that has applications in numerous areas that deal with processing geometrical data, such as computer graphics, computer vision, geographical information systems (GIS), motion planning and CAD. Point in range search can be performed by a brute force search method, however solution becomes increasingly prohibitive when scaling the input to a large number of data points and distinct ranges (polygons). The core issue of the brute force approach is that each data point must be compared to the boundary range requiring both computation and memory access. By using a spatial data structure such as a quadtree, the number of data points computationally compared within a specific polygon range can be reduced to be more efficient in terms of performance on the CPU. While Graphics Processing Unit (GPU) systems have the potential to advance the computational requirements of a number of problems, their massive number of processing cores execute efficiently only in certain rigid execution models such as data-level parallel problems. The goal of this thesis is to demonstrate that the GPU systems can still process irregular data structure such as a quadtree and that it can be efficiently traversed on a GPU to find the points inside a polygon. Compared with an optimized serial CPU implementation based on the recursive Depth-First Search (DFS), the stack based iterative Breadth-first search (BFS) on the GPU has a performance gain of 3x to 500x.

The form and content of this abstract are approved. I recommend its publication.

Approved: Dan Connors

TABLE OF CONTENTS

Appendix

TABLES

Table

FIGURES

Figure

1. Introduction

The point-in-polygon (PIP) problem defines for a given set of points in a plane whether those points are inside, outside or on the boundary of a polygon. The PIP for a plane is a special case of point range finding that has applications in numerous areas that deal with processing geometrical data, such as computer graphics, computer vision, geographical information systems (GIS), motion planning and CAD. Point in range search can performed by a brute force search method, however for solution becomes increasingly prohibitive when scaling the input to a large number of data points and distinct ranges (polygons).

Range search algorithms, which make use of spatial data structures, perform much better than the ones that do not partition the data before processing. Quadtree is a hierarchical spatial data structure that is used for both indexing and compressing geographic database layers due its applicability to many types of data, its ease of implementation and relatively good performance. As done traditionally, the quadtree is built on the CPU. To speed up the range searching problems, it is essential to increase the threshold on the number of queries processed within a given time frame. Purely sequential approach to this will demand increase in processor speeds.

Graphics Processing Units (GPUs) have proven to be a powerful and efficient computational platform. An increasing number of applications are demanding more efficient computing power at a lower cost. The modern GPU can natively perform thousands of parallel computations per clock cycle. Relative to the traditional power of a CPU, the GPU can far out-perform the CPU in terms of computational power or Floating Point Operations per Second (FLOPS). Traditionally GPUs have been used exclusively for graphics processing. Recent developments have allowed GPUs to be used for more than just graphics processing and rendering. With a growing set of applications these new GPUs are known as GPGPUs (General Purpose GPUs). NVIDIA® has developed the CUDA (Compute Unified Device Architecture) API

(Application Programming Interface) which enables software developers to access the GPU through standard programming languages such as 'C'. CUDA gives developers access to the GPU's virtual instruction set, onboard memory and the parallel computational elements. Taking advantage of this parallel computational power will result in significant speedup for multiple applications. One such application is computer vision algorithms. From the assembly line to home entertainment systems, the need for efficient real-time computer vision systems is growing quickly. This paper explores the potential power of using the CUDA API and NVIDIA® GPUs to speedup common computer vision algorithms. Through real-life algorithm optimization and translation, several approaches to GPU optimization for existing code are proposed in this report.

In the past few years, there has been a rapid adoption of GPGPU parallel computing techniques for both high-performance computing and mobile systems. As GPUs exploit models of data-parallel execution that generally describes a common task across different parallel computing elements, there can be severe limitations for any irregular individual thread behaviors. Irregular execution disrupts the efficiency of the rigid GPU groups of threads by causing workload disparity that effectively leads to idle or underutilized resource units. Most research focus is on the hardware aspects of execution disparity such as branch divergence [7], local memory bank conflicts [5] and non-coalesced global memory accesses [15]. There are a number of proposed architecture concepts to mitigate some of the performance downfalls of handling non-uniform data on GPU architectures [10]. Many of the proposed solutions reduce the frequency of the occurrences but do not fully address the inherent run-time execution characteristics of an algorithm. Overall, irregular data patterns limit the performance potential of GPGPU algorithms.

While irregular algorithms restrain the full use of data-level resources of GPU systems, GPU implementations may still achieve performance benefit over multicore

implementations. In effect, the raw number of GPU resources serves as a brute-force method of carrying out computations with significant arithmetic intensity. Nevertheless, there are alternative and emerging software-based models of distributing execution tasks on GPUs such as dynamic parallelism support. Another technique proposed is persistently scheduled thread groups [6] that abandons the traditional data-level model for stable thread groups assigned to GPU compute units that dynamically retrieve computation tasks from software-based workload queues. The result of persistently scheduled groups can be better load balance, utilization and reduced overhead in thread block creation. At the same time, such techniques have not fully addressed exploiting patterns and variations in model data specific to algorithms.

Generally tasks that involve irregular data or non-deterministic algorithms are not effectively mapped to GPU systems. For example, in graph-based algorithms, the irregular nature of the edge connectivity between graph nodes is not well suited for data-level task definition on GPU computing units. In this case, a group of neighboring GPU threads may be assigned a diverse workload of processing nodes with a few edges as well as nodes with thousands of edges. This form of imbalance is characterized as *static workload disparity* as a portion of the runtime utilization can be traced to the static connectivity of graph nodes. Only if a graph's structure is persistent, not changing over several evaluations, might there be well-reasoned opportunities to reorganize the data, effectively performing static load balancing in which each GPU thread group is assigned data with less variation in work. However, in such cases, there is cost to the partitioning graph nodes to the model data.

This thesis investigates the potential of processing quadtrees for PIP search problems that execute on GPUs. As GPUs operate in a heterogeneous system in which both the CPU and GPU perform some fraction of the computational work, there are unique performance constraints to explore. This thesis considers two primary parameters in scaling optimal GPU quad-tree solutions: data point problem size and

characteristics of polygons being searched.

This thesis is organized as follows: Chapter 2 discusses the motivation and background of computer vision applications. Chapter 3 examines several examples of the PIP problem solving on GPUs. The experimental results section, Chapter 4, shows performance data for the various optimization cases. Finally, Chapter 5 concludes this thesis.

2. Background and Motivation

2.1 Quadtree Background

A quadtree is a tree data structure that designates each internal node to have exactly 4 children. Quadtrees are used to partition a 2D space by recursively subdividing it into 4 quadrants or regions. The regions may be square or rectangular or may have arbitrary shapes. Quadtrees may be classified according to the type of data represented such as areas, points, lines and curves. Figure 2.1 demonstrates a set of 2D data points that have been used to compose a quadtree. The reasoning for using a quadtree structure is that any region search that requires the data points within a polygon area simply can reference the tree rather than the data. There are clear search benefits from the data to tree representations. In the case of the lower right quadrant, any region search will consult the tree structure and determine that only a limited number of point values (X, Y) are required for comparison. In short, the tree representation saves the computational calculations for the polygon search by immediately directing that only a limited number of points exist in that entire quadrant. In the comparison case, it is easier for a polygon that resides in the lower right quadrant to consult the tree versus consult the full data set using full brute force checking of every datapoint within the polygon range.

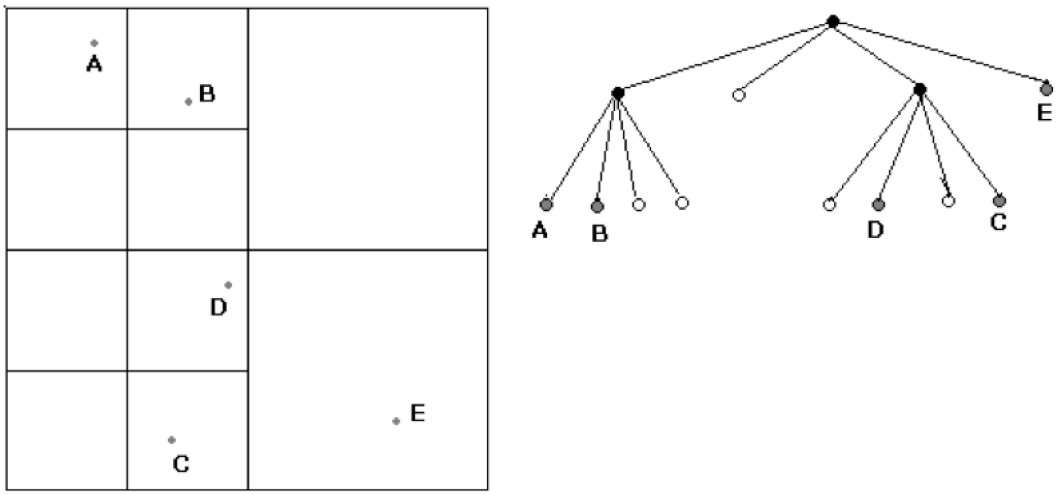

Figure 2.1: Example of Data Points Mapped within a Quadtree Data Structure.

2.1.1 Quadtree Construction

Figure 2.2, Figure 2.3, Figure 2.4, Figure 2.5, demonstrates quadtree construction on the CPU up to level 4. The Figure 2.2, shows the sample data points at level 1 and it represents the root node containing all the data points. The Figure 2.3, shows the sample data points at level 2 and it represents the four child nodes of the root node. The Figure 2.3 is further subdivided to generate level 3 of the quadtree as shown in Figure 2.4. The Figure 2.5, represents the tree at level 4. There are two empty nodes at level 4, as there no points in that direction.

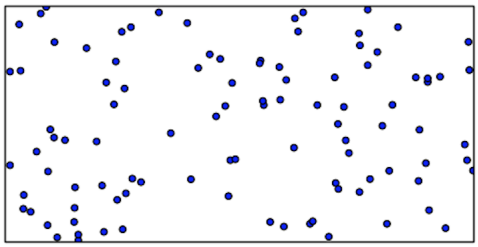

Figure 2.2: Level 1 of Quadtree.

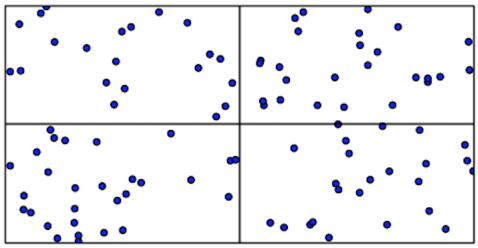

Figure 2.3: Level 2 of Quadtree.

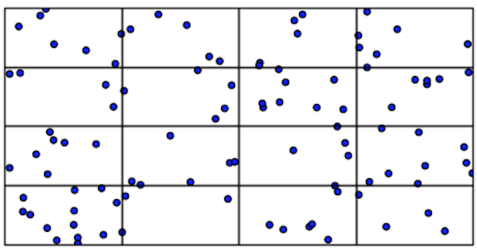

Figure 2.4: Level 3 of Quadtree.

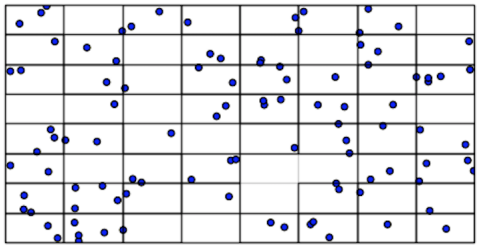

Figure 2.5: Level 4 of Quadtree.

The point quadtree structure recursively divides space into four equal rectangles based on the location of the points. The root of the quadtree represents the whole 2D space. Successive points divide each new sub-region into quadrants until all points are indexed. With each subsequent level, quadtrees can have a maximum of four children for every parent node. Each node represents a region in the 2D space which contains a subset of the data points.

The common algorithm to construct a quadtree is based on sequential point insertion and this sequential approach is not suitable for massively parallel processors. Therefore the quadtree construction is done on the CPU. The performance of quadtrees, for Insertion is O(n log n), for searching is O(log n) and optimization for the purposes of balancing the tree is O(n log n) [8]. This makes quadtrees an ideal data structure for multiple insertion of points and range searching [8]. Though the linear quadtrees reduce the overhead involved in transferring the tree from the CPU main memory to the GPU memory, the pointer based quadtree has several other advantages. The pointer based quadtrees are more memory space efficient and it can be accessed in any traversal order by following the pointers whereas the linear quadtrees only allow preorder traversal and require logarithm time searches to find the next child for any other order [14].

The quadtree is built using pointers. The tree is built by recursive function calls and sequentially inserting the points. Each parent node has pointers to all of its children. There are 3 structures in the quadtree, the node, the point and the point buffers.

Each node is one of three types which are the root node, link node and the leaf node. The root node, corresponds to the entire array of input points. The four children of the root node represent the 4 quadrants (Northwest - NW, Northeast - NE, Southwest - SW, Southeast - SE). The link node is a node that can be further subdivided and the leaf nodes correspond to the quadrants for which further subdivision is not necessary.

The point structure is the input points to the quadtree which occupies a 2D space. A leaf node has a list of point buffers, each of which holds predetermined maximum number of points. Starting at the root, the quadrant or the direction (NW, NE, SW or SE) in which the input point lies is determined and a child node is built in that direction. This step is repeated till the leaf node is reached.

The Figure 2.6 shows the node index and its corresponding direction on its parent node. Then the point is added to that leaf node's buffer of points. If the buffer exceeds some predetermined maximum number of elements, the points are moved into the next buffer in the buffer list. The root node is subdivided recursively. The quadtree with k levels including the root would have $(4(k-1))$ nodes on the k^{th} level and ((4k) -1) nodes in total [8].

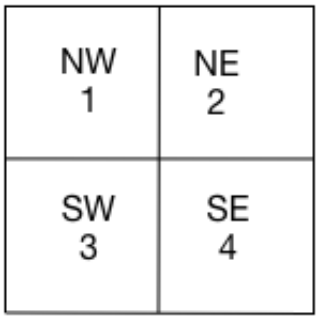

Figure 2.6: NW - North west, NE - North east, SW - South west and SE - South east.

2.2 Related Work

Traversal path for each polygon may differ, therefore linearizing the tree based on the traversal path for each polygon and iterating over the linear traversal order is computationally intensive. Several application-specific approaches have been proposed to handle the problem. In algorithms like Barnes-Hut, a point's traversal can be computed without executing the entire algorithm and a preprocessing pass can determine each point's linear traversal, avoiding repeatedly visiting interior nodes during vector operations. However, for PIP search, where a point's full traversal is only determined as the traversal progresses, the preprocessing step can be expensive.

Goldfarb et al. [2015] [4] demonstrates a GPU implementation of tree traversal algorithms by installing ropes, extra pointers that connect a node in the tree not to its children, instead to the next new node that a point would visit if its children are not visited. But the drawback of using ropes is that it requires an additional traversal prior to performing the actual traversals to create the ropes into the nodes of the tree structure. As a result, earlier attempts to use ropes to accelerate tree traversals have relied on application-specific transformations that leverage the semantics of the

algorithm to efficiently place ropes. The problem with using "Autoropes" [4] is that since the rope pointers are computed during traversal and they are stored on the stack, it causes overhead due to stack manipulation and the efficiency is compromised.

Work by Zhang et al. [2013] [16] presents a speed up of 90x by constructing a quadtree using parallel primitives by transforming a multidimensional geospatial computing problem into chaining a set of generic parallel primitives that are designed for one dimensional arrays.

Another work by Bedorf et al. [2015] [1] presents an octree construction where the tree is constructed on the GPU after mapping particles in 3D space to linear array. And a vectorized BFS traversal is done on the octree.

All of these implementations have relied on linearizing the tree for BFS traversal on the GPU or they have preprocessed the tree to either linearize or install extra pointers on the tree. CUDA GPU ray traversal through a hierarchical data structure such as a bounding volume hierarchy (BVH) is usually carried out using depth-first search by maintaining a stack. This paper explores the parallelization of breadth-first search (BFS) to implement range search on GPUs. The kernel is optimized to minimize thread divergence and memory access. BFS is used instead of depth-first search (DFS) as it is easier to parallelize and implement efficiently on the GPU. BFS and DFS are fundamental building blocks for more sophisticated algorithms used in many computational fields that range from gaming, image processing to social network analysis. BFS is used as a core computational kernel in a number of benchmark suites, including Rodinia, Parboil and the Graph500 supercomputer benchmark.

2.3 Graphics Processing Unit Architecture

The underlying architecture of the GPU is optimized for data-level parallelism. Taking a closer look at the NVIDIA® GPU reveals the chip is organized into an array of highly threaded streaming multiprocessors (SMs). Each streaming multiprocessor

consists of several streaming processors (SPs). Streaming multiprocessors are grouped into thread processing clusters (TPCs). The number of SPs per SM depends on the generation of the chip.

NVIDIA® has chips that increase the number of streaming processors (SP) from 240 to 2048. Each SP has a multiply-add unit plus an additional multiply unit. The combined power of that many SPs in this GPU exceeds one teraflop [9]. Each SM contains a special function unit which can compute floating-point functions such as square root and transcendental functions. Each streaming processor is threaded and can run thousands of threads per application. Graphics cards are commonly built to run 5,000-12,000 threads simultaneously on this GPU. The GTX680 can support 192 threads per SMX and a total of 1536 threads simultaneously [12]. In contrast, the Intel® Core™i7 series can support two threads per core.

GPUs are optimized via the execution throughput of a massive number of threads. The hardware takes advantages of this by switching to different threads while other threads wait for long-latency memory accesses. This methodology enables very minimal control logic for each execution thread [9]. Each thread is very lightweight and requires very little creation overhead.

From a memory perspective, the GPU is architected quite differently than a CPU. Each GPU currently comes with up to four gigabytes of Graphics Double Data Rate (GDDR) DRAM which is used as global memory. The GPU architecture is designed to exploit arithmetic intensity and data-level parallelism.

Figure 2.7 shows the architecture of the GTX680. This generation of the CUDA enabled GPU devices consists of an array of 8 next generation streaming multiprocessors (SMX), 4 GPCs and 4 memory controllers. Each GPC has a dedicated raster engine and two SMX units. Figure 2.8 shows the architecture of a SMX. Each SMX unit contains 192 cores and four warp schedulers. Each warp scheduler is capable of dispatching two instructions per warp every clock [12].

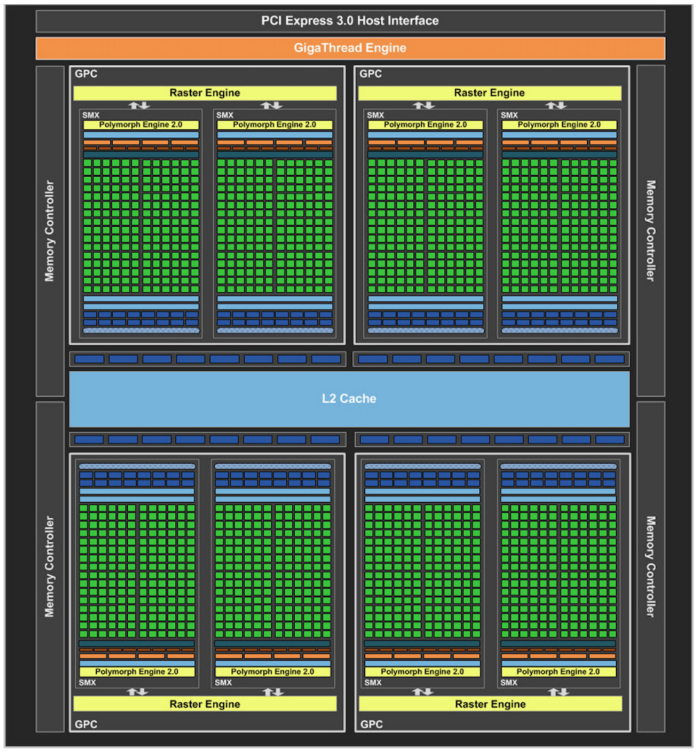

Figure 2.7: Kepler Architecture.

