Accelerating SWHE Based PIRs Using GPUs

Wei Dai(✉), Yarkın Doröz, and Berk Sunar

Worcester Polytechnic Institute, Worcester, USA
wdai@wpi.edu

Abstract. In this work we focus on tailoring and optimizing the computational Private Information Retrieval (cPIR) scheme proposed in WAHC 2014 for efficient execution on graphics processing units (GPUs). Exploiting the mass parallelism in GPUs is a commonly used approach in speeding up cPIRs. Our goal is to eliminate the efficiency bottleneck of the Doröz et al. construction which would allow us to take advantage of its excellent bandwidth performance. To this end, we develop custom code to support polynomial ring operations and extend them to realize the evaluation functions in an optimized manner on high end GPUs. Specifically, we develop optimized CUDA code to support large degree/large coefficient polynomial arithmetic operations such as modular multiplication/reduction, and modulus switching. Moreover, we choose same prime numbers for both the CRT domain representation of the polynomials and for the modulus switching implementation of the somewhat homomorphic encryption scheme. This allows us to combine two arithmetic domains, which reduces the number of domain conversions and permits us to perform faster arithmetic. Our implementation achieves 14–34 times speedup for index comparison and 4–18 times speedup for data aggregation compared to a pure CPU software implementation.

Keywords: Private information retrieval · Homomorphic encryption · NTRU

1 Introduction

A Private Information Retrieval (PIR) permits Alice to store a database D at a remote server Bob with the promise that Alice can retrieve $D(i)$ without revealing i or $D(i)$ to Bob. An information theoretic PIR scheme was first introduced in [1] where Bob's knowledge of i was limited using information theoretic arguments. Chor and Gilboa [2,3] introduced the concept of computational PIRs (cPIR). In cPIR, Alice is content to have Bob facing instead a computationally *difficult* problem to extract any significant information about i or $D(i)$. In [4] Kushilevitz and Ostrovsky presented the first single server PIR scheme based on the computational difficulty of deciding the quadratic residuosity of a number modulo a product of two large primes. Other cPIR constructions include [5] which is based on the computational difficulty of deciding whether a prime p divides $\phi(m)$ for any composite integer m of unknown factorization where $\phi()$

© International Financial Cryptography Association 2015
M. Brenner et al. (Eds.): FC 2015 Workshops, LNCS 8976, pp. 160–171, 2015.
DOI: 10.1007/978-3-662-48051-9_12

denotes Euler's totient function. In [6] another cPIR scheme was presented that generalizes the scheme in [5] while using a variation on the security assumption. The construction in [6] achieves a communication complexity of $O(k + d)$ where k is the security parameter satisfying $k > \log(N)$, N is the database size, and d is the bit-length of the retrieved data. In [7] Lipmaa presented a different cPIR scheme that employs an additively homomorphic encryption scheme with improved communication performance. Later in [8], an efficient PIR scheme was constructed using a partially homomorphic encryption algorithm. The first lattice based cPIR construction was proposed by Aguilar-Melchor and Gaborit [9]. Olumofin and Goldberg [10] revisited the performance analysis and found that the lattice-based PIR scheme by Aguilar-Melchor and Gaborit [9] to be an order of magnitude more efficient than the trivial PIR. These schemes utilize a combination of clever approaches and a diverse set of tools to construct cPIR schemes. Clearly given a fully or somewhat homomorphic encryption (FHE or SWHE) scheme achieving a cPIR construction would be conceptually trivial. With the recent advances and renewed interest in homomorphic encryption, new FHE schemes [11–15] and optimizations such as modulus and key switching [16], batching and SIMD optimizations [17] have become available. The more recent work by Doröz, Sunar and Hammouri [18] leveraged an NTRU based leveled SWHE scheme along with optimizations to construct an efficient cPIR. The rather simple cPIR construction has excellent bandwidth performance compared to previous implementations, i.e. about three orders of magnitude. However, to enable SWHE evaluation the scheme uses large parameters and therefore the computational cost is excessively higher than traditional PIR constructions, i.e. about 1-2 orders of magnitude.

Our Contribution. We present an Nvidia GPU implementation of a cPIR scheme based on SWHE proposed by Doröz et al. We develop optimized CUDA code to support large degree/large coefficient polynomial arithmetic operations such as modular multiplication/reduction, and modulus switching. For efficiency, we utilize number theoretical transform (NTT) based polynomial multiplication while the operands are kept in the CRT domain representation. The CUDA arithmetic library is then used to implement the two modes of the Doröz et al. cPIR [18]. While the bandwidth requirements are the same as in [18] our implementation is significantly faster. For instance, for a database size of 64 K entries, our index comparison implementation is about 33 times faster while the data aggregation operation is 18 times faster than the implementation of [18].

