High-speed Private Information Retrieval Computation on GPU

Carlos Aguilar Melchor Benoit Crespin Philippe Gaborit Vincent Jolivet
Pierre Rousseau

XLIM-DMI,
Université de Limoges,
123, av. Albert Thomas
87060 Limoges Cedex, France
IMPORTANT NOTE: Our implementation can be downloaded from
`http://www.assembla.com/spaces/pir_gpgpu2008`
E-mail: {carlos.aguilar, benoit.crespin, philippe.gaborit, vincent.jolivet, pierre.rousseau}@xlim.fr

Abstract

A Private Information Retrieval (PIR) scheme is a protocol in which a user retrieves a record out of n from a replicated database, while hiding from the database which record has been retrieved, as long as the different replicas do not collude.

A specially interesting sub-field of research, called single-database PIR, deals with the schemes that allow a user to retrieve privately an element of a non-replicated database. In these schemes, user privacy is related to the intractability of a mathematical problem, instead of being based on the assumption that different replicas exist and do not collude against their users.

Single-database PIR schemes have generated an enormous amount of research in the privacy protection field during the last two decades. However, many scientists believe that these are theoretical tools unusable in almost any situation. It is true that these schemes usually require the database to use a lot of computational power, but considering the large number of applications these protocols have, it is important to develop practical approaches that provide acceptable performances for as many applications as possible.

We present in this article a proof-of-concept implementation of a single-database PIR scheme proposed by Aguilar and Gaborit [2, 3]. This implementation can run in a CPU or in a GPU using CUDA, nVidia's library for General Purpose computing on Graphics Processing Units (GPGPU). The performance results highlight that linear algebra PIR schemes allow to process database contents several orders of magnitude faster than previous protocols.

1 Introduction

Usually, to retrieve an element from a database, a user will send a request pointing out which element he wants to obtain, and the database will send back the requested element. Which element a user is interested in may be an information he would like to keep secret, even from the database administrators. For example, the database may be :

- an electronic library, and which books we are interested in may provide information about our political or religious beliefs, or other details about our personality it may be desirable to keep confidential,
- stock exchange share prices, and the clients may be investors reluctant to divulge which shares they are interested in,
- a pharmaceutical database, and some client laboratories may wish that nobody may learn which are the active principles they may want to use,

To protect his privacy, a user accessing a database may therefore want to retrieve an element without revealing which element he is interested in. A trivial solution is for the user to download the entire database and retrieve locally the element he wants to obtain. This is usually unacceptable if the database is too large (for example, an electronic library), quickly obsolete (for example, stock exchange share prices), or confidential (for example, a pharmaceutical database).

978-0-7695-3329-2/08 $25.00 © 2008 IEEE
DOI 10.1109/SECURWARE.2008.55

Private Information Retrieval (PIR for short) schemes aim to provide the same confidentiality to the user (with regards to the requested element) than downloading the entire database, with sub-linear communication cost. PIR was introduced by Chor, Goldreich, Kushilevitz, and Sudan in 1995 [6]. In their paper, they proposed a set of schemes to implement PIR through replicated databases, which provide users with information-theoretic security as long as some of the database replicas do not collude against the users. Remark that PIR schemes do not ensure database confidentiality: a user may retrieve more than a single database element with a PIR scheme without the database learning it. A PIR scheme ensuring that users retrieve a single database element with each query is called a Symmetric PIR (or SPIR) scheme.

In this paper, we will focus on PIR schemes that do not need the database to be replicated, which are usually called single-database PIR schemes. Users' privacy in these schemes is ensured only against computationally-bounded attackers. It is in fact proved that there exists no information-theoretically secure single-database PIR scheme with sub-linear communication cost [6]. The first single-database PIR scheme was presented in 1997 by Kushilevitz and Ostrovsky [14], and since then improved schemes have been proposed by different authors [24, 4, 5, 15, 10]. To lighten our text, PIR will implicitly represent single-database Private Information Retrieval. When dealing with schemes that need the database to be replicated we will explicitly say replicated-database PIR.

A major issue with PIR schemes is that they are computationally expensive. Indeed, in order to answer a query, the database must process all of its entries. If in a given protocol it does not process some entries, the database will learn that the user is not interested in them. This would reveal to the database partial information on which entry the user is interested in, and therefore it is not as private as downloading the whole database and retrieving locally the desired entry.

The computational cost for a server replying to a PIR query is therefore linear on the database size. Moreover, current schemes have a very expensive cost per bit in the database, a multiplication over a large modulus. This limits both the database size and the throughput shared by the users, limiting as well their usage for many databases as for other applications such as low-latency unobservable communications [1] or private keyword search [18].

Recent proposals [9, 2] introduce noise-based protocols in which the cost per bit in the database is a vector addition which, depending on the parameters, can be much cheaper than the modular multiplications used in number theory schemes.

We present in this paper an implementation of one of these noise-based protocols. This protocol was proposed by Aguilar and Gaborit in a preliminary version in the 2007 Western European Workshop on Research in Cryptology (WEWoRC'07) [2]. A more refined version, including a security proof sketch, has been presented by the same authors at the 2008 IEEE International Symposium of Information Theory [3].

This scheme has two major drawbacks and a minor one. The first major drawback is that query size is really huge compared to the other protocols. The second major drawback is that it is based on new security assumptions and thus its security may be easier to break. The last drawback is that the database reply expansion factor is larger than with other schemes, but considering the obtainable throughputs with existing PIR schemes, having a small or a large expansion factor is not such a major issue.