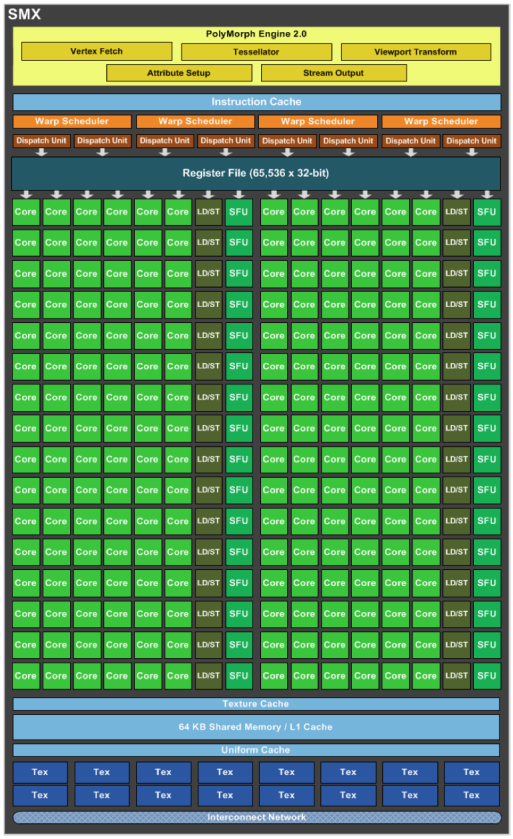

Figure 2.8: GTX680 SMX.

In Kepler, as shown in Figure 2.9, each SMX has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache or a 32KB / 32KB split between shared memory and L1 cache. In addition to the L1 cache, Kepler introduces a 48KB cache for data that is known to be read-only for the duration of the function and it also has a 1536KB of L2 cache memory. The L2 cache services all loads, stores, and texture requests and provides high speed data sharing across the GPU [11].

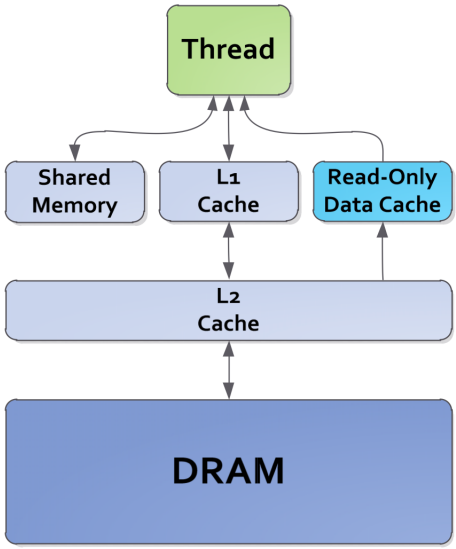

Figure 2.9: Kepler Memory Hierarchy.

Graphics processors have traditionally been designed for very specific specialized tasks. Most of their transistors perform calculations related to 3D computer graphics rendering. Typically GPUs perform memory-intensive work such as texture mapping and rendering polygons. The GPU also performs geometric calculations such as rotation and translation of vertices into different coordinate systems. The on-chip programmable shaders can manipulate vertices and textures.

NVIDIA® has developed a parallel computing architecture which is known as the Compute Unified Device Architecture (CUDA). This computing engine, which is the core of modern NVIDIA® GPUs, is accessible to software developers through extensions of industry standard programming languages. The development of CUDA has enabled developers to access the virtual instruction set and memory of the GPU. This has enabled the exploitation of the native parallel computational elements of the NVIDIA® GPU.

The Kepler architecture that has the Compute Capability 3.5 or higher supports dynamic parallelism, in which the GPU can launch new grids by itself. This feature allows algorithms that involve nested parallelism, recursive parallelism to be implemented on GPUs. This results in better GPU utilization than before. In Figure 2.10, the Kepler host to GPU workflow shows the Grid Management Unit, which allows it to manage the actively dispatching grids, pause dispatch, and hold pending and suspended grids [11].

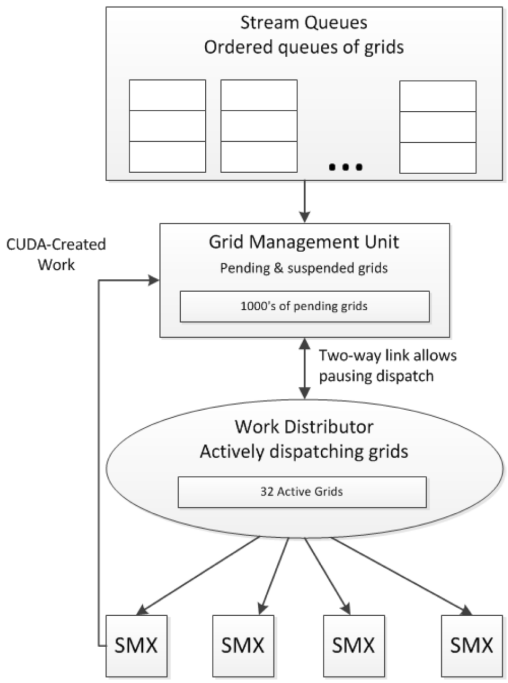

Figure 2.10: Kepler Work Flow.

The CUDA model is optimized for maximum compatibility. NVIDIA® utilizes a soft instruction set which enables the GPU designers to change the low level hardware and instruction set without having to address backwards compatibility. Similarly, NVIDIA® has also built in scalability to the CUDA model. These GPUs are scalable

in that the CUDA code that is written is not tied to a specific release of the NVIDIA® GPU. This can be contrasted to traditional CPUs where the hard instruction set is published. CPU software developers often optimize their programs for how many cores are available which can change as new CPUs are released.

A CUDA program is comprised of multiple execution phases. Depending on the phase, the execution involves the CPU, the GPU or both. Portions of the CUDA code are executed on the GPU while other portions are executed on the CPU. The NVIDIA® compiler known as *nvcc* translates the code for the GPU and CPU accordingly. This model is very easy to work with because the device code is written in ANSI C extended with keywords for labeling data-parallel functions, called *kernels*, and their associated data structures [13].

GPU implements a Single instruction multiple data (SIMD) model unlike the traditional CPU architectures where each thread may execute its own set of instructions. Multiple processing units are controlled by the same control signal from the control unit. Though each unit executes the same instruction, they have different data that corresponds to the CUDA threads.

The GPU hardware maps the data blocks to the SMX through time and space multiplexing. Since each SMX has limited hardware resources such as shared memory, registers, number of threads that can be tracked and scheduled (scheduling lots), careful selection of block sizes allow the SMX to accommodate more blocks simultaneously thus increasing the throughput.

2.4 Motivation

In theory, for n items, the quadtree gives a performance of $(n * log(n))$. Compared to a brute force method's performance of $n^{(2)}$ [3], a quadtree is extremely fast. The performance gain by using a quadtree is $(log(n))/n$. As shown in Figure 2.11, the empirical investigation of quadtree search and brute force search shows that the

quadtree has a 9x better performance for medium number of queries and 8x improvement for larger problems. The main purpose of quadtree lies in localizing the queries. The brute force search on a CPU checks sequentially if every single point of the input data satisfies the criteria whereas quadtrees help isolate the region of interest faster by ignoring the quadrants of the tree that lie outside the region of interest at every level, thus reducing the number of quadrants to be processed at the next level. Therefore the quadtree gives better performance for a dense more concentrated data than a dense equally distributed data set.

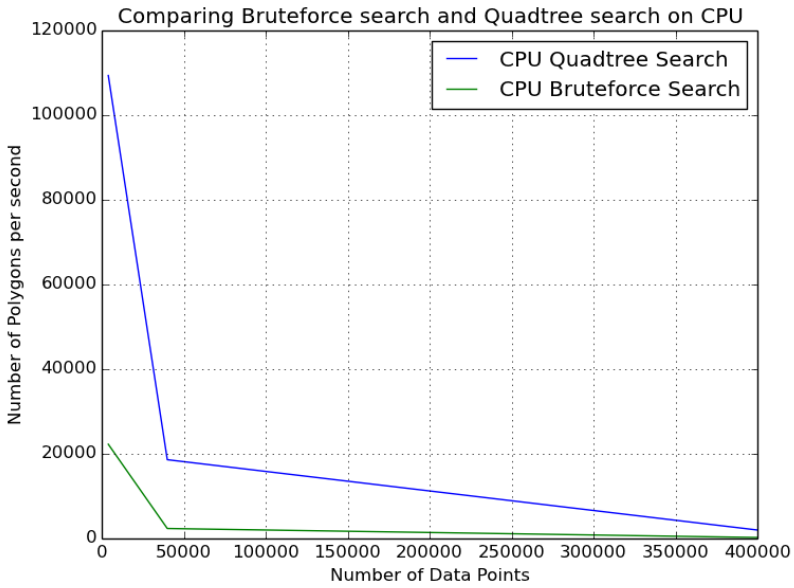

Figure 2.11: Comparing Bruteforce Search and Quadtree Search on CPU.

As shown in Figure 2.12, the GPU provides a better performance for brute force search compared to CPU for larger datasets. But this performance is still lower than the CPU performance using a quadtree. The same algorithm is used for both CPU and GPU brute force search to check if a point is within the Polygon. But the GPU

is capable of launching 65535 blocks with a maximum of 1024 threads, thus massively parallelizing the search. The Geforce GTX680 Kepler architecture used, has a new parallel geometry pipeline optimized for tessellation and displacement mapping. And also it has a better performance / watt [12].

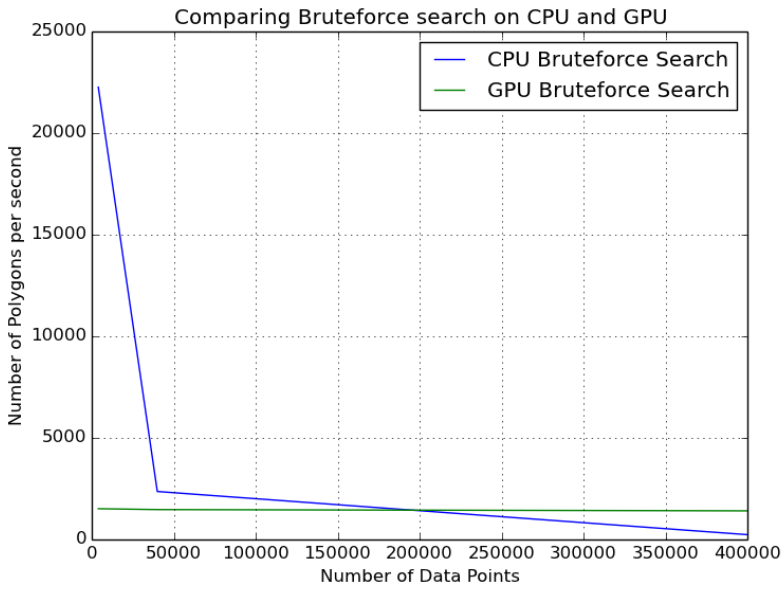

Figure 2.12: Comparing Bruteforce Search on CPU with GPU.

Figure 2.13 shows the quadtree performance for different sized polygons. The small polygons occupy 10 to 20 percent of the quadtree area, the medium sized polygons occupy 30 to 60 percent of the quadtree area and the large polygons occupy 70 to 90 percent of the quadtree area. The CPU implementation performs better on small polygons compared to larger polygons. The performance improvement in this case relates to how quickly the quadrants of the quadtree that contain the polygon can be isolated. The amount of computation done for a larger polygon that occupies all four quadrants is higher compared to a smaller polygon that lies inside only one

quadrant.

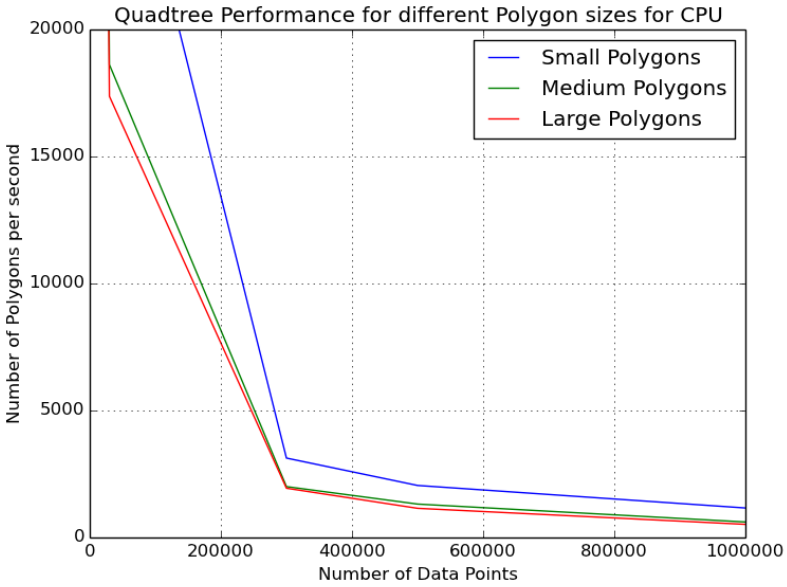

Figure 2.13: CPU Performance of Different Sized Polygons using Quadtree.

3. Approach

3.1 Exploitation of Quadtree on GPUs

3.1.1 CPU Implementation

The traversal starts from the root node of the quadtree and performs DFS by recursion. The traversal runs recursively till leaf node is reached. Depth First Search algorithm (DFS) traverses a graph in a depth wise motion and uses recursive function to get the next node to start a search when the bottom of the tree is reached. The traversal starts at the root node and moves down the tree till the bottom of the tree is reached along each branch before backtracking. While visiting each node, it checks for all three conditions to see if a node is completely overlapping, partially overlapping or not overlapping.

Table 3.1 illustrates the conditions to satisfy each scenario. N0, N1, N2 and N3 are the four corners of a node and the boundary of the polygon is marked by the four corners, P0, P1, P2, P3 of a rectangle. A checkmark indicates that the corner of the node is within the boundary of all four edges of a polygon. A x mark indicates that the corner of the node is outside the boundary of all four edges of a polygon. If all the corners of a node is within the four edges of a polygon, then the node is completely overlapping the polygon. If all the corners of a node is outside the four edges of a polygon, then the node is not overlapping the polygon. If at most three of the corners of a node is outside the four edges of a polygon, then the node is partially overlapping the polygon. All the conditions apart from the ones shown in Table 1 are partially overlapping.

Table 3.1: Criteria for Quadtree Node Classification.

Case	N0	N1	N2	N3	Condition
P0 P1 P2 P3	✓	✓	✓	✓	Completely overlapping
P0 P1 P2 P3	✗	✗	✗	✗	Not overlapping

Before classifying the node as not overlapping, the polygon is also checked to see if it lies inside the node. If the node is classified as completely overlapping node, then the boundary details of this node is stored and tree is not traversed further from this node. The boundary of the node represents the range of points the node contains and all these points within the node are considered as points within the polygon. If the node is classified as not overlapping node, then the tree is not traversed further from this node. If the node is classified as partially overlapping node then the tree is traversed further till the leaf node is reached. The points are then extracted from the leaf nodes of the quadtree.

In the case of partially overlapping node, every point needs to be checked to see if it lies within the boundary of the polygon. This check is optimized by classifying the kind of intersection between the node and the Polygon in such a way that for certain scenarios only either x or y coordinate of the points need to be verified.

If the intersection of the node is along the horizontal edge of the polygon as in Figure 3.1, then only the y coordinate of the data points inside the node needs to be checked as all the x-coordinate of all points is within the polygon boundary limits. If the intersection between the node and the polygon is along the vertical edge of the polygon as in Figure 3.2, then only the x coordinate of the points needs to be checked as all the y-coordinate of all points is within the polygon boundary limits. For all the other overlap conditions, x-y coordinate of every point is checked with the boundary limits of the polygon.

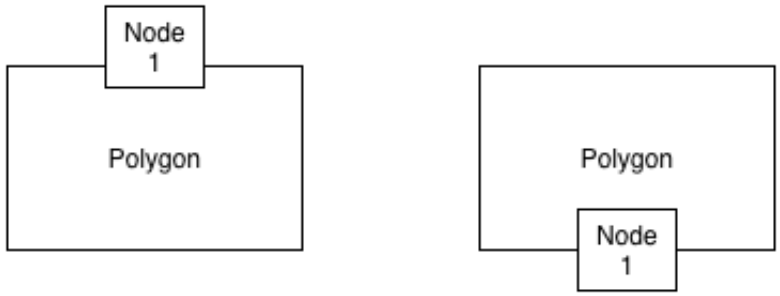

Figure 3.1: Overlap of Node on the Horizontal Edge of Polygon.

Figure 3.2: Overlap of Node on the Vertical Edge of Polygon.

In the case of GPU implementation, x-y coordinates of all points of the partial node are checked against the polygon edges without classifying them based on the type of node intersection as it will lead to more conditional statements which would increase thread divergence and reduce the performance.

3.1.2 GPU Implementation

Efficient implementation of quadtree traversal in GPU using CUDA is challenging due to the following reasons. 1. The quadtree is an irregular data structure whose memory accesses and work distribution are both irregular and data-dependent. And traversing irregular data structures results in thread divergence. 2. Traversal of a quadtree built using pointers results in lot of pointer-chasing memory operations.

Pointer-chasing leads to many un-coalesced memory accesses which affects performance [2]. 3. As done typically the quadtree is built using recursive depth first style in the CPU. Recursive DFS in GPU would be advantageous as it will reduce un-coalesced memory accesses considering the memory for all the structures is pre-allocated. Recursion can be implemented only on the latest GPUs with CUDA 6.5 version that supports dynamic parallelism, but recursion could lead to low performance due to function call overhead. Recursion on a GPU, requires explicit stack space per thread and deep recursion will cause overflow of the available stack space and in this case CUDA may reduce the maximum physical threads per launch. 4. Transferring a pointer based quadtree to a GPU proves to be a challenge. Though this task can be implemented with "cudamallocamanaged" function, debugging becomes harder.

To take advantage of the parallel nature of the GPUs, BFS is used instead of DFS. The quadtree is queried to find the points inside a polygon by first finding the quadtree nodes that overlap the polygon. Once the nodes that overlap the polygon are determined, the points inside the polygon can be found from these nodes. Given a set of nodes, arranged hierarchically, the system needs to find the minimum set which solves the query correctly. This can be done by assigning one thread to a polygon but there is not enough stack space. Since our GPU implementation method requires an explicit array in the shared memory for each polygon, assigning one thread to a polygon will also pose a limit on the number of polygons processed simultaneously by a block due to the limitation in the shared memory size. The stack space per thread needs to be reduced and optimization for thread divergence and memory access should be done. To minimize thread divergence and reduce stack space per thread and also achieve maximum parallelization, a polygon is assigned to a warp instead of a thread or a block of threads. To optimize for memory access, memory preallocation for the input points, query polygons, point buffer (list that holds points within a leaf node

of quadtree) is done on the CPU.

CUDA applications exploit massive data parallelism and it is capable of processing massive amount of data within a short period of time. But data accesses to GDDR DRAM global memory limits this performance due to limitations in global memory bandwidth. Therefore algorithms and techniques need to be employed in order to reduce the demand on the DRAM bandwidth and make efficient use of the data accessed from the global memory [9].