2 Background

In this section, we briefly explain the Doröz-Sunar-Hammouri (DSH) SWHE based PIR construction. First, we give details about the NTRU based SWHE scheme proposed by López-Alt, Tromer and Vaikuntanathan (ATV) [15]. Second, we explain the PIR construction proposed by Doröz et al. which is based on ATV-SWHE scheme.

ATV-SWHE Scheme: The NTRU scheme [19] was originally proposed as a public key encryption scheme. It was later modified by Stehlé and Steinfeld [20] to create a variant whose security is based on the ring learning with error (RLWE) problem. With a number of optimizations, López-Alt, Tromer and Vaikuntanathan extended the construction to a multi-key FHE scheme [15].

Doröz, Sunar, and Hammouri [18] tailored the ATV-SWHE to construct an efficient cPIR. Here we briefly summarize the construction: The polynomials are sampled from a probability distribution χ on B-bounded polynomials in $R_q := \mathbb{Z}_q[x]/(x^n+1)$ where a polynomial is "B-bounded" if all of its coefficients lie in $[-B, B]$. The sampled polynomials are used to compute public/secret keys. The scheme can support XOR and AND operations using polynomial additions and multiplications respectively. Noise grows significantly in AND operations, so after each AND operation noise is managed using a technique call modulus switch that is introduced by Brakerski, Gentry and Vaikuntanathan [12]. In modulus switch, a decreasing sequence of moduli $q_0 > q_1 > \cdots > q_d$ are selected for each level of the circuit. Also each modulus holds following property: $q_i | q_{i+1}$. The primitive functions of the DSH scheme is as follows:

- KeyGen: Choose m-th cyclotomic polynomial $\Phi_m(x)$ of degree $n = \varphi(m)$ as the modulus polynomial. Sample u and g from the distribution χ and compute $f = 2u + 1$ and $h = 2g\,(f)^{-1}$ in ring $R_{q_i} = \mathbb{Z}_{q_i}[x]/\langle \Phi(x) \rangle$.
- $\mathsf{Encrypt}$: Sample s and e from χ distribution and compute $c = hs + 2e + b$, where $b \in \{0,1\}$ is message and h is public key.
- $\mathsf{Decrypt}$: Message m is computed by: $m = cf^{(i)} \pmod 2$. The $f^{(i)}$ corresponds to private key for i^{th} level that holds: $f^{(i)} = f^{2^i} \in R_{q_i}$.
- XOR: The addition of two ciphertexts $c_1 = \mathsf{Enc}(b_1)$ and $c_2 = \mathsf{Enc}(b_2)$ corresponds to XOR operation, i.e. $c_1 + c_2 = \mathsf{Enc}(b_1 \oplus b_2)$.
- AND: The multiplication of two ciphertexts corresponds to AND operation, i.e. $c_1 \times c_2 = \mathsf{Enc}(b_1 \cdot b_2)$. After each multiplication, noise level is controlled by applying modulus switch: $\tilde{c}(x) = \left\lfloor \frac{q_{i+1}}{q_i} \tilde{c}(x) \right\rceil_2$.

DSH-PIR Scheme: For a given database D with $|D| = 2^\ell$ rows and given index $x \in \{0,1\}^\ell$ we may retrieve data d_x as follows: $\sum_{y \in [2^\ell]} (x = y) d_y \pmod 2$, where y is an index of the database. The equality check $(x = y)$ is computed using the bits of the indices as: $\prod_{i \in [\ell]} (x_i + y_i + 1)$. Basically, we compare if the bits of x are same with the bits of y for the same positions. If all the bits are same, the product yields a 1. Otherwise, the product evaluates to a 0. Therefore, we can retrieve d_y by computing the sums of products between the comparison results and the corresponding data d_y entries. In the protocol, the bits of the search index x_i are given in encrypted form while the comparison index y_i is in cleartext. Therefore, the $(x = y)$ circuit has to compute the product of ℓ polynomials, i.e. $\prod_{i \in [\ell]} (\mathsf{Enc}(x_i) + y_i + 1)$. This is evaluated with a depth $\log_2(\ell)$ circuit by using a binary tree. Furthermore, the scheme may be extended into a symmetric PIR in which also data is encrypted. In other words, we may encrypt the data as $\mathsf{Enc}(d_y)$ and multiply it with the corresponding ciphertext. This increments the depth of circuit level $\log_2(\ell) + 1$.

