Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/092L32

Massive Parallelization for Random Linear Network Coding

Seong-min Choi[1], *Kyogu Lee*[2] *and Joon-Sang Park*[1,*]

[1] Dept. of Computer Engineering, Hongik University, Seoul, Korea
[2] Dept. of Transdisciplinary Studies, Seoul National University, Seoul, Korea

Received: 14 Jun. 2014, Revised: 16 Aug. 2014, Accepted: 18 Aug. 2014
Published online: 1 Apr. 2015

Abstract: In this paper, we propose a general-purpose graphics processing unit (GPGPU) based parallelization technique for random linear network coding (RLNC). RLNC is recognized as a useful tool for enhancing performance of networked systems, and several parallel implementation techniques have been proposed in the literature to overcome its high computation overhead. However, existing parallel methods cannot take full advantage of GPGPU technology on many occasions. Addressing this problem, we propose a new RLNC parallelization technique that can exploit GPGPU architectures in full. Our method exhibits as much as a 5x increase in throughput compared to existing parallel RLNC decoding schemes leveraging GPGPU.

Keywords: Network Coding, Parallel algorithm, GPGPU

1 Introduction

Network coding [1] in which intermediate nodes (or routers) perform coding operations on packet contents has gained popularity as a useful tool for enhancing networked system performance. There are a number of code construction methodologies for network coding such as random (linear) network coding [2] and XOR code [3], with random network coding being one of the most widely adopted, owing to its applicability to various real-world scenarios. It is known that random network coding achieves multicast capacity in wired networks[1], reduces bandwidth requirements for data replications in distributed storage systems for big data systems [4], and reduces file-downloading latency in peer-to-peer (P2P) file sharing systems [5]. In typical P2P file sharing systems, a file is partitioned into multiple pieces, which can be exchanged among peers independently, and downloading a complete file involves collecting all the pieces belonging to the file. In such systems, downloading delay can be dramatically reduced, if multiple pieces can be simultaneously downloaded from multiple peers; however, communicating peers must be selected carefully, because the file download latency can be very large, if improper choices are made. In contrast, when network coding is used, the pieces are encoded into coded blocks such that all the coded blocks are equally important and indistinguishable, requiring the collecting node to gather a specific number of equally important coded blocks from other peers. This eliminates the peer/piece selection problem in normal P2P systems and reduces file download latency. There are also many other advantages of using random network coding in networked systems; however, its computational overhead may hamper the use of random network coding in practice. When random network coding is used, data must be encoded before being transmitted at the sending node, and data received at a destination must be decoded for recovery of the original data. The decoding process of random network coding is implemented as a variation of Gaussian elimination. Since the complexity of Gaussian elimination, $O(n^3)$, where n is the number of blocks comprising a file, is quite high with larger file sizes, the time overhead spent for decoding would eliminate all the benefits of the reduced transmission time obtained from using random network coding. Thus, it is critical to assure short decoding latency when implementing random network coding in practice.

A number of studies have investigated reducing the decoding latency of random network coding. Parallelized decoding techniques for multi-core processors have been proposed in [6,7]. In addition, it has been shown that

* Corresponding author e-mail: jsp@hongik.ac.kr

parallel decoding using General Purpose Graphics Processing Unit (GPGPU) technology such as NVIDIAs Compute Unified Device Architecture (CUDA) [8] can dramatically reduce the decoding latency in random network coding [9,10,11,12,13] and also a variant of random network coding called pipeline network coding [14]. In this paper, we propose a parallel implementation technique for random linear network coding leveraging GPGPU technology. Recently, GPGPU technology has paved the way for parallelizing random network coding; however, existing parallelization techniques for random linear network coding cannot utilize GPGPU technology fully in many cases. We propose a new random network coding parallelization technique that can fully exploit GPGPU architectures. Our parallel method exhibits as much as a 5x increase in throughput compared to existing state-of-the-art parallel implementations for random network coding leveraging GPGPU.

The remainder of this paper is organized as follows. Section 2 gives an overview of random network coding and discusses related work. Section 3 details our proposed parallelized random network coding scheme. Section 4 shows the performance advantage of our proposed scheme. We conclude in Section 5.

2 Background

In this section, we first give an overview of random linear network coding and then discuss related work.

