A GPU-Based Framework for Parallel Spatial Indexing and Query Processing

Zhila Nouri Lewis
University of South Florida

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Scholar Commons Citation
https://scholarcommons.usf.edu/etd/8660

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
A GPU-Based Framework for Parallel Spatial Indexing and Query Processing

by

Zhila Nouri Lewis

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Yi-Cheng Tu, Ph.D.
Sriram Chellappan, Ph.D.
Hao Zheng, Ph.D.
Tapas Das, Ph.D.
Sagar Pandit, Ph.D.

Date of Approval:
October 24, 2019

Keywords: High performance computing, Query processing optimization, GPU computing

Copyright © 2019, Zhila Nouri Lewis
Dedication

I dedicate this dissertation to my loving parents, Reza and Sedigheh who taught me the value of education. I will always appreciate their words of encouragement and loyal supports. They never stop giving of themselves in numerous ways.

I also dedicate this dissertation to my beloved husband, Michael who constantly supports me. He made me a better, stronger and more accomplished person. I am truly thankful for having him and his love in my life.
Acknowledgments

This journey would not have been possible without the support of my caring family and my knowledgeable supervisor. I would like to express my deepest appreciation to my supervisor, Dr. Yi-Cheng Tu, whose expertise was invaluable in conducting this research. He has been a great mentor for me during my PhD education.

I would also like to thank my warmhearted and lovely parents, Reza and Sedigheh for standing by my side with their true love and endless support.

My sincere thanks also goes to my husband, Michael who made me believe in myself. He has always been an unfailing source of support and encouragement.

Last but not the least, I would like to thank my sister, Leila and my niece, Romina for their spiritual support throughout my life.
# Table of Contents

List of Tables iii

List of Figures iv

Abstract vi

Chapter 1: Introduction 1
  1.1 High Performance in Tree Construction and Query Processing 2
  1.2 Handling Data Updates 3
  1.3 Multi-GPU Support 4
  1.4 Multi-Query Optimization 4
  1.5 Overview of Our Approach 5
  1.6 Dissertation Organization 7

Chapter 2: Literature Review 8
  2.1 Overview of GPU Architecture and CUDA 9
  2.2 Related Work on Tree Construction and Update on GPUs 12
  2.3 Related Work on Query Processing on GPUs 15

Chapter 3: Tree Construction in G-PICS 19
  3.1 Space-driven Partitioning in Tree Construction 21
    3.1.1 Overview of G-PICS Tree Construction 21
    3.1.2 Tree Construction Routines 23
    3.1.3 Cost Modeling 26
  3.2 Data-driven Partitioning in Tree Construction 27
    3.2.1 Overview of Our Approach 27
    3.2.2 Tree Construction Routines 30
  3.3 Tree Construction in Multiple GPUs 31
    3.3.1 Tree Construction Routines 32
  3.4 Experimental Evaluations 36
    3.4.1 G-PICS Performance in a Single GPU Environment 36
    3.4.2 G-PICS Performance in a Multi-GPU Environment 38
List of Tables

Table 3.1  User-specified parameters for tree construction  20
Table 6.1  Symbols and notations 65
## List of Figures

| Figure 1.1 | Traditional tree search versus parallel linear search against all leaf nodes | 3 |
| Figure 1.2 | Overview of G-PICS framework | 6 |
| Figure 2.1 | Modern GPU architecture | 10 |
| Figure 3.1 | An example of quadtree construction in G-PICS with $MC = MH = 3$ | 24 |
| Figure 3.2 | Final quadtree built based on the data inputs in Figure 3.1 | 26 |
| Figure 3.3 | Combining private histograms in all blocks to output the final histogram | 29 |
| Figure 3.4 | An example of finding the median value via histogramming | 31 |
| Figure 3.5 | Transferring Counter and Split arrays in Figure 3 via CPU master to four other GPUs comparing to GPU master approach with three CUDA streams, each containing two kernels on master GPU | 33 |
| Figure 3.6 | An example of data transfer between GPUs | 33 |
| Figure 3.7 | Constructing the second level of the quadtree in Figure 3.1 using four GPUs | 34 |
| Figure 3.8 | Full quadtree construction for data input in Figure 3.1 using four GPUs | 35 |
| Figure 3.9 | Final quadtree built based on the data inputs in Figure 3.8 | 35 |
| Figure 3.10 | G-PICS tree construction time, and speedup over comparative algorithms | 37 |
| Figure 3.11 | Speedup of G-PICS tree construction using GM over using CM and Nvidia code in a multi-GPU system | 38 |
| Figure 4.1 | An example of registering range search queries into the query lists of leaf nodes | 41 |
Figure 4.2  An example of direct output buffer for GPU threads  
Figure 4.3  Processing time for search queries in G-PICS  
Figure 4.4  Speedup ((a): step I only; (b): step II only; (c): total time) of G-PICS over M-STIG, M-GAT, and P-CPU in processing 1,000 to 4,000,000 concurrent queries  
Figure 4.5  Speedup in query processing time under low concurrency ((a): step I only; (b): step II only; (c): total time) of G-PICS over M-STIG, M-GAT, and P-CPU  
Figure 4.6  G-PICS multi-GPU search query processing Speedup over single GPU  
Figure 4.7  G-PICS multi-GPU query processing time for two million input queries  
Figure 5.1  Example of updating a quadtree with $MC = 3$ and $MH = 3$  
Figure 5.2  Tree update procedure in G-PICS  
Figure 5.3  G-PICS update performance over Scratch  
Figure 5.4  G-PICS multi-GPU update performance over Scratch  
Figure 6.1  G-PICS SDH computation time, and speedup over CPU  
Figure 6.2  G-PICS step I spatial join time, and speedup over M-STIG
Abstract

Support for efficient spatial data storage and retrieval have become a vital component in almost all spatial database systems. Previous work has shown the importance of using spatial indexing and parallel computing to speed up such tasks. While GPUs have become a mainstream platform for high-throughput data processing in recent years, exploiting the massively parallel processing power of GPUs is non-trivial. Current approaches that parallelize one query at a time have low work efficiency and cannot make good use of GPU resources. On the other hand, many spatial database applications are busy systems in which a large number of queries arrive simultaneously. In this research, we present a comprehensive framework named G-PICS for parallel processing of a large number of spatial queries on GPUs. G-PICS encapsulates efficient parallel algorithms for constructing a variety of spatial trees with different space partitioning methods. G-PICS also provides highly optimized programs for processing major spatial query types, and such programs can be accessed via an API that could be further extended to implement user-defined algorithms. While support for dynamic data inputs is missing in existing work, G-PICS implements efficient parallel algorithms for bulk updates of data. Furthermore, G-PICS is designed to work in Multi-GPU environments to support datasets beyond the size of a single GPU’s global memory. Empirical evaluation of G-PICS shows significant performance improvement over the state-of-the-art GPU and parallel CPU-based spatial query processing systems. In particular, G-PICS achieves double-digit speedup over such systems in tree construction (up to 53X), and query processing (up to 80X). Moreover, tree update procedure outperforms the tree construction from scratch under different levels of data movement (up to 16X).
Chapter 1: Introduction

Spatio-temporal data has become a critical element in a broad range of applications such as geographic information systems, mobile computing, scientific computing, epidemic simulation, and astrophysics. Due to the high data volume and large query quantities, support for efficient spatial data storage and query processing has become a vital component in such systems. Popular spatial queries are spatial point search, range search, within-distance search, and k-nearest neighbors (kNNs) \[1, 2\]. Parallelization is a common approach in achieving high performance while dealing with spatial databases. Previous work has also demonstrated the great potential of parallel computing in achieving high performance query processing \[3, 4\]. However, if parallelism is adopted without spatial data indexing in query processing, the performance gain obtained will fade away quickly as data size increases \[5, 6\].

Previous work \[7\] has shown the importance of the two-step spatial query processing using spatial indexing as follows: (1) the scope of the search is filtered; and (2) candidates that satisfy the search condition in the filtered area are retrieved \[8, 3, 4, 9, 10\].

Over the last decade, many-core hardware has been widely used to speed up high-performance computing (HPC) applications. Among them, Graphical Processing Units (GPUs) have become a mainstream platform \[11\]. Spatial query processing on GPUs has also attracted much attention from the research community. Related work in this topic \[12, 13, 14\] focuses on parallelizing one search query at a time on GPUs. In \[12\], a GPU-based spatial index called STIG (Spatio-Temporal Indexing using GPUs) based on kd-tree

\[1\]Some part of this chapter was published in ACM proceedings of the 30th International Conference on Scientific and Statistical Database Management. Permission is included in Appendix A.
is presented. In [14], a framework called GAT (GPU-accelerated Framework for Processing Trajectory Queries) is developed to support processing trajectory range queries and top-k similarity queries on GPUs. GAT is based on a quadtree-like index and cell-level trajectory representations. In [13], another variation of quadtree called Scout is developed to support spatio-temporal data visualization on GPUs. With plausible innovations, the above systems successfully demonstrated the potential of GPU-based spatial indexing, and also generated abundant opportunities for further research. This research aims at a more comprehensive spatial indexing framework with even better performance and support of functionalities beyond query processing. In particular, we address the following issues:

1.1 High Performance in Tree Construction and Query Processing

In all the aforementioned work, a spatial tree consists of intermediate nodes, and a set of leaf blocks to store the spatial data records in consecutive memory locations. A two-step spatial query processing strategy is adopted in such work: (1) all leaf blocks are searched in a brute-force manner to identify those that satisfy the search conditions; and (2) all data points in the identified leaf blocks are examined to determine the final results. It is not easy to achieve a high degree of parallelism in the first step using traditional logarithmic tree search, especially when higher levels of the tree are visited. Hence, they adapt a data parallel solution for the first step on GPU, in which all the leaf blocks are transferred to GPU and scanned in a brute-force manner. However, by scanning all leaf nodes, such an approach is inefficient as it literally changes the total amount of work (work efficiency) from logarithmic to linear (Figure 1.1). Although they take advantage of the thousands of GPU cores to process leaf nodes concurrently, the speedup can be quickly offset by the growing number of leaf nodes in large datasets. In GAT, to achieve a logarithmic work efficiency, the first step is done on CPU by using a quadtree-based filtering method. Then, only the leaf blocks identified in the first step are transferred to GPU, and the second step is parallelized.
on GPU. Although the overall query processing performance is improved in GAT, it still suffers from the overhead caused by transferring the intersecting leaf blocks to GPU global memory thus has much room for improvement.

1.2 Handling Data Updates

An essential element that is missing from existing work is the support of data updates. In such work, the tree is constructed in host memory and transferred to GPU’s global memory for query processing. In large datasets, building a tree is costly, and furthermore, the overhead of transferring data from CPU to GPU is significant. For static data, it is not an essential issue as tree construction and transferring is a one-time cost. However, almost all location-based services involve dynamic data. Without explicitly handling tree updates as in existing work, the tree will have to be rebuilt and transferred to the GPU every time there is update of data.
1.3 Multi-GPU Support

With today’s data-intensive applications, efficient support for datasets that cannot fit into the GPU’s global memory is necessary. To address that, GAT [14] uses a memory allocation table (MAT) to keep track of the leaf blocks residing in global memory. Therefore, before a query is launched on GPU, the MAT is first checked to see if queries’ intersecting leaf blocks are in global memory. If not, such blocks have to be copied to global memory before query processing. In case the global memory does not have enough capacity for new leaf blocks, following a LRU swapping strategy, some leaf blocks are swapped out from global memory to make capacity for new blocks. Therefore, each time the overhead of transferring data from host memory to GPU memory is added to query processing. Thus, an essential step towards developing high performance spatial query processing in large datasets, is to reduce such overhead.

1.4 Multi-Query Optimization

In existing approaches, by processing one query at a time, optimization opportunities among different queries in a workload are wasted. For example, in the second step of the search, since each query scans a list of leaf nodes to find their data records, the same data record can be accessed many times by different queries in a workload. Consequently, the program easily hits a performance ceiling due to congestion of global memory while other high performance resources are either insufficiently utilized or largely unused (e.g., shared memory). Another drawback of these approaches is that query processing cannot proceed without CPU intervention.

It is well-known that many location-based applications are busy systems with very high query arrival rate [15, 16]. For example, in scientific simulations such as molecular and astrophysical simulations, millions of spatial queries such as kNNs and range searches are
issued at every step of the simulation [17]. Therefore, there are optimization opportunities in co-processing concurrent queries. In [18], GAT is extended to support processing more than one query at a time by parallelizing each individual input query using the solution introduced in GAT. However, in this approach the number of queries that can be run simultaneously is limited to those that their intersecting leaf blocks can fit in GPU global memory. Therefore, the degree of parallelism is low and this approach cannot be used in query processing systems with high query arrival rate.