Query Modes: Doröz et al. [18] propose two query modes: Single Query and Bundled Query by two complementary uses of a batching technique introduced by Smart and Vercauteren [16,17]. The technique relies on the CRT where a pack of message bits are encoded into a single binary polynomial using inverse-CRT. This allows us to evaluate a circuit on multiple independent data inputs simultaneously by embedding them into the same ciphertext. The working mechanisms of the two query protocols is as follows:

- **Bundled Query.** In Bundled Query, a client creates multiple queries and batches them into a ciphertext, so the server can process multiple requests at a time. First, the client prepares multiple queries $\beta[j]$, which $j \in \varepsilon$ and ε is the total message slot number, with the following bit representation $\{\beta_{\ell-1}[j], \ldots, \beta_0[j]\}$. Then, the client encrypts and encodes the message polynomials for each bit index as: $\tilde{\beta}_i(x) = \mathsf{Enc}(\mathrm{CRT}^{-1}(\beta_\mathrm{i}[1],\ \beta_\mathrm{i}[2],\ \ldots\ ,\ \beta_\mathrm{i}[\varepsilon]))$. For each bit location we have a ciphertext which is ℓ in our case. Once the ciphertexts $\tilde{\beta}_i(x)$ are ready, they are sent to the server. The server computes the PIR using the formula: $r(x) = \sum_{y \in [2^\ell]} \left(\prod_{i \in [\ell]} \left(\tilde{\beta}_i(x) + y_i(x) + 1 \right) \right) d_y(x)$. Here $y_i(x)$ is the batched and encoded row index bits $\{y_i, y_i, \ldots, y_i\}$, which results in a single bit result $y_i(x) = y_i$, since each message bit has same value. Once the $r(x)$ is evaluated the server sends the result to the client who then decrypts and decodes the ciphertext and forms a list of retrieved values $\{d_0, d_1, \ldots, d_{\varepsilon-1}\} = \mathsf{Decode}(\mathsf{Dec}(r(x)))$.
- **Single Query.** In this mode, the client only gives one index query. The server encodes the indices and the database entries to perform comparison operations for different entries in parallel for same index query. Basically, the client takes the bits of a query β, encrypts each bit $\tilde{\beta}_i(x) = \mathsf{Enc}(\beta_i)$ and sends the ciphertexts to the server. The server computes the PIR operation: $r(x) = \sum_{y \in [2^\ell]} \left(\prod_{i \in [\ell]} \left(\tilde{\beta}_i(x) + y_i(x) + 1 \right) \right) d_y(x)$. However, this time $y_i(x)$ and $d_y(x)$ are binary polynomials since we are comparing a single query to a block of entries. The term $y_i(x)$ is computed by encoding bits at the same locations of different entries, i.e. $y_i(x) = \text{inverse-CRT}\{y_i[1], \ldots, y_i[\varepsilon]\}$. Similarly, d_y is also equal to inverse-CRT$\{d_0, \ldots, d_{\varepsilon-1}\}$. The process compares ε indices simultaneously in each iteration which decreases the overall runtime of the scheme. Once $r(x)$ is computed, the client receives the computed ciphertext which he then decrypts and decodes. If all the bits on the message slots are zero, than $d_y = 0$ and $d_y = 1$ otherwise.

3 GPU Implementation

Here we present an overview of the NTRU based PIR protocol implementation on CUDA GPUs. The client sends an encrypted query to the server. The server homomorphically evaluates the retrieval request, and returns a single ciphertext to the client. We optimize the scheme to better fit our target GPU device, i.e. NVIDIA GeForce GTX690. Nevertheless, the parallelization and optimization

techniques we employ here should work on other GPUs as well. Our GPU device consists of two GK104 chips where each chip holds 1536 cores and 2 GBytes of memory. We make use of both chips and almost evenly distribute workload to the two chips. The performance can be easily improved with a multi-GPUs setup, since the algorithm we present here has a high degree of parallelism.

Platform Overview and Design Strategy. A GPU based server or cluster consists of multiple GPUs. Every GPU is an efficient many-core processor system designed to reach high performance by exploiting parallelism in the computational task. Each *core* can execute a sequential thread. All cores in the same group execute the same instruction at the same time. With a GPU with *warp size* of 32, the code is executed in groups of 32 threads. Threads are further grouped into blocks. All threads in the same block are executed on a single multiprocessor, and therefore are able to share a single software data cache and memory. With general-purpose computing on GPUs, a portion of code that can achieve significant speedup when executed in parallel runs on the GPU, while other code remains on the CPU. Basically, data sets are sent to GPU memory, processed on the GPU and returned from GPU memory. This will pay off as long as the speedup of processing on GPU over on CPU outweighs the latency introduced by the input and output transfers. When GPU cores are handling tasks, besides the computation, memory access latencies also limit the performance gain. Memory on GPU is classified into several types which we list according to the access latency from low to high along with sizes for the Nvidia GeForce GTX 690 GPU: constant memory (64 KB), shared memory (48 KB per block), and global memory (4 GB). Given the drastically different make up of our target platform, it becomes clear that when translating our algorithms into GPU code we need to create a high degree of parallelism and minimize dependencies between data entries to take advantage of the multi-core architecture. Also data transfers between GPU and CPU needs to be reduced to a minimum. In our target application, we are processing a PIR database. The PIR database information is preloaded into the GPU memory. With a database index of b bits (e.g. $b = 32$, 16 or 8 bits), we send the query packaged into b ciphertexts to the server. After retrieval the query returns a single ciphertext per database entry bit back to the client. The entire retrieval computation is performed on the GPUs. For polynomial coefficient-wise operations, since we chose a large polynomial size ($n = 4096, 8190$ or 16384), we achieve a significant level of parallelism without any effort simply due to the way parameters are selected for security. For polynomial operations we introduce two conversions: CRT and NTT. CRT is able to divide any type of polynomial operation into independent operations. NTT is crucial to efficiently support parallel large polynomial multiplication.

