Efficient And Scalable Evaluation Of Continuous, Spatio-temporal Queries In Mobile Computing Environments

Jonathan M. Cazalas
University of Central Florida

EFFICIENT AND SCALABLE EVALUATION OF CONTINUOUS, SPATIO-TEMPORAL QUERIES IN MOBILE COMPUTING ENVIRONMENTS

by

JONATHAN M. CAZALAS
B.S. University of Central Florida, 2006
M.S. University of Central Florida, 2009

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2012

Major Professor: Ratan Guha

ABSTRACT

A variety of research exists for the processing of continuous queries in large, mobile environments. Each method tries, in its own way, to address the computational bottleneck of constantly processing so many queries. For this research, we present a two-pronged approach at addressing this problem. Firstly, we introduce an efficient and scalable system for monitoring traditional, continuous queries by leveraging the parallel processing capability of the Graphics Processing Unit. We examine a naive CPU-based solution for continuous range-monitoring queries, and we then extend this system using the GPU. Additionally, with mobile communication devices becoming commodity, location-based services will become ubiquitous. To cope with the very high intensity of location-based queries, we propose a view oriented approach of the location database, thereby reducing computation costs by exploiting computation sharing amongst queries requiring the same view. Our studies show that by exploiting the parallel processing power of the GPU, we are able to significantly scale the number of mobile objects, while maintaining an acceptable level of performance.

Our second approach was to view this research problem as one belonging to the domain of data streams. Several works have convincingly argued that the two research fields of spatio-temporal data streams and the management of moving objects can naturally come together. [IlMI10, ChFr03, MoXA04] For example, the output of a GPS receiver, monitoring the position of a mobile object, is viewed as a data stream of location updates. This data stream of location updates, along with those from the plausibly many other mobile objects, is received at a centralized server, which processes the streams upon arrival, effectively updating the answers to the currently active queries in real time.

For this second approach, we present GEDS, a scalable, Graphics Processing Unit (GPU)-based framework for the evaluation of continuous spatio-temporal queries over spatio-temporal data streams. Specifically, GEDS employs the computation sharing and parallel processing paradigms to deliver scalability in the evaluation of continuous, spatio-temporal range queries and continuous, spatio-temporal kNN queries. The GEDS framework utilizes the parallel processing capability of the GPU, a stream processor by trade, to handle the computation required in this application. Experimental evaluation shows promising performance and shows the scalability and efficacy of GEDS in spatio-temporal data streaming environments. Additional performance studies demonstrate that, even in light of the costs associated with memory transfers, the parallel processing power provided by GEDS clearly counters and outweighs any associated costs.

Finally, in an effort to move beyond the analysis of specific algorithms over the GEDS framework, we take a broader approach in our analysis of GPU computing. What algorithms are appropriate for the GPU? What types of applications can benefit from the parallel and stream processing power of the GPU? And can we identify a class of algorithms that are best suited for GPU computing? To answer these questions, we develop an abstract performance model, detailing the relationship between the CPU and the GPU. From this model, we are able to extrapolate a list of attributes common to successful GPU-based applications, thereby providing insight into which algorithms and applications are best suited for the GPU and also providing an estimated theoretical speedup for said GPU-based applications.

To my wife and children

ACKNOWLEDGEMENTS

Verily, all praise belongs to Allaah, the Most High. We praise Him, we seek His help, and we seek His forgiveness. Indeed, without the blessings and guidance from our Lord, mankind would be lost. I have been tremendously blessed in my life, and every good that I have achieved has been a generous bounty from Him. As such, my utmost gratitude and thanks are to my Lord and Creator.

Words cannot express the thanks that I owe my esteemed advisor, Professor Ratan Guha. He has been a constant source of encouragement in my journey towards my doctorate degree. Dr. Guha has a thirst for knowledge and research that he seems to pass along to all his students. His contributions, detailed comments, and extensive research experience have proved invaluable. He helped shape me into the kind of researcher and educator that I hope to continue to be over the rest of my life. I would also like to thank the other members of my dissertation committee, Professor Mostafa Bassiouni, Dr. Ali Orooji, and Professor Haitham Al-Deek, who monitored my work, provided valuable feedback, and took effort in reading this dissertation.

I would like to express my deepest gratitude and love towards my late parents, Peter and Nancy Cazalas. I wish they could be here, for they would be very proud. They taught me the value of hard work and dedication and instilled in me a passion for learning and achieving my goals. It was always a dream of theirs to see me receiving a doctorate, and they would always beam with pride as the spoke of their “PhD” son. I love them, miss them, and can only hope for opportunities to honor them in the future.

I would like to thank my wife for her love and support. She enjoys reminding me that I’ve been a perpetual student ever since we met. (Perhaps we shouldn’t tell her that Instructors and Researches will always be perpetual students, even after graduation!) She has been a constant source of support, taking care of the home, raising our children, homeschooling them both, and dealing with a litany of other responsibilities. In short, I am extremely grateful for her love, patience, and encouragement. On this same note, I would also like to thank the loves of my life, my children, for their patience. Too often, they simply wanted to play with their "baba", only to find that I was busy with school or business. One day, they will hopefully realize the obvious: this PhD was for them and their future. I thank my children for being amazing children, and I love them unconditionally. I would also like to thank my mother-in-law, Nuzhat Khan, for simply being there for my wife and children. It has truly been a blessing to have her stay with us, and while I do miss my children during the long hours at school, it comforts me knowing their "Anna" is at home, spending time with them in my absence.

Finally, I would also like to thank the Department of Electrical Engineering & Computer Science at the University of Central Florida for funding me throughout my PhD career. It does not escape me that I've been uniquely fortunate in this regard, and I am grateful for the opportunity. I am specifically grateful for the opportunity to teach within the department for the last two years, as the experience simply solidified my desire to finish my degree and pursue teaching and research for a career.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture – a parallel computing architecture developed by Nvidia
FLOPS	FLoating point OPerations per Second
GEDS	GPU Execution of Continuous Queries in Spatio-Temporal Data Streams
GPGPU	General Purpose Computation on Graphics Processing Units
GPS	Global Positioning System
GPU	Graphics Processing Unit
kNN	k Nearest Neighbors
LBS	Location Based Service
SM	Streaming Multiprocessor

CHAPTER 1: INTRODUCTION

The worldwide proliferation of GPS-enabled devices, coupled with the ever-increasing use of smart phones [ChCZ09], has helped to create an environment where data access truly is anywhere at any time. Advancements in wireless technology have allowed the wireless communication market to grow by leaps and bounds. These new technologies bring a flood of opportunities and, hence, applications for moving object databases, including, but not limited to, the ability of rescuers, in natural disasters, to mark locations of found survivors, and alert rescue teams to move in; an individual commuting to work may want to search for nearby taxis as they are walking down the street; similarly, traffic management systems, upon finding dense traffic, may want to send messages alerting travelers and suggesting alternative routes. For these applications, it is not only desirable, but often critical, to perform real-time updates and provide them to the respective users [CaHu06]. An example of such an application, often referred to as a Location Based Service (LBS), is shown in Figure 1.

1.1 Location-Dependent Query Processing in Location-based Services

Location based services (LBSs) are information services accessed via a mobile device that make use of the internet, as well as global positioning systems, to provide value-added services to the user. LBSs can be subdivided into three main categories based on the mobility of the client and the object the client querying: mobile clients that query static objects (e.g. searching for the nearest coffee house while driving through a new town), static clients that query mobile objects (e.g. traffic management systems), and mobile clients that query mobile objects (e.g. pedestrian searching for a taxi). Throughout this work, when we say query, we are not referring to a conventional range query, which is simply a snapshot at some point in time; rather, we are referring to a continuous query, which is executed continuously until specifically terminated by the user. For example, the individual searching for a taxi would ideally use a continuous query that constantly refreshes itself based on his/her movement and that of the taxis. The processing and monitoring of these continuous queries is quite challenging and has been an area of intense research recently [CaHu06, GeLi06, SPPD08, WoIU08].

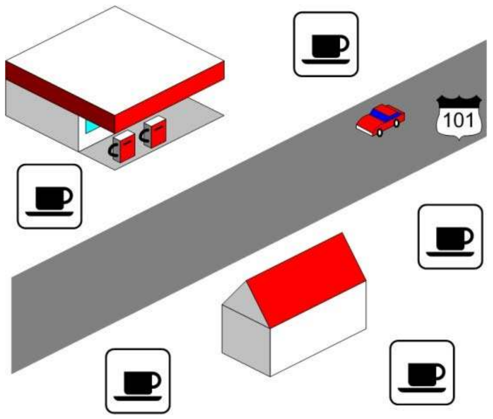


Figure 1: Example of LBS. A traveler searches for the nearest coffee house.

In order to reduce the computational workload, some of the solutions developed make assumptions about the trajectory of the moving objects, such as Domino, by Wolfson et al. [WJSC99, WSCY99, WCKY02], and CAT, by Trajcevski et al. [TWHC04] Other solutions developed do not make this assumption but do require cooperation from the mobile objects, such as MQM by Cai et al. [CaHu06, CaHu02] and MobiEyes by Gedik et al. [GeLi06] Additionally, other research has focused on the computation of predictive nearest neighbor queries [TaPS03] as well as continuous nearest neighbor queries [MPBT05, MoHP05]. Finally, other research has utilized ideas from both of these models and has focused the study on range queries in road networks. [BCMK10, SPPD07]

Each of these solutions has proven to be effective and indeed has advanced the research within this area. However, we find a scalability limitation with each of the aforementioned solutions. No matter which method one uses, whether it be trajectory-based, requiring cooperation of the mobile object, or other similar methods, we end up with a computational bottleneck when we try to scale the number of mobile objects. We propose to address this limitation in two ways:

1. exploit computation sharing to reduce computation cost, and
2. leverage the parallel processing capability of the Graphics Processing Unit (GPU) to significantly speed up computation.

Additionally, with the rapid increase in mobile devices and the expected growth of the industry, the number of concurrent queries an LBS needs to support in the near future will be enormous. As discussed in previous research [CaHu09], it might be advantageous to pre-compute some views of the database to better support popular queries. In addition, using a materialized view is a form of computation sharing among queries that are based on this view. In this work, we explore this view concept for moving object databases. In particular, we consider *Proximity Area*, a materialized view over the raw location database that pre-computes and stores the result of the range query for each moving object in the location database. Such a view is useful for more complex queries, such as finding out if objects A, B, and C are in proximity. This approach can significantly reduce communication cost, as well as response time, for queries based on the view, a desirable capability when supporting a very large user community with enormous number of concurrent queries.

Using traditional techniques, maintaining such a view would be computationally prohibitive. We propose to offload computation onto the GPU, which serves as a Range Processor that constantly performs range queries over all mobile objects in the database, thereby allowing us to materialize the *Proximity Area* view. Although this seems simple enough, the challenge in GPU computing is in the algorithmic design. While some algorithms are very easy to write on the CPU, their GPU counterparts are often several orders of magnitude more difficult.

1.1.1 Contributions

In general, the contributions of this first work can be summarized as follows:

1. We propose to leverage the parallel processing capability of the GPU as a solution to the scalability limitation of previous methods.
2. We propose a view oriented approach of the location database, thereby reducing computation costs by exploiting computation sharing amongst queries requiring the same view.
3. We provide statistical analysis of our proposed approach, as compared to its CPU-based counterpart, thereby showing the efficacy of using a GPU for location-dependent query processing.

1.2 GEDS: GPU Execution of Continuous Queries in Spatio-Temporal Data Streams

Traditionally, mobile object databases augment the standard database model of persistent data storage and complex querying by adding new models and index structures geared to store, track, and process the locations of moving objects efficiently. [Gut84, BKSS90, CaHu06, GeLi06, WSCY99] R-trees [Gut84] have been the most popular mechanism for spatial indexing, and many variants have been proposed, including the R*-tree [BKSS90], X-tree [BeKK96], Lazy Update R-tree [KwLL02], and a plethora of other suggestions. As briefly highlighted in the previous section, there is a large body of research focused on reducing the computational burden of continuously monitoring and evaluating real-time queries over these mobile objects. Such works include MQM [CaHu06], MobiEyes [GeLi06], Domino [WJSC99, WSCY99], and CAT [TWHC04], to name a few. While these models and structures did initially extend the research in this area, the past few years have witnessed the emergence of a new class of data intensive applications that often require the continuous processing of potentially unbounded sequences of transient data, called data streams. [KrSe09] Examples include financial tickers, internet traffic, sensor data, and transaction logs. Unfortunately, the massive data sizes of these spatio-temporal data streams, along with their respective high arrival rates, makes it infeasible for traditional DBMS techniques to store, query, or index these streams and therefore dictates the need for better solutions. [MoAr08]

In simplest terms, a data stream can be defined as “a sequence of characters or bits that is too large to be viewed in its entirety.” [HaWa90] Several works have convincingly argued that the two research fields of spatio-temporal data streams and the management of moving objects can naturally come together. [IlMI10, ChFr03, MoXA04] For example, the output of a GPS receiver, monitoring the position of a mobile object, is viewed as a data stream of location updates. This data stream of location updates, along with those from the plausibly many other mobile objects, is received at a centralized server, which processes the streams upon arrival, effectively updating the answers to the currently active queries in real time. From this model, it becomes clear that additional applications could benefit from modeling location updates as streaming data, including, but not limited to, network traffic, time series data, telephone records, weather data, web click streams, and the list goes on.

Unfortunately, most of the recent research in data stream management systems [ACCC03, BaWi01, ChFr03] is insufficient, as they overlook the spatial and temporal qualities of both the data streams and the continuous queries over these streams. [MoAr08] And it is these two qualities, specifically, that distinguish continuous query processing in spatio-temporal data streams from traditional data streams. Because both queries and data can continuously change their locations, spatio-temporal data streams are viewed as a series of location updates as opposed to the append-only model of classical data streams. [MoAr08] Additionally, the temporal quality stipulates that a mobile object may be added to or removed from the result set of the spatio-temporal query, an example being GPS-equipped vehicles moving in and out of a query region. Because these queries are continuous in nature, any delay would result in an obsolete response. Therefore, it is vital to procure scalable and efficient algorithms for the processing of continuous spatio-temporal queries over data streams. To this end, SINA was proposed to address this issue by exploiting shared execution and incremental evaluation. [MoXA04] The main drawback of SINA was the reliance on physical disk-storage to perform its operations. SOLE was then proposed as a scalable, in-memory algorithm, which uses an incremental evaluation paradigm and a grid structure to evaluate concurrent, continuous spatio-temporal queries over data streams. [MoAr08] SOLE avoids the slow, physical data storage, but is also limited based on memory. As a result, a load-shedding algorithm is applied resulting in the expulsion of certain objects from memory and causing uncertainty. Finally, other methods focus on efficient evaluation of sliding window queries. [KrSe09, GHMA07]

Each of the aforementioned solutions has proven to be effective and has indeed advanced the research in this area. However, all of the solutions are bottlenecked by virtue of using an ill-equipped processor, namely a CPU. Trying to solve a data streaming problem without using a stream processor is analogous to mowing your lawn with a pair of scissors; sure it will work, but the CPU effectively handicaps any proposed solutions. We are trying to process spatio-temporal queries over data streams, and this environment shouts out the need for a Graphics Processing Unit (GPU), a stream processor by trade. [ACCC03] The GPU is uniquely suited to perform the necessary computation required by the concurrent, continuous spatio-temporal queries over spatio-temporal data streams. We propose GEDS, a scalable, GPU-based framework that employs the computation sharing and parallel processing paradigms to deliver scalability in the evaluation of continuous spatio-temporal queries over spatio-temporal data streams. Specifically, we leverage the parallel processing capability of the GPU to significantly speed up computation in the GEDS framework.

Additionally, with the ever-increasing use of GPS-enabled devices, LBSs will be forced to support an enormous amount of queries. As discussed in the previous section, it can be advantageous to pre-compute certain views over the streaming data to better support popular queries. So for the purpose of facilitating this research and demonstrating the efficacy of GEDS in data streaming environments, we propose to pre-compute views over this streaming data. Specifically, we consider two materialized views over the incoming data stream of location updates: *Proximity Area* and *Neighboring Objects*. The *Proximity Area* view pre-computes and temporarily stores the result of a range query for each moving object in the data stream, while *Neighboring Objects* pre-computes the result of a k-nearest neighbors (kNN) query. Both the *Proximity Area* and the *Neighboring Objects* views are evaluated in real-time and sent back as a result data stream. Since the GPU is not handicapped in the processing of these queries/views, GEDS does not suffer from the uncertainty of previous methods.

An illustration of the GEDS framework is shown in Figure 2. The output from each of the GPS-enabled devices is viewed as a data stream of location updates coming into the GEDS framework. The location updates are processed in-memory and sent to the GPU for query evaluation. The materialized *Proximity Area* view (query results) is then outputted as a data stream.

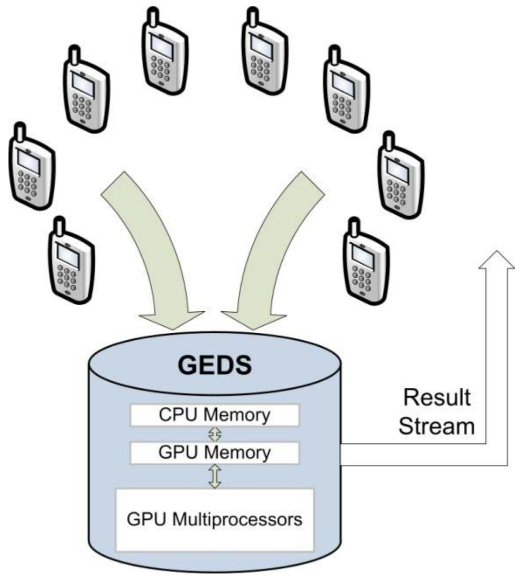


Figure 2: Illustration of GEDS Framework

Finally, GEDS is not meant as a comprehensive, all-inclusive framework, thereby attempting to incorporate the best of SINA or SOLE and the best new query algebra for views of data streams [GELA10]. Rather, the focus of this study is on the plausible use of the GPU in the spatio-temporal data streaming environment. Other works can take advantage of GEDS by simply mapping their algorithm onto the GEDS framework.

1.2.1 Contributions

In general, the contributions of this work can be summarized as follows:

1. We propose GEDS as the first attempt at using a stream processor (GPU) to evaluate concurrent, continuous spatio-temporal queries over spatio-temporal data streams.
2. We develop novel GPU-based algorithms that allow us to materialize queries over the GPU-based framework.
3. We show the adaptability of the GEDS framework by extending is functionality to solve continuous, spatio-temporal kNN queries
4. We demonstrate that the parallel processing capability of GEDS provides significant speedup and facilitates the accurate evaluation of the spatio-temporal *Proximity Area* and *Neighboring Object* views, thereby eliminating any uncertainty of previous methods.
5. We provide statistical analysis of the GEDS framework, as compared to its CPU-based counterpart, thereby showing the efficacy of GEDS in spatio-temporal data

streaming environments, even as the costs of CPU-GPU memory transfers are taken into account

1.3 Performance Modeling of Spatio-Temporal Algorithms over GEDS Framework

Finally, while our GEDS framework was clearly a success, we also take a broader approach in our analysis of GPU computing. What algorithms are appropriate for the GPU? What types of applications can benefit from the parallel and stream processing power of the GPU? And can we identify a class of algorithms that are best suited for GPU computing? To answer these questions, we developed an abstract performance model, detailing the relationship of the CPU and the GPU. To gauge the efficacy of our model, we then run a variety of simulations, comparing the actual run-time to the model-based, theoretical run-time. From this model, we are then able to extrapolate a list of attributes common to successful GPU-based applications, thereby providing insight into which algorithms and applications are best suited for the GPU and also allowing the user to gauge approximate performance increases.

1.4 Organization of Dissertation

The remainder of the work is organized as follows. Chapter 2 reviews related work on both research domains: traditional querying of mobile objects in Location-based Services and spatio-temporal queries over spatio-temporal data streams. Chapter 3 discusses Superscalar Computing, the computational benefits of the GPU, and NVidia's CUDA. In Chapters 4 and 5, we discuss our proposed techniques as well as performance statistics. Chapter 6 provides additional analysis of the GPU and the impact on speedup caused by memory transfers. In Chapter 7, we describe our abstract performance model. Finally, we conclude this dissertation in Chapter 8 by summarizing the advantages of our GPU-based framework and by outlining future research directions.

CHAPTER 2: RELATED WORK

In this chapter, we present previous work done in traditional location-dependent query processing and then focus on the previous works in the domain of spatio-temporal queries over spatio-temporal data streams.

2.1 Location Dependent Query Models in Location-Based Services

There is a large body or research on continuous queries, with each study focused on reducing the computational burden of continuously monitoring and evaluating these queries. The work on monitoring and evaluating continuous queries can be classified into two main categories: (1) assume known movement trajectories of the objects, or (2) make no assumptions on the movement patterns, but do require cooperation from the mobile objects. In an effort to avoid continuous location updates, Wolfson et al. [WCKY02, WJSC99, WSCY99] proposed DOMINO, which introduced the idea of dynamic attributes. These attributes, which include initial time, initial position, and a function that evaluates the objects future position at some new time, are reported to the server. As such, the object only needs to report its position once the difference between its actual location and the computed location (based on the function) exceeds some threshold. Trajcevski et al. [TWHC04] proposed CAT (Correct Answers of continuous queries using Triggers), which uses triggers to determine when to reevaluate continuous queries, the idea being that there is no need to reevaluate a query if nothing has changed. Stojanovic and Predic propose ARGONAUT, a framework to provide moving object data management via

HTTP/SOAP [PrSt05, SPPD08]. Each of these solutions relies on trajectories to limit location updates. This immediately puts a limitation on these methods, as not all objects follow trajectories; some objects (e.g. people) more freely.

The second main category makes no assumptions regarding movement patterns; it does however require cooperation from the mobile devices. If the objects are stationary, Zhang et al. [ZZPT03] suggests that the current location is returned as well as a "validity scope" where the result will remain the same. Hence, queries are only reevaluated when they exit the validity scope. Extensive studies have been conducted on continuous monitoring of moving objects. Prabhakar et al. proposed a Q-index technique, which indexes queries using a structure similar to an R-tree. [PXKA00, PXKA02]. This method also uses the idea of a safe region, defined as a region containing the object's current location that does not overlap with any of the query boundaries. Although this method did reduce location updates, determining safe regions often required intensive computation.

Cai et al. [CaHu02, CaHu06] proposed MQM, which focused on the problem of continuous static range queries over mobile objects. In MQM, the query processing is distributed over the moving objects by having each object monitor the query regions it affects. They propose the idea of a resident domain and develop a spatial access method called BP-tree. Each object is assigned a resident domain and is notified of the queries inside it. The mobile object then monitors its spatial relationship with the queries present in its resident domain, and upon crossing a query boundary, the mobile object contacts the server to update any affected query results. An object must also monitor its movement against its resident domain, as once it exits the resident domain, a new resident domain must be determined. To address this problem, the database domain is dynamically partitioned into many disjoint subdomains. The areas of a query that overlap with a subdomain are called monitoring regions. This partitioning and query decomposition allows one or more subdomains to be used as the object's resident domain, "as long as the number of monitoring regions inside it does not exceed the processing capability of the mobile object." [CaHu02] The disadvantage to this approach is that it is not completely distributed; all moving objects must communicate with a centralized server, which not only represents a single point of failure but also could be a communication bottleneck.

