Practical Random Linear Network Coding on GPUs[*]

Xiaowen Chu[1], Kaiyong Zhao[1], and Mea Wang[2]

[1] Department of Computer Science, Hong Kong Baptist University, Hong Kong
{chxw,kyzhao}@comp.hkbu.edu.hk
[2] Department of Computer Science, University of Calgary, Alberta, Canada
meawang@ucalgary.ca

Abstract. Recently, random linear network coding has been widely applied in peer-to-peer network applications. Instead of sharing the raw data with each other, peers in the network produce and send encoded data to each other. As a result, the communication protocols have been greatly simplified, and the applications experience higher end-to-end throughput and better robustness to network churns. Since it is difficult to verify the integrity of the encoded data, such systems can suffer from the famous pollution attack, in which a malicious node can send bad encoded blocks that consist of bogus data. Consequently, the bogus data will be propagated into the whole network at an exponential rate. Homomorphic hash functions (HHFs) have been designed to defend systems from such pollution attacks, but with a new challenge: HHFs require that network coding must be performed in GF(q), where q is a very large prime number. This greatly increases the computational cost of network coding, in addition to the already computational expensive HHFs. This paper exploits the potential of the huge computing power of Graphic Processing Units (GPUs) to reduce the computational cost of network coding and homomorphic hashing. With our network coding and HHF implementation on GPU, we observed significant computational speedup in comparison with the best CPU implementation. This implementation can lead to a practical solution for defending the pollution attacks in distributed systems.

Keywords: applications and services, network coding, pollution attack, GPU computing.

1 Introduction

In recent years, peer-to-peer (P2P) content distribution applications (e.g., BitTorrent) and video streaming applications (e.g., ppLive) have become popular and constitute more than 30% of today's Internet traffic. The new coding technique, random linear network coding, is recently adopted by P2P applications [6-12], leading to simpler communication protocols, higher throughput, better resilience to network churns, and many more benefits to be discovered [6-12]. With network coding, the source

[*] This research was supported in part by Hong Kong RGC under grant HKBU 210406, and FRG grant HKBU FRG/07-08/II-36.

segments the to-be-distributed content into n data blocks of equal size. Each peer (including the source) sends out encoded data blocks, each of which is a linear combination of the original data blocks. After receiving n linearly independent encoded data blocks, a peer is able to decode the original data blocks by solving n linear equations with n variables. Since it is difficult to verify the integrity of the encoded data, such systems can suffer from the famous pollution attack, in which a malicious node can send bad encoded blocks that consist of bogus data. Consequently, the bogus data will be propagated into the whole network at an exponential rate. To defend against such attacks, homomorphic hash functions (HHFs) have been proposed to provide a mechanism for verifying the integrity of the encoded data blocks received from the network. In a nutshell, HHFs offer a nice property that the hash value of any encoded data block can be derived from the hash values of the original data blocks, based on which we can identify the bad encoded data blocks without decoding them [14] [15]. Hence, it can effectively prevent the propagation of the bogus data blocks.

The theoretical property of HHFs is very attractive to practical P2P applications. Unfortunately, the computational complexity posed by HHF is the stumbling stone to this realization. First, HHF itself is computationally expensive. On a contemporary CPU, say 3.0GHz Pentium 4 PC, we can only hash hundreds of kilobit per second [13]. Second, Homomorphic hashing requires extensive modular exponentiation operations over a very larger prime modulus p (e.g., 1024 bit). This require that the data must be encoded in large finite field, GF(q), where q is a large prime number (e.g., 257 bit). This greatly increases the computational cost of network coding, nearly impractical on CPUs. Our imperative goal is to remove the barrier by reducing the computational cost. The key enabling technologies here are the modern GPUs and the CUDA programming model for non-graphical application development on GPUs.

On the hardware level, recent advances in GPUs open a new era of GPU computing [20]. For instance, NVIDIA's GTX 280 can achieve 933 GFLOPS of computing power, about 8 times faster than the Intel Harpertown 3.2GHz CPU. However, using GPU for non-graphic applications has been considered very difficult, mostly due to the limited API support. Nonetheless, the introduction of CUDA programming model makes it easier for software developers to develop non-graphic applications on GPUs [1]. In CUDA, GPU is treated as a dedicated coprocessor to the CPU, and multiple threads based on the same code can run simultaneously on the GPU, working on different data set. With supports from CUDA, it is now possible to implement network coding and HHFs on GPUs. In this paper, we propose to use GPU to accelerate random linear network coding as well as homomorphic hashing. We designed and developed massively parallel network encoding and decoding algorithms and homomorphic hashing. By carefully applying optimization techniques, we successfully achieved a significant performance boost: 95x speedup for network encoding, 33x speedup for network decoding, and 15x speedup for homomorphic hashing on a contemporary GPU. This makes network coding and HHF a practical solution for pollution attacks in P2P systems.

