Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1870 – 1879

Procedia Computer Science

International Conference on Computational Science, ICCS 2012
Workshop on using Emerging Parallel Architectures (WEPA 2012)

Speeding up spatial database query execution using GPUs

Bogdan Simion[1,*], Suprio Ray[*], Angela Demke Brown[*]

Department of Computer Science, University of Toronto

Abstract

Spatial databases are used in a wide variety of real-world applications, such as land surveying, urban planning, and environmental assessments, as well as geospatial Web services. As uses of spatial databases become more widespread, there is a growing need for good performance of spatial applications. In spatial workloads, queries tend to be computationally-intensive due to the complex processing of geometric relationships. Furthermore, a significant fraction of spatial query execution time is spent on CPU stalls due to memory accesses, caused by the ever-increasing processor-memory speed gap. With the advent of massively-parallel graphics-processing hardware (GPUs) and frameworks like CUDA, opportunities for speeding up spatial processing have emerged. In addition to massive parallelism, GPUs can also better hide the memory latency.

We aim to speed up spatial query execution using CUDA and recent GPU cards. One of the main challenges in using GPUs is the transfer time from main memory to GPU memory. We implement a set of six typical spatial queries and achieve a baseline speedup (without the transfer cost) of 62-318x over the CPU counterparts. We show that the transfer cost can be amortized over the execution of each individual query. For simpler spatial queries, the transfer time is a significant fraction of the query execution time, but we still achieve a 6-10x speedup. For more complex spatial queries, the transfer time becomes negligible compared to the processing time, and we obtain a 62-240x speedup.

Keywords:
Spatial databases, Parallel query execution, GPU processing

1. Introduction

Spatial data support has recently emerged as a vital component in a wide spectrum of applications. A range of Geospatial Web services such as Google Maps, in-vehicle GPS navigation systems, GPS-enabled mobile phones, and a host of accompanying location-based services have become part of our daily experience. At the other end of the spectrum, applications such as land surveying, urban planning, natural disaster prevention, natural resource management or environmental applications that have been around for decades are increasingly using large spatial datasets as the basis for complex studies, real-time analyses and reporting.

The volume of spatial data generated and consumed is rising and new applications are emerging as the costs of storage, processing power and network bandwidth continue to decline. Efficient support for processing spatial data

[*]This research was funded by the NSERC Discovery Grants program.
[1]Corresponding author: Bogdan Simion (bogdan@cs.toronto.edu)

doi:10.1016/j.procs.2012.04.205

is becoming a crucial component of every database system. With increasing dataset sizes and complex user requests, long-running spatial queries can cripple the performance of a spatial DBMS.

To illustrate some typical geospatial queries, consider a map representation including locations of landmarks, rivers, roads, land parcels, etc. Simple queries might include: "Find the closest hospital to home" or "Find the number of supermarkets within walking distance". More complex analytical queries include environmental applications such as finding waterways close to a toxic spill to determine pollution spread, determining flood risk areas based on proximity to rivers and terrain elevation, or creating land information management reports.

Spatial database workloads have not been studied in depth thus far, and to the best of our knowledge there has been no research on identifying the underlying system issues leading to performance problems in spatial databases. Preliminary work that we have conducted over the past year suggests that the main performance bottleneck becomes the CPU as database buffer cache sizes increase and the workload complexity grows.

Spatial query execution is a two-step process involving filtering of shapes based on their minimum bounding rectangles (MBRs) and then refining the candidate result set by comparing the actual geometries. This involves heavy CPU processing, in the form of complex computational geometry operations (e.g., determining intersections or detecting overlap between complex shapes, etc.). These geometric computations are performed on large sets of independent spatial data objects and thus, naturally lend themselves to parallel processing.

The ever-increasing gap between CPU and memory speeds is also an important factor. We analyzed the microarchitectural behavior of spatial workloads and observed that CPU stalls (due to memory access, branch mispredictions and unavailable functional units) account for 45% of execution time on average. While this is less than has been observed for decision support workloads in relational databases [1], it is still significant and likely to grow.

Graphics processing units (GPUs) are gaining popularity for fast parallel processing of large datasets [2]. The massively parallel architecture of GPUs offers significant parallelization potential and can achieve huge speedups compared to traditional CPUs, especially if the applications are characterized by a high degree of parallelism. GPUs also benefit from fast interprocessor communication through local memory, and from coalesced memory accesses. They therefore manage to hide the latency of memory stalls much better than modern CPUs. As a result, leveraging the parallel processing power of GPUs for speeding up spatial analytical queries is a promising avenue of exploration.

The main drawback with processing data on a GPU is the high cost of transferring data from the host (CPU memory) to the device (GPU memory). To use GPUs for spatial query processing, the problem of the memory transfer cost has to be somehow mitigated. If we assume that the data is kept in GPU memory, then spatial queries are perfectly suited for processing on the GPU. If on the other hand, the dataset needs to be transferred from host to device memory, this cost needs to be amortized over the execution of one or several spatial queries on the same dataset.