Gedik et al. [GeLi06] proposed MobiEyes, which is similar to MQM. It partitions the domain with a grid, and every query is then associated with a monitoring region. A query's location and speed are sent to all objects within its monitoring region. An object must then notify the server when it enters or exits the query's region. Mokbel et al. [MoXA04] proposed a hash-based algorithm (SINA) suitable for both range and kNN queries. Kalashnikov et al. [KaPH04] proposed a main memory evaluation of query objects. Yu et al. [YuPK05] made a similar proposal for the evaluation of kNN queries. Additionally, Yuen et al. [WoIU08] proposed a distributed version of MQM called DMQM.

2.2 <u>kNN Queries in Location-based Services</u>

Along with range queries, the *k*-nearest neighbor (kNN) query is one of the most common queries in database literature. Due to limited computational resources, this classical problem has traditionally focused on static objects; they were simply snapshot, or one-time queries. Most solutions utilized variants of the R-tree [Gut84] to facilitate a spatial indexing of the data. Using a branch-and-bound algorithm, the tree would be traversed, and a list of candidate nearest neighbors is maintained in a priority queue. [HaSa99, RoKV95] The problem, however, was that such methods were incapable of handling continuous kNN queries. Moving towards this direction, Lee at al. [LHJCT03] proposed what they referred to as a bottom-up approach, which allowed them to support frequent location updates in R-trees. In 1999, Kollios *et al.*[KoGT99] published the first work that answered kNN queries for moving objects. While the algorithm was perhaps limited, in that it only worked for a 1D space, and could be extended to 1.5 dimensions, it set the state for several works to come. Saltenis *et al.* [SJLL00] utilized the time-parameterized R-tree (TPR-tree) to represent the mobile objects in 2D and higher dimensional spaces. Several other works [BJKS02, IwSK03, RaPM03, TaPa02] used variants of the TPR-tree as the underlying index structure.

These approaches focused on predictive queries and utilized a basic assumption: the velocity of a mobile object will remain constant until a newly received update indicates otherwise. In order to answer the queries, the proposed algorithms would predict the future trajectory of the object based off of their current position and velocity. The approach used by Bensen *et al.* [BJKS02] was to extend the R*-tree, augmenting it with velocity vectors. The time-parameterized kNN queries could then be answered by a depth-first traversal of the TPR-tree. [YuPK05] Tao *et al.* [TaPa02] also proposed a time-parameterized approach in which conventional kNN algorithms keep track of the next set of mobile objects that may possibly affect the current result set. Unfortunately, this approach was very CPU and I/O intensive, as it required multiple traversals of the TPR-tree. By combining ideas presented in the two previous approaches, Raptopoulou *et al.* [SPTL04] reduced the CPU costs and provides an alternative, faster solution to the processing of kNN queries. However, as pointed out by Sun *et al.*

[SPTL04], TPR-trees are only viable if the future trajectories are known at query time; otherwise, the TPR-tree is too expensive to maintain.

Several works moved beyond the use of trajectories in an attempt to provide more accurate, and realistic, query results. Koudas *et al.* [KOTZ04] proposed methods to provide approximate answers to NN queries over data streams, with a guaranteed error bound. Mouratidis *et al.* [MoHP05] propose conceptual partitioning (CPM), a "comprehensive partitioning technique", which handles location updates only from mobile objects that fall within the vicinity of the given query. In their proposed algorithm, SEA-CNN, Xiong *et al.* [XiMA05] use incremental evaluation and shared execution. They perform a spatial join between the queries and the set of mobile objects, thereby reducing the computation cost to answer the query. Yu *et al.* [YuPK05] also use a grid-based structure, with one indexing objects and the other indexing the queries themselves.

2.3 Spatio-Temporal Queries Over Spatio-Temporal Data Streams

Using concepts from both temporal databases and data streams, Yuang and Jensen proposed one of the first, stream-based frameworks for continuous queries in mobile environments. [HuJe04] As mobile objects move, their location updates are sent to the server as incoming data streams, which are then processed and used in the result set of a location-based query. This answer is then presented to the user as an outgoing data stream. Patroumpas and Sellis proposed STREAM and Telegraph [PaSe04], two frameworks that modeled the continuous movement of objects as trajectory streams, and it was concluded that the data streaming paradigm was indeed powerful enough to effectively manage data about mobile objects. Spatio-temporal objects can be indexed effectively with the TPR*-tree. However, there are no built-in mechanisms allowing for continuous queries in this method.

Sliding windows are a common means of dealing with queries in data streams. Incremental query evaluation was studied in [GHMA07], with the main focus on the input-triggered approach (ITA) and the negative tuples approach (NTA). The authors proposed two wo optimization techniques to enhance the performance of NTA. The first method focuses on the join operator, avoiding the re-evaluation with negative tuples. The second optimization dynamically tunes the query pipeline, allowing it to work in ITA or NTA depending on the tuples flowing in the pipeline. Although sliding windows over data streams has been researched extensively, very little attention had been given towards developing a general purpose stream algebra until [KrSe09]. A semantically sound query language was developed that rivals CQL and stays close to SQL:2003 standards. The problem with sliding window methods is that most do not support spatio-temporal queries that are interested on the current state of the database or data stream.

Perhaps most relevant to our proposed approach, in the context of spatio-temporal data stream processing, are SINA [MoXA04] and SOLE [MoAr08]. Scalable Incremental hash-based Algorithm (SINA) utilizes an incremental evaluation and shared execution paradigm, which is achieved by joining the continuous spatio-temporal queries with the set of mobile objects. [MoXA04] SINA executes in three phases: hashing phase, invalidation phase, and joining phase. An in-memory, hash-based join is used during the hashing phase, resulting in a set of positive updates. The invalidation phase produces a set of negative updates and is triggered every *T* time units or when memory is full. Thirdly, the joining phase produces a set of positive and negative updates that stem from joining the in-disk data with the in-memory data. The immediate drawback of SINA is that it relies on physical disk-storage to perform its operations.

To address this issue, the scalable on-line execution (SOLE) algorithm was proposed. [MoAr08] SOLE is an in-memory algorithm that processes every data input as it is received by the system. Since memory is scarce, SOLE keeps track of only those objects that satisfy at least one active continuous query. This does, however, result in uncertainty regions in which some objects may not be reported in the result set. Similar to SINA, SOLE operates as a spatio-temporal join of the stream of spatio-temporal objects and the stream of spatio-temporal queries. SOLE incorporates a load-shedding approach that dynamically removes any insignificant objects, which allows it to cope with possible high-arrival rates and limited computational ability of the CPU. Finally, SOLE is implemented inside the PLACE server [MXHA05], a prototype data stream management system supporting location-aware environments.

Materialized views are common in many applications based on relational databases, as they are used to simplify the formation of complex queries and often yield a more efficient query execution plan. Several works discuss views in relation to Location Based Services [CaHu09, GELA10], with the latter specifically reviewing views over data streams. Synchronized SQL (SyncSQL) query language was introduced, defining a data stream "as a sequence of modify operations against a relation." [GELA10] As part of a framework that supports views in data stream management systems, SyncSQL expresses composable queries (views) over spatio-temporal data streams.

2.4 Performance Modeling in Parallel Computing Architectures

Most of the work done on performance modeling deals with traditional, parallel computing architectures and often requires detailed analytical models and even source codes for reasonable results. Xu *et al.* [XuZS96] developed a two-level, hierarchical performance model, which used a graphical model at the top level to represent a high-level abstraction of the program. [SaVa08] The running times for individual segments are calculated using a low-level model, which utilizes analytical and experimental values to express the execution time. Also famous are the PACE toolkit [NKPP00] and the POEMS project [AdVe04]. Using portions of the source code and details of the hardware configuration, PACE forms a set of performance model objects using the CHIPS performance model language. From these objects, it can perform predictions on heterogeneous systems. The POEMS project is a comprehensive modeling infrastructure, utilizing a custom specification language, component models, and a database to store task dependencies and performance results. As with the PACE toolkit, POEMS also requires the source code for accurate prediction. [SaVa08]

Grove *et al* [GrCo05] requires the developer to detail the complexity of the serial portions of the program and tries to model a limited kind of non-dedicatedness. Yan *et al.* [YaZS96] propose a "performance prediction model for parallel computing on non-dedicated heterogeneous networks of workstations." They use a two-level model with the top level being a semi-deterministic task graph. They also model application characteristics through the use of Program Execution Graphs (PEGs). This work was extended by using Petri-Net models to characterize application behavior. [Ang98]. Newer works have focused on modeling techniques for predicting execution times in order to efficiently schedule the applications on grid resources (dedicated and non-dedicated clusters). [SaVa08]

While traditional, parallel computing architectures have been around for decades, general purpose computing on graphics processing units (GPGPU) has just recently surged in popularity. To be expected, the performance models are few and most are specific to the underlying applications. However, several works have identified what is, arguably, the main bottleneck in GPU computing: the PCI-Express. Recognizing the limited bandwidth of the PCI Express, these works focus on reducing the data transfers between the CPU and GPU. Owens *et al.* [OHLG08] and Fan *et al.* [FQKY04] propose the rewriting of algorithms to limit data transfers over the PCI-Express. Cohen and Molemaker [CoMo09] and Dotzler *et al.* [DoVK10] also recommend the need for revised algorithms. Schaa and Kaeli [ScKa09] detail a multi-GPU design space and identify data transfers as the central bottleneck.

Other works propose ideas on how to mitigate the limitations of data transfers. In their "Asymmetric Distributed Shared Memory" (ADSM) model, Gelado *et al.* [GSCP10] propose two types of memory updates, lazy and rolling, which determine whether or not the data should be moved on or off the GPU. Their model maintains an asymmetric shared memory space for CPUs to access data (objects) on the GPU, but not vice versa. Al-Kiswany *et al.* [AGSY08] propose a distributed storage system (StoreGPU), which utilizes pinned memory on the CPU, thereby reducing the cost of data transfers. Becchi *et al.* [BBCC10] propose a scheduler that only transfers data when the memory overhead is acceptable. Their approach intercepts function calls to well-known kernels and schedules the kernels to the CPU or GPU based on their argument size and data location. Although there are two underlying memory subsystems, the transparent managing of data placement provides a unified memory view to the programmer. Gregg and Hazelwood [GrHa11] expand this approach by proposing a taxonomy to categorize GPU kernels. Schedulers can then use this taxonomy to determine whether to run a given kernel on the GPU or on the CPU.

CHAPTER 3:
SUPERCOMPUTING AND THE GPU

The 1970's witnessed the emergence of the first microprocessor, the Intel 4004, which had approximately 2300 transistors and incorporated the functionality of a central processing unit (CPU) onto a single microchip, thereby reducing the cost of processing power significantly. Since those humble beginnings, Moore's law [Moor65, Moor75], which states that the number of transistors that can be placed onto a microchip will double every two years, has been uncannily accurate, with new microprocessors having upwards of three billion transistors. Although originally intended to govern the component density on a microchip, Moore's law is indirectly linked to processor speed. The exponential growth of transistors translates to newer chips having more registers, wider data paths, and larger memory caches. As these logic and memory components get packed closer and closer together, processor speed increases simply by virtue of a shorter electrical path. [PaHe93]

For years, this extrapolation of Moore's law onto computing power and processor speed had also been surprisingly accurate, with clock rates growing at a seemingly exponential rate. However, with the arrival of the Intel Pentium 4 in 2002, the first CPU with a clock rate of 3 GHz, modern microprocessors began to reach a limit of three to four gigahertz, simply due to the power constraints and heat problems that accompany these higher frequencies. Thus began a new trend in processor development: multi-core architectures. Instead of fighting a losing battle against clock rates, manufacturers began integrating multiple processors on one chip. Assuming programs take advantage of said multiple cores, this does result in a, at least theoretically, shorter runtime. It is now commonplace to find dual-core and quad-core computers in the average household. Fueled by an insatiable desire for computational power, programmers have also sought out the use of co-processors that are uniquely suited to perform certain tasks.

3.1 General Purpose Computation on Graphics Processing Units

Graphical Processing Units (GPUs) have become as ubiquitous as CPUs, as they are now found in nearly all personal computers and game consoles. Although the GPU has traditionally been used for transforming, rendering, and texturing geometric primitives, GPUs are increasingly being used alongside CPU as co-processors (Figure 3). [GLWL04]

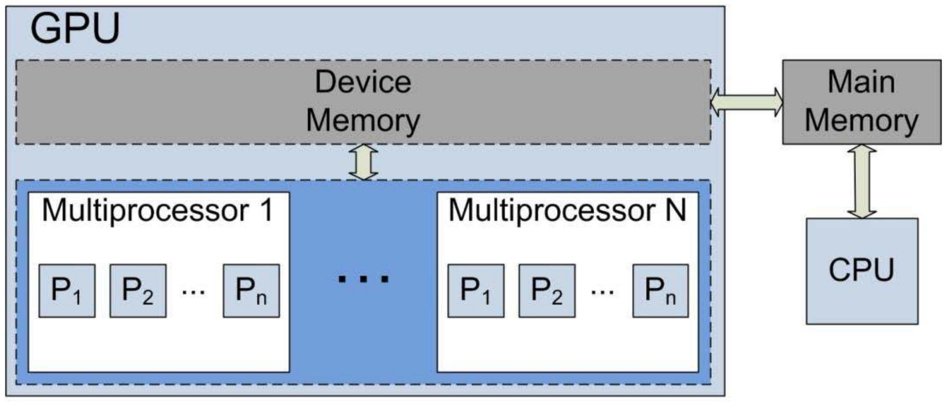


Figure 3: The GPU as a co-processor to the CPU. The GPU consists of several multiprocessors and has a large amount of device memory.

GPUs are extremely fast, with newer models capable of processing more than 1 TFLOP (one trillion floating point operations per second). Compare this to one of Intel's high-end processors, the Intel Core i7 965 XE, which can process just over 70 GFLOPS (10^9 FLOPS), and one can immediately see why programmers are flocking to the GPU (Figure 4). Developers can create algorithms that run on the GPU, theoretically speaking, for almost any computation that works as a stream-computing model.

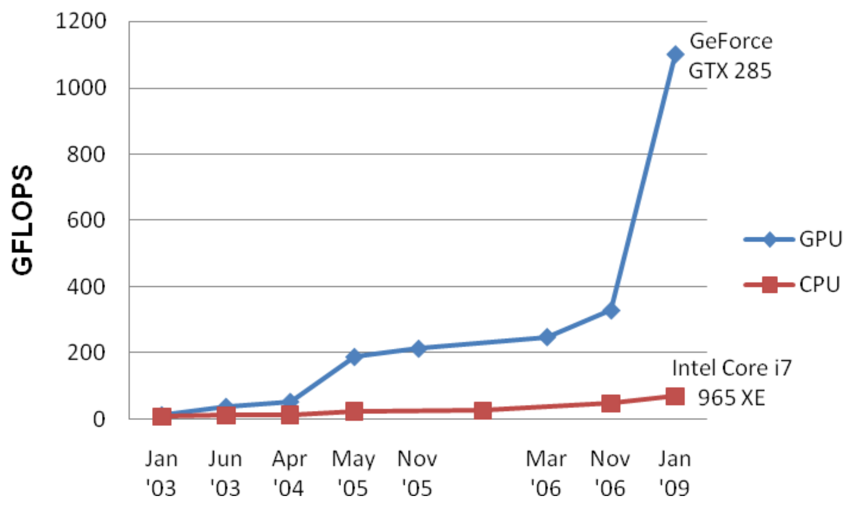


Figure 4: Comparison of floating point operations per second between the GPU and CPU

Historically, general purpose computing on the graphical processing unit, known as GPGPU, was painstakingly difficult. Programmers had to fully understand the GPU pipeline, learn how they could take advantage of certain tests in the pixel processing engine, and then essentially write a program, or hack, that would solve their calculations using the limited functionality of the GPU. Even with these difficulties, researchers were unwavering in their commitment to harness the power of the GPU. Govindaraju et al. [GLWL04] showed that primitive database operations, such as calculating predicates, boolean operations, and aggregates can all be evaluated on the GPU using various algorithms that usually included one or more of the tests on the pixel processing engine. Many other common database operations have successfully been evaluated on the GPU including sorting [GGKM06], similarity joins [LiSS08], and relational joins [HYFL08]. GPGPU quickly gained popularity, and a variety of data parallel algorithms were ported to the GPU. Famous problems such as MRI reconstruction, protein folding, and fluid dynamics achieved remarkable speedups.

3.2 NVidia's CUDA - Compute Unified Device Architecture

As more and more programmers sought new methods of programming on GPUs, companies such as Nvidia chose to open up their API to the programmer, thus allowing for a cleaner, more straightforward functionality. The birth of modern day “GPU Computing” occurred with Nvidia’s development of a closed source language, CUDA, which is a C compiler and set of development tools that allow developers to use the C-programming language to code algorithms on Nvidia’s GPU processors. The initial advantage comes at not having to learn a new language; CUDA is essentially the C-language with some extensions. However, the major advantage is that Nvidia provides a 16KB shared memory region on its GPUs that can be shared amongst threads. Programmers no longer need to hack their way by simulating their data as textures and applying a variety of GPU-specific pixel tests. Developers now how direct access to easily program on this shared area.

CUDA programs usually follow a simple three-step format: (1) copy the data to the CUDA-enabled device, (2) perform the calculation(s), and (3) retrieve the results. The first and third steps are quite straightforward, as we simply use extensions to C-language programming for the copying commands. The calculation section is also straightforward and simply includes the lines of code required to perform the necessary tasks. Twenty or so lines of C-code later, developers now have a C-language program that can calculate their data utilizing the tremendous computational power of the GPU.

3.3 <u>Overview of FERMI Architecture</u>

As GPGPU programming increased in popularity, the demand for innovative technologies grew as well. In 2006, Nvidia introduced the G80 architecture, the first GPU to support C. G80 also introduced the single-instruction, multiple-thread (SIMT) execution model as well as shared memory and barrier synchronization, allowing for inter-thread communication. In 2008, Nvidia's revised unified architecture was introduced, the GT200, which increased the number of streaming processor cores to 240. Additionally, the register file on each processor was doubled in size, facilitating the execution of a greater number of threads on-chip at any given time. While these advances were cutting-edge and met with enthusiasm, Nvidia's Next Generation CUDA Compute Architecture, otherwise known as FERMI (Figure 5), was revolutionary, focusing on the following key areas of improvement: more shared memory, a true cache hierarchy, faster context switching, better double precision performance, ECC support, and faster atomic operations. With these improvements, Nvidia effectively created the world's first computational GPU.

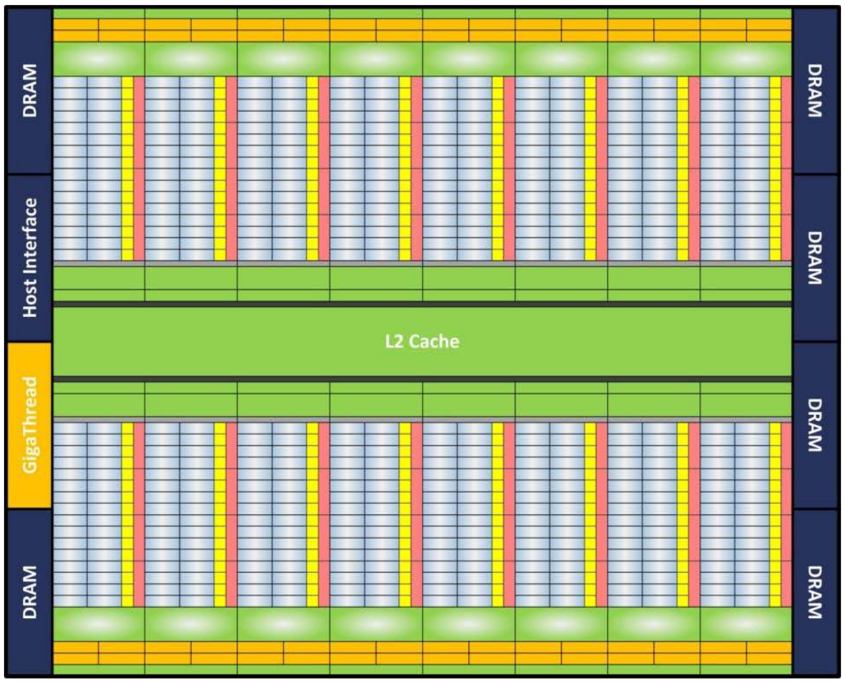


Figure 5: Illustration of FERMI Architecture

Architecturally, FERMI-based GPUs utilize three billion transistors and feature up to 512 CUDA cores, which are organized into 16 streaming multiprocessors of 32 cores each (a four-fold increase over the GT200). Figure 6 shows a detailed view of an individual streaming multiprocessor (SM), along with an exploded view of a single CUDA core. Each of the 32 CUDA cores has a fully pipelined integer arithmetic logic unit and floating point unit, and the ALU supports full 32-bit precision for all instructions. Each SM has sixteen Load/Store units, which allows source and destination addresses to be calculated at sixteen threads per clock. Special instructions, such as sin, cosine, and square root, are executed on one of the four Special Function Units (SFU), built into each SM, at a rate of one instruction per thread, per clock. Fermi also introduced the use of a dual warp scheduler and dispatch unit, allowing two warps to be issued and executed concurrently. The scheduler selects two warps, and then one instruction from each warp is issued to the group of sixteen cores, Load/Store units, or SFUs. Finally, while previous architectures only allowed one kernel to run at a time, Fermi supports up to sixteen concurrent kernels. This dual issue model and concurrent kernel support allows Fermi to achieve near peak hardware performance.