The rest of the paper is organized as follows. Sec. 2 provides background information on network coding, homomorphic hashing, the GPU architecture, and the CUDA programming model. Sec. 3 presents the parallel algorithms for random linear network coding in GF(q). Sec. 4 presents the parallel homomorphic hash algorithm. Our experimental results are presented in Sec. 5, followed by the conclusions in Sec. 6.

2 Background and Related Work

This section provides the necessary background knowledge of network coding, homomorphic hash function, the GPU architecture, and the CUDA programming model.

2.1 Network Coding

Network coding has been originally proposed in information theory to achieve the optimal throughput in a multicast session [6]. Since then, it has been applied in various communication networks for better throughput and robustness to network dynamics. The essence of network coding is a paradigm shift to allow coding at intermediate nodes between the source and the receivers in one or multiple communication sessions. The seminal work of network coding has been studied in [6] [7] [9], which has shown that a multicast session can achieve the data rate of multicast upper bound if network nodes are allowed to perform coding. The framework of random network coding was proposed in [8], which makes network coding theory applicable to practical applications. Since then, there are quite a number of proposal to apply network coding in practical systems for performance enhancement. The Avalanche project by Microsoft Research applied random linear network coding in a P2P content distribution application [10]. Similarly, Lava incorporates random linear network coding into a live multimedia streaming system [12]. Network coding has also been applied in other fields, such as distributed storage systems [11] and wireless networks [21].

In network coding, each data block is treated as a vector of elements in the finite field, and an encoded block is simply a vector representing the linear combination of a set of data blocks (vectors) with randomly generated coefficients in the finite field. The network coding operations, encoding and decoding, are implemented in finite fields, i.e., prime fields GF(q) or extension fields GF(q^r), where q is a prime number and r is a positive integer. The computational performance of random linear network coding in GF(2^r) has been previously studied in [17]. In [18] [22], GPUs have been used to accelerate the performance of network coding in GF(2^r). To the best of our knowledge, this is the first paper studying the computational performance of random linear network coding in prime field GF(q), which has a much higher demand of computing power than that of network coding in GF(2^r).

2.2 Homomorphic Hashing

As discussed in Sec. 1, network coding enabled P2P applications are prone to the pollution attacks, in which a malicious peer can easily inject bogus data blocks into an encoded block without being noticed. When network coding is not deployed, peers will receive original data blocks from each other. Hence it is possible to use normal hash functions, such as SHA1, to verify the correctness of a data block by comparing the hash of each received data block to the corresponding hash provided by the source. With network coding, the effect of pollution attack becomes more serious and harder to detect [14] [15] [16] for two reasons: First, each bogus block can be encoded with regular data blocks before being propagated in the network. Second, the traditional hash functions, e.g., SHA1, are no longer practical since the encoded blocks received by each peer cannot be predetermined by the source. Some workarounds have been proposed to address these issues. In [14], a cooperative scheme is proposed, in which peers perform probabilistically block verification and inform others when a malicious node has been identified. However, this scheme cannot detect the bogus blocks at the earliest stage and could potentially have false alarms.

To this end, homomorphic hash functions (HHFs) are currently the best solution to address this security issue with network coding. HHFs have the property that the hash value of an encoded block can be constructed by the hash values of the original blocks. In other words, a peer only need to get the hash values of the original blocks from the source, it then can easily verify the integrity of an encoded block immediately after receiving the encoded block. Although the HHFs can theoretically resolve the pollution attack problem, it is technically not practical on today's desktop CPUs. A 3 GHz Pentium 4 CPU can only achieve around 300 Kbps of hashing throughput [13]. Furthermore, it requires network coding to be performed in GF(q) with large value for q, which makes it computationally expensive.

