Towards Efficient Private Spatial Information Retrieval Using GPUs

Mihai Maruseac[1], Gabriel Ghinita[1], Ming Ouyang[1], Razvan Rughinis[2]
[1]UMass Boston, [2]Politehnica University
{mmarusea,gghinita,mouyang}@cs.umb.edu, razvan.rughinis@cs.pub.ro

ABSTRACT

Latest generation mobile devices allow users to receive services tailored to their current locations. Location-based service providers perform spatial queries based on the user locations, but may also share them with various third parties. User whereabouts may disclose sensitive details about an individual's health status, political views or lifestyle choices, and therefore must be thoroughly protected. *Private information retrieval (PIR)* methods support blind execution of range and NN queries with cryptographic-strength security, but incur significant performance overhead. We employ graphical processing units (GPUs) to speed up the crypto operations required by PIR. We identify the challenges that arise when using GPUs for this purpose, and we propose solutions to address them. To the best of our knowledge, this is the first work to use GPUs for efficient private spatial information retrieval, and an important first step towards GPU-based acceleration of a broader range of secure spatial data operations.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Security, Experimentation

Keywords

Location Privacy, Private Information Retrieval, GPU

1. INTRODUCTION

Location-based *service providers (SP)* support spatial queries, e.g., range, nearest-neighbor (NN), based on users' locations. However, SPs may not be trustworthy, and may share locations with various third parties for commercial profit. The whereabouts of an individual may disclose sensitive details about a person's private life [3], e.g., health status, political and religious views, etc.

Several categories of techniques address the location privacy problem. In dummy-generation techniques [4] each user sends the SP a number of fake locations together with the real one. The SP processes the query for each location. However, a sophisticated adversary can easily filter out fake locations based on knowledge of the road network and typical movement properties. Generalization techniques build a *cloaking region (CR)* including the query source and at least $k-1$ *real* users, according to the spatial k-anonymity principle [3]. The SP should not be able to distinguish among the k users in the CR. However, as discussed in [2] CR-based techniques fail to protect against adversaries with background knowledge, and are not suitable for continuous queries by moving users. In cryptographic techniques [2, 1] users send to the SP only their encrypted coordinates. Next, the SP executes a *private information retrieval (PIR)* protocol that blindly processes the query, and returns as result an encrypted token that only the client can decode to obtain the query answer. Techniques in this category are provably secure, but are computationally expensive, as they require large amounts of cryptographic primitive evaluations.

We investigate the use of *graphics processing units (GPUs)* and *Compute Unified Device Architecture (CUDA)* to speed up the execution of private spatial information retrieval techniques. GPU devices consist of large numbers (i.e., thousands) of simple computing cores that are able to perform basic operations in parallel. However, the execution and programming model of GPUs is significantly different than general-purpose CPUs. GPUs have small amounts of memory resources, and rigid patterns of data access that must be followed for fast performance. Deploying applications for GPUs is a challenging task that requires specific design choices and optimizations for each individual problem.

To the best of our knowledge, this work is the first attempt to use GPUs for private spatial information retrieval. We focus on the problem of privately answering NN queries [2], whereby an NN query is first transformed into a query-by-index, and then an instance of the *Kushilevitz-Ostrovsky (KO)* protocol [5] is executed. Nevertheless, our techniques can be applied to a broader spectrum of problems that are amenable to the transformation of queries-by-content to queries-by-index (e.g., range, top-k queries).

Our specific contributions are:

- We propose acceleration of private spatial information retrieval using GPUs. This work is an important first step towards efficient secure processing of spatial data, and the design principles outlined here can be extended to a broader spectrum of secure location-based queries.
- We focus on the specific technique of answering private NN queries using Voronoi diagrams and the KO protocol, and provide details for deploying this technique on GPUs.
- We provide experimental evaluation results which show that using GPUs can lead to significant improvement in PIR performance compared to CPU implementations.