The quadtree is built on the CPU from heap objects and each heap object has several fields. For example, each quadtree node object contains 4 child pointers and data fields which stores the attributes of the node. Other heap objects include data points and point buffers. Though the CPU implementation benefits from the dynamic memory allocation, the use of the C library malloc is not preferred for the CPU-GPU implementation because this dynamic memory-management function provides allocation and de-allocation of arbitrary segments of memory. In order to improve performance on the GPU, the access to memory segments need to be coalesced. The memory preallocation places the consecutive points and point buffers sequentially in memory and therefore results in memory coalesced access as the points close to one other are more likely to be within the same polygon boundary. But preallocating memory for the quadtree nodes is not expected to improve performance as the nodes are stored in a depth first fashion in the CPU but a BFS traversal is done on the GPU.

The DRAM cores are slower as the market demands a high capacity rather than a lower latency DRAM. These cores operate at much lower speed compared to the interface speed. GDDR4 operates at 1/8th of the interface speed and this difference will increase with every generation of GDDR as the storage capacity increases [9]. Since DRAM cores are slower than the interface, the system needs to increase the buffer width to feed into faster interface. The modern DRAM banks are designed to be

accessed in burst mode. If the accesses are not to sequential locations, the transferred burst bytes are discarded. In order to take full advantage of the DRAM system in CUDA environment, the system needs to make sure that the processor makes use of all the data bursted out of the DRAM chip. Thus from a memory throughput standpoint to get peak memory performance, computation must be structured around sequential memory accesses [9]. Random memory access leads to a huge performance penalty due to random memory reads.

3.1.2.1 Drawbacks on implementing DFS on GPU

One option to traverse down the tree for every polygon is to use DFS by assigning a warp or a thread to each polygon. But the DFS implementation in the GPU would cause a overhead due to the data movement as the thread moves up and down the tree. In this case the interior nodes of the tree are repeatedly visited as a thread has to return to higher level of the tree to traverse down to other children [4]. This results in additional work for each thread.

3.1.2.2 BFS Implementation on GPU

Stack based breadth first tree-traversal allows parallelization. To take advantage of the parallel nature of the GPU, the threads are launched at level 3 of the quadtree instead of the root node. Starting at the root node will result in only one thread being active for level 1 traversal of quadtree. Starting at a deeper level prevents this and results in more parallelism. Initially the index of the node from level 3 of the quadtree is stored in an array in the shared memory. This index is used to fetch the quadtree node from the global memory.

Here the tree is traversed down to find the boundaries of a polygon and thus extracting the quadtree nodes within that boundary or partially overlapping the boundary. For completely overlapping nodes, the boundary of the node gives the range of points within the polygon. This method is more efficient than traversing down the quadtree to find the individual points within the polygon in terms of computation and also memory. And for partially overlapping nodes, each point needs to

be checked against the boundary of the polygon. With this information, the range of points within the polygon region can be found and this range is stored instead of individual points to save memory on the GPU. If there are a very large number of input points and query polygons, then storing individual points for each polygon will result in overflow of the memory.

The Polygons are associated with a CUDA warp, where the number of threads in a warp depends on CUDA configuration. By launching thousands of GPU blocks in parallel, 'N' number of polygons can be queried, where N = (size of each block / 32)*(total number of blocks). Each warp executes the same algorithm but on a different polygon. Assigning a warp to a polygon will result in less thread divergence. Each thread in a warp reads x, y co-ordinate of all four corners of a polygon which is required for the tree traversal.

The GPU kernel replaces the recursive call with stack push operations that save indices of each child of a node traversed, that satisfies the query. Traversal is facilitated by a loop that simultaneously assigns the indices of the nodes in the stack to threads in warp until the stack is empty, indicating there are no more nodes to visit at that level for a polygon. To save stack space, node indices are stored in the stack instead of the node. At the beginning of each iteration of the loop the nodes are popped from the stack. After popping the node from the stack, the function which checks for the overlapping conditions is executed on that node, possibly pushing more nodes onto the stack or returning when it reaches a node that fails the criteria, an empty node or a leaf node.

If the number of nodes in the stack exceeds the number of threads assigned to a polygon, then loop over with increments of the number of threads in a warp is done. Within each loop, each node is assigned to one thread. Each thread reads the quadtree node's properties and tests whether or not to traverse the tree further down from this node. If the criteria to traverse the tree down from this node is satisfied,

then the indices of the node's all four children is added to the stack, provided the child is not an empty node. One stack space is assigned to each polygon. Traversal at each level of quadtree is synchronized, so that the same stack can be re-used for a polygon. The tree-traverse loop is repeated until all the levels of the quadtree is visited. The number of GPU threads used per level is equal to the number of nodes that was pushed on to the stack from the previous iteration.

The warp starts the traversal at level 3 of the quadtree. The indices of children of the nodes that satisfy the query are placed in an array. The condition checks whether a node is overlapping the polygon. Once the indices of children of the nodes that satisfy the query are placed in an array and the other nodes, which do not satisfy the condition are ignored, the next level is evaluated. Once the leaf node is reached, the nodes are classified as completely and partially overlapping nodes. And from these nodes all the points that are inside the Polygon are taken. The boundaries of the completely overlapping nodes which give the range of points within the nodes are stored.

In the case of partially overlapping nodes, each point within the node is checked against the boundary of the polygon and the range of points that lie within the polygon is stored. At the last level of the tree, the warp still holds the polygon and the threads within a warp is assigned to a leaf node that is either partially or completely overlapping the polygon. Each of these threads compute the points within each node that are inside the polygon.

This approach will minimize the use of shared memory space, as only the indices of nodes are stored starting from level 3 in shared memory instead of the node itself that satisfy the required criteria. The indices of the nodes are used to fetch the corresponding node from the global memory. The order in which the nodes are stored in the stack does not matter, as every level in the quadtree is processed independent of the previous and every node is processed independent of other nodes in the stack.

The indices of the nodes are saved in the stack array in the shared memory and the the stack is cleared after computing every level. Shared memory blocks are used in order to gain significant speedups of approximately 150 to 200 times [8]. If a node index returns -1, then the node is empty and it is not fetched from the global memory. The number of nodes that needs to be visited at the next level is also stored in a shared memory variable for every polygon and it is incremented atomically. The iterative loop launches threads based on the count on this variable. The loop executes till the last level of quadtree is reached.

To implement this algorithm, the maximum number of levels in a quadtree should be 4 because of the limitations in the size and the number of stacks in the shared memory. As the number of levels in a quadtree increases, the maximum stack space required per polygon would also increase and this will limit the number of polygons processed.

The size of shared memory per block on the GTX680 is 49152 bytes [12]. As there are 32 warps in a block by default, each block is assigned 32 polygons. Each polygon requires an array in the shared memory in order to store the node indices from each level that needs to be further traversed. The maximum array size required for a 4 level quadtree is 64 as the maximum number of nodes at the leaf is 64. This allocation occupies a shared memory space of 32 * 64 *4 = 8192 bytes, which is well within the limit. The array size for each polygon needs to be increased to 256 for a 5 level quadtree. This allocation results in a shared memory usage of 256 * 32 * 4 = 66560 bytes, which exceeds the available shared memory size. Therefore unless a larger size of shared memory becomes available in the future, the quadtree cannot not be subdivided beyond level 4 in this case.

In Figure 3.3, the check mark indicates that the node satisfies the query and the tree will be further traversed from this node. The cross mark indicates either that the node does not satisfy the query or the node is empty and the threads terminate

at this point without proceeding further.

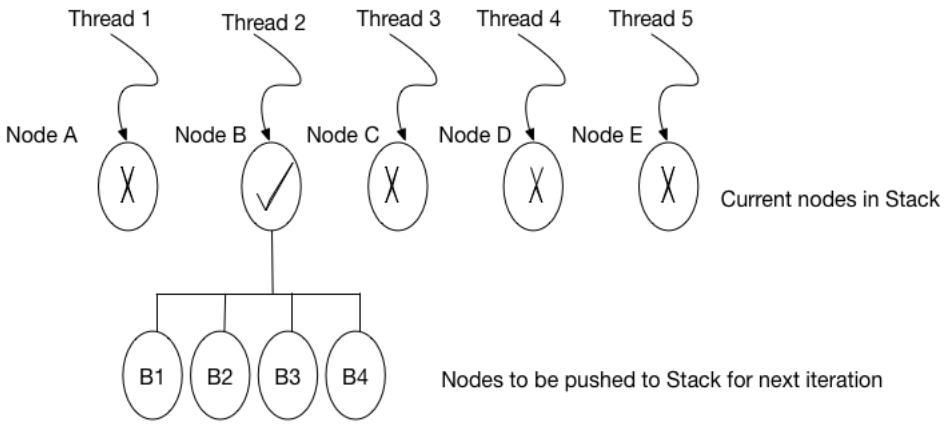

Figure 3.3: BFS GPU Implementation.

The traversal starts at level 3 of the quadtree. The children of the nodes that satisfy the overlap conditions are pushed in to the array. The array is initialized after every iteration. The cross mark indicates that the node is either not satisfying the overlap conditions or it is empty. The nodes are stored in the array in any order. The process repeats till bottom of the tree is reached. The number of stack array in the shared memory per block depends on the number of warps in a block. At the most, the iterative loop processes 32 nodes at a time by default (equal to the number of threads in a warp). The conditional statements are changed to avoid function return statements to prevent the traversal loop from prematurely exiting and preventing the remainder of the loop body from executing.

A polygon is checked against the node of the quadtree for 3 scenarios such as, completely overlapping, partially overlapping and not overlapping. All nodes are checked for "not overlapping" conditions till leaf node is reached. If a node satisfies the condition then the tree is not traversed from this node and if the node does not satisfy the condition, then its children are placed in the stack and the traversal

continues.

The number of checks for a partially overlapping node is more and use of "if" statements decreases the performance due to control divergence. If threads within a warp take different paths, then 2 passes on the different path of the code is required to execute all threads. Since different execution paths are serialized, it leads to performance penalty.

Therefore in order to reduce conditional statements (if statements), the condition for completely overlapping node is first computed on the leaf nodes and the nodes that satisfies the condition is classified as completely overlapping nodes and nodes that does not satisfy the condition is classified as partially overlapping nodes. The Figure 3.4, shows the high level overview of the algorithm to find the points in polygon using quadtree on a GPU.

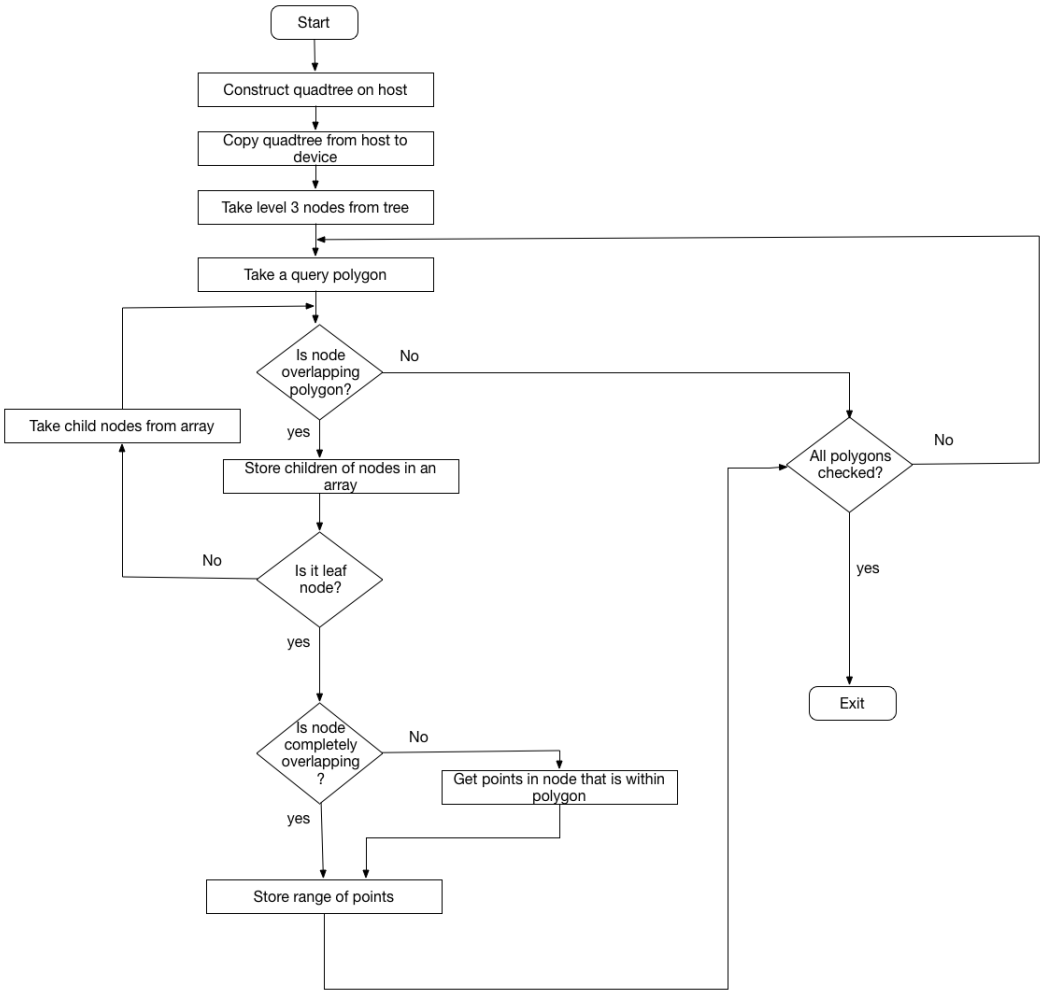

Figure 3.4: Process Flow.

3.1.3 BFS Implementation on GPU - Different Scenarios of Query Polygon

The Figure 3.5, Figure 3.6, Figure 3.7 and Figure 3.8 show the data points (left) and its corresponding tree representation (right). These data points will be analyzed for different polygon overlap scenarios in the next sections. The numbers inside the circle represent the index of the node at that level. Figure 3.5 shows the root node that contains all input data points. The root node is subdivided into 4 equal regions

to form 4 nodes at level 2. Figure 3.6 shows the four children of the root node. Root node has all four children as the data points are present in all four directions (NW, NE, SW, SE) of the root node. The nodes at level 2 are again subdivided equally to form the child nodes. Figure 3.7, shows the nodes at level 3 of the quadtree. Child node 3 of parent node 1 from level 2, child node 1 and child node 4 of parent node 3 from level 2 and child node 3 of parent node 4 from level 2 are empty as the parent nodes did not have any points in those directions. Figure 3.8 shows the child nodes at level 4. Many of the nodes at this level are empty as they do not contain any points.

Figure 3.5: Level 1 Quadtree.

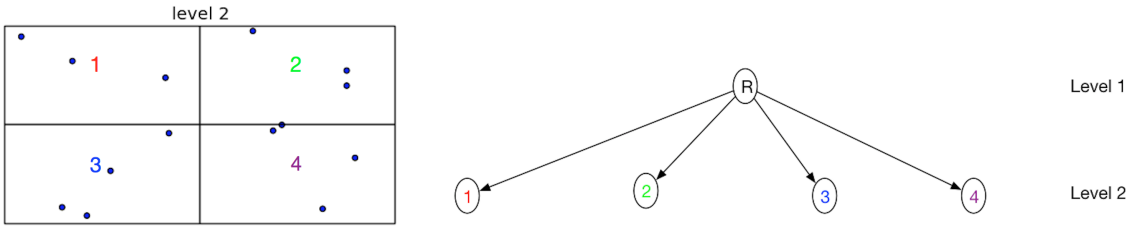

Figure 3.6: Level 2 Quadtree.

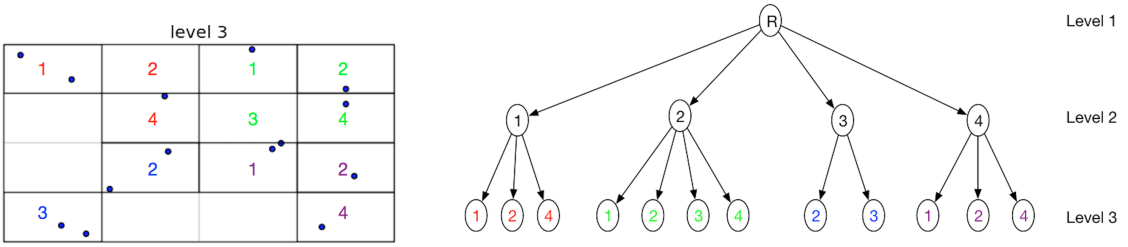

Figure 3.7: Level 3 Quadtree.

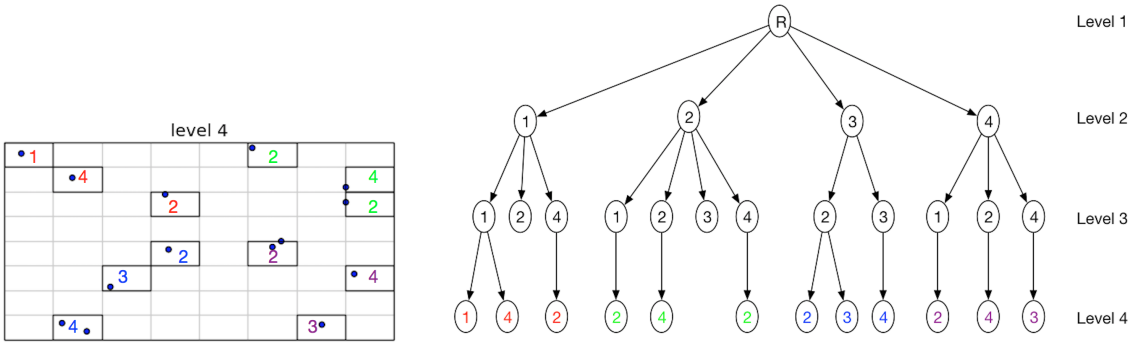

Figure 3.8: Level 4 Quadtree.

3.1.3.1 Query Polygon inside a Leaf Node

Figure 3.9 shows the first scenario that demonstrates the best case scenario where the polygon lies within one quadrant at level 4 of the quadtree and the algorithm provides best performance.

Figure 3.9: Sample Datapoints (left) and the Quadtree (right) at Level 1.

Figure 3.10 shows that at level 2, the algorithm isolates one quadrant (first child of the root node) and ignores all other quadrants, thus bringing down the number of quadrants to be processed to 1 from 4 (4 is the total number of nodes at this level).

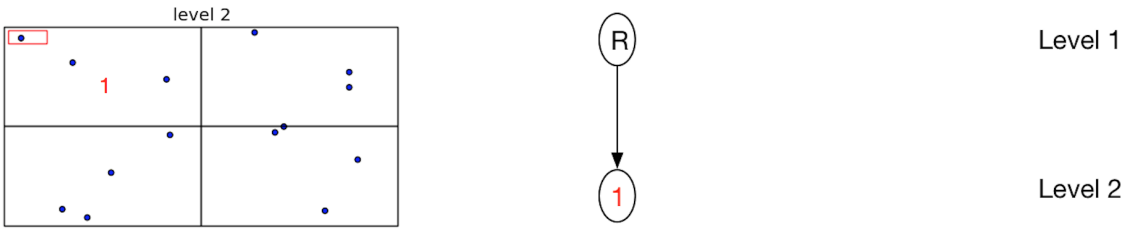

Figure 3.10: Sample Datapoints (left) and the Quadtree (right) at Level 2.

Figure 3.11 shows that at level 3, the algorithm again isolates one quadrant which is the first child of the quadrant from level 1. This step reduces the number of quadrants to be processed to 1 from 3 (3 is the total number of children of the node output from level 1).

