Accelerating Network Coding on Many-core GPUs and Multi-core CPUs

Xiaowen Chu and Kaiyong Zhao
Department of Computer Science, Hong Kong Baptist University, Hong Kong, China
Email: {chxw, kyzhao}@comp.hkbu.edu.hk

Mea Wang
Department of Computer Science, University of Calgary, Calgary, Canada
Email: meawang@ucalgary.ca

Abstract— **Network coding has recently been widely applied in various distributed systems for throughput improvement and/or resilience to network dynamics. However, the computational overhead introduced by network coding operations is not negligible and has become the obstacle for practical deployment of network coding. In this paper, we exploit the computing power of commodity many-core Graphic Processing Units (GPUs) and multi-core CPUs to accelerate the network coding operations. We propose a set of parallel algorithms that maximize the parallelism of the encoding and decoding processes and fully utilize the power of GPUs. This paper also shares our optimization design choices and our workarounds to the challenges encountered in working with GPUs. With our implementation of the algorithms, we are able to achieve significant speedup over existing solutions on CPUs.**

Index Terms— **Network coding; GPU computing**

I. Introduction

Recent advances in Graphics Processing Units (GPUs) open a new era of GPU computing [18]. Commodity GPUs like NVIDIA's GTX 280 has 240 processing cores and can achieve 933 GFLOPS of peak computational horsepower. GPUs are traditionally used for graphical applications. Since the release of CUDA programming model, it becomes much easier to develop non-graphical applications on GPUs [1] [3]. CUDA treats the GPU as a dedicated coprocessor of the host CPU, and allows the same code to be simultaneously running on hundreds of GPU processing cores as threads.

Network coding has been originally proposed to improve throughput in a multicast session. Since the landmark paper on randomized network coding by Ho et al. [21], there has been a gradual shift in research focus, from theoretical studies to more practical studies. The studies in [10-12] have shown that the computational complexity of network coding is an obstacle of practical network coding applications, especially for large data set. In this paper, we seek to take advantages of the GPU computing power to mitigate the computational overhead, so that network coding can be more practical. The contributions of this paper are as follows:

- We have designed and implemented a massively parallel batched encoding algorithm that maximizes the parallelism of the encoding process on GPUs.
- We have designed and implemented a parallel decoding algorithm that utilizes the many-core GPUs and multi-core CPUs. The multi-core CPU performs matrix inversion and the many-core GPU performs matrix multiplication.
- We have thoroughly evaluated the proposed algorithms by real experiments on three different GPUs. We have achieved a new record of network coding throughput on commodity GPUs: 325 MB/s for encoding and 212 MB/s for decoding, when coding 128 data blocks.

The implementation of network coding operations on GPU is not as simple as it seems. There are quite a number of challenges to address and optimization decisions to make. Just to name a few: First, we wish to utilize each one of the cores available on the GPU. Second, we need to minimize overheads introduced by memory access. Third, we must design the implementation to meet the special GPU architecture. Last but not least, we have to tailor the memory usage on the GPU to achieve the highest throughput.

The rest of the paper is organized as follows. Sec. II provides background information on network coding, GPU architecture, as well as the CUDA programming model. Sec. III presents the design of massively parallel network coding. Experimental results are presented in Sec. IV, and we conclude the paper in Sec. V.

II. Background and Related Works

A. Network Coding

Network coding has been originally proposed to achieve the optimal throughput in a multicast session [5]. 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. It has been shown that random linear codes using Galois Fields are sufficient to implement network coding in a practical network setting [7]. Linear network coding regards the messages as vectors of elements in a

Manuscript received March 20, 2009; revised June 30, 2009; accepted July 30, 2009.
This work is supported by FRG grant FRG2/08-09/098.

© 2009 ACADEMY PUBLISHER
doi:10.4304/jcm.4.11.902-909

finite field, and the encoding function is a simple linear combination over the finite field.

In our implementation, we define the linear network coding operations as follows. The data to be distributed is divided into n original blocks (b_1, b_2, …, b_n), where each block b_i consists of m codewords ($b_{i,1}$, $b_{i,2}$, …, $b_{i,m}$). An encoded block e_i is a linear combination of the n original blocks. Each of the m codewords in e_i is calculate as:

$$e_{i,k} = \sum_{j=1}^{n} c_{i,j} \cdot b_{j,k}, k \in \{1, \ldots, m\} \quad (1)$$

where ($c_{i,1}$, $c_{i,2}$, …, $c_{i,n}$) are the coding coefficients in the vector format. We can rewrite Eq. 1 in matrix format as $E = C \times B$, where $E = \{e_{i,k}\}$, $C = \{c_{i,j}\}$, and $B = \{b_{j,k}\}$. Upon receiving n encoded blocks (e_1, e_2, …, e_n), whose coding coefficient vectors (c_1, c_2, …, c_n) are all linearly independent of each other, the receiver can recover the original blocks as follows:

$$b_{i,k} = \sum_{j=1}^{n} c'_{i,j} e_{j,k}, \; k \in \{1, \ldots, m\}, i \in \{1, \ldots, n\}. \quad (2)$$

In other words, we have $B = C^{-1} \times E$, where $C^{-1} = \{c'_{i,j}\}$ is the inverse of the coding matrix C.