On the other side, the fact that this scheme uses linear algebra has allowed us to implement a really fast PIR scheme over Graphics Processing Units that can be used for many applications unreachable before. The performance results are not tailored to Aguilar and Gaborit's scheme and would be similar with other linear algebra schemes (if the matrices' dimensions do not vary radically). The results highlight that linear algebra can lead to high-performance PIR schemes despite the communication cost overhead.

The paper is organized as follows. Section 2 describes basic concepts on General Purpose computing on Graphics Processing Units and on PIR schemes. Section 3 introduces Aguilar and Gaborit's scheme and our implementation. In Section 4 we provide some comparative performance results and finally, in Section 5 we conclude.

2 Basic concepts

2.1 GPGPU

The concept of General-Purpose computation using Graphics Processing Units (or GPGPU) has arosen in recent years. GPGPU applications make use of graphics processing units (GPUs) as massively parallel processors, turned away from their original purpose but nonetheless highly efficient in numerous application domains. Combining both speed and bandwidth due to their parallel architecture (and accessible cost), GPUs are an attractive choice for complex computations compared to traditional CPUs since they exhibit significant performance overheads, and are supported by constantly improving high-level programming language provided by both GPU vendors and the academic community. GPUs have now evolved from highly-specialized graphics processors into powerful, flexible programmable units, and become increasingly popular for a wide range of applications [11, 19].

2.1.1 Programming on GPU

To implement a GPGPU application, one has to choose amongst different kinds of Application Programming Interfaces (API). *General* API, namely OpenGL and DirectX, give access to textures and memory transfers between CPU and GPU, and control of the rendering process.

Until 2001, all the operations required to display an object on screen used dedicated hardware through a fixed pipeline. 2001 saw the advent of vertex shaders and fragment shaders, written using *shader languages* such as Cg, HLSL and GLSL. This allowed programmers to grasp more control over the rendering process. However, these shaders suffered from several limitations at first: limited instruction count, no support for integer types or branching, etc. Throughout the years, these limitations were pushed away to a state where they no longer restrict the programmers.

Geometry Shaders were introduced in 2007 with Shader Model 4.0 to complement the possibilities offered by vertex shaders and fragment shaders. With so much freedom given to the developpers, GPU vendors had to rethink their architecture. Some applications needed much computing power for vertex processing and little for fragment processing, while other applications needed the opposite. This led to a sub-optimal use of the GPU, where fragment processors could be left idle waiting for vertex processors to output data (or vice-versa).

To avoid this, GPU architecture is now based on *general-purpose stream processors* capable of being used alternately as vertex, geometry of fragment processors, and recent specialized API are designed using these general-purpose processors as the basic building block. General-purpose stream processors tend to become a natural choice for developping GPGPU applications since they are not limited by a programming model based on the classic graphics pipeline. These API include NVIDIA's CUDA, ATI's CTM and Microsoft's Accelerator.

As the time of writing, CUDA ("Compute Unified Device Architecture") seems to exhibit promising results [16, 20]. This technology allows developpers to program NVIDIA's Geforce 8xxx graphics cards as SIMD machines, using C language (to which some extensions are added). It allows direct memory access, scatter and gather operations, and arbitrary computations with no need for a graphics display. With CUDA, GPGPU becomes native, where it was previously a matter of hacking the hardware (rendering arbitrary data to textures, performing scatter through vertex shaders and gather through fragment shaders, requiring a graphics display).

2.1.2 nVidia CUDA GPU Architecture

A strong knowledge of the underlying architecture is required for efficient CUDA programming. Applications

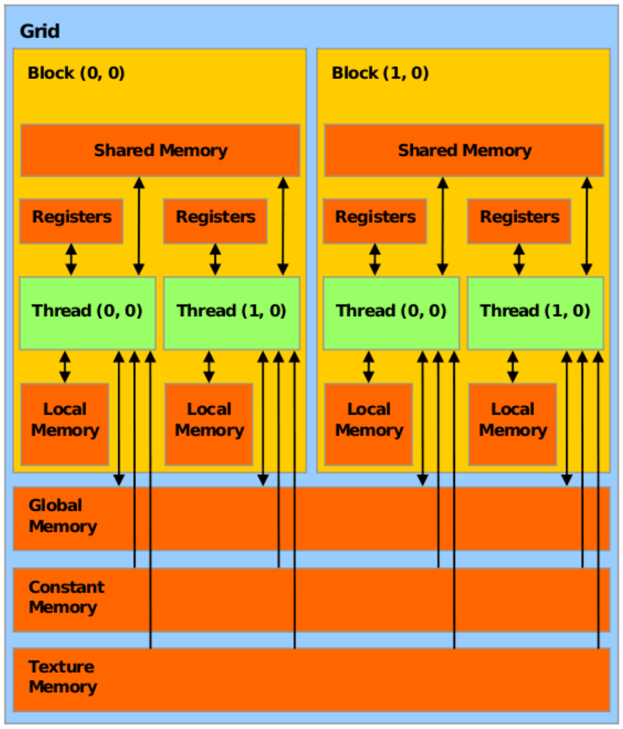

Figure 1. CUDA memory model, from [7]

which can be parallelized into thousands of threads, each thread performing a few simple computations on native data types are best suited for high performance porting to CUDA. To take advantage of the parallel nature of the device, computation must be divided into *blocks*, themselves being subdivided into *threads*. Each *thread* executes a *kernel*, which outputs a portion of the data to be computed.