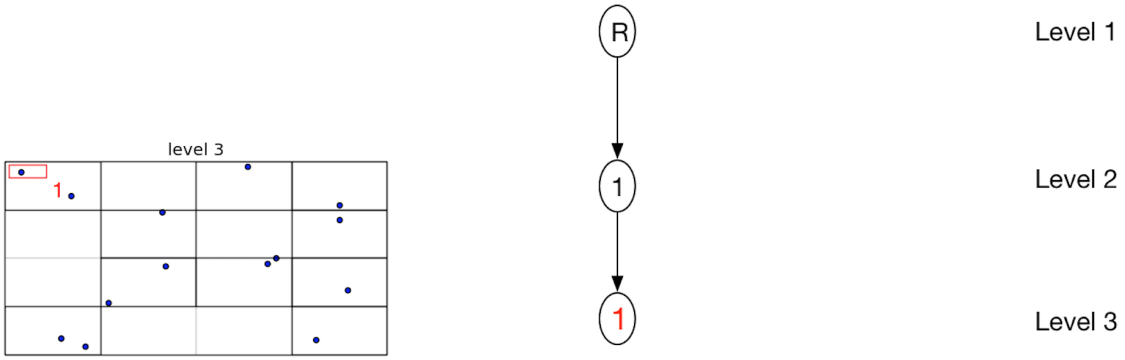

Figure 3.11: Sample Datapoints (left) and the Quadtree (right) at Level 3.

In Figure 3.12 at level 4, the algorithm isolates one quadrant which is the first child of the quadrant from level 3. This step reduces the number of quadrants to be processed to 1 from 2 (2 is the total number of children of the node output from level 3). Finally only one quadrant at level 4 needs to be processed in order to get the points inside the polygon in this case.

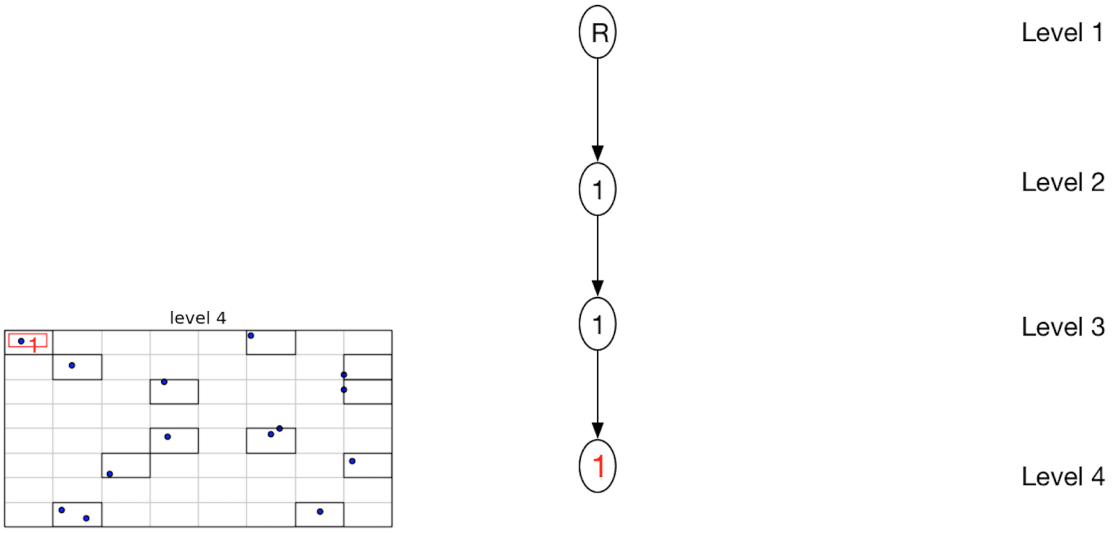

Figure 3.12: Sample Datapoints (left) and the Quadtree (right) at Level 4.

3.1.3.2 Polygon overlapping 2 nodes

Figure 3.13 shows the second scenario that demonstrates a condition where the query polygon overlaps 2 nodes at level 2 and the algorithm provides a medium performance.

Figure 3.13: Sample Datapoints (left) and the Quadtree (right) at Level 1.

Figure 3.14 shows that at level 2, the algorithm isolates two quadrants (second

and fourth child of the root node) and ignores the other two quadrants, thus bringing down the number of quadrants to be processed to 2.

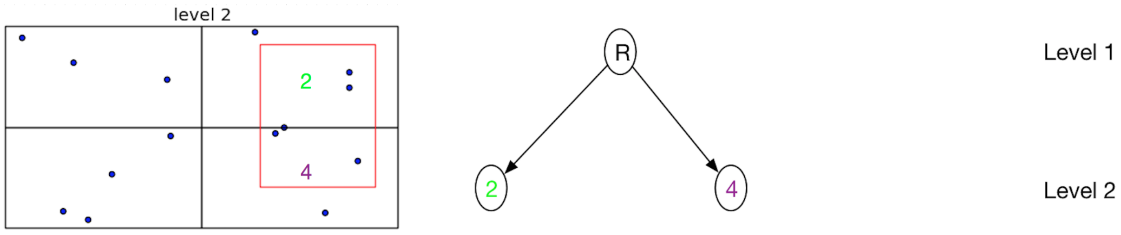

Figure 3.14: Sample Datapoints (left) and the Quadtree (right) at Level 2.

Figure 3.15 shows that level 3, the algorithm isolates 7 quadrants which are the all four children of the second child of root node and first, second and fourth child of the fourth child of root node. This step reduces the number of nodes to be processed by one as one of the child of the nodes output from the previous level is empty.

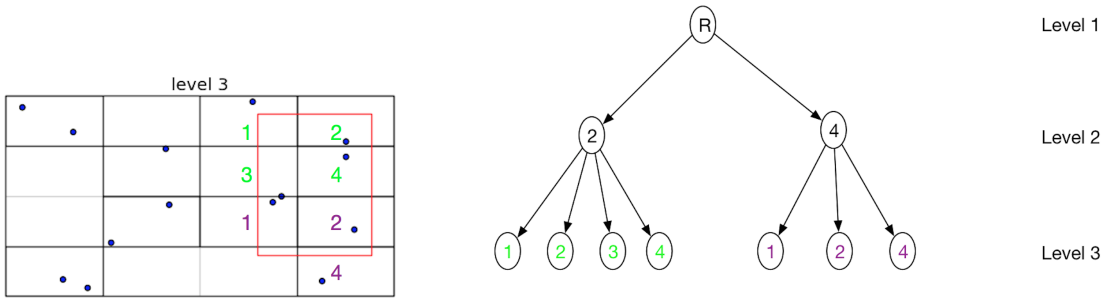

Figure 3.15: Sample Datapoints (left) and the Quadtree (right) at Level 3.

Figure 3.16 shows that at level 4, the algorithm isolates 5 quadrants from the children of 7 quadrants from previous level. By traversing down to level 4, the number of data points that need to be evaluated are reduced by ignoring 3rd child of the 4th

child from level 3. If the quadrant that is ignored at level 4 has a larger number of data points, then using a 3-level quadtree would have resulted in a huge performance penalty. Finally only 5 quadrants at level 4 needs to be processed in order to get the points inside the polygon in this case.

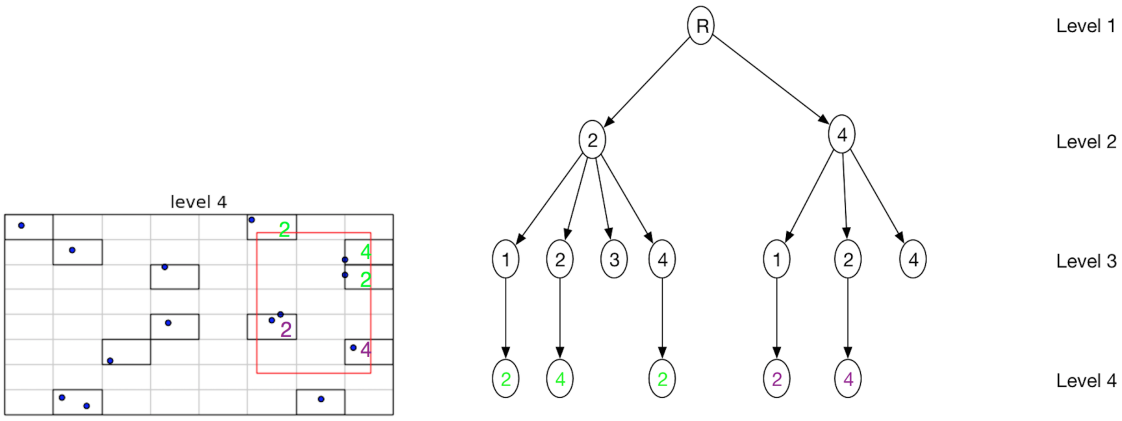

Figure 3.16: Sample Datapoints (left) and the Quadtree (right) at Level 4.

3.1.3.3 Query Polygon Overlapping all Leaf Nodes

Figure 3.17 shows the last scenario that demonstrates a condition where the polygon contains maximum number of quadrants at level 4 of the quadtree and the algorithm provides least performance compared to all other scenarios.

Figure 3.17: Sample Datapoints (left) and the Quadtree (right) at Level 1.

In Figure 3.18 at level 2, the algorithm overlaps all 4 quadrants at this level. Therefore the thread has to descend the tree from all four nodes.

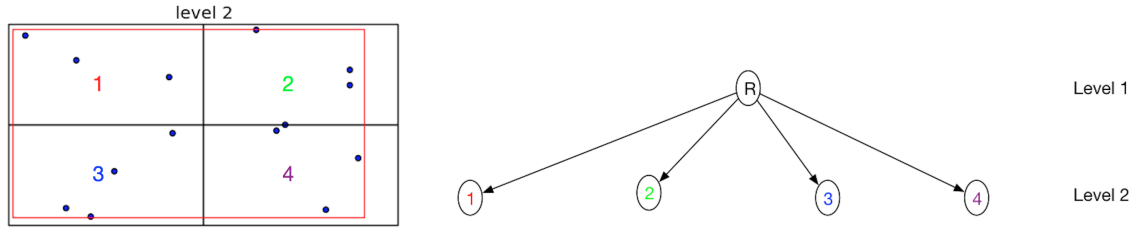

Figure 3.18: Sample Datapoints (left) and the Quadtree (right) at Level 2.

In Figure 3.19 at level 3, the algorithm isolates all 12 quadrants at this level. The maximum possible number of nodes at this level is 16 but four of those nodes are empty nodes and these four nodes are ignored even though these nodes lie within the polygon.

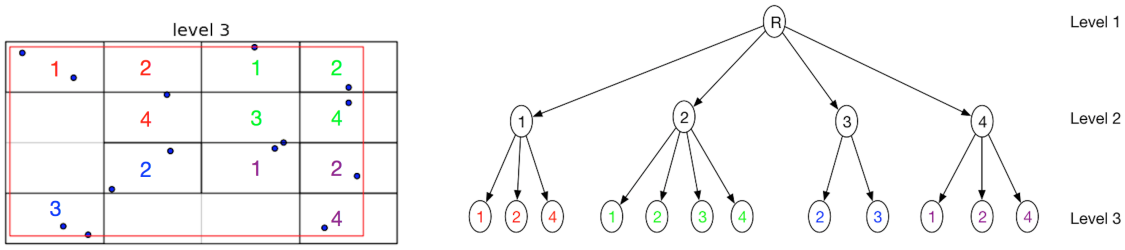

Figure 3.19: Sample Datapoints (left) and the Quadtree (right) at Level 3.

Figure 3.20 shows that at level 4, the algorithm isolates all the 12 quadrants at this level. The maximum possible number of nodes at this level is 64 but 52 of those nodes are empty nodes and these 52 nodes are ignored even though these nodes lie within the polygon. Finally only 12 quadrants at level 4 need to be processed in order to get the points inside the polygon in this case. If there are a larger number of data points distributed equally across the entire 2 D space and if there are no empty nodes, then all the 64 nodes need to be processed.

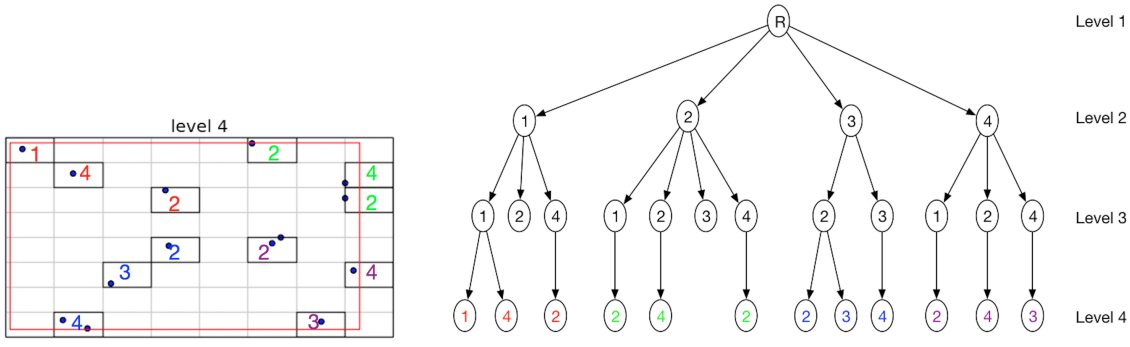

Figure 3.20: Sample Datapoints (left) and the Quadtree (right) at Level 4.

Figure 3.21, shows the stack based iterative BFS traversal on GPU, in reference to Figure 3.19 and Figure 3.20.

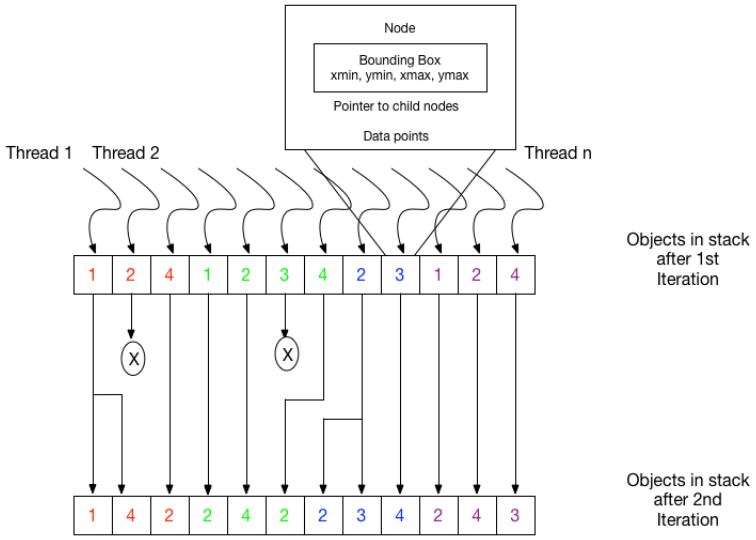

Figure 3.21: Stack Based Iterative BFS Traversal.

3.1.3.4 Query Polygon Containing no Points

Figure 3.22 shows one case where the polygon overlaps a region where there are no points.

Figure 3.22: Sample Datapoints (left) and the Quadtree (right) at Level 1.

Figure 3.23 shows two distinct cases. At level 2, the algorithm isolates first and third child of the root node. At level 3, the polygon does not overlap with any of the children of the nodes from level 2. The nodes that it overlaps are empty nodes and therefore the tree is not traversed any further.

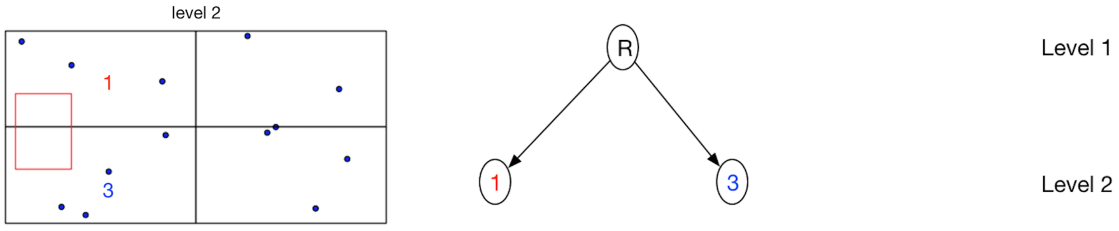

Figure 3.23: Sample Datapoints (left) and Quadtree (right) at Level 2.

4. Experimental Results

The experimental results show that in many cases the GPU out-performs the CPU as expected.

4.1 Results

The performance is compared in terms of the number of polygons that can be queried per second. The number of polygons queried per second is measured for a wide range of input points. As the size of data points increase, the performance of CPU diminishes at a faster rate than the GPU. In the case of GPUs, as the size of input increases, performance is affected as the amount of work done per thread increases and also the overhead of memory transfer of large amount of data between CPU and GPU increases. But once the quadtree is transferred to the GPU, any number of polygons can be queried using iterative BFS traversal method described above.

Figure 4.1, Figure 4.2 and Figure 4.3 show that the CPU-GPU approach gives a performance improvement by a factor of 3.2 for small datasets and a performance improvement by a factor of 449 for very large datasets in the case of small polygons, a performance improvement by a factor of 4 for small datasets and a performance improvement by a factor of 594 for very large datasets in the case of medium sized polygons and a performance improvement by a factor of 3.6 for small datasets and a performance improvement by a factor of 591 for very large datasets in the case of large sized polygons respectively.

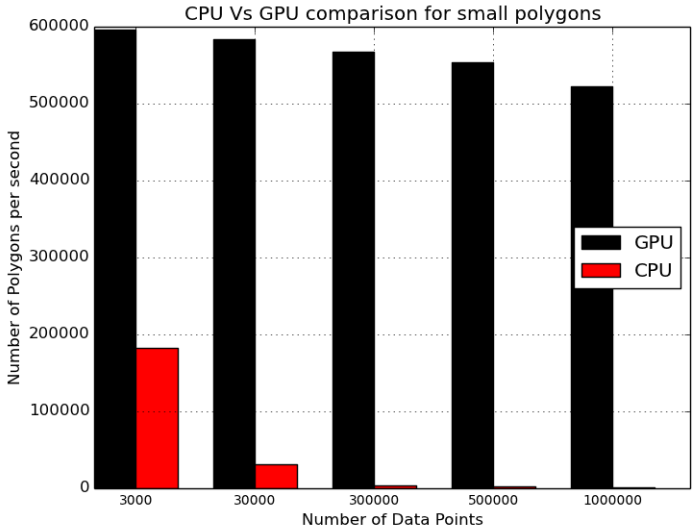

Figure 4.1: Performance Comparison between CPU and GPU for Small Polygons.

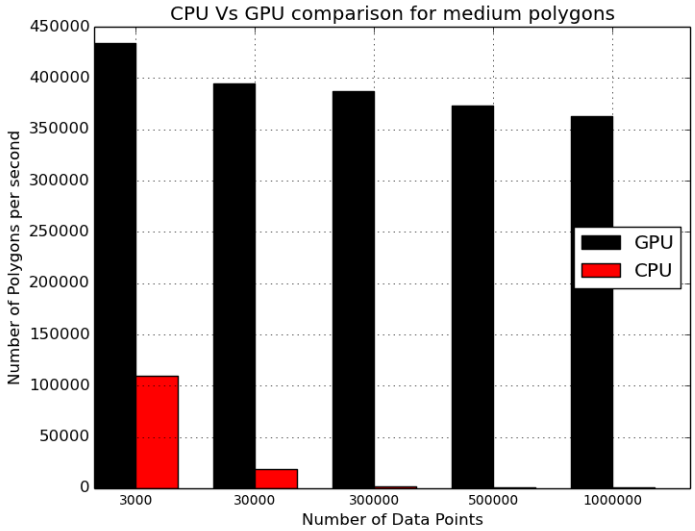

Figure 4.2: Performance Comparison between CPU and GPU for Medium Polygons.