The rest of the paper is organized as follows: Section 2 presents necessary background information on PIR and GPUs. Section 3 presents the challenges that arise when deploying PIR protocols on GPUs, as well as our solution to address them. We present experimental evaluation results in Section 4. Concluding remarks and directions for future work are summarized in Section 5.

2. BACKGROUND

Private Nearest Neighbor Retrieval. *Private Information Retrieval (PIR)* protocols allow clients (we use the terms *user* and *client* interchangeably) to retrieve an object X_i from a set $X =$

SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA

http://dx.doi.org/10.1145/2666310.2666431.

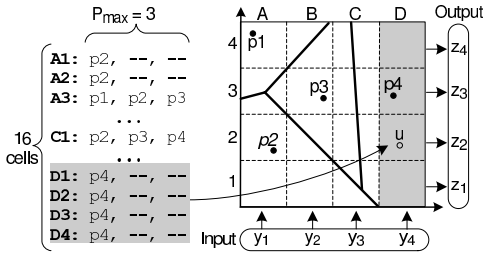


Figure 1: NN PIR with Voronoi Diagrams

$\{X_1 \dots X_n\}$ stored by a server, without the server learning the value of i. *Kushilevitz-Ostrovsky (KO)* is a computational PIR protocol [5] that relies on the *Quadratic Residuosity Assumption (QRA)*, which states that it is computationally hard to find the quadratic residues in modulo arithmetic of a large composite number $N = q_1 \times q_2$ where q_1, q_2 are large secret primes.

Specifically, given a number $y \in \mathbb{Z}_N^{+1}$ ($\mathbb{Z}_N^{+1}$ is the sub-set of $\mathbb{Z}_N$ for which the Jacobi symbol is $+1$) it is computationally hard (without knowing the factorisation of N) to determine whether y is a quadratic residue (QR) (i.e., $\exists x \in \mathbb{Z}_N | y = x^2 \mod N$) or a non-residue (QNR). Assume that all objects in X are bits. The client sends the server an ordered *query array* of n numbers $Y = [y_1 \cdots y_n]$, such that y_i is QNR and all others are QR. The server performs a *masked* multiplication of values in Y, i.e., it multiplies together only the y_j values for which $X_j = 1$. The client, who knows the factorisation of N, can determine that if the result of the multiplication is QNR, then $X_i = 1$, otherwise $X_i = 0$.

The ExactNN method from [2] builds on the KO protocol to answer NN queries. KO supports only queries-by-index (i.e., return i^{th} element), while in practice most queries are content-based. To bridge this gap, ExactNN builds the Voronoi tessellation of the set of data points, and then overlays a regular grid on top of it, as shown in Figure 1. A Voronoi tessellation is a set of polygonal cells, one for each data point, with the property that all locations in the cell of a point have that point as nearest neighbor. The contents of each grid cell is set to contain all the points for which the corresponding Voronoi cells intersect that grid cell.

In Figure 1, there are four points of interest $p_1 \dots p_4$, and hence four Voronoi cells. The regular grid has 16 cells. Grid cell $A1$ contains only point p_2, since that is the only Voronoi cell that intersects $A1$. On the other hand, cell $C1$ contains p_2, p_3, p_4. At query time, the user u first retrieves the grid granularity, then computes the grid cell that encloses her location. The contents of the grid cell are retrieved using KO, and it is guaranteed that the user will receive her exact NN (and possibly several other points whose Voronoi cells intersect the grid cell of the user).

