Tsunami: Massively Parallel Homomorphic Hashing on Many-core GPUs

Xiaowen Chu
Department of Computer Science
Hong Kong Baptist University
Hong Kong, P.R.C
chxw@comp.hkbu.edu.hk

Kaiyong Zhao
Department of Computer Science
Hong Kong Baptist University
Hong Kong, P.R.C
kyzhao@comp.hkbu.edu.hk

Zongpeng Li
Department of Computer Science
University of Calgary
Alberta, Canada
zongpeng@ucalgary.ca

ABSTRACT

Homomorphic hash functions (HHF) play a key role in securing distributed systems that use coding techniques such as erasure coding and network coding. The computational complexity of HHFs remains to be a main challenge. In this paper, we present a massively parallel solution, named *Tsunami*, by exploiting the widely available many-core Graphic Processing Units (GPUs). *Tsunami* includes the following optimization techniques to achieve the highest ever hashing throughput: (1) using Montgomery multiplication and pre-computation to speed up modular exponentiations; (2) using a clean implementation of Montgomery multiplication in order to decrease the demand of registers and shared memory and increase the utilization ratio of GPU processing cores; (3) using our own assembly code to implement the 32-bit integer multiplication, which outperforms the assembly codes generated by the native compiler by 20%; (4) exploiting memory alignment and constant memory on GPUs to improve the efficiency of memory access. Integrating the above techniques, our *Tsunami* achieves a significant improvement over existing results. Specifically, the hashing throughput achieved by *Tsunami* on a GTX295 GPU is about 33 times of the existing solution on a Quad-core CPU. We also show that the hashing throughput grows almost linearly with the number of GPU cores.

Keywords: Homomorphic hash function, GPU, CUDA

1. INTRODUCTION

In recent years, peer-to-peer (P2P) distributed systems such as file sharing application (e.g., BitTorrent) and multimedia streaming applications (e.g., ppLive, ppStream), have become the killer Internet applications. Erasure coding and network coding are very promising mechanisms to improve the performance of such P2P applications [7-15]. However, P2P applications with coding techniques suffer from the notorious pollution explosion problem: a malicious node can send out bogus packets which will be merged into other genuine packets and propagated into the whole network at an exponential speed [10] [12] [13]. To resolve this problem, homomorphic hash functions have been designed such that the homomorphic hash of any encoded packet can be effectively derived from the hashes of the original packets, which enables the detection of bogus packets before a peer encodes it with other packets [10]. Unfortunately homomorphic hash functions rely on multiple-precision modular operations and are computationally expensive. This computational challenge becomes the main obstacle that limits the application of coding techniques in peer-to-peer systems [12] [13].

On the other side, GPU computing has become a well-accepted parallel computing paradigm which shifts the domain of parallel computing from mainframes in data centers to personal computers that are widely available [19]. Commodity GPUs like NVIDIA's GTX 280 has 240 processing cores and can achieve 933 GFLOPS of computational horsepower. Furthermore, the NVIDIA CUDA programming model enables developers to easily develop non-graphic applications using GPU [1] [4]. In CUDA, the GPU becomes a dedicated coprocessor to the host CPU, which works in the principle of Single-Program Multiple Data (SPMD) where multiple threads based on the same code can run simultaneously. GPUs have recently been proposed to accelerate network coding [22-24]. Our previous work also applied GPU computing in accelerating homomorphic hashing [24-26].

In this paper, we propose the *Tsunami* system that uses GPUs for homomorphic hashing. The homomorphic hash function needs to multiply a large number of exponentiations, so the critical design concern of *Tsunami* is to optimize the modular exponentiation operation for high-precision large integers. To this end, we focus our work on the design, implementation, and optimization of modular exponentiation operations on GPU. Contemporary GPUs are optimized for floating-point operations; and integer operations are relatively poorly supported. Hence there are a lot of challenges to optimize high-precision large integer operations on GPUs. Nevertheless, we optimize *Tsunami* at the assembly code level, and achieved very promising result. The contribution of our work is threefold: First, we designed and implemented a fast modular exponentiation algorithm using Montgomery reduction and also precomputation, for the CUDA architecture. Second, we achieved 34.7 Mbps of hashing throughput using a graphic card with GTX295 GPU, which is 33 times of the throughput of existing solution on a quad-core CPU. Third, we developed a multiple-precision modular arithmetic library for CUDA, which could be applied to a lot of security applications such as RSA, and ElGamal schemes. We also make our source code available to the research literature.

The rest of the paper is organized as follows. Section 2 provides background information on Montgomery multiplication, homomorphic hash functions, GPU architecture and CUDA programming model. Section 3 presents the design of Montgomery multiplication on GPU. Section 4 presents the parallel implementation of homomorphic hash function. Experimental results are presented in Section 5, and we conclude the paper in Section 6.

2. BACKGROUND AND RELATED WORK

In this section, we provide the required mathematical background; introduce homomorphic hash function; and then briefly present GPU architecture and CUDA programming model.