2.3 GPU Computing and CUDA

GPUs are dedicated hardware for manipulating computer graphics. Due to the huge demand for computing for real-time applications and high-definition 3D graphics, GPUs have been evolved into highly paralleled multi-core processors. The NVIDIA GeForce GTX260 has 24 Streaming Multiprocessors (SMs), and each SM has 8 Scalar Processors (SPs). At any given clock cycle, all SPs of the same SM must execute the same instruction, but can operate on different data. Each SM has four different types of on-chip memory: constant cache, texture cache, registers, and shared memory. The properties of the different types of memories have been summarized in [1] [19]. A general optimization principle is that registers and shared memory should be carefully utilized to amortize the global memory latency cost.

The exceptional GPU computing power is very attractive to general-purpose system development. The first generation of GPU computing (namely GPGPU) requires that non-graphics application must be mapped through the graphics application programming interfaces, which is very challenging. In early 2007, one of the major GPU vendors, NVIDIA, announced a new general-purpose parallel programming model, Compute Unified Device Architecture (CUDA) [1], which extends the C programming language for general-purpose application development. Meanwhile, another GPU vendor AMD introduced Close To Metal (CTM) programming model that provides an assembly language for application development [2]. Intel is also planning to release Larrabee [3], a new multi-core GPU architecture specially designed for GPU computing. 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 [18-20] [22] [23]. For these reasons, we chose to use CUDA in our research. Nevertheless, out algorithms can be easily implemented on other GPU computing models.

In the CUDA model, the GPU is regarded as a coprocessor capable of executing a great number of threads in parallel. A single program consists of *host code* to be executed on CPU and *kernel code* to be executed on GPU. The kernel code is usually computational-intensive, data-parallel and multi-threaded. Threads are organized into *thread blocks*, where each block is associated with one SM. Threads belonging to the same thread block can share data through the shared memory and can perform barrier synchronization. CUDA does not provide any direct synchronization methods between threads that belong to different thread blocks, however. When a thread block terminates, a new thread block can be launched on the vacant SM.

3 Parallel Network Coding on GPUS

To facilitate the development of network coding, we have implemented a set of library functions of multiple-precision modular arithmetic on the CUDA platform. These library functions simplify the development of the network coding system and homomorphic hash functions. Our multiple-precision library includes the following functions: comparison, subtraction, modular addition, modular subtraction, multiplication, division, multiplicative inversion, Montgomery reduction, Montgomery multiplication.

Assume the original data to be distributed is divided into n equally sized data blocks ($b_1, b_2, \ldots, b_n$), where each data block b_i contains 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 also contains 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* $(c_{j,1}, c_{j,2}, \ldots, c_{j,n})$: $e_{j,k} = \sum_{i=1}^{n} c_{j,i} \cdot b_{i,k}$, $k\in \{1,\ldots,m\}$. Obviously the encoding process is a vector-matrix multiplication. A peer can decode the original n data blocks as soon as it has received n linearly independent encoded data blocks ($e_1, e_2, \ldots, e_n$), by solving the set of linear equations $e_{j,k} = \sum_{i=1}^{n} c_{j,i} \cdot b_{i,k}$, $k\in \{1,\ldots,m\}$, $j\in \{1,\ldots,n\}$. In a P2P application with network coding, 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.

3.1 Network Encoding in GF(q)

When network coding is performed in GF(q) where q is a predefined large prime number, the encoding process will creates a sequence of encoded blocks e_j, each of which contains m codewords $e_{j,k}$. e_j is generated based on a random coefficient vector $c_j = (c_{j,1}, c_{j,2}, \ldots, c_{j,n})$: $e_{j,k} = \sum_{i=1}^{n} c_{j,i} \cdot b_{i,k} \bmod q$, $k\in \{1,\ldots,m\}$. Here $c_{j,i}$ are positive 32-bit integers. The encoding process includes two steps: (1) generating the coefficient vector c_j; (2) modular vector-matrix multiplication. The CUDA library provides a very high efficient random number generator using Mersenne Twister method, which can generate tens of millions of random numbers per second using GPU. As the time of generating n random numbers are negligible as compared with the encoding time, we will focus on the vector-matrix multiplication operation, as shown in Figure 1(a). We implement the computing of each codeword $e_{j,k}$ by a CUDA thread. Hence encoding a single block requires m threads. Each thread computes a dot product and then performs a modular operation, using our multiple-precision CUDA library. In order to fully exploit the computing power of GPUs, thousands of threads are a normal requirement. Therefore we propose a batched encoding approach for small values of m, which encodes K blocks simultaneously, as shown in Figure 1(b). In this case, the number of threads equals mK.