1.5 Overview of Our Approach

We present the G-PICS (GPU-based Parallel Indexing for Concurrent Spatial data processing) framework for high performance spatial data management and concurrent query processing. G-PICS is implemented as an extensible software package that supports various types of spatial trees under different hardware (GPU) specifications. Query processing approach in G-PICS bears logarithmic work efficiency for each query yet overcomes the problem of low parallelism. Therefore, instead of parallelizing a single tree-search operation, our strategy is to parallelize multiple queries running concurrently. Batch query processing, due to the effective sharing of computing resources, has been heavily studied in the database field [19, 20, 21], and large-scale web search on GPUs [22]. The batch query processing approach in our solution achieves task parallelism on GPUs, allowing each thread to run an individual query. A search query can therefore be done in logarithmic steps. Because each query carries roughly the same work and is independent to others, it is easy to justify the use of parallel hardware.

G-PICS encapsulates all the key components for efficient parallel query processing within GPU with little CPU intervention. It includes highly efficient parallel algorithms for constructing a wide range of space partitioning trees based on user-input parameters. For example, users can choose to build trees with different node degrees and node partitioning
G-PICS provides APIs for processing major spatial queries including spatial point search, range search, within-distance search, and kNNs. Such APIs enable efficient development of more complex spatial data retrieval algorithms and applications. Figure 1.2 shows an overview of G-PICS framework.

G-PICS processes a group of spatial queries at a time, with each query assigned to a thread. Similar to existing work, query processing is accomplished in two steps. In the first step, following a traditional tree search approach, the leaf node(s) that contain the resulting data of each query are identified. However, instead of retrieving all the resulting data, we only register each query to its corresponding leaf nodes. In the second step, following a *query-passive* design, the data in each leaf node is scanned only once and distributed to the
list of queries pre-registered with that leaf node. The highly-organized access to data records yields great locality therefore can make good use of GPU cache. Meanwhile, all the accesses to the global memory are coalesced. We conduct comprehensive experiments to validate the effectiveness of G-PICS. Our experimental results show performance boosts up to 80X (in both throughput and query response time) over best-known parallel GPU and parallel CPU-based spatial query processing systems. The G-PICS tree construction algorithms remarkably outperform the best-known parallel GPU-based algorithms – speedup up to 53X is reported. Moreover, tree update procedure outperforms the tree construction from scratch even under very high rate of data movement (up to 16X). G-PICS takes advantage of multiple GPU cards in a system to support large datasets with good scalability – by increasing the number of GPUs, we observe almost linear speedup.

In addition, we exploit the spatial indexing solutions developed using the algorithms within G-PICS to optimize the performance of existing problem such as spatial join and distance-based histogram approach.

### 1.6 Dissertation Organization

In Chapter 2, we review some of the related work to this research; in Chapter 3, we introduce the tree construction algorithms developed in G-PICS framework; in Chapter 4, we elaborate on query processing algorithms within G-PICS; in Chapter 5, we present parallel update algorithms designed in G-PICS to support dynamic datasets on GPUs; in Chapter 6, we present two case studies in which developed algorithms in G-PICS are exploited to improve the performance of these problems on GPUs, and in Chapter 7, we conclude our work.
Chapter 2: Literature Review

The presentation of multidimensional data has been an essential issue in spatio-temporal applications such as spatial database management systems, computer graphics, geographic information systems, mobile computing, scientific computing, astrophysics, just to name a few. There exist many feasible representations for a set of multidimensional data objects. However, the feasibility of these representations highly depends on the number of attributes and their domains. The easiest way to store a set of data points is using a sequential list, which does not presume any order for any of the attributes. However, with today’s high data volume, support for efficient spatial data storage has become very critical. Index-based access methods (trees), which reduce the amount of data to be processed [7], are shown to be the prominent techniques to efficiently organize and store big datasets [23]. Having index-based data structures, efficient algorithms that utilize the partitioned nature of data in these data structures for data processing usually outperform others that do not use partitioned data by orders of magnitude.

On the other hand, support for efficient spatial query processing in spatio-temporal systems using index-based data structures has become a vital component. Popular spatial queries are spatial point search, range search, within-distance search, and k-nearest neighbors (kNNs) [1, 2]. Cost of index-based data structure construction and update, and consequently query processing over the index-based data structures is very significant in big datasets. Construction and updating of index-based data structures by itself is very signifi-

\[\text{Some part of this chapter was published in ACM proceedings of the 30th International Conference on Scientific and Statistical Database Management. Permission is included in Appendix A.}\]
cant due to the costs involved in such tasks in big datasets. Previous work has demonstrated the great potential of parallel computing in achieving high performance in performing such tasks [3, 4]. It has been proven that merely sequential algorithms have shown very poor performance for most of the applications dealing with large data sets. Parallelism refers to the use of multiple processors simultaneously to solve a single problem.

Over the last decade, many-core hardware has been adapted to speed up high-performance computing (HPC) applications [24]. Among them, Graphical Processing Units (GPUs) have become a mainstream platform for massively parallel data processing [11]. In this chapter, first, we overview the GPU architecture and its programming model. Then, we review some of the previous work that utilizes GPUs for parallel index-based structure construction on GPUs. Finally, we discuss the previous work that exploit GPUs for query processing on GPUs.

2.1 Overview of GPU Architecture and CUDA

Figure 2.1 [25] shows the architecture of a typical modern GPU. A modern GPU usually has thousands of thin computational cores that are organized into an array of streaming multiprocessors (SMs). Each SM contains a number of streaming processors (cores) that share instruction cache and control logic for handling the computational processing. A GPU device has several layers of memory with different accessibility, and accessing costs. First, there are multiple gigabytes of off-chip Global Memory which is a DRAM and uses a DDR5 technology to which all the SMs have read and write accesses simultaneously, and has the largest capacity on GPUs; however, it is the most costly to access. This memory is connected to CPU memory using a PCI-E link. Therefore, CPU can transfer data to the GPU and save it there for future processing on GPU or GPU can transfer results back to CPU through this link. Global memory is not cached; however, one way to hide costly access latency is coalescing the memory transactions. There are also two other memories in the same location
Figure 2.1: Modern GPU architecture
of the global memory with the same visibility: read-only data cache (texture memory) and constant memory. These two memories are used to cache read-only data (data that cannot be overwritten in the program). The difference among these two constant memories is based on their caching strategies. Global loads can cache in L2 cache which is shared by all SMs. Second, each SM has programmable high-speed shared memory (usually of size about 48KB) and non-programmable L1/L2 cache for local memory accesses.

Considering the software side for programming, Nvidia designed a parallel computing platform named CUDA to develop programs in order to run on GPUs. A parallel program developed to run on GPUs is called a kernel. In order to execute a kernel function, CUDA programming paradigm allows a great number of threads to be launched simultaneously. The entire collection of launched threads is called grid, which also will be organized into smaller groups called blocks. The number of threads in a block will be determined based on the size of available shared memory, GPU architecture and size of input data to be processed. Every thread in a block is identified with a thread ID and each block in a grid also has a block ID. Therefore, each thread in a grid is identified by its block ID and thread ID in that block. All threads in a block will be run in the same SM. Moreover, each SM can run multiple blocks. Each GPU block is divided into single-instruction multiple-data (SIMD) batched of threads which called warps. Each warp consists of 32 threads with consecutive thread IDs. Although there are a lot of threads in a block, just those in a warp can execute simultaneously. Each warp is synchronous. Therefore, the slowest thread determines the run time for a warp. This should be taken into account in order to develop efficient programs on GPUs. Every thread has its own private local memory as well as registers. Threads in a block can share information through shared memory on SM.

Every GPU device has a specific compute compatibility number for the memory size, number of registers, etc. It also has support for specific type of instructions or programming models which are determined by the architectural model. Over the past few years, the GPUs
architectural model has improved and developed through multiple generations starting from Fermi [26], and has the following revolutionary trend: Kepler [27], Maxwell [28], Pascal [29, 30], and Volta. The newer an architecture is, the more computational resource it has. Moreover, recent architectures provide more support for programming. For example, Kepler architecture allows for calling a GPU kernel inside an existing kernel to support dynamic parallelism. Due to the complex hardware architecture and programming model, developing programs optimized towards achieving high performance on GPUs is non-trivial and traditional wisdom geared towards (even multi-core) CPU implementations is often found to be ineffective.

2.2 Related Work on Tree Construction and Update on GPUs

The needs of many applications require efficient data storage and retrieval via spatial indexes (space-partitioning trees) such as quadtree, and k-d tree [31]. Space-partitioning trees are hierarchical data structures in which a certain space is recursively divided into subregions. There are two groups of space-partitioning trees: data-driven, and space-driven. In the data-driven scheme, space partitioning is based on the input data, versus in space-driven the partitioning is only based on the space-specific rules. Selection of the appropriate space-partitioning tree for indexing the input data highly depends on the input data type and the search applications. For example, due to the highly balanced aspect ratio in space-driven partitioning trees, these trees are highly qualified for a large number of applications such as proximity searches, and range searches. In addition, space-driven partitioning trees are well-suited for indexing the uniformly-distributed or dynamic datasets [32]. Moreover, space-driven partitioning trees are shown to be highly compatible with parallel architecture especially GPUs [33, 34]. On the other hand, data-driven partitioning trees can be easily balanced (logarithmic depth) which is optimized for indexing highly skewed datasets.
Early work on GPU-based spatial indexes focused on computer graphics applications [35, 36, 37], with an aim for efficient triangle partitioning. In most existing work about spatial query processing on GPUs, spatial indexes are constructed on CPU and shipped to GPU for query processing [12, 13, 14, 14]. There are few work that focused on building the tree structure on GPUs, which are discussed in this section. In [34] an algorithm for parallel construction of Bounding Volume Hierarchies (BVHs) on GPUs was developed. BVHs play a significant role in various scientific fields such as computer graphics and geometry. Point-Region (PR) quadtrees are simplified variations of BVHs in 2-D space. The idea introduced in [34] was later used in [38, 39, 40] to convert the PR quadtree construction on GPUs to an equivalent level-by-level bucket sort problem – quadtree nodes are considered as buckets and input data points are sorted into the buckets based on their locations at each level.

In those parallel quadtree construction approaches, the nodes’ partitioning process at each level is parallelized by applying an out-place quadtree bucket sort algorithm on the data list of the nodes that have to get partitioned in that level– nodes that number of points in them exceed maximum node capacity ($MC$) before reaching maximum height ($MH$) of the tree. The partitioning process continues until there is no more node that needs to get partitioned or $MH$ of the tree is reached. In [38], the parallelization at each level is done by assigning one thread to each node that should get partitioned in that level; since the parallelization at higher level is poor, the first few levels usually are built on CPU, and the lower levels of the tree are built on GPU. Consequently, the first few levels are built sequentially at a higher cost. Moreover, the cost of transferring those levels from CPU to GPU is added to the total tree construction cost. In order to improve this problem, in [39] each block of threads is assigned to one node that should get partitioned at higher levels of the tree. Each active thread in a block is assigned to a set of points in that node to find the position of those points in the next level of the tree. However, in lower levels of the tree each node that needs to get partitioned is assigned to one thread. This idea was later optimized
by Nvidia to build a quadtree [40], in which each block of threads is assigned to a node that has to get partitioned, and each warp in a block works independently on building a child node in the next level of the tree. Then, each overflow child node launches another block of threads to build the next level of the tree from that node using CUDA dynamic parallelism. However, since the number of non-empty nodes is not known in advance, in [38, 39] in order to allocate node memory for the next level of the tree, four times of the number of nodes at the current level will be allocated for the next level. In [40], node memory allocation is done in advance by allocating the the maximum number of possible nodes based on the input $MH$. Therefore, all these solutions suffer from empty node expansion problem on GPU, and consequently leads to inefficient use of GPU global memory. In addition, moving data points from parents to children at each level is out-place through global memory accesses which is of a high cost on GPUs.

K-d tree is a data-driven data structure, which is generally constructed level-by-level by dividing data points in each node into two partitions based on the median value of a chosen dimension. The dimensions in the data domain are chosen in a round-robin fashion at consecutive levels of the tree. In [41] a parallel solution for k-d tree construction on GPUs using a breadth first search (BFS) algorithms is presented. In this approach, at each BFS step, a single thread is assigned to each node with the same distance from the root that needs to get partitioned. Then, in each node the median value is selected by sorting the data points in the developing dimension using the CUDA Thrust library. Although recent sorting implementations on GPUs show high throughput, sorting does more work than necessary for finding the median value [42]. Moreover, sorting is not a good solution in dynamic datasets, because every time input data points move to new locations, the entire dataset in each dimension should be sorted again.