GPU Memory. We store polynomials as 1D arrays of integers. To prevent memory contention between the read and write operations by the kernels, we pre-allocate memory pools on GPUs and divide them into smaller chunks with enough space for the CRT domain polynomials. We not only avoid the overhead associated with frequently allocating memory, but also can reclaim memory and reduce the overall memory usage. We always use the faster memory type

available, minimize the global memory access in the kernels, and adjust the number of threads per block to match the shared memory size. We store some data used to generate database indices in advance, since it is constant and takes only 64 MBytes in our largest setting. The input/output data transfer latencies are partially hidden behind database index generation and inverse-CRT operations. We use shared memory as data buffers in CRT and NTT kernels to ensure coalesced memory access.

Mapping the PIR Computation to CUDA Kernels. As described earlier, the Doröz et al. PIR [18] computation is $\sum_{y\in[2^\ell]}\left[\prod_{i\in[\ell]}(x_i+y_i+1)\right]d_y$. The query retrieval scheme mainly consists of polynomial multiplications over R_q. In addition, as circuit level increases, a modular reduction operation on ciphertexts, i.e. modulus switching, is performed on the polynomial coefficients. For ℓ bits of data, there are $(\ell-1)$ multiplications and $(\ell-1)$ modular reductions. We aim at optimizing these two operations. The input ciphertexts are polynomials in the ring R_q. The size of q is very large according to our parameter selection (e.g. 512 bits). We use CRT to convert large polynomials with large integer coefficients into a set of large polynomials with coefficient size small enough to permit streamlined processing. In the transformation we use a series of prime moduli $p_1, p_2, \ldots, p_l$. The result is recovered by computing the CRT inverse.

Chinese Remainder Theorem. With l prime numbers $p_1,\ p_2,\ \ldots\ ,\ p_l$, we obtain a set of independent polynomials from the polynomial $a(x)\in R_q$ as $a(x)=\mathrm{CRT}^{-1}\{a[1](x),\ a[2](x),\ \ldots\ ,\ a[l](x)\}$. If $a(x)=a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+\cdots+a_1x+a_0$, we have: $a_i=\mathrm{CRT}^{-1}\{a_i[1],\ a_i[2],\ \ldots\ ,\ a_i[l]\},\ i\in\mathbb{Z}_n$. Since the inverse-CRT returns a result in $\mathbb{Z}_{p_ip_{i-1}\ldots p_1}$ instead of $\mathbb{Z}_q$, modular reduction on coefficients is needed. Wang et al. [21] introduced a large integer modular reduction implementation on GPUs. Given that a single modular reduction costs multiple threads, processing all the coefficients in a polynomial would be inefficient. In [13], the authors proposed a way to combine inverse-CRT and modulo q reduction steps. We generate q as the product of a sequence of prime numbers and use the prime numbers for CRT: $q_i=\prod_{j=1}^{l-i}p_j,\ i<j$. The size of the primes, should be large enough to control the noise growth during homomorphic evaluations. The upper bound on the prime numbers depends on the polynomial multiplication scheme which will be explained later. After the initial CRT conversion on the input ciphertexts, all the computations are performed in the CRT domain, until we have to compute the inverse-CRT on the output ciphertexts. Focusing more closely on inverse CRT operation, we notice that after the inversion is carried out the coefficients remain in $\mathbb{Z}_{p_1p_2\cdots p_{l-i}}=\mathbb{Z}_{q_i}$, which means that modulo q reduction is no longer needed. We can also obtain modulo q_j result from a coefficient in $\mathbb{Z}_{q_i}$ for $j>i$. For all $z=\mathrm{CRT}^{-1}\{z[1],\ z[2],\ \ldots\ ,\ z[l-i]\}\in\mathbb{Z}_{q_i}$, given that $j>i$, we have $z\pmod{q_j}=\mathsf{CRT}^{-1}\{z[1],\ z[2],\ \ldots\ ,\ z[l-j]\}$.