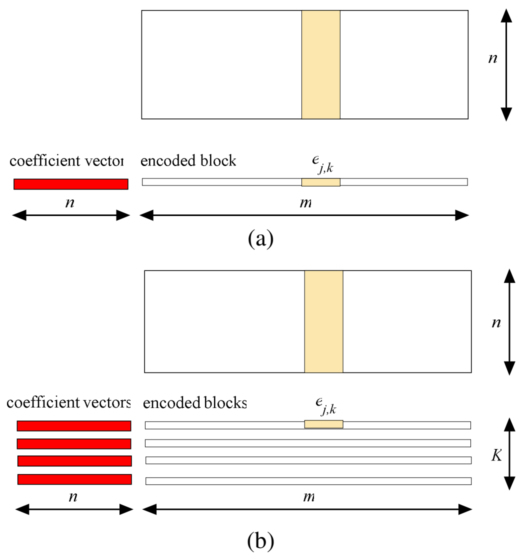


Fig. 1. Network encoding: (a) Encode a single block (b) Encode multiple blocks in a batch

3.2 Network Decoding in GF(q)

The decoding process includes two steps: (1) matrix inversion; and (2) matrix multiplication. Matrix inversion for floating-point numbers on GPU has been recently studied in [23]. Our problem is very different because we are operating in GF(q). We use Gauss-Jordan elimination for the matrix inversion, which brings a matrix to its reduced row echelon form. There is no stability issue because we are operating in finite field. To overcome the synchronization challenge, our parallel matrix inversion algorithm uses both CPU and GPU, as shown in Table 1. The non-parallel parts, e.g., finding the multiplicative inverse, are done by the CPU, whilst the parallel parts, e.g., reducing to row echelon form, are done by GPU.

The matrix multiplication can be implemented in a similar way as the vector-matrix multiplication. Each element in the output matrix is computed by one thread; hence the total number of threads is n^2, which is sufficient to fill the GPU cores since n is normally no less than 64 in practice. Difference from the encoding process, the integer multiplications here are performed between two large integers, since the corresponding values of the coefficients grow as the blocks are being re-encoded at each peer. In this case, a straightforward parallel implementation cannot achieve satisfactory performance due to the global memory latency. GPU's on-chip shared memory can be exploited to amortize the global memory latency, and we propose to use a titled version of matrix multiplication, in which the matrix is divided into a number of sub-blocks [1] [19]. As illustrated in Figure 2, the computing of sub-block B_{sub} is done by a thread block. The threads in this block cooperatively load the data from the two tiles in coefficient matrix and E^T into shared memory. These threads compute the partial dot product in shared memory, and then continue with the next tile. The size of the tile should be controlled such that two tiles can be accommodated by the shared memory of a SM.

Table 1. Algorithm of Matrix Inversion in GF(q)

```
Algorithm 1. Matrix Inversion in GF($q$)
INPUT: An $n$ x $n$ non-singular matrix $M$, an $n$ x $n$ unit matrix $U$
OUTPUT: the inverse of $M$
1:  $lead \leftarrow 0$;
2:  $row \leftarrow n$; $col \leftarrow n$;
3:  for ( $r$ = 0 to $row$ - 1)
4:     $i \leftarrow r$;
5:     while $M[i, lead]$ equals 0
6:        $i$++;
7:     Swap rows $i$ and $r$ of $M$ and $U$;
8:     $t \leftarrow$ multiplicative inverse of $M[r, r]$;  /* on CPU */
9:     Multiply row $r$ of $M$ and $U$ by $t$;           /* on GPU */
10:    for all rows $j$ except row $r$ of $M$ and $U$
11:       For $M$, subtract $M[j, lead]$ multiplied by row $r$ from row $j$; /* on GPU */
12:       For $U$, subtract $M[j, lead]$ multiplied by row $r$ from row $j$; /* on GPU */
13:    end for
14:    $lead$++;
15: end for
16: return $U$
```