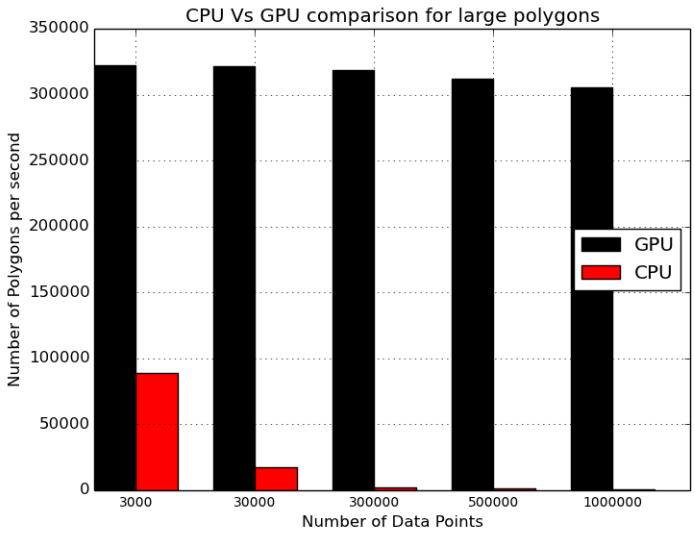

Figure 4.3: Performance Comparison between CPU and GPU for Large Polygons.

The CPU performance diminishes at a faster rate than the GPU mainly because of the quadtree structure and the memory hierarchy. Since the tree is a pointer based structure, there is a cache miss every time a node is accessed. At the leaf node, the points are stored in a linked list, where each node in the list contains a pre-determined number of points and there is a cache miss every time a new buffer in the list is accessed. Therefore cache miss penalties are higher for larger datasets.

In the case of GPUs, the point buffers are sequentially stored in memory through memory preallocation. Though there is a penalty when a node is accessed, cache miss penalties are not encountered while reading data points from a node. And also, in the GPU implementation, the traversal is accelerated by starting the BFS at level 3 of the quadtree.

The Figure 4.4 shows the execution time of different polygon sizes on GPU. Similar to CPU, the GPU performs better on smaller polygons compared to the larger

polygons. In this case, the smaller polygons occupy 10 to 20 percent of the region and the larger polygons occupy 70 to 90 percent of the region. Once the leaf nodes are reached, the number of nodes that need to be taken into account to compute points are very less but for a large polygon which could contain a maximum of 64 nodes for a level 4 quadtree, points within all these 64 nodes need to be taken into account.

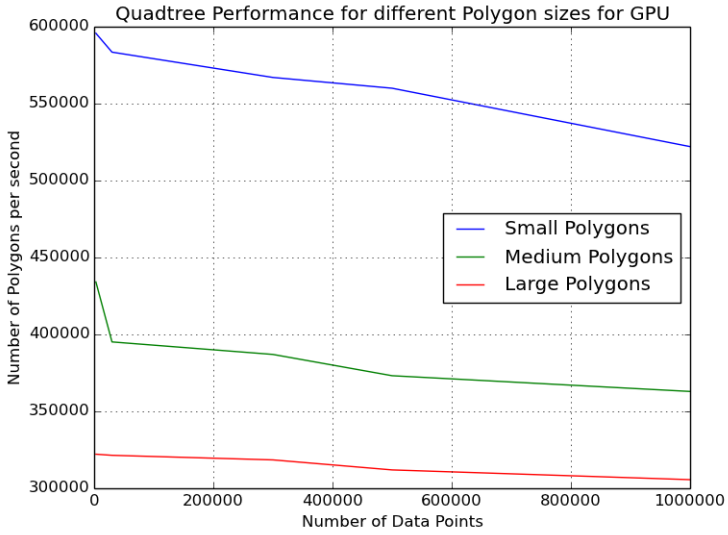

Figure 4.4: Performance Comparison for Different Sized Polygons using GPU.

The performance is determined by the number of partial nodes a polygon contains because for partial nodes, each point inside a node needs to be checked against a polygon boundary whereas in the case of completely overlapping nodes only the range of points a node contains is stored. The Figure 4.5 shows the execution time of individual medium sized polygons with a data input size of 3000. The polygons are ranked based on the number of partial nodes it contains and its execution time is measured. As the number of partial nodes increase, the execution time per polygon increases. The number of polygons calculated per second is the average of the output

of all these individual polygons.

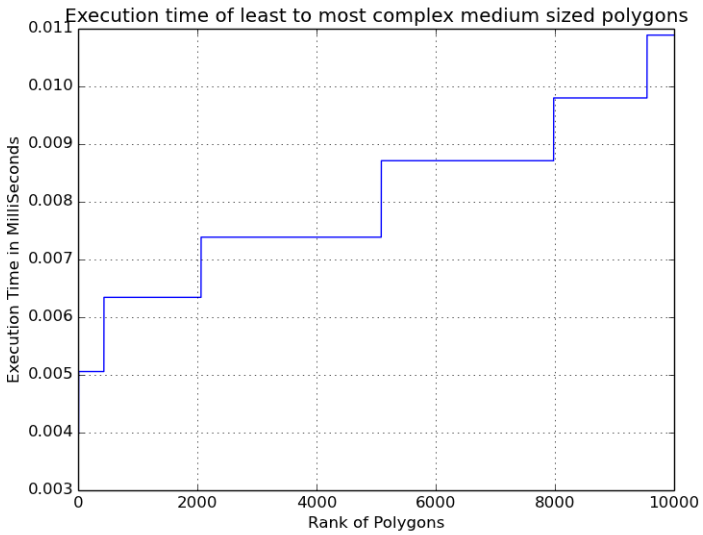

Figure 4.5: Execution Time of Least to Most Complex Medium Sized Polygons.

Figure 4.6 shows the performance comparison between CPU and GPU measured for different sizes of polygons. The significant speed up on GPU is due to the memory coalesced access and due to accelerating the traversal by starting the BFS from level 3. In this case, the algorithm traverses one level down the tree to quickly find the set of nodes that satisfy the query,

The maximum performance gain is achieved for the medium polygons which are more computationally intensive than small polygons. Very less number of threads are active for small polygons and this does not take advantage of the throughput oriented nature of the GPU. The work on medium sized polygons is optimum for the GPU as the performance decreases a little for larger polygons. The performance is mainly determined by the number of partial nodes a polygon contains. In the case of CPU, all the partial nodes in a polygon are processed sequentially but in the case of GPU,

32 partial nodes within a polygon are computed in parallel with 48 (Total number of SMX x (CUDA cores per SMX / number of polygons per warp)) [12] polygons being computed simultaneously.

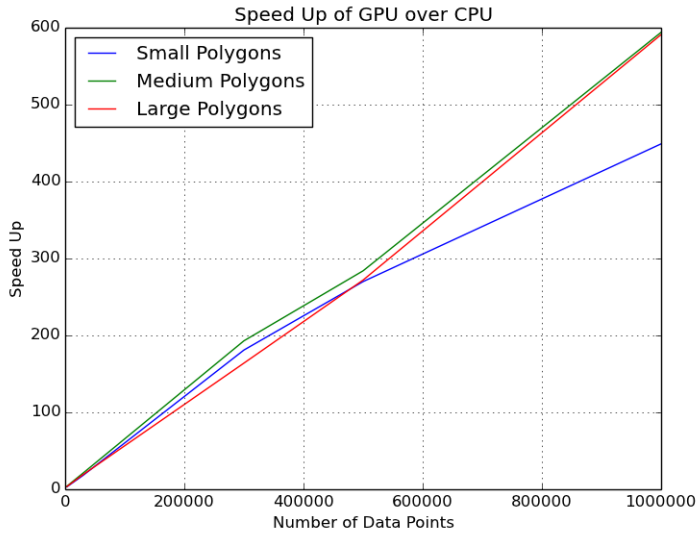

Figure 4.6: Speed Up of GPU over CPU.

5. Conclusion

The range search which is a fundamental operation in GIS and spatial databases always performs better when a spatial data structure is used and its performance is further improved when the search is implemented on a GPU. This paper has proved that an irregular pointer based quadtree need not be linearized in order to achieve a significant performance gain on GPU. And the CPU-GPU approach provides a speed up of 3x to 500x than the pure CPU approach. In a real world scenario, the range search problem is carried out on irregular polygons. For future work, the work presented in this paper can be extended to implement PIP search on irregular polygons. The work on the GPU can be extended to compute larger data sets as the size of RAM per GPU increases. With increase in shared memory size, quadtrees with deeper layers can be traversed. There would be a significant improvement in the performance with increase in shared memory size and increase in global memory bandwidth.

REFERENCES

[1] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the gpu processor. *J. Comput. Phys.*, 231(7):2825–2839, April 2012.

[2] Martin Burtscher and Keshav Pingali. An efficient cuda 6 implementation of the tree-based barnes hut n-body algorithm.

[3] David Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Exp. Algorithmics*, 5, December 2000.

[4] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 10:1–10:12, New York, NY, USA, 2013. ACM.

[5] Georgi Gou, C.; Gaydadjiev. Addressing gpu on-chip shared memory bank conflicts using elastic pipeline. *International Journal of Parallel Programming*, 41(3), 2013.

[6] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, page 14, May 2012.

[7] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[8] Maria Kelly and Alexander Breslow. Quadtree construction on the gpu: A hybrid cpu-gpu approach. https://www.sccs.swarthmore.edu/users/10/mkelly1/quadtrees.pdf.

[9] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[10] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3):235–246, June 2010.

[11] NVIDIA. White paper keplertm gk110. https://www.nvidia.com/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf, accessed, 2012.

[12] NVIDIA. White paper nvidia geforce gtx 680., 2012.

[13] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[14] Clifford A. Shaffer and Patrick R. Brown. A paging scheme for pointer-based quadtree. https://people.cs.vt.edu/shaffer/papers/paging.pdf.

[15] Bo Wu, Zhijia Zhao, Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[16] Jianting Zhang and Simin You. High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives. *International Journal of Geographical Information Science*, 27(11):2207–2226, 2013.

```
/** ************************************************************************
 *  File name : bruteForce.cu
 *  Create random points and polygons.
 *  Perform a brute force search to find points in polygon on GPU.
 ** ************************************************************************/
/**<************************# Includes **********************/
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/time.h>
#include <stdbool.h>
#include "hw4_4A_timing.h"
#include<cstdlib>
#include <cuda.h>

/**<************************# Defines **************************/
#define __host__
#define __shared__
#define BUILD_FULL 1
#define BUILD_ADAPTIVE 2
#define MODE_RANDOM 1
#define MODE_FILE 2
#define TRUE 1
#define FALSE 0
#ifndef RANGE
#define RANGE 1024
```

```
#endif
#ifndef SIZE
#define SIZE 20000
#endif

/**<************************ Global variables **************/
int pointMode = MODE_RANDOM;
char *inputPointFileName;
char *outputTreeFileName;
int rangeSize = RANGE;
int bucketSize = 32;
int numPoints = SIZE;
int numPolygon =50;
int pointRangeX = RANGE;
int pointRangeY = RANGE;
typedef int POLYID;

/**<************************ Structure definition ************/
// Input data point
typedef struct POINT {
   int x;
   int y;
   int index;
} POINT;

// Point coordinates.
typedef struct NODEPP{
   int xminPP;
   int yminPP;
```

```
}NODEPP;

// Query polygon
typedef struct Polygon {
   // coordinates of corners of a polygon and its width and height.
   int xmin;
   int ymin;
   int width;
   int height;
   int xmax;
   int ymax;
   int count;
   int index;
   NODEPP nodePoint[20000];
} Polygon;

/**<****************** Performance functions ***************/
typedef struct perf_struct {
   double sample;
   struct timeval tv;
} perf;

perf timeRecord;

void init_perf(perf *t) {
   t->sample = 0;
}

// Start time measurement.
```

```
void start_perf_measurement(perf *t) {
   gettimeofday(&(t->tv), NULL);
}

// Stop time measurement.
void stop_perf_measurement(perf *t) {
   struct timeval ctv;
   gettimeofday(&ctv, NULL);
   struct timeval diff;
   timersub(&ctv, &(t->tv), &diff);
   double time = diff.tv_sec + diff.tv_usec / 1000000.0;
   t->sample = time;
}

// Print time difference.
void print_perf_measurement(perf *t) {
   double total_time = 0.0;
   double stdev = 0.0;
   printf("%f", t->sample);
}

/**<********************** Random number generator *********************/
/**
 * Generate random numbers.
 * 1- Generate random data points.
 * 2- Generate random numbers for polygon corners.
 ** ********************************************************************/

// Generate a random number within a range.
```

```
int randn(int range) {
   int a;
   a = rand() % range;
   return a;
}

// Create random query Polygons
Polygon *createRandomPolygon(unsigned int nPolygon, unsigned int range) {
   Polygon *polyarray = (Polygon *) malloc(sizeof(Polygon) * nPolygon);
   unsigned int index;
   for (index = 0; index < nPolygon; index++) {
      polyarray[index].xmin = randn(512);
      polyarray[index].ymin = randn(512);
      polyarray[index].width = 400;
      polyarray[index].height = 300;
      polyarray[index].xmax = polyarray[index].xmin +
          polyarray[index].width;
      polyarray[index].ymax = polyarray[index].ymin +
          polyarray[index].height;
      polyarray[index].index = index;
   }
   return polyarray;
}

// Create random data points.
POINT *createRandomPoints(unsigned int nPoints, unsigned int range) {
   POINT *pointArray = (POINT *) malloc(sizeof(POINT) * nPoints);
   unsigned int index;
   for (index = 0; index < nPoints; index++) {
```

```
      pointArray[index].x = randn(range);
      pointArray[index].y = randn(range);
      pointArray[index].index = index;
   }
   return pointArray;
}

POINT *createPoints(int numPoints) {
   POINT *pointArray;
   pointArray = createRandomPoints(numPoints, rangeSize);
   return pointArray;
}

/**<************************ Kernel function ********************/
/**<************************ Brute force search *********************/
/**
 * Function to search points inside polygon using brute force search
    method.
 * 1- Assign each thread to a data point.
 * 2- The thread checks if the point is within a given polygon.
 * 3- Repeat till all polygons are checked.
 * 4- Store the attributes of the polygon which satisfy the query.
 * 5- Then grid stride is done, so that the thread loops over to get the
    next point.
 ** ***********************************************************************/

// Brute force search
__global__ void bruteForce(POINT* d_pointArray, Polygon* d_Polygon, int
   numPoints, int numPolygon)
```

```
{
   // Get global thread index.
   int tidG = threadIdx.x + blockIdx.x*blockDim.x;
   int pid;
   // Boundary check for threads.
   if(tidG < numPoints)
   {
      d_Polygon[pid].count = 0;
      // Loop till all points are checked.
      for(int tid = tidG; tid < numPoints; tid = tid + blockDim.x*gridDim.x)
      {
         // Loop till all polygons are checked
         for(pid = 0; pid < numPolygon; pid++)
         {
            // Check every point with every polygon.
            if ((d_pointArray[tid].x >= d_Polygon[pid].xmin) &&
                (d_pointArray[tid].x <= d_Polygon[pid].xmax) &&
                (d_pointArray[tid].y >= d_Polygon[pid].ymin) &&
                (d_pointArray[tid].y <= d_Polygon[pid].ymax))
            {
               // Store the points that lie within a polygon.
               d_Polygon[pid].nodePoint[d_Polygon[pid].count].xminPP =
                   d_pointArray[tid].x;
               d_Polygon[pid].nodePoint[d_Polygon[pid].count].yminPP =
                   d_pointArray[tid].y;
               d_Polygon[pid].count++;
            }
         }
      }
```

```
    }
}

/**<********************** Main function **************************/
/**
 * Main function.
 * 1- Create points randomly.
 * 2- Create polygons randomly.
 * 3- Copy points and polygons from host to device.
 * 4- Launch CUDA kernel to perform the search.
 * 5- Time the kernel execution.
 ** ********************************************************************/
int main(int argc, char **argv) {
    // Host variables
    int i;
    float diff_GPU;
    struct Polygon *randomPoly;
    int index;
    int j,p;
    float time = 0.0;
    int NumberofPolygons;
    // Device variables
    POINT *d_pointArray;
    Polygon *d_Polygon;
    // Create data points and polygons randomly.
    POINT *pointArray = createPoints(numPoints);
    randomPoly = createRandomPolygon(numPolygon, rangeSize);
    // Allocate device memory.
    cudaMalloc((void**)&d_pointArray, sizeof(POINT)*numPoints);
```

```
cudaMalloc((void**)&d_Polygon, sizeof(Polygon)*numPolygon);
// Memory transfer from host to device
cudaMemcpy(d_pointArray, pointArray, sizeof(POINT)*numPoints,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_Polygon, randomPoly, sizeof(Polygon)*numPolygon,
    cudaMemcpyHostToDevice);
// Time brute force search.
perfTimer timeRecord;
initTimer(&timeRecord);
cudaEvent_t start_GPU, stop_GPU;
cudaEventCreate(&start_GPU);
cudaEventCreate(&stop_GPU);
cudaEventRecord(start_GPU, 0);
// Kernel launch
bruteForce<<<(numPoints/1024), 1024>>>(d_pointArray, d_Polygon,
    numPoints, numPolygon);
cudaError_t cudaerr = cudaDeviceSynchronize();
if (cudaerr != CUDA_SUCCESS){
   printf("kernel launch failed with error \"%s\".\n",
       cudaGetErrorString(cudaerr));
}
cudaEventRecord(stop_GPU, 0);
cudaEventSynchronize(stop_GPU);
cudaEventElapsedTime(&diff_GPU, start_GPU, stop_GPU);
time = diff_GPU;
// Calculate number of polygons per second
NumberofPolygons = numPolygon / time;
printf("%d\n", NumberofPolygons);
cudaEventDestroy(start_GPU);
```

```
    cudaEventDestroy(stop_GPU);

    return 0;

}
```