2.1 Mathematical Preliminaries

In modular arithmetic, all operations are performed in group Z_w, i.e., the set of integers $\{0,1,\cdots,w-1\}$. In this paper, the modulus w is represented in radix b as $(w_{s-1}\cdots w_1 w_0)_b$ where $w_{s-1}\neq 0$. s is called the length of w. Each symbol w_i, $0\le i<s$, is referred to as a radix b digit. Non-negative integers x and y, $x<w$, $y<w$, are represented in radix b as $(x_{s-1}\cdots x_1 x_0)_b$ and $(y_{s-1}\cdots y_1 y_0)_b$ respectively.

The following gives the definitions of Montgomery reduction and Montgomery multiplication, which are well-known techniques for efficient implementation of modular exponentiation [5] [6]:

Definition 1 Given positive integers w and R such that $R > w$ and gcd(w, R) = 1. For integer x that $0 \le x < wR$, we define the Montgomery reduction of x modulo w with respect to R as xR^{-1} mod w.

Definition 2 Given positive integers w and R such that $R > w$ and gcd(w, R) = 1. For integers x and y with $0 \le x, y < wR$, we define the Montgomery multiplication of x and y modulo w with respect to R as $Mont(x, y) = xyR^{-1}$ mod w.

In practice, R is set to be a power of 2, because divisions by a power of 2 are simply shift operations which are efficiently supported by most platforms.

2.2 Homomorphic Hash Functions

We use network coding as an example to illustrate the importance of homomorphic hash functions. Consider a P2P content distribution application. The data to be distributed is divided into n blocks $(b_1, b_2, \ldots, b_n)$, where each block b_i is further divided into m codewords $b_{i,k}$, $k \in \{1,\ldots,m\}$. An encoded block e_j is a linear combination of the n original blocks and it is also divided into m codewords $e_{j,k}$, $k \in \{1,\ldots,m\}$. The linear relationship between e_j and the original n blocks is described by e_j 's ***global coefficient vector*** $(c_{j,1}, c_{j,2}, \ldots, c_{j,n})$, i.e., $e_{j,k} = \sum_{i=1}^{n} c_{j,i} \cdot b_{i,k}$, $k \in \{1,\ldots,m\}$. In a P2P application, a peer receives encoded data blocks from upstream peers, and also creates new encoded data blocks by randomly and linearly combining its received encoded blocks, and then disseminates the new encoded blocks to its downstream peers. Notice that the global coefficient vector should be derived and sent along with each new piece of encoded data block. A peer can recover/decode the original n blocks as soon as it has

received n linearly independent coded blocks $(e_1, e_2, \ldots, e_n)$, by solving the set of linear equations $e_{j,k} = \sum_{i=1}^{n} c_{j,i} \cdot b_{i,k}$, $k \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$.

P2P networks are prone to the pollution attacks in which bogus data blocks are disseminated into the network by malicious peers. When network coding is not deployed, each peer will receive original data blocks directly from other peers, and hence it is possible to use hash functions such as SHA1 to verify the correctness of a data block simply by comparing the hash of each received data block to the corresponding hash provided by the source. For P2P networks with erasure coding or network coding, the effect of pollution attack becomes much more difficult to handle [10] [12] [13], because each bogus block could be mixed with valid blocks and propagated throughout the network, resulting the pollution explosion. The traditional hash functions such as SHA1 cannot be applied here because a peer receives random encoded packets which cannot be predetermined by the source. Homomorphic hash functions are currently the only solution to this security issue, which enable a peer to detect the bogus data block once it has been received. Homomorphic hash functions have the property that the hash value of a linear combination of the input blocks can be constructed by the hash values of those input blocks. One such homomorphic hash function, $h(\cdot)$, has been proposed in [10], which requires to decide a set of hash parameters $G = (p, q, g)$ in advance. The parameters p and q are large prime numbers of order λ_p and λ_q chosen such that $q \mid p - 1$. The parameter g is a vector of m numbers (denoted by g_k, $1 \le k \le m$), each of which can be written as $x^{(p-1)/q} \bmod p$ where $x \in \mathbf{Z}_q$ and $x \neq 1$. The method of creating the parameter set can be found at [10]. The homomorphic hash of a data block b_i is then calculated as

$$h(b_i) = \prod_{k=1}^{m} g_k^{b_{i,k}} \bmod p. \tag{1}$$

The homomorphic hash values of the original blocks $(b_1, b_2, \ldots, b_n)$ are $h(b_1)$, $h(b_2)$, …, $h(b_n)$ respectively. Given an encoded block e_j with global coefficient vector $(c_{j,1}, c_{j,2}, \ldots, c_{j,n})$, the homomorphic hash function $h(\cdot)$ can be proved to satisfy the following condition:

$$h(e_j) = \prod_{i=1}^{n} h^{c_{j,i}}(b_i). \tag{2}$$

This property can be used to verify the authenticity of an encoded block. Typical values of the parameters are summarized in Table 1.