To determine the parallelization potential of spatial queries on GPUs, we implemented a set of basic spatial functions on a GPU using the CUDA framework. We tested several representative analytical queries on a medium-size synthetic dataset of up to 16 million spatial data shapes. The queries ran 62-318x times faster on a GPU compared to the CPU-equivalent code and, if we include the memory transfer times between host and device, 6-10x faster for simple spatial queries and 62-240x in case of complex spatial queries that render the transfer time negligible. Since the transfer time is easily amortized over the execution of each single query, then given multiple queries on the same dataset, the benefits are expected to be even higher.

The rest of this paper is organized as follows. In Section 2, we give some brief background on geospatial databases. Next, we describe the query execution model and the implementation of our algorithm on the GPU in Section 3. Section 4 describes the experimental setup and our results. In Section 5, we describe some of the challenges of using GPUs and discuss some possible workarounds. We discuss related work in Section 6, and conclude in Section 7.

2. Background

We first provide a brief introduction to spatial databases, including spatial data representation and the spatial query execution mechanism. Next, we present some background on general purpose GPU architecture and processing.

2.1. Spatial databases

The term "spatial database" refers to a relational database management system that supports spatial data types in the same way as any other data. Although spatial support was once a niche feature provided by high-end systems for a small set of customers, it is now widely used with new applications appearing regularly.

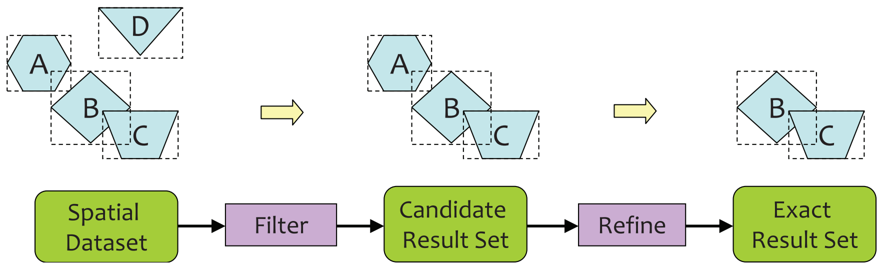


Figure 1: Example of spatial filtering and refinement

Spatial data types. Spatial data types hide the details of data storage and are crucial in a spatial database system. There are two types of spatial data: raster data and vector data, each distinct in the information they contain and in the way the features are represented. Raster data contains representations of various types of maps (geographical, meteorological, etc.), and are in essence high-resolution images, generally obtained from satellite imagery, that are relevant for a particular field of geographic study. In contrast, vector data types capture the fundamental abstractions for geometric shapes representing the most relevant geographical features, such as points for landmarks or buildings, lines for rivers and roads, or polygons for lakes and land parcels.

A vector data object is a representation of the spatial feature, using coordinates in a spatial reference system. A spatial object is generally a single element, consisting of an instance of one of the supported primitive types, or, in rare cases, it can consist of a collection of geometries (e.g., a set of islands). Thus, we can view a vector data object as a set of coordinates describing the corresponding geographic shape. Approximations for spatial data, such as the minimum bounding rectangle/box (MBR), are frequently used to speed up the spatial query processing.

Spatial query processing. Spatial databases support *spatial queries*, which allow for the use of geometric data types (such as points, lines and polygons) and can consider the spatial relationships between the geometric shapes in the spatial datasets. Spatial relationships that can be used to retrieve information from spatial data include topology and spatial analysis. Examples of topological relationships include determining if a geometric shape equals another shape, or if two geometric shapes intersect (e.g., line intersects line). Spatial analysis functions involve a quantitative description of a shape, such as length, area, centroid, or convex hull.

Finally, spatial query processing, unlike traditional query processing, involves a two-step process comprised of filtering and refinement stages. This process makes use of approximations for common spatial data [3]. In the first step, the records are filtered based on the minimum bounding rectangles (MBRs) of the data shapes, to narrow down the possible matches to a much smaller candidate result set. For each candidate (or pair of candidates in case of spatial join) the exact geometry is checked in the second step. Essentially, the records in the candidate result set from the first step are processed to determine which shapes satisfy the spatial criteria.

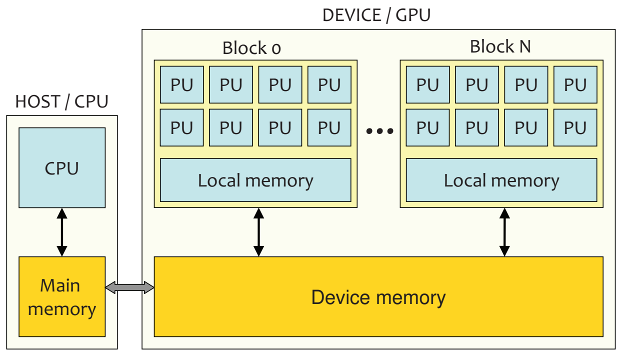
The objective of the filter step with approximations is to eliminate as many non-matching spatial objects as possible before performing more costly operations on the actual representations of the complex spatial objects. Because all the filtered primitives must be checked, the second step is the most costly and time-consuming procedure.

