Advance Access publication on 20 January 2013
doi:10.1093/comjnl/bxs173

A Highly Parallelized Decoder for Random Network Coding leveraging GPGPU

JOON-SANG PARK[1,**], SEUNG JUN BAEK[2] AND KYOGU LEE[3]

[1]*Department of Computer Engineering, Hongik University, 94 Wausan-ro, Maop-gu, Seoul 121-791, South Korea*
[2]*College of Information and Communications, Korea University, Anam-dong 5-ga, Seongbuk-gu, Seoul 136-713, South Korea*
[3]*Department of Transdisciplinary Studies, Seoul National University, Seoul, South Korea*
[*]*Corresponding author: jsp@hongik.ac.kr*

Network coding has been shown to improve various performance metrics in computer networks. However, the use of network coding, especially random linear network coding, incurs serious time delay in the decoding process and thus it is imperative to use a network coding implementation that has low decoding latency characteristics, e.g. a parallelized implementation. In this paper, we investigate the problem of parallelizing Pipeline network coding, a variant of random linear coding recently developed in order to alleviate the problems of random linear coding. We propose a novel massively parallelized decoding algorithm leveraging General Purpose Graphics Processing Unit (GPGPU) and show its performance enhancement by up to 100% compared with previous GPGPU-based parallel algorithms via experiments on real systems.

Keywords: network coding; GPGPU; parallelization

Received 4 August 2012; revised 4 August 2012
Handling editor: Jongsung Kim

1. INTRODUCTION

Network coding [1] refers to the notion of performing coding operations on the contents of packets throughout a network. It is known that network coding enhances various performance metrics in computer networks. For example, network coding can increase the multicast throughput in wired networks [1]. In Peer-to-Peer (P2P) file-sharing systems, network coding can improve the system performance since it mitigates the piece selection (or transfer scheduling) problem [2]. In P2P file swarming systems, a file to be shared is divided into multiple pieces each of which can be downloaded independently from many different peers. To download a complete file, a node should collect all the pieces belonging to the file and if the node downloads multiple pieces simultaneously from peers, it dramatically reduces downloading delay, which is the main advantage of using the file swarming technique. However, the selection of peers and pieces to download has a big impact on the overall performance, which is generally referred to as the piece selection problem. When using network coding, the data are 'encoded' into blocks such that all the blocks are equally important, i.e. blocks being exchanged are indistinguishable, and thus the collecting node is only supposed to gather a specific number of equally important blocks from other peers. Also network coding helps exploit unique opportunities offered in wireless P2P environment, the broadcast nature of wireless medium and node mobility.

Although the advantages mentioned above render network coding attractive in the networking systems, the computational overhead of network coding may hinder the use of network coding in practice. When using random network coding, the data have to be encoded before being transferred at the sending node and the received data at the receiving node has to be decoded to recover the original information. Moreover, the decoding process of random network coding is implemented as a variation of Gaussian elimination which has $O(n^3)$ computational complexity where n is the number of blocks comprising a file. This complexity is quite expensive especially when the file size is huge. The time overhead spent for decoding

would cancel out all the benefits of reduced transmission time in network coding. Thus, it is critical to guarantee short decoding latency when implementing random network coding in practice.

There have been many attempts to overcome the high decoding latency of random network coding. To accelerate decoding speed, parallelized decoding algorithms based on multithreaded programming techniques for multi-core processors have been proposed in [3, 4]. Also, it has been shown that massively parallel decoding algorithms using General Purpose Graphics Processing Unit (GPGPU), e.g. Compute Unified Device Architecture (CUDA) architecture [5], can dramatically speed up the decoding process of network coding [6–9]. In this paper, we propose a new parallelization technique for the decoding process of random network coding using GPGPU. Our massively parallelized algorithm based on GPGPU further enhances the decoding speed of random network coding by up to 100% compared with existing GPGPU-based parallel decoding schemes through maximizing parallelism in the decoding process of random network coding. Our focus in this paper is Pipeline network coding [10], a variant of random network coding recently developed.

The rest of the paper is organized as follows. Section 2 discusses related works, Section 3 details our proposed parallelized decoding scheme, Section 4 shows the performance advantage of our proposed scheme and finally we conclude the paper in Section 4.

2. RELATED WORK

Network coding is generally attributed to Ahlswede *et al.* [1], who showed the utility of the network coding for multicast. The work of Ahlswede *et al.* was followed by other work by Koetter and Medard [11], which showed that codes with a simple, linear structure were sufficient to achieve the capacity of multicast connections in wireline networks. This result was augmented by Ho *et al.* [12], who showed that a random construction of the linear codes, i.e., random network coding, was sufficient. Chou *et al.* [13] proposed a practical way of implementing random network coding: network codes are carried along with packets. Gkantisidis *et al.* [2] have shown that random network coding is beneficial in large-scale P2P systems. Random network coding has been known to be a helpful technique for smooth, fast downloads and efficient server utilization [14] as well as mobile P2P system [15]. Park *et al.* showed improvements in the reliability of ad hoc network systems [16].