There is no previous work directly targeted parallel data updates on spatial trees. In [43] a sequential algorithm on CPUs for updating PR quadtrees in which the maximum node
capacity is equal to one was proposed. Their simulation mainly concentrates on consecutive phenomena such as fluids and smoke, in which if a point moves in a tree, it usually moves to its sibling quadrants. The proposed updating procedure adaptively divides or merges the simulation area based on data movement. They show their approach outperforms two existing solutions for handling the updates – constructing the tree from scratch, and deleting a point from its current location and inserting to a new location.

2.3 Related Work on Query Processing on GPUs

In the past decade, researchers have taken advantage of the computing capabilities of GPUs to speed up computational tasks in many application domains with large datasets. In the database field, GPUs were exploited in relational operations such as aggregations [44], and join [45]. Significant speedups were reported as a result of integrating GPUs with databases in processing spatial operations [46]. Regarding the focus of this research, most of the previous work that parallelized queries on GPUs focused on parallelizing individual spatial queries instead of treating the entire workload as a whole [12, 13, 14], which are discussed in more detail in this section.

In [12] a GPU-based spatial index called STIG (Spatio-Temporal Indexing using GPUs) is presented to support spatio-temporal search queries. STIG uses a k-d tree variation to index the input dataset, which consists of intermediate nodes, and a set of leaf blocks to store the data records in consecutive memory locations. A STIG tree is constructed using a serial algorithm on the CPU, and then transferred to the GPU for query processing. Spatial query execution over STIG consists of two steps: (1) leaf nodes satisfying the search conditions are identified; and (2) all data points in the identified nodes are examined to determine the final results. STIG processes one query at a time. However, in parallelizing one tree search, it is not easy to achieve a high degree of parallelism, especially when the algorithm visits higher levels of the tree. Hence, STIG adapts a data parallel solution for this step on GPU
in which all the leaf blocks are transferred to GPU and scanned in a brute-force manner. Then, in the second step of the search, data points in the identified leaf nodes from first step are scanned in a brute-force manner to output the final matching results.

The idea introduce in [12] was later followed by Chavan et al. [13] for spatio-temporal data visualization. In this work, a variation of the quadtree called Scout was used for indexing the input data. Similar to STIG, the tree structure is built on CPU, and leaf nodes with their attached data records are transferred to GPU for query processing. The two-step query processing utilized in this work is the same as the one introduced in STIG.

We argue that the STIG [12] and Scout [13] approach, which scans all leaf nodes, is fundamentally inefficient as it literally changes the total amount of work (work efficiency) from logarithmic to linear. Although they take advantage of the thousands of GPU cores to process leaf nodes concurrently, the speedup can be quickly offset by the growing number of leaf nodes in large datasets.

In [14], a framework called GAT (GPU-accelerated Framework for Processing Trajectory Queries) is developed to support processing trajectory range queries and top-\(k\) similarity queries on GPUs. GAT is based on a quadtree-like index and cell-level trajectory representations. In GAT, to achieve a logarithmic work efficiency, the first step of query processing is done on CPU. Then, only leaf blocks identified in the first step are transferred to GPU, and the second step is parallelized on GPU. Although the overall query processing performance is improved in GAT comparing to STIG and Scout, however, for every new query, its intersecting leaf blocks has to be transferred to GPU global memory if it is already not copied there.

It is well-known that many location-based applications are busy systems with very high concurrent query arrival rate [15, 16]. Moreover, in scientific simulations such as molecular and astrophysical simulations, millions of spatial queries such as \(k\)NNs and range searches are issued at every step of the simulation [17]. Therefore, using the one-query-at-a-time
parallelism approaches in those systems is inefficient. Batched query processing, due to the effective sharing of computing resources, has been heavily studied in the database field [19, 20, 21], and large-scale web search on GPUs [22].

Moreover, another issue in the one-query-at-a-time parallelism approaches is that in the second step of the search the high performance GPU resources are not efficiently utilized. Since each query scans a list of leaf nodes to find their data records, the same data record can be accessed many times by different queries in a workload. Consequently, the program easily hits a performance ceiling due to congestion of global memory while other high performance resources are either insufficiently utilized or largely unused (e.g., shared memory).

Furthermore, an important feature that is missing from all these work is the support for tree construction and update on GPUs. In large datasets, building a tree is costly, and cost of transferring data from CPU to GPU is significant. The CPU communicates with GPU’s global memory through a PCI-E bus which is of much lower bandwidth and higher latency as compared to the global memory. For static data, it is not an essential issue as tree construction and transferring is a one-time cost. However, almost all location-based services involve dynamic data and thus tree updates. Following those approaches, the cost of handling updates will be extremely high as transmission of the entire tree is needed every time there is an update to the data. Another drawback of these approaches is that query processing cannot proceed without CPU intervention.

Last but not least, an important feature that is missing from existing work is the efficient support for datasets that cannot fit into the GPU global memory, especially when processing more than one query at a time. In [18], GAT framework is extended to support running multiple queries on GPUs. However, the number of queries that can be run simultaneously is limited due to the limited GPU global memory capacity. Therefore, in this work, the initial input query batch is divided into multiple smaller mini-batches, and just those queries in smaller mini-batches are parallelized together. A memory allocation table (MAT) is designed
to keep track of the leaf blocks existing in global memory. Therefore, before a mini-batch is
launched on GPU, first MAT is checked to see if queries’ intersecting leaf blocks are in GPU
global memory. If leaf blocks are not in global memory, they have to be copied to global
memory before query processing. In case the global memory does not have enough capacity
for new leaf blocks, following a LRU swapping strategy some leaf blocks are swapped out
from global memory to make capacity for new blocks. Therefore, each time the overhead
of transferring data from CPU to GPU is added to a mini-batch query processing, while
queries in last batches may bear longer response time. Therefore, an essential step towards
developing high performance spatial query processing in large datasets, is to overcome such
limitations.
Chapter 3: Tree Construction in G-PICS

3 We present the G-PICS (GPU-based Parallel Indexing for Concurrent Spatial data processing) framework for high performance spatial data management and concurrent query processing. G-PICS is implemented as an extensible software package that supports various types of spatial trees under different hardware (GPU) specifications. To this end, it includes highly efficient parallel algorithms for constructing a wide range of space partitioning trees based on user-input parameters. For example, users can choose to build trees with different node degrees and node partitioning method (e.g., space-driven or data driven). In addition, G-PICS takes advantage of multiple GPU cards in a system to support large datasets beyond the capacity of single GPU’s global memory. In this chapter, we discuss the encapsulated algorithms in G-PICS for tree construction.

As discussed earlier, in previous work, the tree is usually constructed in host memory and then transferred to GPU for query processing. Due to the limited CPU computing power and memory bandwidth, the cost of building the tree on host memory is high. Moreover, the overhead of transferring the tree from host to GPU is significant - with its microsecond-level latency and 10GB/s-level bandwidth [47], the PCI-E bus is the weakest link in the entire GPU computing platform. Therefore, the first step towards enabling G-PICS for efficient spatial query processing lies in efficiently building a tree data structure within the GPU. In G-PICS, we provide support for building space-driven, and data-driven partitioning trees.

\(^3\)Some part of this chapter was published in ACM proceedings of the 30th International Conference on Scientific and Statistical Database Management. Permission is included in Appendix A.
Moreover, G-PICS supports tree construction for datasets whose sizes are bigger than a single GPU’s global memory.

G-PICS supports many space partitioning trees that decomposes the space recursively to generate a fixed number of disjoint partitions each time. In these trees the fan-out of the tree is usually limited (i.e., fan-out in quadtree is four and in k-d tree is two) [31]. To construct a space partitioning tree, a user only needs to provide several parameters as inputs to G-PICS such as: (1) input data type, which determines type of data to be saved in the leaf nodes, (2) total number of input data \(N\) (3) number of space partitioning \(NP\), which determines the number of disjoint partitions at each node decomposition (fan-out), (4) maximum height \(MH\) (resolution), which determines the maximum level in the tree, and (5) maximum capacity \(MC\) (bucket size), which determines the maximum number of data items in a node – it also serves as the split threshold. Such parameters are summarized in Table 3.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>total number of input data</td>
</tr>
<tr>
<td>(NP)</td>
<td>node degree (number of disjoint partitions)</td>
</tr>
<tr>
<td>(MH)</td>
<td>maximum tree height (number of levels)</td>
</tr>
<tr>
<td>(MC)</td>
<td>maximum number of data items in a node</td>
</tr>
</tbody>
</table>

If data points in a node exceeds \(MC\), that node is partitioned into \(NP\) child nodes. The decomposition continues until there is no more node to partition, or \(MH\) of the tree is reached [48] (stopping criteria). There are two different types of nodes in those trees: nodes that got partitioned are link (internal) nodes, and others are leaf (data) nodes. G-PICS implements both space-driven and data-driven space partitioning methods to allow construction of a wide range of trees: with space-driven partitioning we can build point-region (PR) quadtree, region quadtree, PM quadtree, PMR quadtree, MX-PR quadtree, fixed grid, and point-region
k-d tree [49, 48]. On the other hand, the data-driven partitioning allow users to build spatial k-d trees and other variants of Binary Space Partitioning trees. By changing the quantity $NP$, we will get trees with different degrees (e.g., degree 4 for quad-tree).

3.1 Space-driven Partitioning in Tree Construction

In space-driven partitioning trees, the space decomposition is totally independent to the dataset. For example, a PR quadtree is a type of trie, in which each link node has at most four children. If data points in a node exceeds $MC$, that node is partitioned into four equal-sized child nodes. The decomposition continues until the stopping criteria are met. There are unique challenges in the construction of space-driven partitioning trees on GPUs. First, to achieve high efficiency, our solution requires good utilization of GPU resources, especially the large number of cores. The traditional way for such is done by parallelizing the nodes’ partitioning process level by level [38, 39]. Clearly, this approach suffers from low parallelism, especially when building top levels of the tree. Second, the total number of non-empty nodes in such trees is generally not known in advance. This is a major problem in GPUs as dynamic memory allocation on the thread level carries an extremely high overhead [45]. The easiest solution to tackle this problem, which was adapted in previous work [38, 39, 40], is to nevertheless allocate memory for empty nodes. This results in inefficient use of (global) memory, which is of limited volume on GPUs, and becomes more crucial when dealing with skewed datasets. Finally, the main design goal of G-PICS trees is to allow efficient query processing. Hence, placing data points in a leaf node in consecutive memory locations is necessary, as it allows coalesced memory access in a data parallel program.

3.1.1 Overview of G-PICS Tree Construction

To address above challenges, we propose a top-down parallel algorithm on GPUs that achieves a high level of parallelism in the tree construction process. Furthermore, our ap-
approach avoids empty node expansion, and guarantees coalesced memory access in processing the data points in a leaf node.

G-PICS handles empty nodes expansion by delaying the actual node memory allocation until the exact number of non-empty nodes in the tree is determined. In particular, in the beginning, it is assumed that the tree is a full tree according to its $MH$ – in a full quadtree all the intermediate nodes in the tree have exactly $NP$ children. Let us use a full quadtree (Figure 3.1) as an example. The maximum number of nodes in such a tree ($NP=4$) with height of $H$ can be calculated as follows:

$$\sum_{i=0}^{i=(H-1)} 4^i = (4^H - 1)/(4 - 1) = (4^H - 1)/3$$

(3.1)

Each node in a full quadtree has an ID, which is assigned based on its position in the tree. Starting from the root node with the ID equals to zero, and allocating the directions based on the children location (ranging from 0 to 3), an ID for each node is determined as follows: $Node_{id} = (Parent_{id} \times 4) + direction + 1$, in which $Parent_{id}$ is the ID of the node’s parent. node ID array holds such IDs.

### Algorithm 1: Space-driven Tree Construction Routine

**Var:** $splitNum$ (number of nodes to be partitioned) $\leftarrow 1,$
- $Split$ (array to track nodes’ split status),
- $C_{node}$ (array to keep current node for data points) $\leftarrow 0,$
- $Cur_{level}$ (current level developing in the tree) $\leftarrow 1$

1: $Split[0]$ $\leftarrow$ True
2: **while** $Cur_{level} < MH$ and $splitNum > 0$ **do**
3: $Cur_{level}$++; 
4: Tree-Partitioning on GPU;
5: update $splitNum$
6: **end while**
7: Node-Creation on GPU;
8: Point-Insertion on GPU;

The main idea behind G-PICS tree construction is a new parallel algorithm that maintains a high level of parallelism by novel workload assignment to GPU threads. Instead of binding
Algorithm 2: Tree-Partitioning on GPU

Global Var: Input (array of input data points),
      Counter (array to reflect the number of points in each node)
Local Var: t (Thread id)
      L\text{cnt} (counter value after adding a point to a node)
1: for each Input[t] in parallel do
2:   if Split[C\text{node}[Input[t]]] == True then
3:      C\text{node}[Input[t]] ← find position of Input[t] in the children of C\text{node}[Input[t]]
4:      L\text{cnt} ← atomicAdd(Counter[C\text{node}[Input[t]]], 1)
5:   if L\text{cnt} == MC+1 then
6:      Split[C\text{node}[Input[t]]] ← True
7: end if
8: end if
9: end for

a thread to a tree node which was adapted in previous work [38, 39, 40], each GPU thread is assigned to one input data point. By that, the process of locating the node to which each point belongs is parallelized. Each thread keeps the ID of such nodes and such IDs are updated at each level till a leaf node is reached. The tree construction (Algorithm 1) is done in three steps: Tree-Partitioning, Node-Creation, and Point-Insertion.