These steps are shown [4] in Figure 1. The query searches for the spatial objects that intersect with B. The MBRs of the objects B and D do not intersect, therefore the objects also cannot intersect. Thus, the filter step can eliminate D without retrieving the exact geometry. Next, the refinement has to compare the exact geometries to determine which ones actually intersect with B. This yields the exact result set, containing shape C.

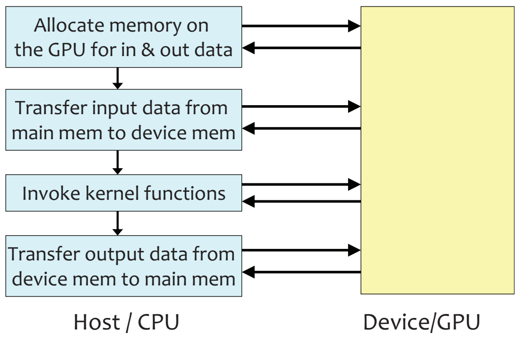
Most spatially-enabled databases comply with this two-step evaluation. Thus, we also employ a similar strategy when evaluating operators involving spatial functions within a query in our GPU implementation.

2.2. Graphics Processing Units

Initially designed for fast geometric computations in video cards, GPUs have evolved into highly parallel many-core systems allowing highly efficient manipulation of large datasets, especially for embarrassingly-parallel applications. In a nutshell, a GPU is a collection of a high number of SIMD multiprocessors. The architecture of a GPU is shown in Figure 2(a). The multiprocessors, or *blocks* of processing units, are structured in a grid. Each of the processing units within each block performs the same computations, but operating on different data.



(a) GPU architecture


Allocate memory on the GPU for in & out data
Transfer input data from main mem to device mem
Invoke kernel functions
Transfer output data from device mem to main mem
Host / CPU
Device/GPU

(b) GPU processing

Figure 2: GPU architecture and programming model

The GPU makes use of a *device memory* or *global memory*, which is shared between all blocks of processors. Device memory is large (on the order of a few GB in current generation GPUs), and is characterized by high bandwidth, but high latency as well (roughly 200 cycles per access). Within each block, the processing units make use of a *local memory*. The local memory is shared within the block, it has a much smaller size than the device memory (a few tens of KB), and it is much faster to access. An interesting feature of GPUs is *memory coalescing*, which allows accesses from several processing threads to consecutive memory cells to be grouped together into one access.

The main differences between CPUs and GPUs stem from their hardware architecture and the goal for which they were designed. GPUs provide parallel low-frequency execution over thousands of processors in a SIMD fashion, whereas CPUs (including current generations of multi-cores) are mostly designed for high-frequency out-of-order superscalar execution on a much smaller number of cores. GPUs are ideal for large scale computations that are inherently parallel or for whom embarrassingly parallel algorithms are well-known. GPU memory is much smaller than main memory in a traditional processor, but GPUs possess much higher memory bandwidth, allowing multiple pieces of data to be manipulated in parallel by a large number of processors.

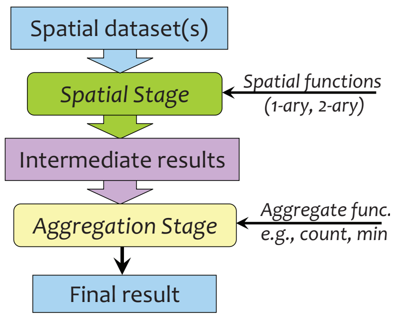
The architecture of general purpose GPUs (GPGPUs) has been exposed more clearly with the introduction of hardware programming frameworks such as CUDA (Compute Unified Device Architecture) and OpenCL. In these frameworks, there is a clear separation between *host code* (CPU-side) and *kernel code* (GPU-side). The host code contains all the processing done on the CPU, it can manipulate data transfers between main memory and GPU memory, and can launch kernel code on the GPU. The kernel code is executed in parallel on the GPU in a SIMD fashion.

Generally, a piece of code designed to perform computations on the GPU contains several steps, as seen in Figure 2(b). First, the host code must allocate memory on the GPU for input and output data. Next, the host code uses API calls to transfer data from main memory into GPU memory, into the input data previously allocated. After the memory transfers complete, the host code can invoke kernel functions on the GPU. The kernel launches the parallel task processing on the GPU and after the tasks complete, control is restored to the CPU. The host code can then use API calls to transfer the output from GPU memory to main memory.