Polynomial Multiplication. In [22], the authors present an efficient polynomial multiplication algorithm on CUDA GPUs. They basically follow Strassen's scheme [23] to multiply two polynomials $a(x)=\sum_{i=0}^{n-1}a_ix^i$ and $b(x)=\sum_{i=0}^{n-1}b_ix^i$. Consider the n coefficients of a polynomial as elements in a 0 padded

array of $2n$ elements: $a = \{a_0, a_1, \ldots, a_{n-1}, 0, \ldots, 0\}$ and $b = \{b_0, b_1, \ldots, b_{n-1}, 0, \ldots, 0\}$. Perform $2n$-point NTT on a and b to obtain arrays $A = \mathsf{NTT}(a)$ and $B = \mathsf{NTT}(b)$. Compute the element-wise product $C = A \cdot B$. Finally recover $c = \mathsf{NTT}^{-1}(C)$, in which the elements are the coefficients of $c(x) = a(x) \cdot b(x)$. Four-step Cooley-Tukey NTT iterations [24] are adopted for a fast NTT computation and hence create parallelism for CUDA GPU processors. A special prime number $P =$ `0xFFFFFFFF00000001` is chosen for better performance [25]. Since P should be larger than the possible largest coefficient of the polynomial product, we have our limit for the size of CRT prime numbers: $p_i < \sqrt{P/n},\ (i = 1, 2, \ldots l)$.

Polynomial Reduction. In the generic NTRU scheme all polynomial operations are performed in $R_q = \mathbb{Z}_q/\langle \Phi_m(x) \rangle$. Polynomial multiplication is therefore followed by a polynomial reduction. The cyclotomic polynomial modulus $\Phi_m(x)$ is a factor of the special polynomial $(x^m - 1)$. We need to perform a polynomial reduction after every polynomial multiplication with Barrett Reduction which by itself costs three polynomial multiplications. Instead we keep the product in $R'_q = \mathbb{Z}_q/\langle x^m - 1 \rangle$ form during query retrieval and only further (fully) reduce the polynomials to $R_q = \mathbb{Z}_q/\langle \Phi_m(x) \rangle$ in decryption. Polynomial reduction can be achieved by $c(x) \pmod{x^m - 1} = c(x) \pmod{x^m} + c(x)/x^m$. However, the latter method could possibly increase the size of polynomial operands greatly. For instance, with parameters $(n, m) = (16384, 21845)$, we have to conduct 65536-point NTT to multiply two 21845-degree polynomials, instead of 32768-point NTT for 16384-degree multiplication. Nevertheless, assuming the overhead of single sized multiplication is T, double sized NTT takes about $2T$, which is better than the method with Barrett Reduction [26] that takes more than $4T$ in total. Moreover, with parameters $(n, m) = (8190, 8191)$, we can still use the 8192-point NTT.

Modulus Switching. In the NTRU based SWHE, when the circuit level increases, e.g. from level i to level $(i+1)$, polynomials should be scaled from ring R_{q_i} to ring $R_{q_{i+1}}$ without disturbing the message embedded within the ciphertext. This requires a modular reduction operation on coefficients, namely modulus switching. For a ciphertext $c \in R_{q_i}$ at level i, obtain $c' \in R_{q_{i+1}}$ at level $(i+1)$ as follows. First compute $c' = c \pmod 2$. Note that c' will be close to $c \cdot \frac{q_{i+1}}{q_i}$. This operation is coefficient independent, hence can be executed in parallel. We explain the procedure on a single coefficient. Let a be the target coefficient in $\mathbb{Z}_q$. One way is to first compute $a' = \lfloor a \cdot \frac{q_{i+1}}{q_i} \rceil$. Then if $a' \neq a \pmod 2$, add or subtract 1 for a' to satisfy the equality. The method requires a to stay in $\mathbb{Z}_{q_i}$. Since in our implementation all operands are kept in the CRT domain, with a straightforward implementation we would have to call the expensive inverse-CRT in every level of the circuit. A technique to avoid the conversion by performing modulus switching in the CRT domain was proposed in [13]. Since $q_i = \prod_{j=1}^{l-i} p_j$ is the product of a sequence of CRT prime numbers, the first step of the previous method can be represented as $a' = \lfloor \frac{a}{p_{l-i}} \rceil + \epsilon, \epsilon \leftarrow \{-1, 0, 1\}$. If $r = a \pmod{p_{l-i}}$, we have $a' \cdot p_{l-i} = a - r + \epsilon \cdot p_{l-i}$. Let $a^* = r - \epsilon \cdot p_{l-i}$. If and only if a^* is even, $a' = a \pmod 2$. Therefore, $a' = \frac{a - a^*}{p_{l-i}} \in \mathbb{Z}_{q_{i=1}}$ is the result and we only need a^*. Starting from $a = \mathrm{CRT}^{-1}\{a[1],\ a[2],\ \ldots,\ a[l-i]\}$, let $a^* = a \pmod{p_{l-i}} = a[l-i]$.