The key to the practical implementation of network coding is generating the coding coefficients to be used by each of the intermediate nodes in the session. Deterministic algorithms have been proposed and shown to be polynomial time computable [4], but they require extensive exchanges of control messages. Ho et al. [7] proposed the concept of randomized network coding, in which the coding coefficients are generated independently and randomly at each node.

Network coding operations are performed in Galois Field, which preserves the size of the original data, i.e., no additional bandwidth is required [6] [8]. The actual computational overhead incurred by network coding operation in GF(2^r) has been extensively studied in [10-12]. In this paper, we will focus on GF(2^8), which implies that each codeword consists of one byte of data. A network-coding accelerator is proposed in [11] to exploit SSE2 on x86 and AltiVec SIMD vector instructions on PowerPC processors. However, with the increase of n, the encoding and decoding throughput drop quickly. In this paper, we aim to further improve the scalability of network coding operations by using GPUs. Very recently, GPUs have been used as an efficient approach to overcoming the computational complexity of network coding [22-24]. The *Nuclei* system can achieve an encoding throughput of 66.9 MB/s and decoding throughput of 47 MB/s for n = 128 by using NVIDIA GeForce 8800 [23]. Their approach is based on a highly optimized loop-based GF(2^8) multiplication. Different from the work in [23], our previous work uses a table lookup approach [22]. In this paper, we further extend and elaborate more on the table lookup approach and thoroughly evaluate the performance on three different GPUs. We have achieved a new record of encoding throughput at 325 MB/s and decoding throughput at 212 MB/s for n = 128 by using NVIDIA GTX 280. We believe that loop-based approach and table lookup approach are complementary with each other.

B. GPU Computing

Due to the huge demand for powerful computing for real-time applications and high-definition 3D graphics, GPUs have been evolved into highly paralleled and multithreaded many-core processors. As shown in Fig. 1, the NVIDIA GeForce GTX280 has 30 Streaming Multiprocessors (SMs), each with 8 Scalar Processors (SPs). The design of the SMs is based on the Single-Instruction Multiple-Data (SIMD) architecture, i.e., at any given clock cycle, all SPs of the same SM execute the same instruction, but on different data.

Off-chip memories on Graphic cards, such as local memory and global memory, have relatively long access time, usually 400 to 600 clock cycles [3]. Inside a GPU, each SM has four types of on-chip memory, namely, constant cache, texture cache, registers, and shared memory [16]. Constant cache and texture cache are both read-only memories shared by all SPs. In general, the shared memory should be carefully utilized to amortize the global memory latency cost.

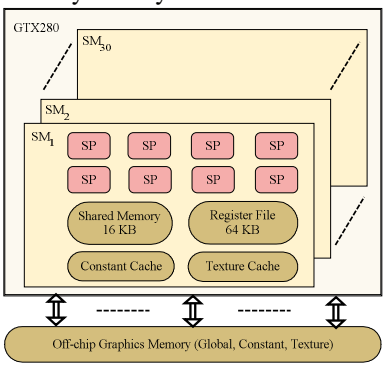


Figure 1. The architecture of NVIDIA GeForce GTX280

C. CUDA Programming Model

The first generation of general-purpose computing on GPUs (GPGPU) requires that non-graphics application must be mapped through the graphics application programming interfaces [14]. NVIDIA announced a general-purpose parallel programming model, namely Compute Unified Device Architecture (CUDA) [1] [3], which extends the C programming language for general-purpose application development. Meanwhile, AMD introduced Close To Metal (CTM) programming model that provides an assembly language for application development [2]. Intel will release Larrabee [20], 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 [15-19]. For these reasons, we chose to use CUDA in our research.

In CUDA, the GPU is regarded as a coprocessor capable of executing a number of threads in parallel. A single program consists of the *host code* to be executed on CPU and the *kernel code* to be executed on GPU. The kernel code is usually computational-intensive and data-parallel, and is executed on the GPU. Intuitively, the kernel code is naturally multi-threaded. Threads are organized into *thread blocks*, where each block is

© 2009 ACADEMY PUBLISHER

associated with one SM. Threads belonging to the same thread block can share data through the shared memory and can perform barrier synchronization.

III. Massively Parallel Network Coding

Our objective is to conduct large scale network coding in a parallel fashion on many-core GPUs and multi-core CPUs when appropriate. This section discusses the design, the challenges encountered, as well as our workarounds.

A. Encoding

The inputs of the encoding process are the matrix B (n original blocks consist of m codewords each) and the matrix C (a series of $1 \times n$ coding coefficient vectors). The output of the process is E, a series of encoded blocks, and each encoded block consists of m codewords. Hence, we need to first generate C and then perform the encoding operation, $E = C \times B$, to produce the encoded blocks. The random elements in C can be easily generated by the random number generator available in the CUDA library, which can generate tens of millions of random numbers per second. The time required to generate C is negligible as compared with the whole encoding process. In the following, we will discuss how to perform block-by-block encoding on GPUs. We then accelerate the encoding process by introducing the batched encoding approach.

a. Encoding a single block

The encoding process invokes addition and multiplication in GF(2^8). Addition in GF(2^8) is the same as XOR which is an efficient built-in operation on GPUs; but multiplication in GF(2^8) is more complicated and not directly supported by GPUs. The highly optimized *Nuclei* system uses 11 cubin instructions to implement the GF-multiplication operation [23]. Different from *Nuclei*, we choose to use a table-lookup approach, due to the fact that GPUs have not only huge computing power but also a sophisticated hierarchical memory system.