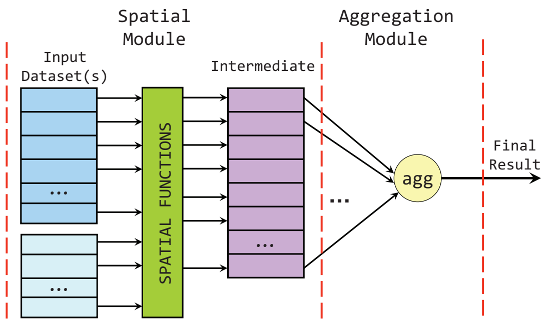
3. Implementation

3.1. Query execution model and query classes

In relational databases, a query planner and optimizer is responsible for processing the SQL query and generating an optimized query plan. The query plan consists of a tree of operators (such as selection, join, aggregation operators, etc.). Most analytical spatial queries involve two stages of processing: a stage performing a spatial function, followed by an aggregation operation. Consequently, we implement a two-stage query processing framework. In Stage 1, a spatial operation is performed on one or several spatial tables. This can be either a basic spatial selection where a spatial function (e.g., area) is applied to an entire table, a spatial join with a given spatial object, or a spatial all pair join between two datasets. In this paper, we focus on the latter two, since they are far more complex. Stage 1 generates an intermediate result set which is then used as input for the Stage 2 operator. Stage 2 applies an aggregation function (such as count, min, max etc.) to the intermediate result set, to obtain the final result. Figure 3(a) describes this query execution model, which takes the geospatial dataset as input.



(a) Query execution on the CPU



(b) Query processing on the GPU

Figure 3: The query execution algorithm involves a spatial stage and an aggregation stage (left). On the GPU, each stage is implemented as a separate kernel module (right) that can be invoked on the corresponding data.

For Stage 1, we choose two classes of spatial queries: class A, involving a spatial join with a given object, where essentially a spatial function is applied to a spatial dataset (e.g., "distance" from a given point, "within" a given area, and "intersects" a given line), and class B, where two relations are joined on the spatial attribute, using a topological function (e.g., "intersects" between two line tables, such as is the case for determining all crossings between roads and rivers). In Stage 2, the intermediate results are then aggregated into a final number (e.g., find the "minimum" of the distances from a given point, or "count" all records that are within a given area).

3.2. *Algorithm implementation*

The query execution model is implemented in both the GPU and the CPU, and is comprised of two stages, as seen in Figure 3(b). In the spatial processing stage, the function corresponding to the user request, such as distance (from a given object), within (a given area), intersects (a given line or line set) etc., is applied (in parallel on the GPU) to the entire dataset, then the intermediate results are fed to the second stage which aggregates them into the final result.

On the GPU, each stage is implemented as a separate kernel. In Stage 1, each thread is assigned an equal number of records to process (one or more, depending on the dataset size and the maximum number of threads and thread blocks on the GPU). Stage 2 uses an optimized parallel reduction algorithm, which makes use of the fast access shared memory on the GPU and optimizes for memory coalescing and avoiding memory bank conflicts.

Since the fast local memory is limited, and global memory has higher latency, the use of texture memory seems promising due to the caching it provides. However, Bakkum and Skadron [5] noted that texture memory does not influence performance for database queries in a significant way because the row-column format of databases enables most memory accesses to be coalesced, which in turn reduces the need for caching. Thus, we chose to use only global memory for the spatial stage and a combination of global and local shared memory for the aggregation stage.

One of our secondary goals was to allow the code to be easily extended with additional querying functionalities. We aimed to implement both the spatial functions and the aggregation functions as generically as possible, so that the kernels executing the query processing stages could be passed a function pointer, representing any user-defined function. For the CPU implementation this was straightforward; however, for the GPU, there was no support for function pointers in the test and development environment we had available. In newer-generation Fermi cards, support for several features including function pointers has been added, so we expect our GPU implementation can also be adjusted, with minimal coding effort, to transparently plug-in new spatial functionalities.

4. Evaluation

4.1. *Experimental setup*

In our experimental testbed we used an Intel Quad-Core CPU @ 2.83GHz, 4GB RAM and a GeForce GTX 480 card (480 cores), processor clock rate 1.4GHz and graphics clock 700MHz, with 1.5GB of GDDR5 global memory and 48KB of shared memory per thread block and 512KB of local memory per thread, with a CUDA 4.0 driver.

We create an artificial dataset comprised of points, lines and polylines, represented using floating point single-precision 2D coordinates and stored in GPU global memory. The TIGER (Topologically Integrated Geographic

Table 1: Test scenarios

Query#	Scenario Name	Description	Shapes
Q1	ClosestHospital	Finds closest hospital to a given location point	Points/Point
Q2	StoresWithinRange	Determines how many supermarkets are within a 10 km radius of a given location	Points/Point
Q3	ToxicSpill	Determines how many toxic spills are close to a given river	Points/Line
Q4	RiverCrossings	Computes the number of roads crossing a given river	Lines/Line
Q5	Bridges	Computes the number of all pairs of roads and rivers which intersect each other	Lines/Lines
Q6	Bridges-PolyLine	Same as Bridges, but more complex geometries	PolyLines/PolyLines

Encoding and Referencing system) [6] spatial dataset served as a model for our dataset. The shapes in the TIGER dataset tables use points to represent locations such as landmarks, and lines to represent features such as roads and rivers. These are described using 2D floating point latitude-longitude coordinates, thus we chose the same data types for our experiments. Our dataset consists of 1 to 16 million records containing randomly generated floating point 2D coordinates.