If a^* is odd, add or subtract p_{l-i} to a^* so that $a^* \in (-p_{l-i}, 2p_{l-i})$ is even. Let $a'[j] = (a[j] - a^*)/p_{l-i} \pmod{p_j}$, $j = 1, 2, \ldots, l-i-1$. Then we have $a' = \mathrm{CRT}^{-1}\{a'[1],\ a'[2],\ \ldots\ ,\ a'[l-i-1]\}$ as the new coefficient in $\mathbb{Z}_{q_{i+1}}$.

Algorithm 1. Modulus Switching on Coefficients

Input: Coefficient $a = \mathrm{CRT}^{-1}\{a[1],\ a[2],\ \ldots\ ,\ a[l-i]\}$ from level i
Output: Coefficient $a' = \mathrm{CRT}^{-1}\{a'[1],\ a'[2],\ \ldots\ ,\ a'[l-i-1]\}$ for level $(i+1)$
1: $a^* \leftarrow a[l-i]$
2: **if** $a^* = 1 \pmod 2$ **then**
3:     **if** $a^* > (p_{l-i} - 1)/2$ **then**
4:         $a^* \leftarrow a^* - p_{l-i}$
5:     **else**
6:         $a^* \leftarrow a^* + p_{l-i}$
7:     **end if**
8: **end if**
9: **for** $j \leftarrow 1$ **to** $l-i-1$ **do**
10:     $a'[j] \leftarrow (a[j] - a^*)/p_{l-i} \pmod{p_j}$
11: **end for**

CUDA Kernels on Devices. As described earlier, the GPU devices receive b ciphertexts. Let d be the number of circuit levels for computing the product of the ciphertexts. Using a binary tree: $2^{d-1} < b \leqslant 2^d$. In Fig. 1, we show the process of computing a single product term of $\sum_{y\in[2^\ell]} \left[\prod_{i\in[\ell]} (x_i + y_i + 1)\right] d_y$ on a GPU for $b = 32$. After the last step an additional multiplication operation is required for generating the response. This consists of either a single multiplication or multiple multiplications depending on bundled query or single query mechanism. Ignoring the last step, we have $(b-1)$ polynomial multiplications or one binary tree of d depth, and $(b-1)$ modulus switchings:

- **Multiplication.** First we convert the polynomials to the CRT domain with l prime numbers. In the first level we have $(l \times b)$ polynomials. In the subsequent levels of the computation tree, l is decremented after each modulus switching operation. The number of parallel computation threads is initialized with b and is reduced by half after each multiplication level. At the end we obtain a $((l-d) \times 1)$ polynomials. Ideally we would like to distribute the workload evenly to all devices. Since the Nvidia GeForce GTX 690 only has two GPUs, each device is provided with $(l \times (b/2))$ polynomials to process. Until the last final multiplication the devices work independently.
- **Modulus Switching and Reduction.** In the second stage we process a half binary tree with modulus switching on each device. We need 3 kernels per polynomial per CRT element for each of NTT and INTT, 1 kernel per polynomial for coefficient wise multiplication in NTT domain and 1 kernel per polynomial for modulus switching. A polynomial reduction is performed after multiplication. We hide polynomial reduction at the beginning of modulus switching and inverse-CRT, instead of an extra kernel only for polynomial reduction.