We propose two different table-lookup algorithms for GF-multiplication operation. The first one is to use a large 2D table with size 256 × 256, i.e., 64 KB, to store the results of all possible GF-multiplication. As shown in Fig. 2, this algorithm requires a single memory read. The 2D table can be stored in either global memory or texture memory. Their performance turns out to be very different, and will be presented in Sec. IV. The second one is to use two small tables: a logarithmic table and an exponential table. As shown in Fig. 3, this algorithm requires three memory reads and one addition operation. Although the sizes of logarithmic table and exponential table are small enough to be stored in shared memory, but doing so will leave less shared memory for storing the data and the overall performance will be affected. To alleviate this problem, we propose to store the two tables in texture memory that has on-chip cache which can help to hide part of the memory latency and also reduce the memory bandwidth demand.

```
Algorithm 1: Multiplication in GF(2^8)
Input: x, y, GF[ ][ ]
Output: xy
1. int Mul (uchar x, uchar y)
2. {
3.     return GF[x][y] ;
4. }
```

Figure 2. Multiplication in GF(2^8) based on a single 2D table

```
Algorithm 2: Multiplication in GF(2^8)
Input: x, y, LOG[ ], EXP[ ]
Output: xy
1. int Mul (uchar x, uchar y)
2. {
3.     if ( x ==0 || y == 0)
4.         return 0 ;
5.     temp1 = LOG[x];
6.     temp2 = LOG[y];
7.     return EXP[temp1 + temp2];
8. }
```

Figure 3. Multiplication in GF(2^8) based on logarithmic and exponential tables

When computing an encoded block, the n original blocks should be first transferred from the host (CPU) memory to the GPU global memory. After the encoding process, the encoded block will be transferred back to host memory. Nonetheless, the use of the GPU global memory is expensive in terms of access time (hundreds of GPU cycles) which could be the bottleneck of the entire process. In contrast, the shared memory has much faster access time, but with much smaller capacity. It is obvious that a coefficient vector can be easily stored in the shared memory, given its small size (normally hundreds of bytes). However, the data blocks are too large to fit in the shared memory. To this end, we divide the $n \times m$ data matrix B into BLOCKs $(B'_{1,1}, B'_{1,2}, ..., B'_{2,1}, B'_{2,2}, ..., B'_{s,t})$ where $s = n/k$ and $t = m/k$, such that each BLOCK is a small $k \times k$ matrix that can be stored in the shared memory. Correspondingly, the coefficient vector c is divided into sub-vectors of size $1 \times k$, $(c'_1, c'_2, ..., c'_s)$. With this design, we can utilize the parallelism nature of the GPU. We divide the $C \times B$ operation into multiple smaller tasks, each of which computes a partial encoded block $e'_j = c'_j \times B'_{j,l}$, where $1 \le j \le s$ and $1 \le l \le t$. The actual encoded block can be obtained by concatenating $\sum_{j=1}^{s} e'_{j,l}$ for all l. The concept of this algorithm is illustrated in Fig. 4.

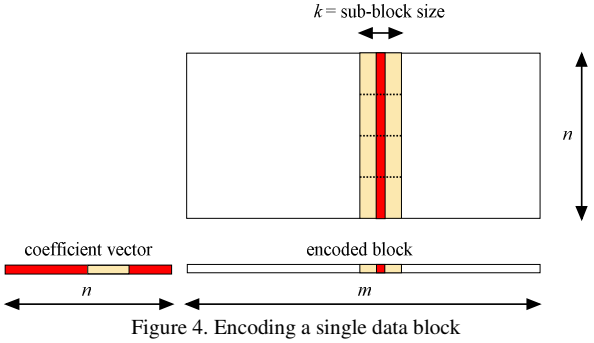


Figure 4. Encoding a single data block

© 2009 ACADEMY PUBLISHER

Naturally, we can assign each task, i.e., the computing of a BLOCK, to one thread block. The conventional implementation of the encoding process produces one codeword at a time following Eq. 1. Since the access time of a 1-byte ***char*** is the same as that of a 4-byte ***int*** in CUDA, we choose to use ***int*** to represent our data for faster table lookup and data fetching. In other words, each thread in the thread block computes 4 codewords that are stored in one 32-bit word. Without loss of the generality, we assume that n and m are multiples of k, and k is a multiple of 4. The size of a BLOCK is, therefore, limited by the number of threads supported by a thread block, which is 512 in the current CUDA model. The pseudo code of the kernel function is shown in Fig. 5.

```
Algorithm 3: Encoding a single block
1.  sum = 0;
2.  ty = threadIdx.y;
3.  for ( j=0; j < s; j++) {
4.     __shared__ int Bs[k][k];
5.     __shared__ uchar Cs[k];
6.     Load corresponding elements to Bs and Cs;
7.     uchar* pb, * psum = (uchar*)sum;
8.     __syncthreads( );
9.     for ( i = 0; i < k; i++){
10.       pb = (uchar*)& Bs[i][ty];
11.       psum[0]= XOR( sum, Mul(Cs[i], pb[0]) );
12.       psum[1]= XOR( sum, Mul(Cs[i], pb[1]) );
13.       psum[2]= XOR( sum, Mul(Cs[i], pb[2]) );
14.       psum[3]= XOR( sum, Mul(Cs[i], pb[3]) );
15.    }
16.    __syncthreads( );
17. }
18. output sum;
```