We implemented six representative spatial queries which make use of three different spatial functions, including spatial and topological relations between vector data types. As shown in Table 1, the first and second scenarios simulate every-day user requests, the third is an environmental scenario, and the last three are land information queries.

The most important metric for spatial queries is execution time, since the response time is what an end-user experiences when issuing a particular query. Therefore, we measure the execution time of each spatial query on the GPU and then on the CPU-equivalent implementation using the same dataset. We present the speedup as the time measured on the CPU divided by the time seen on the GPU.

We ensure the dataset is present entirely in CPU memory, to eliminate any interference from disk accesses and conduct a fair comparison. Since vector data is generally more compact than raster data, this setup is reasonable. For example, a realistic TIGER dataset for the state of Texas is just under 2GB, while the entire continental US fits into approximately 32GB. We also assumed that our generated dataset is present in GPU memory. Nevertheless, we also measured the transfer time from host (CPU) memory to device (GPU) memory, to estimate if the memory transfer cost is significant and if it can be easily amortized over the execution of one or multiple queries on the same dataset.

Next, we present our experimental results that show significant speedups when executing the test queries on the GPU over the CPU variant of the implementation. We also show that the cost of the memory transfer can be amortized over the execution of a single spatial query on the experimental dataset.

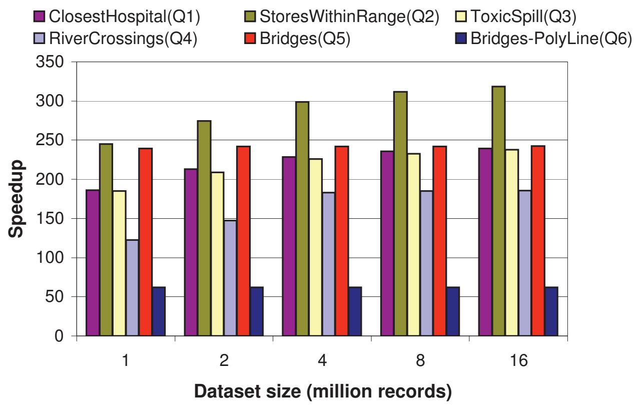
4.2. Experimental results

All spatial query scenarios were tested on incrementally larger datasets, ranging from 1 million to 16 million shapes, to determine the speedup variation with increasing dataset size. For the Bridges scenario, the roads dataset contains 1-16 million roads, while the rivers dataset contains 1024 rivers. The Bridges-PolyLine scenario contains more complex PolyLine geometries, which take longer to process since intersections must be calculated with all the basic geometries encompassed within each PolyLine.

We calculate speedup using the average execution time of 10 trials for each experiment. The time variation between trials is insignificant (less than 1%). We first present the speedup results excluding the cost of the memory transfer from host (CPU) memory to device (GPU) memory, then present the speedups including this overhead.

To explain the speedup variation across different scenarios, one must consider that each of the queries involve different spatial data types (e.g., points, lines, polylines), and/or different spatial operations (e.g., distance, intersects, within), each with a different processing cost on the CPU and GPU. For example, the StoresWithinRange and ToxicSpill scenarios operate on similar data but perform different computations, with the latter exhibiting more complex computations for both the CPU and each of the processing units on the GPU, which translates into lower gains for the slower-clock GPU. Furthermore, the ToxicSpill scenario operates on point shapes, while RiverCrossings operates with line shapes. Although they perform the same spatial operation ("intersects"), the processing for line intersections involves more complex computations, which explains the lower speedup.

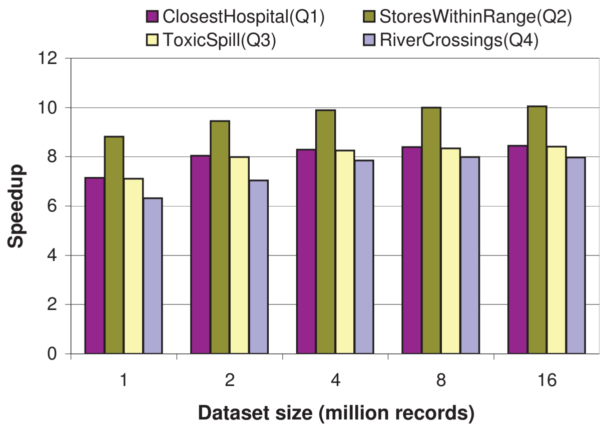
The GPU speedups range from 121x to 318x over the CPU processing (4(a)) for queries 1-5. This is mostly because the spatial operation stage and the aggregation stage are performed on the CPU sequentially, while their GPU