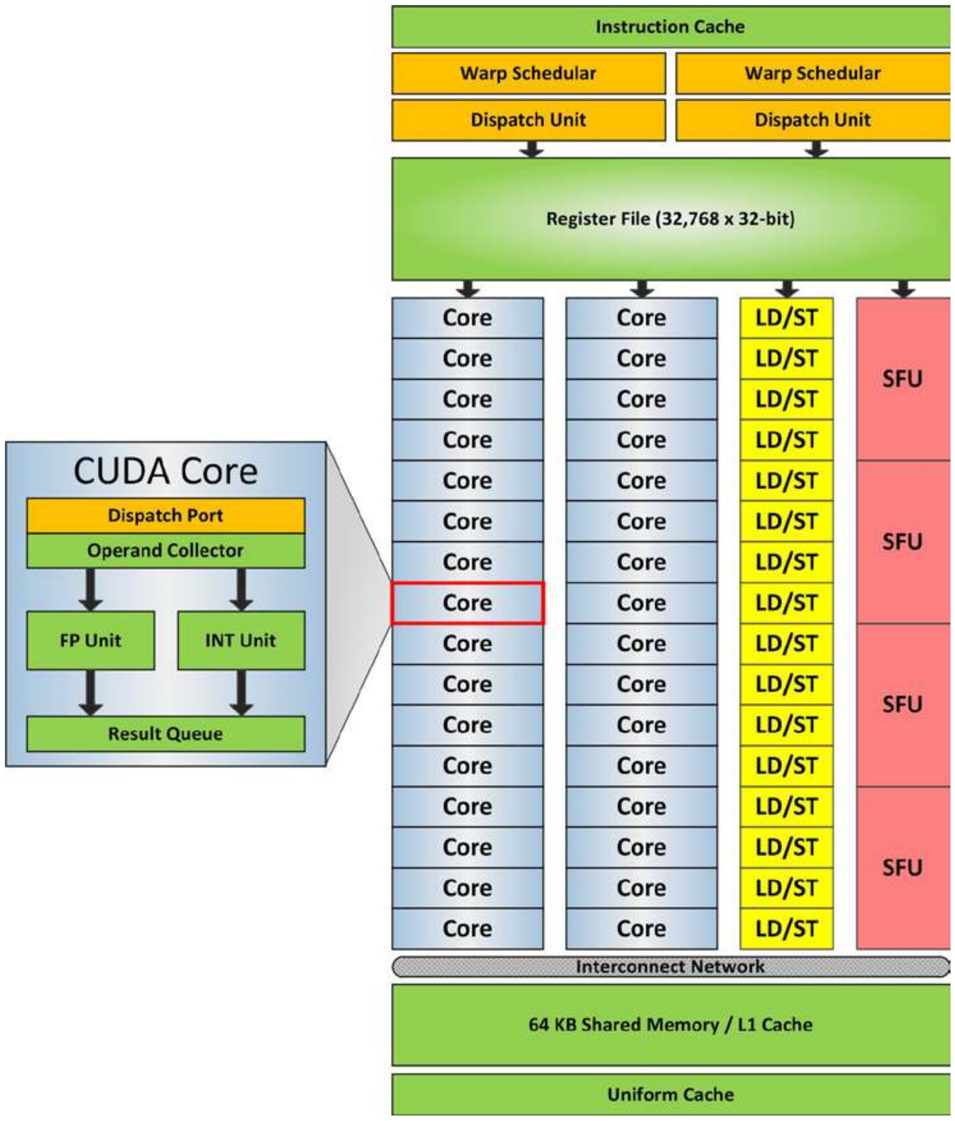


Figure 6: Illustration of a Fermi Streaming Multiprocessor (SM)

While the transistor count and increase in cores is impressive, perhaps the most significant innovation in the Fermi architecture is that of a true cache hierarchy. On-chip shared memory was one of the key architectural innovations in GPU computing, as it improved both the programmability and performance of GPU-based applications. However, shared memory does not necessarily benefit all problems. While some algorithms naturally map to shared memory, others map better to a cache, and others perform best with a combination of both. With Fermi, each SM has 64 KB of memory that can be configured as 48 KB of L1 cache and 16 KB of shared memory, or 16 KB of L1 cache and 48 KB of shared memory (tripling that of the GT200 architecture). Additionally, each Fermi GPU also has a 768 KB, unified L2 cache that services all operations and provides efficient data sharing across the GPU.

CHAPTER 4:
PROPOSED TECHNIQUES

Following the arrangement of Chapter 1 (Introduction), we first detail our proposed techniques for solving location-dependent query processing in the traditional domain of Location Based Services, and then we describe the techniques used proposed for the GEDS framework, which focuses on the data streaming problem in the domain of Spatio-Temporal Data Streams.

4.1 Location Dependent Query Processing in Location-based Services

With the rapid increase in mobile devices and the expected growth of the industry, the number of concurrent queries an LBS needs to support in the near future will be enormous. That is, the number of concurrent queries can be far greater than the number of moving objects in the database. As discussed in previous research [CaHu09], it might be advantageous to pre-compute some views of the database to better support popular queries. Using views is a standard protocol in many applications based on a relational database. These views can be created to simplify the formulation of more complex queries. In addition, using a materialized view is a form of computation sharing among queries that are based on this view. In this work, we explore this view concept for moving object databases. In particular, we consider *Proximity Area* as a view over the raw location database. The *Proximity Area* view can be formally defined as follows:

$$\textit{Proximity Area} = \{[O(i), \{O(j) \mid \text{distance}(O(i),O(j)) < D_{max}\}] \mid O(i) \in O \wedge O(j) \in O\}, \qquad (1)$$

where O is the set of objects in the location database the view is based on and $Dmax$ is the threshold distance. This materialized view pre-computes and stores the result of the range query for each moving object in the location database. Such a view is useful for more complex queries such as finding out if objects A, B, and C are in proximity. This approach can significantly reduce communication cost, as well as response time, for queries based on the view, a desirable capability when supporting a very large user community with enormous number of concurrent queries.

The standard environment for an LBS is illustrated in Figure 7, in which a mobile client sends a query to the query processor, which uses a standard CPU for query execution. The query processor then uses the backend Raw Database (Location Database) to generate results; upon execution, the Query Processor sends the results back to the client.

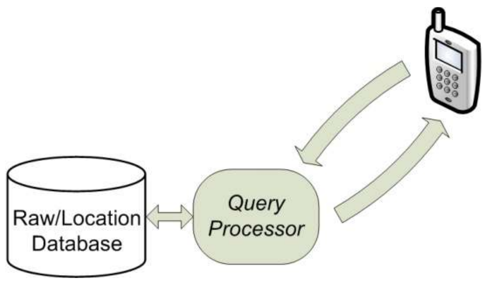


Figure 7: Standard environment for location-based services

The view environment we propose is illustrated in Figure 8, in which the *Proximity Area* view (i.e., the Range Database) is pre-computed to facilitate computation sharing for more efficient processing of end-user queries.

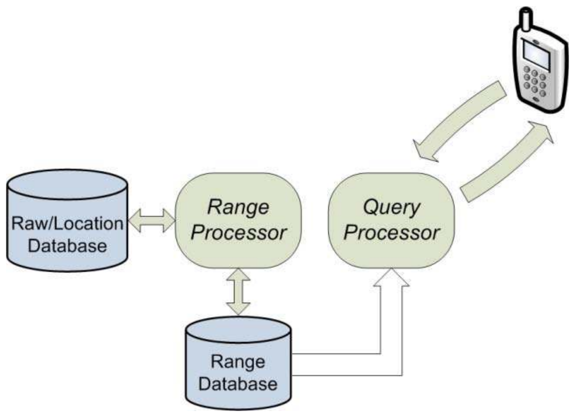


Figure 8: A Location-based service accessing the Proximity Area View

Unlike views in traditional databases, maintaining a view in an LBS is more challenging due to the high frequency of location updates. In this work, we address this issue using a GPU on the central server, thereby allowing for a significant speedup in computation. We note that our solution is orthogonal to distributed computing; we would be able to leverage the GPUs in each of the servers in a distributed environment. Some examples of distributed design for LBSs are presented in [LiHX08, WaZK06].

In this environment, the GPU serves as a Range Processor that constantly performs range queries over all mobile objects in the database to materialize the *Proximity Area* view. This view is stored in a separate database called a Range Database. The Query Processor now reads from this new Range Database instead of the backend Raw/Location Database.

We first develop a CPU-based simulator based on the proposed environment. This simulator quickly reaches a computational bottleneck, as the CPU is ill-equipped to handle a large number of continuous queries. We then extend this simulator by offloading the constant computation of Euclidean distances onto the GPU. Not only is it our intention to show that the GPU can be used as a co-processor to the CPU for the evaluation of *Proximity Area* in this new environment and for evaluating traditional, range-monitoring queries, but we also illustrate the efficacy of the GPU in this new environment. We started by making a generic moving object simulator as described below.

4.1.1 CPU-Based Simulator

The proposed environment has upwards of 100,000 mobile objects randomly moving around, for each unit of time, simulating a mobile environment. Additionally, at each unit of time, all mobile objects are performing queries as follows:

$$\forall \text{ objects, } i \text{ and } j \text{, compute,} \quad d(i,j) = \sqrt{\left(x(i) - x(j)\right)^2 + \left(y(i) - y(j)\right)^2}$$

$$\text{if } d(i,j) < D_{max},\ j \in ProximityArea(i)$$

The algorithm simply calls three main functions, shown in Figure 9, with the inputs to the functions being `MOs`, `xArray`, and `yArray`. `MOs` is an array of structs, each struct representing a mobile object. The contents of each struct include the ID of the mobile object, the number of neighbors each object has, and an array of neighboring objects. For the sake of simplicity, each struct would ideally contain the x and y coordinates representing the position of the mobile object. However, the code was designed to be portable to the GPU, and referencing arrays of structures (accessing the x and y coordinates) is not efficient on the GPU because of the memory access patterns. As such, in addition to an array of structs, we have `xArray` and `yArray`, which are arrays of the x and y coordinates of all mobile objects.

CPU-Based Implementation

```
for (i = 0; i < UnitsOfTime; ++i) {
     moveAllMOs(MOs, xArray, yArray);
     clearNeighborLists(MOs);
     queryAllMOs(MOs, xArray, yArray);
}
```

Figure 9: CPU-based implementation

The first function, `moveAllMOs`, moves each mobile object a random distance in the x and y direction as specified in the program. The struct of each object has an array called `neighboringObjects`, which maintains a list of all neighbors to the given object, within a distance, *r*. `clearNeighborLists` simply clears this list for each object. The third function in the for loop, `queryAllMOs`, is the main component of the simulator and is shown in Figure 10. The function simply calculates the Euclidean distance between each and every object, and if the distance between object A and B is less than some threshold, `queryRadius`, then the ID for object B is saved in A's neighbor list.

queryAllMOs Function

```
void queryAllMOs (MobileObject *MOs, int *xArray, int *yArray) {

     int i, j, k;
     for (i = 0; i < numMobileObjects; ++i) {
        for (j = 0; j < numMobileObjects; ++j) {
           if (i != j) {
              distance =  // Euclid's formula
              if (distance < queryRadius) {
                 AddNeighbor(MOs, i, j);
              }
           }
        }
     }
}
```

Figure 10: queryAllMOs function

This simulator represents the third type of LBSs discussed previously: continuously moving mobile clients that query mobile objects. Each object queries all other objects, calculates the distance metric between it and all other objects, and then generates a neighbor list of all neighbors within some distance, *r*. This method is indeed overkill and is far from realistic for many applications. For example, surely a pedestrian walking down Manhattan searching for a nearby taxi would not need to know the locations of all taxis in New York; rather, they would only need to know the results of any taxis within a few blocks.

However, our motivation for this setup is twofold. Firstly, we wanted the simulator to resemble the view oriented approach mentioned previously. In this approach, *Proximity Area* represents a range query over every object in the location database; there are *n* objects and *n* continuous range-monitoring queries over these objects. The Range Processor then materializes the *Proximity Area* view by performing continuous range queries over all objects in the database. Additionally, with respect to traditional range-monitoring query models, one of the more challenging applications currently being researched is that of military simulation/training or even games, specifically massively multiplayer online (MMO) games. For these applications, it is not only within the realm of possibility for mobile objects (players, soldiers, etc.) to need to know the whereabouts of all other objects, but it is sometimes critical. Typical database problems may have one million nodes and perhaps a couple hundred queries. However, games or simulations may have both one million nodes and one million queries. As such, this simulator models these scenarios. The CPU-based simulator reaches a bottleneck, and we extend this with the GPU, allowing us to scale the number of mobile objects while maintaining the same level of performance.

4.1.2 GPU-Based Simulator

For the GPU-based simulator, the first and second steps of the for loop in the CPU-based simulator, `moveAllMOs` and `clearNeighborLists`, are still present and execute on the CPU. The third step, querying the distances between each and every object and generating neighbor lists, is done on the GPU. The first step is to allocate the necessary memory on the GPU and then copy over any required data. The CUDA provided function calls for this are shown below.

```
cudaMalloc((void**)&x_onDevice, size);
cudaMalloc((void**)&y_onDevice, size);
cudaMalloc((void**)&distanceArray_onDevice, size_d);
cudaMemcpy(x_onDevice, xArray, size, cudaMemcpyHostToDevice);
cudaMemcpy(y_onDevice, yArray, size, cudaMemcpyHostToDevice);
```

The first two lines create memory for the x and y arrays that will be sent over to the device for the purposes of calculating the Euclidean distance matrix, while the third line creates memory for the to-be-computed distance array. The fourth and fifth instructions simply move the data in the `xArray` and `yArray` over to their corresponding x and y arrays on the device. With data on the device, the GPU can now perform meaningful computation. To execute code on a GPU, we write a Kernel, which is a C-like function that, when executed, runs *N* times in parallel by *N* different CUDA threads. To launch this kernel, we must first setup the Execution Configuration of the kernel as shown in Figure 11.

Execution Configuration of Kernel

```
dim3 dimBlock(blockSize/16, blockSize/16);
dim3 dimGrid(numMobileObjects/dimBlock.x,
             numMobileObjects/dimBlock.y);

computeDistanceArray<<<dimGrid, dimBlock>>> (parameter list);
```

Figure 11: Execution configuration of CUDA

CUDA supports a large number of threads and organizes them into a hierarchy of blocks, which are then organized into grids. Blocks can be in one, two, or three dimensions, while a grid can be one or two dimensions. The first two lines represent the number of threads, to be launched, per block and the number of blocks per grid. CUDA current allows up to 512 threads per block organized in a grid of size up to 65535x65535 blocks. For our simulator, we set the blockSize equal to 256 and launch 2-dimensional blocks of size 16x16. If there are *n* mobile objects, the distance matrix will be of size *n*x*n*, representing the distance between each and every object. As such, our simulator launches n^2 threads, with each thread calculating one cell of that matrix. The second instruction in Figure 11 simply launches a 2-dimensional grid that has enough blocks to house the n^2 threads. We assume that the number of mobile objects is a multiple of 256; if not, we can simply increase the grid by one block in each dimension, and we address the extra threads in the kernel. Lastly, `computeDistanceArray` is the kernel (function) call that launches the kernel with the number of threads specified and with a given list of parameters.

Figure 12 shows the code for the kernel. In CUDA, each thread is referenced through its thread and block indices, `threadIdx` and `blockIdx`, respectively. By referencing these indices, we can assign each thread a cell of the distance array to work on. This is done in the fourth and fifth lines of the code. `idx` calculates the storage mapping function used to save the computed values into the distance array. `xDiff` and `yDiff` simply compute the differences between the x and y coordinates of two objects. If the number of mobile objects were not a multiple of the block size (256), this means there are unnecessary threads. We check this with the if statement on line nine. Lastly, the distance between each object is calculated and saved into the distance array as shown on line ten.

GPU-Based Implementation (GPU-1)

```
1    __global__ void computeDistanceArray
2      (int *x_d, int *y_d, int *distance_d, int numObjects) {
3        int i = blockIdx.x*blockDim.x + threadIdx.x;
4        int j = blockIdx.y*blockDim.y + threadIdx.y;
5        int idx = i*numObjects + j;
6        int xDiff = x_d[i] - x_d[j];
7        int yDiff = y_d[i] - y_d[j];
8
9        if (i < numObjects && j < numObjects)
10          distance_d[idx]=
11             (int)sqrt((double)(xDiff*xDiff) + (yDiff*yDiff));
12   }
```

Figure 12: GPU-based implementation (GPU-1)

Now that the distance array has been computed, we simply copy the array back to the host (CPU) and free the memory on the device, as shown below.

```
result = cudaMemcpy(distanceArray, distanceArray_onDevice,size_d,
          cudaMemcpyDeviceToHost);
cudaFree(x_onDevice);
cudaFree(y_onDevice);
cudaFree(distanceArray_onDevice);
```

Lastly, the distance array is used to evaluate the neighbors for each mobile object. As on the CPU, this process iterates continuously, resembling that of moving objects.

In addition to developing this initial GPU simulator, we also made a second GPU simulator, GPU-2, with minor changes. In CUDA, each thread block has shared memory, which allows all threads within a block to cooperate among themselves and synchronize their execution for more efficient memory accesses. The second GPU simulator, GPU-2, simply takes advantage of this shared memory region and allows the threads within each block to work more effectively. In the main program, we create a new `int` variable, `sharedMemSize`, and we set it equal to `blockSize*sizeof(int).` This `sharedMemSize` is sent to the Kernel via an updated Execution Configuration. We also use the intrinsic function, `__syncthreads()`, which works as barrier forcing all threads in a block to wait until all other threads, within that block, arrive before being allowed to proceed. Figure 13 shows the algorithm for GPU-2. We make two arrays on lines 10 and 11, `newX` and `newY`, that reference values in shared memory, and we move the current x and y arrays, on the device, into these new arrays. The number of moves, per array, equals the number of mobile objects. As such, we must have at least that many threads performing the copy. We do this by calculating the `thread_id` as shown in the algorithm. And again, because this `thread_id` has more threads than needed, we must account for that as we did in the first algorithm. The rest of the implementation is essentially the same as GPU-1.

GPU-2 Implementation

```
__global__ void computeDistanceArray
   (int *x_d, int *y_d, int *distance_d,
    int numObjects) {
     int i = blockIdx.x*blockDim.x + threadIdx.x;
     int j = blockIdx.y*blockDim.y + threadIdx.y;
     int idx = i*numObjects + j;

     extern __shared__ int coords[];
     int *newX, *newY;
     newX = (int*)coords;
     newY = (int*)&newX[128];
     int thread_id = blockIdx.x*blockDim.x + threadIdx.x;

     if ( thread_id < numObjects) {
        newX[thread_id] = x_d[thread_id];
        newY[thread_id] = y_d[thread_id];
     }

     __syncthreads();

     int xDiff = newX[i] - newX[j];
     int yDiff = newY[i] - newY[j];

     __syncthreads();

     if (i < numObjects && j < numObjects)
        distance_d[idx] = (int)sqrt((double)(xDiff*xDiff) +
                          (yDiff*yDiff));
}
```

Figure 13: GPU-2 implementation

4.2 GEDS: GPU Execution of Continuous Queries on Spatio-Temporal Data Streams

Using previous frameworks as a starting point [ACCC03, BaWi01, ChFr03], we first develop a CPU-based framework for evaluating spatio-temporal queries over data streams. As with GEDS, we use this framework to evaluate the *Proximity Area* view over each object in the stream. In addition to evaluating the *Proximity Area* view, we also consider a second view, *Neighboring Objects*, and we define it as follows:

$$\textit{Neighboring Objects} = \{[O(i), \{O(j) \mid O(j) \in O(i)\text{'s } kNN\}] \mid O(i) \in O \wedge O(j) \in O\}, \qquad (2)$$

where *O* is the set of objects in memory that the view is based on, and *kNN* represents the *k* neighbors that are nearest to object *O(i)*. While *Proximity Area* computes a range query over the incoming data stream of location updates, *Neighboring Objects* computes the *k* nearest neighbors (*kNN*) for each of these same moving objects in the data stream.

Considering the limited computational resources of the CPU, this framework quickly reaches a computational bottleneck. We then develop GEDS, a GPU-based framework, and offload the evaluation of the queries onto the GPU. Specifically, we address this computational bottleneck by using a GPU on the central server, thereby allowing for a significant speedup in computation. We note that our solution is orthogonal to distributed computing; we would be able to leverage the GPUs in each of the servers in a distributed environment.

4.2.1 CPU-Based Framework

The CPU-based framework has upwards of 100,000 mobile objects randomly moving around, for each unit of time, simulating a mobile environment. Additionally, at each unit of time, the *Proximity Area* view is being materialized over all mobile objects in the data stream as follows:

$$\forall \text{ objects, } i \text{ and } j\text{, compute,} \quad d(O_i, O_j) = \sqrt{\left(x(O_i) - x(O_j)\right)^2 + \left(y(O_i) - y(O_j)\right)^2}$$

$$\text{if } d(O_i, O_j) < D_{max}, \;\; O_j \in ProximityArea(O_i)$$

For all objects in the data stream, compute the distances between object O_i and O_j, and if that distance is less than a user-defined threshold, D_{max}, object O_j is to be included in the output stream of object O_i's *Proximity Area* view.

To facilitate this study, we use a CPU to simulate the moving of mobile objects, the receipt of the incoming stream of location updates, and the evaluation of the *Proximity Area* view. The simulator, shown in Figure 14, calls two main functions, with the input to these functions simply being an array of structs, `MOs`, representing the GPS-enabled mobile objects. Each struct contains the ID of the mobile object as well as its x and y coordinate. The first function, `simMovement`, simulates the movement of these mobile objects by moving them randomly some arbitrary distance in the x and y directions. This function concludes with each object sending a location update to the server.

CPU-Based Stream Implementation (*Proximity Area*)

```
for (i = 0; i < UnitsOfTime; ++i) {
      simMovement(MOs);
      evalProxArea(MOs);
}
```

Figure 14: CPU-Based Framework (*Proximity Area*)

The second function, `evalProxArea`, materializes the *Proximity Area* view, as defined in Section 4.1, and is shown in Figure 15. This function performs range queries over each and every object found in the data stream. The Euclidean distance between all objects is calculated, and if the distance between object A and B is less than the threshold, `queryRadius`, then the ID for object B is included in the output result stream of object A's *Proximity Area* view.

evalProxArea Function

```
void evalProxArea(MobileObject *MOs) {
      int i, j;
      for (i = 0; i < numMobileObjects; ++i) {
         for (j = 0; j < numMobileObjects; ++j) {
            if (i != j) {
               distance =  // Euclid's formula
               if (distance < queryRadius)
                  addToView(MOs, i, j);
            }
         }
      }
}
```

Figure 15: evalProxArea Function

This simulator represents the most challenging type of LBS discussed in literature: continuously moving mobile objects that query other mobile objects. Each object moves about, queries all other objects, calculating the distance between it and all others, and then has its *Proximity Area* view published. From the perspective of mobile users with smart phones, this method is overkill and far from realistic. For example, assuming military commandos wanted to perform a *Proximity Area* query searching for a number of allies within a certain, small distance metric, surely there would be no need to search each and every commando on the battlefield that is sending location updates. Similarly, a pedestrian searching for a taxi would not need to know the locations of all taxis within a fifty kilometer radius; rather, they would only need to know the results of those taxis within a few kilometers.

However, the motivation for this model was twofold. Firstly, although this model is disproportionate with respect to many pedestrian-based applications, one of the more challenging applications being researched is that of military simulations and massively multiplayer online (MMO) games. For these applications, it is not only desirable, but often critical for mobile objects (players/soldiers) to know the whereabouts of all other objects. These games and simulations can have millions of active users with far more active queries. As such, we model this more challenging scenario. Secondly, the effectiveness of GEDS, versus a CPU-based model, needs to be demonstrated via common queries that actually occur in LBS environments. Additionally, in order to show the superiority of GEDS, the model must fully tax the CPU, pushing it to its computational limit. kNN and range queries are extremely common, and the *Proximity Area* view represents a range query over every object in the incoming data stream; there are *n* objects and *n* continuous spatio-temporal queries over these objects. The *Neighboring Objects* view, representing a kNN query over every object in the incoming data stream, also quickly taxes the CPU. As the number of mobile objects increase to even reasonably large values, a CPU simply lacks the computational power to materialize these views at every unit of time. We extend this framework with the GPU, allowing us to significantly scale the number of mobile objects while maintaining acceptable performance.

4.2.2 Materializing the *Neighboring Objects* View

As with the Proximity Area view, at each unit of time, the CPU-based framework will materialize the *Neighboring Objects* view over all mobile objects in the data stream, and this is done as follows:

$$\forall \text{ objects}, O_i, \text{ compute the set}, \quad kNN(O_i) \subseteq O \text{ , such that}$$

$$|kNN(O_i)| = k \quad \text{and} \quad \forall O_p \in kNN(O_i), O_q \in O - kNN(O_i), d(O_i, O_p) \leq d(O_i, O_q) \,.$$

For all objects, O_i, in the data stream, we identify the *k* nearest neighbors to that object by first computing the distances between all objects, O_i and O_j. The distances, for each object, are then sorted via Quick Sort, and the first *k* distances are then included in the output stream of object O_i's *Neighboring Objects* view.

The second CPU-based simulator, shown in Figure 16, follows that of the first simulator, simulating the movement of the mobile objects. The second function, `evalkNN`, materializes the *Neighboring Objects* view as defined in Section I and is shown in Figure 17.