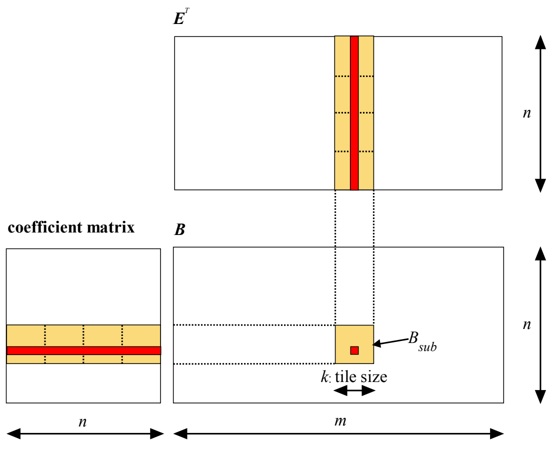


Fig. 2. Decoding: tiled matrix multiplication

4 Parallel Homomorphic Hashing on GPUS

The homomorphic hash function, $h(\cdot)$, proposed in [13] requires a set of hash parameters $G = (p, q, g)$. 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, each of which can be written as $x^{(p-1)/q}$ mod p where $x \in Z_q$ and $x \neq 1$. The method of creating the parameter set can be found in [13]. Typical values of the parameters are summarized in Table 2. The homomorphic hash of a data block b_i is then calculated as

$h(b_i)=\prod_{k=1}^{m} g_k^{b_{k,i}}$ mod p. The hash values of the original data blocks $(b_1, b_2, \ldots, b_n)$ are $h(b_1)$, $h(b_2)$, …, $h(b_n)$, respectively. Given an encoded data 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 shown to satisfy the following condition: $h(e_j)=\prod_{i=1}^{n} h^{c_{j,i}}(b_i)$ mod p. This property can be used to verify the integrity of an encoded block, as illustrated in Figure 3. The content publisher first calculates the homomorphic hash values for each of the data blocks. The downloaders need to download a copy of these hash values for the purpose of verifying every single encoded data block.

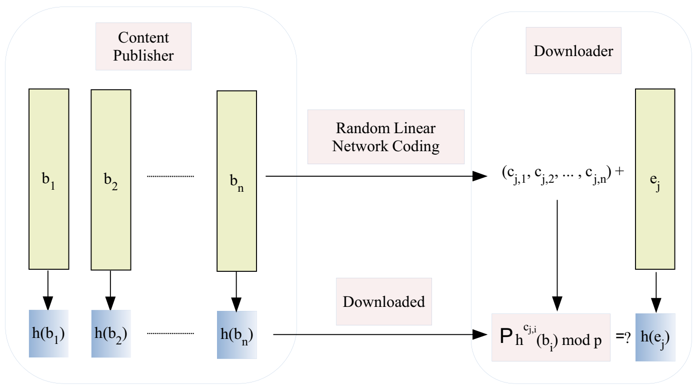

Content Publisher
b_1
b_2
b_n
$h(b_1)$
$h(b_2)$
$h(b_n)$
Random Linear Network Coding
Downloaded
Downloader
$(c_{j,1}, c_{j,2}, \ldots, c_{j,n})$ +
e_j
$\prod h^{c_{j,i}}(b_i)$ mod p
=?
$h(e_j)$

Fig. 3. Data verification using homomorphic hashing in network coded P2P applications

Table 2. 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, q	Random primes, $\lvert p\rvert=\lambda_p$, $\lvert q\rvert=\lambda_q$, $q \mid p-1$	
m	Number of codewords per data block	512
n	Number of data blocks	128

As shown earlier, homomorphic hashing, i.e., $h(b_i)=\prod_{k=1}^{m} g_k^{b_{k,i}}$ mod p, involves m modular exponentiations and m-1 modular multiplications. The m-1 modular multiplications can be easily parallelized by a regular reduction process. The m modular exponentiations can be very time consuming. We distribute the m modular exponentiations to the GPU processing cores. The challenge is to implement modular exponentiation on GPU in the most efficient way. On the current CUDA platform, integer division and modulo operations are very costly. Therefore we choose to use the

Montgomery exponentiation algorithm (as shown in Table 3) which can decrease the number of division operations significantly.

Table 3. Algorithm of multiple-precision Montgomery exponentiation

Algorithm 2. Multiple-precision Montgomery Exponentiation

INPUT: integer m with n radix b digits and gcd(m, b) = 1, $R = b^n$, positive integer x with n radix b digits and $x < m$, and positive integer e = $(e_t \cdots e_0)_2$.