Figure 5. Pseudo kernel codes for encoding a single data block

From Eq. 1, it is obvious that encoding a single codeword requires n GF-multiplications and n-1 GF-additions. Overall, it takes $\Theta(n)$ time to encode a single codeword from n original blocks. For Algorithm 1, each GF-multiplication requires a single memory access. For Algorithm 2, each GF-multiplication requires 3 table lookups, thus, 3 texture memory access. The on-chip texture cache is much faster than off-chip graphic memories, however. Next, we will further explore ways to optimize the coding throughput.

b. Batched encoding

So far, encoding a single block requires m/k thread blocks, where each has $k/4$ threads, i.e., a total of $m/4$ threads, each of which computes 4 codewords in the encoded block. Today's GPU requires tens of thousands of threads in order to fully exploit its computing power and memory bandwidth. Hence the GPU may not be fully utilized when m is small. This section demonstrates how to take full advantages of the powerful GPU processor by computing encoded blocks in batch.

Let p be the batch size which is defined as the number of encoded blocks to be produced per time; thus, the p coefficient vectors form a $p \times n$ coefficient matrix. We employ tiled matrix multiplication, an optimization technique [3] [16], to bring the parallelism of the GPU to the next level. The encoded data matrix is divided into a set of square sub-matrices of size $k \times k$, $(E'_{1,1}, E'_{1,2}, ..., E'_{2,1}, E'_{2,2}, ..., E'_{s,t})$ where $s = p/k$ and $t = m/k$. Without loss of generality, we assume that n, m, and p are all multiples of k. Correspondingly, the coefficient matrix is divided into sub-matrices of size $k \times n$, $(C'_1, C'_2, ..., C'_s)$. The original data matrix is divided into sub-matrices of size $n \times k$, $(B'_1, B'_2, ..., B'_t)$. The same divide-and-conquer approach as in Algorithm 3 is applied here: each sub-matrix in E is computed as $E'_{i,j} = C'_i \times B'_j$, where $1 \le i \le s$ and $1 \le j \le t$. The concept of this batched encoding algorithm is illustrated in Fig. 6.

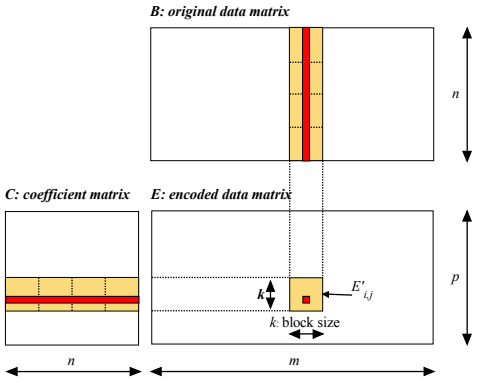


Figure 6. Batched encoding

To utilize the parallelism nature of GPU, we let each thread block be responsible for the calculation of one sub-matrix $E'_{i,j}$. Since there might not be enough shared memory to hold the two rectangular sub-matrices, C'_i and B'_j, the computation of $E'_{i,j}$ is done through another level of divide-and-conquer mechanism. Both C'_i and B'_j are further divided into $k \times k$ sub-matrices: $(C'_{i,1}, C'_{i,2}, ..., C'_{i,r})$ and $(B'_{j,1}, B'_{j,2}, ..., B'_{j,r})$, respectively, where $r = n/k$. In each round of the calculation, we load the two k x k sub-matrices into the shared memory, and then compute $D_l = C'_{i,l} \times B'_{j,l}$, where $1 \le l \le r$. After r rounds, we can obtain $E'_{i,j} = \sum_{l=1}^{r} D_l$. The value of k is carefully selected such that the two $k \times k$ sub-matrices can be loaded into the shared memory. Another consideration is that k should be a multiple of 16 so that the number of threads per thread block is a multiple of the *warp* size 32. In a thread block, each thread is responsible for computing 4 matrix elements in a row, given that an integer is used to store 4 codewords. The final pseudo kernel code for batched encoding is given in Fig. 7.

With batched encoding, we increase the number of thread blocks to pm/k^2 and the number of threads in each block to $k^2/4$, which allows us to take full advantages of the parallelism of the GPU.

B. Decoding

As mentioned in Sec. II, the decoding process is to solve a set of linear equations after receiving n linearly independent encoded blocks (e_1, e_2, …, e_n). These n