3.1.2 Tree Construction Routines

The Tree-Partitioning kernel (Algorithm 2) is launched with N threads, with each thread working on one data point. Figure 3.1 shows an example of quadtree construction in G-PICS where MC and MH are three. Some auxiliary arrays (current node, node ID, counter, and split Array) with the length equal to the maximum number of nodes in a full tree are allocated on GPU and deleted when the tree construction is completed. Starting from the root node, each thread finds the child node to which its assigned point belongs, saves the child node ID in the node ID array, and increments the counter of the number of points in the child node counter array. Since such counts (i.e., counter array) are shared among all threads, we use atomic instructions to modify the counts and maintain correctness. When
the counts are updated, if a node’s data count exceeds $MC$ and $MH$ of the tree has not been reached yet, the corresponding value in the split array will be set, meaning the node should be further partitioned. Upon finishing operations at one level (for all threads), the following information can be seen from auxiliary arrays: current node array indicates the nodes to which data points belong, node counter array reflects the number of data points in each node, and split array indicates if each node has to be partitioned. If there are nodes to be partitioned, the same kernel is launched again to develop the next level of the tree. For example, in Figure 3.1, there are two nodes – $N_2$ (with 5 points) and $N_3$ (with 7 points) – to be partitioned when second level of the tree is built. The kernel is relaunched with three new auxiliary arrays, the length of which corresponds to the number of the child nodes of only $N_2$ and $N_3$. Likewise, counter and split values of the nodes in this level are updated. This routine will continue until the stopping criteria are met. Our approach maintains a high level of parallelism by having $N$ active threads at all times.

The Node-Creation kernel (Algorithm 3) is called to create the actual non-empty nodes in the tree. Having finished the previous step, the following information is known: each point
Algorithm 3: Node-Creation on GPU

Global Var: $leaf_{datalist}$ (array to store leaf nodes data),
$Node_{ID}$ (array holding IDs of nodes)

Local Var: $t$ (Thread id)

1: for each non-empty $Node_{ID}[t]$ in parallel do 
2:     create node $Node_{ID}[t]$
3:     if $Split[Node_{ID}[t]] == False$ and $Counter[Node_{ID}[t]] > 0$ then 
4:         Allocate consecutive memory of size $Counter[Node_{ID}[t]]$ in $leaf_{datalist}$ 
5:     end if 
6: end for

has the leaf node to which it belongs, the total number of non-empty nodes in the entire tree with their types (leaf or link), and the total number of points in each leaf node. Therefore, the required information for creating the nodes (in a parallel way and without wasted space) is known. Consequently, the Node-Creation kernel is launched with as many active threads as the number of non-empty nodes, each thread creates a non-empty node. While building nodes, memory for each leaf node’s data list is allocated in consecutive memory locations. In Figure 3.1, the total number of non-empty nodes is 12 (while the full quadtree has 21 nodes).

The Point-Insertion kernel (Algorithm 4) is called to insert the input data points to the tree. Having this setup, all the points in each leaf node are saved in consecutive memory locations. The input data points in a quadtree have two dimensions (x and y). To ensure coalesced memory access in query processing, the data lists should be saved using two arrays of single-dimension values rather than using an array of structures which holds two-dimensional data points. The final quadtree structure built using the aforementioned algorithm is shown in Figure 3.2, in which each leaf node points to the beginning of its data list in the leaf nodes data list array.
Algorithm 4: Point-Insertion on GPU

Local Var: $t$ (Thread id)
1: for each $Input[t]$ in parallel do
2:   insert $Input[t]$ to $leaf_{data list}[node[Input[t]]]$
3: end for

Figure 3.2: Final quadtree built based on the data inputs in Figure 3.1

3.1.3 Cost Modeling

The total cost of tree construction can be evaluated as follows:

$$C = C_T + C_I + C_P$$

(3.2)

where $C_T$, $C_I$, and $C_P$ are the costs of Tree-Partitioning, Node-Creation, and Point-Insertion, respectively. Let $N$ be the number of data points, and $n$ the number of tree nodes, then, $C_T = O(N \log N)$, $C_I = O(n)$, and $C_P = O(N)$. Although $C_T$ is of higher complexity, it is not the main bottleneck in practice. Instead, although $C_P$ has a linear complexity, it is the dominating cost in tree construction. This is because Point-Insertion requires concurrent writes into the leaf node’s data list, and this should be done via atomic instructions to ensure consistency. Atomic instructions are known to bear a large overhead. For example, our experiments show that on average 60 to 70 percent of the tree construction time is spent on the Point-Insertion kernel.
3.2 Data-driven Partitioning in Tree Construction

Data-driven space decomposition is done based on the spatial distribution of the data to be indexed. A salient example is the spatial k-d tree, where each link node divides the space into two partitions (left and right subtrees) using a splitting hyperplane. Each link node in the tree is affiliated with one of the k-dimensions in the input dataset, with the hyperplane perpendicular to that dimension’s axis. For instance in a node with ”y” axis as splitting axis, all points in the left subtree have ”y” values smaller than those in the right subtree. The splitting plane are chosen following a round-robin fashion. To ensure the tree is balanced, it is required that the pivot value for splitting be the median value of the coordinate values in the axis used to develop the splitting plane in each node. We face the following challenges in building data-driven trees. First, similar to space-driven partitioning trees, existing work [41] parallelizes the node’s partitioning process by assigning each node to one GPU thread, and this causes low parallelism in building higher levels of a tree. Second, the median value for partitioning in each node is usually selected by sorting the data points based on the splitting axis [41]. Sorting is not an efficient way for finding the median value, as the entire sorted list is not required. Therefore, sorting does more work than necessary for finding the median value [42]. Again, placing data points belonging to a leaf node in consecutive memory locations is required. Empty node expansions is not an issue in such trees since the partitioning is done based on the input data.

3.2.1 Overview of Our Approach

To address above-mentioned challenges, we propose a top-down parallel level-by-level tree construction on GPUs that achieves high level of parallelism. In addition, we use an efficient solution to find the median value for partitioning in each node. Last but not least,
our approach guarantees coalesced memory access by storing the data points belonging to a leaf node in consecutive memory locations.

The key innovation in our data-driven tree partitioning is a parallel approach for finding the median value of a list of values without sorting. In order to find a median value in an unsorted list, data points in that list are organized into multiple continuous buckets using a range-based histogramming approach. The histogram is designed in a way to divide the input range of data lists into $H$ histogram buckets. Each input data point ($Input[i]$) is assigned to one histogram bucket based on its location using a linear projection as follows:

$$
\left\lfloor \frac{(H - 1)}{(max - min)}(Input[i] - min) \right\rfloor
$$

where $max$ and $min$ are the maximum and minimum value in the input list, respectively. Each bucket has a counter showing the number of points belonging to that bucket. By assigning the points to histogram buckets, the bucket that contains the median value can be determined based on the buckets’ counter values. Thus, only that bucket will be sorted in order to find the median value. This approach eliminates extra candidates for sorting. In order to assign points to histogram buckets, each data point in a list is assigned to one GPU thread to find the bucket to which it belongs. Obviously, this approach achieves a high level of parallelism.

The main bottleneck lies in updating the counter values for each bucket by concurrent threads. Since, such protected memory locations is not cached in CUDA, cannot be accessed in a parallel manner. To avoid race condition, atomic instructions have to be used for such updates yet atomic instructions are executed in a sequential manner. For that, we adopt an output privatization technique for outputting the histogram buckets’ counters. Using this technique, private copies of histogram buckets’ counters are stored to be accessed by a subset of the threads in the on-chip cache of GPUs. Considering the fact that read-only data cache cannot be overwritten during the lifetime of the kernel, cannot be used for
Private histogram for block 0
Private histogram for block 1
Private histogram for block 2
Private histogram for block B-2
Private histogram for block B-1
Final histogram in global memory

Figure 3.3: Combining private histograms in all blocks to output the final histogram

private outputting. Consequently, private copies of bucket counters are stored in on-chip cache called *shared memory* to be accessed by a subset of the threads. Using such technique, calculating the histogram buckets’ counters is done in two steps: (1) Each thread that finds the histogram buckets to which it belongs, increases the related private histogram bucket’s counter via the atomic write operation. Even though atomic operation is still used, the high bandwidth of shared memory minimized the overhead. (2) when all threads find their histogram bucket, the private copies of histogram buckets’ counters are combined together to generate the final histogram buckets’ counter (Figure 3.3). The final output is generated using a parallel *reduction* algorithm presented in [50].

After generating the final histogram output, the bucket that has the median value can be identified by reduction of counter values – median should divide the input list into two lists with equal size. Consequently, just the points in that bucket need to be sorted to find the median value. We implement two strategies for such sorting. If the number of points in that bucket are small enough to fit into shared memory, the sorting is done in shared
Algorithm 5: CUDA sorting in shared memory

**Global Var:** $A$ (array of input data points),
$N$ (number of points in $A$),
$B$ (block size),
$SHCache$ (input data points in shared memory)

**Local Var:** $t$ (Thread id),
$count$ (local counter) ← 0

1: $j ← t$
2: while $j < B$ do
3: $SHCache[j] = A[j];$
4: $j ← j + B;$
5: end while
6: syncthreads
7: for $i=1$ to $N$ do
8: if $SHCache[t] > SHCache[i]$ then
9: $count$ ++;
10: end if
11: end for
12: syncthreads
13: $A[count] = SHCache[t];$

memory using an efficient in-place fast shared memory sorting (Algorithm 5) [51]. If the size of the input list is larger than shared memory, the sorting can be done using fast Radix sort libraries available in CUDA libraries such as Thrust or CUDPP. After sorting the points in the determined bucket, the median value in the input list is determined. Figure 3.4 shows an example of finding the median value using the aforementioned histogramming approach.

3.2.2 Tree Construction Routines

G-PICS k-d tree construction follows a level-by-level approach on GPUs to maximize the level of parallelism. At higher levels of the tree, every node in the developing level is assigned to a GPU thread, which launches another kernel using CUDA dynamic parallelism to perform the node partitioning. To partition a node, the median value in the developing dimension’s axis is determined following the approach mentioned earlier. Having determined the median
value, points can be organized into the sub-list (left or right) to which they belong. However, moving data points into sub-lists is not in-place, and consists a lot of atomic operations in global memory to find the next available pointer in each sub-list. To tackle such issue, in each GPU block, two private counters are defined in shared memory. Using the private counters, the total number of points belonging to each sub-list in each block is determined. Then, the private counters are used to increment the global counters in global memory. After moving data points to the sub-lists, nodes can be created. If the stopping criteria are met, a node is a leaf node; otherwise it is a link node and should get split in building the next level of the tree. On the other hand, in the lower levels of the tree, the same approach is followed, however, each node that has to get partitioned is assigned to one GPU block for partitioning.

### 3.3 Tree Construction in Multiple GPUs

So far we assume that the entire tree plus all the data can fit into one GPU’s global memory. However, there are some datasets that are larger than the capacity of a single GPU’s global memory. For datasets with size beyond the size of one GPU’s global memory, multiple GPUs should be used. However, in Multi-GPU implementations, the main challenge is to minimize data synchronization and communication among GPUs. For example, in order to determine whether a node should be split, summation of the auxiliary array values from
all GPUs is required. Traditional wisdom adopts a CPU master approach [52] by using the CPU to perform such synchronization and communication. Specifically, all GPUs send their local copies of run-time parameters to the host CPU, and the CPU conducts a parallel reduction to get the global value and send the results back to the GPUs. However, this approach requires a lot of communications to transfer the parameters between all GPUs and CPU through PCI-E bus, which has low bandwidth and high latency. In addition, due to limited parallel computing resources, the reduction operation is much slower on the CPU. To overcome such limitations, we take a GPU master approach, in which one GPU becomes the central control, and all other GPUs send their local copies to the master GPU to conduct parallel reduction.