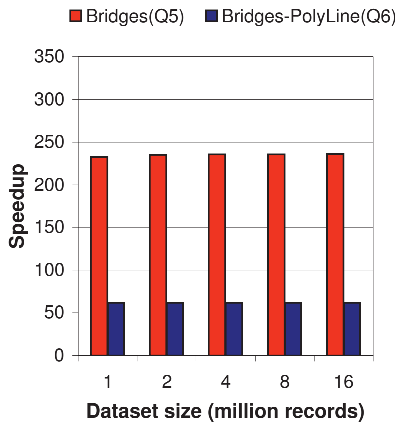
(a) Without transfer time

	CPU	GPU	Transfer
Q1	242.66	1.02	27.64
Q2	297.28	0.95	28.82
Q3	239.29	1.03	27.64
Q4	680.94	3.67	81.65
Q5	648828.65	2680.19	83.25
Q6	5356816.49	86074.91	723.52

(b) Absolute times (ms) for 8 million records



(c) Including transfer time (Q1-4)



(d) Including transfer time (Q5-6)

Figure 4: Speedup results of the GPU over the CPU implementation, varying the dataset from 1 million to 16 million records

counterparts use massive parallelism. For query 6, the lower speedup of just 62x is due to more complex processing for each shape on both the CPU and on the GPU. Since each of the individual processing units on the GPU is less powerful than the CPU, the higher processing complexity for each individual shape implies less overall GPU speedup.

Although Figure 4(a) shows that the speedup seems to increase somewhat with the number of records in the dataset, it is actually fairly constant as the number of records increases. This is primarily because we take advantage of all the processing units on the GPU to perform the computations, which means that when doubling the dataset size, each individual processing unit will have twice as much work to do, and thus the execution time will always double. Similarly, on the CPU, our results show a linear increase of the execution time with dataset size. The choice of aggregation function does not impact the speedup, since different aggregation functions have roughly the same execution time for a given set of intermediate results.

Figure 4(a) does not include the cost of memory transfer from host (CPU) memory to device (GPU) memory. This is a known drawback when executing computations on the GPU. In general, the cost of the memory transfer must be offset by obtaining significant performance gains from the execution on the GPU. For spatial queries, we expected that the initial cost of memory transfer would be amortized over the execution of multiple queries on the same dataset.

We also measured the time to transfer the dataset to GPU memory. Figures 4(c) and 4(d) show the GPU speedups including the memory transfer time ($Speedup = T_{CPU}/(T_{GPU}+T_{trx})$. Fortunately, our expectations were exceeded. In the first four scenarios we still achieve a roughly 6-10x speedup, when including the initial transfer cost. This suggests that the memory transfer cost was amortized by running just a single query, yielding an 8x speedup on average. When running more queries on the same dataset, the benefit is expected to be even higher, since the data will already be present in GPU memory and there will be no additional transfer cost.

Additionally, in the fifth scenario, which involves a much longer running query performing an all pair join between 1-16 million roads and 1024 rivers, the transfer time is negligible compared to the actual execution time. As a result, the speedup when including the transfer time remains around 240x. Consequently, the more complex the spatial queries, the higher the benefit that can be gained using GPUs, as the transfer time becomes insignificant compared to the actual computations. This is also supported by query 6, Bridges-PolyLine, which also exhibits negligible transfer time compared to the processing time, and achieves the same 62x speedup as before.

For the Bridges-PolyLine scenario, the GPU memory could not fit the entire dataset when testing with 2-16 million records. Instead, we used a staged approach for the tests with more than 1 million records. We transferred the dataset in chunks of 1 million records and performed the processing in several rounds. The reported speedups include the total transfer times for all of the chunks, as well as the total times for processing all the rounds (for both Figures 4(a) and 4(c)). The speedup remains the same as for 1 million records (where the entire dataset does fit into GPU memory without the necessity of multiple transfers) since the transfer time is a negligible component of the overall execution time. The CPU processing does not involve a staged approach (the datasets fit into RAM at all tested sizes), but since each of the 1-million record chunks is processed 62 times faster on the GPU, the overall speedup remains the same.

5. Discussion and Limitations

GPU hardware and the CUDA programming framework have been continuously evolving to provide more flexibility to programmers aiming to extract more parallelism from various applications. However, there are still several limitations which are being addressed in newer generations of graphics cards. Some of the less significant limitations only affect programming flexibility or precision, while from a hardware perspective, one of the major limitations is the GPU global memory size.