Random Linear Network Coding

Network coding involves performing coding operations throughout a network. In conventional networks, an outgoing packet on intermediate nodes is a copy of an input packet, whereas in network coding, a packet on the output links of an intermediate node is a function (or a combination) of input packets. There are multiple ways to combine packets (or code construction methodologies), such as random linear network coding and XOR code, with random linear network coding being one of the most widely adopted schemes, owing to its applicability to various real-world scenarios, and we focus on the random linear network coding in this paper. To transfer a set of data such as a single file using random linear network coding, the source node generates a set of coded packets from the original file and transmits them towards destination nodes. To this end, the data at the source are first divided into a number of blocks. We use $\mathbf{p}_k$ to denote k^{th} block. A coded packet $\mathbf{c}_i$ is a linear combination of the original blocks. That is $\mathbf{c}_i = \sum_{k=1}^{G} e_{ik}\mathbf{p}_k$, where G is the number of blocks, and the coefficient e_{ik} is an element randomly chosen in a finite field F [2]. The coded packet $\mathbf{c}_i$ is transmitted to destination nodes along with a coefficient vector $\mathbf{e}_i = [e_{i1}, \ldots, e_{iG}]$ describing how the coded packet is constructed and stored in the header [16]. We refer to the concatenation of a coded packet and its coefficient vector as a transfer unit. On reception of coded packets, intermediate nodes on the path to a destination generate a linear combination of received coded packets and send them to downstream nodes. For a destination to be able to decode a set of received coded packets and recover the original file, it needs to collect at least G coded packets with linearly independent coefficient vectors. Suppose that a destination has collected n coded packets, $\mathbf{c}_1, \ldots, \mathbf{c}_g$, and let $\mathbf{E}^T=[\mathbf{e}_1^T \ \ldots \mathbf{e}_G^T]$, $\mathbf{C}^T=[\mathbf{c}_1^T \ \ldots \mathbf{c}_G^T]$, where superscript T denotes the transpose operation. Since the relationship among $\mathbf{C}$, $\mathbf{E}$, and $\mathbf{P}$ can be expressed as $\mathbf{C}=\mathbf{EP}$, the destination can recover the original file $\mathbf{P}$ by multiplying the inverse of $\mathbf{E}$ by $\mathbf{C}$, assuming that $\mathbf{E}$ is invertible, i.e., all the coefficient vectors $\mathbf{e}_k$ are linearly independent. In random linear network coding, a Gaussian elimination variant called progressive decoding [6] is widely used to calculate $\mathbf{P} = \mathbf{E}^{-1}\mathbf{C}$. Conventional Gaussian elimination requires collecting G transfer units and having the $G \times G$ coefficient matrix before beginning. However, waiting for the entire matrix to be formed is not optimal in random linear network coding. In fact, packets are delivered one by one, and the time gap between the arrival of the first transfer unit and the last can be very long. Thus, instead of waiting for all the transfer units to arrive, partial decoding can be performed on reception of each transfer unit. The received transfer units containing coefficient vectors and coded data blocks are organized as an augmented matrix in which a transfer unit constitutes a row such that the progressive decoding/Gaussian elimination can be run on the matrix. On each transfer unit's arrival, a new row is inserted into the augmented matrix, and then the procedure is applied to the matrix in order to obtain its reduced row-echelon form at the end.

Related Works

Network coding technique was originally proposed by Ahlswede et al. [1] as a capacity-achieving scheme for multicast connections in wireline networks. Following their work, Ho et al. [2] showed that a random construction of linear codes, i.e., random (linear) network coding, was sufficient to achieve multicast capacity in wireline networks. Chou et al. [16] proposed a practical way of implementing random network coding: network codes are carried along with packets. Dimakis et al. [4] showed that random network coding reduces maintenance bandwidth, i.e., bandwidth requirement for data replications, in distributed storage systems which is essential in big data systems. Gkantisidis et al. [5] have shown that random network coding is beneficial in large-scale P2P systems. Random network coding has been known as a helpful technique also for mobile P2P systems [17]. As mentioned previously, one of the main problems in random network coding is its high encoding/decoding latency and several approaches have been proposed to mitigate the problem. In [15], a variant of random network coding called Pipeline Network

Coding (PNC) has been proposed. PNC reduces encoding/decoding delay by using a special form of encoding/coefficient matrix. A new encoding scheme with a lower computational complexity than that of conventional random network coding was proposed in [18]. Shojania et al. [6] suggested a parallelized decoding technique for multi-core CPUs with SIMD (Single Instruction Multiple Data) instructions, e.g., Intel's Streaming SIMD Extensions (SSE). The decoding process in [6] is based on Gaussian elimination, but it progressively decodes data on arrival of each partial data block. This distinctive feature, progressive decoding, reduces overall decoding latency when the arrivals of partial data blocks to be decoded span a long period of time. Note, however, that the original data can be recovered and the decoding process can be completed only at the arrival of the final data, even though a method such as progressive decoding is used. Conventional parallelized Gaussian elimination algorithms, such as the parallel adaptive Gauss-Jordan algorithm [19] and other related algorithms, such as parallel matrix inversion [20] and parallel LU decomposition [21], require the entire data set before starting the decoding process and thus incur additional decoding latency on the receiver compared to progressive decoding. Park et al. [7] have also proposed an efficient parallelized progressive random network coding algorithm with dynamic partitioning algorithms for multi-core CPUs and Kim et al. [22] proposed a parallel algorithm targeting specifically for *Cell* heterogeneous multi-core processor architecture. Shojania et al. proposed a parallelized progressive decoding algorithm for GPGPU [9] and a parallel multi-segment decoding algorithm for buffered data [10]. In these GPGPU-based parallelized schemes, hundreds of GPU threads encode and decode data blocks simultaneously and thus easily outperform parallel schemes for multi-core CPUs. In [11], Lee et al. proposed an optimization of the GPGPU implementations for handling multiple simultaneous streams/downloads. Park et al. proposed a GPGPU-based PNC implementation [14]. Chu et al. showed improved GPGPU encoding throughput with an aid of CPU [12], and Kim et al. further enhanced GPGPU decoding performance by applying optimized memory access patterns [13]; however, the decoding methodologies used in [12] and [13] are not classified as progressive decoding. Finally, Kim et al. [23] discussed random network coding implementations on programmable hardware such as FPGA (Field Programmable Gate Arrays.)