© 2009 ACADEMY PUBLISHER

coded blocks form an $n \times m$ matrix, E, in which each row corresponds to one encoded block. The n coefficient vectors form an $n \times n$ coefficient matrix C, in which each row corresponds to the coefficients of one encoded block. The decoding process is to recover the original blocks In other words, we have $B = C^{-1} \times E = (b_1, b_2, \ldots, b_n)$, where C^{-1} is the inverse of the coefficient matrix C. Let first assume that C^{-1} is given, then the decoding process is the same as the matrix multiplication, which can be implemented in the same way as the batched encoding progress presented in Algorithm 4 by replacing p with n.

```
Algorithm 4: Batched encoding
1.  uchar sum[4] = {0};
2.  ty = threadIdx.y;
3.  for (j = 0; j < s; j++) {
4.    __shared__ unsigned int Bs[k][k];
5.    __shared__ unsigned int Cs[k][k];
6.    Load corresponding elements to Bs and Cs;
7.    uchar* pb, * pc, * psum;
8.    __syncthreads();
9.    for (i = 0; i < k; i++) {
10.     pb = (uchar*)& Bs[i][tx];
11.     pc = (uchar*)& Cs[ty][i];
12.     for (m = 0; m < 4; m++) {
13.       psum =(uchar*)sum[m];
14.       psum[0]= XOR( psum[0], Mul(pc[m], pb[0]) );
15.       psum[1]= XOR( psum[1], Mul(pc[m], pb[1]) );
16.       psum[2]= XOR( psum[2], Mul(pc[m], pb[2]) );
17.       psum[3]= XOR( psum[3], Mul(pc[m], pb[3]) );
18.     }
19.   }
20.   __syncthreads();
21. }
22. output sum;
```

Figure 7. Pseudo kernel codes for batched encoding

If the time used to decode n data blocks is t seconds and each block has m bytes, then the decoding throughput equals mn/t bytes per second. We divide the whole decoding process into two steps: matrix inversion and matrix multiplication. The decoding time t then equals $t_{inverse} + t_{mul} + t_o$, where $t_{inverse}$ is the time for matrix inverse, t_{mul} is the time for matrix multiplication, and t_o is the system overhead time which mainly includes kernel invocation time and the time of data transfer between CPU and GPU. Given the high throughput of PCI Express interface, t_o is several orders less than $t_{inverse}$ and t_{mul} for practical settings. Since matrix inversion takes $O(n^3)$ operations and matrix multiplication takes $O(n^2m)$ operations, the time for matrix inverse can be well amortized when m is much greater than n. However, for small values of m, $t_{inverse}$ will dominate t, given the fact that the $O(n^2m)$ operations will be executed on hundreds of GPU cores in parallel. Therefore it is of particular importance to minimize $t_{inverse}$.

Existing parallel algorithms for matrix inverse are not suitable for the CUDA platform, mainly because CUDA does not provide any direct synchronization methods between threads that belong to different thread blocks. There are two possible approaches: (1) to use a single thread block; and (2) to use multiple thread blocks, and use the CPU to synchronize the thread blocks. In the first approach, only a small portion of the GPU power is utilized; while in the second approach, the frequent kernel function calls and memory copies introduce excessive overhead.

Giving the above challenges, we propose to utilize the multiple cores available in contemporary multi-core CPUs for matrix inverse. We implemented a parallel matrix inverse algorithm by OPENMP that utilizes all CPU cores to compute the inverse of a matrix. The pseudocode of our algorithm is shown in Fig. 8. Our experimental results show that, for moderate size of n (e.g., no more than 512), it is advantageous to use the multi-core CPUs to perform the matrix inversion, and then use the GPU to perform the matrix multiplication. It will be worthwhile to speedup the process of matrix inversion in GF(2^8) by using GPUs for large n, and we leave this to our future research.

```
Algorithm 5: Matrix inverse
1.  int invert(uchar *mat, uchar *inv, int sz)
2.  {
3.    int i, j, k, max_val, max_ind;
4.    uchar tmp;
5.    i = j = 0;
6.    while (i < sz && j < sz) {
7.      if (max_val != 0) {
8.        if (max_ind != i ) {
9.  #pragma omp parallel for
10.         Switch rows i and max_ind;
11.       } // end of if
12. #pragma unroll
13.       Divide row i by max_val;
14. #pragma omp parallel for private(tmp)
15.       for ( k = 0; k < sz; k++) {
16. #pragma unroll
17.         Perform elementary row operation;
18.       } // end of for
19.       i++;
20.     } // end of if
21.     j++;
22.   } // end of while
23.   return i;
24. }
```

Figure 8. Pseudocode for matrix inverse on multi-core CPU

IV. Experimental Results

We have implemented the network coding in GF(2^8) using CUDA and tested it on three different GPUs: (1) GeForce GTX280; (2) GeForce GTX260; and (3) GeForce 9800. The parameters of the three GPUs used in our experiments are summarized in Table I. The host computer is equipped with a 2.4GHz Intel Quad-core CPU Q6600.

TABLE I. Parameters of GPUs used in our experiments

GPU NAME	GTX280	GTX260	GeForce 9800
Number of SPs	240	192	128
Processor clock	1.30 GHz	1.24 GHz	1.80 GHz[1]
Memory bandwidth	141 GB/s	112 GB/s	70 GB/s
Memory amount	1 GB	896 MB	512 MB

The following general optimization principles have been applied in our implementation to speedup the coding throughput:

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

© 2009 ACADEMY PUBLISHER

- To use a large number of thread blocks and a large number of threads in each thread block.
- To use texture cache to decrease the demand of memory bandwidth.
- To exploit coalesced memory accesses when accessing global memory.
- To use loop unrolling to reduce loop overhead. This applies to both GPU optimization and CPU optimization.

Despite all the detailed attentions paid to memory access and thread management, we chose to focus on the measurement of the encoding and decoding throughputs, i.e., the number of bytes coded/decoded per second, as the block size and the number of blocks varies. We seek to identify the maximum performance that could be achieved on commodity GPUs. All experiments have been run for a number of times and the average results are presented.