CPU-based Simulator (*Neighboring Objects*)

```
for (i = 0; i < UnitsOfTime; ++i) {
      simMovement(MOs);
      evalkNN(MOs);
}
```

Figure 16: CPU-based Simulator (*Neighboring Objects*)

This function performs *kNN* queries over each and every object found in the data stream. The brute force approach calculates the Euclidean distance between all objects, and if the distance between object A and B is less than the distance between object A and any of the objects from its current set of *k* nearest neighbors, then the ID for object B is included in the output result stream of object A's Neighboring Objects view, replacing the neighbor with the farthest distance from object A. Although this approach may work, it is computationally prohibitive. The more reasonable approach, adopted in this study, is to sort the distances in O(nlogn) time, for each object A, and then simply choose the k smallest distances as the k neighboring objects of object A. We use Quick Sort since it has one of the faster running times, O(nlogn), in practice. Finally, the resulting view is published as a live, outbound data stream. As with computing the *Proximity Area* view, this *Neighboring Objects* view also quickly reaches a computational bottleneck. We extend them both on our GEDS framework, utilizing a GPU on the central server, facilitating computation and allowing us to significantly scale the number of mobile objects while maintaining acceptable performance.

evalkNN Function

```
void evalkNN(MobileObject *MOs) {
      int i, j;
      MOdistance *distRow;
      distRow = (MOdistance*)malloc(sizeof(MOdistance) *
                 numMobileObjects);
      for (i = 0; i < numMobileObjects; ++i) {
         for (j = 0; j < numMobileObjects; ++j) {
            if (i != j) {
               distance =  // Euclid's formula
               distRow[j].ID = j + 1;
            }
            quicksort(distRow, 0, numMobileObjects-1);
            for (k = 0; k < kNN; k++) {
               MOs[i].neighboringObjects[k] = distRow[k].ID;
            }
         }
      }
}
```

Figure 17: evalkNN ***function***

4.2.3 GEDS: GPU Execution of Spatio-Temporal Queries over Spatio-Temporal Data Streams

The GEDS framework follows the CPU-based model. GEDS receives the spatio-temporal data stream of location updates from the GPS-enabled devices and then evaluates the *Proximity Area* view over all objects in the stream. GEDS is simulated using the same two functions as the CPU-based simulator. However, the input to the functions changes. Although it makes sense to store the x and y coordinates of a given object within that object, this is illadvised with the GPU. Referencing arrays of structures is not efficient on the GPU because of memory access patterns. As such, in addition to the array of mobile objects, `MOs`, we now have two coordinate arrays, `xArray` and `yArray`. The first function, `simMovement`, is still present and executes on the CPU. The second function, `evalProxArea`, is now performed on the GPU. As discussed in Chapter 3, the first step of a CUDA program is to allocate the necessary memory on the GPU and copy over any the pertinent data. CUDA provides function calls for this as shown below.

```
cudaMalloc((void**)&xArray_onDevice, size);
cudaMalloc((void**)&yArray_onDevice, size);
cudaMalloc((void**)&distanceArray_onDevice, size_d);
cudaMemcpy(x_GEDS, xArray, size, cudaMemcpyHostToDevice);
cudaMemcpy(y_GEDS, yArray, size, cudaMemcpyHostToDevice);
```

Immediately, one can see the appeal of CUDA, as it resembles C code. The first two lines create memory placeholders for the x and y coordinate arrays of the mobile object positions. The third call to `cudaMalloc` creates memory for the distance array, which will be computed and filled-in on the GPU. The final two instructions, `cudaMemcpy`, simply move the actual coordinate data over to the corresponding arrays on the device. The GPU can now evaluate the *Proximity Area* views over the mobile objects. In order to execute code on a GPU, we must write a Kernel, which is analogous to a C function, except that, when executed, it runs *N* times in parallel by *N* different CUDA threads. Just as a C function is launched by calling the function, a CUDA kernel is launched by setting up and launching an Execution Configuration, as shown in Figure 18.

Execution Configuration of GEDS Kernel

```
dim3 dimBlock(blockSize/16, blockSize/16);
dim3 dimGrid(numMobileObjects/dimBlock.x,
             numMobileObjects/dimBlock.y);

evalProxArea<<<dimGrid, dimBlock>>> (parameter list);
```

Figure 18: Execution Configuration of GEDS Kernel

The first line of the Execution Configuration represents the number of threads per block that will be launched on the GPU. The second line specifies how many blocks are organized into a grid. We launch 2-dimensional blocks of size 16x16, with each block having 256 threads. Assuming there are n mobile objects sending location updates to GEDS, the distance matrix will be of size nxn. Therefore, the simulator will launch n^2 threads, with each thread computing one cell of the matrix. So more precisely, the second instruction of the Execution Configuration launches a 2-dimensional grid with enough blocks to house the n^2 threads. For the purpose of simplicity, we assume that the number of mobile objects is a multiple of 256, equivalent to the number of threads per block. In reality, this would rarely happen, and we can address this by increasing the grid by one block in each dimension and dealing with the extra threads in the kernel. The third and final instruction of the Execution Configuration, `evalProxArea`, is the kernel ("function") call that launches the kernel, with its specified threads and parameters, on the GPU.

GEDS Kernel Implementation

```
1   __global__ void evalProxArea
2      (int *xarr_d, int *yarr_d, int *distance_d, int numMOs) {
3        int i = blockIdx.x*blockDim.x + threadIdx.x;
4        int j = blockIdx.y*blockDim.y + threadIdx.y;
5        int idx = i*numMOs  + j;
6
7        extern __shared__ int coords[];
8        int *shareX, *shareY;
9        shareX = (int*)coords;
10       shareY = (int*)&shareX[128];
11       int threadID = blockIdx.x*blockDim.x + threadIdx.x;
12       if ( thread_id < numMOs ) {
13            shareX[threadID] = xarr_d[threadID];
14            shareY[threadID] = yarr_d[threadID];
15       }
16       __syncthreads();
17
18       int xDiff = shareX[i] - shareX[j];
19       int yDiff = shareY[i] - shareY[j];
20       __syncthreads();
21
22       if (i < numMOs && j < numMOs)
23          distance_d[idx]= (int)sqrt((double)(xDiff*xDiff) +
24                           (yDiff*yDiff));
25  }
26
```

Figure 19: GEDS Kernel Implementation

Figure 19 shows a portion of the implementation of the *Proximity Area* view within the GEDS Kernel. Once execution begins, there will be hundreds of threads working, in parallel, to materialize this view. CUDA references these threads through their thread and block indices, `threadIdx` and `blockIdx`, respectively. This allows us to assign each thread precisely one cell of the distance array to work on. This is done with the fourth and fifth lines of the code. The sixth line creates `idx`, the storage mapping function used to save the newly computed values into the distance array. Lines 7 through 16 allow us to take advantage of CUDA's shared memory. Each thread block in CUDA has shared memory that allows all threads within a block to cooperate and synchronize their execution for more efficient memory access. When we launch the kernel from the Execution Configuration, we include, inside the parameter list, a variable called `sharedMemSize`, which is set equal to `blockSize*sizeof(int).` This represents the size of the shared memory to be used by cooperating threads. Within the kernel, this shared memory is initialized on line 7. Lines 8 through 15 create two new arrays, `shareX` and `shareY`, that reference the shared memory, and then we move the coordinate data into them. Line 16 calls the instrinsic function `__syncthreads()`, which acts as a barrier forcing all threads within a block to synchronize their execution. `xDiff` and `yDiff` on lines 18 and 19 simply compute the differences between the x and y coordinates of two objects, and then the threads are synchronized a second time. The `if` statement on line 22 addresses the possibility of extraneous threads in the situation where mobile objects are not a multiple of 256. The distance array is computed with lines 23-24. Next, *n* threads are used to iterate over this distance array, one for each row, to identify the objects within proximity of a given object. If the distance between object A and B is less than the threshold, `queryRadius`, then the ID for object B is included in the output result stream of object A's *Proximity Area* view. Important to note is that we assume a maximum number of objects in proximity of a given object, thereby allowing us to minimize the size of the data transfers back to the host. While the size is fixed during a given simulation, it is user-defined and can be increased as needed.

4.2.4 Materializing the *Neighboring Objects* View on GEDS

Although both views require the calculation of distance between all objects, the implementation of the *Neighboring Objects* view was more complex in that it required the array of calculated distances, for each object, to be sorted, thereby placing the k nearest neighbors at the front of the array. Instead of calculating the entire two-dimensional distance matrix with a single kernel invocation, the kernel simply evaluates one row of this matrix. As such, we have a for loop on the host side (CPU) that iterates over the number of mobile objects. For each iteration, we invoke the GEDS kernel, which calculates the distances between a given mobile object and all other mobile objects. Finally, the row of distances is sorted on the GPU before being sent back to the CPU, allowing for quick access to the k nearest neighbors. We note that the sort used is not a stable sort, as this is not required for our application.

Lastly, although these simulators are applicable to the specific models mentioned at the beginning of this section, we also wanted to evaluate GEDS in a more common environment. As such, instead of modeling every object as a spatio-temporal query, we randomly choose ten percent of the objects for this purpose. The remaining objects in the stream simply perform location updates. We then evaluate the *Proximity Area* and *Neighboring Objects* views over those ten percent of objects and publish the results. Both the CPU-based simulator and GEDS were modified accordingly. The rest of the implementation remained the same.

CHAPTER 5: PERFORMANCE STUDIES

For our performance studies, we used the following hardware and software configuration. The testing computer is equipped with an Intel Core 2 Duo E8500 processor, overclocked to 4GHz, and with 4GB of RAM. For the first study, Location Dependent Query Processing in Location-based Services, we used an Nvidia GeForce 8600 GT GPU card. The GeForce 8600 is a lower end video card that has 512 MB of memory and only four multiprocessors, each running at 540MHz. For the second study, GEDS, we test a second video card, the Nvidia GeForce GTX 460 1GB GDDR5 GPU card, which is a Fermi-based GPU. The GTX 460 is a mid-range video card that has 1GB of memory and 336 CUDA cores, each running at 1350MHz. For the CPU simulator, computation time was measured with the `clock_t` function, and for the GPU simulator, computation time was measured by the CUDA built in timer. We begin by detailing the results of the GPU simulator used in the traditional environment of Location-based Services. The second subsection provides the results of our proposed GEDS framework

5.1 Location Dependent Query Processing in Location-based Services

We examined the time taken to simulate large numbers of mobile objects on the CPU simulator and on both versions of the GPU simulator. It is important to point out that the CPU-based simulator was not designed for multi-core programming. However, even if it were, the GPU implementation would still clearly outpace that of the CPU implementation simply by virtue of a tenfold increase in GFLOPS.

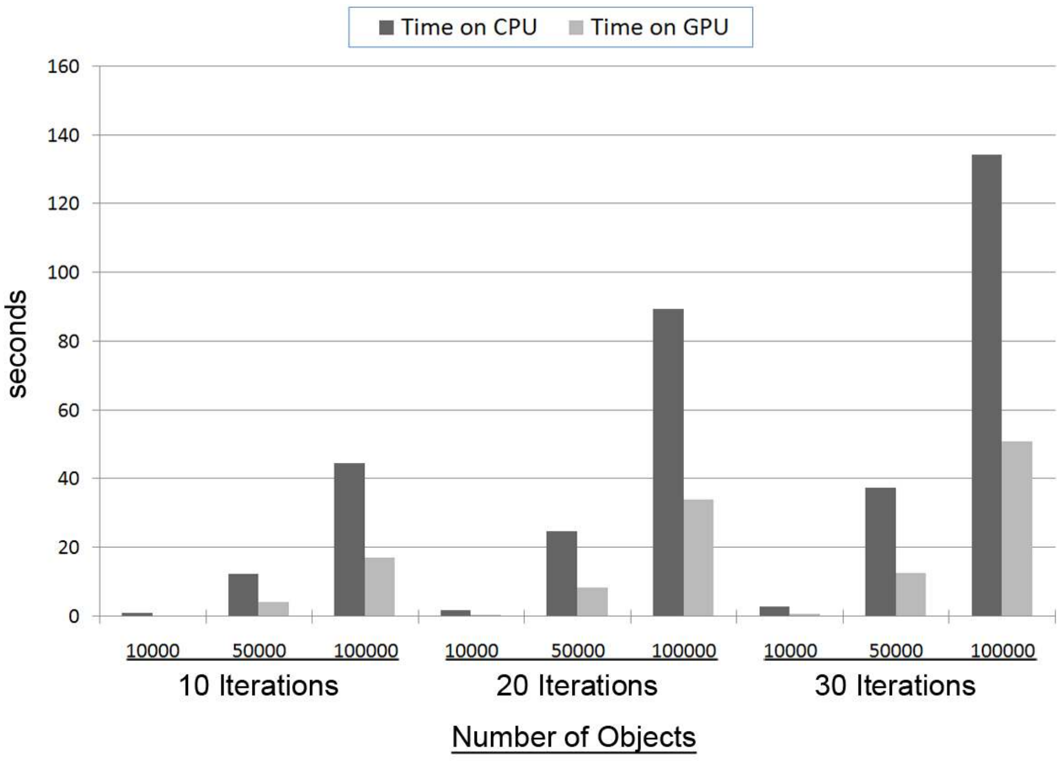
5.1.1 Performance Comparison Between the CPU Simulator and GPU-1

We tested the CPU simulator and both GPU simulators with 10,000, 50,000, and 100,000 mobile objects. To simulate the movement of these objects, we tested the simulators over a variety of iterations, where each iteration represents a unit of time. As shown in Table I, the first GPU simulator resulted in a speed up of 260 – 440% over that of the CPU based simulator, with the lower percentage corresponding with the larger number of mobile objects. As expected, in all cases, the speedup factor remained the same as the iterations increased. Figure 20 shows a graphical representation of this data.

Table 1: Performance Comparison Between the CPU Simulator and GPU-1.

Num. of objects	**Iterations**	**CPU time**	**GPU-1**	
			time	**speedup**
10000	10	0.97	0.22	4.4
10000	20	1.92	0.45	4.3
10000	30	2.98	0.67	4.4
50000	10	12.45	4.29	2.9
50000	20	24.92	8.48	2.9
50000	30	37.48	12.76	2.9
100000	10	44.71	17.04	2.6
100000	20	89.52	34.12	2.6
100000	30	134.33	51.05	2.6

Time is in seconds.



Time is in seconds.

Figure 20: Performance comparison between the CPU simulator and GPU-1.

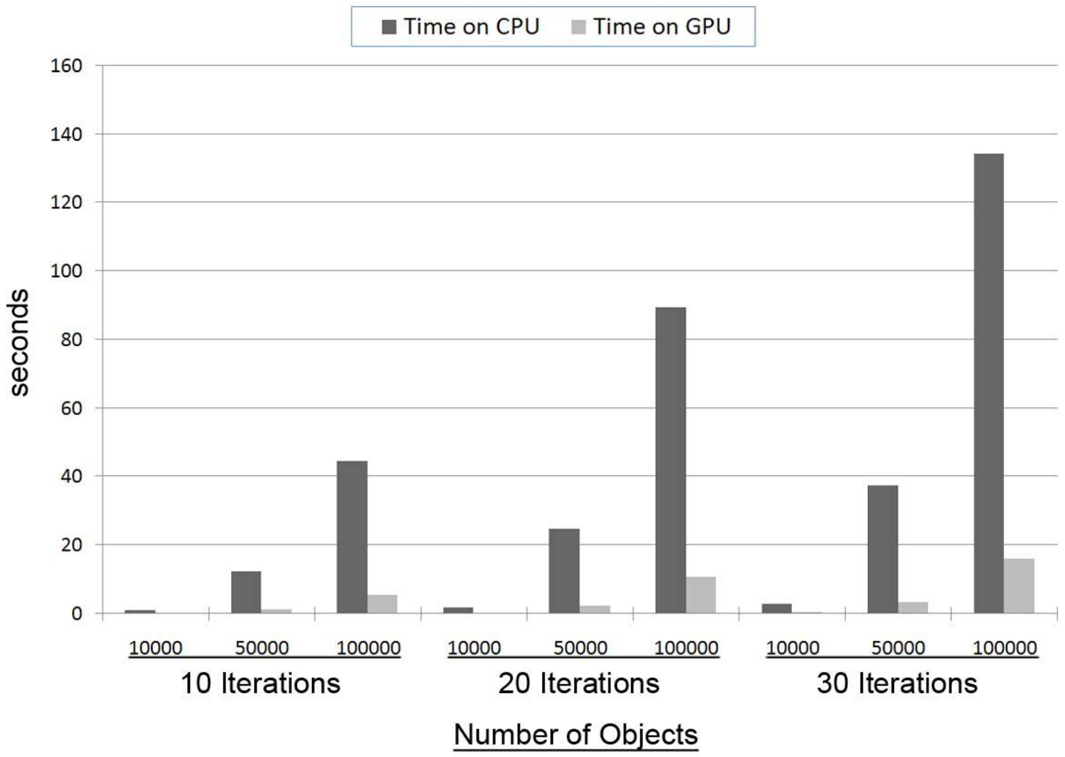
5.1.2 Performance Comparison Between the CPU Simulator and GPU-2

Table 2 shows the results of GPU-2 in comparison to the CPU simulator. GPU-2 resulted in a speedup of 840 – 1330% over the CPU simulator. Again, in all cases, the speedup factor remained the same as the iterations increased. Figure 21 shows the graphical representation of the data in Table 2, allowing us to visualize the speedup between the two versions of the GPU simulator.

Table 2: Performance Comparison Between the CPU Simulator and GPU-2.

Num. of objects	Iterations	CPU time	GPU-2 time	GPU-2 speedup
10000	10	0.97	0.073	13.3
10000	20	1.92	0.15	12.8
10000	30	2.98	0.228	13.1
50000	10	12.45	1.10	11.3
50000	20	24.92	2.22	11.2
50000	30	37.48	3.34	11.2
100000	10	44.71	5.30	8.4
100000	20	89.52	10.69	8.4
100000	30	134.33	15.92	8.4

Time is in seconds.



Time is in seconds.

Figure 21: Performance comparison between the CPU simulator and GPU-2.

5.1.3 Comparison of speedup between GPU-1 and GPU-2

Lastly, Figure 22 gives a graphical representation of the speedup for both versions of the GPU. The average speedup was 350% for GPU-1 and approximately 1100% for GPU-2, and by taking full advantage of the memory access patterns, coalescing memory accesses and avoiding bank conflicts, we expect that we can further speed up GPU-2 by a factor of 5x, resulting in a net speedup of over 5000%.

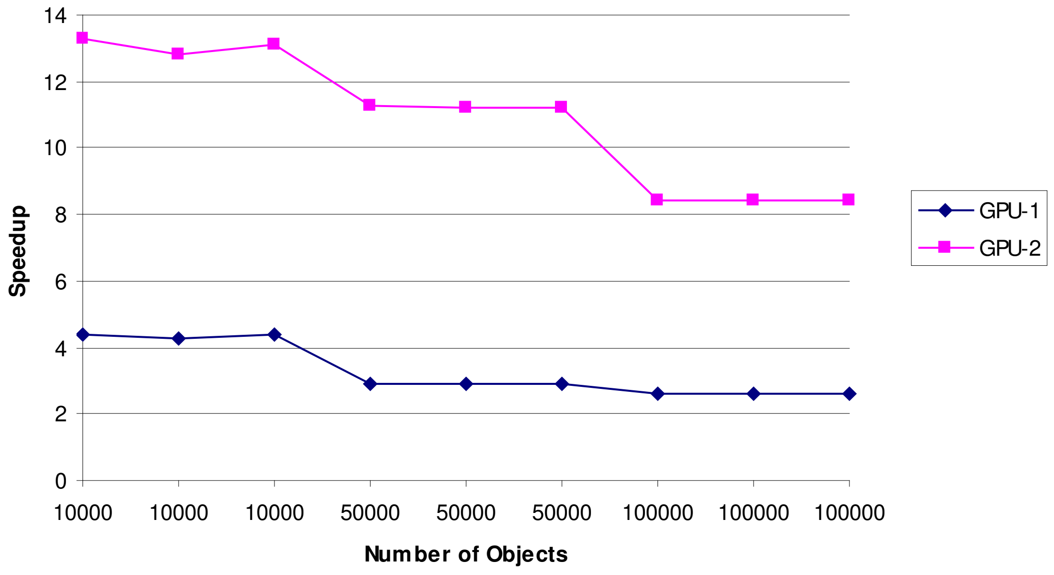


Figure 22: Comparison of speedup between GPU-1 and GPU-2.

5.2 GEDS: GPU-Based Framework

Using the same hardware and software configurations, we ran the simulation with large numbers of mobile objects sending streams of location updates and examined the time taken to materialize the *Proximity Area* and *Neighboring Objects* views on both versions of the CPU-based simulator as well as GEDS.

5.2.1 GEDS and the CPU-based Simulator (*Proximity Area* View)

Both the CPU and GPU simulators were tested with 10,000, 50,000, and 100,000 mobile objects. The movement of these objects was simulated over a certain number of iterations, where each iteration represented a unit of time. Table 3 shows the performance comparison between GEDS and the CPU-based simulator with the *Proximity Area* view evaluated over all objects. As shown, GEDS resulted in a net speedup of 4640 – 5470%, with the lower percentage corresponding to the larger number of mobile objects. With the number of mobile objects fixed, we see that the speedup factor remained the same as the iterations increased. However, when the number of mobile objects increased, the speedup did decrease. This is due to the longer wait times for copying memory between the CPU and GPU. Figure 23 shows a graphical representation of this data.

Table 3: Performance Comparison Between GEDS and the CPU-based Simulator (*Proximity Area* View)

Num. of objects	Iterations	CPU time	GEDS time	GEDS speedup
10000	10	1.04	0.019	54.7
10000	20	2.05	0.037	54.7
10000	30	3.19	0.058	54.7
50000	10	13.32	0.271	49.2
50000	20	26.66	0.542	49.2
50000	30	40.1	0.815	49.2
100000	10	47.84	1.030	46.4
100000	20	95.79	2.064	46.4
100000	30	143.73	3.098	46.4

All objects perform queries. Time is in seconds.