3 Maximizing Parallelism in Random Linear Network Coding

In this section, we present our parallelization technique for random linear network coding after giving a brief overview of GPGPU technology.

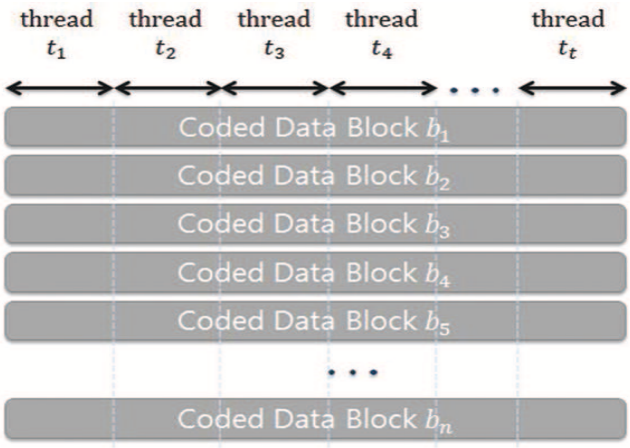


Fig. 1: Data partitioning in baseline method

GPGPU refers to the technology enabling general-purpose computation on GPU/graphics hardware. NVIDIA's CUDA [8] is one of the first programming models that provide general-purpose programmability on GPUs. A GPGPU device, in general, is composed of an array of computation units designed to execute arithmetic operations. The CUDA computing architecture consists of a group of streaming multiprocessors (SMs/SMXs), each of which contains of a number of scalar processors (SPs) called CUDA cores. In NVIDIA's GeForce GTX 670, which is based on Kepler, the GK104 architecture, and Compute Compatibility 3.0, there exist seven SMs each of which contains 192 CUDA cores and on-chip/local (or shared) memory. The shared memory can be shared only by the CUDA cores belonging to a specific SM. The global memory can be used for communication between CUDA cores belonging to different SMs and the host processor/CPU. In a CUDA program, parallelism is achieved through a simultaneous run of multiple GPU threads. In fact, hundreds (or over a thousand) such threads can be executed simultaneously in a CUDA GPGPU device. GPU threads are grouped into thread blocks (TBs), each of which is assigned to an SM(X). Synchronization and data sharing are allowed only for threads in the same TB. If there are many TBs, more than one TB can be assigned to one SM. For efficient thread processing, every 32 GPU instructions in a TB are grouped into a warp, a basic scheduling unit in the CUDA model. A warp is a set of instances of an instruction in each of 32 GPU threads, executed on a set of CUDA cores during four or two clock cycles, depending on the CUDA hardware. This model is referred to as single instruction, multiple thread (SIMT). The SIMT unit of an SM selects a warp ready to be executed and issues instructions to the active threads of the warp in out-of-order fashion. If a warp is stalled for some reason (e.g., memory access) the SIMT unit switches immediately to another warp. Therefore, creating a

sufficiently large number (e.g., hundreds or over a thousand) of threads is essential for maintaining maximum utilization of parallel hardware and/or to hide memory access latency. CUDA architectures, in general, have large memory access latency compared to modern CPU architectures with a deep memory hierarchy; thus it is crucial for CUDA programs to have a component hiding such large memory access latency. One common method is to generate a much larger number of concurrent threads than the maximum number of threads concurrently executable in a CUDA device such that there always exist ready-to-be-issued threads, while some threads are stalled for accessing memory.

The decoding process is usually implemented as a Gaussian elimination variant, as mentioned previously. Following the notation used in the previous section, the problem of decoding consists in solving and obtaining $\mathbf{P} = \mathbf{E}^{-1}\mathbf{C}$, where $\mathbf{E}$ and $\mathbf{C}$ are given. Usually $\mathbf{C}$ is an $m \times n$ matrix, where $n > m$. To parallelize such a decoding process, encoded data, i.e., the matrix $\mathbf{C}$, are partitioned column-wise into units (as shown in Figure 1), with each unit assigned a GPU thread through which it can be decoded independently. We refer to this scheme as the baseline parallel decoding scheme. All existing GPGPU-based parallel decoding schemes [9,10,11,12,13] use this approach. The main problem of this baseline parallel scheme is that on many occasions it cannot fully exploit opportunities for parallelism offered by modern GPGPUs. That is, the baseline scheme is unable to create a sufficient number of GPU threads to maintain full utilization of the parallel hardware and hide memory access latency. Our scheme aims to maximize the utilization of the parallel hardware by generating a sufficiently large number of GPU threads to be able to hide memory access latency. Before we get into the details of our proposal, we begin with a serial version progressive decoding.