A. *Encoding Performance*

We first compare the performance of algorithm 1 and algorithm 2. Algorithm 1 uses a single 2D table for GF-multiplications and requires one memory access for each multiplication; Algorithm 2 uses two small tables, i.e., logarithmic table and exponential table, and requires three memory accesses for each multiplication. The large 2D table can be stored in global memory or texture memory. The best encoding throughput of algorithm 2 obtained on GTX 280 is shown in Fig. 9 (a) and (b), for the case of global memory and texture memory, respectively. Notice that these results are obtained when batched encoding method (i.e., Algorithm 4) is used. When the 2D table is stored in global memory, the encoding throughput is very poor: only 30 MB/s for n = 128. This is because the table look-up access has a random pattern and hence the access to global memory cannot be coalesced. The high memory bandwidth of GPU can only be achieved with coalesce accesses. When we utilize texture memory to store the table, the encoding throughput becomes very promising: 250 MB/s for n = 128. This is because GPUs have on-chip cache for texture memory. Although the on-chip cache is small in size, it can still help to reduce the memory bandwidth demand significantly. Another benefit of using texture memory is that it does not require coalesced access pattern to achieve good performance. Nevertheless, algorithm 2 turns out to be the best solution for all the GPUs we have tested. It stores the two small tables in texture memory which can be fully stored into on-chip texture cache. Although each multiplication operation takes three memory accesses, our experiment results show that it still outperforms the single-table approach by 30%. We achieved 325 MB/s of encoding throughput on GTX 280 for n = 128, which is a new record to the best of our knowledge. For comparison, the *Nuclei* system achieved 66.9 MB/s of encoding throughput on Geforce 8800 [23], which can be approximately matched to about 124 MB/s on GTX 280 (i.e., 66.9 MB/s × 240 × 1.3GHz / (112 × 1.5 GHz)) when considering the number of cores and the working frequencies. In the following, we will focus on the experimental results that use algorithm 3 for GF-multiplications.

The encoding throughput drops as n increases, since bigger n incurs more memory references. On the other hand, the throughput grows as each block gets larger. The reason is that larger block size leads to more concurrent threads. We need a huge number of threads to fully utilize computing power of the GUP and also to hide the global memory latency. The encoding throughput is relatively low for smaller block sizes, due to lack of parallelism on the GPU. Our batched encoding mechanism, i.e., algorithm 4, is specially designed to fully exploit the parallel nature of the GPU.

To evaluate the performance of batched encoding, we first fixed block size m at 16384 bytes and varied the batch size p from 16 to 2048, and the results are shown in Fig. 10 (a)-(c) for the three GPUs, respectively. It is obvious that the batched encoding mechanism offers significant gain in throughput. One interesting result is that, GeForce 9800 has a much higher peak performance than GTX 260, but it performs worse than GTX 260 when the batch size is less than 128. This is because GeForce 9800 has a much higher process clock rate but less number of SPs.

We then fixed batch size p at 512 and 2048, and varied the block size m from 1024 bytes to 32768 bytes, so as to investigate the impact of block size on the encoding throughput. The results are shown in Fig. 11 (a)-(c). We observe that GeForce 9800 gets saturated for a small block size; GTX 260 requires a moderate block size to saturate; and GTX 280 requires a very large block size to saturate.

B. *Decoding Performance*

Next, we present the experimental results of decoding throughput, which is defined as the number of decoded bytes per second. The matrix inverse time, $t_{inverse}$, is shown in Table II. We also implanted a single thread version of matrix inverse for comparison purpose. The speedup of the OPENMP version over the single thread version increases from 2.9 to 3.75 when n increase from 128 to 512.

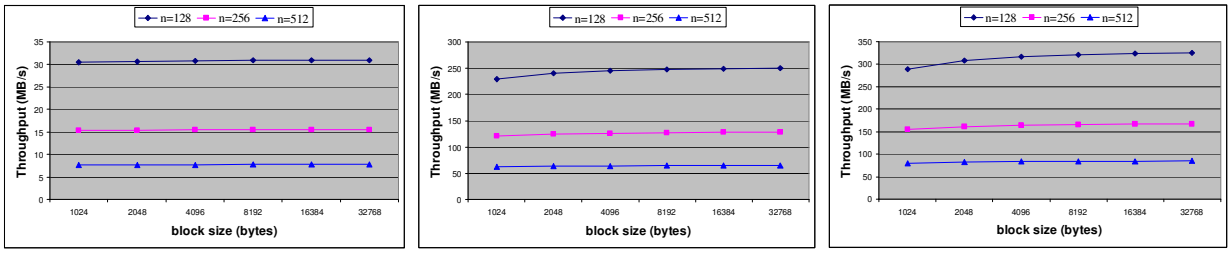
TABLE II. Matrix Inverse Time

n	**128**	**256**	**512**
Single-core	8.32ms	67.20ms	522.00ms
Multi-core	2.86ms	18.93ms	139.08ms

The decoding throughput of single thread CPU version is shown in Fig. 12. We observed that the number of blocks, n, and the block size, m, have similar impact on the decoding throughput as they do on the encoding throughput. As expected, due to the complexity in matrix inverse, the decoding throughput degrades significantly when n is large. For example, when n = 512, the highest decoding throughput we obtained is only 22.9 MB/s on GTX280. The performance is even worse when m is small.