To solve the high decoding latency problem in random network coding, several approaches have been proposed. In [17], a new coding scheme with a lower computational complexity compared with the conventional random network coding has been proposed. Shojania *et al.* [3] suggested a parallelized decoding technique for multi-core CPUs with Single Instruction Multiple Data (SIMD) instructions, e.g. Intel's SSE. The decoding process in [3] is based on the well-known Gauss–Jordan elimination algorithm but it can 'progressively' decode 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 over a long period of time. Note that, however, the original data only can be recovered and decoding process can be finished at the arrival of the last data even though such as progressive decoding is used. Conventional parallelized approaches for the Guass-Jordan elimination and other related algorithms, e.g. [18–20], require the entire data before starting the decoding process and thus will incur additional decoding latency on the receiver compared with the progressive decoding. Park *et al.* also have proposed efficient parallelized progressive network coding algorithm with dynamic partitioning algorithms for multi-core CPUs [3]. Shojania *et al.* [5] have proposed GPGPU-based parallelized progressive decoding algorithm and in [6] parallel multi-segment decoding algorithm for buffered data. In these GPGPU-based parallelized schemes, hundreds of GPU threads can simultaneously encode and decode data blocks and thus can easily outperform multi-core CPU-based implementations. Chu *et al.* [8] showed improved encoding throughput on GPU with an aid of CPU. The decoding methodology used in [8] is not a progressive one as in [3]. Our massively parallelized algorithm on GPGPU further enhances the decoding speed of random network coding compared with existing GPGPU-based parallel decoding schemes by up to 100% through maximizing parallelism in the decoding process.

3. PARALLELIZATION OF RANDOM NETWORK CODING USING GPGPU

In this section, we describe our massively parallelized decoding algorithm for random network coding, especially Pipeline network coding. First, we introduce Pipeline network coding and GPGPU architecture and then we propose a massively parallelized decoding scheme using GPGPU for Pipeline network coding.

3.1. Pipeline network coding

Pipeline network coding [10] is basically the random (linear) network coding [12] with a small variance. Its main difference from the conventional random network coding is depicted in Fig. 1.

As we can see in the figure where **c**'s and **p**'s represent coded data and original data, respectively, the encoding matrix in Pipeline network coding becomes a low triangular matrix whereas an usual encoding matrix in the conventional random network coding does not contain zero elements (or does not form a low triangular matrix as the figure.) The encoding

$$\begin{bmatrix} \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_k \end{bmatrix} = \begin{bmatrix} e_1^{(1)} & 0 & \cdots & 0 \\ e_1^{(2)} & e_2^{(2)} & & \vdots \\ \vdots & & \ddots & 0 \\ e_1^{(k)} & e_2^{(k)} & \cdots & e_k^{(k)} \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_k \end{bmatrix}$$

FIGURE 1. Encoding matrix of Pipeline network coding.

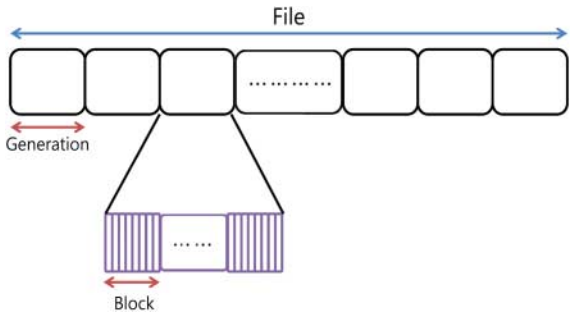

FIGURE 2. File partitioning with generations and blocks.

and decoding process of Pipeline network coding work as follows.

Encoding: A file to be transferred is divided into n blocks/pieces, $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n$ where the subscripts denote unique and consecutive *sequence numbers* (SNs) in $\mathbb{Z}_+$. Each $\mathbf{p}_k$ is a vector $[p_{k,1}\ p_{k,2}\ \cdots\ p_{k,B}]$ of B elements and for simplicity, we assume the size of each element $p_{k,l}$ is one byte. The blocks/pieces are logically reorganized into a set of *generations*, each of which is a set of pieces with adjacent SNs (see Fig. 2). Each generation is associated with a unique ID called *generation ID* (GID) which is defined as the smallest SN of the frames belonging to the generation. We use GIDs to denote generations. For a given GID g, a frame $\mathbf{p}_k$ is said to be in the generation g if $g \leq k < g + u_g$, where u_g denotes the size of the generation g. If we denote the generation that follows next to the generation g by $\bar{g}$, then $\bar{g} = g + u_g$. GIDs are not consecutive SNs in general.

When using network coding, nodes transfer *coded frames* instead of the original file pieces. A node can generate a *coded frames* $\mathbf{c}_{(g,s)}$ using the original file pieces which share the same GID g. The coded frames $\mathbf{c}_{(g,s)}$ is a linear combination of the frames in the generation g such that $\mathbf{c}_{(g,s)} = \sum_{k=1}^{s-g+1} e_k \mathbf{p}_{k+g-1}$, where e_k is drawn according to the uniform distribution over $\mathbb{F}$. In the header of a coded frames $\mathbf{c}_{(g,s)}$, the *encoding vector* $\mathbf{e} = [e_1 \cdots e_{s-g+1}]$ is stored along with the GID g and the SN s. Namely s is the largest among the SNs of the frames which are used to generate $\mathbf{c}_{(g,s)}$. For example, a coded frames $\mathbf{c}_{(1,1)} = e_1\mathbf{p}_1$ and $\mathbf{c}_{(1,2)} = e_1\mathbf{p}_1 + e_2\mathbf{p}_2$, and a coded frames $\mathbf{c}_{(1,3)} = \sum_{k=1}^{3} e_k\mathbf{p}_k$ can be generated and transmitted from a node. In this paper, lowercase boldface letters represent vectors, file pieces or coded frames, and uppercase letters represent matrices. All arithmetic operations are over a finite field $\mathbb{F}$ so that file pieces $\mathbf{p}_i$ and coded frames $\mathbf{c}$ are regarded as vectors over $\mathbb{F}$.