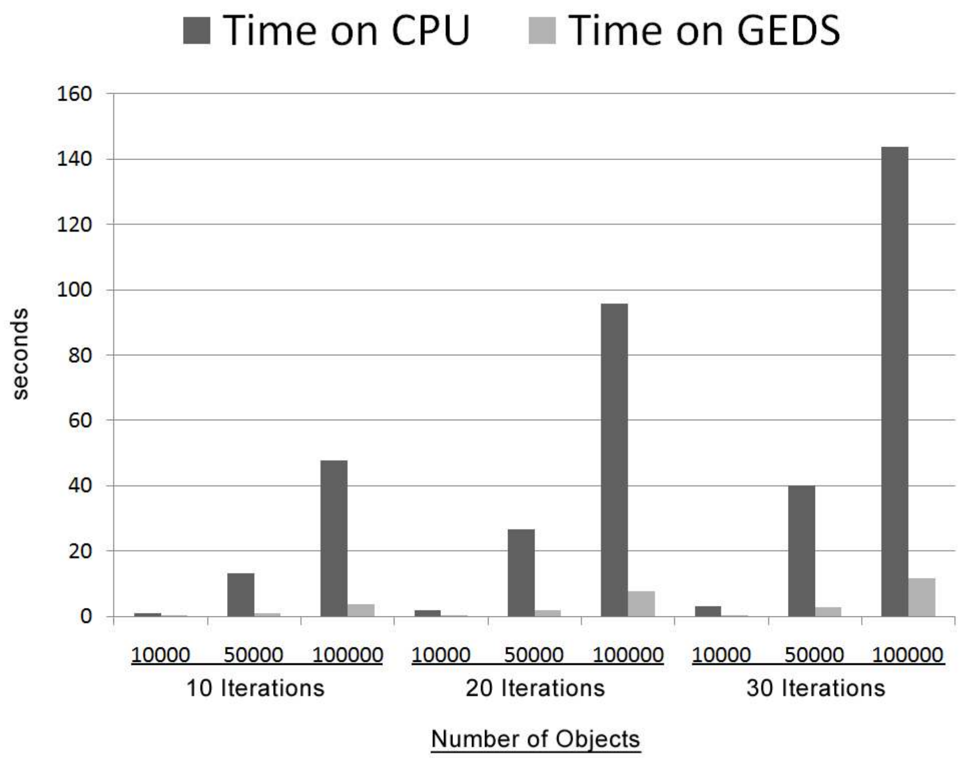


Figure 23: Performance Comparison Between GEDS and CPU-based Simulator (*Proximity Area* view evaluated over all objects)

5.2.2 GEDS and the CPU-based Simulator (*Neighboring Objects* View)

Table 4 shows the performance comparison between GEDS and the CPU-based simulator with the *Neighboring Objects* view evaluated over all objects. Because of the complexity of sorting each of the rows of distances, we reduced the number of mobile objects tested to 1000, 5000, and 10000. We also increased the number of iterations to get a higher resolution view of the time. For this query, GEDS resulted in a net speedup of 1830 – 3250%, again with the lower percentage corresponding to the larger number of mobile objects. As expected, with the number of mobile objects fixed, we see that the speedup factor remained the same as the iterations increased. Again, we note that the speedup decreased as the number of mobile objects increased. Figure 24 shows a graphical representation of this data.

Table 4: Performance Comparison Between GEDS and the CPU-based Simulator (*Neighboring Objects* View)

Num. of objects	Iterations	CPU time	GEDS time	GEDS speedup
1000	100	7.29	0.22	32.5
1000	200	14.64	0.45	32.5
1000	300	21.92	0.67	32.5
5000	100	206.1	7.96	25.9
5000	200	411.6	15.89	25.9
5000	300	617.2	23.83	25.9
10000	100	869	47.48	18.3
10000	200	1738.5	95.00	18.3
10000	300	2604.9	142.34	18.3

All objects perform queries. Time is in seconds.

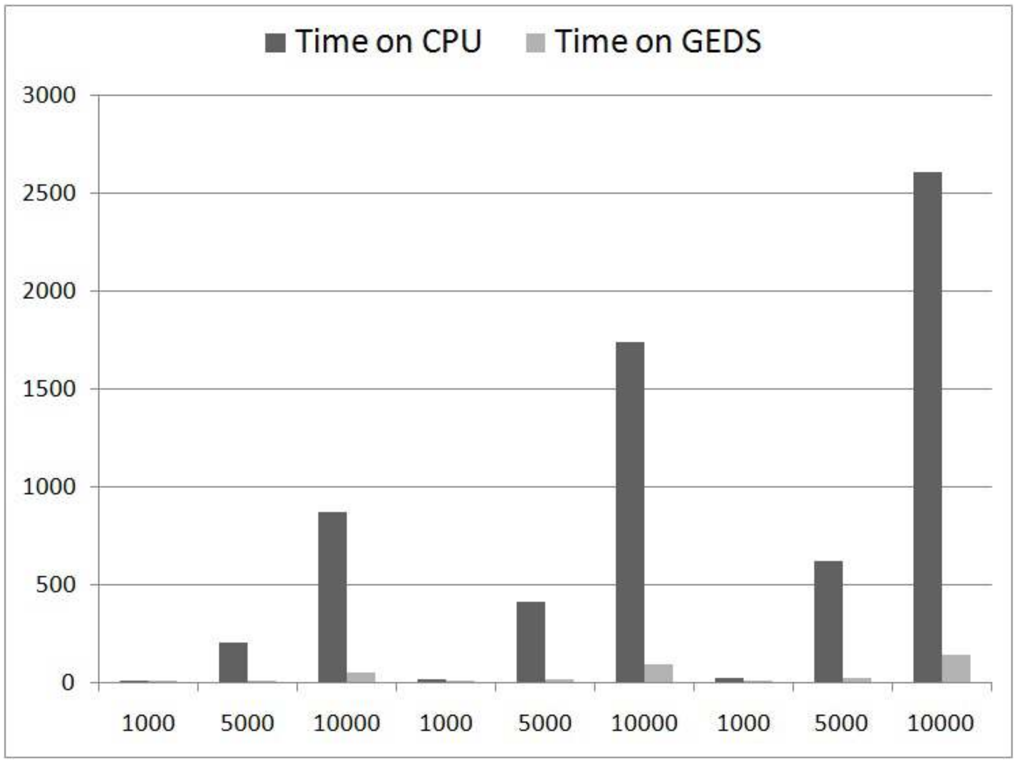


Figure 24: Performance comparison between GEDS and CPU-based simulator (*Neighboring Objects* view evaluated over all objects)

5.2.3 GEDS and the CPU-based Simulator (10% of Objects Perform Queries)

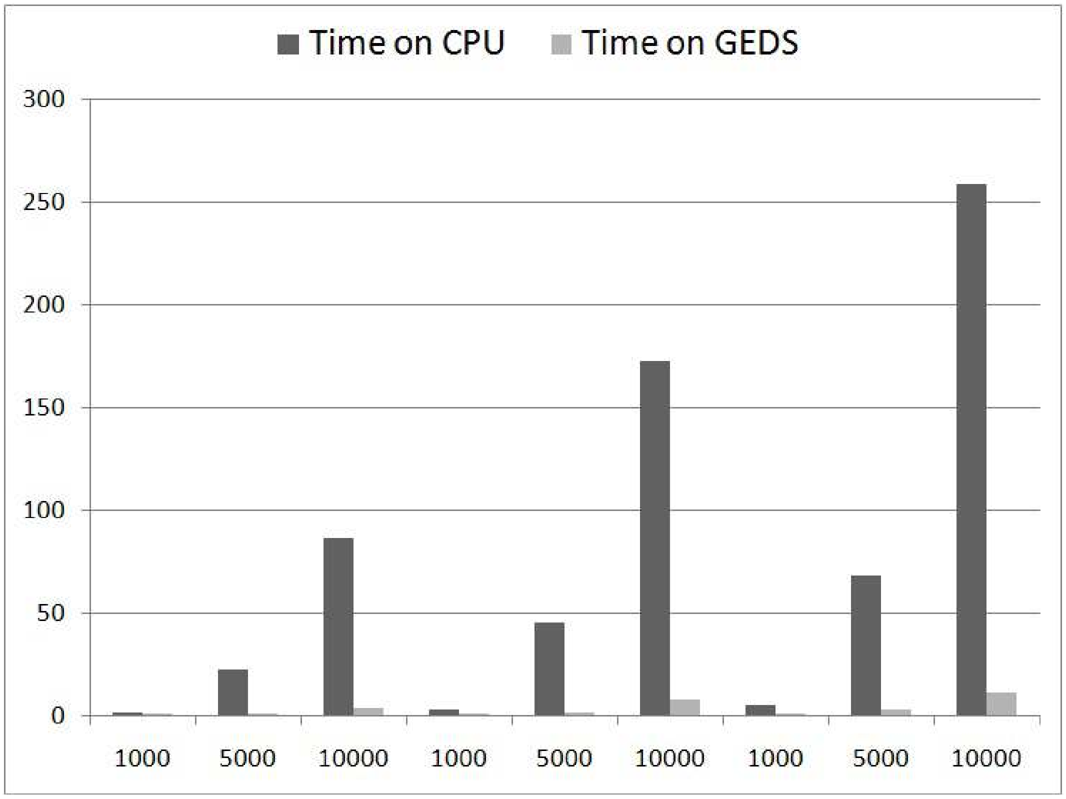
For the second version of the GEDS simulator, only ten percent of the mobile objects perform queries over the incoming data stream of location updates. As such, we again increased the iterations by a multiple of ten, in order to offset the lowered computation in this model. Table 5 shows the performance comparison between GEDS and the CPU-based simulator with the *Proximity Area* view materialized on only ten percent of the mobile objects. As expected, the inherent parallelism of GEDS allows it to far outperform its CPU-based counterpart. GEDS resulted in a speedup of 2240 – 2640% over the CPU-based simulator. As with the previous setup, the speedup factor remained the same as the iterations increased. Figure 25 shows the graphical representation of this data.

Finally, this simulation of materializing the *Proximity Area* and *Neighboring Objects* views over GPS-enabled devices was simply one of the many ways we could have used to demonstrate the efficacy of GEDS in spatio-temporal data streaming environments. We could have chosen to perform other common queries over the streaming data or perhaps even materialized some other view. The means used were simply a tool in this study. What is important, however, is that the evaluation of queries over data streams should dictate the use of a stream processor. The benefits of using a stream-processing framework in a data streaming environment seem to be intuitively obvious, and statistical analysis confirms this hypothesis.

Table 5: Performance Comparison Between GEDS and 2nd CPU-based Simulator

Num. of objects	Iterations	CPU time	GEDS time	GEDS speedup
10000	100	1.56	0.059	26.4
10000	200	3.07	0.450	26.4
10000	300	4.78	0.674	26.4
50000	100	22.64	0.913	24.8
50000	200	45.32	1.827	24.8
50000	300	68.17	2.749	24.8
100000	100	86.11	3.844	22.4
100000	200	172.42	7.697	22.4
100000	300	258.71	11.550	22.4

10% objects perform queries. Time is in seconds.



Proximity Area view evaluated over 10% of objects.

Figure 25: Performance Comparison Between GEDS and 2nd CPU-based Simulator

CHAPTER 6:
IMPACT OF MEMORY TRANSFERS ON SPEEDUP

A closer analysis of the data in all three Tables shows that the speedup decreases as the number of mobile objects increase. This seems to suggest that the larger number of objects has a negative impact on speedup, which goes against what one would typically expect, similar speedup for all numbers of objects. We worked from the premise that this negative impact on speedup usually results from one thing: cost of memory copying from the CPU to the GPU, and then back. To confirm this theory and also to give a more accurate representation of the true GPU speedup, we modify the GEDS simulator by isolating the GPU execution, independent of the memory copying costs, thereby allowing us to gauge the true GPU speedup.

Please note, that although there are costs involved in using a GPU, even after those costs, we are still experiencing speedups upwards of 5500% for the Proximity Area view. So, in no way, should this cost be a negative against the GPU. Rather, the benefits of the GEDS framework, namely significant computational speedups, clearly counter and outweigh any associated costs.

6.1 GEDS with Modified *Proximity Area* View (GEDSv2)

To be clear, the only purpose of this third simulator is to establish that the reduction in speedup is a result of the memory copying costs to move the data from the CPU to the GPU and then back after execution. The only way to confirm this is to isolate the GPU execution, independent of memory copying costs. Meaning, for this testing simulator, we will not be copying the mobile objects back and forth for the units of time specified. An initial set of mobile objects will initially be copied over to the GPU, and the GPU will constantly re-compute the distance matrix, over these objects, for each unit of time. We note that these changes make the simulator unusable in practice; it is not calculating anything of importance. Rather, the simulator simply recalculates the same data over and over again. However, for the purpose of our study, this is acceptable. The goal was to test the speed of the GPU, independent of memory copying, and the most straightforward way to accomplish this is by removing the memory copying altogether.

With the mobile objects now on the GPU, we can now execute the necessary instructions on the GPU. Specifically, we constantly re-compute the distance matrix, over the given objects, for each unit of time. To do this, we simply execute the GPU Kernel the necessary number of times, as shown in Figure 26. At the same time, on the CPU, we constantly recomputed the neighbor lists for each unit of time. As with the GPU calculations, the neighbor lists are constantly re-computed based off of the initial set of mobile objects. Again, this is acceptable for the purpose of this study.

GEDSv2 Kernel Call

```
for (i = 0; i < numMobileObjects; i++) {
     computeDistanceArray<<<dimGrid, dimBlock>>>(parameter list);
}
```

Figure 26: GEDSv2 Kernel Invocation

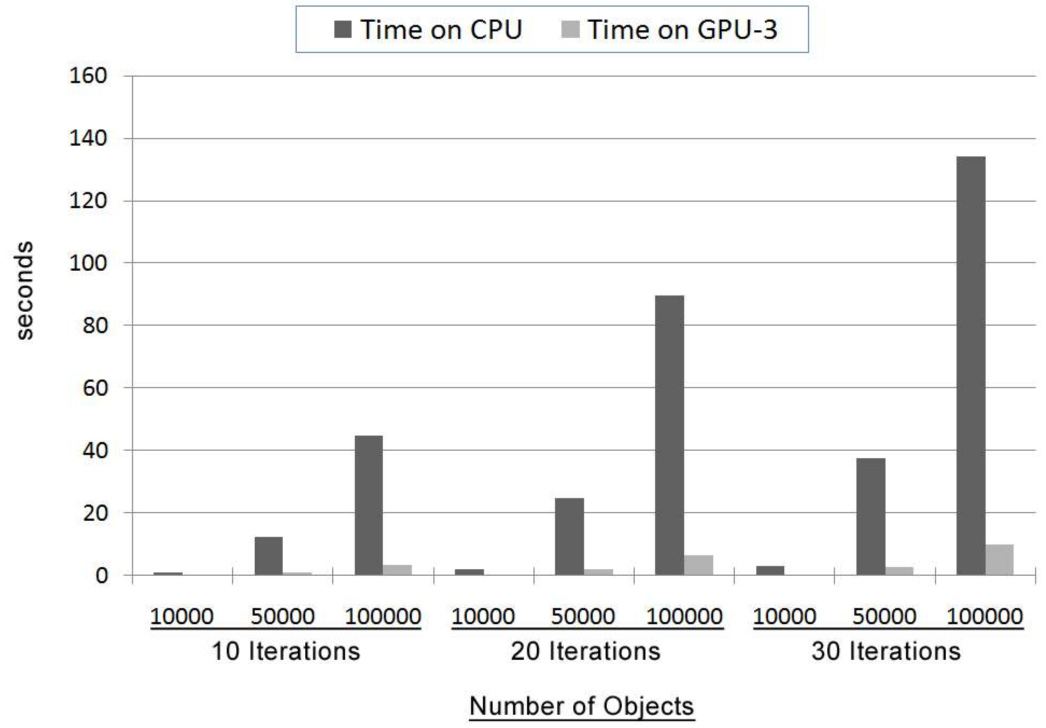
6.2 Performance Analysis of GEDSv2

We tested this modified simulator using the same test bed as the previous simulators and with the same criteria. Table IV shows the results of GEDSv2 in comparison to the CPU simulator. GEDSv2 resulted in a speedup of approximately 6240% over the CPU simulator. Fig. 27 shows the graphical representation of the data in Table 6, allowing us to visualize the speedup offered from GEDSv2. Again, in all cases, the speedup factor remained the same as the iterations increased. However, for the purpose of this simulator, we are only looking at whether or not the speedup stays the same as the number of mobile objects increase. And by examining the results, the hypothesis is confirmed. For all intents and purposes, all numbers of objects, and at all units of time, have the same speedup.

Table 6: Performance Comparison Between the CPU Simulator and GEDSv2.

Num. of objects	Iterations	CPU time	GEDSv2 time	GEDSv2 speedup
10000	10	0.97	0.016	62.4
10000	20	1.92	0.031	62.4
10000	30	2.98	0.048	62.3
50000	10	12.45	0.200	62.4
50000	20	24.92	0.401	62.2
50000	30	37.48	0.601	62.4
100000	10	44.71	0.718	62.3
100000	20	89.52	1.435	62.4
100000	30	134.33	2.153	62.4

Time is in seconds.



Time is in seconds.

Figure 27: Performance comparison between the CPU simulator and GEDSv2.

Although the simulation of GEDSv2 was naïve, it allowed us to confirm what seemed obvious: there is a cost involved in copying memory from the host to the device, and then backwards after computation. This cost cannot and should not be overlooked. However, even when these costs are considered, the value of the parallel processing power of the GPU, clearly counters and outweighs any associated costs.

CHAPTER 7: PERFORMANCE MODELING OF SPATIO-TEMPORAL ALGORITHMS OVER GEDS FRAMEWORK

While our GEDS framework was clearly a success, we also take a broader approach in our analysis of GPU computing. What algorithms are appropriate for the GPU? What types of applications can benefit from the parallel and stream processing power of the GPU? And can we identify a class of algorithms that are best suited for GPU computing? To answer these questions, we develop an abstract performance model, detailing the hardware of both the CPU and GPU and detailing the relationship between them. An abstract data model will serve to identify bottlenecks within the CPU-GPU paradigm such as those described by Gregg and Hazelwood. [GrHa11] Additionally, this model will allow users to reasonably predict system behavior and performance, thereby allowing users to determine not only the practicality of porting algorithms to the GPU, but also allowing them to gauge approximate performance increases. Important to note is that mathematical models already exist within the field of parallel computing; however, they are only designed for traditional, CPU-based, multi-core architectures. Our model incorporates the GPU, details the hardware specifications that result in such phenomenal parallel processing power, and then describes the relationship between the CPU and the GPU, allowing us to develop a list of attributes common to successful GPU-based applications.

7.1 Maximum Performance Benefit

Before delving into the details of the CPU-GPU relationship, it is beneficial to have at least a rudimentary understanding of the theoretical speedups one can expect with GPU processing. In general, the amount of speedup an algorithm can realize has a direct relationship with the percentage of the program that can be parallelized. It is only when algorithms can be sufficiently parallelized that they can take advantage of additional processing cores.

7.1.1 Amdahl's Law

Amdahl's law [Amd67] describes the maximum theoretical speed-up one can expect when parallelizing segments of a serial program and can be stated as

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \quad , \tag{3}$$

where N is the number of processors, P is the proportion of the serial code that can be parallelized, and *(1-P)*, therefore, is the proportion of the program that cannot be parallelized (remains serial). For the sake of simplicity, we can assume an unlimited number of CUDA processing cores, thereby reducing the fraction, *P/N*, to zero, resulting in a new equation for speed-up:

$$S = \frac{1}{(1-P)} \quad . \tag{4}$$

If 90% of a program can be parallelized, even if we have an infinite number of processors, the maximum possible speed-up will be 1 / (1 – 0.9) = 10. According to Amdahl, speed-up is effectively limited by the fraction of the program that remains serialized. The greater the value of *P*, the greater the speed-up one can expect.

The idea behind Amdahl's law is quite intuitive, and we can see clear examples of it in practice. As an example, the task of building a network of thousands of computers can certainly be parallelized. Each additional technician added to the job would certainly reduce the overall time required to complete the task. Indeed, speed-ups would be experienced until the number of computers in the network equals the number of active technicians. Assuming *n* computers are in the network, *n* technicians would complete the task approximately *n* times faster than a single technician. Considering that this task of building a network has a limited S value, the speedup would be equal to the number of technicians working, that is, until the number of technicians becomes so numerous that they are no longer even helpful. At this point, diminishing returns will be seen for each additional technician added to the job.

Now consider the example of building the central server used for this network. As opposed to the building of the large network, which can be parallelized by having hundreds or even thousands of technicians on the job, perhaps two or three technicians, at most, could work together on building this central server. However, beyond this small number, any additional technicians would be useless and would potentially complicate the task due to the small working space (one computer/server). The serial nature of the job simply means that limited speedups can be experienced.

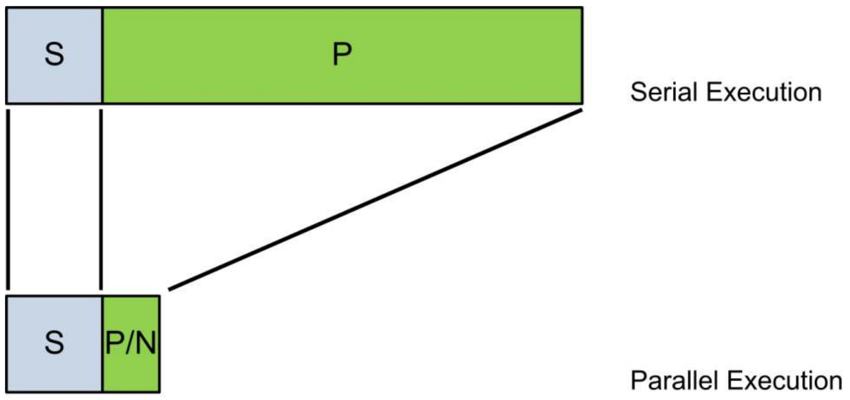


Figure 28: Illustrative Speedup Using Multiple Processors

Figure 28 gives a graphical depiction of Amdahl's Law at work, illustrating that the execution time for the parallel portion of the program is reduced as more processors are utilized. One can also see that as more and more processors are added to the system, the runtime becomes dominated by the serial portion of the code. Finally, the time spent executing the serial portion of the program remains constant regardless of the number of processors added to the system.

Figure 29 shows a graph of Equation 3 with the number of processors (*N*) set to 512. It seems reasonable to limit the serial portion of a program to 5%, thereby parallelizing the other 95% percent. Upon examination of the graph, however, the maximum speedup would be only 20 times, even though 512 processors were used! There is little to no tangible benefit from 492 of those processors, illustrating that there is no purpose of running such problems on large numbers of processors.

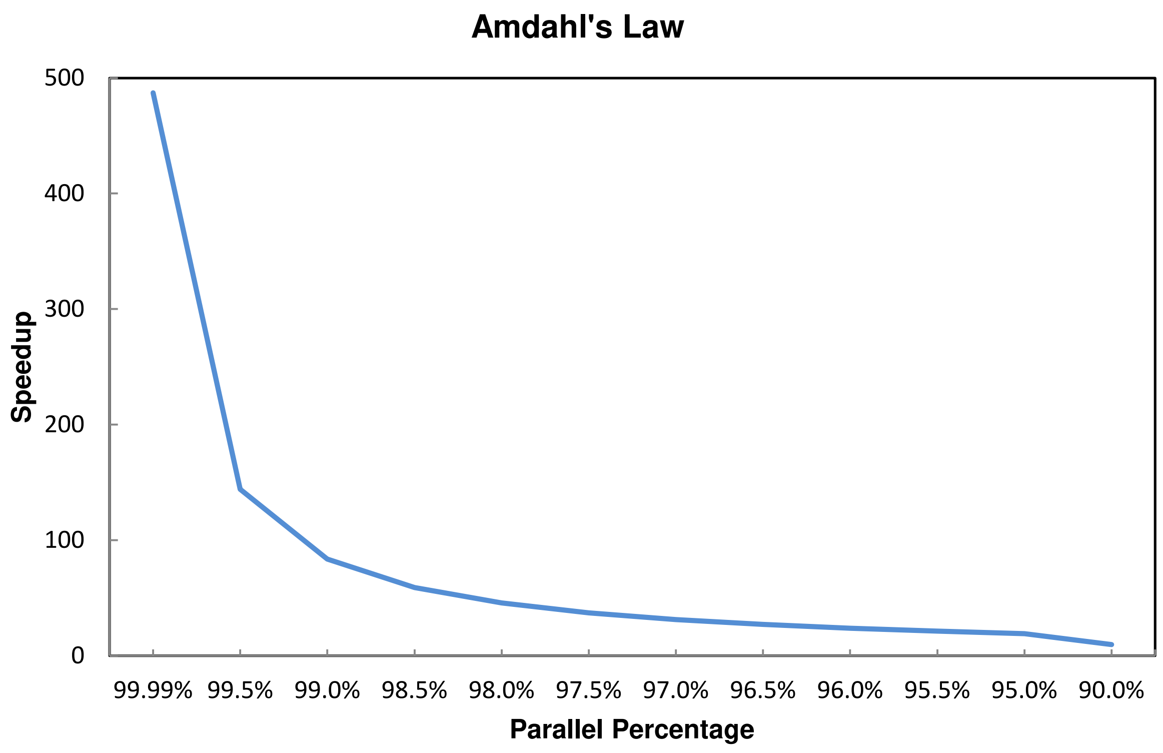


Figure 29: Speedup using Amdahl's Law (N = 512)

7.1.2 Gustafson's Law

The one shortcoming of Amdahl's Law was that it assumes a fixed problem size and does not scale the availability of computing power as the number of processors increases. Even if 512 processors are used, the fixed problem size assumed by Amdahl's Law limits the theoretical speedup. In 1988, John L. Gustafson proposed the idea of Scaled Speedup [Gus88], a model in which the problem size is scaled when executed on more powerful platforms. His model, which later became referred to as "Gustafson's Law", states that the theoretical speedup of a system can be shown by

$$S(N) = N - \alpha \cdot (N - 1) \quad ,$$

where N is the number of processors and α is the non-parallelizable part of the process. As α diminishes with larger problem sizes, the theoretical speed-up then approaches N, as expected.