OUTPUT: x^e mod m.

```
1:  $\tilde{x} \leftarrow Mont(x, R^2\ mod\ m)$;
2:  $A \leftarrow R$ mod $m$;
3:  for ( $i$ from $n$ down to 0)
4:      $A \leftarrow Mont(A, A)$;
5:      if $e_i == 1$
6:      then $A \leftarrow Mont(A, \tilde{x})$;
7:  end for
8:  $A \leftarrow Mont(A, 1)$;
9:  return $A$;
```

5 Experimental Results

We have implemented the proposed network encoding/decoding and parallel homomorphic hashing algorithm using CUDA. For comparison purpose, we also implemented network coding and homomorphic hash function for CPU in C language, by utilizing the GNU MP arithmetic library, version 4.2.3 [4]. These implementations are running on an Intel Core2 CPU 1.6 GHz. We tested all our algorithms on XFX GTX280 graphic card with an NVIDIA GeForce GTX280, which has 240 processing cores. On GTX280, there are 30 Streaming Multiprocessors (SMs), and each SM has 8 Scalar Processors (SPs), 16384 32-bit registers and 16KB shared memory.

5.1 Performance of Encoding in GF(q)

The throughput of encoding process is shown in Figure 4(a) in log-scale. In theory, the encoding time complexity is linear to the size of n. This is in accordance with our experimental results. The throughput of network encoding on CPU is very poor: only 10.3 Mbps for n = 128. The GPU performance is very impressive: around 800 Mbps can be achieved for n = 128 with a small batch size K of 32. We observe that larger batch sizes can lead to better performance, until some threshold value has been met. In our testing environment, K = 128 is the optimal setting. The speedup of GPU over CPU has been plotted in Figure 4(b), for different batch sizes and n. With a batch size larger than 32, the speedup is greater than 66x. The highest speedup of 95x is obtained when n = 128 and K = 128.

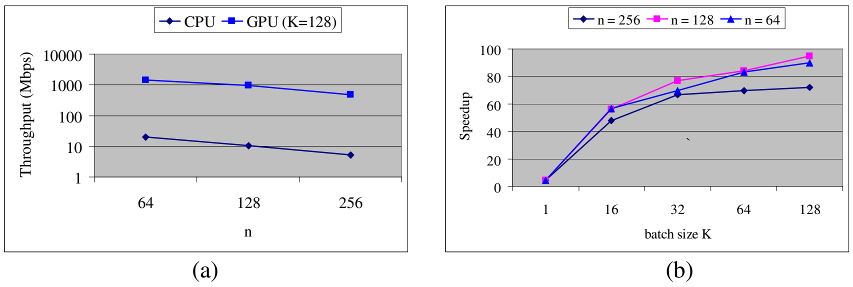


Fig. 4. Performance of network encoding: (a) Encoding throughput (b) Speedup over CPU

5.2 Performance of Decoding

As mentioned before, the decoding process includes two steps: (1) matrix inversion; (2) matrix multiplication. The time used for matrix inversion is shown in Figure 5(a) using log-scale, and the speedups on GPU over CPU are plotted in Figure 5(b). It is a well known fact that matrix inverse using Gauss-Jordan elimination has a time complexity of $O(n^3)$. Our experimental results on CPU follow this pattern as well. The performance of parallel matrix inversion on GPU is a bit more complicated due to the kernel loading overhead and communication overhead between the CPU and GPU. Figure 5(b) shows that the speedup on GPU grows as the number of blocks, n, increase: 17 for n = 128 and 24 for n = 256. This is where the benefit of GPU manifests itself, i.e., the overheads are amortized by the increasing parallelism in the computation.

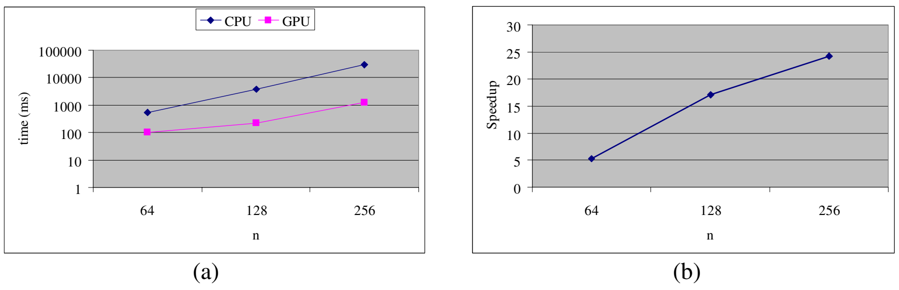
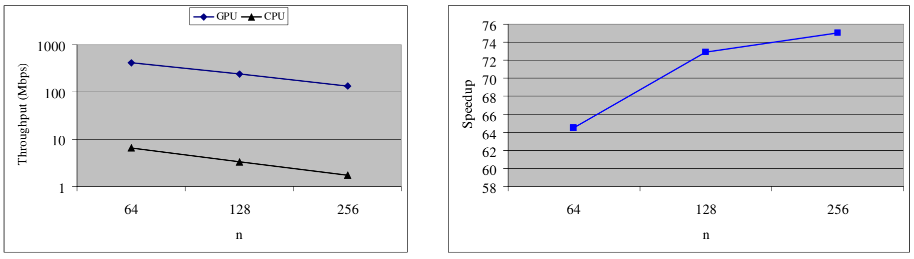