Decoding: The decoding process is basically solving the system of equations given in Fig. 1. For a given SN i, let $\mathbf{c}_k$ be a coded frames with the GID g and with an SN less than or equal to i in the local memory, $\mathbf{e}_k$ be the encoding vector prefixed to $\mathbf{c}_k$, and $\mathbf{p}_{g+k-1}$ be the frame to be recovered where $k = 1, \ldots, i - g + 1$. Now, let $\mathbf{E}^{\mathrm{T}} = [\bar{\mathbf{e}}_1^{\mathrm{T}} \cdots \bar{\mathbf{e}}_{i-g+1}^{\mathrm{T}}]$, $\mathbf{C}^{\mathrm{T}} = [\mathbf{c}_1^{\mathrm{T}} \cdots \mathbf{c}_{i-g+1}^{\mathrm{T}}]$ and $\mathbf{P}^{\mathrm{T}} = [\mathbf{p}_g^{\mathrm{T}} \cdots \mathbf{p}_i^{\mathrm{T}}]$. We obtain the original frames from $\mathbf{P} = \mathbf{E}^{-1}\mathbf{C}$ assuming $\mathbf{E}$ is invertible. Such a decoding process is usually implemented using the Guass–Jordan elimination algorithm.

In the conventional random network coding, the entire generation has to be collected to recover original data. The decoding process itself may start as soon as any coded frame is received to reduce overall decoding delay, i.e. progressive decoding, any original data can be recovered only after the entire generation, i.e. the same number of coded frame as the rank of the encoding matrix, is collected. Whereas in Pipeline network coding, a part of original data can be recovered from a part of generation collected.

3.2. General purpose graphic processing unit

General Purpose Graphics Processing Unit (GPGPU) refers to the notion of using GPUs as computation units for computational intensive applications, e.g. NVIDIA's *CUDA compute architecture* [5]. GPGPU is originated from the *programmable rendering pipeline* GPU architecture and the *shader* concept. In general, the programmable GPU hardware is composed of an array of computing units which are designed to execute arithmetic and floating point operations similar to general purpose processors. NVIDIA's CUDA (Compute Unified Device Architecture) is one of the first programming models that provide programmability on graphics hardware. In this paper, we leverage the CUDA model to develop a massively parallelized decoding scheme for random network coding. The CUDA compute architecture consists of a group of several streaming processors (SMs) each of which contains of a number of scalar processors (SPs), a.k.a., CUDA cores. In NVIDIA's GeForce GTX 460 (Fig. 3), there exist 7 SMs (blue squares) each of which contains 48 CUDA cores (green squares) and on-chip shared memory (depicted as a yellow square). The on-chip/local memory can be shared only by the CUDA cores belonging to the specific SM. Global memory can be used for communication between CUDA cores belonging to different SMs.

In a CUDA program, parallel execution is achieved via generating multiple GPU threads. In fact, hundreds of threads can be executed simultaneously in a CUDA GPGPU. GPU threads are grouped into thread blocks and each thread block is assigned to a SM. Synchronization and data sharing are allowed only for the threads in the same thread block. If there are a large number of thread blocks, more than one thread block

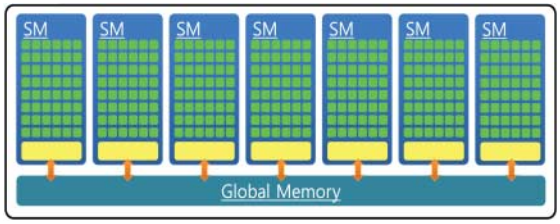

FIGURE 3. NVIDIA GeForce 460 architecture.

can be assigned on one SM. For efficient thread processing, every 32 GPU instructions in a thread block is grouped into a warp, which is a basic scheduling unit in the CUDA model. A warp is set of one same instruction of 32 GPU threads, and it is executed on a set of CUDA cores during four or two clock cycles (depending on CUDA hardware.) This model is referred to as SIMT (Single-Instruction, Multiple-thread). The SIMT unit of a 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 to another warp immediately. Therefore, creating hundreds (or over a thousand) of threads is essential to maintain maximum parallel hardware utilization and/or to hide memory access latency.