A CUDA-capable graphics card contains several types of memory, each one serving a different purpose (see Fig. 1). The *global* memory has a high storage capacity (typically, more than 256MB), but each memory access needs at least 400 clock cycles. This latency can be hidden if threads have computation to do while the data is getting retrieved. Accesses to global memory by multiple threads within a block may be grouped into a single access (said to be *coalesced*) if the threads access contiguous memory blocks, with pointer addresses aligned to multiples of 32 bytes. Threads also have access to a fast on-chip *shared* memory which helps limiting accesses to global memory, but is restricted to 16KB. This memory is shared between the threads belonging to a given block. A typical usage is to first retrieve all the data which is to be processed by threads in block from global memory to shared memory with one coalesced access, process it and write the result back to global memory. *Constant* and *texture* memory are

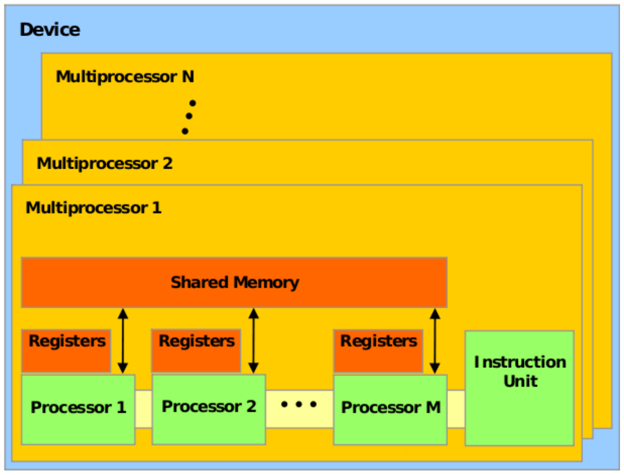

Figure 2. CUDA hardware model, from [7]

read-only portions of the global memory benefitting from a cache, which are to be used if shared memory isn't sufficient and a high level of caching can be achieved. *Local* memory is also a portion of global memory, dedicated to each thread, and meant to replace registers when a thread needs too many of them.

At runtime, threads are batched in *warps* of 32 threads, and sent to one of the GPU's *multiprocessors* for execution (see Fig. 2). Low-end GPUs (such as the Geforce 8400) host 2 multiprocessors, while high-end GPUs (such as the Geforce 8800) have 16 multiprocessors. Each multiprocessor contains 8 stream processors, treating threads in an *SIMD (Simple Instruction Multiple Data)* fashion. To perform their computation, threads have access to 8192 32-bits registers per multiprocessor. The number of registers required by each thread for its execution helps to define the *occupancy* ratio, which is the number of active warps per multiprocessor over the maximum number of warps. Programmers hence must be careful to maximize this ratio, by limiting the number of registers required by a thread, and wisely choosing the number of threads per block.

2.2 Private Information Retrieval

In a PIR protocol, a user wanting to retrieve an element of index i from a database, uses a PIR query generation algorithm with input i and sends the resulting query to the database. The database combines its elements to the query using a reply generation algorithm and obtains a result which is sent back to the user. Finally, the user decodes the answer through a reply decoding algorithm. The protocol is said to be correct if the decoding results in the i-th element of the database, and private if the database is unable to learn anything about i from the query or the reply generated.

To be more precise, we provide a small formal definition of a PIR protocol, as well as of its correctness and privacy. The reader can directly jump to the following section if not acquainted with formal definitions.

Definition 1 *Let DB be a database with n elements $\{e_1, ..., e_n\}$. A Single-Database Computationally-Private Information Retrieval Protocol is a polynomial-time probabilistic Turing machine Γ_n, that for an input k outputs the description of a polynomial-time randomized algorithm $Q(\cdot)$ (the query generator), and two polynomial-time deterministic algorithms, $R(\cdot,\cdot)$ (the reply generator), and $X(\cdot,\cdot)$ (the reply decoder), with the two following properties:*

(Correctness) $X(R(Q(i), \{e_1, ..., e_n\}), \mathcal{S})$, $\mathcal{S}$ being any secret information needed for reply decoding, outputs the i-th element of DB.

(Privacy) For any $c > 0$, and any $i_1, i_2 \in \{1, ..., n\}$, there is a K such that for $k > K$ and any polynomial-size probabilistic circuit family $C = \{C_k(\cdot,\cdot)\}$,

$$Pr(C_k(\alpha, i_1, i_2, \mathcal{P}(k)) = i | Q \leftarrow \Gamma_n[k]; \alpha \leftarrow Q(i)) < 1/2 + k^{-c}$$

i being randomly chosen among $\{i_1, i_2\}$, and $\mathcal{P}(k)$ being the set of all the public information about the PIR protocol.

2.2.1 Recursive usage

PIR requests are usually formed of a set of n query elements, one per each database element. Each of these query elements is combined with the database element it is associated with, and then the results are combined between them to obtain the PIR reply (see Figure 3). This reply is usually a constant factor F_{PIR} times larger than a database element. We call this constant factor the database reply expansion factor.

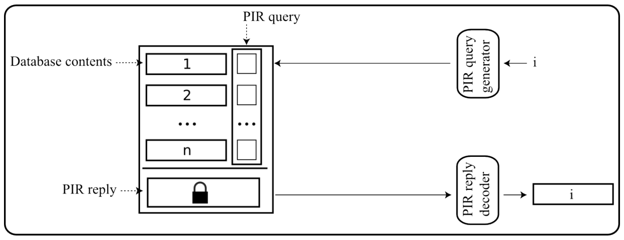

Figure 3. PIR retrievals.