In addition, to minimize the communication cost, data transfer and in-core computation should be overlapped whenever it is possible. For that purpose, we adapt a GPU mechanism called CUDA streams to allow concurrent kernel function execution. For example Figure 3.5 shows transferring counter and split arrays in Figure 3.1 via CPU master to four other GPUs comparing to GPU master approach with three CUDA streams, each containing two kernels on master GPU. Specifically, we simultaneously launch several CUDA streams containing GPU kernel calls via several OpenMP threads, and the data transferring is done using asynchronous direct GPU-To-GPU data transfer (Figure 3.6). In all the algorithms in our work, whenever it is possible CUDA streams and asynchronous direct GPU-To-GPU data transfer are used to transfer data among GPUs.

3.3.1 Tree Construction Routines

Tree construction is still done in three steps. Since the input dataset size is bigger than one GPU’s global memory, the input dataset should be split into multiple sets based on the number of available GPUs in a Multi-GPU cluster, and each set is assigned to one GPU.
Figure 3.5: Transferring Counter and Split arrays in Figure 3 via CPU master to four other GPUs comparing to GPU master approach with three CUDA streams, each containing two kernels on master GPU.

Figure 3.6: An example of data transfer between GPUs.

Each GPU in a work-set can work independently on finding enclosing node for points in its partition on the tree level being developed (Algorithm 2). Threads in each GPU can work independently to find a node to which each point belongs to that level; However, in order to determine whether the tree construction needs to proceed on building the next level of the tree, the nodes’ data point counters should be updated based on the total sum of the counter values in all GPUs for that level. Consequently, the necessity for building the next level of the tree can be determined. To this end, each GPU keeps a local copy of the node ID array, counter array, and split array. Then, after building each level, the corresponding global counter value for each node in that level is updated with sum of the values from each GPU using the GPU master approach (Figure 3.7).

Figure 3.8 shows constructing the tree using four GPUs for input data points in Figure 3.1. Therefore, the input dataset is partitioned into four partitions and each partition is assigned to one GPU in the work-set. In the first run, each thread in each GPU, finds the
Figure 3.7: Constructing the second level of the quadtree in Figure 3.1 using four GPUs

node to which its assigned data point belongs, and updates the local counter array for that node. Then, local counter arrays are transferred to the master GPU for parallel reduction to the global counter array. Having updated the global split array there are two nodes ($N_2$ and $N_3$) that need to get partitioned. Therefore, the global split array values for this level are transferred to other GPUs for building the next level of the tree. Then, the tree partitioning kernels are relaunched again on GPUs to find the position of the points in the child nodes of the overflow nodes in the next level of the tree. Parallel reduction of the local counter arrays to the global one in this level, shows that the stopping criteria are met.

The node creation kernel is launched on the master GPU (Algorithm 3). Since in such design all leaf node’s data list cannot be stored in one GPU card, leaf node’s data lists are allocated in consecutive memory locations on the available GPUs. Each leaf node on the master GPU keeps track of the physical memory location of its data list.

To insert the input data points to the data lists, each data point should be located in the same GPU global memory as its data list. Data points in each GPU that do not meet this condition, are grouped together based on the GPU to which they belong, and are copied to their corresponding GPU – using multiple CUDA streams and direct GPU-To-GPU data transfer in parallel. Then, the Point-Insertion kernel (Algorithm 4) is launched on each GPU to insert data points to their data lists. In the final quadtree structure, each leaf node points to the beginning of its data list in the leaf nodes data list array stored on one GPU global
Figure 3.8: Full quadtree construction for data input in Figure 3.1 using four GPUs

Leaf Node

Link Node

Root Node

Leaf nodes data list stored in multiple GPUs' global memory

Tree structure stored in master GPU (GPU 0) global memory

Figure 3.9: Final quadtree built based on the data inputs in Figure 3.8
memory in a cluster. The final quadtree structure built using the aforementioned algorithm for input dataset in Figure 3.8 is shown in Figure 3.9.

3.4 Experimental Evaluations

In this section, we present empirical evaluation of G-PICS tree construction performance. For that, we implemented a CUDA version of tree construction algorithms within G-PICS. All the experiments in this dissertation are conducted on a workstation running Linux (Ubuntu 16.04 LTS) with an Intel Core i9-7920X 2.90GHz CPU with 12 cores, 64GB of DDR4 3200 MHz memory, equipped with four Nvidia Tesla P100 GPU cards with 16GB global memory in each card. All implementations in G-PICS and evaluation benchmarks are highly optimized in terms of efficient use of registers, choosing the best block size in each GPU kernel. Such optimizations are done according to our previous work in GPU kernel modeling and optimization [53]. All the experiments are conducted over a real dataset [54] generated from a large-scale molecular simulation study of a lipid bi-layer system. \(^4\) In the all the experimental evaluations, we report and analyze the absolute and relative total processing time (i.e., speedup) used for processing supported algorithms within G-PICS over various baseline programs.

3.4.1 G-PICS Performance in a Single GPU Environment

In evaluation tree construction algorithms performance, only the tree construction time is measured and compared. The parallel quadtree construction codes introduced in [38] and [39] are not publicly available. We implemented the algorithms following their descriptions. However, such algorithms showed very poor performance comparing to ours (G-PICS achieves more than 1000X speedup). Due to very low level of parallelism and inefficient mem-

\(^4\)The data contains coordinates of 268,000 atoms recorded over 100,000 time instances. We superimpose the coordinates of different time instances to generate data of an arbitrary (large) size for experimental purposes.
Figure 3.10: G-PICS tree construction time, and speedup over comparative algorithms

ory use, the above academic code can hardly represent the state-of-the-art. Consequently, to have a more meaningful evaluation, we compare G-PICS tree construction with the parallel tree construction developed by Nvidia [40]. G-PICS kdtree construction performance is evaluated by comparing with parallel kdtree construction based on sorting all data points using the CUDA Thrust library [41].

Figure 3.10 shows the G-PICS quadtree and kdtree construction time, and speedup over comparative programs. In the following experiments, $MH$ is set to 1024 and $MH$ is set to 12. As shown, G-PICS quadtree clearly outperforms the Nvidia ([40]) code (up to 53X) in all cases. By increasing the number of input data points, G-PICS speedup increases, and it remains constant at very large input datasets. While building the tree in G-PICS, if $MH$ is reached, a leaf node could accommodate more points than $MC$. Under the same situation, the Nvidia code crashes. Moreover, G-PICS does not materialize the empty nodes, while Nvidia suffers from empty node expansion. Likewise, G-PICS kdtree construction beats the tree construction using sorting (up to 13X) in all cases.
3.4.2 G-PICS Performance in a Multi-GPU Environment

We evaluate our multi-GPU algorithms with a focus on performance scalability. For that purpose, we use a dataset that can fit into one GPU’s global memory but distribute it to multiple GPUs. The experiments are conducted by using one to four GPUs. In the following experiments, \( MH \) is set to 1024 and \( MH \) is set to 14. As discussed earlier, we can build the tree in G-PICS via a GPU master (\( GM \)) or a CPU master (\( CM \)) approach. As expected, the \( GM \) approach shows much better performance than \( CM \) (Figure 3.11, left). \( GM \) shows at least a 2.5X speedup over \( CM \) under all numbers of GPUs. Figure 3.11 (right) shows the performance of \( GM \) over the Nvidia single GPU tree construction code [40]. With increasing sizes of input data, the performance speedup over Nvidia becomes more remarkable. In addition, the performance of G-PICS increases with more GPUs used, indicating good scalability.
Chapter 4: G-PICS Query Processing

As mentioned earlier, G-PICS framework is designed for concurrent query processing. Query processing approach in G-PICS solves the limitations in existing work, in which instead of parallelizing a single tree-search operation, our strategy is to parallelize multiple queries running concurrently. Therefore, G-PICS query processing bears logarithmic work efficiency for each query yet overcomes the problem of low parallelism. G-PICS encapsulates all the key components for efficient parallel query processing within GPU with little CPU intervention. As mentioned in Chapter 3, G-PICS encapsulates highly efficient parallel algorithms for constructing a wide range of space partitioning trees. G-PICS provides APIs for processing major spatial queries including: (1) spatial point search, which retrieves data associated with a point in the data domain, (2) range search, which finds a set of data points intersect with a query shape object, (3) within-distance search, which retrieves objects within a specific distance from the search query, and (4) k-nearest neighbors, which retrieves k closest objects to a query point. Such APIs enable efficient development of more complex spatial data retrieval algorithms and applications. G-PICS query processing algorithm is independent from the index structure (e.g., space-driven or data-driven). In addition, G-PICS takes advantage of multiple GPU cards in a system to support large datasets beyond the capacity of single GPU’s global memory. In this chapter, we elaborate on query processing algorithms in G-PICS.

Some part of this chapter was published in ACM proceedings of the 30th International Conference on Scientific and Statistical Database Management. Permission is included in Appendix A.
4.1 Query Processing Algorithms in G-PICS

A typical spatial query is processed in two steps: (1) identification of leaf nodes satisfying the search conditions; and (2) examining the identified nodes to determine the final output data points. G-PICS is designed to process multiple spatial queries running concurrently. To this end, in the first step, using the traditional tree search is necessary to achieve logarithmic work efficiency. Recall that in the second step, reading the data records from GPU global memory is the main bottleneck, as the same data record can be accessed many times by different queries in a workload. To tackle these issues, G-PICS introduces a push-based paradigm for the second step of query processing. Specifically, a query list is attached to each leaf node for saving the list of queries intersecting that leaf node for processing. In the first step of query processing, queries intersecting with a leaf node are registered in the query list attached to that leaf node. In the second step, queries in each list are processed together to minimize accesses to global memory, and take advantage of the other available low-latency memories on GPUs. Two GPU kernels are designed to perform query processing: Leaf-Finding, and Leaf-List-Processing.

4.1.1 Step I - Query Registering

For each leaf node, we maintain a query list, which contains IDs of all queries whose outputting data can be found in that leaf node. In the Leaf-Finding kernel, each thread takes one query from QL, and finds the leaf node(s) that intersect with the query search key value or range. Then, registers that query to its intersecting leaf nodes. Figure 4.1 shows an example of such queries registering.
Figure 4.1: An example of registering range search queries into the query lists of leaf nodes

Input query list (QL)

Query list attached to a leaf node

...
4.1.2 Step II - Leaf Node Processing

To process the registered queries in the query list of the leaf nodes, the Leaf-List-Processing kernel is launched with as many GPU blocks (i.e., a group of threads) as the number of leaf nodes. Then, each registered query in the query list is assigned to one thread in that block. In order to output the results, all the queries in a leaf query list have to read the data records in that leaf node and, based on their query types, perform the required computation. Therefore, in each GPU block, if the number of registered queries is greater than one, all the data points belonging to the leaf node assigned to that GPU block are copied from global memory to shared memory. Shared memory is much faster than global memory - its access latency is about 28 clock cycles (versus global memory’s 350 cycles) [55]. The copying from global memory to shared memory is not only parallelized, but also coalesced because points in each leaf node are saved in consecutive memory locations. Using this strategy, the number of accesses to each leaf node data lists in global memory are reduced to one. This is in sharp contrast to the traditional approach that retrieves each leaf node once for each relevant query. Having copied all the points in each leaf node data list to shared memory, each active thread in that block takes one query from the query list attached to its corresponding leaf, calculates the Euclidean distance between that query point and all the points in that leaf node (located in shared memory), and outputs those that satisfy the search criteria. This step is shown in Algorithm 6.

4.1.3 Query-Specific Implementation Details

The point search, range search, and within-distance search are implemented following the 2-step approach in a straightforward way: in the first step, a within-distance search retrieves the leaf nodes falling within a specific distance from the search key; a range search retrieves the leaf nodes intersecting with the query range; a point search retrieves the leaf
Algorithm 6: Leaf-List-Processing on GPU

Local Var:  
- $b$ (Block id),
- $t$ (Thread id),
- $M$ (number of points in leaf[$b$]),
- $lqL$ (query list for leaf[$b$]),
- $sL$ (leaf[$b$] data list)

1: if $lqL > 1$ then
2:   $sL ←$ load leaf$\_datalist$[leaf[$b$]] to shared memory in parallel
3: else
4:   $sL ←$ leaf$\_datalist$[leaf[$b$]] from global memory
5: end if
6: for each $lqL[t]$ in parallel do
7:   for $i = 1$ to $M$ do
8:     $d ←$ computeSearchFunction($lqL[j], sL[i]$)
9:     if $d$ meets the search condition then
10:        Add $sL[i]$ to the Output list of $lqL[j]$
11:   end if
12: end for
13: end for

node that contains the query search key. Then in the second step of the search, data lists in the identified leaf nodes from the first step are examined to output the final results. The kNNs in G-PICS is treated as a within-distance search followed by a $k$-closest selection from the within-distance search result set. The within-distance search is initialized with a radius based on the input distribution and $k$. If the number of output items for a query is less than $k$, the within-distance search will be performed again with a larger radius.