3.3. Massively parallelized decoding leveraging GPGPU

The decoding process of random network coding is usually implemented as a variant of Gauss–Jordan elimination algorithm. Following the notation used in the previous Section 3.2, the problem of decoding is solving and obtaining $\mathbf{P} = \mathbf{E}^{-1}\mathbf{C}$ when $\mathbf{E}$ and $\mathbf{C}$ are given. Usually $\mathbf{C}$ is a $m \times n$ matrix where $n > m$. To parallelize such a decoding process, encoded data, i.e. the matrix $\mathbf{C}$, is partitioned column-wise into several units (as shown in Fig. 4) and each unit is assigned a CPU or GPU thread such that each unit can be decoded independently. We refer to this scheme as the *baseline* parallel decoding scheme. All the existing GPGPU-based parallel decoding schemes [5–7] follow this approach. One of the problems of this baseline parallel decoding scheme is that in many cases it cannot fully exploit parallelism when the decoding process is run on a high-end GPGPU. Our scheme attempts to maximize parallelism in the decoding process such that every computational unit in a GPGPU is fully utilized. Now we explain our massively parallelized decoding algorithm in the framework of Pipeline network coding.

For ease of explanation, we assume that there is only one generation, i.e. all the original file pieces as well as coded frames belong to the same generation, and the size of all the original file pieces and coded frames (e.g. $\mathbf{p}_n$'s and $\mathbf{c}_n$'s) is B bytes. In case of Pipeline network coding, when a receiver receives nth coded frame $\mathbf{c}_n$ along with $\mathbf{e}^n$ vector, the decoding process to restore source data, $\mathbf{p}_n$ can be expressed as (1) owing to

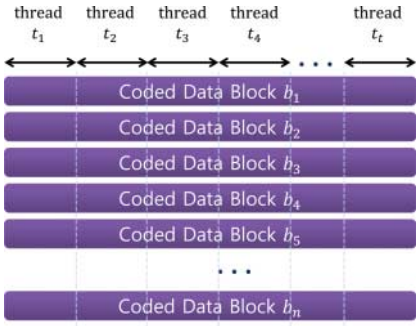

FIGURE 4. Data/job partitioning in baseline parallelization method.

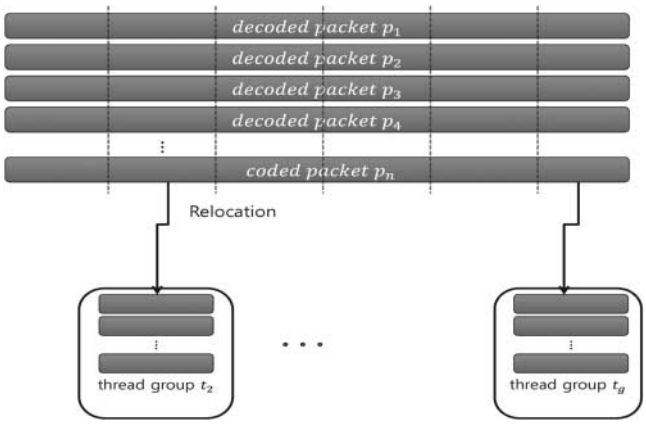

FIGURE 5. Baseline parallel decoding scheme for Pipeline network coding.

the special characteristics in the encoding matrix of Pipeline network coding shown in Fig. 1.

$$\mathbf{p}_n = (\mathbf{c}_n - \mathbf{p}_1 e_1^n - \mathbf{p}_2 e_2^n - \cdots - \mathbf{p}_{n-1} e_{n-1}^n)/e_n^n. \quad (1)$$

Using the above equation, the receiver can instantly recover $\mathbf{p}_n$ once $\mathbf{c}_n$ is received (assuming that the receiver has already received all the coded frames between $\mathbf{c}_1$ and $\mathbf{c}_{(n-1)}$ and restored all the original data from $\mathbf{p}_1$ to $\mathbf{p}_{(n-1)}$ using the same equation.) Note that it takes $\mathcal{O}(n \times B)$ to decode and retrieve $\mathbf{p}_n$ (again assuming that the receiver already has restored all the original data from $\mathbf{p}_1$ to $\mathbf{p}_{(n-1)}$) since there are n scalar/vector multiplications (e.g. $\mathbf{p}_1 e_1^n$) and each scalar/vector multiplication takes $\mathcal{O}(B)$ because the size of each vector is B bytes. To decode $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n$ and retrieve $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n$ all at once, it will take $\mathcal{O}(\sum_{i=1}^{n} i \times B) = \mathcal{O}(n^2 \times B)$.

The baseline parallel decoding scheme for Pipeline network coding is solving Equation (1) in a parallelized way as shown in Fig. 5. The coded frame $\mathbf{p}_n$ is divided (column-wise) into a number of partitions and all of partitions are being decoded simultaneously using GPU threads. Pseudo code for the baseline parallel decoding scheme is given below assuming that the size of each partition is T bytes.

for $i = 1$ to ceiling(B/T) **do in parallel**

$$\begin{aligned}\mathbf{p}_{n,[(i-1)\times T,i\times T]} = (&\mathbf{c}_{n,[(i-1)\times T,i\times T]} \\ &- \mathbf{p}_{1,[(i-1)T,iT]}e_1^n \\ &- \mathbf{p}_{2,[(i-1)T,iT]}e_2^n \\ &- \cdots \\ &- \mathbf{p}_{n-1,[(i-1)\times T,i\times T]}e_{n-1}^n)/e_n^n,\end{aligned}$$

where $\mathbf{p}_{n,[s,e]}$ is a vector $[p_{n,s+1}\ p_{n,s+2}\ \cdots\ p_{n,e}]$.