1. $\mathbf{p}_n = \mathbf{c}_n$
2. **for** k=1 **to** (n−1)
3. $\quad \mathbf{p}_n = \mathbf{p}_n - \mathbf{p}_k e_{nk}$
4. $\quad \mathbf{e}_n = \mathbf{e}_n - \mathbf{e}_k e_{nk}$
5. $\mathbf{p}_n = \mathbf{p}_n / e_{nn}$
6. **for** k=1 **to** (n−1)
7. $\quad \mathbf{p}_k = \mathbf{p}_k - \mathbf{p}_n e_{kn}$

Fig. 2: Simplified Progressive Decoding

For simplicity, we assume that all of the $\mathbf{c}_n$'s arrive with independent coefficient vectors. The progressive decoding runs the (serial) code shown in Figure 2 on arrival of each $\mathbf{c}_n$. Note that n = 1, 2, 3, ..., G on arrival of the first, second, third, ..., G^{th} packet, respectively, where G is the total number of blocks comprising a file. In addition, note that until the arrival of the last $\mathbf{c}_n$, i.e., $\mathbf{c}_G$, each $\mathbf{p}_k$ (k = 1, ... n) contains partially decoded data. After running the algorithm on arrival of the last $\mathbf{c}_n$, each $\mathbf{p}_k$(k = 1, ..., n) contains fully decoded, i.e., original, data. (As a final note on the algorithm, the pseudo code shown in Figure 2 is a simplified version of the progressive decoding, presented for ease of understanding, and is valid only when the assumption of independent coefficient vectors holds.)

In the baseline parallel decoding scheme, $\mathbf{p}_k$'s are partitioned into units and all are decoded in parallel using GPU threads. As mentioned above, one of the problems of this baseline parallel scheme is that it cannot exploit fully parallelism opportunities that GPGPUs offer. For example, when four-byte data partitions are used, the baseline scheme can generate 256 GPU threads with 1,024-byte long $\mathbf{p}_k$'s and 4,096 GPU threads with 16K-byte long $\mathbf{p}_k$'s. We claim that neither 256 threads nor even 4,096 threads are sufficient for full utilization of modern GPGPUs in parallel random linear network coding implementations. To take full advantage of modern GPGPUs with large memory access latency, a parallel algorithm must generate sufficiently many concurrent threads for hiding memory access latency. To address this problem, we solve the random linear network coding progressive decoding problem as follows.

1. $\mathbf{p}_n = \mathbf{c}_n$
2. **for** k=1 **to** (n−1) **do in parallel**
3. $\quad \mathbf{p}\prime_k = \mathbf{p}_k e_{nk}$
4. $\quad \mathbf{e}\prime_n = \mathbf{e}_k e_{nk}$
5. **for** $k = 1$ to ceiling((n−1)/S) **do in parallel**
6. $\quad$ **for** l=(k−1)× S+1 to min(k× S, n−(k−1)× S)
7. $\quad\quad \mathbf{q}_k = \mathbf{q}_k - \mathbf{p}\prime_l$
8. $\quad\quad \mathbf{f}_k = \mathbf{f}_k - \mathbf{e}\prime_l$
9. **for** $k = 1$ to ceiling((n−1)/S)
10. $\quad \mathbf{p}_n = \mathbf{p}_n - \mathbf{q}_k$
11. $\quad \mathbf{e}_n = \mathbf{e}_n - \mathbf{e}_k$
12. $\mathbf{p}_n = \mathbf{p}_n / e_{nn}$
13. **for** k= 1 **to** (n−1) **do in parallel**
14. $\quad \mathbf{p}_k = \mathbf{p}_k - \mathbf{p}_n e_{kn}$

where $\mathbf{q}_k$ and $\mathbf{f}_k$ are zero-initialized vectors and S is a natural number, e.g., 16.

The code in lines 2-11 corresponds to lines 2-4 of the serial version (in Figure 2), which requires n - 1 scalar and vector multiplications (e.g., $\mathbf{p}_k$ e_{nk}) and n−1 vector subtractions (e.g., $\mathbf{p}_n$ − $\mathbf{p}_k$.) (Here, we regard the concatenation of $\mathbf{e}_k$ and $\mathbf{p}_k$ as one vector.) The scalar and vector multiplications are first parallelized in lines 2-4. In those lines, **for** k = 1 to (n−1) do in parallel indicates that for each case where k has a value between 1 and n − 1, respective instruction execution units are generated and processed concurrently. In other words, when the receiver receives the coded frame $\mathbf{c}_n$, scalar and vector multiplication $\mathbf{p}\prime_k$=$\mathbf{p}_k$ e_{nk} operations must be executed simultaneously for each $\mathbf{p}\prime_k$, k = 1, ..., n−1. Note that each scalar and vector multiplication operation $\mathbf{p}\prime_k$=$\mathbf{p}_k$ e_{nk}