If the user sends n query elements and the database a PIR reply, the total communication cost is $O(n)$. To reduce this cost, it is possible to use the scheme recursively as Kushilevitz and Ostrovsky proposed in [14]. The idea is to

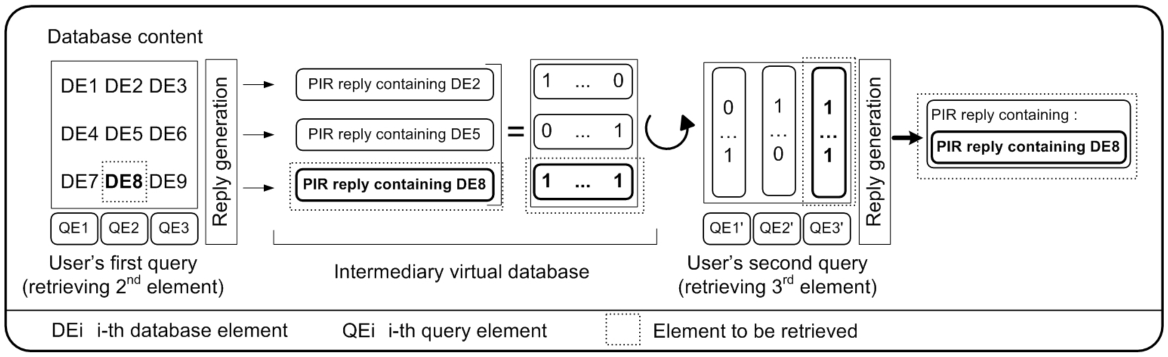

Figure 4. Recursive usage of a PIR scheme.

represent the n-element database in two dimensions, with $\sqrt{n}$ lines and $\sqrt{n}$ columns. The user sends $\sqrt{n}$ query elements, one for every column in the matrix, and the database generates iteratively $\sqrt{n}$ PIR replies, one for each line in the matrix. These replies are not sent to the user, but will form an intermediary *virtual* database (of elements F_{PIR} times larger than the original database elements). The user sends a second query of $\sqrt{n}$ elements to retrieve the PIR reply he is interested in.

In figure 4, the first recursion results in three PIR replies that are used as a virtual database for the second recursion. When recursion is used, the query size shrinks (from $O(n)$ to $O(n^{1/2})$), and the database reply expansion factor increases (from F_{PIR} to F_{PIR}^2).

If the database is represented as a cube of size $n^{1/3}$, the user will send three queries with $n^{1/3}$ elements each. The database will compute a matrix of $n^{1/3} \times n^{1/3}$ PIR replies from the first query. This matrix can be seen as a virtual database and the second query will be used to retrieve one column of $n^{1/3}$ PIR replies. This column will also be used as a virtual database of $n^{1/3}$ elements, and the third query will be used to obtain the PIR reply containing the element the user is interested in. Generally, if the database is represented by a d-dimension hyper-cube, d recursions are possible. With such a representation the user will send d requests, each composed of only $n^{1/d}$ residues. This allows a user to shrink greatly the queries' size, but the size of the PIR replies increases quickly (exponentially in most cases) in d. A trade-off must be made, depending on the application the PIR scheme is used for.

3 Implementation

3.1 Basic description of the scheme

In this section we describe an overview of the PIR scheme. Query generation and information extraction from the database reply are not described for two reasons. First, the bottleneck in PIR schemes is reply generation, not query generation or reply extraction. Second, all our implementation efforts and the GPGPU computation are on the reply generation phase. Query generation and reply extraction have been implemented straightforwardly from the scheme description. The formal description of query generation and information extraction phases is available on the appendix.

We describe a database as a set of n elements. Each element a_i (for $i \in \{1, \cdots, n\}$) is split in ℓ_0-bit sub-elements and represented as a matrix A_i

$$A_i = \begin{bmatrix} a_{i,1,1} & \cdots & a_{i,1,N} \\ \vdots & \vdots & \vdots \\ a_{i,L,1} & \cdots & a_{i,L,N} \end{bmatrix}$$

N and ℓ_0 being two security parameters (which in our implementation we set respectively to 48 and 16). Noting S_{max} the size of the largest element in the database, parameter L is set to $L := \lceil S_{max}/(N \times \ell_0) \rceil$. If a database element is smaller than $L \times N \times \ell_0$ the end of the matrix is filled up with a standard padding technique.

A user wanting to retrieve an element generates a query formed of n matrices $(B_1, \cdots, B_n)$, one for each database element. Each matrix is of dimension $N \times 2N$ with scalars in $\mathbb{Z}/p\mathbb{Z}$, p being a $3\ell_0$-bit prime. If the user wants to retrieve element a_{i_0}, he generates a query such that B_{i_0} has a special property that is invisible to the server. This property ensures that the user will be able to extract from the server reply the desired element.

The user sends the query to the server hosting the database. To generate the reply, the server multiplies the column concatenation of the database element matrices and the row concatenation of the query matrices, as shown in Figure 5.

The resulting matrix, which we note R, is a $L \times 2N$ matrix with $3\ell_0$-bit scalars (in $\mathbb{Z}/p\mathbb{Z}$) and thus, is six times larger than a database element matrix. Query size is $n \times N \times 2N \times 3\ell_0$, which for the given parameters results in $28 \times n$

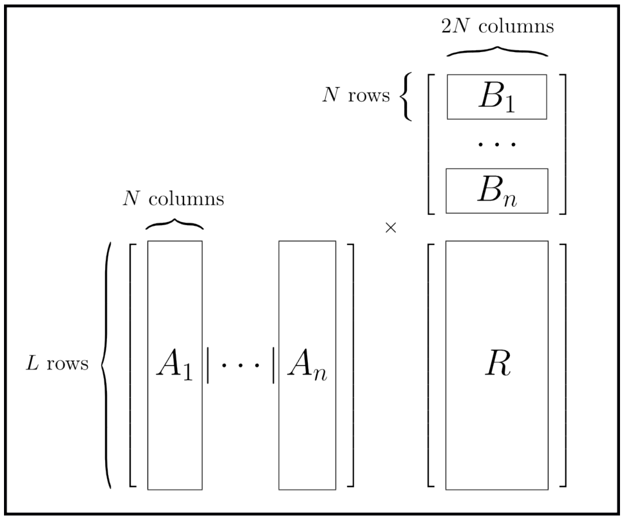

Figure 5. Scheme overview.