APPENDIX B. Header File for Memory Allocation

```
/** ********************************************************************
 *  File Name : MemoryManager.h
 *  Define objects of quadtree.
 *  Define query polygons.
 *  Preallocate memory for all quadtree objects on CPU.
 *  Memory allocation on the GPU.
 ** ********************************************************************/
/**<************************# Includes ********************************/
#include<stdlib.h>
#include<stdio.h>
#include <stdbool.h>

/**<************************ Structure definition *********************/
typedef int POINTID;
typedef int NODEID;
typedef int BUFFID;
typedef int POLYID;

/**
 * The CPU has three main structures, NODE (nodes of the quadtree),
 * POINT (the input points for the quadtree) and
 * POINT_BUFFER (the array of points which each leaf node of the quadtree
    holds).
 */
typedef struct POINT {
  int x;
```

```
    int y;
    int index;
} POINT;

typedef struct POINT_BUFFER {
    // Array of points
    POINTID pointArray;
    unsigned int pointCount;
    // use a linked list to point to another NODE at same level.
    BUFFID nextId;
} POINT_BUFFER;

typedef struct NODE {
    // Level
    unsigned int level;
    // Keep track of type of NODE : LEAF, LINK
    unsigned int type;
    // Location in 2D space
    int posX;
    int posY;
    // Size of quadrant
    int width;
    int height;
    // Description of points
    int count[4];
    int total;
    int index;
    int parent_index;
    int offset;
```

```
   // For Adaptive implementation
   int open;
   NODEID child[4];
   // This handles the 4 regions 0,1,2,3 representing sub-quadrants
   BUFFID pBuffer;
   bool P0, P1, P2, P3;
} NODE;

// Definition of query polygon
typedef struct Polygon {
   // Size of Polygon.
   //xmin, ymin, xmax and ymax defines 4 corners of a polygon
   int xmin;
   int ymin;
   int width;
   int height;
   int xmax;
   int ymax;
   int count;
   int index;
   POLYID *PolyPointId;
} Polygon;

// Structure to store points from completely overlapping nodes in GPU.
typedef struct CP{
   int polygonIndexCP;
   int CPcount;
   NODEID nodeNumCP[64];
}CP;
```

```
// Structure to store points from partially overlapping nodes in GPU
typedef struct NODEPP{
   int xminPP;
   int yminPP;
   int xmaxPP;
   int ymaxPP;
}NODEPP;

typedef struct PP{
   int polygonIndexPP;
   int PPcount;
   NODEPP nodePP_array[64];
}PP;

/**<************************ Memory preallocation **********************/
// Preallocating memory for point structure.
typedef struct{
   POINT* pArray;
   int currCount,maxSize;
}point_array_t;
point_array_t pArr;

void allocatePointMemory(){
   pArr.pArray = (POINT*)malloc(100000000*sizeof(POINT));
   pArr.currCount = 0;
   pArr.maxSize = 1000000000;
   if (pArr.pArray != NULL) {
   }
```

```
}

// Function to fetch memory for point from preallocated memory.
POINT* getPoint(int id){
   if(id >= pArr.currCount || id < 0){
      exit(-1);
   }
   return &pArr.pArray[id];
}

// Allocate memory for a new point.
POINTID getNewPoint(){
   if (pArr.currCount >= pArr.maxSize){
      pArr.pArray = (POINT*)realloc(pArr.pArray, pArr.maxSize + 10000);
      pArr.maxSize+=10000;
   }
   return pArr.currCount++;
}

// Fetch memory for array of points.
POINTID getPointArray(int nelems){
   int strtId = pArr.currCount;
   pArr.currCount+=nelems;
   if (pArr.currCount >= pArr.maxSize){
      pArr.maxSize = (pArr.maxSize + 10000) > pArr.currCount ? pArr.maxSize
          + 10000:pArr.currCount + 1000;
      pArr.pArray = (POINT*)realloc(pArr.pArray,
          (pArr.maxSize)*sizeof(POINT));
   }
```

```
   return strtId;
}

// Preallocating memory for node structure.
typedef struct{
   NODE* nodeArray;
   int currCount,maxSize;
}node_array_t;
node_array_t nodeArr;

void allocateNodeMemory(){
   nodeArr.nodeArray = (NODE*)malloc(1000*sizeof(NODE));
   nodeArr.currCount = 0;
   nodeArr.maxSize = 1000;
}

// Function to fetch memory for point from preallocated memory.
NODE* getNode(int id){
   if(id >= nodeArr.currCount || id < 0){
      exit(-1);
   }
   return &nodeArr.nodeArray[id];
}

// Allocate memory for a new node.
NODEID getNewNode(){
   if (nodeArr.currCount >= nodeArr.maxSize){
      nodeArr.nodeArray = (NODE*)realloc(nodeArr.nodeArray, nodeArr.maxSize
         + 10000);
```

```
      nodeArr.maxSize+=10000;
   }
   return nodeArr.currCount++;
}

// Preallocating memory for point buffer structure.
typedef struct{
   POINT_BUFFER* bufferArray;
   int currCount,maxSize;
}buff_array_t;
buff_array_t bufferArr;

void allocatePOINTBUFFERMemory(){
   bufferArr.bufferArray =
       (POINT_BUFFER*)malloc(4000000*sizeof(POINT_BUFFER));
   bufferArr.currCount = 0;
   bufferArr.maxSize = 4000000;
}

// Function to fetch memory for point buffer from preallocated memory.
POINT_BUFFER* getBUFFER(int id){
   if(id >= bufferArr.currCount || id < 0){
      exit(-1);
   }
   return &bufferArr.bufferArray[id];
}

// Allocate memory for a new buffer.
BUFFID getNewBUFFER(){
```

```
   if (bufferArr.currCount >= bufferArr.maxSize){
      bufferArr.bufferArray = (POINT_BUFFER*)realloc(bufferArr.bufferArray,
          bufferArr.maxSize + 10000);
      bufferArr.maxSize+=10000;
   }
   return bufferArr.currCount++;
}

// Get the node indices of the level 3 nodes.
typedef struct {
   NODEID *Level3Nodes;
   int count;
}level3Node;
level3Node Level3NodeArray;

void levelThreeNode()
{
   Level3NodeArray.Level3Nodes = (NODEID*)malloc(64*sizeof(NODEID));
   int current = getNewNode() - 1 ;
   int i;
   int j = 0;
   for (i = 0; i<=current; i++){
      if(getNode(i)->level == 2){
         Level3NodeArray.count++;
         Level3NodeArray.Level3Nodes[j] = i;
         j++;
      }
   }
}
```

```
/**<****************** CUDA device memory allocation ****************/
// Allocate CUDA memory for Point structure.
POINT *d_POINT;
void allocatePointMemoryCuda(){
   cudaMalloc((void**)&d_POINT, sizeof(POINT)*pArr.currCount);
}

// Allocate CUDA memory for Point buffer structure.
POINT_BUFFER *d_POINT_BUFFER;
void allocatePoint_BufferMemoryCuda(){
   cudaMalloc((void**)&d_POINT_BUFFER,
       sizeof(POINT_BUFFER)*bufferArr.currCount);
}

// Allocate CUDA memory for Node structure.
NODE *d_NODE;
void allocateNodeMemoryCuda(){
   cudaMalloc((void**)&d_NODE, sizeof(NODE)*nodeArr.currCount);
}

//Allocate memory for level 3 nodes.
NODEID *d_NODE_In;
void allocateNodeIDMemoryCuda(){
   cudaMalloc((void**)&d_NODE_In, sizeof(NODEID)*Level3NodeArray.count);
}

/**<****************** Memory transfer from CPU to GPU **************/
// Copy Point structure to GPU.
```

```
void PointCudaCopy(){
   cudaMemcpy(d_POINT, pArr.pArray, sizeof(POINT)*pArr.currCount,
       cudaMemcpyHostToDevice);
}

// Copy Point buffer structure to GPU.
void Point_BufferCudaCopy(){
   cudaMemcpy(d_POINT_BUFFER, bufferArr.bufferArray,
       sizeof(POINT_BUFFER)*bufferArr.currCount, cudaMemcpyHostToDevice);
}

// Copy Node structure to GPU.
void NodeCudaCopy(){
   cudaMemcpy(d_NODE, nodeArr.nodeArray, sizeof(NODE)*nodeArr.currCount,
       cudaMemcpyHostToDevice);
}

//Copy level 3 nodes to GPU.
void NodeIDCudaCopy(){
   cudaMemcpy(d_NODE_In, Level3NodeArray.Level3Nodes,
       sizeof(NODEID)*Level3NodeArray.count, cudaMemcpyHostToDevice);
}
```

```
/** ********************************************************************
 *  File name : quadtreeGPU.cu
 *  Construct quadtree in CPU.
 *  Traverse quadtree in GPU to find PIP.
 ** ********************************************************************/
/**<************************# Includes ********************************/
#include<stdio.h>
#include<stdlib.h>
#include"MemoryManager.h"
#include<unistd.h>
#include<sys/time.h>
#include <stdbool.h>
#include<stdlib.h>
#include<cstdlib>
#include <cuda.h>
#include <math.h>
#ifdef __CDT_PARSER__

/**<************************# Defines *********************************/
#define __host__
#define __shared__
#define CUDA_KERNEL_DIM(...)
#else
#define CUDA_KERNEL_DIM(...) <<< __VA_ARGS__ >>>
#endif
#define BUILD_FULL 1
```

```
#define BUILD_ADAPTIVE 2
#define MODE_RANDOM 1
#define MODE_FILE 2
#define TRUE 1
#define FALSE 0
#define pMax 32
#ifndef RANGE
#define RANGE 1024
#endif

/**<****************** Global variables ****************************/
int pointMode = MODE_RANDOM;
char *inputPointFileName;
char *outputTreeFileName;
int rangeSize = RANGE;
int bucketSize = 32;
int numPoints = 3000;
int numLevels = 3;
int numSearches = 0;
int printTree = 0;
int outputTree = 0;
int quadTreeMode = BUILD_FULL;
int numPolygon = 1099120;
int pointRangeX = RANGE;
int pointRangeY = RANGE;
int completeIndex = 0;
int NotIndex = 0;
int PartialIndex = 0;
int arraysize = 100;
```

```
/**<****************** enums ******************************/
enum {
   TYPE_NONE = 0, TYPE_ROOT, TYPE_LINK, TYPE_LEAF
};

enum {
   FullyOverlapped = 0, PartiallyOverlapped
};

/**<******************* Generic Functions ****************/
/**<******************* Timing Functions *****************/
typedef struct perf_struct {
   double sample;
   struct timeval tv;
} perf;

perf timeRecord;

void init_perf(perf *t) {
   t->sample = 0;
}

// Start time measurement.
void start_perf_measurement(perf *t) {
   gettimeofday(&(t->tv), NULL);
}

// Stop time measurement.
```

```
void stop_perf_measurement(perf *t) {
   struct timeval ctv;
   gettimeofday(&ctv, NULL);
   struct timeval diff;
   timersub(&ctv, &(t->tv), &diff);
   double time = diff.tv_sec + diff.tv_usec / 1000000.0;
   t->sample = time;
}

// Print time difference.
void print_perf_measurement(perf *t) {
   double total_time = 0.0;
   double stdev = 0.0;
}

/**<************************ Random number generator *********************/
/**
 * Generate random numbers.
 * 1- Generate random data points.
 * 2- Generate random numbers for polygon corners.
 ** ********************************************************************/
int randn(int range) {
   int a;
   a = rand() % range;
   return a;
}

int randnRng(int N, int M)
{
```

```
   int r;
   // Initialize random seed
   srand (time(NULL));
   // Generate number between a N and M
   r = M + random() / (RAND_MAX / (N - M + 1) + 1);
   return r;
}

// Creates random polygon with random xmin, ymin, width and height.
Polygon *createRandomPolygon(unsigned int nPolygon, unsigned int range) {
   Polygon *polyarray = (Polygon *) malloc(sizeof(Polygon) * nPolygon);
   int PointArraySize = getNewPoint() -1 ;
   unsigned int index;
   for (index = 0; index < nPolygon; index++) {
      polyarray[index].xmin = randn(723);
      polyarray[index].ymin = randn(723);
      polyarray[index].width = randnRng(100, 300);
      polyarray[index].height = randnRng(100, 300);
      polyarray[index].xmax = polyarray[index].xmin +
          polyarray[index].width;
      polyarray[index].ymax = polyarray[index].ymin +
          polyarray[index].height;
      polyarray[index].count = 0;
      polyarray[index].index = index;
   }
   return polyarray;
}

// Generate random data points, given a number of points and its range.
```

```
POINTID createRandomPoints(unsigned int nPoints, unsigned int range) {
   POINTID pointArray = getPointArray(nPoints);
   unsigned int index;
   for (index = 0; index < nPoints; index++) {
      POINT *p=getPoint(pointArray+index);
      p->x=randn(range);
      p->y=randn(range);
      p->index=index;
   }
   return pointArray;
}

/**<************************ Tree Functions ***************************/
/**<************************ Set node *********************************/
/**
 * Set node parameters.
 * 1- Set appropriate values for x, y, width, height and level of a node.
 * 2- Initialize rest of the node parameters.
 ** ********************************************************************/
void setNode(NODEID nodeid, int x, int y, int w, int h, int type, int
    level) {
   // Get memory for node.
   NODE* node=getNode(nodeid);
   // Set the 5 parameters.
   node->posX = x;
   node->posY = y;
   node->width = w;
   node->height = h;
   node->level = level;
```

```
   // Reset all of the tracking values.
   int i;
   for (i = 0; i < 4; i++)
   {
      node->child[i] = -1;
      node->count[i] = 0;
   }
   node->total = 0;
   node->index = 0;
   node->offset = 0;
   node->open = TRUE;
   node->type = type;
   node->pBuffer = -1;
}

/**<*************** Count number of nodes ******************/
int countNodesQuadTree(NODEID nodeid) {
   int sum = 0;
   int i;
   if(nodeid == -1)
      return 0;
   // Depth first traversal to find the total number of nodes.
   if (getNode(nodeid)->type == TYPE_LEAF) {
      return 1;
   } else {
      for (i = 0; i < 4; i++) {
         sum = sum + countNodesQuadTree(getNode(nodeid)->child[i]);
      }
   }
```

```
   return sum + 1;
}

/**<*************** Assign index and offset ******************/
/**
 * 1- DFS traversal of tree to assign indices.
 * 2- Count data points in leaf node.
 ** ***********************************************************/
void walkNodeTable(NODE *parent, NODEID nodeid, int *offset, int *index) {
   NODE* node=getNode(nodeid);
   BUFFID ptrID;
   int i;
   if (node == NULL)
      return;
   if (parent)
      node->parent_index = parent->index;
   else
      node->parent_index = -1;
   // Assign index and offset.
   node->index = *index;
   node->offset = *offset;
   // Advance the next index.
   *index = *index + 1;
   if (node->type == TYPE_LEAF) {
      int count = 0;
      // Get indices of points
      for (ptrID = node->pBuffer; ptrID != -1; ptrID =
          getBUFFER(ptrID)->nextId){
         POINT_BUFFER *ptr= getBUFFER(ptrID);
```

```
         count = count + ptr->pointCount;
      }
      // Assign total number of points in node.
      node->total = count;
      *offset = *offset + count;
   } else {
      for (i = 0; i < 4; i++) {
         walkNodeTable(node, node->child[i], offset, index);
      }
   }
}

/**<************************ Build quadtree ****************************/
/**<************************ Create new point buffer *******************/
/**
 * Create new point buffer.
 * 1- Get memory for new buffer from preallocated memory.
 * 2- Allocate memory inside each buffer to hold 32 data points.
 * 3- Assign first point to buffer and set point count.
 * 4- Initialize pointer to next buffer.
 ** ********************************************************************/
BUFFID newPointBuffer(POINT *p) {
   // Allocate memory for point buffer.
   BUFFID ptrID = getNewBUFFER();
   POINT_BUFFER *ptr= getBUFFER(ptrID);
   // Allocate a bucket for 32 data points.
   (ptr->pointArray) = getPointArray(bucketSize);
   // Get point and set parameters.
   *(getPoint(ptr->pointArray)) = *p;
```

```
   ptr->pointCount = 1;
   ptr->nextId = -1;
   return ptrID;
}

/**<************************ Assign point to a node ********************/
/**
 * Add a point to leaf node.
 * 1- If a buffer of a leaf node is not full, add points to that buffer.
 * 2- If the buffer is full, then add new buffer to end of list.
 ** ***************************************************************/
void addPointToNode(NODE *node, POINT *p) {
   BUFFID ptrID;
   BUFFID lastBufferID;
   // Add points till buffer is full.
   if (node->pBuffer != -1) {
      for ( ptrID = node->pBuffer; ptrID != -1; ptrID =
          getBUFFER(ptrID)->nextId) {
         POINT_BUFFER *ptr= getBUFFER(ptrID);
         // Add to the end of the list.
         if (ptr->pointCount < bucketSize) {
            *(getPoint((ptr->pointArray)+(ptr->pointCount))) = *p;
            ptr->pointCount++;
         }
         BUFFID lastBufferID = ptrID;
      }
      // Boundary case of adding a new link.
      getBUFFER(lastBufferID)->nextId = newPointBuffer(p);
   } else {
```

```
      node->pBuffer = newPointBuffer(p);
   }
}

/**<****************** Get direction of node ***************/
/**
 * 1- Get node direction based on input data point.
 * 2- A point is checked to see if it lies in NW, NE, SW or SE direction
     of a node.
 * 3- Return direction to calling function.
 ** ********************************************************/
int getNodeDirection(NODE *nParent, struct POINT *p) {
   int posX, posY;
   int x, y;
   int index;
   // Get the point.
   x = p->x;
   y = p->y;
   // Child width and height
   int width = nParent->width / 2;
   int height = nParent->height / 2;
   // Decide direction (North west (NW), North east (NE), South west (SW),
       South east (SE) of a point).
   for (index = 0; index < 4; index++) {
      switch (index) {
      case 0: // NW
         posX = nParent->posX;
         posY = nParent->posY + height;
         if ((x >= posX) && (x < posX + width) && (y >= posY)
```

```
            && (y < posY + height)) {
         return 0;
      }
      break;
   case 1: // NE
      posX = nParent->posX + width;
      posY = nParent->posY + height;
      if ((x >= posX) && (x < posX + width) && (y >= posY)
            && (y < posY + height)) {
         return 1;
      }
      break;
   case 2: // SW
      posX = nParent->posX;
      posY = nParent->posY;
      if ((x >= posX) && (x < posX + width) && (y >= posY)
            && (y < posY + height)) {
         return 2;
      }
      break;
   case 3: // SE
      posX = nParent->posX + width;
      posY = nParent->posY;
      if ((x >= posX) && (x < posX + width) && (y >= posY)
            && (y < posY + height)) {
         return 3;
      }
      break;
   }
```

```
    }


    exit(-1);
    return (-1);
}


/**<******************* Build node ****************************/
/**
 * 1- Check the direction of the node.
 * 2- Calculate and assign the node's x, y, width and height based on
     direction.
 * 3- Assign the node level.
 ** ************************************************************/
void buildNode(NODEID node, NODE *nParent, int direction) {
    int posX, posY, width, height, level;
    switch (direction) {
    case 0: // NW
        posX = nParent->posX; //0
        posY = nParent->posY + nParent->height / 2; //512
        break;
    case 1: // NE
        posX = nParent->posX + nParent->width / 2;
        posY = nParent->posY + nParent->height / 2;
        break;
    case 2: // SW
        posX = nParent->posX;
        posY = nParent->posY;
        break;
    case 3: // SE
```

```
      posX = nParent->posX + nParent->width / 2;
      posY = nParent->posY;
      break;
   }
   // Width and height of the child node is simply 1/2 parent.
   width = nParent->width / 2;
   height = nParent->height / 2;