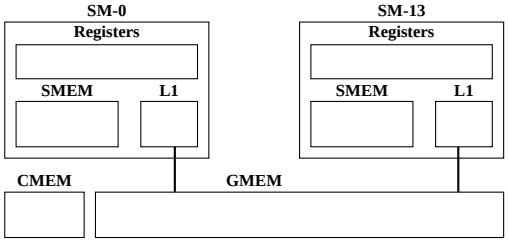


Figure 2: Main CUDA memory hierarchy modules

CUDA Architecture. *Compute Unified Device Architecture (CUDA)*[1] is a parallel computing platform and programming model designed by NVIDIA [7]. A CUDA program consists of a set of

[1]http://www.nvidia.com/object/cuda_home_new.html

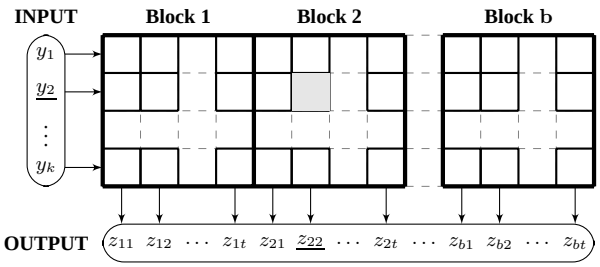


Figure 3: Allocation of work to CUDA threads and blocks

kernels, i.e., functions that run on the GPU and contain instructions from CUDA's own instruction set architecture (PTX ISA). Kernels are executing across a set of parallel *threads* which are organized into *blocks*. Each thread has its own ID (unique inside the block), program counter, register file and private memory. Threads within a block can cooperate through barrier synchronization and have access to a per-block *shared (SMEM)* memory area. Blocks are organized inside a *grid*, i.e., an array of blocks that execute the same kernel. All threads in the grid have access to the GPU's *global (GMEM)* and *constant (CMEM)* memory regions. The GPU has a set of *streaming multiprocessors (SM)*, and each SM controls a number of CUDA *cores*. A SM executes one or more thread blocks, and each core executes one instruction per clock for a thread. The SM scheduling unit is a *warp*, which is a group of 32 threads belonging to the same block.

While programmers can generally ignore warp execution and write applications according to the hierarchy of grids, blocks and threads, the warp concept is very important from a performance standpoint. An important performance metric is *occupancy*, which is defined as the ratio between the maximum number of warps which can be scheduled simultaneously during kernel execution divided by the maximum number of warps that the device supports. Occupancy is influenced by the number of registers a thread uses, the amount of shared memory used per block, and the *block size*, i.e., the count of threads in a block.

Figure 2 shows the main components of the CUDA memory hierarchy and their visibility across SMs. The register file on the SM stores the thread registers for the current warp. The rest of the memory space on the SM is split between the SMEM and a L1 cache that buffers accesses to GMEM. The programmer has some flexibility in choosing the split allocation between SMEM and L1. There are three key aspects that dictate the performance of memory accesses. First, to complete a round of instructions with a single bus transfer, accesses to GMEM must be *coalesced*, i.e., threads in the same warp must access consecutive addresses, and the first thread in the warp must access an address aligned at a multiple of 16 bytes. Second, the SMEM is divided into 32 banks, and to ensure a single bus transfer per instruction round, all threads in a warp must access distinct SMEM banks. And third, if threads use a larger number of registers than physically available on the SM, *spilling* occurs, which goes all the way to the GMEM. This is significant, as latency of the GMEM is considerably higher than that of SMEM or the registers, so the amount of spills should be minimized.

3. DEPLOYING KO PIR ON CUDA

In deploying PIR on CUDA, we need to consider two aspects: how to map PIR execution to the grid-block-thread hierarchy, and how to represent and operate on large integers (CUDA does not directly support large-integer multiplications).

We use a grid representation of the problem, whereby each thread receives a sequence of numbers and returns as result their product. Figure 3 illustrates this approach, assuming there are b