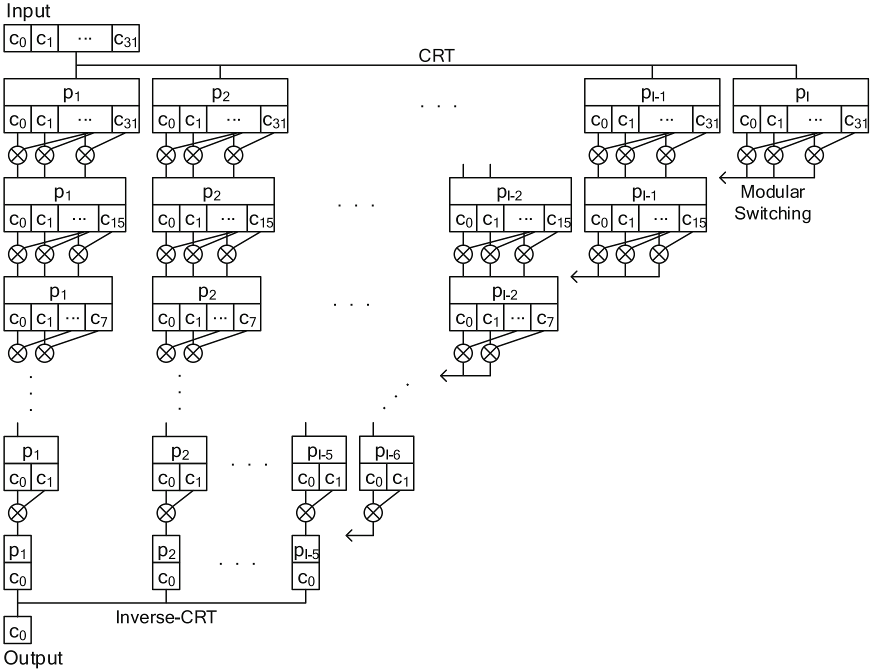

Fig. 1. Realization of the comparison circuit in the PIR scheme using single GPU for database with 2^{32} entries.

4 Performance

We implemented the DSH-PIR with both the Single and Bundled Querying modes using Nvidia GeForce GTX 690[1] running @915 Mhz with 3072 stream processors and 4 GBytes of memory. In Table 1, we compare our query/response sizes with DSH scheme [18] for different entry sizes $N = 2^{2^d}$. In our implementation query/response sizes are slightly larger because we choose the closest modulus in DSH scheme that holds $q = 24^k$. In Table 2, we compare our Bundled/Single Query computation times with DSH implementation which the timings are normalized with message slot size ε. For an index comparison of a Single Query, we achieved 15 times speedup for $d = 5$ and ~ 33 times for $d = \{4, 3\}$ cases. In data aggregation[2], we achieved 4–6 times speedup compared to DSH Scheme. In Table 3, we compare the Query Size of our scheme with BGN, O-K and DSH schemes for various database sizes. Our ciphertext sizes are (almost) identical to those of the DSH scheme. When compared to the

[1] The NVIDIA GeForce GTX 690 series actually consist of two GTX 680 series graphical processors.

[2] In cases where we extract entries with more than 1-bit size, we use the same index comparison result to process the remaining bits of a database entry. Also timings that given on the table are per polynomial operation and they are not normalized.

Table 1. Polynomial parameters and Query/Response sizes necessary to support various database sizes N.

Schemes	N	$(\log q,\ n)$	ε	Query size (MB)	Response size (KB)
DSH [18]	2^{32}	(512, 16384)	1024	32	784
	2^{16}	(250, 8190)	630	3.9	154
	2^{8}	(160, 4096)	256	0.625	44
Ours	2^{32}	(528, 16384)	1024	33	796
	2^{16}	(264, 8190)	630	4.12	164
	2^{8}	(168, 4096)	256	0.656	46.8

Table 2. Index comparison and data aggregation times per entry in the database for (d, ε) choices of $(5, 1024), (4, 630)$ and $(3, 256)$ on GPU.

	Depth (d)	Bundled query (msec)			Single query (msec)		
		5	4	3	5	4	3
DSH [18]	Index comparison	4.45	0.71	0.31	4.56	2.03	1.29
	Data aggregation	0.22	0.09	0.04	37	7.45	3.40
Ours	Index comparison	0.26	0.04	0.02	0.31	0.06	0.04
	Data aggregation	0.037	0.005	0.004	9.60	1.26	0.71
Speedup	Index comparison	×**17**	×**18**	×**15**	×**15**	×**34**	×**32**
	Data aggregation	×**6**	×**18**	×**10**	×**4**	×**6**	×**5**

Table 3. Comparison of query sizes for databases upto 2^{32}, 2^{16} and 2^{8} entries. Bandwidth complexity is given in the number of ciphertexts; α denotes the ciphertext size.

	BW compl.	α			Query size		
		$d = 5$	$d = 4$	$d = 3$	$d = 5$	$d = 4$	$d = 3$
Boneh-Goh-Nissim	$\alpha\sqrt{N}$	6144	6144	6144	96 MB	384 KB	24 KB
Kushilevitz-Ostrovsky	$\alpha\sqrt{N}$	2048	2048	2048	32 MB	128 KB	8 KB
DSH [18] (Single)	$\alpha \log N$	1 MB	249 KB	80 KB	32 MB	3.9 MB	640 KB
DSH [18] (Bundled)	$\alpha \log N$	1 KB	406 B	130 B	32 KB	6.32 KB	2.5 KB
Ours (Single)	$\alpha \log N$	1.03 MB	263 KB	84 KB	**33 MB**	**4.1 MB**	**672 KB**
Ours (Bundled)	$\alpha \log N$	1.03 KB	429 B	336 B	**33 KB**	**6.34 KB**	**2.6 KB**

K-O scheme in Bundled Query mode, our ciphertext sizes are three orders of magnitude smaller for $d = 5$ and an order of magnitude smaller for $d = \{4, 3\}$.