Kbytes. As noted in Section 2.2, it is possible to use the scheme recursively to lower the query size to $28 \times d \times n^{1/d}$ Kbytes (for d levels of recursion). However recursion must be used carefully as it results in a larger reply expansion factor, which becomes 6^d. Recursion usage does not change much PIR computational performance and thus we have not included it in the current implementation.

3.2 Reply generation using GPGPU

All the existing PIR schemes are highly parallelizable. However, in the classic schemes, the basic operation for reply generation (per bit on the database) is a multiplication over a 1024 or 2048 bit modulus. Stream processors are not adapted to do directly such operations and trying to do a straightforward parallelization would result in very poor performance. The correct approach is to use the whole set of stream processors to split an exponentiation among them. However, even if this approach has been proven to run 100 times faster than CPU computation for a 190 bit modulus (see [8]), it only runs 2-3 times faster for 1024 bits modulus (see [17]).

The situation is very different for the protocol we have decided to implement. Indeed, the basic operation (per group of ℓ_0 bits in the database) is a multiplication of an $\ell_0 = 16$-bit scalar by a $3 \times \ell_0 = 48$-bit scalar, which can be efficiently encoded through SIMD (Simple Instruction Multiple Data) instructions, on which GPUs are specialized. Each of these operations can be executed inside a single stream processor and the large number of these processors in modern GPUs results in a significative performance improvement.

3.3 Package description

Our implementation (on the client-side) uses NTL [22], the Number Theory Library of Victor Shoup, and is distributed with it.[1] It is possible to compile the server and client with or without CUDA in order to test GPU and CPU or CPU-only performance. All the instructions for installation are given with the package.

We have implemented a basic server that is compiled in the `server` directory during the installation process. The command `PIRServer configfile [port]` runs the server on port `port`. The file `configfile` must contain a set of files (for example a few songs) that the server will use to form the database element matrices $A_1, \cdots, A_n$. Once these matrices are set up, the server waits for a client connexion.

The client is located in the `client` directory and is run with `PIRClient [port]`. When run, the client first asks for the server IP (the default being `127.0.0.1` for an easy test). Then, it connects to the server, and receives a list of the filenames available. The user chooses an index for a file and the client generates a query for this element using NTL. After the query is generated (following the query generation algorithm described in Appendix A), it is sent to the server.

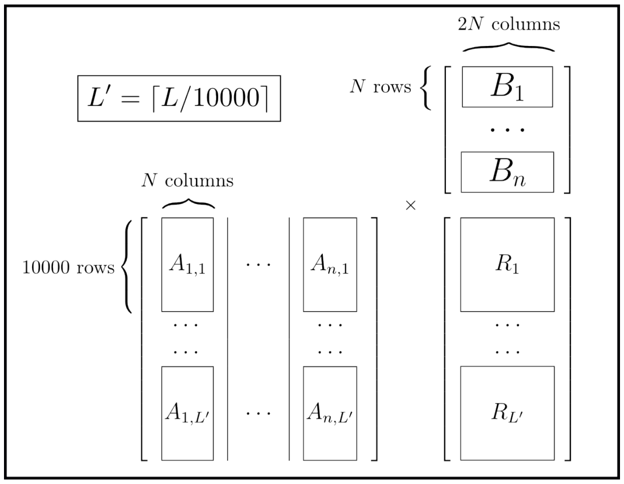

Figure 6. Block matrix multiplication.

Upon reception of the query, the server begins the reply computation. In the current implementation the server only handles one client at a time. In order to optimize cache usage, we have chosen to use a block matrix multiplication algorithm.

The matrices representing database elements are split in matrix blocks of 10000 rows (if they have less than 10000

[1]The `./install` script provided with the package proposes the user to build and install it if needed.

rows they are just not split) and the server asks iteratively the GPU to compute $A_{i,1} \times B_i$ and sum the results progressively (see Figure 6). After n iterations the server obtains R_1 and sends it to the user while restarting the process for each of the following block lines until sending $R_{L'}$.

Each iteration of the block matrix multiplication uses as many threads as possible to maximize the *occupancy* of the hardware (see section 2). In practice we generate a block of $2N \times 2$ threads which yields an excellent occupancy. Each of these blocks computes two lines of the result matrix. Five thousand such blocks of threads are executed on the GPU to ensure all the stream processors are used.

All the elements from a given row of $A_{i,j}$ participate in the computation of each element of the corresponding row of R_i. This makes $A_{i,j}$ a perfect candidate for shared memory storage. However, each element of $A_{i,j}$ is stored in a 16-bits integer, which can't natively yield coalesced accesses. We thus had to cast the pointer to $A_{i,j}$ to a 32-bits integer type (each integer containing two elements), which allows coalesced accesses. The NVIDIA® test tools show our occupancy is very high, and that all our memory accesses are coalesced, which means that hardware usage is optimal in our implementation.