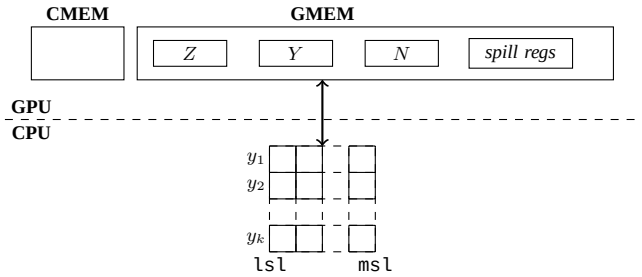


Figure 4: Query Array Representation in GPU PIR Approach

blocks of t threads each. For a database of n items, each thread receives $k = \frac{n}{b \times t}$ large integers denoted by $y_1, \ldots y_k$ as input (this is the query array $Y = [y_1 \ldots y_S]$), and computes as result their masked product z, one element of the result array Z. In the diagram, the user asks for the value of the shaded element in the database, mapped to the second thread of the second block. The user sets y_2 to be QNR, and all other y values to be QR. At the end of the protocol, the user determines the value of the requested bit based on whether z_{22} has the QNR property or not.

To represent large integers, we split each number into a set of *limbs*, i.e., consecutive blocks of fixed bit length. For instance, if each limb is a 4-bit string, it represents a digit in base 16 ($\beta = 16$). A number encoded as (3927) in this representation (first digit is most significant limb *msl*, last one is the least significant limb *lsl*) is equal to $3\times(16)^3+9\times(16)^2+2\times(16)^1+7\times(16)^0 = 14631_{10}$. In our implementation, since we have native 32-bits integers, each limb has 32 bits integer, i.e., $\beta = 2^{32}$.

The pseudocode in Figure 5 summarizes the approach. The CUDA kernel executes a set of Montgomery multiplications [6] for each thread. The query elements y_i are positioned consecutively in the CPU memory (i.e., a matrix of limbs), as illustrated in Figure 4, and they are transferred in the same configuration to the GPU's GMEM. Since the number arrays are large, they are transferred between the GMEM and registers upon access by the threads.

GPU_PIR

Input: Query Array Y, Modulus N
Output: Query Result Array Z
1. $sendToGpu(Y)$;
2. $sendToGpu(N)$;
3. Z = $LaunchKernel()$;

Figure 5: GPU PIR Approach Pseudocode

Prepare_For_Coalesced_Read

Input: Query Array Y, work per thread k (number of database matrix columns)
Output: Padded Array pY
Local Variables: pk: minimum multiple of 16bytes larger than k
1. $pk = pad(k)$;
2. $pY = newarray())$;
3. **for** $i = 0 : k - 1$ **do**
4. **for** $j = 0$ **to** $N - 1$;
5. $pY[pk * j + i] = Y[i][j]$;

Figure 6: Limb Interleaving Pseudocode

The next step is to ensure coalesced accesses to GMEM. Recall from Section 2 that non-coalesced accesses lead to reduced bandwidth transfers between the GMEM and the registers. To prevent this problem, we introduce a pre-processing step whereby the CPU interleaves the limbs from all Y items from the lsl to the msl. Furthermore, a small padding is introduced to ensure that the next sequence of limbs starts at a memory address that is a multiple of 16 bytes. This layout ensures that all reads are coalesced, hence all