In Table 4, we compare our timing and ciphertext size estimates for a real time application given by Aguilar-Melchor and Gaborit [9]. The information given in the table is for $N = 1024$ entries with each entry holding 3 MBytes of data. We select $d = 4$ and assume both GPUs are running data aggregation tasks. In bundled query mode, we are better both in query size and amortized timings compared to other schemes results with the exception for query size of Gentry and Ramzan [6].

Table 4. Comparison of various schemes for real time applications.

Scheme	Query size	Computation time
Lipmaa	2 Mb	33 h
Gentry and Ramzan	3 Kb	17 h
Aguilar-Melchor and Gaborit	300 Mb	10 min
Ours (Single)	**20.6 Mb**	**8.8 h**
Ours (Bundled)	**33.4 Kb**	**1.5 min**

Acknowledgments. Funding for this research was in part provided by the US National Science Foundation CNS Award #1319130.

References

1. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. J. ACM **45**, 965–981 (1998)
2. Chor, B., Gilboa, N.: Computationally private information retrieval (extended abstract). In: Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing, STOC 1997, pp. 304–313. ACM, New York (1997)
3. Ostrovsky, R., Shoup, V.: Private information storage (extended abstract) (1996)
4. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: single database, computationally-private information retrieval. FOCS **1997**, 364–373 (1997)
5. Cachin, C., Micali, S., Stadler, M.A.: Computationally private information retrieval with polylogarithmic communication. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, p. 402. Springer, Heidelberg (1999)
6. Gentry, C., Ramzan, Z.: Single-database private information retrieval with constant communication rate. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 803–815. Springer, Heidelberg (2005)
7. Lipmaa, H.: An oblivious transfer protocol with log-squared communication. In: Zhou, J., López, J., Deng, R.H., Bao, F. (eds.) ISC 2005. LNCS, vol. 3650, pp. 314–328. Springer, Heidelberg (2005)
8. Boneh, D., Goh, E.-J., Nissim, K.: Evaluating 2-DNF formulas on ciphertexts. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 325–341. Springer, Heidelberg (2005)
9. Aguilar-Melchor, C., Gaborit, P.: A lattice-based computationally-efficient private information retrieval protocol (2007)
10. Olumofin, F., Goldberg, I.: Revisiting the computational practicality of private information retrieval. In: Danezis, G. (ed.) FC 2011. LNCS, vol. 7035, pp. 158–172. Springer, Heidelberg (2012)
11. Gentry, C., Halevi, S.: Implementing gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011)
12. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd ITCS, ITCS 2012, pp. 309–325. ACM, New York (2012)

13. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer, Heidelberg (2012)
14. Bos, J.W., Lauter, K., Loftus, J., Naehrig, M.: Improved security for a ring-based fully homomorphic encryption scheme. In: Stam, M. (ed.) IMACC 2013. LNCS, vol. 8308, pp. 45–64. Springer, Heidelberg (2013)
15. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the 44th Annual ACM STOC, STOC 2012, pp. 1219–1234. ACM New York (2012)
16. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 465–482. Springer, Heidelberg (2012)
17. Smart, N., Vercauteren, F.: Fully homomorphic SIMD operations. Des. Codes Crypt. **71**, 57–81 (2014)
18. Doröz, Y., Sunar, B., Hammouri, G.: Bandwidth efficient PIR from NTRU. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) FC 2014 Workshops. LNCS, vol. 8438, pp. 195–207. Springer, Heidelberg (2014)
19. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: a ring-based public key cryptosystem. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 267–288. Springer, Heidelberg (1998)
20. Stehlè, D., Steinfeld, R.: Making `NTRU` as secure as worst-case problems over ideal lattices. In: Paterson, K. (ed.) Advances in Cryptology-EUROCRYPT 2011. LNCS, vol. 6632, pp. 27–47. Springer, Heidelberg (2011)
21. Wang, W., Hu, Y., Chen, L., Huang, X., Sunar, B.: Accelerating fully homomorphic encryption using GPU. In: HPEC, IEEE, pp. 1–5 (2012)
22. Dai, W., Doröz, Y., Sunar, B.: Accelerating ntru based homomorphic encryption using gpus. (2014)
23. Schönhage, A., Strassen, V.: Schnelle multiplikation großer zahlen. Computing **7**, 281–292 (1971)
24. Cooley, J., Tukey, J.: An algorithm for the machine calculation of complex fourier series. Math. Comput. **19**, 297–301 (1965)
25. Emmart, N., Weems, C.C.: High precision integer multiplication with a gpu using strassen's algorithm with multiple fft sizes. PPL **21**, 359–375 (2011)
26. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (1987)