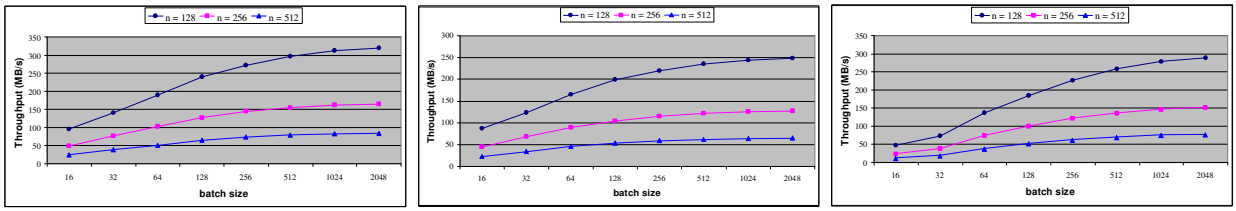
We then enabled our multi-threaded approach to fully utilize the computing power of the quad-core CPU. This approach improves the decoding throughput significantly.

© 2009 ACADEMY PUBLISHER



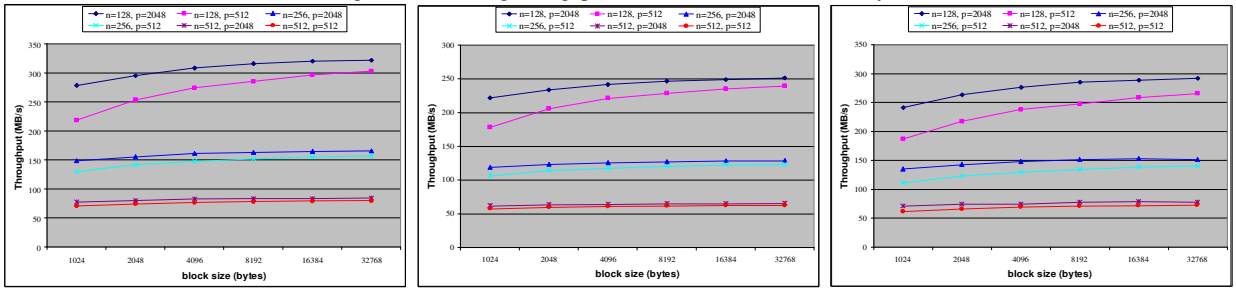
(a) Algorithm 1: single table in global mem. (b) Algorithm 1: single table in texture mem. (c) Algorithm 2: two tables in texture mem.

Figure 9. Comparison of algorithm 1 and 2: encoding throughput on GTX 280



(a) GTX 280 (b) GTX 260 (c) GeGeForce 9800

Figure 10. Encoding throughput versus batch size, m = 16384 bytes



(a) GTX 280 (b) GTX 260 (c) GeForce 9800

Figure 11. Encoding throughput versus block size

Fig. 13 shows that the decoding throughput of GTX280 is 181 MB/s, 87 MB/s, and 34 MB/s, for m = 16384 and n = 128, 256, 512, respectively. For comparison purpose, the *Nuclei* system achieves 47 MB/s, 26 MB/s, and 10 MB/s, for m = 16384 and n = 128, 256, 512, respectively [23]. Notice that increasing m will increase the decoding throughput.

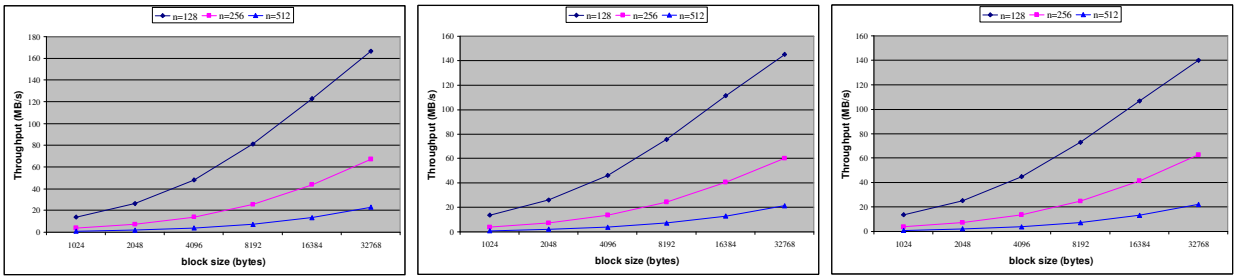
V. CONCLUSIONS

In this paper, we proposed a high-performance random network coding implementation on GPUs. More specifically, we design a massively parallel implementation of random linear network coding using the CUDA programming model. We are able to achieve a significant speedup over existing solutions in terms of encoding and decoding throughput. For the decoding process, we developed a highly optimized matrix inverse algorithm for the multi-core CPU, in combination with the parallel model for data block decoding on the GPU. With this design, the achievable decoding throughput is comparable to the encoding throughput; hence, the decoding process will no longer be the bottleneck. By exploiting the computing power of many-core GPU and multi-core CPU, we show that the computational complexity of random network coding will no further be the performance bottleneck in practical applications, even when coding a very large number of blocks.