4.1.4 Outputting Results

A special challenge in GPU computing is that, in many applications, the output size is unknown when the GPU kernel is launched. Examples of such in G-PICS are the number of output results in a typical window range, or within-distance query. In CUDA, memory allocation with static size is preferred - in-thread dynamic memory allocation is possible but carries a huge performance penalty [45]. A typical solution is to run the same kernel twice:
in the first round, output size is determined. In the second run, the same algorithm will be run again and output written to the memory allocated according to the size found in the first round. In G-PICS, we utilize an efficient solution introduced in [56], which allows our algorithms to compute and output the results in the same round for those categories of queries that their output size is unknown in advance using a buffer management mechanism. In this design, an output buffer pool with a determined size is allocated. The allocated memory is divided into pages of a particular size. In order to record the location of the first available page in the buffer pool, a global pointer (GP) is kept. Each thread gets one page from the buffer pool and outputs its results to that page. It also keeps track of its own local pointer to the next empty slot within that page. Once a thread has filled a page completely and has more results, it will get a new page from the buffer pool by increasing the GP using the GPU atomic add operation. Using this solution, conflicts among threads is minimized because the GP is updated only when a page is completely filled. Figure 4.2 shows an example of direct output buffer for GPU threads, showing Thread 3 acquiring a new page as its output buffer.

4.2 G-PICS Query Processing in Multiple GPUs

So far we assumed that the datasets, query lists and all associated data structures can fit into a single GPU global memory. However, as mentioned earlier, G-PICS supports datasets
that are too large to fit into one GPU global memory. In Chapter 3, we mentioned that the tree structure for such large datasets is stored on one GPU’s global memory (master GPU), and the leaf nodes’ data are stored in multiple GPUs. Therefore, query lists are added to the tree structure in the master GPU. Since all data structures for the first step are stored in one GPU memory, the first step in query processing (Query Registering) is done on the master GPU. Since the tree structure is stored on the master GPU, the first step of query processing is done on the master GPU. Then, the second step is parallelized using multiple GPUs. Query processing for such datasets is done in three steps as follows:

4.2.1 Step I - Query Registering

The Leaf-Finding kernel is done on the master GPU using the same approach mentioned in single GPU approach.

4.2.2 Step II - Copying Query Lists

After registering queries into the query lists attached to the leaf nodes in the master GPU, in order to retrieve the final results, the query lists should be transferred to the corresponding GPUs where their intersecting leaf nodes’ data lists are stored. The query list transferring is again done using multiple GPU streams and direct GPU-To-GPU transfer.

4.2.3 Step III - Leaf Node Processing

Each leaf node on each GPU has a list of queries that need to read and process its data list to retrieve their final results. All the GPUs can run this kernel simultaneously in parallel without any dependency. To process the registered queries in the query list of the leaf nodes on each GPU, the Leaf-List-Processing kernel (Algorithm 6) is launched with as many GPU blocks as the number of leaf nodes stored on that GPU to output the query results.
4.3 Experimental Evaluations

In this section, we present empirical evaluation of G-PICS query processing performance. For that, we implemented a CUDA version of G-PICS including algorithms for processing the following search queries: window-based range, within-distance, kNNs, and point search. All implementations in G-PICS and evaluation benchmarks are highly optimized in terms of efficient use of registers, choosing the best block size in each GPU kernel. In the following text, we report and analyze the absolute and relative total processing time (i.e., speedup) used for processing supported algorithms within G-PICS over various baseline programs.

4.3.1 G-PICS Performance in a Single GPU Environment

STIG and GAT are not designed to support concurrent query processing for large number of queries. Queries have to be put into a queue and processed one after another. Therefore, for processing multiple queries using those solutions, input queries should be stored in a list. Then, while the list is not empty, one query is extracted from the list and parallelized on GPU. Consequently, the total processing time for each query is the sum of its processing time on GPU, and its waiting time in the list. For each query, the longer its index distance from the beginning of the list, the longer its total processing time. Consequently, the performance speedup achieves by parallelism is quickly offset by increasing the number of queries. For example, for processing 1000 queries using this approach G-PICS gains a performance speedup of 5000X and 180X over STIG and GAT, respectively. Such speedup shows a dramatic upward trend by increasing the number of input queries – with 10k queries, G-PICS speedup reaches 15000X and 500X over STIG and GAT, respectively. In addition, in GAT, leaf node blocks are copied to global memory, when they are accessed for the first time, which increases the processing time – such time is not considered in the above speedup.
comparison. STIG and GAT are not designed to support concurrent query processing for large number of queries.

For a meaningful evaluation, we compare G-PICS with the following three baseline algorithms: (1) a parallel CPU algorithm (P-CPU), which is implemented based on OpenMP. Note that P-CPU is highly optimized, and performs a parallel traditional tree search in Step I of query processing to bear the logarithmic work efficiency. Additional techniques for improving the P-CPU performance using OpenMP are applied including: choosing the best thread affinity for the thread scheduler, best thread scheduling mode, and best number of active threads; (2) M-STIG and (3) M-GAT, which are task parallel GPU programs for processing multiple queries at a time developed following the descriptions in STIG [12] and GAT [18]. Specifically, in Step I of query processing, M-GAT performs a parallel tree search on the CPU. Therefore, M-GAT performs the same as P-CPU in Step I for processing multiple queries concurrently. Then, the list of intersecting leaf nodes for queries are transferred to GPU. In Step II of the search, M-GAT parallelizes each query separately on GPUs. Hence, M-GAT performs the same as M-STIG in Step II for processing multiple queries concurrently. As the query processing is done in two steps, the total running time and that in each step (Step I and II) are compared. Note that the total time is end-to-end query processing time. This includes the time to ship query list into and the query results out of the GPU. This allows a fair comparison to P-CPU, which does not have such CPU-to-GPU transmission costs. The time takes to transfer the list of intersecting leaf nodes from CPU to GPU is added to M-GAT total processing time, however, we did not consider the cost of transferring leaf data blocks from CPU to GPU in M-GAT.

The experiments in this section are run under the input dataset size of 16.5 million points, which are indexed by a quadtree with $MC$ and $MH$ equal to 1024 and 14, respectively. The query processing performance is evaluated under different numbers of concurrent queries (up to 4 Million). Since the output size of a typical range search and within-distance search is
unknown in advance, we use the buffer pool solution discussed in Section 4 for outputting the query results in G-PICS, M-GAT and M-STIG.

The absolute processing time of queries in G-PICS is shown in Figure 4.3. Since kNNs in G-PICS is done using within-distance searches followed by $k$ closest selections in the within-distance search results, the processing time of Step II in kNNs is the sum of the processing time to perform both of these operations. We implement the same selection kernel for finding kNNs in G-PICS and M-STIG. Here, we did the experiments for kNNs with a large $k$ for each query (where $k$ is greater than 100). For all 4 types of queries in both Steps, the processing time increases linearly with the number of concurrent queries. However, the time for running Step II dwarfs that for Step I therefore contributes more to the total processing time. Figure 4.4 shows the performance speedup of G-PICS over P-CPU, M-GAT, and M-STIG in processing different search queries. The logarithmic tree search in Step I noticeably outperforms the brute-force leaf search under all circumstances (more than 100X average speedup). The performance speedup over M-GAT and P-CPU in Step I is less remarkable (up to 20X) comparing to those over M-STIG. Such performance boost over M-STIG is certainly in conformity with the advantage of logarithmic tree search. Since there is less computation involved in processing Step I of the point search queries (each query just
Figure 4.4: Speedup ((a): step I only; (b): step II only; (c): total time) of G-PICS over M-STIG, M-GAT, and P-CPU in processing 1,000 to 4,000,000 concurrent queries
intersects with one leaf node), the speedup reaches a very high level (up to 300X). Generally, Step II speedup of G-PICS is not as high as that in Step I (up to 46X). It starts with a small number under low concurrency, then increases linearly with the number of input queries, and levels off afterwards. Such an increase of speedup is the result of using shared memory for holding data – the savings caused by caching are higher with more queries. When the shared memory bandwidth is fully utilized, the speedup levels off. As Step II dominates, the trends of speedup in total running time are similar to those found in Step II – even by considering the cost of transferring outputs back to CPU, G-PICS outperforms P-CPU remarkably.

4.3.1.1 Performance Under Low Concurrency

Although G-PICS is designed for query processing systems with high query arrival rate, we run a set of experiments to evaluate the performance of G-PICS under low concurrency. Such results are shown in Figure 4.5.

In Step I, the logarithmic tree search speedup in G-PICS over the brute-force leaf search in M-STIG is more remarkable under lower concurrency comparing to higher concurrency. Global accesses to the GPU global memory are cached in L2 cache attached to the GPU global memory. By increasing the number of queries, the hit rate for reading leaf nodes data through L2 cache increases. Therefore, this results in a slight improvement in brute-force leaf search accessing time in Step I. Considering Step II performance in G-PICS, if the number of registered queries in a leaf node is small, the performance speedup achieved using accessing leaf nodes’ data lists through shared memory is not considerable – note that if there is one registered query in a leaf node, reading leaf nodes’ data lists is done through global memory. Therefore, G-PICS speedup in Step II over M-STIG and M-GAT under low concurrency fluctuates based on query distribution in leaf nodes. However, since G-PICS performance speedup over M-STIG in Step I is highly considerable, G-PICS overall speedup outperforms M-STIG under all circumstances. On the other hand, having small number of threads, we
Figure 4.5: Speedup in query processing time under low concurrency ((a): step I only; (b): step II only; (c): total time) of G-PICS over M-STIG, M-GAT, and P-CPU
are not able to utilize the GPU resources. Therefore, G-PICS Step I speedup over P-CPU and M-GAT is small. For the same reason, in Step II under very low concurrency, P-CPU performs better than G-PICS. However, by increasing the number of registered queries in each leaf node queries in Step II, the performance speedup of G-PICS increases.

4.3.2 G-PICS Performance in a Multi-GPU Environment

We evaluate our multi-GPU algorithms with a focus on performance scalability. For that purpose, we use a dataset that can fit into one GPU’s global memory but distribute it to multiple GPUs. The experiments are conducted by using one to four GPUs.
Results related to the performance of spatial query processing algorithms under multiple GPUs are shown in Figure 4.6, in which we plot the speedup of the algorithm running on 2-4 GPUs over that of a single GPU. In such experiments, we gain performance by distributing the computational workload to more GPUs while at the same time introducing data transmission overhead. A speedup higher than 1X is achieved when the gain overshadows the overhead. When two GPUs are used, the speedup is close to 1.5X for within-distance, kNNs, and range search queries while it goes slightly below 1X for point search. This is because the demand for in-core computation for the point search is low thus the communication cost dominates. However, when three or four GPUs are used, the performance increases significantly for all four query types. With four GPUs, the speedup reached 3.4X for within-distance, 3.5X for kNNs, 3.3X for range search, showing an (almost) linear scalability. For reasons mentioned above, the largest speedup for point search is only about 2X. In all query types, the speedup increases slightly or stays the same with the increase of the number of input queries.

4.3.2.1 Scalability to Larger Data

We also demonstrate that by using multiple GPU cards, we are able to handle much larger datasets. Figure 4.7 shows such experiments for processing two million input queries in G-PICS using multiple GPU cards.

Since in processing point search queries, each query just intersects with one leaf node, and has a single output, we are able to process much larger datasets comparing to other queries. As mentioned in Section 4, query processing in G-PICS using multiple GPU cards is done in three steps: (1) query registering on the master GPU, (2) copying query lists from the master GPU to other GPUs, and (3) leaf nodes’ data lists processing and outputting the results on each GPU. Query registering is done on the master GPU using logarithmic tree search; therefore, by increasing the input dataset size, this cost increases in a logarithmic
Figure 4.7: G-PICS multi-GPU query processing time for two million input queries

manner. Since the number of input queries is fixed, the cost of copying the query lists from the master GPU to other GPUs using multiple CUDA streams and direct GPU-To-GPU transfer does not change by increasing the input data size. However, the fewer the number of GPU cards, the higher the cost of transferring those lists to each GPU. This is because each GPU holds more leaf nodes’ data lists. Considering the cost of leaf nodes’ data lists processing, in point search queries, this cost does not increase a lot by increasing the input dataset – each query just intersects with one leaf node. Therefore, query processing time in processing point search queries follows a logarithmic increase by increasing the input dataset size. However, in other query processing types (range search, within-distance search, and kNNs), the same query range is used for all datasets. Therefore, increasing the input dataset size leads to increase in the volume of data to be processed, and accordingly increase in the output size for each query. Consequently, query processing time in those queries follows a linear increase by increasing the input dataset size. On the other hand, by adding more GPU cards, query processing time decreases linearly. Therefore, by adding more GPU cards, not only processing larger datasets is possible, but also linear speedup in terms of query processing performance is achieved. Thus, we can conclude that query processing algorithms within G-PICS scale very well across multiple GPU cards.
Chapter 5: Tree Updates in G-PICS

Support for dynamic data inputs is an important feature that is missing from existing GPU-based spatial tree work. However, G-PICS provides an efficient parallel update algorithm to support dynamic datasets on GPUs. Data movement may change the tree structure - it can be viewed as a deletion followed by an insertion. Both operations are costly because dynamic memory allocation at kernel runtime carries an extremely high overhead on GPUs, especially when there are group movements. At the end of each move, a data point can either stay in the same leaf node, or move into another node. After processing all updates, the number of points in some leaf nodes may exceed $MC$. Consequently, if $MH$ has not reached, the nodes should be partitioned, and points in them moved to their children. Alternatively, neighboring leaf nodes could lose data points and should be merged together. Moreover, in space-driven partitioning G-PICS trees, empty nodes are not materialized; therefore, there may be cases that points move into an empty node that needs to be materialized on-the-fly.