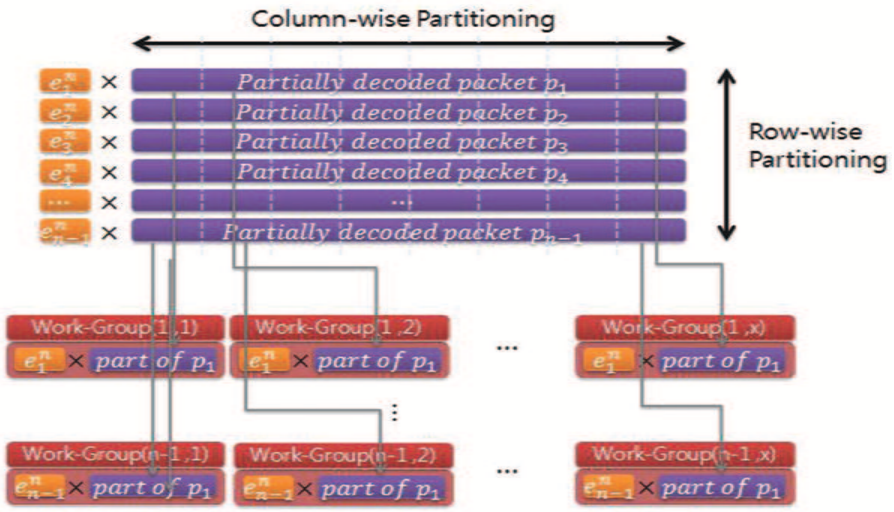


Fig. 3: Job partitioning in both column-wise and row-wise for increased the number of concurrent threads

is also parallelized by using the same method as in the baseline parallel decoding method, i.e., a vector is divided into several units and an independent thread works on each part. The vector subtractions are parallelized in lines 5-11. The optimal parallel algorithm in terms of time complexity for multiple addition/subtraction operations would be the *parallel reduction tree* method, but in typical CUDA devices the reduction tree algorithm experiences performance degradation owing to the high access latency of the global memory used for communication among threads. The code in lines 5-11 illustrates our method, in which subtraction operations are organized as groups, and parallelism is achieved only among groups, not in individual operations within a group. In our implementation, we use groups of size 16, i.e., S = 16.

Figure 3 depicts how coded data are partitioned and assigned to concurrent threads in our parallelized decoding scheme. In addition to the fact that the multiplication operation on each $\mathbf{p}_k$, $k = 1, \ldots, n-1$, is executed in parallel (row-wise partitioning), each $\mathbf{p}_k$, $k = 1, \ldots, n-1$, is divided into T-byte units and the multiplication operation on each unit is also executed in parallel (column-wise partitioning.) Therefore, the total number of concurrent threads generated and running concurrently is $(n-1) \times$ B/T where B is the size of $\mathbf{p}_k$. In our implementation, we use varying T, i.e., starting with four bytes we gradually increase T to 32 bytes as the block size increases. This is to adjust the maximum number of concurrent threads executed in parallel. To bring the best out of a parallel architecture, it is critical to create a proper number of concurrent threads in a program. If the concurrent threads are too few, the parallel architecture will be under-utilized, and if they are too many, excessive overhead caused by maintaining too many threads will degrade overall performance. As indicated previously, threads are organized as TBs (denoted as Work-Groups in Figure 3), and synchronization is allowed only among the threads in the same TB. Thus, each TB maintains its own coefficient matrix, i.e., there exist multiple copies of the coefficient matrix. To optimize memory access latency, the newly arrived (or last) row of the data and coefficient matrix is stored in the shared memory and the remaining rows are stored in the global memory. The reason is that in the progressive decoding process, only the newly arrived (or last) row is accessed multiple times, and most elements in the remaining rows are accessed once. The shared (on-chip/local) memory is fast, but space-limited (e.g., 48KB per SM(X)), and content must be copied from the global memory, i.e., data cannot be copied directly from the main (host) memory to the shared memory.

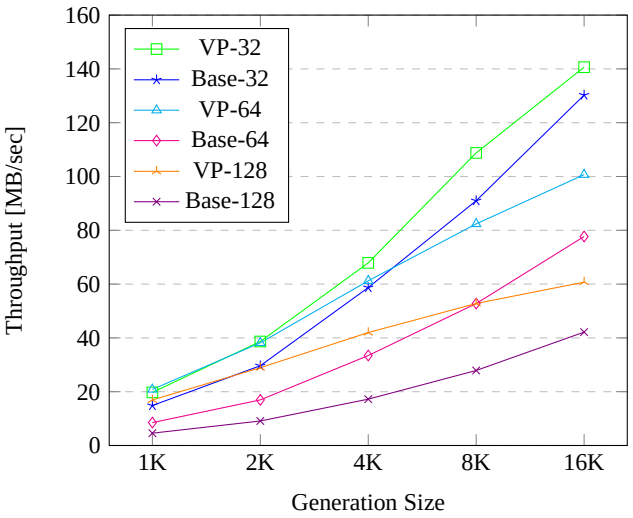
4 Performance Evaluation

In this section, we investigate the performance of our parallel method. To this end, we implemented our vertical partitioning (VP) technique and the baseline parallel method and performed experiments on real GPGPU-equipped systems. For the experiments, we used two different CUDA GPGPU units, GeForce GTX460 and GTX 670, featuring 336 (675MHz) and 1344 (915 MHz) CUDA cores, respectively. Seven SM(X)s are installed in both devices, and thus different numbers of cores exist in each SM depending on the device. The experimental systems are also equipped with an Intel-i7 960 3.2GHz quad-core CPU, CUDA Toolkit 3.2, and Windows 7, with MS Visual Studio 2010 as the compiler. In the experiments, the data or file being decoded is