Although the homomorphic hash function can theoretically resolve the pollution attack problem, it is computationally expensive for today's desktop CPUs. A 3 GHz Pentium 4 CPU

can only achieve around 300 Kbps of throughput for verifying a single 16 KB data block, using the parameters in Table 1 [10]. Given the computational challenge of homomorphic hash functions, some compromised solutions have been proposed which sacrifices the security level [12] [13] [18]. This motivates us to explore other approaches to overcoming the computational challenge.

2.3 GPU Computing and CUDA

GPUs are dedicated hardware for manipulating computer graphics, and recently they have been evolved into highly paralleled, multithreaded, many-core processors. As an example, the NVIDIA GeForce GTX280 has 30 Streaming Multiprocessors (SMs), and each SM has 8 Scalar Processors (SPs), resulting in a total of 240 processor cores. The design of the SMs is based on the Single-Instruction Multiple-Data (SIMD) architecture, i.e., at any given clock cycle, all SPs of the same SM must execute the same instruction, but can operate on different data.

Off-chip memories on Graphic cards, such as local memory and global memory, have relatively long access time, usually 400 to 600 clock cycles [3]. Inside a GPU, each SM has four types of on-chip memory, namely, constant cache, texture cache, registers, and shared memory. The properties of the different types of memories have been summarized in [16]. Constant cache and texture cache are both read-only memories shared by all SPs. On GeForce GTX280, each SM has 16384 32-bit registers and 16KB shared memory that are almost as fast as the registers. Shared memory is divided into banks of equal size for simultaneous access. The banks are organized in a way such that successive 32-bit words belong to consecutive banks. If two memory requests fall into the same bank, it is referred to as a bank conflict; thus, the memory access must be serialized. For optimal memory access performance, one should minimize the chance of bank conflicts, and also utilize on-chip memory as much as possible.

The exceptional GPU computing power is very attractive to general-purpose system development, which is referred to as general-purpose computing on GPUs (GPGPU). Recently one of the major GPU vendors, NVIDIA, announced their new general-purpose parallel programming model, namely Compute Unified Device Architecture (CUDA) [1] [3], which extends the C programming language for general-purpose application development. Meanwhile, another GPU vendor AMD also introduced Close To Metal (CTM) programming model that provides an assembly language for application development [2]. Intel will release Larrabee [21], a new multi-core GPU architecture specially designed for GPU computing. Very recently, an open standard named OpenCL has been proposed for general-purpose parallel programming of heterogeneous systems [27]. It is expected to be supported by many-core GPUs as well as multi-core CPUs.

Currently, CUDA is the best available programming model, and is the most well accepted model by the research and development community. Since the release of CUDA, it has been used for speeding up a large number of applications [16] [17] [20]. More importantly, the OpenCL standard has a very similar structure to CUDA; hence the code written in CUDA can be easily migrated to OpenCL in the future. For these reasons, we chose to use CUDA in our research. More details of CUDA programming model can be found in [4].

3. Montgomery Multiplication on GPUs

Homomorphic hash functions require the calculation of a lot of modular exponentiations. As today's GPUs have a very slow implementation of integer divisions, it is advantageous to use Montgomery multiplication methods to implement fast modular exponentiations. In this section, we present the implementation of Montgomery multiplication on GPUs. A number of different implementations of Montgomery multiplication have been discussed in [28]. These implementations have the same number of multiplications but different number of additions and memory accesses. We first present our implementation of the Coarsely Integrated Operand Scanning method on GPU. We then present an alternate design by integrating the Karatsuba multiplication into the Separated Operand Scanning method.

3.1 Coarsely Integrated Operand Scanning (CIOS)

We choose to implement the Coarsely Integrated Operand Scanning (CIOS) method on GPU due to its low demand on memory space as compared with other alternatives. The algorithm computes $Mont(x, y) = xyR^{-1} \bmod w$ for multiple-precision integers x and y. In the CIOS algorithm shown in Figure 1, we choose radix b to be 2^{32} because the current CUDA platform uses the native 32-bit operations. Our Montgomery multiplication algorithm (i.e., Algorithm1 shown in Figure 1) chooses $R = b^s = 2^{32s}$ which satisfies the condition of gcd(w, R) = 1, as w is odd. An integer $w' = -w^{-1} \bmod b$ should be pre-computed by using extended Euclidean algorithm. The three multiple-precision integers x, y, w are stored in arrays $x[\ ]$, $y[\ ]$, and $w[\ ]$, respectively. A temporary array $t[\ ]$ is required to store intermediate results.