Fig. 5. Performance of matrix inversion: (a) Matrix inversion time in *ms* (b) Speedup over CPU

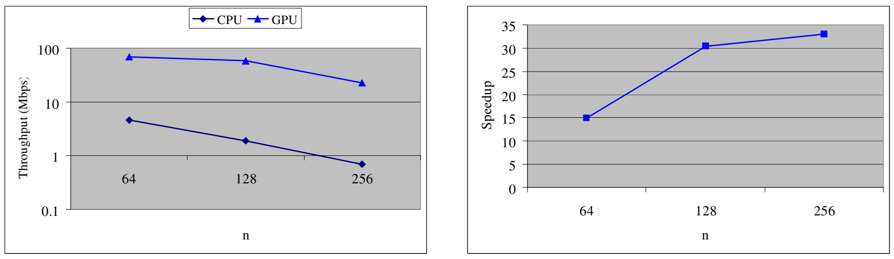
The performance of the matrix multiplication process is shown in Figure 6. As expected, the throughput is much slower than the encoding process. Even so, the GPU can achieve 243 Mbps of throughput for n = 128 when shared memory is utilized. The speedup on GPU over CPU ranges from 64x to 75x for n = 64, 128, 256 respectively when shared memory is used.

The performance of the whole decoding process is shown in Figure 7, for m = 512. Since the speedup of matrix multiplication is much larger than the speedup of matrix inversion, the speedup of the overall decoding process is limited by the performance of matrix inversion. The overall decoding throughput when n = 128 is 58 Mbps which includes the matrix inversion and matrix multiplication. The decoding performance can be further enhanced by using a larger value of m, because the same matrix inverse operation is now used for a larger data volume. The speedup ranges from 15x to 33x for different values of n.



(a) (b)

Fig. 6. Performance of matrix multiplication: (a) Throughput (b) Speedup over CPU



(a) (b)

Fig. 7. Performance of network decoding. (a) Throughput of decoding (b) Speedup over CPU.

5.3 Performance of Homomorphic Hashing

Our CPU version of homomorphic hashing achieves 130 Kbps of throughput, which is relatively lower than the results reported by [13] [14] due to our relatively lower CPU frequency.

The parallel homomorphic hashing uses Algorithm 2 to calculate exponentiations. The CUDA architecture requires a large number of threads to hide the memory latency and to fully utilize the computing power. The number of threads per thread block (denoted by ***TB***), and also the number of thread blocks (denoted by ***NB***), are the two main factors that affect the hashing throughput. We plot the throughput for different configurations in Figure 8. It is easy to observe that more threads per block can generally achieve better throughput. When the number of threads per block is fixed, the throughput can be improved by creating more thread blocks, until some threshold has been reached. Since our GPU has 30 SMs, the number of thread blocks should be a multiple of 30. Better throughput can be achieved if the following conditions are satisfied: (1) ***TB*** is a multiple of 32 (i.e., the *warp* size [1] [19]). In CUDA, a *warp* is formed by 32 parallel threads and is the scheduling unit of each SM. If the number of threads in a block is not a multiple of warp size, the remaining instruction cycles will be wasted. (2) ***NB*** is a multiple of 30 (i.e., the number of SMs); (3) ***TB*** x ***NB*** > 6144.

For example, if $m = 512$, we should perform the homomorphic hashing for 12 different data blocks simultaneously. The highest throughput of 1.9 Mbps is obtained when ***TB*** = 128 and ***NB*** = 90, which is 15 times faster than the CPU implementation.