When there are V processing cores, using the baseline parallel scheme above, it will take $\mathcal{O}(n \times B/V)$ to decode and retrieve $\mathbf{p}_n$. However, in real implementations the baseline scheme cannot achieve the computational complexity in most cases since at least $\mathcal{O}(n)$ operations and/or data must be grouped together when distributing operations/data to multiple cores using only the column-wise partitioning method. That is, when $V > B$, some number of processing cores cannot be assigned jobs/data, i.e. the system is under-utilized. As mentioned previously, one of the problems of the baseline scheme is that it is incapable of fully exploiting parallelism opportunities that GPGPUs offer in many cases. In a CUDA GPGPU, hundreds of threads can be executed concurrently. Moreover, due to the large memory access latency in CUDA architectures it is recommended that a CUDA program generates a number of threads an order of magnitude larger than the number of threads that a CUDA GPGPU can execute in parallel such that memory latency can be hidden, e.g. while some threads are stalled for memory access other threads can be executed. Therefore, creating a very large number of threads is essential to maintain the maximum utilization of the parallel hardware. To address this issue, we propose a highly parallelized decoding scheme for Pipeline network coding. Our highly parallelized decoding scheme aims to generate a sufficiently large number of concurrent threads to take full advantage of CUDA GPGPUs. (Note that, same as the baseline scheme, our highly parallelized decoding scheme does not require the entire generation to be presented before starting the decoding process. It starts decoding as soon as the first coded frame arrives and can retrieve an original data piece every time a new coded frame arrives.)

We solve Equation (1) in a highly parallelized way as follows.

Step 1:
  for $i = 1$ to $(n-1)$ **do in parallel**
    $\mathbf{p}'_i = e_i^n \mathbf{p}_i$
Step 2:
  for $i = 2$ to $(n-1)$
    $\mathbf{p}'_1 = \mathbf{p}'_1 + \mathbf{p}'_i$
  $\mathbf{p}_n = (\mathbf{c}_n - \mathbf{p}'_1)/e_n^n$

In Step 1, **for** $i = 1$ to $(n-1)$ **do in parallel** indicates that independent instruction execution units for each $i = 1, \ldots, (n-1)$ are to be generated and processed concurrently. Put another way, scalar/vector multiplication operations, $\mathbf{p}'_i = e_i^n\ \mathbf{p}_i$ for each $i = 1, \ldots, (n-1)$, should be 'concurrently' executed. In fact, the algorithm described above is shown in a simplified form for the purpose of comparison to the baseline case. Each scalar/vector multiplication operation ($\mathbf{p}'_i = e_i^n \mathbf{p}_i$) in Step 1 and each vector addition operation ($\mathbf{p}'_1 = \mathbf{p}'_1 + \mathbf{p}'_i$) in Step 2 must also be parallelized using the same (column-wise) partitioning method used in the baseline parallel decoding method. Moreover, serially executed $(n-2)$ vector addition operations in Step 2 can be further parallelized using the *parallel tree reduction* method. The final form of our highly parallelized decoding algorithm is as follows.

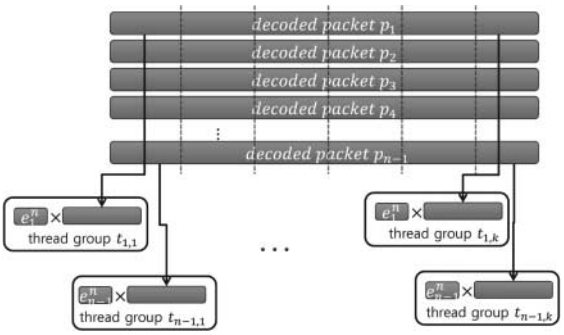

FIGURE 6. Data/Job partitioning in highly parallelized decoding method.

Step 1:
  for $i = 1$ to $(n-1)$ **do in parallel**
    for $j = 1$ to ceiling(B/T) **do in parallel**
      $\mathbf{p}_{i,[(j-1)T,jT]} = e_i^n \mathbf{p}_{i,[(j-1)T,jT]}$
Step 2:
  for $i = 1$ to ceiling$(\log_2 n)$
    for $j = 1$ to ceiling$(n/2^i)$ **do in parallel**
      $k = 1 + (j-1) \times 2^i$
      for $m = 1$ to ceiling(B/T) **do in parallel**
        $\mathbf{p}'_{k,[(m-1)T,mT]} = \mathbf{p}'_{k,[(m-1)T,mT]} + \mathbf{p}'_{(k+2^{i-1}),[(m-1)T,mT]}$
  for $i = 1$ to ceiling(B/T) **do in parallel**
    $\mathbf{p}_{n,[(i-1)T,iT]} = (\mathbf{c}_n - \mathbf{p}'_{1,[(i-1)T,iT]})/e_n^n$

where each $\mathbf{p}_s$, $s > n-1$, is an all-zero vector at start and T is the size of each partition when performing column-wise partitioning.