4 Computational performance

From a computational point of view, reply generation is the limiting factor for single-database PIR. We will therefore not analyze the computational cost for a user to generate a query and to decode a PIR reply.

In this section, we present the performance results of the GPU and the CPU approach for Aguilar and Gaborit's scheme. We also provide a comparison with two number-theory based schemes. Lipmaa's scheme [15] is the outcome of a set of protocols [14, 24, 5] based on homomorphic encryption. Similarly, Gentry and Ramzan's scheme [10] is the outcome of the second known approach to obtain number-theory PIR schemes. Note that other linear algebra based schemes than Aguilar and Gaborit's exist. However, their security has either been broken [13] or require parameters resulting in unrealistic communication costs [9], and thus they have not been included in our performance comparison.

Currently two PIR implementations are known to the authors. The first is a single-database PIR implementation [21] that it is not at all focused on performance optimization and thus, no comparison is done with it. The second is a replicated-database PIR implementation presented at IEEE Security & Privacy 2007 [12]. Computational issues in replicated-database PIR schemes are very different from the ones in single-database schemes. Indeed, even the seminal replicated-database PIR schemes presented by Chor et al. in [6] were optimal from a computational cost point of view (one bit operation per bit on the database). Computational complexity derives from other issues like trying to add robustness properties to the distributed protocol, which has nothing to do with our work. We therefore not compare our single-database PIR results with the implementation proposed in [12] for robust replicated-database PIR schemes.

4.1 Experimental setup

We have not implemented Lipmaa's and Gentry and Ramzan's schemes. On the other hand, a pretty good performance estimation can be done. Indeed, in these schemes, the server's computational cost is just a large set of standard repetitive operations (a 1024 or 2048 bit modular multiplication per bit in the database). It is therefore very easy to make an analytical approximation of this cost. Moreover, standard utilities, such as the openssl library, provide implementations of these operations optimized for many years by the research community. These utilities allow to transform out analytical results into practical figures with pretty good confidence that the result will be very close to the one of a real implementation. We have set all the parameters in these protocols to optimize computational performance. The results we present for these schemes may decrease if these parameters vary.

We have run the tests over six different systems: three systems for the CPU tests and three for the GPU tests. The three processors used for CPU tests correspond to:

System 1:	
An Athlon™ 5000+	(2GHz, 1MB of cache)
System 2:	
A dual-core Xeon™ 5160	(3GHz, 4MB of cache)
System 3:	
A quad-core Xeon™ 5345	(2.33GHz, 4MB of cache)

All the GPU tests have been done on a machine with an Athlon™ 5000+, using the following graphics cards:

System 1:	
An ASUS® 8600 GTS	(32 SPs at 540MHz)
System 2:	
An MSI® 8800 Ultra	(128 SPs at 660MHz)
System 3:	
Two SLI enabled 8800 GTX	(2×128 SPs at 550 MHz)

SP = Stream Processor

The CPUs and GPUs used in the corresponding systems have similar prices (namely 150$, 500$ and 1100$).

4.2 Results

Table 1 presents the throughput at which the database was processed in our tests (mean value out of twenty trials). These values correspond to a database with twelve files of 3 Mbytes each. Mean throughput is an increasing function on file size and on the number of files. On the other hand the dispersion of the trials is a decreasing function on these parameters. In the GPU tests, optimal values are attained when the number of files reaches a few hundreds and file size a few megabytes. In the CPU tests, optimal values are attained when the number of files reaches a few dozens, and file size a few hundred kilobytes. We provide in Figure 7 this evolution for the GPU tests on System 1 as an example. Similar results are obtained for other GPU and CPU tests. The results of Table 1 for our implementation are thus a little bit (around 10%) under what is attainable for large databases.

Scheme	Processing throughput		
	System 1	System 2	System 3
Lipmaa[1]	160 Kbits/s	280 Kbits/s	490 Kbits/s
Gentry and Ramzan[1]	490 Kbits/s	890 Kbits/s	1500 Kbit/s
Aguilar and Gaborit (CPU)[2]	33 Mbits/s	130 Mbits/s	230 Mbits/s
Aguilar and Gaborit (GPU)[2]	270 Mbits/s	1.2 Gbits/s	2 Gbits/s

[1] Estimation using `openssl` to evaluate analytical complexity (see Section 4.1).
[2] Experimental results using our implementation over 12 files of 3 Mbytes each.

Table 1. Computational performance results.

With any of the presented PIR schemes, when processing an n-element database, only one n-th of the computation processes the element the user wants to get. Thus, in order to obtain the throughput at which a user gets the element (after decoding the reply data), the results in Table 1 must be divided by n. For example if the user retrieves a song from a 1000 file database, the reply decoding process will output the file at 270 Kbits/s for the GPU System 1.

There is a large leap between the performance of the linear algebra scheme and number theory schemes, namely between a factor 100 using the CPU implementation and a factor 1000 with the GPU implementation. Of course this result has an important impact in PIR usability. Indeed a server will be able to use such a PIR scheme over larger databases, handling more users, and providing more throughput.

This performance leap is critical. Indeed, in [23], Sion and Carbunar show that in a large variety of situations number theory schemes are much slower than the trivial solution to the PIR problem (i.e. downloading the whole database). The speedup of the trivial solution is a factor between 10 and 1000 (depending on whether the communications are done through a home connection, or a high-speed LAN).