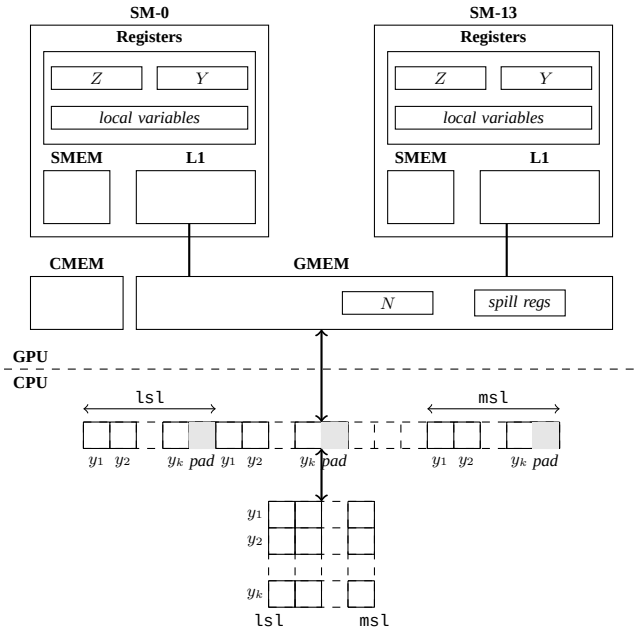


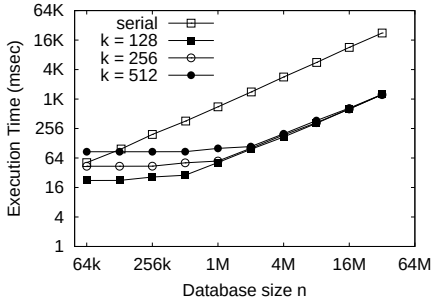
Figure 7: Limb Interleaving for Coalesced GMEM Reads

threads can read simultaneously in a single transfer the j^{th} limb in their respective y_i elements.

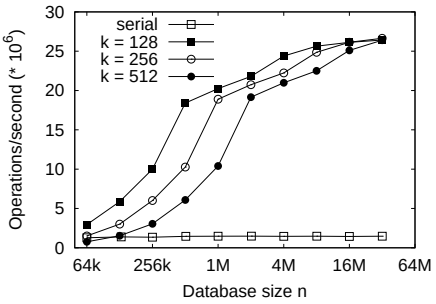
The pseudocode of this approach is illustrated in Figure 6. The query array Y, represented as a two-dimensional matrix in the CPU memory space, is first transformed into a linear array (as shown in Figure 7), before transferring it to the GPU. For example, in the case of 4-bit limbs ($\beta = 16$ and native integers of 4 bits) and two integers represented as (3927) (i.e., 14631_{10}) and (4983) (i.e., 18819_{10}), the CPU first creates a new vector aligned at a memory address multiple of 16 which contains in sequence the lsl values 3 and 7, followed by a padding of 30 4-bit integers to reach the 16 bytes alignment, then the values 2 and 8 followed by another padding, and so forth for the remaining limbs. The resulting vector is passed to the GPU and stored in GMEM.

Coalescing accesses to GMEM brings significant improvements, but a large number of registers are required to store all limbs. This high register pressure results in a large number of spills to GMEM, which has relatively high latency compared to registers, L1 and SMEM (the later three reside on the SM). We minimize the amount of L1, and designate most of the split L1/SMEM 64KB space as SMEM. Specifically, 48KB are designated as SMEM, with the remaining 16KB for L1. Placing in the SMEM region the limbs of the large integers to be multiplied gives us the opportunity to control explicitly the placement and eviction of limbs in memory, and to optimize memory latency for the Montgomery algorithm. We emphasize that there is no actual need for sharing data across threads. The problem remains an embarrassingly parallel one, and there is no additional overhead for thread synchronization. The only reason we resort to SMEM is to gain direct control to a faster memory region where we can store our data, without being subjected to the L1 eviction.

To further reduce the SMEM requirement per thread, we adopt an optimization whereby the accumulator is split across SMEM and registers. The *least significant portion (lsp)* of the accumulator is stored in SMEM, whereas the *most significant portion (msp)* is stored in thread memory (via registers and L1 cache), and thus subject to GMEM spills.