Figure 6 depicts how encoded data are partitioned and assigned to concurrent threads in our highly parallelized decoding scheme. Besides the fact that the multiplication operation on each $\mathbf{p}_k$, $k = 1, \ldots, n-1$, is executed in parallel, i.e. 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 executed also in parallel, i.e. column-wise partitioning. Therefore, the total number concurrent threads generated and being executed in parallel is $(n-1) \times B/T$ as opposed to B/T in the baseline case. In the experiments to be presented in the next section, we use 4 bytes as T and varying B. 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.

4. PERFORMANCE EVALUATION

In this section, we investigate the performance of our highly parallelized decoding scheme for Pipeline network coding via experiments on real systems. To this end, we have implemented our highly parallelized decoding scheme as well as the baseline parallel decoding method and performed experiments on real GPGPU equipped systems. (Hereafter, we denote our highly parallelized decoding scheme as HPDS.) And for the comparison purpose, we have also implemented and used a multithreaded parallel decoding method capable of full utilization of multicore CPUs as in [3]. For the experiments, we use the system with CUDA Toolkit 3.2, Windows 7 Ultimate OS, MS Visual Studio 2010 compiler, Intel-i7 960 3.2 GHz quad-core CPU and GeForce GTX460 675 MHz GPU unless otherwise specified. As indicated previously, GeForce GTX460 GPU is equipped with 7 SMs each of which contains 48 CUDA cores.

In the experiments, we vary the generation size from 8 to 2048 and the block size from 1024 bytes to 32 768 bytes. All the results are averaged over 100 runs. The data (or file) being decoded consists of one generation and its total size varies with the generation size and block size used in the experiments. For example, if the generation size is 2048 and the block size is 32 768 bytes, then the total data size is $2048 * 32\,768 = 64$ Mbytes. The metrics we use to compare the three schemes is decoding $throughput$ (Mbytes/s) which is calculated as the total size of data being decoded divided by the time spent for decoding. The average decoding latency per coded frame can be calculated as $throughput$ multiplied by $blocksize$ and it takes more time to decode and retrieve the last block in a generation than the first one. All the data being decoded are available in the main system memory (not in GPU memory) from the start of the decoding process. (Thus, the decoding time includes memory transferring to/from GPU memory.) We investigate the throughput as a function of the generation size and the block size but not a function of the number of generations comprising a file (or data.) One can easily estimate the decoding latency of a file with multiple generations by multiplying time to decode one generation of the file and the number of generations in the file. We use the look-up table-based Galois Field multiplication/division method.

Figure 7 compares the throughputs of the three schemes when the generation size is set to 2048 and the block size varies from 1024 to 4096. As we can see in the figure our proposed scheme denoted as GPU-HPDS outperforms the baseline scheme denoted as GPU-baseline and the CPU-based

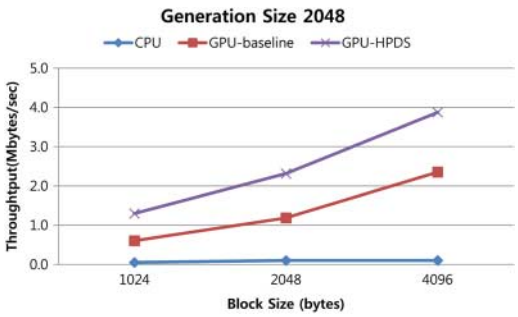

FIGURE 7. Experiment results with generation size of 2048.

decoding scheme labeled as CPU. GPU-HPDS exhibits over 2-fold higher throughput with 1024-byte blocks, 95% higher throughput with 2048-byte blocks and 65% higher throughput with 4096-byte blocks than GPU-baseline. The performance of the CPU-based parallel scheme is very low and incomparable to GPU schemes. The GeForce GTX460 GPGPU unit used in our experiments is based on the Fermi architecture, CUDA compute capability 2.1, in which an SM can issue two warps concurrently (with its *dual warp scheduler*). A naive calculation of the maximum number of threads that can be 'concurrently executed' in a GeForce GTX460 GPU is 2×32 (no. threads per warp) $\times 7$ (no. of SMs in GTX460) $= 448$ threads. In the baseline scheme, the number of concurrent threads generated is the block size divided by 4. (Each thread is responsible for computing 4 bytes of each row/block.) When the block size is 2048, the number of threads the baseline scheme generates is 512 which is a little more than the number of threads that can be issued in parallel in GTX460. Apparently, however, 512 threads are not enough for full exploitation of a GTX460 GPU. As indicated previously, the number of threads has to way exceed the maximum number of threads concurrently executable in a CUDA device since otherwise large memory access latency of CUDA architectures cannot be hidden, i.e. some times no instruction is executed because all threads are stalled for memory access. If there are enough number of threads, some threads can be executed while some other threads are waiting for accessing memory. CUDA architectures in general has large memory access latency compared with modern CPUs with deep memory hierarchy and thus it is crucial for CUDA programs to implement tactics to hide such large memory latency to achieve full utilization of CUDA architecture, i.e. generate a large number of concurrent threads such that there always exist ready-to-be-issued threads while some threads are stalled for accessing memory. GPU-HPDS creates a large number of threads and thus shows enhanced performance over the baseline scheme.