Going back to the computer network example, if we only need ten computers networked, hiring one thousand technicians is overkill and will certainly not speed up the build any more than having ten to twenty technicians. Per Amdahl, the speedup is, understandably, limited by the portion of the task that remains serial. However, we can see a clear application of Gustafson's Law in this example. Even with the serial portion of the task remaining constant, if we scale the problem size, i.e., the number of computers to be networked, we can certainly use, take advantage of, and benefit from additional technicians.

Mapping this now to a Fermi-based Nvidia GPU with 512 CUDA cores, if α is minimized by means of a very large problem size, the theoretical speed-up could indeed reach up to 512x. Regardless of which law one follows, the message is clear: developers should focus their efforts on increasing P and minimizing the amount of serial code, thereby taking the most advantage of the computation power of the GPU.

7.2 Modeling the CPU-Memory-GPU Relationship

The GPU has increasingly been used as a coprocessor along with the CPU, allowing the CPU to offload processor-intensive tasks, resulting in better system performance. However, not all programs are suitable for processing on the GPU, as they may be more serial in nature, may require significant branching or synchronization, or may require large and frequent data transfers between the host (CPU) and the device (GPU). As discussed in Chapter 3, in its simplest form, a CUDA program is one that runs on the CPU, transfers data (if needed to the GPU), performs execution on the GPU, returns the data to the CPU, and then does this continuously, depending on the application. For the sake of simplicity, we can assume only one TET (transfer-execute-transfer) cycle. Figure 30 provides an abstract view of the CPU/GPU computing model.

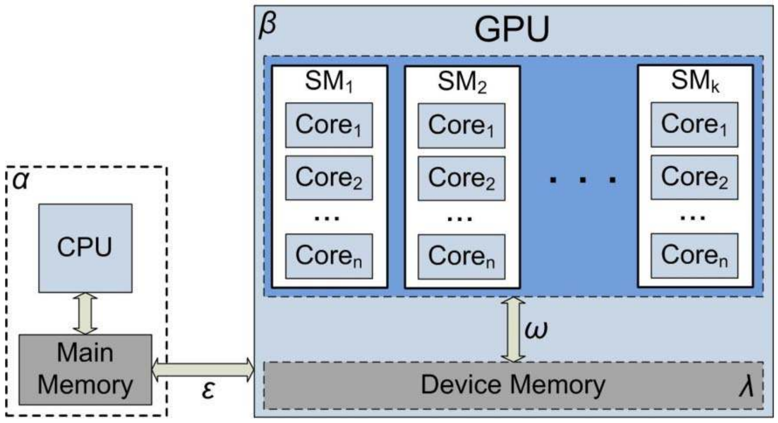


Figure 30: Illustration of GPU Computing Model

From this model, we can decompose the problem into three main components and extrapolate an equation representing the overall execution time as

$$t_{total} = \alpha + \varepsilon_{HD} + \beta + \varepsilon_{DH} \quad , \tag{5}$$

where the CPU execution time (serial portion of program) is represented by α, the GPU execution time (parallel portion of program) is represented by β, ε_{HD} represents the time required to transfer the data from the host to the device, and ε_{DH} represents the time required to transfer the data from the device back to the host. When we compare this equation to that of a completely serial execution time (α), these additional three components (β, ε_{HD}, ε_{DH}) would seem to increase the overall execution time. Of course, the expectation is that α is significantly reduced when offloading computation to the GPU. In theory, the GPU-based execution time should be significant faster.

7.2.1 Modeling GPU Execution

Our performance model needs to account for single GPU execution as well as multiple-GPU execution. We first develop an equation modeling the performance on a single GPU and then use this equation to extrapolate an estimated running time on multiple GPUs. Important to note is that execution times will certainly change based on the hardware used. To facilitate a broad reaching model and to accurately extrapolate GPU execution times across multiple GPUs, our model assumes that multiple-GPU applications will all use the same model of GPU.

The first step is to determine the query execution time of a single mobile object for one unit of time. Based on our GEDS framework, this would be the time to evaluate either the *Proximity Area* view or the *Neighboring Objects* view over one mobile object. This is calculated by taking the total GPU execution time of the reference problem, for one unit of time, and dividing that by the number of mobile objects (*N*) in the data stream. Equation 6 gives us the per-element average execution time of a single mobile object and can be expressed as

$$t_{mobileObject} = \frac{t_{gpu\_refprob}}{N_{mobileObjects}} \quad (6)$$

where $t_{gpu\_refprob}$ is the total GPU execution time of a given reference problem, $N_{mobileObjects}$ is the number of mobile objects in the data stream, and $t_{mobileObject}$ is the newly calculated query execution time of a single mobile object for one unit of time. From this equation, we can then extrapolate the total execution time across *M* GPUs. As seen in Equation 7, we simply multiply the per-element average ($t_{mobileObject}$) times the total number of mobile objects and then divide this by the number of GPUs (*M*) used in the application. The total execution time across *M* GPUs can then be expressed as

$$t_{gpu} = t_{mobileObject} \cdot \left\lceil \frac{N_{mobileObjects}}{M_{gpus}} \right\rceil \tag{7}$$

where $t_{mobileObject}$ is the previously calculated execution time of a single mobile object for one unit of time, $N_{mobileObjects}$ is the number of mobile objects in the data stream, and M_{gpus} is the number of GPUs used in the system, and t_{gpu} is the newly calculated execution time across these M GPUs. As detailed in Chapter 7.3, this method has shown to be quite accurate for many algorithms. The large number of mobile objects used in our model allows for a precise per-element average, thereby providing a highly accurate approximation of the expected running time.

However, there is one specific limitation, namely, that this method will only accurately predict GPU execution for those algorithms whose GPU work scales linearly as the number of mobile objects increase. For those applications whose GPU execution does not scale linearly with the number of mobile objects, we still calculate $t_{mobileObject}$ from our reference GPU problem. However, as the number of mobile objects scales linearly, the GPU execution time scales by an order of magnitude. As a result, the calculated value for $t_{mobileObject}$ can vary greatly, depending on the number of mobile objects in the reference problem. We account for this in our model by multiplying $t_{mobileObject}$ by a variable value, which represents the fraction of current mobile objects (in the predicted problem) to the originally measured number of mobile objects.

7.2.2 Modeling the PCI-Express Interconnect

Unfortunately, until today, all memory transfers between the CPU and the GPU occur over the slow PCI Express connection. Using our Fermi-based GTX 460 GPU, we measured the data transfer bandwidth using the CUDA SDK `bandwidthtest` application. Using paged memory, host to device bandwidth was measured at 2007.4 MB/s, and device to host bandwidth was measured at 1819.9 MB/s. Comparing these measurements to the device to device bandwidth of 59696.0 MB/s illustrates that the data transfer pipeline between the CPU and GPU may arguably be the main limiting factor for many applications.

7.2.2.1 Pinned Memory as an Option

To deal with this, CUDA supports the allocation of non-pageable memory, i.e., memory that is pinned to RAM. The usage of pinned memory reduces data transfer over the PCI Express, as transfers can happen immediately; data does not have to first be placed within known locations in RAM. Unfortunately, this increase in device bandwidth comes at a cost: pinned allocations are more expensive than allocating standard pageable memory. Additionally, system performance can decline considerably if the amount of pinned memory exceeds the capacity of available RAM. Portability also comes into play, as large allocations of pinned memory will only be viable on systems with enough RAM. While CUDA does provide this viable option to the programmer, it should only be used when system RAM can house the entire data set.

7.2.2.2 Data Transfers

Table 7 gives the pinned and pageable data transfer bandwidths for our testbed GPU. A quick examination of the data illustrates the limited throughput over the PCI Express, especially when compared to the device-to-device throughout, which is upwards of 30x the speed. While pinned memory does offer better performance than its paged counterpart, the transfer of data between the CPU and GPU is still a key limiting factor in the calculation of expected speedup. And as GPUs increase in compute performance, this bottleneck becomes more evident, simply by virtue of the proportionally smaller time they spend computing kernels compared to slower GPUs with the same PCI Express bandwidth.

Table 7: Data Transfer Throughput Using Paged vs Pinned Memory

GPU Type	*Memory Type*	*Host-Dev BW (MB/s)*	*Dev-Host BW (MB/s)*	*Device-Device BW (MB/s)*
GTX 460	Paged	1924.4	1895.8	59687.2
GTX 460	Pinned	5251.0	5786.5	59696.7

This data transfer bottleneck gets worse with the use of multiple GPUs. While the use of multiple GPUs certainly allows for a greater theoretical speedup, each of them is connected to the same PCI Express. Figure 31 depicts a four-GPU shared system in which the GPUs share the PCI Express. The PCI-e bus switch allows multiple GPUs to either exclusively use all 16 PCI-e channels, or the channels will be divided amongst the requesting GPUs. The unfortunately reality is that one or more GPUs will experience delays while receiving data, resulting in delayed execution.

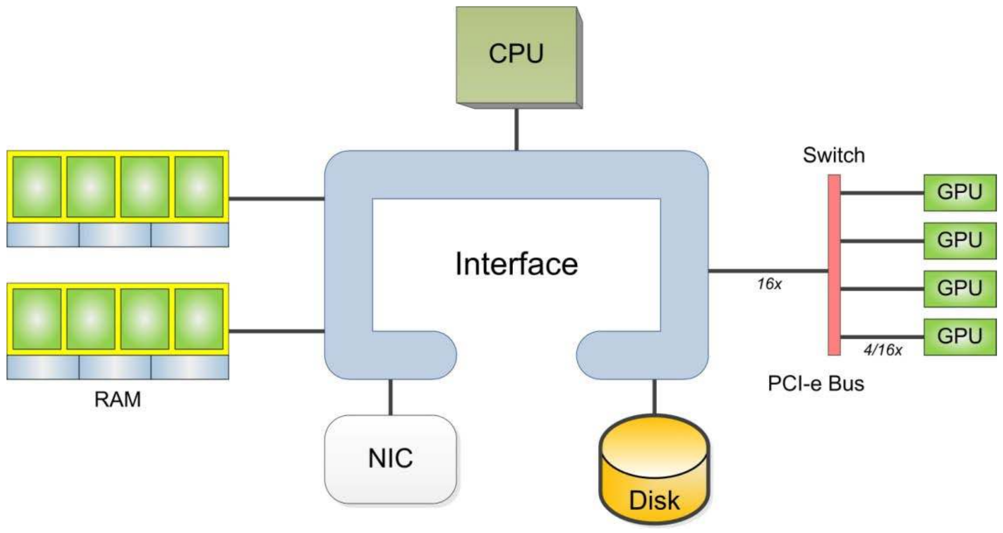


Figure 31: PCI Express Configuration (four-GPU System)

7.2.2.3 Taxonomy of GPU Kernels

To address this issue of data transfer, Gregg and Hazelwood [GrHa11] developed a taxonomy for GPU kernels, arguing that it is important to label them based off of their data transfer needs. Their taxonomy divides GPU kernels into the following five categories:

1. *Non-Dependent* (ND): Those kernels that are not dependent on the data transfer to or from the GPU, or those kernels that have a very small dependence (such as an initial seeding value or a value returned to the host as a result).
2. *Dependent-Streaming* (DS): Those kernels that are dependent on the data transfers between the host and device but hide this dependency by way of asynchronous streaming memory.
3. *Single-Dependent-Host-to-Device* (SDH2D): Those kernels that are dependent on the data transfers from the CPU to the GPU

4. *Single-Dependent-Device-to-Host* (SDD2H): Those kernels that are dependent on the data transfers from the GPU to the CPU
5. *Dual-Dependent* (DD): Those kernels that are dependent on the data transfers from the CPU to the GPU as well as the data transfers from the GPU to the CPU

On the extreme ends of the data transfer spectrum, we have ND (and DS) kernels, which are not data dependent, and then DD kernels, which are dependent on transfers in both directions. The difference in execution times between ND and DD kernels should be significant, especially when, for many applications, the data transfer times ultimately dominate the overall execution time. To this end, Gregg and Hazelwood [GrHa11] propose a GPU scheduler that would then utilize this taxonomy, determine the overall costs of launching the kernel on the GPU (as opposed to performing the function on the CPU), and then make a decision based on this new information.

7.2.3 Dealing with Large Data Sets

The modeling of large data sets presents additional challenges beyond those already discussed. In order to transfer data to the GPU, it must already be in system RAM. Many data sets in scientific computing have sizes exceeding that of system RAM. Transfer delays will therefore be incurred, because the paged data will need to be retrieved from disk before it can be sent to the GPU. The time required to transfer data into memory depends on several factors including the input data size (S_{input}), the amount of system RAM (S_{RAM}), and the data size that ultimately transfers to the GPU ($S_{transfer}$), and can be stated as

$$t_{disk} = \begin{cases} 0 & S_{input} < S_{RAM} \quad (a) \\ \dfrac{S_{input} - S_{RAM}}{TP_{disk}} & S_{RAM} < S_{input} + S_{transfer} \quad (b) \\ \dfrac{S_{transfer}}{TP_{disk}} & S_{RAM} + S_{transfer} < S_{input} \quad (c) \end{cases} \qquad \textbf{(8)}$$

where t_{disk} is the time taken for a unidirectional transfer between disk and RAM, S_{input} is the data set size, S_{RAM} is the size of the system RAM, $S_{transfer}$ is the size of the data set that needs to be transferred to the GPU, and TP_{disk} is the empirically calculated throughput of the disk being used. While the model is intended to be comprehensive, attempting to model the sheer number of intelligent disk options available is simply unnecessary for our purposes. As will be illustrated in Section 7.3, the throughput of the specific disk used will be empirically calculated and then used as needed in Equation 8. Equation 8(a) shows a transfer time of zero, because the entire data set fits into system RAM and does not require paging. In Equation 8(b), some of the data is in RAM and some requires paging due to its residing in disk storage. Finally, in Eq. 8(c), the entire data set must be paged in since it none of it is in RAM. Important to note is that since t_{disk} is a unidirectional transfer between disk and RAM, this cost will most likely occur several times during execution, of course, depending on the application and the data transfer requirements.

To facilitate the accurate estimation of disk paging in our performance model, we make the following assumptions:

1. As is the case with GEDS, we assume the GPU is processing streaming data. We can infer, therefore, that data access occurs on the most recently arriving (and least recently used) elements.
2. If system RAM cannot house the entire data set, the GPU will require data that is on disk. This translates to the paging out of obsolete data to disk storage and the paging

in of the requested data to RAM, which equates to two, unidirectional transfers (t_{disk}), as detailed in Eq. 8.

3. As is also the case with GEDS, many *Dual-Dependent* (DD) GPU applications perform multiple iterations, often resulting in continuous transfers between the host and device. The output data being sent back to the host will not need the use of paging, because any input data still in RAM can be invalidated by the OS. This will hold true as long as system RAM can house the entire output data set.

With these assumptions in place, our model is able to reasonably predict the impact of latencies due to disk paging.

7.2.4 Additional Model Considerations

Moving beyond the data transfer costs, there are several factors that affect the execution time on the GPU itself (β), including, but not limited to, the number of streaming multiprocessors, the number of CUDA cores per multiprocessor, the clock speed of the individual cores, the size of the memory, the speed of the memory, and the size of the shared memory. Clearly, parallel applications will find improvement with the addition of extra streaming multiprocessors or extra cores per multiprocessor, as this allows for the launching of even more concurrent threads. And it goes without stating that a faster core clock certainly increases performance. A faster memory, and one with a larger bandwidth, facilitates global loads and stores and decreases latency. Finally, a larger shared memory, per core, yields significant improvements for many applications, especially those that are bandwidth constrained.

7.2.5 Modeling GEDS

While our model is intended to be comprehensive and allows for large datasets (far exceeding the size of RAM), the simulations in Section 7.3 are based off of our GEDS framework, in which the entire dataset fits into memory. As a result, we will only need *Equation 8a* for the calculation of the disk latency (zero), and we will be able to ignore the contingencies provided by Equations 8b and 8c. Additionally, the two views that we materialize on the GEDS framework, Proximity Area and Neighboring Objects, result in a dual-dependent (DD) kernel. The GPU requires data from the CPU in order to begin processing, and the CPU then requires the result to be transferred back before it can be published as an output data stream. As a result, we see that data transfers between the host and device play a central role in the GEDS framework.

7.2.6 Summary

In this section, we describe the core of our performance model, which allows us to predict the GPU execution time of a problem running on *M* GPUs. Expectedly, this requires the use of a reference, single-GPU implementation, thereby allowing us to extrapolate the running time of a single mobile object and then extending that out over *M* GPUs. We also introduced techniques for modeling the PCI-Express interconnect and disk throughput. Equation 9 describes the overall communication cost and can be stated as

$$t_{comm} = \begin{cases} t_{DISK} + t_{PCI} & paged\ memory \\ t_{alloc\_pinned} + t_{PCI} & pinned\ memory \end{cases}, \qquad (9)$$

where t_{DISK} represents the combination of times from the unidirectional disk transfers (t_{disk}), the time needed to transfer data via the PCI Express is represented by t_{PCI} , and t_{alloc_pinned} is the time required for CUDA to make a pinned memory allocation. Referring back to Equation 5, which represents the overall execution time of our application,

$$t_{total} = \alpha + \varepsilon_{HD} + \beta + \varepsilon_{DH} \quad ,$$

these communication costs (t_{comm}) are included in both the time required to transfer the data from the host to the device (ε_{HD}) and the time required to transfer the data from the device back to the host (ε_{DH}). With the communication costs modeled and incorporated into Equation 5, we now predict execution time over *M* GPUs and compare it to the actual running times, allowing us to gauge the efficacy of our performance model.

7.3 <u>Results Using Performance Model</u>

We now evaluate our performance model by running a variety of GEDS simulations, comparing the predicted execution time to the actual execution time. To this end, we materialize the *Proximity Area* and *Neighboring Objects* views using both paged and pinned memory, and we also materialize these views without the use of GPU shared memory. For each simulation, we vary the number of mobile objects (size of the input data set), thereby allowing us to gauge how the input size affects the throughput over the PCI-express. Finally, we also consider performance when we remove the PCI-express, and therefore data transfers, out of the equation, simply illustrating the effectiveness of our approach for modeling pure GPU execution.

7.3.1 Theoretical Calculation Preliminaries

Before we can effectively predict execution time, we need to know both the basic specifications of the CPU and GPU in our system as well as any theoretical bandwidth calculations. As discussed in Chapter 5, Performance Studies, our testing computer is equipped with an Intel Core 2 Duo E8500 processor with 4GB of RAM connected to an Nvidia GTX 460 GPU via a PCI-e 2.0 x16 bus. The specifications of the GPU and the necessary bandwidth computations are shown in Table 8.

Table 8: System GPU Specifications

Processor Clock	1350 MHz
Memory Clock	1800 MHz
Memory Interface Width	256-bit
Memory Bandwidth	115.2 GB/sec
RAM Type	DDR (double data rate)
CUDA Cores	336

The theoretical, peak memory bandwidth is calculated using the hardware specifications shown in the table as follows:

$$\frac{\left(1800 \times 10^6 \times \left(\frac{256}{8}\right) \times 2\right)}{10^9} = 115.2\ GB/sec \quad .$$

We first convert the memory clock rate into hertz. This is then multiplied by the memory interface width, which is divided by 8, to convert the bits to bytes, and multiplied by 2 to account for the double data rate. Finally, this newly calculated result is divided by 10^9, allowing the memory bandwidth to be expressed in GB/sec. While knowing the clock speeds of the processors in our system perhaps allows for a finer granularity of our model, they simply are not required for our calculations. As described in Section 7.2.1, we calculate the query execution time of a single mobile object, for one unit of time, based off of the GPU execution time of a given reference problem. We therefore bypass the need for specific processor clock speeds, as they are utilized, indirectly, by way of the reference problem; we note that this also allows for a more generic approach in our model.

In order to predict the execution time required to materialize the *Proximity Area* view throughout the following simulations, we need to calculate the query execution time required to simply materialize said view over one mobile object for one unit of time. Because we want this result to only represent GPU execution time, independent of memory copying costs and CPU execution, we refer to the data from GEDSv2 (Table 6) as our reference GPU problem. From Equation 6, we can now calculate the execution time to materialize the *Proximity Area* view over one mobile object (and for one unit of time) as follows:

$$t_{mobileObject\_PA} = \frac{t_{gpu\_refprob}}{N_{mobileObjects}} = \frac{1.6\ ms}{10{,}000} = 0.00016\ ms$$

Similarly, in order to predict the execution time required to materialize the *Neighboring Objects* view, we need to calculate the query execution time required to simply materialize said view over one mobile object for one unit of time, and we do so as follows:

$$t_{mobileObject\_NO} = \frac{t_{gpu\_refprob}}{N_{mobileObjects}} = \frac{474\ ms}{10{,}000} = 0.0474\ ms$$

7.3.2 *Proximity Area* View Using Paged Memory

To predict the execution time required to materialize the *Proximity Area* view, a single-GPU simulation is used, which runs over a specified number of time units. The equations introduced in Section 7.2 allow us to compute the CPU and GPU execution times as well as the PCI-express transfer costs (disk paging is not required since the data set fits in RAM). Figure 32 depicts how we use these to predict the execution time for up to 8 GPUs (line 1).