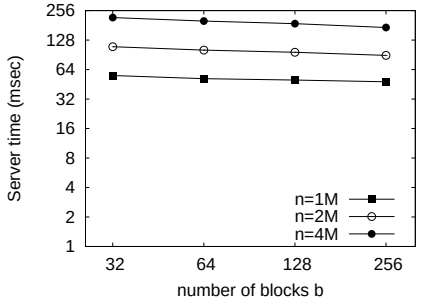


(a) Time

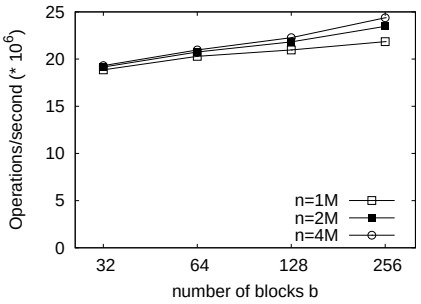


(b) Operations per second

Figure 8: Impact of database size



(a) Time



(b) Operations per second

Figure 9: Impact of number of blocks

4. EXPERIMENTAL EVALUATION

We evaluate experimentally the performance of the proposed CUDA-based NN PIR approach. Our experimental testbed consists of a Tesla K20Xm card with a GK110 GPU which supports Compute Capability 3.5. Our board is configured with 14 SMs, each having 192 (for a total of 2688 cores). The board has 6TB of global memory available (in total for both kernel code and kernel data) and 64KB of constant memory space. The GPU clock rate is at 0.73GHz while the memory clock ticks at a rate of 2600MHz.

We use the Sequoia dataset with $65,000$ points of interest in California. We vary the database size n by using the entire Sequoia dataset and using several settings of grid granularity. The resulting input size for the KO PIR protocol ranges from 64 thousand to 32 million items. We use encryption strength of 1024-bits.

Figure 8(a) shows query execution time as a function of database size n. We consider several settings of k (i.e., number of multiplications per thread). Higher values of k correspond to longer execution times. However, for fixed n, the number of threads launched is $b \times t = \frac{n}{k}$. Since we maintain a number of $t = 128$ blocks per thread, after a certain number of blocks ($b = \frac{n}{128k}$) we will reach the computational limits of the device, i.e., there are no more CUDA cores available (the test device has $2,688$ cores). After this limit is reached, the execution time increases linearly with database size, similar to the serial implementation. From Figure 8(a), we can observe that doubling k also doubles the database size threshold at which the linear increase occurs. Furthermore, doubling k doubles execution times in the non-saturated region.

This trend can also be observed in Figure 8(b), which shows the number of operations per second as a function of the database size n. In the non-saturated area we have a steep increase in terms of operations per second. However, as soon as all CUDA cores are in use, all values of k will yield the same number of operations.

Next, we vary the number of blocks b. Figure 9(a) illustrates execution time for typical b settings and three distinct database sizes: $n = 1M, 2M$ and $4M$. Due to the limit of at most 1024 threads per block we need to have $k \leq 1024$, which translates to $b \geq 32$ (limit reached for $n = 4M$). As b increases the execution time decreases, because k is inversely proportional to b, thus doubling b halves k which yields shorter execution times per thread. Figure 9(b) illustrates the same effect on the operation throughput measurement.

5. CONCLUSION

We proposed a GPU-based solution to accelerate the execution of private spatial information retrieval. Our results show that a careful design customized for the PIR problem can yield performance improvements of more than one order of magnitude. Although we focused on nearest-neighbor queries, our solution can be extended to other spatial problems that are amenable to a query-by-content to query-by-index conversion. To the best of our knowledge, this is the first work to consider acceleration of private spatial information retrieval using GPUs, and we believe these results are an important first step to adopt CUDA for related problems. In the future, we plan to extend our results to other types of queries, e.g., private spatial skyline queries.

Acknowledgment: The work reported in this paper has been partially supported by the NSF grant CNS-1111512.

6. REFERENCES

[1] G. Ghinita, P. Kalnis, M. Kantarcioglu, and E. Bertino. Approximate and exact hybrid algorithms for private nearest-neighbor queries with database protection. *Geoinformatica*, 15(4):699–726, 2011.

[2] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K.-L. Tan. Private queries in location based services: Anonymizers are not necessary. In *SIGMOD Conference Proceedings*, pages 121–132. ACM, June 2008.

[3] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. In *Proc. of USENIX MobiSys*, 2003.

[4] H. Kido, Y. Yanagisawa, and T. Satoh. An anonymous communication technique using dummies for location-based services. In *Proc. of the 2005 IEEE Intl. Conference on Pervasive Services*, pages 88–97, Santorini, Greece, 2005.

[5] E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: Single database, computationally-private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[6] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar. 2008.