In the following, we enumerate some of the challenges we faced when using GPUs for spatial database queries, as well as to what extent well-known GPU limitations affect our goals.

a. GPUs lack hardware support for some complex data types. For example, double-precision data types are not universally available on all GPUs. Precision can be very important for scientific computations, as well as spatial database operations (coordinates are represented as double-precision values in general). However, this shortcoming will be addressed by next generation GPUs. In our implementation, all the spatial coordinates were represented in single-precision floating point.

b. GPUs lack support for handling concurrency, such as read/write conflicts between concurrent threads. Since hardware support is not available, carefully designed patterns in the GPU programs are imperative to ensure correctness. However, newer generation GPUs have added support for programming with atomic operations, which may alleviate this problem in the near future. Nevertheless, in the context of spatial databases, most datasets (such as land information, maps etc.) rarely get updated and therefore can be assumed to be mostly read-only. Therefore, this problem may not affect spatial query processing significantly.

c. In many cases, the GPU can be used solely as a co-processor for speeding up some of the heavy computations in a large pre-existing CPU-side application. Our goal is to implement a full-fledged spatial DBMS in which most of the database internals run on the CPU, while only the heavyweight spatial processing is re-written and offloaded to the GPU. However, as noted in Section 3, state-of-the-art GPU programming frameworks lack support for valuable programming concepts (e.g., recursion, function pointers). Additionally, device functions (GPU-side) cannot invoke host functions (CPU-side), which means that in some cases there is no way to avoid re-implementing on the GPU some of the subroutines that are already available and well-optimized for the CPU. These issues can hinder the programming task considerably. Fortunately, newer architectures progressively address these problems.

d. Data skew can be a significant problem in any large parallel system. Spatial data in particular has an inherently large skew, since spatial objects have variable size and thus geometric computations on spatial data do not take the same amount of processing time. One must strive to balance the workload to leverage the parallel processing power efficiently. In our work, we assume a fairly homogeneous dataset where shapes of the same type do not exhibit huge variations in size. As future work, we shall attempt to address this problem for datasets with considerable skew.

e. Perhaps the most important limitation of a GPU with respect to large spatial databases is the rather limited amount of global memory. The maximum global memory on some of the newer high-end cards is around 1-3GB. Although this is sufficient for a reasonable size dataset of a few million records, it may not be enough for larger datasets. The global memory size constrains the amount of data that can be present in GPU memory at query execution time. In our experiments, the artificial datasets used fit entirely in GPU memory (except for Query 6), without needing to perform multiple consecutive transfers when executing a spatial query that touches the entire dataset. A realistic spatial dataset that fits into GPU memory is possible, given the rapid growth in graphics card capabilities and the fact that vector data is much more compact than raster data due to its internal representation of geographic features.

However, we considered a few solutions when the spatial dataset cannot be fully stored into GPU memory. One option would be to make use of the zero-copy functionality in CUDA [7]. Unfortunately, this is highly impractical for two reasons: first, this feature has prohibitively low bandwidth, and second, it requires memory to be declared as pinned, and pinned memory is limited both by the GPU and the operating system itself to much less than 4GB. Consequently, our solution was to partition large datasets that did not fit in the GPU memory into chunks and transfer them in stages between the host and the device during the execution of a spatial query. Based on our experiments, we observed that even in this case, the transfer time can be easily mitigated, since the component transfer times for the individual data chunks can be compensated by considerable speedups when processing each individual partition.

In the future, we would like to explore the use of data compression. In addition to reducing the memory footprint on the GPU, compression also could potentially reduce the transfer time from CPU to GPU memory, thus bringing an increased benefit. Fang et al. [8] showed that partial data compression and decompression on the GPU is fast, reduces the memory footprint and reduces the transfer time overhead.

6. Related work

Although many works [9, 10, 11, 12] have focused on spatial and non-spatial query processing on the GPU, they mostly develop OpenGL/DirectX programs to take advantage of video memory and accelerate various query operations. The GPU processing pipelines in DirectX and OpenGL are generally slow, and the implementations contain too much graphics-related overhead (e.g., aside from the overhead of copying data to video memory, there is an overhead associated with texture coding and decoding operations). Nevertheless, these works underline the potential of using GPUs for speeding up relational operations.

The work of He et al.[13] is perhaps the first attempt to leverage the highly parallel GPU hardware, high memory bandwidth and low-latency local memory to implement relational join algorithms. Their main goal is to implement a set of four relational join algorithms: non-indexed and indexed nested-loop joins (NINLJ and INLJ), sort-merge join (SMJ) and hash join (HJ), on a GPU. Several special functions (e.g., map, split, scatter, gather, sort etc.), are implemented as kernels on the GPU, to serve as primitives for implementing the relational join operations. Their implementation takes advantage of several key features of modern GPUs: the massive thread parallelism, fast interprocessor communication through local memory and coalesced memory accesses, and manages to hide the latency of memory stalls. Overall, the speedup over the CPU counterparts is 2-7x, across the 4 types of join that were implemented.