Complexity Analysis: The CIOS algorithm takes $2s^2+s$ 32-bit multiplications and $4s^2+4s$ 32-bit additions. It requires a storage space of $4s$ words. Notice that the 32-bit multiplication used in the above algorithm should output a 64-bit result, which is unfortunately not efficiently supported by the CUDA platform. This is because today's GPUs are optimized for processing floating-point numbers. For instance, on contemporary NVIDIA GPUs, single-precision floating-point multiplication requires 4 clock cycles, but a 32-bit integer multiplication requires 16 clock cycles. Even worse, the native 32-bit integer multiplication in CUDA only provides the low-32 bits of the 64-bit result. In order to get the 32 most significant bits of the 64-bit result, we have to call another function **_umulhi**(x, y) [4]. CUDA also provides an efficient implementation of 24-

bit multiplication and also multiplication-add (mad), i.e., **_umul24**(x, y) and **_umad24**(x, y), which require only 4 clock cycles. Therefore we develop our own implementation of full 32-bit multiplication on top of the 24-bit arithmetic. As to be shown in Section 5, this optimization technique improves the overall performance by 20%. Nevertheless, the optimized full 32-bit multiplication still takes more than 60 clock cycles, which is much slower than the CPU counterpart.

Algorithm 1 CIOS for Multiple-precision Montgomery Multiplication

INPUT: integers w, x, y with s radix b digits, $x < w$, $y < w$, and gcd(w, b) = 1, $R = b^s$, $w' = -w^{-1} \bmod b$.

OUTPUT: $T = x \cdot y \cdot R^{-1} \bmod w$.

```
1:  for (i from 0 to s-1)
2:      C ← 0
3:      for ( j from 0 to s-1)
4:          S ← t[j] + x[j] × y[i] + C
5:          t[j] ← S
6:          C ← S >> 32
7:      end for
8:      S ← t[s] + C
9:      t[s] ← S
10:     t[s+1] ← S >> 32
11:     C ← 0
12:     m ← t[0] × w'
13:     for (j from 0 to s-1)
14:         S ← t[j] + m × w[j] + C
15:         t[j] ← S
16:         C ← S >> 32
17:     end for
18:     S ← t[s] + C
19:     t[s] ← S
20:     t[s+1] ← t[s+1] + S >> 32
21:     for (j from 0 to s)
22:         t[j] ← t[j+1]
23:     end for
24: end for
```

Figure 1. CIOS for Multiple-precision Montgomery Multiplication

3.2 Karatsuba Montgomery Multiplication (KMM)

Karatsuba multiplication can potentially improve the efficiency of large integer multiplications. Its basic idea is to reduce a 2s-digit multiply into three s-digit multiply and a number of additions and subtractions, which is illustrated by the following example. Assume $N_1 = x_1 2^s + x_0$ and $N_2 = y_1 2^s + y_0$. We can have:

$$\begin{aligned} N_1 N_2 &= (x_1 2^s + x_0)(y_1 2^s + y_0) \\ &= x_1 y_1 2^{2s} + (x_1 y_0 + x_0 y_1) 2^s + x_0 y_0 \\ &= x_1 y_1 2^{2s} + ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) 2^s + x_0 y_0 \end{aligned}$$

By recursively applying this trick, the multiplication of two s-digit numbers takes $3s^{\log_2 3}$ single-digit multiplications at the expense of more additions and subtractions. We integrate Karatsuba multiplication into the Separated Operand Scanning method of Montgomery multiplication. One challenge of implementing Karatsuba multiplication on GPU is that, CUDA does not support recursion in the kernel code. We tackle this problem by hardcoding different levels of Karatsuba multiplication as different functions. Due to the limited space, we do not show the detailed pseudocode for Karatsuba Montgomery Multiplication. However, in Sec. 5 we will show that the performance of Karatsuba Montgomery Multiplication is in fact worse than the CIOS method on contemporary GPUs.

4. PARALLEL HOMOMOPHIC HASHING ON GPUS

The core of *Tsunami* system is a fast modular exponentiation engine implemented on GPUs. CPU gathers enough codewords from the application, and delivers the codewords to GPU to compute the required modular exponentiations. Recall that homomorphic hashing takes two steps: (1) perform m modular exponentiations; (2) perform m-1 modular multiplications. The first step is executed by GPU cores. Although step (2) can also be done by GPU using a standard parallel reduction technique, some GPU resources will be wasted during the reduction process. As Step (2) is not computationally demanding, we exploit a new CUDA feature named Asynchronous Concurrent Execution, which allows CPU to compute simultaneously with GPU: after the GPU finishes the calculation of modular exponentiations, the results are transferred back to CPU to perform Step (2); meanwhile, CPU transfers the next batch of codewords to GPU for calculating the modular exponentiations. By doing so, *Tsunami* achieves higher throughput by utilizing the computing power of GPU and CPU. The overall structure of *Tsunami* is shown in Figure 2.

Pseudocode of *Tsunami* on CPU

```
for ( ; ; ) {
    Get a number of data blocks;
    Form codewords and transfer them to GPU global memory;
    Call GPU kernel function to perform Step (1);
    Perform Step (2) for the previous round; // this is in parallel with Step (1)
    Receive results from GPU;
}
```

Figure 2. Pseudocode of *Tsunami* on CPU

4.1 Homomorphic Hashing by Montgomery Multiplication with Precomputation

In [24], homomorphic hashing is accelerated by GPUs using the classical binary Montgomery exponentiation method. In reality, when applying homomorphic hash function in distributed systems, the same set of parameters will be used for a large data set such as a computer file or a video streaming session. For this type of applications, it is possible to speed up the modular exponentiations by pre-computing some exponentiations. *Tsunami* integrates Montgomery multiplication into the precomputation method introduced in [29]. To compute a

modular exponentiation $g^e \bmod w$, we first represent the exponent e using radix $b = 2^k$: $e = \sum_{i=0}^{s-1} a_i b^i$, where $0 \le a_i < b$ and $a_{s-1} \ne 0$. It is easy to see that the length of e is $s = \lceil \log_b e \rceil$. The fast modular exponentiation algorithm requires the precomputation of a set of values, i.e., $g^{(b^i)} \bmod w$ for $1 \le i \le s-1$. *Tsunami* uses Algorithm 2 (shown in Figure 3) to compute $g^e \bmod w$.