In Fig. 8, experimental results for the fixed 1024 generation size and varying block size from 1024 to 8192 are presented. The trend is similar to the case of 2048 generation size. HPDS shows up to 100% enhancement over the baseline. However, as the

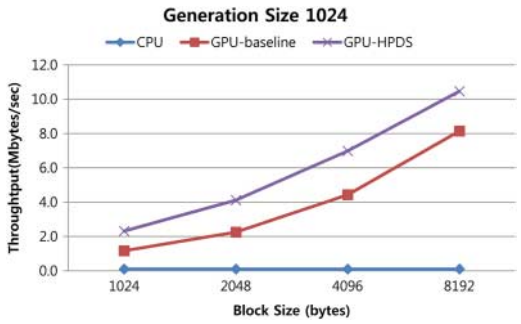

FIGURE 8. Experiment results with generation size of 1024.

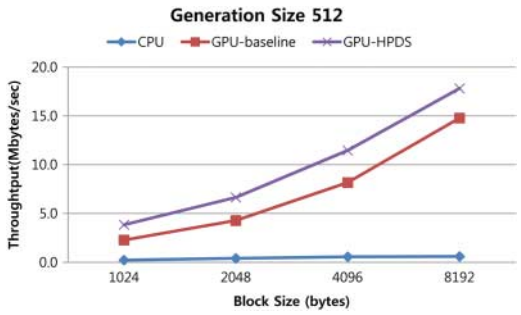

FIGURE 9. Experiments with generation size of 512.

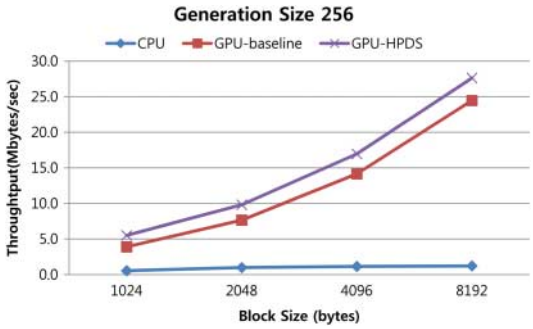

FIGURE 10. Experiments with generation size of 256.

block size gets bigger the performance improvement becomes less significant.

In Fig. 9, we can see that the performance trend when the generation size is 512 is similar to the case of 2048 and 1024 generation sizes but the performance improvement is less significant. GPU-HPDS shows 70% enhancement with 1K blocks and 20% improvement with 8K blocks over GPU-baseline.

When the generation size is 256, GPU-HPDS exhibits 40% enhancement with 1K blocks and 13% enhancement with 8K blocks over GPU-baseline as shown in Fig. 10.

When the generation size is 128, HPDS exhibits 10% enhancement with 1K blocks and 4% enhancement with 8K

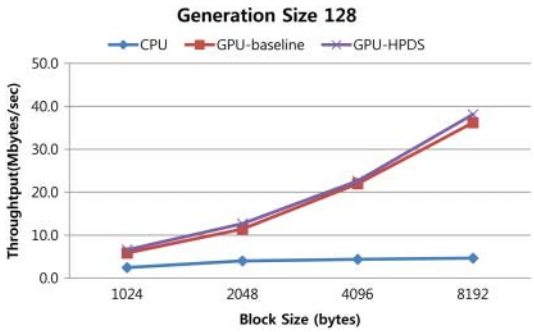

FIGURE 11. Experiments with generation size of 128.

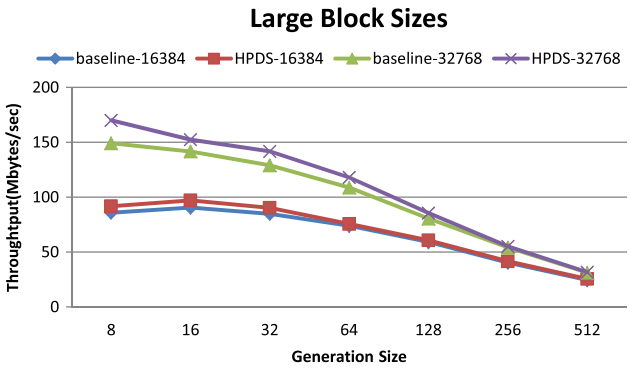

FIGURE 12. Experiments with large block sizes.

blocks over the baseline as shown in Fig. 11. It is worth noting that as the generation size decreases, the performance advantage of HPDS decreases. This is mainly because the overhead for maintaining a larger number of threads cancels out the additional parallelism created by HPDS. HPDS produces $(n-1)$ times many threads at maximum than the baseline scheme (where n is the generation size) so the additional parallelism created by HPDS becomes lower as the number of generations decreases. But there is a price for creating additional threads so only a few more threads may not be helpful in overall performance. With the generation size <128, performance enhancement from HPDS is not observed when the block size is in between 1K and 8K bytes.

Figure 12 compares throughputs of HPDS and the baseline for 16K-byte and 32K-byte blocks size cases. Note that HPDS shows enhancement over the baseline when the generation size is small but the throughputs of the two schemes get closer as the generation size increases. From the previous results, we have learnt that HPDS shows performance enhancement when HPDS can generate a sufficiently large number of threads with large generation sizes. However, when using large block sizes such as 16K and 32K bytes, there already exist a sufficient number of threads with the baseline scheme so HPDS's performance advantage may not appear in such cases.