***Proximity Area* View Prediction (Paged Memory)**

```
1    GPUs = 8
2    MOs = 1000
3    UNITS_OF_TIME = 30
4    N = 20                              # Assumed num. of neighbors
5
6    for i = MOs to 10000                # Predict for all MOs
7       for j = 1 to GPUs                # Accounts for mult. GPUs
8          DATA_SIZE = i * MO_SIZE       # Total size of data set
9          DATA_ON_GPU = DATA_SIZE / j   # Distribute data over GPUs
10         DATA_RETURNED = N * MO_SIZE / 2  # Result returned to device
11
12         CPU_TIME = ALPHA
13         GPU_TIME = T_MOBILEOBJECT_PA * MOs / j
14         T_DISK = 0
15         T_PCI = (DATA_ON_GPU / PCI_BAND)+(DATA_RETURNED / PCI_BAND)
16
17         TIME[j,i] = ( CPU_TIME + GPU_TIME + T_PCI ) * UNITS_OF_TIME
```

Figure 32: *Proximity Area* View Prediction (Paged Memory)

We start with 1000 mobile objects (line 2) and predict execution up to 10,000 mobile objects (line 5). The total size of the data set is determined based off of the number of mobile objects and the size of an individual mobile object to be transferred to the GPU (line 8). We then determine the data size per GPU simply by dividing by the number of GPUs in the system (line 9). For the *Proximity Area* view, the GPU calculates a two-dimensional matrix of the distances between each and every mobile object, and the neighbors of each mobile object, within the specified query radius, are returned to the host (line 10). The CPU execution time (line 12) is empirically measured from the reference implementation. Also from the reference implementation, we calculate the single-GPU execution time (line 13), which is determined based off of the time to materialize the *Proximity Area* view for one mobile object, `T_MOBILEOBJECT_PA`. As the entire data set fits into memory, disk paging is not required (line 14). The only communication costs are those incurred by data transfers across the PCI-express and are calculated by line 15. `MO_SIZE` and `PCI_BAND` are empirically calculated and represent constants in this model. Finally, the predicted execution time is determined by adding the CPU time, GPU time, and transfer costs, and then multiplying this result by the number of iterations (line 17). The remaining simulations used the same technique as described above and detailed in Figure 32. Figure 33 displays the actual execution time along with the predicted times and allows us to visualize the results of our performance model.

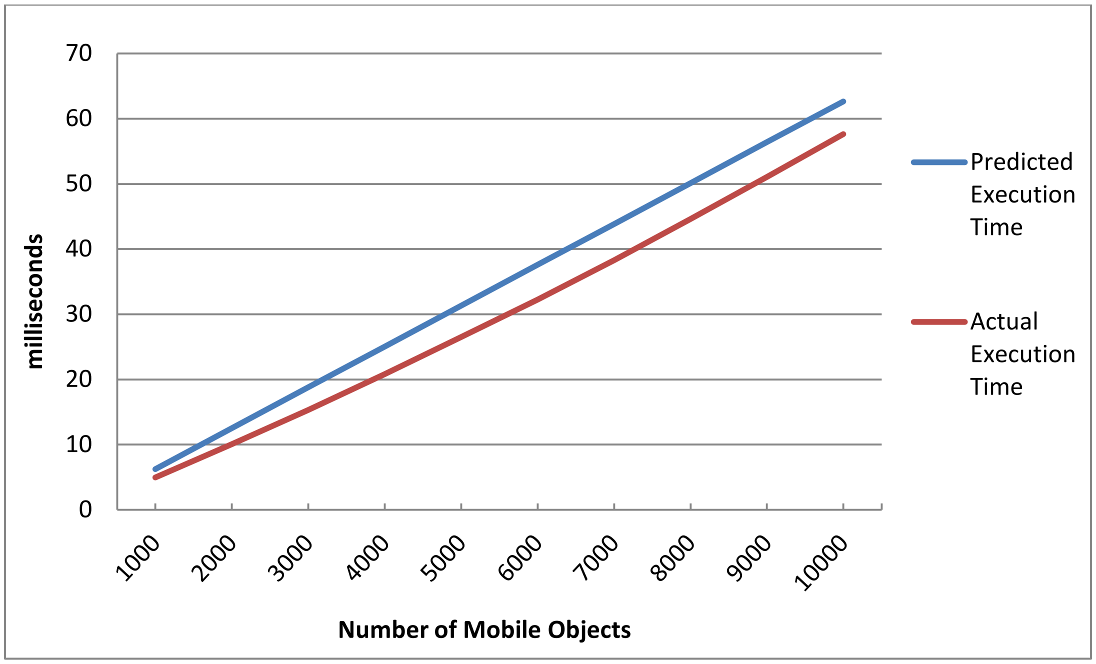


Figure 33: *Proximity Area* View Prediction Results (Paged Memory)

Upon initial review, the results are impressive, with the predicted execution time only slightly higher than the actual execution time. This indicates that the method for modeling GPU execution, as described in Section 7.2.1, is suitable for our model. While the actual and predicted execution times in Figure 33 are reasonably similar, a closer inspection of the data reveals an interesting trend. Figure 34 illustrates that as the number of mobile objects decrease, the percentage difference between the actual and predicted execution increases (Figure 34).

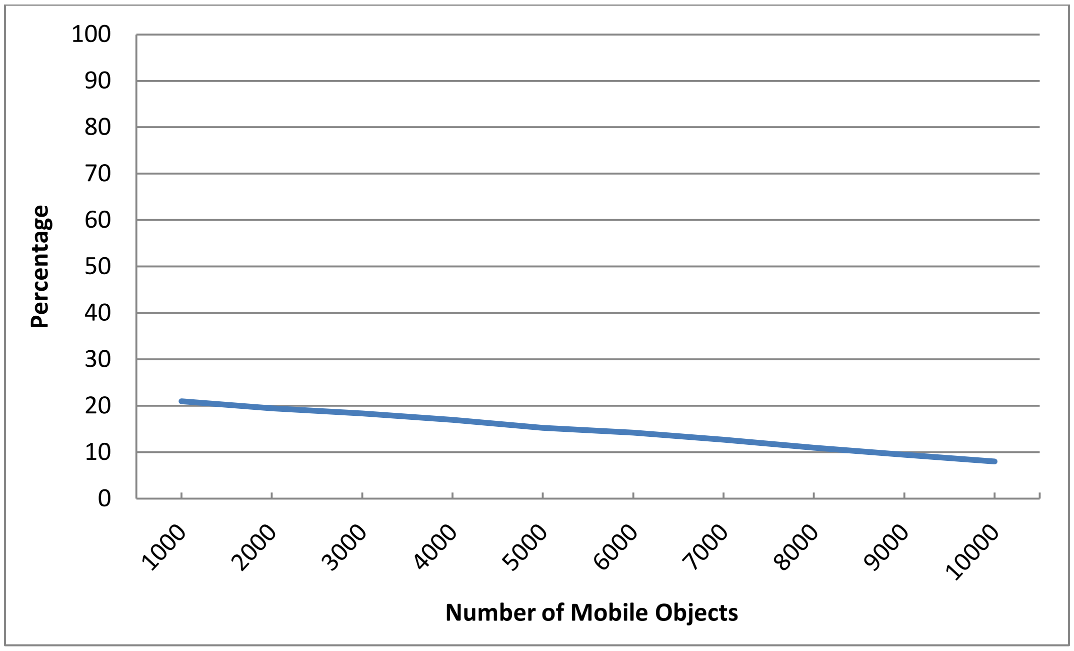


Figure 34: *Proximity Area* View Prediction Results (Paged Memory) (% Difference between Actual and Predicted Execution Time)

The most accurate results were when predicting execution time for 10,000 mobile objects. This makes sense, considering that our estimation for $t_{mobileObject\_PA}$, the query execution time required to materialize the *Proximity Area* view over one mobile object for one unit of time, was calculated based off of 10,000 mobile objects (from the GPU reference problem). We should expect, therefore, that our predictions will be most accurate when the number of mobile objects approaches ten thousand. Figure 34 also illustrates that the prediction accuracy degrades, at a linear rate, as the number of mobile objects decrease. However, we still achieve 79% accuracy with 1000 mobile objects.

7.3.3 *Proximity Area* View Using Pinned Memory

We also tested our performance model when using pinned memory for data transfers. The method for prediction is very similar to that detailed in Figure 32. However, we now need to consider the time required by the CUDA driver to allocate pinned memory, as well as the increased transfer speeds over the PCI-express. Figure 35 shows the results of the actual and predicted execution times. As expected, both are lower than their paged counterparts, and the percentage difference is similar as well.

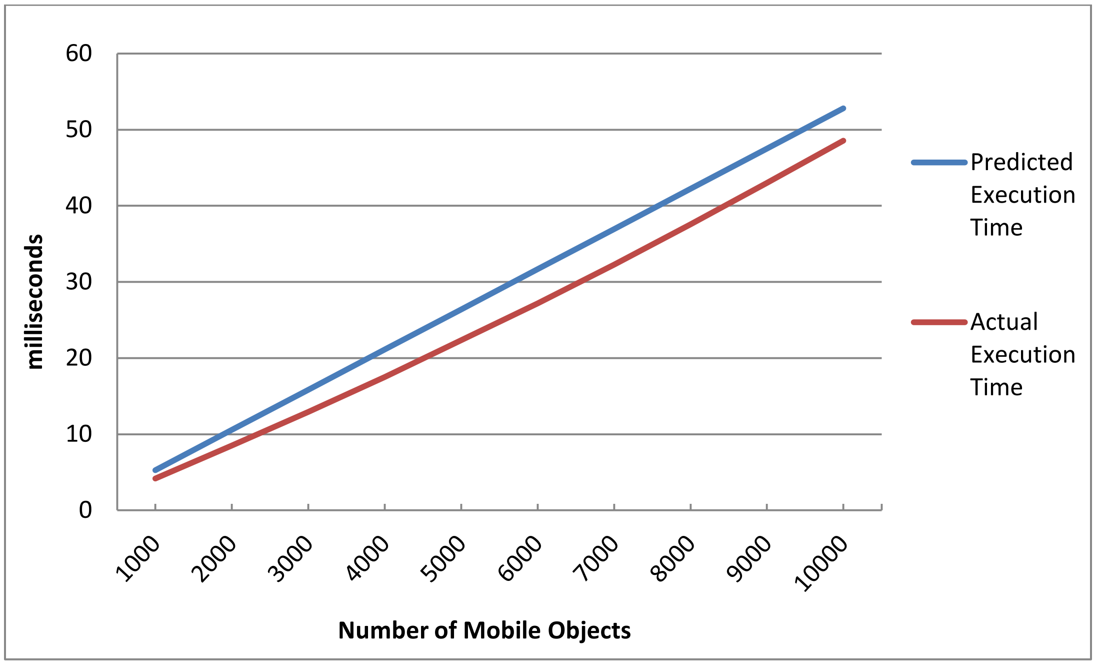


Figure 35: *Proximity Area* View Prediction Results (Pinned Memory)

7.3.4 *Proximity Area* View Without GPU Shared Memory

Our third simulation and prediction test materializes the *Proximity Area* view without using GPU shared memory. The prediction method used follows that of the last two examples. However, the value for $t_{mobileObject\_PA}$ almost triples to 3.8 ms, as it is based on a slower GPU reference problem (without shared memory). Figure 36 displays the results of our predicted execution times and the actual execution times. While the lower end of the accuracy did drop on this example (to 74%), the model is still performing quite well, reaching an accuracy of 87.8% for larger numbers of mobile objects.

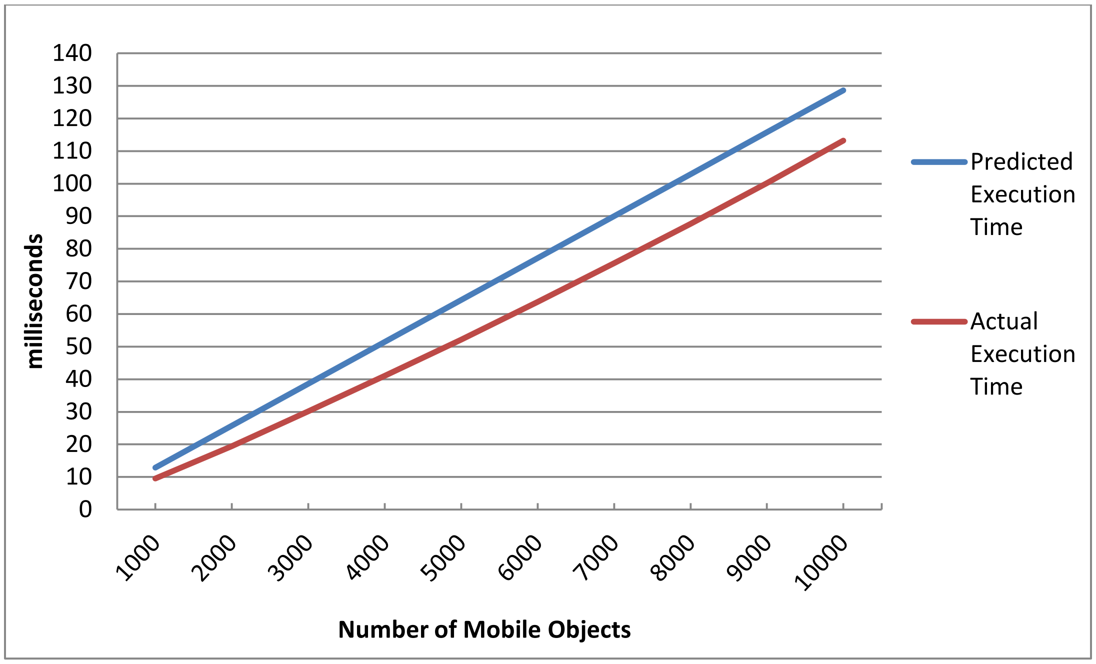


Figure 36: *Proximity Area* View Prediction Results (No GPU Shared Memory)

7.3.5 *Neighboring Objects* View Using Paged Memory

We now use our model to predict the execution time required to materialize the *Neighboring Objects* view, and, again, we start with a single-GPU implementation that runs over a specified number of time units. As discussed in Section 7.2.1, there are some algorithms whose GPU execution time does not scale linearly along with the mobile objects. Once such example is our *Neighboring Objects* view. While the number of mobile objects scales linearly, the GPU execution time increases by an order of magnitude. As such, we use the second method mentioned in Section 7.2.1 for predicting GPU execution time. From our reference problem, we calculated the time to materialize the *Neighboring Objects* view over one mobile object (and for one unit of time), $t_{mobileObject\_NO}$, to be 0.0474 milliseconds, and we use this figure in our model to calculate total GPU execution time.

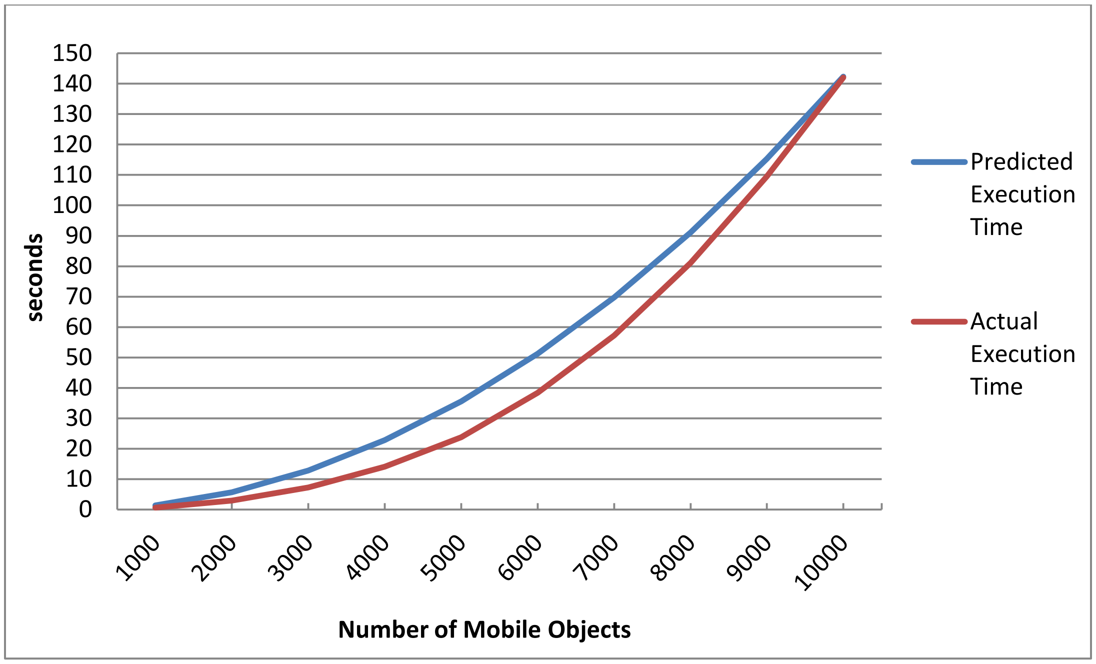


Figure 37: *Neighboring Objects* View Prediction Results (Paged Memory)

Figure 37 displays our predicted results versus the actual execution time. At 10,000 mobile objects, the prediction was 99% accurate. However, as the number of mobile objects decreased, the accuracy dropped off significantly; the predicted execution time for 1,000 mobile objects was only 48% accurate. The average accuracy of 73% was still reasonable and can be increased by coming up with a more accurate multiplier used in prediction.

7.3.6 *Neighboring Objects* View Using Pinned Memory

We also tested our performance model using pinned memory for data transfers while materializing the *Neighboring Objects* view. This prediction follows from the last example, except that we now need to consider the time required by the CUDA driver to allocate pinned memory, as well as the increased transfer speeds over the PCI-express. Figure 38 shows the results of the actual and predicted execution times. Even though pinned memory is used, the execution time is similar to that of its paged sibling. This is simply due to the fact that the overwhelming majority of time is spent actually on the GPU. So while the pinned memory does result in faster data transfers, those savings are barely visible in the final execution time.

7.3.7 *Neighboring Objects* View Without GPU Shared Memory

For our last simulation, we use our performance model to predict execution time to materialize the *Neighboring Objects* view without using GPU shared memory. The prediction method used follows that of the last two examples, with the exception that our empirically measured value, $t_{mobileObject\_NO}$, increases to 0.088 milliseconds due to the slower GPU reference problem. Figure 39 shows the actual and predicted results.

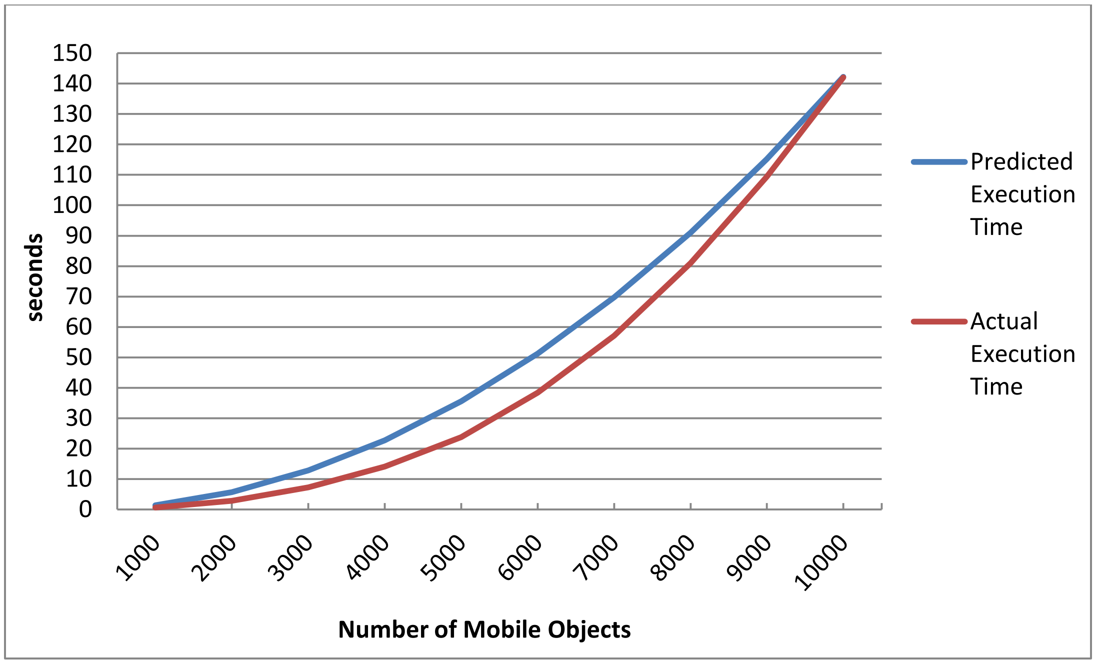


Figure 38: ***Neighboring Objects*** **View Prediction Results (Paged Memory)**

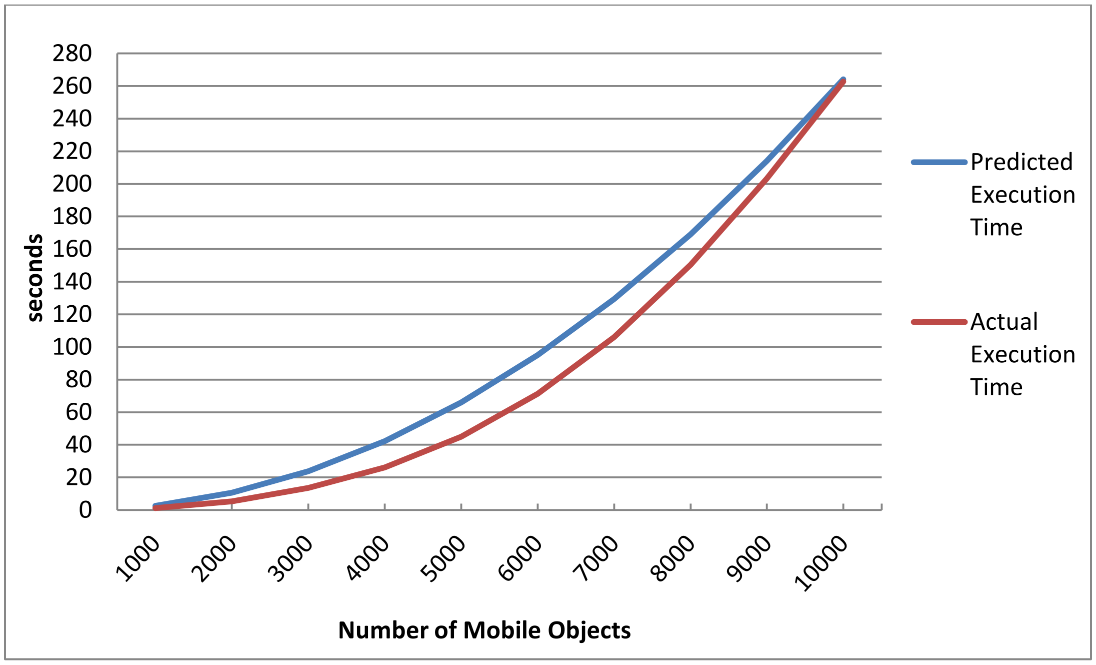


Figure 39: ***Neighboring Objects*** **View Prediction Results (No GPU Shared Memory)**

7.3.8 Reflections on Performance Model

Our performance model over the GEDS framework provides guidance on what algorithms and queries are most suitable for GPU execution. The performance equations introduced in Section 7.2 capture the GPU execution time along with many of the latencies found in the CPU-GPU relationship. While the original intention was to simply model algorithms over the GEDS framework, the generic nature of the model allows it to apply to many GPU-based applications. Finally, the simulations over this model did indeed illustrate one of the key points made with respect to GPU computing: pay attention to the data! As argued by Gregg and Hazelwood [GrHa11], having a solid understanding of the actual data and the transfer needs of the system is paramount to success. Even though GEDS falls into the DD (dual dependent) category, we focused on limiting the amount of data transferred back and forth between the host and the device, thereby allowing us to achieve significant speedups.

7.4 Class of Algorithms Best Suited for General Purpose Computing on the GPU

With a basic understanding of GPU architecture, it is immediately clear that any embarrassingly parallel algorithm can expect significant speedup when ported to the GPU. Beyond these obvious applications, we wish to pull from our model and develop a list of attributes common to successful GPU-based applications. In general, any application whose kernels will ultimately launch large numbers of parallel threads (numbering in the thousands) can expect performance increases. More specifically, if these algorithms exhibit data parallelism, in which many threads do similar work across the data set, further increases will be seen.

Additionally, any applications whose data exchange between threads can be localized to threads nearby in the kernel, thereby allowing the GPU to take advantage of its shared memory region, can certainly expect speedups.