divided into blocks and we refer to the number of blocks comprising a file being as the generation size. The total file size varies with generation size and block size. For example, if the generation size and block size are 1,024 and 16,384 bytes, respectively, then the total data size is $1{,}024 \times 16{,}384 = 16$ Mbytes. The metric used to compare the performance of implementations is decoding throughput (Mbytes/sec), calculated as the total size of decoded data divided by the decoding time. A look-up-table-based Galois field multiplication/division method is used in both implementations.

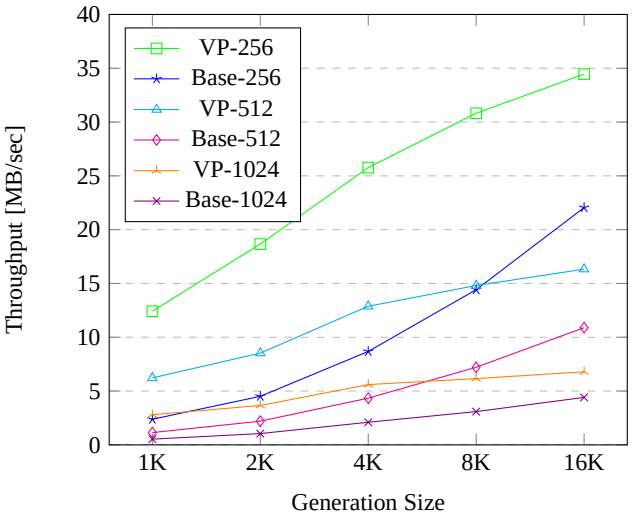
Figure 4 shows the throughputs of the two schemes on the GTX 460-based system with the generation size and block size varying from 32 to 1,024 and from 1,024 to 16K (16,384) bytes, respectively. In the figure, the x- and y-axes represent the block size (bytes) and throughput (MB/sec), respectively. The throughputs are also shown in numbers below the x-axis in a tabular form. As we can observe in Figure 4(b), VP shows over five times higher throughput than the baseline, when the generation size is greater than 256 and the block size is 1,024 bytes. VP shows around a 4x increase in throughput compared to the baseline, when the generation size is greater than 256 and the block size is 2,048. The performance advantage of VP comes from the fact that it maximizes parallelism, i.e., creates as many as $n - 1$ times more GPU threads than the baseline, where n is the generation size, so that every computational unit in a GPGPU device is utilized most of the time. In contrast, the baseline scheme cannot create sufficiently many GPU threads to maintain full utilization of the parallel hardware and to hide memory access latency. As indicated previously, a memory-intensive CUDA program must generate many more threads than the maximum number of threads simultaneously issued in a CUDA device to hide the substantial memory access latency. If there are sufficiently many threads, some can be executed, while others are stalled for memory access. The performance advantage of VP compared to the baseline becomes less dominant as either the generation size decreases or the block size increases. When the block size is 16K bytes, VP shows only 10 percent to 50 percent enhancement over the baseline, depending on the generation size. With block sizes as large as 16K, VP's performance advantage over the baseline is not prominent since the overhead for maintaining a large number of threads becomes much greater in VP's case, and the baseline scheme can also create a large number of threads that utilize the parallel hardware effectively. Any meaningful performance enhancement was not observed, when the generation size was smaller than 32.

Figure 5 compares the throughputs of the two schemes on the GTX 670-based system with the generation size and block size varying from 32 to 1,024 and from 1,024 to 16K (16,384) bytes, respectively. In the figure, the x- and y-axes represent the block size (bytes) and throughput (MB/sec), respectively, as in Figure 4. Similarly to the GTX 460 case, VP shows over five times higher throughput than the baseline, when the generation



	1K	2K	4K	8K	16K
VP-32	19.74	38.66	67.88	108.77	140.64
Base-32	14.85	29.65	58.70	90.99	130.26
VP-64	20.85	38.17	61.23	82.38	100.69
Base-64	8.51	16.95	33.44	52.76	77.69
VP-128	16.93	28.92	41.98	52.79	60.73
Base-128	4.58	9.11	17.22	27.89	42.15