Algorithm 2 Multiple-precision Montgomery Exponentiation with Precomputation

INPUT: integers w, g, $e = \sum_{i=0}^{s-1} a_i b^i$, R, and $Rg^{(b^i)} \bmod w$ for $1 \le i \le s-1$

OUTPUT: $g^e \bmod w$.

```
1:  A ← R, B ← R;
2:  for ( j from b − 1 down to 1)
3:     for i from 0 to s − 1
4:        if a_i == j then B ← Mont(B, Rg^(b^i) mod w);
5:     end for
6:     A ← Mont(A, B);
7:  end for
8:  A ← Mont(A, 1);
9:  return A;
```

Figure 3. Multiple-precision Montgomery Exponentiation with Precomputation

Theorem 1: Algorithm 2 outputs the value of $g^e \bmod w$.

PROOF: We first convert $g^e \bmod w$ into the form of $\prod_{d=1}^{b-1} c_d^d \bmod w$ where $c_d = \prod_{i:a_i=d} g^{(b^i)}$:

$$g^e \bmod w = g^{(a_{s-1}\cdots a_1 a_0)_b} \bmod w = g^{a_0 + a_1 b + \cdots + a_{s-1} b^{s-1}} \bmod w$$
$$= g^{a_0} (g^b)^{a_1} \cdots (g^{(b^{s-1})})^{a_{s-1}} \bmod w = \prod_{d=1}^{b-1} c_d^d \bmod w .$$

Next, we use mathematical induction to show that, after going through the loop from Line 2 to Line 7 in Algorithm 2 for t times, where $1 \le t \le b - 1$, we have $B = c_{b-1} c_{b-2} \cdots c_{b-t} R \bmod w$ and $A = c_{b-1}^t c_{b-2}^{t-1} \cdots c_{b-t} R \bmod w$.

Basic Step. Initially $A = R$ and $B = R$. For the case of $t = 1$ (i.e., $j = b - 1$), in each execution of Line 4, if we have $a_i = b-1$ then $Mont(B, Rg^{(b^i)} \bmod w) = Bg^{(b^i)} \bmod w$. By definition, $c_{b-1} = \prod_{i:a_i=b-1} g^{(b^i)}$. So after Line 5, we have $B = c_{b-1} R \bmod w$ and after Line 6 we have $A = Mont(R, B) = RBR^{-1} \bmod w = c_{b-1} R \bmod w$.

Induction Hypothesis. Assume that, after going through the loop from Line 2 to Line 7 in Algorithm 2 for t times where $1 \le t \le b - 2$, we have $B = c_{b-1} c_{b-2} \cdots c_{b-t} R \bmod w$ and $A = c_{b-1}^t c_{b-2}^{t-1} \cdots c_{b-t} R \bmod w$.

Induction Step. Consider the $(t + 1)th$ loop where $j = b - t - 1$. In each execution of Line 4, if we have $a_i = b-t-1$, then $B = Mont(B, Rg^{(b^i)} \bmod w) = Bg^{(b^i)} \bmod w$. By definition, $c_{b-t-1} = \prod_{i:a_i=b-t-1} g^{(b^i)}$. So after Line 5, we have $B = c_{b-1}c_{b-2}\cdots c_{b-t-1}R \bmod w$ and after line 6 we have $A = c_{b-1}^{t+1}c_{b-2}^{t}\cdots c_{b-t-1}R \bmod w$.

Hence, after $b - 1$ loops, $A = \prod_{d=1}^{b-1} c_d^d \cdot R \bmod w$. After executing Line 8, we have $A = AR^{-1} \bmod w = \prod_{d=1}^{b-1} c_d^d \cdot R \cdot R^{-1} \bmod w = \prod_{d=1}^{b-1} c_d^d \bmod w$.

Thus we have proved that $A = g^e \bmod w$. ■

Complexity Analysis: Algorithm 2 takes at most $s + b - 3$ Montgomery multiplications.