Figure 5.1 shows an example of these scenarios in a quadtree: Figure 5.1(a) shows the tree structure before data movement. The points that moved out from their last-known leaf nodes are shown in red color. By the movements in this tree, the number of points in $N_1$ exceeds $MC$. Therefore, $N_1$ should get partitioned, and points in that node be moved to its children. On the other hand, the total number of points in $N_9, N_{10}, N_{11},$ and $N_{12}$ is less than $MC$. Therefore, these four sibling nodes should be merged together, and points in them moved to their parent ($N_2$). Since $P_{10}$ moved to an empty node, a node should be

---

6 Some part of this chapter was published in ACM proceedings of the 30th International Conference on Scientific and Statistical Database Management. Permission is included in Appendix A.
materialized in that region \((N_{16})\) to hold \(P_{10}\). The final tree structure after all the updates is shown in Figure 5.1(b).

The easiest solution is to build the tree from scratch. However, this solution is inefficient in that all the data will be processed regardless of the amount of data movement. Therefore, our goal is to design an update procedure with running time \(proportional\) to the intensity of data movement.

## 5.1 Bulk Updates in G-PICS

We design a bottom-up parallel algorithm on GPUs to bulk update the tree under both data-driven and space-driven partitioning mechanisms. This, again, reflects our strategy of concurrent processing of queries except now a query is an update. At first, the new position of all the input data points are checked in parallel to see if they moved out from their last-known leaf node. If there is at least one movement, the tree structure should be updated accordingly. Several GPU kernels are designed to update the tree in parallel on GPU as follows: Movement-Check, Leaf-Finding, Node-Partitioning, Node-Merging, and Leaf-Lists-Update. Two counters (\(Moved\_in\_Counter\) and \(Moved\_out\_Counter\)), and two lists \((Moved\_in\_List\) and \(Moved\_out\_List\)) are added to each leaf node to keep track of
points moved-in or moved-out during the update procedure. The work-flow of the entire tree update procedure is shown in Figure 5.2.

The Movement-Check kernel checks if each data point has moved out from its last-known leaf node. This kernel is launched with \( N \) threads. Each thread takes a point, and checks if the point has moved out from its last-known leaf node. In case of a movement, the corresponding thread adds the point to the list of moved data points (\( Moved\_List \)), and update the \( Moved\_out \) counters and lists associated with the last-known leaf node.

The Leaf-Finding kernel is called if \( Moved\_List \) is not empty. This kernel is launched with as many threads as the number of points in \( Moved\_List \). Each thread finds the new leaf node its assigned point moved in, and updates the \( Moved\_in \) counters and lists associated with that leaf node. If updating the number of points in a leaf node makes it qualified for partitioning (the total number of points in that leaf node exceeds \( MC \) and \( MH \) has not been reached), that node is added to the \( node\_split \) list. In case of moving to empty nodes in space-driven partitioning trees, new nodes are first created in parallel, and afterwards points are added to the newly-created nodes in parallel.

With a non-empty \( node\_split \) list, the Node-Partitioning kernel is called to split the nodes in parallel. There are two groups of points that may belong to such nodes: points that previously belonged to those nodes and did not move out, and points from \( Moved\_List \) that moved to those nodes. We call them \( Candidate\_Points \). To maximize the efficiency in moving points from the partitioned nodes to the data list of the newly-built child nodes, only \( Candidate\_Points \) are checked. While updating the counters of child nodes, if a node becomes qualified for partitioning, that node is added to the \( node\_split \) list. Then, the node partitioning process will repeat for any new nodes that become qualified for splitting during this step.

On the other hand, while some nodes are to be partitioned, there may be other nodes that have to be merged. Except the leaf nodes that get partitioned, other leaf nodes have the
Figure 5.2: Tree update procedure in G-PICS
potential of getting merged. The Node-Merging kernel considers those leaf nodes by checking
the sum of the counters of sibling leaves. If the total number of points in those nodes become
less than or equal to $MC$, they are added to the $node_merge$ list. Siblings in this list are
merged together, and their data points moved to their parent, which becomes a new leaf.
The algorithm of moving points in these nodes is similar to the one in Node-Partitioning
kernel. However, the only difference is that in this part the points should be moved to the
parent of their last-known leaf node.

Having finished all these steps, each point has the final leaf node to which it belongs, and
each leaf node has the total number of points in its data point list. To update the leaf nodes’
data lists, a data point can be reinserted form scratch using the same procedure mentioned
in the tree construction algorithm. This approach is adapted in our previous work [57].
However, this approach does more work than necessary, especially under low data movement
– any data movement would cause the entire data list rebuilt, which is the dominant cost. To
increase efficiency, a memory allocation method for the data lists with the aforementioned
page-based mechanism mentioned in Chapter 4 is designed as follows: first the leaf nodes
that are affected by update procedure are identified. Then, only the data lists in those nodes
are updated by calling Leaf-Lists-Update kernel, which assigns each leaf node to a GPU
block. If in a leaf node the number of moved-in points is greater than moved-out points,
then a new page is allocated to store the points moved into that node.

5.1.1 Cost Modeling

The total cost of update procedure can be expressed as:

$$C = C_M + C_L + C_V + C_D$$

(5.1)

where $C_M$ is the cost of running Movement-Check, $C_L$ the cost of Leaf-Finding, $C_V$ the
cost of updating the tree nodes, and $C_D$ the cost of updating leaf nodes’ data list. If $\alpha$ is
the percent of points that moved out from their last-known leaf nodes, $\beta$ is the percent of modified tree nodes, and $\gamma$ is the percent of leaf nodes whose data lists have modified as the result of the update, we have $C_L = O(\alpha N \log N)$ and $C_V = O(\beta n)$ in both updating approaches, which are proportional to level of data movement. However, $C_D = O(\gamma \alpha N)$ in paging approach, and $C_D = O(N)$ non-paging approach. As discussed in Equation (3.2), $C_D$ is the dominating cost. Since this cost is proportional to the level of data movement in paging approach, it is expected that this approach shows better performance.

5.2 Tree Updates in Multiple GPUs

We also modified the bulk update procedure to support updates in multiple GPUs. To that end, the tree node structure (not including the data lists) replicates into all the available GPUs in the cluster. A local copy of counters and lists for keeping track of the movements are assigned to each copy on each GPU. Then, the new positions of points are sent to the GPUs to start the update procedure. Beside the kernels mentioned in the single GPU environment, two other kernels are designed to fulfill the task in Multi-GPU environment: Node-Counter-Updating, and Data-Points-Transferring.

The Movement-Check kernel is launched on each GPU with as many threads as the number of points assigned to that GPU followed by Leaf-Finding kernel call if Moved_List is not empty. To update the total number of points that moved in/out of a node, the local counter values are transferred to the master GPU for parallel reduction. Then, the Node-Counter-Updating kernel is called on the master GPU for updating the actual counter values in each node, and updating the node_split list.

The Node-Creation, Node-Partitioning and Node-Merging kernels are executed on the master GPU if the prerequisites for calling those kernels are satisfied. Then, finding the new location for points in the modified nodes is done using the GPUs where the data lists are stored.
Having finished all these steps, the data lists can be updated. As neighboring leaf nodes’
data lists are most likely stored in the same GPU, data movement can be handled without
communications between two GPUs. However, if there are points that move to a leaf node
residing in another GPU, they will be transferred to corresponding GPUs using Data-Points-
Transferring kernel before updating the leaf nodes data lists. Then, Leaf-Lists-Update kernel
is called to update the data lists. This can be done using the page-based or reinsertion
mechanism mentioned in the single-GPU approach.

5.3 Experimental Evaluations

In this section, we present empirical evaluation of G-PICS tree update performance. For
that, we implemented a CUDA version of tree update algorithms within G-PICS.

5.3.1 G-PICS Performance in a Single GPU Environment

To evaluate the performance of the tree update procedure, we change the positions of
a certain percentage of the input data points to new randomly-generated positions. Then,
the tree is updated accordingly. To measure the performance, the percent of the input data
point that moved out from their last-known leaf nodes is captured. Then, the time it takes
to update the tree is compared with that of building the tree from scratch using G-PICS tree
construction code (we refer to it as Scratch hereafter). In the following experiments, $MH$
is set to 1024 and $MH$ is set to 14. Figure 5.3 shows the speedup of the tree update algorithm
using paging and non-paging approaches over Scratch. For both approaches, G-PICS out-
performs Scratch. As we expected, with the increase of the intensity of data movement, the
update performance decreases. Furthermore, the paging approach outperforms non-paging
approach remarkably, even under very high level of movement. The experimental results
are in compliance with update cost in Equation 5.1, which confirms the dominating cost in
paging approach is proportional to the intensity of data movement.
5.3.2 G-PICS Performance in a Multi-GPU Environment

We evaluate our multi-GPU algorithms with a focus on performance scalability. For that purpose, we use a dataset that can fit into one GPU’s global memory but distribute it to multiple GPUs. The experiments are conducted by using one to four GPUs.

To measure the efficiency of multi-GPU tree updates in G-PICS, both paging and non-paging approaches are implemented and performance are evaluated against Scratch under the same set-up, e.g. tree update performance under four GPUs is measured against Scratch using four GPUs. According to Figure 5.4, the paging approach outperforms the non-paging...
approach using Multi-GPU with a similar trend as a single GPU. This is again in compliance with the update cost in Equation (5.1). Due to high communication cost among multiple GPUs, the multi-GPU update procedure is less scalable than query processing algorithms.
Chapter 6: Case Studies

In this Chapter, we exploit algorithms within G-PICS framework to improve the performance of other existing problems via two case studies: we demonstrate Computing Spatial Distance Histograms as the case study I; and we present a variation of spatial joins as the case study II.

6.1 Case Study I: Computing Spatial Distance Histograms

There exists several variations and forms of Two-body statistics, which play important roles in a broad range of scientific applications. One of the famous variation of the two-body statistics are a group of statistical computations that assesses a computational function between all pairs of data points in an existing dataset. The specific problem we target in this section is spatial distance histogram (SDH) [58]. A typical SHD outputs the histogram of distances between all pairs of data points in a dataset. Such histogram presents a discrete approximation of the continuous probability distribution of data points’ distances. However, the significant bottleneck in successful deployment of SDHs is its worse-case quadratic time complexity ($O(N^2)$). Therefore, some previous work concentrated on developing better algorithms with lower computation complexity. In [58, 54], the computation complexity of SDH problem is reduced to $O(N^{1.5})$ by using a point-region quadtree. However, in those work, quadtree construction and SDH computation are performed using a single thread approach on CPU. Therefore, has much room for improvement. To this end, parallel computing

\footnote{Some part of this chapter was published in ACM proceedings of the 30th International Conference on Scientific and Statistical Database Management. Permission is included in Appendix A.}
can be exploited to speedup the SDH computation in those work. For this reason, we utilize the quadtree construction algorithm within G-PICS to parallelize quadtree construction for SDH computation, and then we parallelize SDH computation introduced in [58] using G-PICS quadtree on GPU.