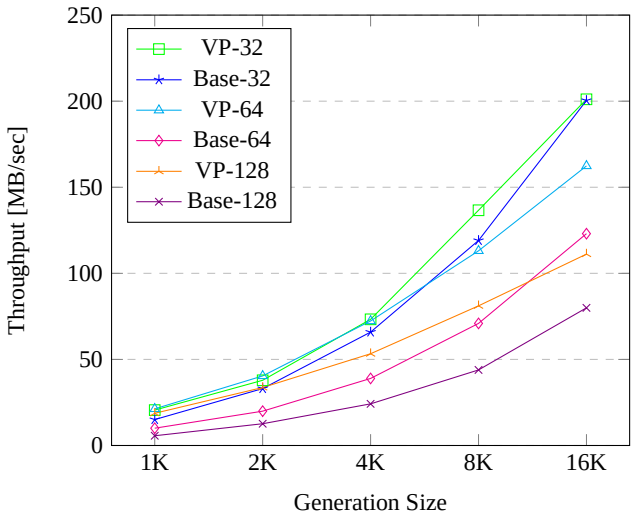
(a) Generation size of 32/64/128



	1K	2K	4K	8K	16K
VP-256	12.42	18.66	25.77	30.82	34.46
Base-256	2.38	4.51	8.68	14.42	22.05
VP-512	6.22	8.53	12.88	14.81	16.33
Base-512	1.14	2.20	4.34	7.21	10.89
VP-1024	2.80	3.66	5.60	6.16	6.79
Base-1024	0.53	1.05	2.10	3.09	4.41

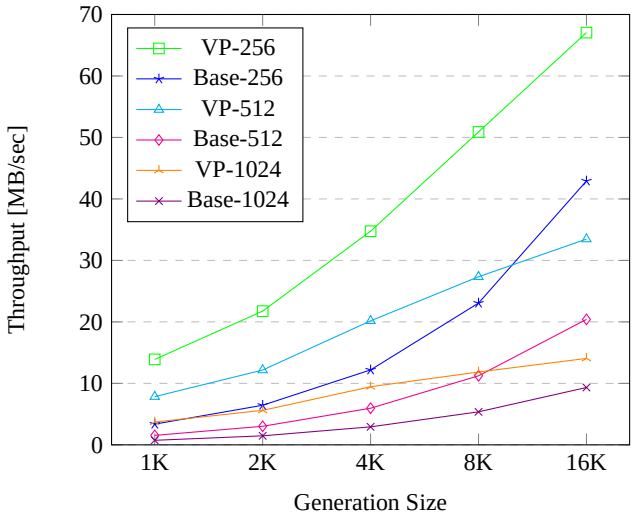
(b) Generation size of 256/512/1024

Fig. 4: Performance on GTX 470



	1K	2K	4K	8K	16K
VP-32	20.56	37.88	73.29	136.64	201.10
Base-32	15.06	33.06	65.87	119.09	200.42
VP-64	21.28	40.41	72.34	113.08	162.39
Base-64	10.04	19.94	38.95	70.92	123.04
VP-128	18.71	33.68	53.30	81.12	111.17
Base-128	5.66	12.64	24.19	43.89	79.95

(a) Generation size of 32/64/128



	1K	2K	4K	8K	16K
VP-256	13.90	21.75	34.75	50.89	67.05
Base-256	3.37	6.44	12.19	23.06	42.92
VP-512	7.84	12.17	20.17	27.34	33.47
Base-512	1.54	3.02	5.96	11.23	20.41
VP-1024	3.71	5.61	9.45	11.85	14.06
Base-1024	0.74	1.47	2.93	5.37	9.33

(b) Generation size of 256/512/1024

Fig. 5: Performance on GTX 670

size is either 512 or 1,024, and the block size is 1,024 bytes, and the enhancement of VP over the baseline diminishes as either the generation size decreases or the block size increases. When the block size is 16K bytes, VP shows 0.5 percent to 60 percent enhancement over the baseline depending on the generation size. VP on GTX 670 exhibits approximately twice the throughput of VP on GTX 460, with large generation sizes (above 512) and block sizes (above 8K). On GTX 670, which is based on the Kepler GK104 architecture, streaming multiprocessors (SMXs) feature the quad warp scheduler issuing twice as many concurrent threads as the dual warp scheduler in Fermi GF104-based GTX 460's streaming multiprocessors; thus, it is natural to observe GTX 670's doubling VP throughput compared to GTX 460.

5 Conclusion

Recently, GPGPU technology has paved the way for parallelizing random network coding; however, existing parallelization techniques for random network coding cannot utilize GPGPU architectures on many occasions. In this paper, we proposed a random network coding parallelization technique that can do so. Our proposed parallel method aimed at maximizing parallelism and showed as much as a 5x increase in throughput compared to an existing implementation on GTX 460/670 GPGPU devices.

Acknowledgement

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2013R1A1A1A05005876).

References

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

[2] T. Ho, M.Medard, R. Koetter, D. Karger, M. Effros, J. Shi, and B. Leong, A random linear network coding approach to multicast. IEEE Transactions on Information Theory, **52**, 4413-4430 (2006).