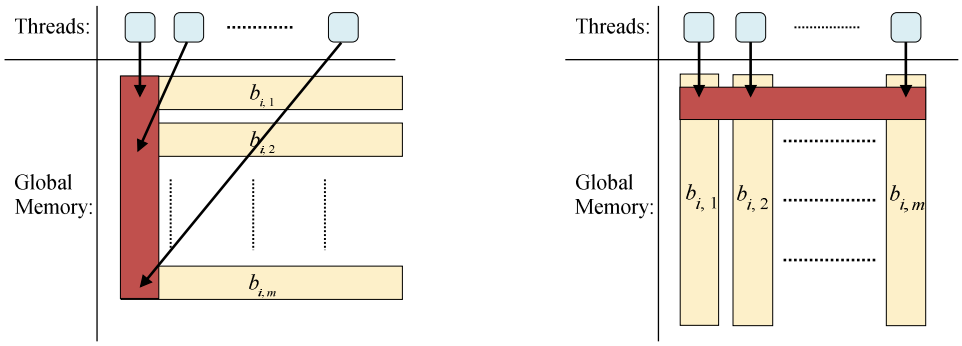
Optimal Value of *k*: For e with 257-bit, $s+b-3 = \lceil 258/k \rceil + 2^k - 3$. It is easy to find that the optimal result is achieved when $k = 4$, i.e, $b = 16$, which takes only 78 Montgomery multiplications in the worst case. As compared with 512 Montgomery multiplications required by the binary method [10] [24], this is a significant speedup.

4.2 Implementation Details

GPUs offer a very high internal global memory bandwidth, usually more than one hundred Gbps. However, the real bandwidth achieved by an application depends on the memory access pattern. Global memory bandwidth can be efficiently utilized when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction of 32, 64, or 128 bytes. When performing homomorphic hashing on a data block b_i, b_i is divided into m codewords $b_{i,k}$, $1 \le k \le m$. Each codeword $b_{i,k}$ takes 4 bytes and will be accessed by a single GPU thread. As shown in Figure 4 (a), if we organize the codewords in a normal two-dimensional array, the memory access pattern will be non-coalesced, and hence a separate memory transaction is issued for each thread which significantly reduces the throughput. *Tsunami* uses a coalesced data structure as shown in Figure 4 (b). For a half-warp (i.e., 16 threads), the 16 memory accesses can be coalesced into a single 64-byte memory transaction.

In GPU computing, shared memory has always been an important but scarce resource. In the design of *Tsunami*, we also tried different ways to make use of shared memory. Each thread needs to store three big integers during the computing of modular exponentiations. If each thread stores the big integers into shared memory, then the number of active threads that can be executed on an SM will be extremely limited, and this results in a poor utilization of GPU cores. The current design of *Tsunami* does not utilize any shared memory, so that the number of active

threads per SM is only limited by the number of registered used per thread. By carefully reducing the number of registers per thread, *Tsunami* can have 384 active threads per SM, which are enough to fully utilize the computing power of GPU cores. However, if the size of shared memory grows in the future generation of GPUs, we believe that making use of shared memory can further improve the throughput under the condition that using shared memory won't limit the number of active threads per SM.



(a) non-coalesced memory access (b) coalesced memory access

Figure 4. Global memory organization of codewords

Tsunami also utilizes constant memory to hide some of the memory access latency. For those constant variables (such as p) that are shared by different threads, we store them into constant memory, because each SM has on-chip cache for constant memory and hence the access to these constant variables will be very efficient.

5. EXPERIMENTAL RESULTS

The CPU version of the homomorphic hash function is implemented in C language using the GNU MP arithmetic library, version 4.2.3 [3]. We have also implemented the different implementations of homomorphic hash function using CUDA. We tested these implementations on tested it on three different GPUs: (1) GeForce GTX295; (2) GeForce GTX280; and (3) GeForce 9800. The properties of the three GPUs used in our experiments are summarized in Table 2. The host computer is equipped with a 2.4GHz Intel Quad-core CPU Q6600. All our experimental results are the average of ten runs.

5.1 Evaluation of Montgomery Multiplication

We first present our experimental results of the three different implementations of Montgomery multiplications. We tested single thread case as well as multiple thread cases on GTX295, and the results are shown in Figure 5. It is interesting to see that the time of computing 960 Montgomery multiplications is only about twice of computing a single Montgomery multiplication. This is because the 480 GPU cores have been fully utilized to perform Montgomery multiplications simultaneously.

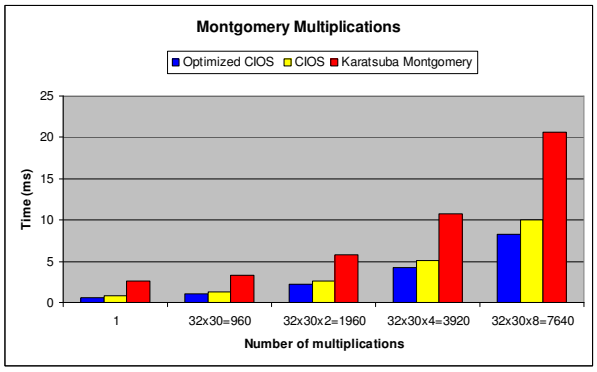


Figure 5. Performance of Montgomery multiplication on GTX295

The theoretically advantageous Karatsuba Montgomery method does not perform well on GPUs. We ascribe this to mainly two reasons: (1) Karatsuba multiplication extensively uses flow control statements, causing threads of the same warp to diverge. In this case, different execution paths within a warp will be serialized, increasing the total number of instructions executed for this warp and decreasing the throughput significantly. (2) Karatsuba multiplication requires a large memory space to hold intermediate results. Given the limited low-latency shared memory, the intermediate results have to be stored in global memory which has very long access latency. The Karatsuba Montgomery multiplication method needs 60 registers and 5132 bytes of local memories. But the CIOS method only needs 14 register and no local memory at all. As a result, the performance of Karatsuba Montgomery algorithm is about half of the CIOS algorithm.