In a later work [14], the same authors propose GDB, an in-memory relational query co-processing system, using the GPU implementation from the previous paper. They propose a cost model for the query processing time and the transfer time between the CPU and GPU. This cost model serves to determine whether a query is worth being executed on the GPU, or if it is better off being run on the CPU to avoid a high memory transfer cost that cannot be compensated by the GPU speedup gains. They show that the GPU-based query processing algorithms are 2-27x faster than the optimized CPU-equivalents, on in-memory data. Overall, the hybrid co-processing system is similar to, or better, than both the GPU-only and the CPU-only query processing implementations. Furthermore, in order to reduce the memory usage on the GPU and alleviate the overhead of data transfers between CPU and GPU memory, a later work [8] proposes compression and shows that although full compression/decompression is expensive, partial decompression succeeds in improving GPU-processing performance.

This body of works [13, 14, 8], only address traditional databases, without dealing with spatial functionality. Spatial query execution is quite different (as discussed in Section 2.1) and generates different query plans. Also, spatial queries involve complex geometric computations, which can put a different type of pressure on the CPU than traditional database workloads. We address the area of spatial processing by showing the potential of GPUs in speeding up spatial query execution.

Finally, most related works implement a set of algorithms for relational operations on the GPU without implementing a complete database engine, and tests are conducted with synthetic datasets, similar to our own work. In the future, we aim to implement a full-fledged DBMS with GPU co-processing capabilities, that offloads most of the heavy processing to the GPU, since our initial results demonstrate great potential for spatial query execution on GPUs.

7. Conclusions and future work

We have shown that the execution of geospatial queries can significantly benefit from processing on a GPU, because of their inherent high degree of parallelism. We tested a set of common spatial queries and showed that GPU execution time is a few orders of magnitude faster (62x - 318x) than the corresponding CPU execution, on considerably large datasets.

One of the major drawbacks associated with GPU execution is the cost of memory transfer from host to device. We present results showing that the transfer cost is easily amortized by the query execution speedups. Even when running a single query on the GPU, we still achieve an 8x speedup on average for simple queries. Furthermore, for a long-running query (i.e., an all-pair join), the transfer time becomes negligible compared to the query processing time, thus rendering a 62-240x speedup. Our results clearly show the potential for massive performance gains.

In the future, we would like to explore the concurrent copy-execute feature in CUDA to potentially improve performance even further. This feature provides the ability to overlap kernel execution and asynchronous data transfers between host and device, thus alleviating some of the memory transfer overhead.

Another future extension to this work would be to analyze the benefit of processing multiple queries per GPU kernel. Since both the spatial stage and the aggregation stage are performed in parallel on a large dataset and given that the GPU threads basically process a fixed set of tuples from the relation, it may be possible to run multiple query instances in parallel on the GPU. and what bottlenecks or restrictions we might encounter.

Finally, our ultimate objective is to develop a GPU-enabled spatial DBMS, to demonstrate that even a highly-complex system can leverage the massive parallelism of the GPU in its query processing engine. We expect we will need to address certain limitations of current GPU programming frameworks, but this task should become more feasible with newer cards starting to support features such as function pointers and recursion, as well as double-precision floating point types.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, D. A. Wood, DBMSs on a Modern Processor: Where Does Time Go?, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 1999, pp. 266–277.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum, 2007 26 (1) (2007) 80–113.

[3] S. Ravada, J. Sharma, Trends in Spatial Databases, in Urban Planning and Development Applications of GIS, American Society of Civil Engineers, 2000.

[4] E. Colak, Portable high-performance indexing for vector product format spatial databases, M.S. Thesis, Airforce Institute of Technology, 2002.

[5] P. Bakkum, K. Skadron, Accelerating SQL database operations on a GPU with CUDA, in: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU, 2010, pp. 94–103.

[6] TIGER®, TIGER/Line® and TIGER®-Related Products, http://www.census.gov/geo/www/tiger.

[7] Nvidia GTC09, http://www.nvidia.com/content/GTC/documents/1122_GTC09.pdf (pages 25-26).

[8] W. Fang, B. He, Q. Luo, Database compression on graphics processors, Proceedings of the VLDB Endowment 3 (2010) 670–680.

[9] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, D. Manocha, Fast computation of database operations using graphics processors, in: Proceedings of the International Conference on Management of Data, SIGMOD, 2004, pp. 215–226.

[10] N. Govindaraju, J. Gray, R. Kumar, D. Manocha, GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management, in: Proceedings of the International conference on Management of Data, SIGMOD, 2006, pp. 325–336.

[11] C. Sun, D. Agrawal, A. El Abbadi, Hardware acceleration for spatial selections and joins, in: Proceedings of the International Conference on Management of Data, SIGMOD, 2003, pp. 455–466.

[12] N. Bandi, C. Sun, D. Agrawal, A. El Abbadi, Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 2004, pp. 1021–1032.

[13] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. Sander, Relational joins on graphics processors, in: Proceedings of the International conference on Management of Data, SIGMOD, 2008, pp. 511–524.

[14] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, P. V. Sander, Relational query coprocessing on graphics processors, ACM Transactions on Database Systems 34 (2009) 21:1–21:39.