5. CONCLUSIONS

In this paper, we have proposed a highly parallelized decoding algorithm for random network coding using GPGPU. Via experiments on real systems, we have shown that our proposed parallel decoding scheme can enhance the throughput of random network coding by up to 100%. We expect that with more powerful GPGPUs and/or multiple GPGPUs, we will achieve a higher performance improvement with HPDS. We plan to investigate the performance of HPDS in different GPGPU platforms in the future. Also there are still more optimization opportunities in maintaining threads in HPDS, which will lead to further performance enhancement. And there may occur disk I/O issues when the file size is much larger than the main memory and/or GPU memory. We leave those for future work as well.

ACKNOWLEDGEMENTS

The authors are grateful to Seong-min Choi for his assistance in experiments and graphics.

FUNDING

This work was supported by the National Research Foundation of Korea Grant funded by the Korean Government (2010-0005334, 2010-0027410).

REFERENCES

[1] Ahlswede, R., Cai, N., Li, S.-Y.R. and Yeung, R.W. (2000) Network information flow. *IEEE Trans. Inf. Theory*, **46**, 1204–1216.

[2] Gkantsidis, C. and Rodriguez, P.R. (2005) Network Coding for Large Scale Content Distribution. *Proc. IEEE INFOCOM '05*, Miami, FL, March 13–17, pp. 2235–2245. IEEE, New York.

[3] Shojania, H. and Li, B. (2007) Parallelized Progressive Network Coding with Hardware Acceleration. *Proc. 15th IEEE International Workshop on Quality of Service*, Evaston, Illinois, USA. IEEE.

[4] Park, K., Park, J.-S. and Ro, W.W. (2010) On improving parallelized network coding with dynamic partitioning. *IEEE Trans. Parallel Distrib. Syst.*, **21**, 1547–1560.

[5] NVIDIA, CUDA Toolkit, http://www.nvidia.com/content/cuda/cuda-toolkit.html.

[6] Shojania, H., Li, B. and Wang, X. (2009) Nuclei: GPU Accelerated Many-Core Network Coding. *Proc. IEEE INFOCOM '09*, Rio de Janeiro, Brazil. IEEE.

[7] Shojania, H. and Li, B. (2009) Pushing the Envelope: Extreme Network Coding on the GPU. *Proc. IEEE Int. Conf. on Distributed Computing Systems '09*, Montreal, Quebec, Canada. IEEE.

[8] Chu, X., Zhao, K. and Wang, M. (2009) Accelerating network coding on many-core gpus and multi-core cpus. *J. Commun.*, **4**.

[9] Lee, S. and Ro, W. (2012) Accelerated network coding with dynamic stream decomposition on graphics processing unit. *Comput. J.*, **55**, 21–34.

[10] Chen, C.-C., Chen, C., Oh, S., Park, J.-S., Gerla, M. and Sanadidi, M.Y. (2011) ComboCoding: combined intra-/inter-flow network coding for TCP over disruptive MANETs. *J. Adv. Res.*, **2**, 241–252.

[11] Koetter, R. and Medard, M. (2003) An algebraic approach to network coding. *IEEE/ACM Trans. Netw.*, **11**, 782–795.

[12] Ho, T., Medard, M., Koetter, R., Karger, D., Effros, M., Shi, J. and Leong, B. (2006) A random linear network coding approach to multicast. *IEEE Trans. Inf. Theory*, **52**, 4413–4430.

[13] Chou, P.A., Wu, Y. and Jain, K. (2003) Practical Network Coding. *Proc. Allerton Conf. on Communication, Control, and Computing '03*, Monticello, Illinois, USA. University of Illinois.

[14] Gkantsidis, C., Miller, J. and Rodriguez, P. (2006) Comprehensive View of a Live Network Coding P2P System. *Proc. ACM SIGCOMM Conf. on Internet Measurement '06*, Rio de Janeriro, Brazil. ACM.

[15] Lee, U., Park, J.-S., Lee, S.-H., Ro, W.W., Pau, G. and Gerla, M. (2008) Efficient peer-to-peer file sharing using network coding in manet. *J. Commun. Netw.*, **10**, 422–429.

[16] Park, J.-S., Lun, D.S., Yi, Y., Gerla, M. and Medard, M. (2006) CodeCast: A network coding based ad hoc multicast protocol. *IEEE Wirel. Commun.*, **13**, 76–81.

[17] Maymounkov, P., Harvey, N.J.A. and Lun, D.S. (2006) Methods for Efficient Network Coding. *Proc. 44th Annual Allerton Conf. on Communication, Control, and Computing*, Monticello, Illinois, USA. University of Illinois.

[18] Melab, N., Talbi, E.-G. and Petiton, S. (2000) A parallel adaptive Gauss–Jordan algorithm. *J. Supercomput.*, **17**.

[19] Csanky, L. (1975) Fast Parallel Matrix Inversion Algorithms. *Proc. IEEE Symp. on Foundations of Computer Science '75*.

[20] Bisseling, R.H. and van de Vorst, J.G.G. (1989) Parallel LU Decomposition on a Transputer Network. *Proc. Shell Conf. on Parallel Computing '89*, Amsterdam, The Netherlands, Lecture Notes in Computer Science, Vol. 384. Springer-Verlag, Berlin.