[3] S. Katti, H. Rahul, W. Hu, D. Katabi, M., Medard, and J. Crowcroft, XORs in the air Practical wireless network coding. IEEE/ACM Transactions on Networking, **16**, 497-510 (2008).

[4] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, Network coding for distributed storage systems. IEEE Transcations on Information Theory, **56**, 4539-4551 (2010).

[5] C. Gkantsidis and P.R. Rodriguez, Network coding for large scale content distribution. Proceedings of IEEE INFOCOM '05, Miami, FL, 13-17 March, pp 2235-2245 (2005).

[6] H. Shojania, and B. Li, Parallelized progressive network coding with hardware acceleration. Proceeding of the 15th IEEE International Workshop on Quality of Service.
[7] K. Park, J.-S. Park, and W. Ro, On improving parallelized network coding with dynamic partitioning. IEEE Transactions on Parallel and Distributed Systems, **21**, 1547-1560 (2010).
[8] NVIDIA, CUDA Toolkit, http://www.nvidia.com/content/cuda/cuda-toolkit.html
[9] H. Shojania, B. Li, and X. Wang, Nuclei: GPU accelerated many-core network coding. Proceedings of IEEE INFOCOM '09 (2009).
[10] H. Shojania, B. Li, Pushing the envelope: Extreme network coding on the GPU. Proceedings of IEEE International Conference on Distributed Computing Systems '09 (2009)
[11] S. Lee, and W. Ro, Accelerated network coding with dynamic stream decomposition on graphics processing unit. The Computer Journal, **55**, 21-34 (2012).
[12] X. Chu, K. Zhao, and M. Wang, Accelerating network coding on many-core GPUs and multi-core CPUs. Journal of Communications, **4**, (2009).
[13] M., Kim, K. Park, and W. Ro, Benefits of using parallelized non-progressive network coding. Journal of Network and Computer Applications, **36**, 293-305 (2013).
[14] J.-S. Park, S. Baek and K. Lee, A highly parallelized decoder for random network coding leveraging GPGPU. The Computer Journal, **57**, 233-240 (2014).
[15] C. Chen, C. Chen, S. Oh, J.-S. Park, M. Gerla, and M. Sanadidi ComboCoding: Combined intra-/inter-flow network coding for TCP over disruptive MANETs, Journal of Advanced Research, **2**, 241-252 (2011).
[16] P. Chou, Y. Wu, and K. Jain, Practical network coding. Proceedings of Allerton Conference on Communication, Control, and Computing '03 (2003).
[17] U. Lee, J.-S. Park, S. Lee, W. Ro, G. Pau, and M. Gerla, Efficient peer-to-peer file sharing using network coding in manet. Journal of Communications and Networks, 10 (2008).
[18] P. Maymounkov, N. Harvey, and D. Lun, Methods for efficient network coding. Proceedings of the 44th Annual Allerton Conference on Communication, Control, and Computing (2006).
[19] N. Melab, E.-G. Talbi, and S. Petiton, A parallel adaptive gauss-jordan algorithm. The Journal of Supercomputing, **17**, (2000).
[20] L. Csanky, Fast parallel matrix inversion algorithms. Proceedings of IEEE Symposium on Foundations of Computer Science '75 (1975).
[21] R. Bisseling and J. van de Vorst, Parallel LU decomposition on a transputer network. Proceedings of the Shell Conference on Parallel Computing '89 (1989).
[22] D. Kim, K. Park, and W. Ro, Network Coding on Heterogeneous Multi-Core Processors for Wireless Sensor Networks, Sensors, **11**, 7908-7933 (2011).
[23] S. Kim, W. Jeong, W. Ro, and J. Gaudiot, Design and Evaluation of Random Linear Network Coding Accelerators on FPGAs. ACM Transactions on Embedded Computing Systems, **13**, 1-24 (2013).

Seong-min Choi received the B.S. degree in computer Engineering from the Hongik University, Seoul, Korea, in 2012. He is currently serving in the Republic of Korea Air Force. His main research interests are parallel processing on GPGPU platforms, Multi-Core processors and Mobile platforms.

Kyogu Lee received the B.S. degree in Electrical Engineering from Seoul National University, Seoul, Korea, in 1996, the M.M. degree in Music Technology from New York University, New York, in 2002, and the M.S. degree in Electrical Engineering and the Ph.D. degree in Computer-based Music Theory and Acoustics from Stanford University, Stanford, CA, in 2007 and 2008, respectively. He worked as a Senior Researcher in the Media Technology Lab at Gracenote from 2007 to 2009. He is now an assistant professor in the Graduate School of Convergence Science and Technology at Seoul National University, Seoul, Korea and is leading the Music and Audio Research Group (MARG). His research focuses on signal processing and machine learning applied to music/audio.

Joon-sang Park received the M.S. degree in computer science from the University of Southern California, Los Angeles, in 2001 and the Ph.D. degree in computer science from the University of California at Los Angeles (UCLA), in 2006. He is currently an Associate Professor with the Computer Engineering Department, Hongik University, Seoul, Korea. His research interests include routing and medium-access control protocols in mobile ad hoc and sensor networks and network coding.