   // Set the level.
   level = nParent->level + 1;
   setNode(node, posX, posY, width, height, TYPE_NONE, level);
}
NODEID createChildNode(NODE *parent, int direction) {
   NODEID node = getNewNode();
   buildNode(node, parent, direction);
   return (node);
}

/**<************************ Build full quadtree ************************/
/**
 * 1- Get node direction.
 * 2- Build node in that direction.
 * 3- Repeat till bottom of tree is reached.
 * 4- If bottom of the tree is reached, then add point to leaf node's
    buffer.
 ** ********************************************************************/
void buildFullTree(NODEID nodeid, unsigned int level, POINT *p) {
   NODEID dirNode;
   NODEID child;
   int direction;
```

```
   NODE*node=getNode(nodeid);
   // Check if bottom of tree is reached.
   if (node->level == level) {
      addPointToNode(node, p);
   } else {
      // Get direction of point in the node.
      direction = getNodeDirection(node, p);
      dirNode = node->child[direction];
      if (dirNode!=-1) {
         buildFullTree(dirNode, level, p);
      } else {
         // Create child node in that direction.
         child = createChildNode(node, direction);
         node->child[direction] = child;
         // Assign node type.
         if (getNode(child)->level == level)
            getNode(child)->type = TYPE_LEAF;
         else
            getNode(child)->type = TYPE_LINK;
         buildFullTree(node->child[direction], level, p);
      }
   }
}

/**<************************ Build adaptive quadtree ********************/
/**
 * Quadtree grows based on input points.
 * 1- Get direction of the node based on the input point.
 * 2- Build node in that direction.
```

```
 * 3- Add points to the node's buffer till predetermined limit is reached.
 * 4- Once limit is reached, divide node to sub nodes.
 * 5- Build child nodes and push points to it and repeat.
 ** ********************************************************************/
void buildAdaptiveTree(NODEID node, unsigned int level, POINT *p);

void pushPointToChildren(NODE *node, unsigned int level, POINT *p) {
   NODEID dirNode;
   NODEID child;
   int direction;
   // Get node direction.
   direction = getNodeDirection(node, p);
   dirNode = node->child[direction];
   if (dirNode) {
      buildAdaptiveTree(dirNode, level, p);
   } else {
      // Build node in that direction.
      child = createChildNode(node, direction);
      node->child[direction] = child;
      getNode(child)->type = TYPE_LEAF;
      buildAdaptiveTree(node->child[direction], level, p);
   }
}

void pushAllNodePointsToChildren(NODE *node, unsigned int level) {
   BUFFID ptrID;
   int link = 0;
   int i;
   ptrID = node->pBuffer;
```

```
   if (ptrID == -1)
      return;
   POINT_BUFFER *ptr= getBUFFER(ptrID);
   // Should have only 1 bucket's worth.
   if (ptr->nextId != -1) {
      printf("pushAllNodePointsToChildren: error\n");
      exit(-1);
   }
   // Get direction of node and push points to the node's buffer.
   for (i = 0; i < ptr->pointCount; i++) {
      pushPointToChildren(node, level, (getPoint((ptr->pointArray)+i)));
   }
   node->pBuffer = -1;
   node->open = FALSE;
   if (node->type == TYPE_LEAF)
      node->type = TYPE_LINK;
}

void buildAdaptiveTree(NODEID nodeid, unsigned int level, POINT *p) {
   NODEID dirNode;
   NODEID child;
   int direction;
   NODE*node=getNode(nodeid);
   // Have we reached the bottom : force to put point there : linked
       buckets
   if (getNode(nodeid)->level == level) {
      addPointToNode(node, p);
      return;
   }
```

```
if (node->open == FALSE) {
   // If got to here, then this is an empty link node and we push point
      down.
   pushPointToChildren(node, level, p);
   return;
}
// All of the following checks assume node is open.
// Node is open but point buffer is empty.
if (node->pBuffer == -1) {
   addPointToNode(node, p);
   return;
}
// Check if the point belongs on this node.
if ((node->pBuffer != -1) && (getBUFFER(node->pBuffer)->pointCount <
    bucketSize)) {
   // Add to current pointBuffer.
   POINT_BUFFER *ptr = getBUFFER(node->pBuffer);
   POINT* pt = getPoint(ptr->pointArray+ptr->pointCount);
   pt = p;
   getBUFFER(node->pBuffer)->pointCount++;
   return;
}
if ((node->pBuffer != -1) && (getBUFFER(node->pBuffer)->pointCount ==
    bucketSize)) {
   // Full Buffer
   pushPointToChildren(node, level, p);
   // Push all points and delete this node's buffer.
   pushAllNodePointsToChildren(node, level);
   return;
```

```
   }
   printf("Should never get here \n");
   exit(-1);
}

/**<************************ Build quadtree ************************/
void buildQuadTree(NODEID node, unsigned int level, POINTID pointArray,
      int nPoints, int treeType) {
   int i;
   for (i = 0; i < nPoints; i++) {
      if (treeType == BUILD_FULL)
         buildFullTree(node, level, getPoint(pointArray+i));
      else
         buildAdaptiveTree(node, level, getPoint(pointArray+i));
   }
}


/**<************************ Print functions ************************/
/**
 * Functions to print quadtree.
 * 1- DFS traversal of quadtree.
 * 2- Print quadtree node details to a file.
 * 3- Print data points in leaf node to a file.
 * ** ********************************************************************/
void printTableNode(FILE *fp, NODEID nodeid) {
   int i;
   if(nodeid==-1)
      return;
```

```
   NODE*node=getNode(nodeid);
   // Print node details of tree to file.
   fprintf(fp, "%d %d : [%d %d] %d %d : %d ", node->index, node->offset,
        node->posX, node->posY, node->width, node->height, node->total);
   fprintf(fp, "next line \n");
   fprintf(fp, " %d :", node->parent_index);
   for (i = 0; i < 4; i++) {
      int index = -1;
      if (node->child[i]!=-1) {
         index = getNode(node->child[i])->index;
      }
      fprintf(fp, " %d", index);
   }
   fprintf(fp, "\n");
}

// Print node details to file.
void printTableNodeDataFile(FILE *fp, NODEID nodeid) {
   int i;
   if (nodeid == -1)
      return;
   printTableNode(fp, nodeid);
   for (i = 0; i < 4; i++) {
      printTableNodeDataFile(fp, getNode(nodeid)->child[i]);
   }
}

// Print index of data point.
void printLeafPointsDataFile(FILE *fp, NODEID nodeid) {
```

```
   BUFFID ptrID;
   int i;
   if(nodeid==-1){
      return;
   }
   NODE* node=getNode(nodeid);
   if (node->type == TYPE_LEAF) {
      // Print indices of points.
      for (ptrID = node->pBuffer; ptrID != -1; ptrID =
          getBUFFER(ptrID)->nextId) {
         POINT_BUFFER *ptr= getBUFFER(ptrID);
         for (i = 0; i < ptr->pointCount; i++) {
            fprintf(fp, "%d\n", getPoint((ptr->pointArray)+i)->index);
         }
      }
   } else {
      for (i = 0; i < 4; i++) {
         printLeafPointsDataFile(fp, node->child[i]);
      }
   }
}

// Print x, y coordinates of points in leaf node to file.
void printQuadTreeDataFile(NODEID root, char *outputFile, POINTID
    pointArray,
      int nPoints) {
   FILE *fp;
   int i;
   fp = fopen(outputFile, "w");
```

```
    if (fp == NULL) {
        puts("Cannot open file");
        exit(1);
    }
    fprintf(fp, "%d %d\n", pointRangeX, pointRangeY);
    fprintf(fp, "%d\n", nPoints);
    for (i = 0; i < nPoints; i++) {
        fprintf(fp, "%d %d\n", getPoint(pointArray+i)->x,
            getPoint(pointArray+i)->y);
    }
    int countNodes = countNodesQuadTree(root);
    fprintf(fp, "%d\n", countNodes);
    int offset = 0;
    int index = 0;
    // Calculate: offset and index per node
    walkNodeTable(NULL, root, &offset, &index);
    // Print
    printTableNodeDataFile(fp, root);
    // Print all points.
    printLeafPointsDataFile(fp, root);
    fclose(fp);
}

// Print point details along with leaf node details.
void printQuadTreeLevel(NODEID nodeid, int level) {
    if(nodeid==-1)
        return;
    NODE* node=getNode(nodeid);
    BUFFID ptrID;
```

```
   if (node->level == level) {
      printf(" Node <%d> L=%d [%d %d] (%d %d)\n", node->type, node->level,
           node->posX, node->posY, node->width, node->height);
      if (node->pBuffer != -1) {
         int link = 0;
         int i;
         for (ptrID = node->pBuffer; ptrID != -1; ptrID =
             getBUFFER(ptrID)->nextId) {
            POINT_BUFFER *ptr= getBUFFER(ptrID);
            printf(" Link %d : ", link);
            for (i = 0; i < ptr->pointCount; i++) {
               printf("(%d : %d,%d) ", getPoint((ptr->pointArray)+i)->index,
                     getPoint((ptr->pointArray)+i)->x,
                         getPoint((ptr->pointArray)+i)->y);
            }
            printf("\n");
            link++;
         }
      }

   }
   int i;
   for (i = 0; i < 4; i++) {
      printQuadTreeLevel(node->child[i], level);
   }
}

// Print details of a node passed to the function.
void printQuadTree(NODEID nodeid) {
```

```
   if(nodeid==-1)
      return;
   NODE* node=getNode(nodeid);
   printf(" Node <%d> L=%d [%d %d] (%d %d)\n", node->type, node->level,
        node->posX, node->posY, node->width, node->height);
   if (node->pBuffer != -1) {
      BUFFID ptrID;
      int link = 0;
      int i;
      for (ptrID = node->pBuffer; ptrID != -1; ptrID
          =getBUFFER(ptrID)->nextId) {
         printf(" Link %d : ", link);
         POINT_BUFFER *ptr = getBUFFER(ptrID);
         for (i = 0; i < ptr->pointCount; i++) {
            printf("(%d : %d,%d) ", getPoint((ptr->pointArray)+i)->index,
                  getPoint((ptr->pointArray)+i)->x,
                     getPoint((ptr->pointArray)+i)->y);
         }
         printf("\n");
         link++;
      }
   }
   int i;
   for (i = 0; i < 4; i++) {
      printQuadTree(node->child[i]);
   }
}

// Print node details along with point density.
```

```
int printNodeStats(NODEID nodeid, int printDetails) {
   int children[4];
   BUFFID ptrID;
   if(nodeid==-1)
      return 0;
   NODE*node=getNode(nodeid);
   int count = 0;
   int sum = 0;
   if (node->pBuffer != -1) {
      int link = 0;
      int i;
      for (ptrID = node->pBuffer; ptrID != -1; ptrID =
          getBUFFER(ptrID)->nextId) {
         POINT_BUFFER *ptr= getBUFFER(ptrID);
         for (i = 0; i < ptr->pointCount; i++) {
            count++;
         }
      }
      sum++;
   }
   int i;
   if (printDetails) {
      for (i = 0; i < 4; i++) {
         children[i] = 0;
         if (node->child[i])
            children[i] = 1;
      }
      printf(" QuadNode <%d> L=%d [%d %d] (%d %d) %d %f : %d %d %d %d \n",
            node->type, node->level, node->posX, node->posY, node->width,
```

```
            node->height, count,
            (float) count / (float) (node->height * node->width),
            children[0], children[1], children[2], children[3]);
   }
   for (i = 0; i < 4; i++) {
      sum += printNodeStats(node->child[i], printDetails);
   }
   return sum;
}

void printQuadTreeStats(NODEID rootNode, unsigned int level, int
    printDetails) // level =4, printDetails =0
{
   int sum;
   sum = printNodeStats(rootNode, printDetails);
}

/**<************************ Search functions *********************/

/**<************************ Search point ***************************/
/**
 * Function to search point sequentially.
 * 1- Measures time taken to search a point from a point array.
 ** ********************************************************************/
// Function to search a given point from a set of random points.
double searchPoints(POINTID pointArray, int numPoints, POINTID searchArray,
      int numSearches) {
   int s, p;
   int matches = 0;
```

```
   long long cost = 0;
   POINT *point, *search;
   // Time the search.
   start_perf_measurement(&timeRecord);
   for (p = 0; p < numPoints; p++) {
      point = getPoint(pointArray+p);
      for (s = 0; s < numSearches; s++) {
         search = getPoint(searchArray+s);
         // Update cost of search and keep count on number of matches.
         cost++;
         if ((search->x == point->x) && (search->y == point->y)) {
            matches++;
         }
      }
   }
   stop_perf_measurement(&timeRecord);
   print_perf_measurement(&timeRecord);
   return (timeRecord.sample);
}

double searchSmartPoints(POINTID pointArray, int numPoints, POINTID
    searchArray,
      int numSearches) {
   int s, p;
   int matches = 0;
   long long cost = 0;
   int i;
   POINT *point, *search;
   char *maskArray = (char *) malloc(sizeof(char) * numSearches);
```

```
   for (i = 0; i < numSearches; i++) {
      maskArray[i] = TRUE;
   }
   // Time the search
   start_perf_measurement(&timeRecord);
   for (p = 0; p < numPoints; p++) {
      point = getPoint(pointArray+p);
      for (s = 0; s < numSearches; s++) {
         if (maskArray[s] == FALSE)
            continue;
         search = getPoint(searchArray+s);
         cost++;
         if ((search->x == point->x) && (search->y == point->y)) {
            matches++;
            maskArray[s] = FALSE;
         }
      }
   }
   stop_perf_measurement(&timeRecord);
   print_perf_measurement(&timeRecord);
   return (timeRecord.sample);
}

/**<********************* Search point in quadtree **********************/
/**
 * Functions to search point using quadtree.
 * 1- Measures time taken to search point in a quadtree.
 * 2- Keeps track of the cost of search and the number of matches.
 * 3- Search function is implemented for both full and adaptive quadtree.
```

```
 ** ************************************************************************/
// Find node in which point lies for a full quadtree.
NODEID findQuadTreeNode(NODEID nParentid, struct POINT *p) {
   int posX, posY;
   int x, y;
   int index;
   if(nParentid==-1)
      return nParentid;
   NODE* nParent=getNode(nParentid);
   if (nParent->type == TYPE_LEAF)
      return nParentid;
   // Get the point.
   x = p->x;
   y = p->y;
   // Child width and height
   int width = nParent->width / 2;
   int height = nParent->height / 2;
   for (index = 0; index < 4; index++) {
      switch (index) {
      case 0: // NW
         posX = nParent->posX;
         posY = nParent->posY + height;
         if ((x >= posX) && (x < posX + width) && (y >= posY)
               && (y < posY + height)) {
            return findQuadTreeNode(nParent->child[0], p);
         }
         break;
      case 1: // NE
         posX = nParent->posX + width;
```

```
            posY = nParent->posY + height;
            if ((x >= posX) && (x < posX + width) && (y >= posY)
                  && (y < posY + height)) {
               return findQuadTreeNode(nParent->child[1], p);
            }
            break;
         case 2: // SW
            posX = nParent->posX;
            posY = nParent->posY;
            if ((x >= posX) && (x < posX + width) && (y >= posY)
                  && (y < posY + height)) {
               return findQuadTreeNode(nParent->child[2], p);
            }
            break;
         case 3: // SE
            posX = nParent->posX + width;
            posY = nParent->posY;
            if ((x >= posX) && (x < posX + width) && (y >= posY)
                  && (y < posY + height)) {
               return findQuadTreeNode(nParent->child[3], p);
            }
            break;
         }
      }
      return -1;
}

// Function to find node in adaptive quadtree.
NODEID descendQuadTreeNode(NODEID nParentid, struct POINT *p) {
```

```
int posX, posY;
int x, y;
int index;
if (nParentid == -1)
   return -1;
NODE*nParent=getNode(nParentid);
// This node has points.
if (nParent->pBuffer != -1)
   return nParentid;
// Get the point.
x = p->x;
y = p->y;
// Child width and height
int width = nParent->width / 2;
int height = nParent->height / 2;
for (index = 0; index < 4; index++) {
   switch (index) {
   case 0: // NW
      posX = nParent->posX;
      posY = nParent->posY + height;
      if ((x >= posX) && (x < posX + width) && (y >= posY)
            && (y < posY + height)) {
         return descendQuadTreeNode(nParent->child[0], p);
      }
      break;
   case 1: // NE
      posX = nParent->posX + width;
      posY = nParent->posY + height;
      if ((x >= posX) && (x < posX + width) && (y >= posY)
```

```
                && (y < posY + height)) {
            return descendQuadTreeNode(nParent->child[1], p);
          }
          break;
        case 2: // SW
          posX = nParent->posX;
          posY = nParent->posY;
          if ((x >= posX) && (x < posX + width) && (y >= posY)
                && (y < posY + height)) {
            return descendQuadTreeNode(nParent->child[2], p);
          }
          break;
        case 3: // SE
          posX = nParent->posX + width;
          posY = nParent->posY;
          if ((x >= posX) && (x < posX + width) && (y >= posY)
                && (y < posY + height)) {
            return descendQuadTreeNode(nParent->child[3], p);
          }
          break;
        }
      }
      return -1;
}

// Search full quadtree.
double searchPointsFullQuadTree(NODEID nodeid, POINTID pointArray, int
    numPoints,
      POINTID searchArray, int numSearches) {
```

```
int i;
int s;
int matches = 0;
int index;
long long cost = 0;
POINT point, *search;
BUFFID ptrID;
NODEID leaf;
// Time the search.
start_perf_measurement(&timeRecord);
for (s = 0; s < numSearches; s++) {
   search = getPoint(searchArray+s);
   // search the node
   leaf = findQuadTreeNode(nodeid, search);
   if (leaf == -1)
      continue;
   // Get the points from node and check for match.
   for (ptrID = getNode(leaf)->pBuffer; ptrID != -1; ptrID =
       getBUFFER(ptrID)->nextId) {
      POINT_BUFFER *ptr= getBUFFER(ptrID);
      for (i = 0; i < ptr->pointCount; i++) {
         point = *(getPoint((ptr->pointArray)+i));
         cost++;
         if ((search->x == point.x) && (search->y == point.y)) {
            matches++;
            break;
         }
      }
   }
```

```
   }
   stop_perf_measurement(&timeRecord);
   print_perf_measurement(&timeRecord);
   return (timeRecord.sample);
}

// Search adaptive quadtree.
double searchPointsAdaptiveQuadTree(NODEID root, POINTID pointArray,
      int numPoints, POINTID searchArray, int numSearches) {
   int i;
   int s;
   int matches = 0;
   int index;
   long long cost = 0;
   POINT point,* search;
   BUFFID ptrID;
   NODEID node;
   // Time the search.
   start_perf_measurement(&timeRecord);
   for (s = 0; s < numSearches; s++) {
      search = getPoint(searchArray+s);
      // Search the node.
      node = descendQuadTreeNode(root, search);
      // No information
      if (node == -1) {
         continue;
      }