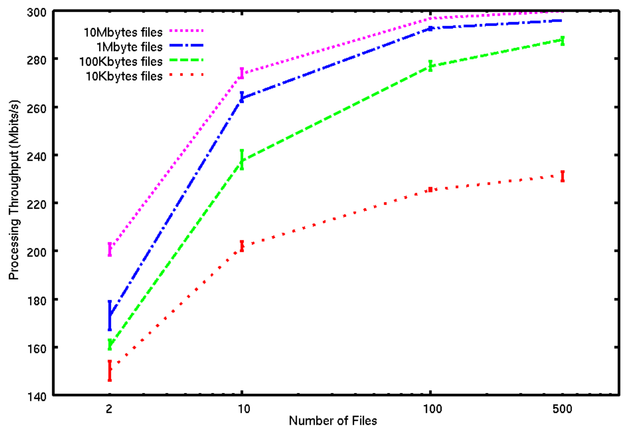

Figure 7. GeForce 8600 GTS GPU tests.

The performance improvement that is brought by linear algebra schemes inverses this factor. Thus, using a linear algebra scheme is the only way to be faster (and much more communication efficient) than the trivial solution in the situations described by Sion and Carbunar.

5 Conclusion

We have presented a proof-of-concept implementation of a linear algebra PIR scheme which can process over 1 Gbit/s of data with commodity graphics cards. This implementation allows also to process 230 Mbits/s of data with a high-end CPU.

Using the `openssl` library to test the performance attainable with existing schemes we have proved that the linear algebra scheme is between 100 (for the CPU implementation) and 1000 (for the GPU implementation) times faster than number theory schemes. We have noted the importance of this improvement, specially considering the previous work on PIR usability by Sion and Carbunar.

As Aguilar and Gaborit note in [2], their scheme results in larger communication costs, but with current bandwidths and through the usage of recursion it remains usable in many situations. On the other side, their security assumptions are pretty new and thus, the security of the system may be easier to break than for other schemes. In any case, the results we provide are true from a general point of view for linear algebra schemes. Indeed, these schemes can be really fast, specially if implemented over a GPU, and other schemes based on linear algebra should be carefully considered (as for example the proposal of Gasarch and Yerukhimovich [9]).

As a consequence of the good performance results we would like to polish and complete our implementation to transform it in a robust usable library for other users. We would also like to use other hardware improvements that GPUs can provide but that we have been unable to test for the moment. Finally we would like to implement an interface to common database languages such a MySQL, etc.

We hope that the provided results will motivate further research in linear algebra PIR schemes, and more generally in the privacy domain.

A Formal description of the PIR scheme

A.1 Request generation

This algorithm takes as an input three parameters: a security parameter N that in practice we set to 48, the number of elements in the database n, and the index chosen by the user i_0. There are three internal parameter that we process differently than in the original scheme: ℓ_0 that originally was set as $\ell_0 = \lceil log(n \times N) \rceil + 1$, q that was set as 2^{2l_0}; and p that was set as a prime larger than 2^{3l_0}. In our implementation we have fixed for the moment $\ell_0 = 16$, q as a random 30 bit integer and $p = 2^{47} + 5$ (which is a prime). These modifications result in an inherent limitation in n as the user may obtain a scrambled reply if $n > 85$. In practice this is not an issue, as recursion (see Section 2.2) is used when databases have many elements. Indeed, for a database with n elements, if d levels of recursion are used, d queries for $n^{1/d}$ element databases are sent. Thus, in practice queries are never done for more than a few dozen elements. Recursion will be implemented in our protocol in the short term. On the other hand, for databases with very few elements, handling a variable value ℓ_0 can result in a performance speedup and thus in the mid-term the original approach will also be implemented.

All the operations hereafter (except the first) are done over $\mathbb{Z}/p\mathbb{Z}$.

Request generation protocol

1. Set $\ell_0 = 16$, $q = 2^{30}$ and $p = 2^{47} + 5$.
2. Generate M_1 and M_2, two random matrices such that M_1 is invertible, and note $M = [M_1|M_2]$.
3. For each $i \in \{1 \cdots n\}$ compute a matrix $B'_i = [B'_{i,1}|B'_{i,2}]$ by multiplying M to the left by a random invertible matrix P_i.
4. Generate the random scrambling matrix Δ as a $N \times N$ random invertible matrix.
5. For each $i \in \{1 \cdots n\} \backslash i_0$ generate the soft noise matrix D_i, a $N \times N$ random matrix over $\{-1, 1\}$, and compute the soft disturbed matrix $B_i = [B'_{i,1}|B'_{i,2} + D_i]\Delta$.
6. Generate D_{i_0}, the hard noise matrix, by:
   - generating a soft noise matrix,
   - replacing each diagonal term by q.
7. Compute the hard disturbed matrix $B_{i_0} = [B'_{i_0,1}|B_{i_0,2} + D_{i_0}]\Delta$.
8. Send the ordered set $\{B_1 \cdots B_n\}$ to the database.

A.2 Information extraction

In order to simplify the protocol for information extraction we describe how we operate for each row vector of the reply. In practice some of these operations are aggregated and done over many row vectors at the same time. To extract the information from a row vector V_i of the database reply, the client operates in two phases. First it recovers the noise included in the vector (steps 1 and 2 of the protocol), and then he will unscramble and filter out this noise to obtain the information (steps 3 to 5). Note that steps 4 and 5 are *not* done over $\mathbb{Z}/p\mathbb{Z}$, as they correspond to an Extended Euclid's division over $\mathbb{Z}$.