REFERENCES

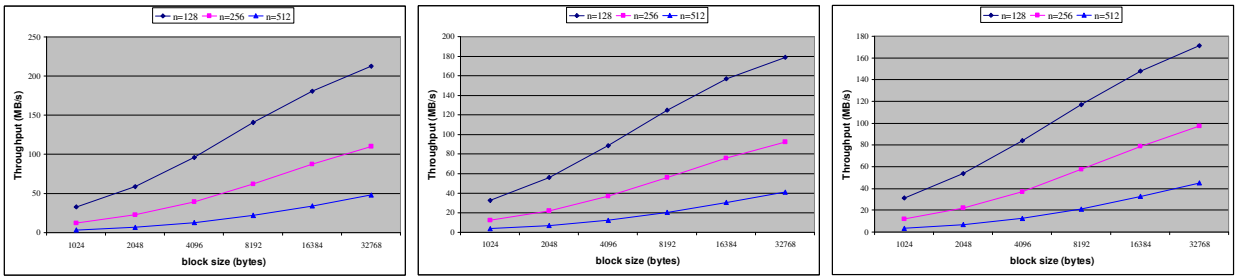
[1] NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html
[2] AMD CTM Guide: Technical Reference Manual. 2006.
[3] NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 2.0beta2, Jun. 2008.
[4] Sanders, P., Egner, S., and Tolhuizen, L. Polynomial Time Algorithm for Network Information Flow. In Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2003), Jun. 2003.
[5] Ahlswede, R., Cai, N., Li S. R., and Yeung, R. W. 2000. Network information flow. IEEE Transactions on Information Theory, 46(4), July 2000, 1204-1216.
[6] Koetter, R. and Medard, M. 2003. An algebraic approach to network coding. IEEE/ACM Transactions on Networking, 11(5), Oct. 2003. 782-795.
[7] Ho, T., Koetter. R., Médard, M., Karger, D.R. and Effros, M. 2003. The benefits of coding over routing in a randomized setting. In Proceedings of IEEE ISIT, 2003.
[8] Li, S.-Y.R., Yueng, R.W., and Cai, N. 2003. Linear network coding. IEEE Transactions on Information Theory, vol. 49, 2003. 371-381.
[9] Gkantsidis, C. and Rodriguez, P. 2005. Network coding for large scale content distribution. In Proceedings of IEEE INFOCOM 2005.
[10] Wang, M. and Li, B. 2006. How practical is network coding? In Proceedings of the 14th International Workshop on Quality of Service (IWQoS), 2006, 274-278.

© 2009 ACADEMY PUBLISHER



(a) GTX 280 (b) GTX 260 (c) GeForce 9800

Figure 12. Decoding throughput: single thread CPU + GPU



(a) GTX 280 (b) GTX 260 (c) GeForce 9800

Figure 13. Decoding throughput: multi-core CPU + GPU

[11] Shojania, H. and Li, B. 2007. Parallelized progressive network coding with hardware acceleration. In Proceedings of the 15th International Workshop on Quality of Service (IWQoS), 2007.

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

[13] Wang, M. and Li, B. 2007. R2: random push with random network coding in live peer-to-peer streaming. In IEEE Journal on Selected Areas in Communications, Dec. 2007, 1655-1666.

[14] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. 2007. A survey of general-purpose computation on graphics hardware. Computer Graphics forum, 26(1), 2007, 80-113.

[15] Manavski, S. A. 2007. Cuda compatible GPU as an efficient hardware accelerator for AES cryptography. In Proceedings of IEEE International Conference on Signal Processing and Communication, Nov. 2007, pp.65-68.

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

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

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

[19] Silberstein, M., Geiger, D., Schuster, A., Patney, A., Owens, J. D. Efficient computation of sum-products on GPUs through software-managed cache. In Proceedings of the 22nd ACM International Conference on Supercomputing, Jun. 2008.

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

[21] Ho, T., Medard, M., Shi, J., Effros, M., and Karger, D., On Randomized Network Coding, in Proceedings of the 41st Allerton Conference on Communication, Control, and Computing, Oct. 2003.

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

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

[24] Chu, X., Zhao, K., and Wang, M. Practical Random Linear Network Coding on GPUs. In Proceedings of IFIP Networking'09, May 2009.

Xiaowen Chu received his B.Eng. degree in the Computer Science from Tsinghua University, P. R. China, in 1999, and the Ph.D. degree in the Computer Science from the Hong Kong University of Science and Technology in 2003. He is currently an Assistant professor in the Department of Computer Science, Hong Kong Baptist University. His research interests include distributed and parallel computing, Wireless Networks, and Optical Networks.

Kaiyong Zhao received his B.Eng. degree in the Aircraft Design and Technology from Beijing Institute of Technology (BIT), P. R. China, in 2005. He is currently an Mphil student in the Computer Science Department of Hong Kong Baptist University. His research focuses on GPU computing.

Mea Wang received her Bachelor of Computer Science (Honours) degree from the Department of Computer Science, University of Manitoba, Canada, in 2002. She received her Master of Applied Science and Ph.D. degrees from the Department of Electrical and Computer Engineering at the University of Toronto, Canada, in 2004 and 2008, respectively. She is currently an Assistant Professor in the Department of Computer Science at the University of Calgary. Her research interests include peer-to-peer networking, multimedia networking, and networking system design and development.

© 2009 ACADEMY PUBLISHER