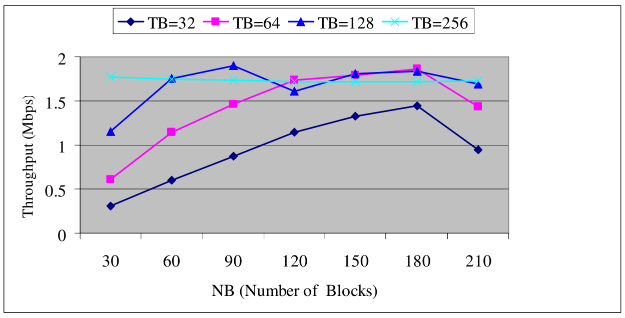


Fig. 8. Throughput of homomorphic hashing on GPU

6 Conclusions

Network coding has been shown as a powerful technique to enhance the throughput and robustness of P2P systems; and homomorphic hash functions are a supplementary tool for defending the pollution attack. The remaining challenges are the computational requirement of network coding in prime field and the homomorphic hashing. This paper demonstrates a practical parallel implementation of network coding and homomorphic hashing using GPUs. Our experimental results show that the computational obstacle of network coding and homomorphic hashing can be overcome by designing efficient parallel algorithms and fully exploiting the computing power of contemporary GPUs that are widely available on today's desktop PCs.

References

1. NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 2.0beta2 (June 2008)
2. AMD CTM Guide: Technical Reference Manual (2006), `http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf`
3. Seiler, L., et al.: Larrabee: a many-core x86 architecture for visual computing. ACM Transactions on Graphics 27(3) (August 2008)
4. GNU MP Arithmetic Library, `http://gmplib.org/`
5. Brickell, E.F., Gordon, D.M., McCurley, K.S., Wilson, D.B.: Fast exponentiation with precomputation: algorithms and lower bound. In: Rueppel, R.A. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 200–207. Springer, Heidelberg (1993)
6. Ahlswede, R., Cai, N., Li, S.R., Yeung, R.W.: Network information flow. IEEE Transactions on Information Theory 46(4), 1204–1216 (2000)
7. Koetter, R., Medard, M.: An algebraic approach to network coding. IEEE/ACM Transactions on Networking 11(5), 782–795 (2003)

8. Ho, T., Koetter. R., Médard, M., Karger, D.R., Effros, M.: The benefits of coding over routing in a randomized setting. In: Proceedings of IEEE ISIT (2003)
9. Li, S.-Y.R., Yueng, R.W., Cai, N.: Linear network coding. IEEE Transactions on Information Theory 49, 371–381 (2003)
10. Gkantsidis, C., Rodriguez, P.: Network coding for large scale content distribution. In: Proceedings of IEEE INFOCOM 2005 (2005)
11. Dimakis, A.G., Godfrey, P.B., Wainwright, M.J., Ramchandran, K.: Network coding for distributed storage systems. In: Proceedings of IEEE INFOCOM 2007 (2007)
12. Wang, M., Li, B.: Lava: a reality check of network coding in peer-to-peer live streaming. In: Proceedings of IEEE INFOCOM 2007 (2007)
13. Krohn, M., FreedMan, M., Mazieres, D.: On-the-fly verification of rateless erasure codes for efficient content distribution. In: Proceedings of IEEE Symposium on Security and Privacy, Berkeley, CA (2004)
14. Gkantsidis, C., Rodriguez, P.: Cooperative security for network coding file distribution. In: Proceedings of IEEE INFOCOM 2006 (2006)
15. Li, Q., Chiu, D.-M., Lui, J.C.S.: On the practical and security issues of batch content distribution via network coding. In: Proceedings of IEEE ICNP 2006, pp. 158–167 (2006)
16. Yu, Z., Wei, Y., Ramkumar, B., Guan, Y.: An efficient signature-based scheme for securing network coding against pollution attacks. In: Proceedings of IEEE INFOCOM 2008 (April 2008)
17. Shojania, H., Li, B.: Parallelized progressive network coding with hardware acceleration. In: Proceedings of the 15th International Workshop on Quality of Service, IWQoS (2007)
18. Chu, X., Zhao, K., Wang, M.: Massively parallel network coding on GPUs. In: Proceedings of the 27th IEEE IPCCC (December 2008)
19. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of ACM PPoPP 2008 (Feburary 2008)
20. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. In: IEEE Proceedings, May 2008, pp. 879–899 (2008)
21. Katti, S., Katabi, D., Balakrishna, H., Medard, M.: Symbol-level network coding for wireless mesh networks. In: Proceedings of ACM Sigcomm 2008 (August 2008)
22. Shojania, H., Li, B., Wang, X.: Nuclei: GPU-accelerated Many-core Network Coding. In: Proceedings of IEEE INFOCOM 2009 (April 2009)
23. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Dolev, S., Haist, T., Oltean, M. (eds.) OSC 2008. LNCS, vol. 5172. Springer, Heidelberg (2008)