6.1.1 Problem Statement

A typical SDH computation can be defined as follows: having \( N \) input coordinated data points and a user-specified distance \( d \), the objective is to calculate the number of point-to-point distances falling into a series of ranges (buckets) of width \( d \): \( [0, d), [d, 2d), [2d, 3d), ..., [(k-1)d, kd] \). Consequently, the SDH generates an ordered list of buckets with an associated positive integers to each bucket \( B = (b_0, b_1, ..., b_{k-1}) \), where the value in each bucket is the total number of distances falling into the range of that bucket. For example, the value in the bucket \( b_i \), shows the total number of distances falling within range \( [id, (i + 1)d) \). List of symbols that are used in this section are summarized in Table 6.1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>total number of input coordinated data points</td>
</tr>
<tr>
<td>( d )</td>
<td>width of histogram buckets</td>
</tr>
<tr>
<td>( b )</td>
<td>histogram bucket associate with bucket ( b_i ) (( 0 &lt; i \leq 1 ))</td>
</tr>
<tr>
<td>( B )</td>
<td>output histogram</td>
</tr>
<tr>
<td>( i )</td>
<td>bucket index</td>
</tr>
<tr>
<td>( \delta )</td>
<td>tree node diagonal length</td>
</tr>
</tbody>
</table>

In processing SDH using the brute-force approach, the bottleneck comes from the fact that distances between any pair of points need to be calculated to build the output distance histogram. We know that width of histogram buckets is always non-zero. Therefore, having a pair of points, their relevant histogram bucket can be determined if a range that their
Algorithm 7: Resolve-Two-Nodes Procedure \((C_1, C_2)\)

**Global Var:**  
\(\text{maximum\_dist}\) (maximum distance between two tree nodes),  
\(\text{minimum\_dist}\) (minimum distance between two tree nodes)

1: calculate \(\text{maximum\_dist}\) and \(\text{minimum\_dist}\) between \(C_1, C_2\)
2: if \(\text{maximum\_dist}\) and \(\text{minimum\_dist}\) fall into the same bucket \(i\) then
3: \(C_1\) and \(C_2\) are resolved
4: \(b_i = b_i + p_1 \times p_2\)
5: else if \(C_1\) and \(C_2\) are not leaf nodes then
6: for each \(C_1'\) in \(C_1\) child nodes do
7: for each \(C_2'\) in \(C_2\) child nodes do
8: Resolve-Two-Nodes \((C_1', C_2')\)
9: end for
10: end for
11: else
12: for each point \(E\) in \(C_1\) do
13: for each point \(F\) in \(C_2\) do
14: \(m \leftarrow \text{distance between } E\) and \(F\)
15: \(i \leftarrow \text{output bucket for } m\)
16: \(b_i = b_i + 1\)
17: end for
18: end for
19: end if

Knowing this fact, can help in saving a lot of time in pair-to-pair distance computations.

In [58], a point region quadtree is utilized to divide the input domain into equally-sized cells (quadtree nodes). Then, the constructed tree is used for SDH computation. The main idea behind this approach is to process clusters of points (tree nodes) to take benefit of the non-zero width of the SDH bucket.

The main procedure in this approach is called Resolve-Two-Nodes, which takes two tree nodes as input. In order to resolve two input tree nodes \(C_1\), and \(C_2\) with total number of \(p_1\), and \(p_2\) points in each node, respectively, the minimum and maximum distances between any points in \(C_1\), and \(C_2\) are computed. Since, in quadtree the coordinates of nodes in input domain are known, this step can be done in constant time. If the calculated minimum and
Algorithm 8: Tree-based-SDH

Global Var: \( l \) (starting level in the tree),
1: build the quadtree for input data points
2: initialize the histogram buckets with zero \((B \leftarrow 0)\)
3: \( l \leftarrow \) the level whose nodes have diagonal length \( \delta \leq d \)
4: for all tree nodes in \( l \) do
5: \( p \leftarrow \) number of points in the node
6: \( b_0 = b_0 + \frac{1}{2}p(p - 1) \)
7: end for
8: for any two nodes \( C_1 \) and \( C_2 \) in \( l \) do
9: call Resolve-Two-Nodes Procedure \((C_1, C_2)\)
10: end for
11: return \( B \)

maximum distances among \( C_1 \), and \( C_2 \) fall into the same bucket \( i \) in the output histogram, these two nodes are resolvable into bucket \( b_i \). In this case, the relevant value in bucket \( b_i \) increases by \( p_1 \times p_2 \). On the other hand, if two nodes are not resolved, all pairs of their child nodes are recursively resolved by calling Resolve-Two-Nodes procedure. This recursion continues until the lowest level of the tree (leaf nodes) is reached. In this case all distances between the points in unresolved nodes are computed to update the final histogram. The aforementioned steps are shown in Algorithm 7.

Another issue that should be targeted in this solution is, which nodes in the tree should be processed using Resolve-Two-Nodes procedure. To this end, the algorithm should start working on tree nodes in a level whose its nodes’ diagonal length \((\delta)\) are smaller than histogram bucket width \( d \) \((\text{node side length } \delta \leq d^{\frac{1}{\sqrt{2}}})\). Therefore if there is no tree level that satisfy this condition, there are no pairs of nodes that are resolvable in the tree. All pairs of point-to-point distances between points in the tree nodes in the starting level are smaller than \( d \), consequently, such distances are added into the first histogram bucket with range \([0, d)\). Then, the Resolve-Two-Nodes procedure is called for all pairs of the tree nodes in the tree level that satisfies the starting level condition (Algorithm 8).
6.1.2 SHD Parallelization using G-PICS on GPU

Although this approach decreases the computation complexity of the SDH computation, it runs using single thread on CPU. Therefore has more room for improvement. Since, G-PICS encapsulates an efficient quadtree construction on GPU, the tree used for this problem can be efficiently built in parallel on GPU. Then, the constructed tree can be utilized to perform the SDH computation and outputting the results in parallel on GPUs. Having the tree, the starting level can be determined in advance, and therefore the total number of nodes in that level. The parallelism paradigm on parallelizing the Resolve-Two-Nodes Procedure (Algorithm 8) depends on the number of nodes that are determined in the starting level for the algorithm. If there are enough nodes to utilize GPU cores efficiently on GPU, a CUDA kernel is launched with as many threads as the number of pair of nodes in that level. Then, each pair of nodes in the starting level is assigned to one GPU thread and the Resolve-Two-Nodes procedure is performed by that thread. However, if the algorithm starts at very high level of the tree, there are not enough nodes to efficiently utilize GPU cores by assigning each pair of nodes to one thread. In this case, at higher levels of the tree each pair of nodes are assigned to one GPU block of threads, and at lower levels of the tree, each pair of child nodes are assigned to one GPU thread. The parallel Resolve-Two-Nodes Procedure algorithm on GPU is shown in Algorithm 9.

To evaluate the performance of the designed algorithm, we implemented its CUDA version. Figure 6.1 shows the performance comparison of the single thread SDH computation on CPU versus our developed parallel version on GPU. In which input dataset is indexed using quadtree. In this experiment $MC$ is set to 128, $MH$ is set to 12, and the number of histogram buckets $(b)$ is set to 80. Figure 6.1 shows as the number of input data points increases, the SDH computation time increases on both CPU and GPU. However, GPICS outperforms CPU remarkably for any input size by up to 96X.
Algorithm 9: Resolve-Two-Nodes Procedure on GPU

Require: \( t \) (Thread id in a block)
- \( \text{maximum\_dist} \) (maximum distance between two tree nodes assigned to \( t \))
- \( \text{minimum\_dist} \) (minimum distance between two tree nodes assigned to \( t \))
- \( l \) (starting level in the tree)

1: \( t \leftarrow C_1, \text{ and } C_2 \) in \( l \)
2: calculate \( \text{maximum\_dist} \) and \( \text{minimum\_dist} \) between \( C_1, C_2 \)
3: if \( \text{maximum\_dist} \) and \( \text{minimum\_dist} \) fall into the same bucket \( i \) then
4: \( C_1 \) and \( C_2 \) are resolved
5: atomicAdd \((b_i, p_1 \times p_2)\)
6: else if \( C_1 \) and \( C_2 \) are not leaf nodes then
7: for each \( C'_1 \) in \( C_1 \) child nodes do
8: for each \( C'_2 \) in \( C_2 \) child nodes do
9: Resolve-Two-Nodes \((C'_1, C'_2)\)
10: end for
11: end for
12: else
13: for each point \( E \) in \( C_1 \) do
14: for each point \( F \) in \( C_2 \) do
15: \( m \leftarrow \text{distance between } E \text{ and } F \)
16: \( i \leftarrow \text{output bucket for } m \)
17: atomicAdd \((b_i, 1)\)
18: end for
19: end for
20: end if

Figure 6.1: G-PICS SDH computation time, and speedup over CPU
6.2 Case Study II: Spatial Join

G-PICS can be extended to support other spatial data retrieval algorithms and applications. For example, we extend G-PICS to support a special type of spatial join named the 2-body constraints problem, which retrieves all pairs of objects that are closer than a user-specified distance \((d)\) from each other. Using query processing algorithms within G-PICS, in the first step of this search, each leaf node registers itself to the query list of all other leaf nodes with distance less than or equal to \(d\). Then, in the second step of the search all matching pairs within those leaf nodes are retrieved and outputted (Algorithm 10). Since the output size of a typical spatial join search is not known in advance, we again use the buffer pool solution discussed in Section 4 for outputting such query results.

To evaluate the performance of implemented spatial join using G-PICS algorithms, we compare its performance against M-STIG. However, in processing this type of queries, the second step of the search for G-PICS and M-STIG is the same (Algorithm 10). Consequently, the first step of the search determines the overall performance. To evaluate the performance of G-PICS we run an experiment over a dataset with 9.5 million data points, which are indexed using the parallel quadtree construction mechanism of G-PICS. The \(MC\) is set to 1024, and \(MH\) is set to 12. Figure 6.2 shows the step I running time in G-PICS and its performance comparison over M-STIG (in which all leaf nodes are searched in a brute-force manner), with variable distances. The cost of query registering in the step I of M-STIG is constant regardless of \(d\). Therefore, Figure 6.2 shows that achieved speedup in G-PICS decreases gradually by increasing \(d\). This is the result of visiting more leaf nodes in the search. However, in all cases G-PICS outperforms M-STIG by up to 14.3X.
Algorithm 10: Second step of spatial join

Local Var: $b$ (Block id),
$t$ (Thread id in a block),
$M$ (total number of points in the leaf[$b$]),
$lqL$ (query list attached to the leaf[$b$]),
$N$ (total number of registered leaf node in $lqL$),
$d$ (search distance),
dL$_i$ (list of points in leaf node i)

1: dL$_b$ ← leaf$datalist[leaf[b]]$
2: for each dL$_b[t]$ in parallel do
3: for $i = t+1$ to $M$ do
4: $r$ ← computeDistanceFunction(dL$_b[t]$, dL$_b[i]$)
5: if $r < d$ then
6: Add dL$_b[i]$ to the Output list of $t$
7: end if
8: end for
9: end for
10: for $j = 1$ to $N$ do
11: dL$_j$ ← leaf$datalist[leaf[lqL[j]]]$
12: for $k = 1$ to dL$_j$.numberOfPoints do
13: $r$ ← computeDistanceFunction(dL$_b[t]$, dL$_j[k]$)
14: if $r < d$ then
15: Add dL$_j[k]$ to the Output list of $t$
16: end if
17: end for
18: end for

Figure 6.2: G-PICS step I spatial join time, and speedup over M-STIG
Chapter 7: Conclusions

We advocate the adaptation of GPUs in spatial query processing, especially in applications dealing with concurrent queries over large input datasets. Existing work in this topic show low work efficiency and cannot make good use of GPU resources. To that end, we present a GPU-based Parallel Spatial Data Indexing framework named G-PICS for high performance spatial data management and concurrent query processing. G-PICS provides new tree construction algorithms on GPUs, which achieves a high level of parallelism and shows a performance boost of up to 53X over the best-known parallel GPU-based algorithms. Moreover, G-PICS introduces a new batch query processing framework on GPUs to tackle the low work efficiency and low resource utilization existing in current one-query-at-a-time approaches. G-PICS supports the processing of major spatial query processing such as spatial point search, range search, within-distance search, and k-nearest neighbors. G-PICS query processing shows a great performance speedup over the best-known parallel CPU-based and GPU-based spatial query processing systems (up to 80X). In addition, G-PICS provides an efficient parallel update procedure on GPUs to support dynamic datasets which outperforms the tree construction from scratch by up to 16X. Furthermore, all algorithms within G-PICS work in Multi-GPU environments to support large datasets beyond the capacity of a single GPU global memory. On the other hand, we utilize algorithms within G-PICS framework to improve the performance of other existing problems via two case studies: computing spatial distance histograms, and 2-body constraints (a variation of spatial joins). Experimental results in both cases show that how utilizing algorithms within G-PICS can remarkably improve the performance in such applications.
References


Appendix A: Copyright Permissions

The reuse permission for parts of Chapter 1, 2, 3, 4, 5 and 6 is listed below.

ACM Copyright and Audio/Video Release

Title of the Work: GPU-Based Parallel Indexing for Concurrent Spatial Query Processing
Submission ID: 03
Author/Presentation: Zhihao Nourl; University of South Florida; Yi-Cheng Tu; University of South Florida
Type of material: Full Paper

Publication and/or Conference Name: 30th International Conference on Scientific and Statistical Database Management Proceedings

1. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and