Our optimized CIOS implementation makes use of the 24-bit arithmetic operations. On average, it can improve the throughput of CIOS by 20%. As mentioned before, our optimized implementation of full 32-bit integer multiplication takes more than 60 clock cycles. If the GPU vendors can improve their support of 32-bit integer multiplication in future generations of GPUs, we believe the performance of Montgomery multiplication (and hence a plenty of security applications) can be boosted significantly.

5.2 Evaluation of Homomorphic Hash Function

The performance of *Tsunami* depends on two system parameters: the number of threads per block (denoted by T), and the number of thread blocks. Our experimental results of GTX 280 with different configurations are shown in Figure 6. We notice that 128 or 256 threads per block can achieve stable and optimal performance. But if there are only 30 thread blocks, 128 threads per block will not be enough to hide the memory latency and hence the computing power cannot be fully utilized.

We summarize our experimental results of different implementations of homomorphic hashing in Table 3. We implemented the single thread version and multi-threaded version of method in [10] on CPU, and obtained a very close throughput: 304 Kbps has been reported in [10]

on a 3GHz Pentium 4 CPU while we got 280 Kbps on a 2.4GHz CPU. When all four CPU cores are utilized, the throughput can be improved to 1.03 Mbps, which is close to four times of the single thread result. As compared with the multithreaded result on the Quad-core CPU, *Tsunami* on GTX295 achieves 33.7x speedup.

GPU was first proposed to accelerate homomorphic hashing in [24]. However, the method used in [24] is a straightforward implementation of binary-exponentiation and it can only achieve 1.90 Mbps on GTX 280. *Tsunami* system achieves a speedup of 10.8x by using a number of optimization techniques.

We can also observe that the performance of *Tsunami* system grows almost linearly to the number of GPU cores. Future generations of GPUs are estimated to have more and more cores, and hence the throughput achieved by *Tsunami* will grow as well.

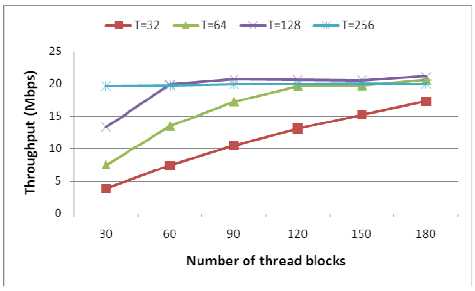


Figure 6. Throughput with different number of thread blocks and different threads per block (*T*), GTX 280

6. CONCLUSIONS

Homomorphic hashing is the key component for data authentication in distributed systems that reply on erasure coding or network coding. Unfortunately it is computationally expensive to perform homomorphic hashing on today's CPUs. In this paper, we show the design and implementation of a highly efficient solution, namely *Tsunami*, for homomorphic hashing by using GPUs. By using a contemporary graphic card, our parallel algorithm can achieve 34.7 Mbps of hashing throughput, which is 33 times faster than the contemporary quad-core CPU. The hashing throughput achieved by *Tsunami* is high enough for today's Internet applications. The excellent flexibility of *Tsunami* makes it a good candidate for future high-bandwidth peer-to-peer applications.

The recently introduced Fermi GPU architecture has enhanced the integer arithmetic unit by supporting native 32-bit integer multiplication. Our currently implementation of 32-bit integer multiplication replies on 24-bit integer arithmetic, which could be replaced by the Fermi native 32-bit integer instructions. It will be our future work to investigate how to optimize homomorphic hashing on Fermi architecture.

ACKNOWLEDGEMENT

This work is supported by the grant HKBU FRG2/09-10/081. We thank NVIDIA for providing the Geforce 9800 card for this research.

REFERENCES

[1] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html

[2] AMD CTM Guide: Technical Reference Manual. 2006. http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[3] GNU MP Arithmetic Library. http://gmplib.org/

[4] NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 2.0beta2, Jun. 2008.

[5] Montgomery, P., 1985. Multiplication without trial division, Math. Computation, vol. 44, 1985, 519-521.

[6] Menezes, A., van Oorshot, P., and Vanstone S., 1996. Handbook of applied cryptography. CRC Press, 1996.

[7] Ahlswede, R., Cai, N., Li S. R., and Yeung, R. W. 2000. Network information flow. IEEE Transactions on Information Theory, 46(4), July 2000, 1204-1216.