Information extraction protocol

1. Unscramble the noisy vector $V_x^{'} = V_x \Delta^{-1}$
2. Retrieve $E = V_x^{'}(D) - V_x^{'}(U) M_1^{-1} M_2$, the inserted noise, $V_x^{'}(U)$ and $V_x^{'}(D)$ being resp. the undisturbed and disturbed halves of $V_x^{'}$ (i.e. the left half and the right half respectively).
3. For each $e_{x,y}$ in $E = [e_{x,1} \cdots e_{x,N}]$, if $e_{x,y} > p/2$ compute $e'_{x,y} = p - e_{x,y}$.
4. For each $e'_{x,y}$ compute $e''_{x,y} = e'_{x,y} - \epsilon$ with $\epsilon := e'_{x,y} \% q$ if $e'_{x,y} \% q < q/2$ and $\epsilon := e'_{x,y} \% q - q$ else.
5. For each $y \in \{1 \cdots n\}$, compute $a_{i_0,x,y} = e_{x,y}^{''} q^{-1}$.

We encourage the reader to read in the original paper [2] the full description of the protocol (which is given with a toy example) and the privacy and correctness proofs.

References

[1] C. Aguilar Melchor, Y. Deswarte, and J. Iguchi-Cartigny. Closed-Circuit Unobservable Voice Over IP. In *23rd Annual Computer Security Applications Conference (ACSAC'07), Miami, FL, USA*. IEEE Computer Society Press, 2007.

[2] C. Aguilar Melchor and P. Gaborit. A Lattice-Based Computationally-Efficient Private Information Retrieval Protocol. In *Western European Workshop on Research in Cryptology (WEWoRC'2007), Bochum, Germany. Book of Abstracts*, pages 50–54, Extended version available on IACR eprints `http://eprint.iacr.org/2007/446`, 2007.

[3] C. Aguilar Melchor and P. Gaborit. A Fast Private Information Retrieval Protocol. In *The 2008 IEEE International Symposium on Information Theory (ISIT'08), Toronto, Ontario, Canada*. To appear. IEEE Computer Society Press, 2008.

[4] C. Cachin, S. Micali, and M. Stadler. Computationally Private Information Retrieval with Polylogarithmic Communication. In *18th Annual Eurocrypt Conference (EUROCRYPT'99), Prague, Czech Republic*, volume 1592 of *Lecture Notes in Computer Science*, pages 402–414. Springer, 1999.

[5] Y.-C. Chang. Single Database Private Information Retrieval with Logarithmic Communication. In *Information Security and Privacy: 9th Australasian Conference (ACISP'04), Sydney, Australia*, volume 3108 of *Lecture Notes in Computer Science*, pages 50–61. Springer, 2004.

[6] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private Information Retrieval. In *46th IEEE Symposium on Foundations of Computer Science (FOCS'95), Pittsburgh, PA, USA*, pages 41–50. IEEE Computer Society Press, 1995.

[7] N. Corp. NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.1. http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf, 2007.

[8] S. Fleissner. GPU-accelerated montgomery exponentiation. In *7th International Conference on Computational Science (ICCS'07), Beijing, China*, volume 4487 of *Lecture Notes in Computer Science*, pages 213–220. Springer, 2007.

[9] W. Gasarch and A. Yerukhimovich. Computational inexpensive PIR, 2006. Draft available online at `http://www.cs.umd.edu/~arkady/pir/pirComp.pdf`.

[10] C. Gentry and Z. Ramzan. Single-Database Private Information Retrieval with Constant Communication Rate. In *32nd Annual International Colloquium on Automata, Languages and Programming (ICALP'05), Lisbon, Portugal*, volume 3580 of *Lecture Notes in Computer Science*, pages 803–815. Springer, 2005.

[11] M. Houston and N. Govindaraju, editors. *GPGPU: general-purpose computation on graphics hardware*. ACM Press, 2007.

[12] Ian Goldberg. Improving the Robustness of Private Information Retrieval. In *The 2007 IEEE Symposium on Security and Privacy (S&P'07), Oackland, CA, USA*, pages 131–148. IEEE Computer Society Press, 2007.

[13] A. Kiayias and M. Yung. Secure Games with Polynomial Expressions. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2001.

[14] E. Kushilevitz and R. Ostrovsky. Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval (extended abstract). In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–373, 1997.

[15] H. Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. In *8th Information Security Conference (ISC'05), Singapore*, volume 3650 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2005.

[16] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. Molecular dynamics simulations on commodity gpus with cuda. In *14th International Conference on High Performance Computing (HiPC'07), Goa, India*, volume 4873 of *Lecture Notes in Computer Science*, pages 185–196. Springer, 2007.

[17] A. Moss, D. Page, and N. Smart. Toward acceleration of rsa using 3d graphics hardware. In *Cryptography and Coding*, volume 4887 of *Lecute Notes in Computer Science*, pages 369–388. Springer, 2007.

[18] R. Ostrovsky and W. E. Skeith III. Private Searching on Streaming Data. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2005.

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

[20] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, and D. B. K. W. mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08), Salt Lake City, UT, USA, to appear*, 2008.

[21] F. Saint-Jean. A Java Implementation of a Single-Database Computationally Symmetric Private Information Retrieval (cSPIR) protocol. Technical Report 1333, Yale University, 2006.

[22] V. Shoup. NTL: A library for doing Number Theory. `http://www.shoup.net/ntl/`, 2007.

[23] R. Sion and B. Carbunar. On the Computational Practicality of Private Information Retrieval. In *14th ISOC Network and Distributed Systems Security Symposium (NDSS'07), San Diego, CA, USA*, 2007.

[24] J. P. Stern. A New Efficient All-Or-Nothing Disclosure of Secrets Protocol. In *13th Annual International Conference on the Theory and Application of Cryptology & Information Security (ASIACRYPT'98), Beijing, China*, volume 1514 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 1998.