      // Get the points from node and check for match.
      for (ptrID = getNode(node)->pBuffer; ptrID != -1; ptrID =
```

```
         getBUFFER(ptrID)->nextId) {
         POINT_BUFFER *ptr= getBUFFER(ptrID);
         for (i = 0; i < ptr->pointCount; i++) {
            point = *(getPoint((ptr->pointArray)+i));
            cost++;
            if ((search->x == point.x) && (search->y == point.y)) {
               matches++;
               continue;
            }
         }
      }
   }
   stop_perf_measurement(&timeRecord);
   print_perf_measurement(&timeRecord);
   return (timeRecord.sample);
}

/**<************************ Support system ***************************/
//  Get data points from file.
POINTID createFilePoints(char *inputFileName) {
   FILE *fdin;
   if (inputFileName == NULL) {
      printf("Error opening file\n");
      exit(1);
   }
   fdin = fopen(inputFileName, "r");
   if (fdin == NULL) {
      printf("Error opening file\n");
      exit(1);
```

```
   }
   if (fscanf(fdin, "%d %d\n", &pointRangeX, &pointRangeY) != 2) {
      printf("Error point file\n");
      exit(1);
   }
   if (fscanf(fdin, "%d\n", &numPoints) != 1) {
      printf("Error point file\n");
      exit(1);
   }
   POINTID pointArray = getPointArray(numPoints);
   int index;
   int x, y;
   for (index = 0; index < numPoints; index++) {
      if (fscanf(fdin, "%d %d\n", &x, &y) != 2) {
         printf("Error point file\n");
         exit(1);
      }
      POINT*p=getPoint(pointArray+index);
      p->x = x;
      p->y = y;
      p->index = index;
   }
   return pointArray;
}

// Create points randomly or read from file.
POINTID createPoints(int numPoints) {
   POINTID pointArray;
   if (pointMode == MODE_RANDOM) {
```

```
      pointArray = createRandomPoints(numPoints, rangeSize);
   }
   else {
      pointArray = createFilePoints(inputPointFileName);
   }
   return pointArray;
}

static void usage(char *argv0) {
   char *help = "Usage: %s [switches] -n num_points\n"
        "       -s number_search_points\n"
        "       -l number_levels\n"
        "       -b bucket_size\n"
        "       -p            : print quadtree \n"
        "       -h            : print this help information\n";
   fprintf(stderr, help, argv0);
   exit(-1);
}

// compilation options.
void setup(int argc, char **argv) {
   int opt;
   while ((opt = getopt(argc, argv, "n:l:s:b:r:i:o:aph")) != EOF) {
      switch (opt) {
      case 'n':
         numPoints = atoi(optarg);
         break;
      case 'l':
         numLevels = atoi(optarg);
```

```
      break;
   case 's':
      numSearches = atoi(optarg);
      break;
   case 'b':
      bucketSize = atoi(optarg);
      break;
   case 'r':
      rangeSize = atoi(optarg);
      break;
   case 'i':
      inputPointFileName = optarg;
      pointMode = MODE_FILE;
      break;
   case 'o':
      outputTreeFileName = optarg;
      outputTree = 1;
      break;
   case 'p':
      printTree = 1;
      break;
   case 'a':
      quadTreeMode = BUILD_ADAPTIVE;
      break;
   case 'h':
   default:
      usage(argv[0]);
      break;
   }
```

```
    }
}


/**<****************** Kernel function ***********************/
/**
 * Iterative BFS traversal of quadtree to find points inside polygon.
 * 1- Start traversal at level 3.
 * 2- Assign warp to polygon and thread within each warp to quadtree nodes.
 * 3- Check if node overlaps polygon.
 * 4- If node overlaps, then traverse tree from this node.
 * 5- If the bottom of the tree is reached, then classify nodes based on
     kind of overlap.
 * 6- Completely overlapping nodes - Get the range of points from node
     boundary.
 * 7- Partially overlapping nodes - Check every point in node with polygon
     boundary.
 * 8- Get the range and store.
 * 9- Loop over to evaluate next set of polygons.
 ** ************************************************************/
// Function to check if a polygon is smaller than a given node.
__device__ bool searchPolygonCUDA(int x0, int x1, int xp0, int xp1, int
    y0, int y1, int yp0, int yp1)
{
    bool N0 = false;
    bool N1 = false;
    bool N2 = false;
    bool N3 = false;
    // Check if a polygon is within a node
```

```
   if ((xp0 >= x0) && ((xp0) <= (x1)) && (yp0 >= y0) && ((yp0) <= (y1))) {
      N0 = true;
   }
   if ((xp1 >= x0) && ((xp1) <= (x1)) && (yp0 >= y0) && ((yp0) <= (y1))) {
      N1 = true;
   }
   if ((xp0 >= x0) && ((xp0) <= (x1)) && (yp1 >= y0) && ((yp1) <= (y1))) {
      N2 = true;
   }
   if ((xp1 >= x0) && ((xp1) <= (x1)) && (yp1 >= y0) && ((yp1) <= (y1))) {
      N3 = true;
   }
   if((N0 == false) && (N1 == false) && (N2 == false) && (N3 == false))
   {
      return true;
   }
   return false;
}

// Function to check completely overlapping conditions.
__device__ void NodeCheckComplete(NODEID nodeid, NODE node, Polygon
    indexOfPolyArray, int**complete, int**partial, int**notN) {
   if(nodeid != NODEID(-1)){
      // x, y coordinates, width and height of node.
      int x0, y0, x1, y1, xp0, yp0, xp1, yp1, w, h;
      int i, j, numOfPoints;
      bool P0 = false;
      bool P1 = false;
      bool P2 = false;
```

```
bool P3 = false;
// x, y coordinates, width and height of polygon.
xp0 = indexOfPolyArray.xmin;
yp0 = indexOfPolyArray.ymin;
xp1 = indexOfPolyArray.xmax;
yp1 = indexOfPolyArray.ymax;
x0 = node.posX;
y0 = node.posY;
x1 = x0 + node.width;
y1 = y0 + node.height;
// Check for completely overlapping node.
if ((x0 >= xp0) && ((x0) <= (xp1)) && (y0 >= yp0) && ((y0) <= (yp1)))
    {
   P0 = true;
}
if ((x1 >= xp0) && ((x1) <= (xp1)) && (y0 >= yp0) && ((y0) <= (yp1)))
    {
   P1 = true;
}
if ((x0 >= xp0) && ((x0) <= (xp1)) && (y1 >= yp0) && ((y1) <= (yp1)))
    {
   P2 = true;
}
if ((x1 >= xp0) && ((x1) <= (xp1)) && (y1 >= yp0) && ((y1) <= (yp1)))
    {
   P3 = true;
}
// If all corners of a node is within a polygon, then classify node
    as completely overlapping.
```

```
      if ((P0 == true) && (P1 == true) && (P2 == true)
            && (P3 == true)) {
         **complete = 1;
      }
      // If all corners of a node is not within a polygon, then classify
          node as not overlapping.
      else if ((P0 == false) && (P1 == false) && (P2 == false)
            && (P3 == false) && (searchPolygonCUDA(x0, x1, xp0, xp1, y0,
                y1, yp0, yp1) == true))
      {
         **notN = 1;


      }
      // Classify remaining nodes as partially overlapping.
      else{
         **partial = 1;
      }
   }

}

// Function to check not overlapping conditions.
__device__ void NodeCheckNotOverlap(NODEID nodeid, NODE node, Polygon
    indexOfPolyArray, int**complete, int**partial, int**notN) {
   if(nodeid != (-1)){
      int x0, y0, x1, y1, xp0, yp0, xp1, yp1, w, h;
      int i, j, numOfPoints;
      bool P0 = false;
      bool P1 = false;
```

```
bool P2 = false;
bool P3 = false;
xp0 = indexOfPolyArray.xmin;
yp0 = indexOfPolyArray.ymin;
xp1 = indexOfPolyArray.xmax;
yp1 = indexOfPolyArray.ymax;
x0 = node.posX;
y0 = node.posY;
x1 = x0 + node.width;
y1 = y0 + node.height;
// Check for not overlapping node.
if ((x0 >= xp0) && ((x0) <= (xp1)) && (y0 >= yp0) && ((y0) <= (yp1)))
    {
   P0 = true;
}
if ((x1 >= xp0) && ((x1) <= (xp1)) && (y0 >= yp0) && ((y0) <= (yp1)))
    {
   P1 = true;
}
if ((x0 >= xp0) && ((x0) <= (xp1)) && (y1 >= yp0) && ((y1) <= (yp1)))
    {
   P2 = true;
}
if ((x1 >= xp0) && ((x1) <= (xp1)) && (y1 >= yp0) && ((y1) <= (yp1)))
    {
   P3 = true;
}
if ((P0 == false) && (P1 == false) && (P2 == false)
      && (P3 == false) && (searchPolygonCUDA(x0, x1, xp0, xp1, y0,
```

```
                y1, yp0, yp1) == true))
      {
         **notN = 1;
      }
      else
      {
         **notN = 0;
      }
   }
}

// Iterative BFS traversal of quadtree to find points inside polygon.
__global__ void searchOverlapNodeCUDA(NODE* d_NODE, NODEID* d_NODE_In,
    Polygon* d_Polygon, int d_nodeCount, int d_level3Count, int
    d_numPolygon, int QuadtreeLevel, POINT* d_POINT, POINT_BUFFER*
    d_POINT_BUFFER, CP* d_cp, PP* d_pp){
   // Global thread index.
   int tid = blockDim.x * blockIdx.x + threadIdx.x;
   int b_tid = blockDim.x * blockIdx.x + threadIdx.x;
   int val = -1;
   int val1 = -1;
   int *completeNode ,*partialNode, *notNode;
   completeNode = &val;
   partialNode = &val1;
   notNode = &val;
   int NumWarp = 32;
   int threadnumber = (64/32);
   // Keeps count of nodes in the shared memory array.
   __shared__ int nv[32];
```

```
__shared__ int nc[32];
// Variables to store range of points.
int minCheckX = 1024;
int maxCheckX = 0;
int minCheckY = 1024;
int maxCheckY = 0;
int nodeIndexPP = 0;
int numlevelsQ = 1;
// Initialize array.
if(threadIdx.x < NumWarp)
{
   nv[threadIdx.x] = 0;
   nc[threadIdx.x] = 0;
}
// one shared memory array per polygon with array size of 64.
__shared__ int boundary[50*64];
// Query polygon global Id.
int qid1 = tid/32;
// Query polygon local Id.
int qidd = threadIdx.x / 32;
// Starting local index of query Polygon.
int qb = (threadIdx.x / 32)*64;
// Thread index within a warp.
int lane_id = tid%32;
int nid;
int iter = 0;
int x0, y0, x1, y1, xp0, yp0, xp1, yp1;
// Initialize shared memory array
if(threadIdx.x < (64*50))
```

```
{
   boundary[threadIdx.x] = -1;
}
// Assign warp to a polygon and loop till all polygons are evaluated.
for(int qid = qid1; qid < d_numPolygon; qid+=(32*gridDim.x))
{
   // Boundary check for warp Id.
   if(qid < d_numPolygon)
   {
      // Initialize arrays to store completely and partially overlapping
          node points.
      d_cp[qid].CPcount = 0;
      d_pp[qid].PPcount = 0;
      // Boundary of polygon
      xp0 = d_Polygon[qid].xmin;
      yp0 = d_Polygon[qid].ymin;
      xp1 = d_Polygon[qid].xmax;
      yp1 = d_Polygon[qid].ymax;
      // Start traversal at level 3 of quadtree and check for overlap
          conditions.
      if(lane_id < d_level3Count){
         int nodeIndex = d_NODE_In[lane_id];
         if(nodeIndex != -1)
         {
            NodeCheckNotOverlap(d_NODE_In[lane_id], d_NODE[nodeIndex],
                d_Polygon[qid], &completeNode, &partialNode, &notNode);
            if((*notNode) == 0)
            {
               // Add nodes that satisfy query to shared memory array.
```

```
            boundary[(qb + atomicAdd(&nc[qidd], 1))] =
                d_NODE_In[lane_id];
            notNode = &val;
        }
    }
}
while(iter < numlevelsQ){
    __syncthreads();
    // Assign node count to local variable.
    int nb = nc[qidd];
    nc[qidd] = 0;
    // Boundary check to launch threads based on the nodes in the
        array.
    if (lane_id < nb){
        nid = boundary[qb + lane_id];
        // Initialize array after each iteration.
        if(threadIdx.x < (64*50))
        {
            boundary[threadIdx.x] = -1;
        }
        for(int i = 0; i < 4; i++)
        {
            if(nid != -1)
            {
                // Check for overlap conditions for child nodes.
                NodeCheckNotOverlap(d_NODE[nid].child[i],
                    d_NODE[d_NODE[nid].child[i]], d_Polygon[qid],
                    &completeNode, &partialNode, &notNode);
                if((*notNode) == 0) {
```

```
                if(nid != -1){

                    // Add nodes that satisfy the query to array.
                    boundary[(qb + atomicAdd(&nc[qidd],1))] =
                        d_NODE[nid].child[i];
                    notNode = &val;

                }
            }
        }
    }
    iter = iter + 1;
}
__syncthreads();
// Assign threads in a warp to nodes and get the points within a
    node that overlaps a polygon.
for(int id = lane_id; id < nc[qidd]; id = id + 32)
{
    nid = boundary[qb + id];
    if(nid != -1){
        // Completely overlapping nodes
        NodeCheckComplete(nid, d_NODE[nid], d_Polygon[qid],
            &completeNode, &partialNode, &notNode);
        if((*completeNode) == 1)
        {

            // Get node boundary and index of the polygon.
            completeNode = &val;
```

```
   BUFFID ptrID;
   NODE node = d_NODE[nid];
   if(node.pBuffer)
   {
      d_cp[qid].nodeNumCP[atomicAdd(&(d_cp[qid].CPcount), 1)]
          = nid;
      d_cp[qid].polygonIndexCP = qid;
   }

}
// partially overlapping nodes
else if (*partialNode == 1)
{

   partialNode = &val;
   BUFFID ptrID;
   NODE node = d_NODE[nid];
   if(node.pBuffer)
   {
      for (ptrID = node.pBuffer; ptrID != -1; ptrID =
          d_POINT_BUFFER[ptrID].nextId)
      {
         if(nid != -1)
         {
            // Check every point within the leaf node against
                polygon boundary.
            POINT_BUFFER ptr = d_POINT_BUFFER[ptrID];
            for (int c = 0; c < (ptr.pointCount); c++)
            {
```

```
            if((d_POINT[(ptr.pointArray)+c].x >= xp0) &&
                (d_POINT[(ptr.pointArray)+c].x <= xp1) &&
                (d_POINT[(ptr.pointArray)+c].y >= yp0) &&
                (d_POINT[(ptr.pointArray)+c].y <= yp1))
            {
               // Store the range of points.
               if(minCheckX > d_POINT[(ptr.pointArray)+c].x
                   )
               {
                  minCheckX = d_POINT[(ptr.pointArray)+c].x;
               }
               if(maxCheckX < d_POINT[(ptr.pointArray)+c].x
                   )
               {
                  maxCheckX = d_POINT[(ptr.pointArray)+c].x;
               }
               if(minCheckY > d_POINT[(ptr.pointArray)+c].y
                   )
               {
                  minCheckY = d_POINT[(ptr.pointArray)+c].y;
               }
               if(maxCheckY < d_POINT[(ptr.pointArray)+c].y
                   )
               {
                  maxCheckY = d_POINT[(ptr.pointArray)+c].y;
               }
            }
         }
      }
```

```
                        nodeIndexPP = atomicAdd(&(d_pp[qid].PPcount), 1);
                        d_pp[qid].nodePP_array[nodeIndexPP].xminPP =
                            minCheckX;
                        d_pp[qid].nodePP_array[nodeIndexPP].yminPP =
                            maxCheckX;
                        d_pp[qid].nodePP_array[nodeIndexPP].xmaxPP =
                            minCheckY;
                        d_pp[qid].nodePP_array[nodeIndexPP].ymaxPP =
                            maxCheckY;
                        d_pp[qid].polygonIndexPP = qid;
                    }
                }

            }
        }
    }

  }

}

/**<************************ Main function ***************************/
/**
 * Two techniques to build QuadTrees
 * 1- full : extend all the way down, only leafs hold points
 *         : counts are kept at intermediate levels
 *         : nulls are still used to know where points are.
```

```
 * 2- adaptive : items are pushed around as needed to form tree
 *         : points of LIMIT pushed down.
 ** ********************************************************************/
int main(int argc, char **argv) {
  setup(argc, argv);
  // Host variables.
  int index;
  NODEID rootNode;
  struct Polygon *randomPoly;
  // Device variable declaration.
  Polygon *d_Polygon;
  CP *d_cp;
  PP *d_pp;
  PP *d_hostPoint;
  // Preallocate memory for all objects in CPU.
  allocatePointMemory();
  allocateNodeMemory();
  allocatePOINTBUFFERMemory();
  d_hostPoint=(PP*)malloc(numPolygon*sizeof(PP));
  // Create random points and polygon.
  POINTID pointArray = createPoints(numPoints);
  randomPoly = createRandomPolygon(numPolygon, rangeSize);
  // Get memory for root node.
  rootNode = getNewNode();
  // Start node : root
  setNode(rootNode, 0, 0, rangeSize, rangeSize, TYPE_ROOT, 0);
  // Create the quadtree.
  buildQuadTree(rootNode, numLevels, pointArray, numPoints, quadTreeMode);
  //Time quadtree construction.
```

```
double buildTime = timeRecord.sample;
printf("QuadTreeBuild Time : %f\n", buildTime);
// Print the quadtree.
if (printTree) {
   printQuadTree(rootNode);
}
printQuadTreeStats(rootNode, numLevels, 0);
if (outputTree)
{
   printQuadTreeDataFile(rootNode, outputTreeFileName, pointArray,
       numPoints);
}
// Search section
if (numSearches > 0)
{
   // Create some search points.
   POINTID searchArray = createRandomPoints(numSearches, rangeSize);
   // Search points in an array.
   double baseTime = searchPoints(pointArray, numPoints, searchArray,
        numSearches);
   double smartTime = searchSmartPoints(pointArray, numPoints,
       searchArray,
        numSearches);
   // Search points using quadtree.
   double quadTime;
   if (quadTreeMode == BUILD_FULL)
      quadTime = searchPointsFullQuadTree(rootNode, pointArray,
          numPoints,
            searchArray, numSearches);
```

```
   else
   {
      quadTime = searchPointsAdaptiveQuadTree(rootNode, pointArray,
            numPoints, searchArray, numSearches);
   }
}
//CUDA memory allocation.
allocatePointMemoryCuda();
allocatePoint_BufferMemoryCuda();
allocateNodeMemoryCuda();
cudaMalloc((void**)&d_Polygon, sizeof(Polygon)*numPolygon);
cudaMalloc((void**)&d_cp, sizeof(CP)*numPolygon);
cudaError_t err = cudaMalloc((void**)&d_pp, sizeof(PP)*numPolygon);
// Time the kernel execution.
float diff_GPU;
cudaEvent_t start_GPU, stop_GPU;
cudaEventCreate(&start_GPU);
cudaEventCreate(&stop_GPU);
cudaEventRecord(start_GPU, 0);
// Copy nodes at level 3.
levelThreeNode();
allocateNodeIDMemoryCuda();
int d_nodeCount = nodeArr.currCount;
int d_level3Count = Level3NodeArray.count;
//Polygon CUDA memory allocation.
PointCudaCopy();
Point_BufferCudaCopy();
NodeCudaCopy();
NodeIDCudaCopy();
```

```
cudaMemcpy(d_Polygon, randomPoly, sizeof(Polygon)*numPolygon,
    cudaMemcpyHostToDevice);
//Launch kernel with 65535 blocks and 1024 threads per block.
searchOverlapNodeCUDA<<<65535, 1024>>>(d_NODE, d_NODE_In, d_Polygon,
    d_nodeCount, d_level3Count, numPolygon, numLevels, d_POINT,
    d_POINT_BUFFER, d_cp, d_pp);
cudaError_t cudaerr = cudaDeviceSynchronize();
if (cudaerr != CUDA_SUCCESS){
   printf("kernel launch failed with error \"%s\".\n",
       cudaGetErrorString(cudaerr));
}
cudaEventRecord(stop_GPU, 0);
cudaEventSynchronize(stop_GPU);
cudaEventElapsedTime(&diff_GPU, start_GPU, stop_GPU);
// Memory transfer from device to host.
cudaMemcpy(d_hostPoint, d_pp, sizeof(PP)*numPolygon,
    cudaMemcpyDeviceToHost);
// Calculation of number polygons per second.
int NumberofPolygons = numPolygon / (diff_GPU/1000);
printf("%d\n", NumberofPolygons);
// Destroy CUDA event.
cudaEventDestroy(start_GPU);
cudaEventDestroy(stop_GPU);
// Free CUDA memory.
cudaFree(d_Polygon);
cudaFree(d_POINT_BUFFER);
cudaFree(d_POINT);
cudaFree(d_NODE);
return 0;
```

```
}
```