[8] Ho, T., Koetter. R., Médard, M., Karger, D.R. and Effros, M. 2003. The benefits of coding over routing in a randomized setting. In Proceedings of IEEE ISIT, 2003.

[9] Li, S.-Y.R., Yueng, R.W., and Cai, N. 2003. Linear network coding. IEEE Transactions on Information Theory, vol. 49, 2003. 371-381.

[10] Krohn, M., FreedMan, M., and Mazieres, D. 2004. On-the-fly verification of rateless erasure codes for efficient content distribution. In Proceedings of IEEE Symposium on Security and Privacy, Berkeley, CA, 2004.

[11] Gkantsidis, C. and Rodriguez, P. 2005. Network coding for large scale content distribution. In Proceedings of IEEE INFOCOM 2005.

[12] Gkantsidis, C. and Rodriguez, P. 2006. Cooperative security for network coding file distribution. In Proceedings of IEEE INFOCOM'06, 2006.

[13] Li, Q., Chiu, D.-M., and Lui, J. C.S. 2006. On the practical and security issues of batch content distribution via network coding. In Proceedings of IEEE ICNP'06, 2006, 158-167.

[14] Wang, M. and Li, B. 2007. Lava: a reality check of network coding in peer-to-peer live streaming. In Proceedings of IEEE INFOCOM'07, 2007.

[15] Wang, M. and Li, B. 2007. R^2: random push with random network coding in live peer-to-peer streaming. In IEEE Journal on Selected Areas in Communications, Dec. 2007, 1655-1666.

[16] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of ACM PPoPP'08, Feb. 2008.

[17] Falcao, G., Sousa, L., and Silva, V. 2008. Massiv parallel LDPC decoding in GPU. In Proceedings of ACM PPoPP'08, Feb. 2008.

[18] Yu, Z., Wei, Y., Ramkumar, B., and Guan, Y. 2008. An efficient signature-based scheme for securing network coding against pollution attacks. In Proceedings of IEEE INFOCOM'08, Apr. 2008.

[19] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. 2008. GPU computing. IEEE Proceedings, May 2008, 879-899.

[20] Al-Kiswany, S., Gharaibeh, A., Santos-Neto, E., Yuan, G., and Ripeanu, M. 2008. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In Proceedings of IEEE Symposium on High Performance Distributed Computing (HPDC), Jun. 2008.

[21] Seiler, L., et. al., 2008. Larrabee: a many-core x86 architecture for visual computing. ACM Transactions on Graphics, 27(3), Aug. 2008.

[22] Chu, X., Zhao, K., and Wang, M. 2008. Massively parallel network coding on GPUs. In Proceedings of IEEE IPCCC'08, Dec. 2008.

[23] Shojania, H., Li, B., and Wang, X. 2009. Nuclei: GPU-accelerated Many-core Network Coding. In Proceedings of IEEE INFOCOM'09, Apr. 2009.

[24] Chu, X., Zhao, K., and Wang, M. Practical random linear network coding on GPUs. In Proceedings of IFIP Networking'09, May 2009.

[25] Zhao, K., Chu, X., Wang, M., and Jiang, Y. 2009. Speeding up homomorphic hashing using GPUs. In Proceedings of IEEE ICC'09, Jun. 2009.

[26] Chu, X. and Jiang, Y. 2010. Random linear network coding for peer-to-peer applications. IEEE Network, 24(4), July-August 2010, 35-39.

[27] OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl/

[28] Koç, Ç. K., Acar, T., and Kaliski, B. S. Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro 16, 3, Jun. 1996, 26-33.

[29] Brickell, E. F., Gordon, D. M., McCurley, K. S., and Wilson, D. B. 1992. Fast exponentiation with precomputation. In Proceedings of Advances of Cryptology: Eurocrypt'92.

Table 1. Homomorphic hash function parameters

NAME	DESCRIPTION	TYPICAL VALUE
λ_p	Discrete log security parameter	1024 bit
λ_q	Discrete log security parameter	257 bit
p	Random prime, $\mid p \mid = \lambda_p$	
q	Random prime, $\mid q \mid = \lambda_q$, $q \mid p-1$	
m	Number of codewords per data block	512
n	Number of data blocks	128
g	1 x m vector of order q	

Table 2. Parameters of GPUs used in our experiments

GPU NAME	**GTX295**	**GTX280**	**GEFORCE 9800**
Number of SPs	480	240	128
Processor clock	1.24 GHz	1.30 GHz	1.80 GHz[1]
Memory bandwidth	224 GB/s	141 GB/s	70 GB/s
Memory amount	1.79 GB	1 GB	512 MB

[1] This card is donated by NVIDIA and has a higher processor clock than commercial NVIDIA GeForce 9800 cards.

Table 3. Throughput of Homomorphic Hashing

PLATFORM	**METHOD**	**THROUGHPUT (Mbps)**
Quad-core CPU	[10]: Single thread	0.28
	[10]: Multi-threaded	1.03
GPU GTX280	[24]	1.90
GPU GeForce 9800	***Tsunami***	10.7
GPU GTX280		20.6
GPU GTX295		34.7