On the flipside, our model allows us to identify attributes common to problems that are not suited for GPU-based computing. Clearly, applications with limited concurrency cannot be considered, as their serial nature limits the production of the GPU cores. Even if an application has many threads, but if all of the threads are doing different work (irregular task parallelism), the GPU will not be utilized efficiently. Finally, considering that the PCI-Express connection between the CPU and the GPU is, arguably, the greatest bottleneck, applications that require frequent communication between the host and device may be limited by the data transfers. The performance enhancement of the computation will ultimately be offset by the costs of communication between the two devices. In summary, the worst type of algorithms for the GPU are those with a small amount of parallelism, those with significant amounts of branching or synchronization, and those algorithms whose data transfer overhead outweighs the computation improvement of the GPU.

The ultimate question remains. Will an application benefit from porting portions of it to the GPU? Going back to Amdahl and Gustafson, the answer lies in how much we can reduce the percentage of the serial portion of the program. Assuming an application can launch enough concurrent threads, it should certainly see speedups as long as α is kept to a minimum. However, even if half of the program remains serial, significant speedups can be achieved, as evidenced by Gustafson's Law of Scaled Speedup. These scaled problem sizes reduce the impact of the program's serial portion on the overall execution time. Surprisingly, although developers must focus on maximizing the parallel portion their application, the serial portion may not be the greatest bottleneck. With modern applications often processing gigabytes and even terabytes of data, the PCI Express interconnect between the CPU and GPU will often be the limiting factor. As a result, best overall application performance is achieved by minimizing the data transfer between the CPU and GPU, even if this requires running kernels on the device that do not demonstrate any sizeable speed-up. To this end, CUDA does allow for asynchronous transfers that overlap with computation. Unfortunately, most developers exclusively use the `cudaMemcpy()` command, which is a blocking transfer; the host most wait for the device to completely receive the data. The `cudaMemcpyAsync()` function is non-blocking and allows control to be immediately returned to the host. The only caveat is that asynchronous transfers require page-locked, or pinned, host memory. If overused, overall system performance can be reduced due to the scarce nature of pinned memory.

CHAPTER 8:
CONCLUDING REMARKS AND FUTURE WORKS

8.1 Concluding Remarks

A variety of research exists for the processing of continuous queries in large, mobile environments. Each method tries, in its own way, to address the computational bottleneck of constantly processing so many queries. For this research, we present a two-pronged approach at addressing this problem. Firstly, we introduce an efficient and scalable system for monitoring traditional, continuous queries by leveraging the parallel processing capability of the Graphics Processing Unit. We examine a naive CPU-based solution for continuous range-monitoring queries, and we then extend this system using the GPU. Additionally, with mobile communication devices becoming commodity, location-based services will become ubiquitous. To cope with the very high intensity of location-based queries, we propose a view oriented approach of the location database, thereby reducing computation costs by exploiting computation sharing amongst queries requiring the same view. Our studies show that by exploiting the parallel processing power of the GPU, we are able to significantly scale the number of mobile objects, while maintaining an acceptable level of performance.

Our second approach was to view this research problem as one belonging to the domain of data streams. Several works have convincingly argued that the two research fields of spatio-temporal data streams and the management of moving objects can naturally come together. [IlMI10, ChFr03, MoXA04] For example, the output of a GPS receiver, monitoring the position of a mobile object, is viewed as a data stream of location updates. This data stream of location updates, along with those from the plausibly many other mobile objects, is received at a centralized server, which processes the streams upon arrival, effectively updating the answers to the currently active queries in real time.

For this second approach, we present GEDS, a scalable, Graphics Processing Unit (GPU)-based framework for the evaluation of continuous spatio-temporal queries over spatio-temporal data streams. Specifically, GEDS employs the computation sharing and parallel processing paradigms to deliver scalability in the evaluation of the proposed, continuous spatio-temporal views: *Proximity Area* and *Neighboring Objects*. The GEDS framework utilizes the parallel processing capability of the GPU, a stream processor by trade, to handle the computation required in this application. Experimental evaluation shows promising performance and shows the scalability and efficacy of GEDS in spatio-temporal data streaming environments. Additional performance studies demonstrate that, even in light of the costs associated with memory transfers, the parallel processing power provided by GEDS clearly counters and outweighs any associated costs.

Finally, in an effort to move beyond the analysis of specific algorithms over the GEDS framework, we take a broader approach in our analysis of GPU computing. What algorithms are appropriate for the GPU? What types of applications can benefit from the parallel and stream processing power of the GPU? And can we identify a class of algorithms that are best suited for GPU computing? To answer these questions, we develop an abstract performance model, detailing the relationship between the CPU and the GPU. We model the execution time of a single mobile object over one unit of time, by using the total GPU execution time of a reference problem. From this, we can then extrapolate the total execution time across *M* GPUs. Our model also details the communication bottleneck of the PCI-express, describes the pros and cons of paged versus pinned memory, and takes into account disk latencies when dealing with large data sets. Finally, to gauge the efficacy of our model, we then run a variety of simulations, comparing the actual run-time to the model-based, theoretical run-time. From this model, we are able to extrapolate a list of attributes common to successful GPU-based applications, thereby providing insight into which algorithms and applications are best suited for the GPU and also providing an estimated theoretical speedup for said GPU-based applications.

8.2 Future Works

With the advent of new technologies, the domain of mobile computing is constantly evolving, resulting in many new and interesting research areas. My future work will extend from the research detailed in this dissertation and will focus on designing efficient GPU-based algorithms to solve several computationally intensive problems within the domain of location-based services.

8.2.1 Incorporating Previous Methods into GEDS Framework

Although the previous solutions, which addressed query processing in mobile environments, were indeed limited by use of a CPU, the proposed methods were certainly novel and state-of-the-art for their time. New technologies, increases in communication bandwidth, and the overall ubiquitous nature of mobile devices indicate that mobile computing will be a focus for some time yet, and the pressure to provide real-time query results will only expand.

Indeed, with the increased use of smart phones and the ever-increasing use of GPS enabled applications, there may come a time where even the speedup provided by a GPU (GEDS) is not sufficient. One immediate future extension is to incorporate some of the previous methods, such as trajectories, incremental evaluation, spatial join of mobile objects and queries, distributed solutions, etc., into the GEDS framework. We can pull from the ideas of previous, successful research and attempt to amalgamate them, if feasible, into our framework. While this idea sounds intriguing in theory, the challenge, as always, is whether or not we can successfully implement said solutions on the GPU, and, even if possible, the second challenge is the difficulty of the GPU-based algorithmic design required.

8.2.2 Processing Approximate Spatio-Temporal Queries

In order to provide accurate and real-time results to queries over the GEDS framework, we make a common assumption found in research, namely, that the mobile objects are constantly sending location updates, which results in the query processor having accurate location information. While many devices do continuously perform location updates with the central server, many other devices do not, and this occurs for a variety of reasons. Previous arguments were that the constant location updates required too much battery consumption. Also pointed out was the communication bottleneck of a central server, which is supposed to receive and process these continuous location updates. Even if all mobile devices constantly performed location updates, the server may not be capable of processing them efficiently. Finally, the location update, once processed, may already be outdated simply due to the mobility of the device in question; it may have moved, perhaps considerably depending on its velocity. All of this points to the fact that there are environments where location updates are only periodically received and may be outdated when required by the query processor. To this end, instead of processing exact spatio-temporal queries based off of accurate location information, another research direction is to process approximate spatio-temporal queries where the query processor only has approximate location information. Understandably, the computation required in this environment is increased due to the uncertainty of the mobile device locations. The GPU is certainly the processor of choice in this environment, as already evidenced in research. The challenge will be in the development of parallel algorithms that can solve the approximate queries.

8.2.3 Private Query Processing in Location-based Services

The idea of preserving one's privacy, while performing queries (service requests), has been an intense area of research for the last few years. The classical approach is that of *pseudonymity*, or anonymity, where users simply employ a fake identity when making queries. This approach, unfortunately, has been shown to be inherently flawed. As an example, if a customer anonymously inquires as to the nearest delivery restaurants in proximity to a particular home address, there is a high statistical chance that the customer is indeed the homeowner, with his identity now revealed. Several solutions have been proposed, with many focusing on the architectural components to preserve privacy, others on the network and broadcasting part of the equation, and still others on the software backend that answers these queries. From the software perspective, solutions include blurring one's own identity. So instead of sending an exact location update to the server, an approximate, or blurred, location is sent instead. Now we have an environment where, for the purpose of privacy, all mobile objects are intentionally blurring their locations, and these same objects expect highly-accurate query results. Unquestionably, this is a challenge even for CPU-based solutions. Important to note is the connection between this possible research area and the last one mentioned. For the processing of approximate spatio-temporal queries, the query processor does not have exact location information. Rather, it only has an approximate location for each object. Now in this case, the mobile objects are intentionally blurring themselves and presenting blurred, or approximate, locations to the query processor. So although these are different research areas, the problem is simply one of answering queries over approximate, or blurred, locations. It may be possible that one cleverly designed algorithm can solve both research problems.

REFERENCES

[ACCC03] Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B. 2003. Aurora: A new model and architecture for data stream management. VLDB J. 12(2), 120–139.

[AdVe04] V. Adve and M. Vernon. Parallel program performance prediction using deterministic task graph analysis. ACMTransactions on Computer Systems 22 (1), pp 94–136, 2004.

[AGSY08] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In Proceedings of the 17th International Symposium on High Performance Distributed Computing, Boston, MA, June 2008, pp. 165–174.

[Amd67] Amdahl, G. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings, pp. 483–485.

[Ang98] C. Anglano. Predicting parallel applications performance on non-dedicated cluster platforms. In Proceedings of the 12th international conference on Supercomputing, 1998.

[BaSk10] P. Bakkum and 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. ACM, 2010, pp. 94–103.

[BaWi01] Babu, S., Widom, J. 2001. Continuous queries over data streams. SIGMOD Record 30(3), 109–120.

[BBCC10] M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar. Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory. In SPAA: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, June 2010, pp. 82–91.

[BCMK10] Jie Bao, Chi-Yin Chow, Mohamed F. Mokbel, and Wei-Shinn Ku. "Efficient Evaluation of k-Range Nearest Neighbor Queries in Road Networks". In Proceedings of the International Conference on Mobile Data Management, MDM 2010, Kansas City, MO, May 2010.

[BeKK96] Berchtold, S., Keim, D. A., and Kriegel, H. 1996. The X-tree: An index structure for high-dimensional data. In Proceedings of the 22th International Conference on Very Large Data Bases (VLDB). Morgan Kaufmann, San Francisco, CA, 28–39.

[BJKS02] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In IDEAS, 2002.

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD). ACM Press, New York, NY, 322–331.

[CaGu10] Cazalas, J. and Guha, R. 2010. GEDS: GPU Execution of Continuous Queries on Spatio-Temporal Data Streams. In Proceedings of the IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing (EUC), pp. 112 - 119.

[CaGu11] Cazalas, J. and Guha, R. 2011. Leveraging Computation Sharing and Parallel Processing in Location-Dependent Query Processing. The Journal of Supercomputing, online July, 2011.

[CaHu02] Ying Cai; Hua, K.A., "Managing continuous range queries in mobile databases," Mobile and Wireless Communications Network, 2002. 4th International Workshop, pp. 441-445, 2002.

[CaHu06] Ying Cai; Hua, K.A.; Guohong Cao; Xu, T. 2006. Real-time processing of range-monitoring queries in heterogeneous mobile databases. Mobile Computing, IEEE Transactions on, vol.5, no.7, pp. 931-942.

[CaHu09] Cazalas, J. and Hua, K. 2009. Leveraging Computation Sharing and Parallel Processing in Location-Based Services. In Proceedings of the 2009 International Conference on Computational Science and Engineering, pp. 221-228.

[ChCZ09] Chang, Y. F., Chen, C. S., and Zhou, H. 2009. Smart phone for mobile commerce. Computer Standards & Interfaces. Volume 31, Issue 4, pp. 740-747.

[ChFr03] Chandrasekaran, S., Franklin, M.J. 2003. PSoup: A system for streaming queries over streaming data. VLDB J. 12(2), 140–156.

[CoMo09] J. Cohen and M. Molemaker. A fast double precision CFD code using CUDA. In Parallel Computational Fluid Dynamics: Recent Advances and Future Directions, Moffett Field, CA, May 2009, pp. 414–429.

[DoVK10] G. Dotzler, R. Veldema, and M. Klemm. JCUDAmp: OpenMP/Java on CUDA. In 3rd International Workshop on Multicore Software Engineering, May 2010, pp. 10–17.

[FQKY04] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In ACM/IEEE Conference on Supercomputing, Pittsburgh, PA, November 2004, pp. 47–58.

[GeLi06] Gedik, B.; Ling Liu. 2006. MobiEyes: A Distributed Location Monitoring Service Using Moving Location Queries. Mobile Computing, IEEE Transactions on, vol.5, no.10, pp.1384-1402.

[GELA10] Ghanem, T. M., Elmagarmid, A. K., Larson, P., Aref, W. G. 2010. Supporting views in data stream management systems. ACM Transactions on Database Systems (TODS). Volume 35, Issue 1, Article 1.

[GHMA07] Thanaa M. Ghanem, Moustafa A. Hammad, Mohamed F. Mokbel, Walid G. Aref, Ahmed K. Elmagarmid. 2007. Incremental Evaluation of Sliding-Window Queries over Data Streams, IEEE Transactions on Knowledge and Data Engineering, v.19 n.1, p.57-72.

[GGKM06] N. Govindaraju, J. Gray, R. Kumar and D. Manocha. 2006. GPUTeraSort: high performance graphics coprocessor sorting for large database management. SIGMOD.

[GLWL04] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. 2004. Fast computation of database operations using graphics processors. SIGMOD.

[GoKM06] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In Proceedings of the ACM SIGMOD International Conference on Management of Data, June 2006, pp. 325–336.

[GrCo05] D.A. Grove, P.D. Coddington. Modeling message-passing programs with a performance evaluating virtual parallel machine. Journal of Performance Evaluation. Volume 60, Issue 1-4, pp 165–187, 2005.

[GrHa11] Gregg, C., Hazelwood, K. 2011. Where is the Data? Why you cannot debate CPU vs. GPU performance without the answer. In Proceedings of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software, (ISPASS), pp. 134-144.

[GSCP10] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In Architectural Support for Programming Languages and Operating Systems, Pittsburgh, PA, March 2010, pp. 347–358.

[Gus88] Gustafson, J. L. 1988. Reevaluating Amdahl's Law. Communications of the ACM, vol. 31, pp. 532–533.

[Gut84] Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD). ACM Press, New York, NY, 47–57.

[HaSa99] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. ACM TODS, 24(2):265-318, 1999.

[HaWa90] Hartzman, C. S. and Watters, C. R. 1990. A relational approach to querying data streams. IEEE Trans. Knowl. Data Eng. 2, 4 (Dec.), 401–409.

[HuJe04] Huang, X. and Jensen, C. S. 2004. Towards a streams-based framework for defining location-based queries. In Proceedings of the 2nd Workshop on Spatio-Temporal Database Management (STDBM). 73–80.

[HYFL08] He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. 2008. Relational joins on graphics processors. In Proceedings of the 2008 ACM SIGMOD international Conference on Management of Data.

[IlMI10] Ilarri, S., Mena, E., and Illarramendi, A. 2010. Location-dependent query processing: Where we are and where we are heading. ACM Computing Surveys (CSUR), Volume 24, Issue 3, Article no. 12.

[IwSK03] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In VLDB, 2003

[KaPH04] Kalashnikov, D. V., Prabhakar, S., and Hambrusch, S. E. Main Memory Evaluation of Monitoring Queries Over Moving Objects. Distrib. Parallel Databases, pp 117 – 135, Mar. 2004

[KoGT99] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In STDM, 1999.

[KOTZ04] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. Approximate NN queries on streams with guaranteed error/performance bounds. In VLDB, 2004.

[KrSe09] Krämer, J., Seeger, B. 2009. Semantics and implementation of continuous sliding window queries over data streams, ACM Transactions on Database Systems (TODS), v.34 n.1, p.1-49.

[KwLL02] Kwon, D., Lee, S., and Lee, S. 2002. Indexing the current positions of moving objects using the lazy update R-tree. In Proceedings of the 3rd International Conference on Mobile Data Management (MDM). IEEE Computer Society Press, Los Alamitos, CA, 113–120.

[LHJCT03] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In VLDB, 2003.

[LiHX08] Fuyu Liu; Hua, K.A.; Fei Xie, "On Reducing Communication Cost for Distributed Moving Query Monitoring Systems," Mobile Data Management, 2008. MDM '08. 9th International Conference on , pp.156-164, 27-30 April 2008.

[LiSS08] M. D. Lieberman, J. Sankaranarayanan, H. Samet. 2008. A fast similarity join algorithm using graphics processing units. ICDE.

[Mac03] Michael Macedonia. 2003. The GPU Enters Computing's Mainstream. Computer, vol. 36, no. 10, pp. 106-108.

[MoAr08] Mokbel, M. F. and Aref, W. G. 2008. SOLE: Scalable on-line execution of continuous queries on spatiotemporal data streams. VLDB J. 17, 5 (Aug.), 971–995.

[MoHP05] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In SIGMOD, 2005.

[Moor65] G. E. Moore. Cramming more components onto integrated circuits. Electronics, vol. 38, pp. 114–117, April 1965.

[Moor75] Moore, G.E. Progress in digital integrated electronics. Electron Devices Meeting, 1975 International, Vol 21, 1975, pp. 11- 13

[MoXA04] Mokbel, M. F., Xiong, X., and Aref, W. G. 2004. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD). ACM Press, New York, NY, 623–634.

[MPBT05] Mouratidis, K.; Papadias, D.; Bakiras, S.; Yufei Tao, "A threshold-based algorithm for continuous monitoring of k nearest neighbors," Knowledge and Data Engineering, IEEE Transactions on, vol.17, no.11, pp. 1451-1464, Nov. 2005.

[MXHA05] Mokbel, M. F., Xiong, X., Hammad, M. A., and Aref, W. G. 2005. Continuous query processing of spatiotemporal data streams in PLACE. GeoInformatica 9, 4 (Dec.), 343–365.

[NKPP00] G. Nudd, D. Kerbysin, E. Papaefstathiou, S. Perry, J. Harper, D. Wilcox. PACE - a toolset for the performance prediction of parallel and distributed systems. The International Journal of High Performance Computing Applications 14 (3), pp 228–251, 2000.

[OHLG08] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. Proceedings of the IEEE, vol. 96, no. 5, pp. 879–899, May 2008.

[PaHe93] Patterson, D., Hennessy, J. Computer Organization and Design: The Hardware/Software Interface. Computer Organization and Architecture. San Francicso, CA. Morgan Kaufmann Publishers Inc, 1993.

[PaSe04] Patroumpas, K. and Sellis, T. K. 2004. Managing trajectories of moving objects as data streams. In Proceedings of the 2nd Workshop on Spatio-Temporal Database Management (STDBM). 41–48.

[PrSt05] B. Predic, D. Stojanovic, A framework for handling mobile objects in location based services, in: Proceedings AGILE Conference, 2005, pp. 419–427.

[PXKA00] S. Prabhakar, Y. Xia, D. Kalashnikov, W.G. Aref, and S. Hambrusch, “Queries as Data and Expanding Indexes: Techniques for Continuous Queries on Moving Objects,” in TR., Dept. of Computer Science, Purdue Univ., 2000.

[PXKA02] S. Prabhakar, Y. Xia, D. Kalashnikov, W.G. Aref, and S. Hambrusch, “Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects,” IEEE Trans. Computers, vol. 15, no. 10, pp. 1124-1140, Oct. 2002

[RaPM03] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. GeoInformatica, 7(2):113–137, 2003.

[RoKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In SIGMOD Conference, 1995.

[SaVa08] H.A. Sanjay and S. Vadhiyar. Performance modeling of parallel applications for grid scheduling. Journal of Parallel and Distributed Computing. Volume 68, Issue 8, 2008.

[ScKa09] D. Schaa and D. Kaeli. Exploring the multiple-GPU design space. In International Parallel and Distributed Processing Symposium., May 2009, pp. 1–12.

[SJLL00] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In SIGMOD Conference, 2000.

[SPPD07] D. Stojanovic, A.N. Papadopoulos, B. Predic, S. Djordjevic-Kajan, A. Nanopoulos: "Continuous Range Query Monitoring of Mobile Objects in Road Networks", Data and Knowledge Engineering, Special Issue with Selected Papers from the 8th International Conference on Enterprise Information Systems (ICEIS), 2007.

[SPPD08] Stojanovic, D., Papadopoulos, A. N., Predic, B., Djordjevic-Kajan, S., and Nanopoulos, A. Continuous range monitoring of mobile objects in road networks. Data Knowl. Eng. Jan. 2008, pp. 77-100.

[SPTL04] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the past, the present and the future in spatio-temporal databases. In ICDE, 2004.

[TaPa02] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In SIGMOD Conference, 2002.

[TaPS03] Yufei Tao, Dimitris Papadias, Jimeng Sun: The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. VLDB 2003: 790-801.

[TFPL04] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In SIGMOD Conference, 2004

[TWHC04] Trajcevski, G., Wolfson, O., Hinrichs, K., and Chamberlain, S. 2004. Managing uncertainty in moving objects databases. ACM Trans. Database Syst. 29, 3 (Sep. 2004), 463-507.

[WaZK06] Haojun Wang, Roger Zimmermann, Wei-Shinn Ku: Distributed Continuous Range Query Processing on Moving Objects. DEXA 2006: 655-665.

[WCKY02] Wolfson, O., Chamberlain, S., Kalpakis, K., and Yesha, Y. 2002. Modeling Moving Objects for Location Based Services. In Revised Papers From the NSF Workshop on Developing An infrastructure For Mobile and Wireless Systems. Lecture Notes In Computer Science, vol. 2538. Springer-Verlag, London, 46-58.

[WJSC99] Wolfson, O., Jiang, L., Sistla, A. P., Chamberlain, S., Rishe, N., and Deng, M. 1999. Databases for Tracking Mobile Units in Real Time. In Proceedings of the 7th international Conference on Database theory. Lecture Notes In Computer Science, vol. 1540. Springer-Verlag, London, 169-186.

[WSCY99] Wolfson, O., Sistla, A. P., Chamberlain, S., and Yesha, Y. 1999. Updating and Querying Databases that Track Mobile Units. Distrib. Parallel Databases 7, 257-387.

[WoIU08] Wong Cheow Yuen; Ibrahim, H.; Udzir, N.I., "Distributed Real-Time Processing of Range-Monitoring Queries in Heterogeneous Mobile Databases," ICCIT 2008, pp.74-81.

[XiMA05] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In ICDE, 2005.

[XuZS96] Z. Xu, X. Zhang, L. Sun. Semi-empirical multiprocessor performance predictions. Journal of Parallel and Distributed Computing 39 (1), pp14–28, 1996.

[YaSV06] J. Yagnik, H.A. Sanjay, S. Vadhiyar. Performance modeling based on multidimensional surface learning for performance predictions of parallel applications in non-dedicated environments. In Proceedings of the International Conference on Parallel Processing, 2006.

[YaZS96] Y. Yan, X. Zhang, Y. Song. An effective and practical performance prediction model for parallel computing on nondedicated heterogeneous NOW. Journal of Parallel and Distributed Computing. Volume 38, Issue 1, pp 63–80, 1996.

[YuPK05] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In Proc. ICDE, 2005.

[ZZPT03] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In Proc. SIGMOD, 2003.