Pushing the Envelope: Extreme Network Coding on the GPU

Hassan Shojania, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

Abstract

While it is well known that network coding achieves optimal flow rates in multicast sessions, its potential for practical use has remained to be a question, due to its high computational complexity. With GPU computing gaining momentum as a result of increased hardware capabilities and improved programmability, we show in this paper how the GPU can be used to improve network coding performance dramatically. Our previous work presented the first attempt in the literature to maximize the performance of network coding by taking advantage of not only multi-core CPUs, but also hundreds of computing cores in commodity off-the-shelf Graphics Processing Units (GPU). This paper represents another step forward, and presents a new array of GPU-based algorithms that improve network encoding by a factor of 2.2*, and network decoding by a factor of* 2.7 *to* 27.6 *across a range of practical configurations. With just a single NVIDIA GTX 280 GPU, our implementation of GPU-based network encoding outperforms an 8-core Intel Xeon server by a margin of at least* 4.3 *to* 1 *in all practical test cases, and over* 3000 *peers can be served at high-quality video rates if network coding is used in a streaming server. With* 128 *blocks, for example, coding rates up to* 294 *MB/second can be achieved with a variety of block sizes*[1].

1. Introduction

First introduced by Ahlswede *et al.* [1] in information theory, *network coding* has received significant research attention in the networking community. The fundamental advantage of network coding hinges upon the *coding* capabilities of intermediate nodes, in addition to forwarding and replicating incoming messages. In theory, network coding helps to alleviate competition among flows at the bottleneck, thus improving session throughput in general. Wu *et al.* [2] and Gkantsidis *et al.* [3] have both proposed to apply random network coding, first proposed in [4], in practical content distribution systems. Extensive simulation studies have shown that network coding delivers close to theoretically optimal performance levels.

Unfortunately, to date, there has been no commercial real-world applications or protocols reported in the literature that take advantage of the power of network coding. We believe that the main cause of this observation — and the main disadvantage of network coding — is the high computational complexity of randomized network coding with random linear codes, especially as the number of blocks to be coded scales up. We believe that it is critically important to optimize the implementation of random linear codes such that its real-world coding performance is maximized on modern off-the-shelf hardware platforms, such as media streaming servers.

Towards this objective, our previous work [5] has shown an accelerated multi-threaded implementation of network coding, that takes advantage of both multiple CPU cores with aggressive multi-threading, and SSE2 and AltiVec SIMD vector instructions on x86 and PowerPC processors as well. Further, our previous work on *Nuclei*, the first network coding implementation on *Graphics Processing Units (GPU)* in the literature [6], achieved encoding rates of up to 114 MB/second by combining 8-core Intel Xeon CPUs and a mainstream NVIDIA GeForce 8800 GT GPU.

This paper represents another substantial step forward, and presents a new array of optimization techniques and algorithms to further improve the performance of GPU-based network coding by a significant margin beyond our previous work. With our new algorithms, A single NVIDIA GeForce GTX 280 performing network coding at 128 blocks achieves encoding rates up to 294 MB/s and decoding rates up to 254 MB/s, far beyond the computation bandwidth required to saturate a Gigabit Ethernet interface on streaming servers. At such high rates, we argue that GPU alone is sufficient to deploy network coding on streaming servers setting the CPU cores free for other CPU-intensive tasks.

We have made the following new observations. *First*, the NVIDIA GeForce GTX 280, featuring 240 cores, far outperforms our CPU-based implementation on a Dual Quad-core 2.8 GHz Intel Xeon server (8-core Mac Pro). *Second*, unlike the CPU, the GPU can benefit from a novel and highly optimized table-based encoding technique that outperforms the loop-based encoding technique employed in [6] by a factor of 2.2. *Third*, by parallel decoding of multiple segments, the performance of GPU-based network decoding can be improved by a factor of 2.7 to 27.6, across a range of practical configurations.

The remainder of this paper is organized as follows. Sec. 2 discusses related work. Sec. 3 provides an overview of both random network coding and GPU computing. Sec. 4 presents

1. The completion of this research was made possible thanks to the NSERC Strategic Grant STPGP 321948-05 and a research grant from the Bell Canada University Laboratories R&D Program.

the basis of parallel coding on GPUs along with network coding performance results for the GTX 280. Sec. 5 presents and evaluates a variety of novel techniques to maximize the real-world network coding performance with GPUs. Finally, Sec. 6 concludes the paper.

2. Related Work

Ho *et al.* [4] has been the first to propose the concept of *randomized network coding* using random linear codes, in which an intermediate node transmits on each outgoing link a linear combination of incoming messages, specified by independently and randomly chosen *code coefficients* over some finite field. Wu *et al.* [2] have concluded that randomized network coding can be designed to be "robust to random packet loss, delay, as well as any changes in network topology and capacity." Ghantsidis *et al.* [3] has further proposed that randomized network coding can be used for bulk content distribution, and has demonstrated the feasibility of network coding [7]. The work has concluded that "network coding incurs little overhead, both in terms of CPU and I/O, and it results in smooth and fast downloads." However, a small number of blocks per segment has been used, limiting the benefits of network coding as it is only performed within each segment.

Theoretically, the computational complexity of random linear codes has been well known: it has been a driving force towards the development of more efficient codes in content distribution applications, including traditional Reed-solomon (RS) codes, *fountain codes* [8], and more recently, *chunked codes* [9], however, all of them come with their own drawbacks.

While there is no doubt that more efficient codes exist, they may not be suitable for randomized network coding in a practical setting. In contrast, random linear codes are simple, effective, and can be recoded without affecting the guarantee to decode. We believe that our work on a high-performance implementation of random linear codes may help academics and practitioners to realize the full potential of randomized network coding in a real-world setting. Our previous work [5] has evaluated the performance of our real-world implementation of network coding, taking advantage of multi-core CPUs and modern SIMD vector instruction sets, such as Intel SSE2. Our recent work [6] has provided the first network coding implementation on GPUs, where we provided a highly optimized Galois Field multiplication technique for GPUs, and achieved satisfactorily high encoding rates, especially when a NVIDIA GeForce 8800 GT GPU was employed in parallel with an 8-core Intel Xeon server. However, the decoding performance of the GPU was not satisfactory.

Though in [6] we have achieved a level of performance that has not been previously reported, this paper takes another step forward to improve the coding performance by significant margins, particularly decoding. With our new results, we argue that a single GPU can achieve a sufficiently high level of performance to satisfy the needs of most application scenarios. Further, for network coding applications, the price/performance ratio of GPUs is far superior to multi-core servers. For the exceptionally demanding applications, multiple GPUs can be employed in parallel. By setting free the CPUs from computation-intensive network coding, a far more reliable system with a better performance guarantee can be deployed, which is especially beneficial to media streaming servers.

3. Background

Modern GPUs have gradually evolved from specialized engines operating on fixed pixels and vertex data types, into programmable parallel processors with enormous computing power [10]. NVIDIA's Tesla GPU architecture, introduced in November 2006 and now employed in a wide range of professional and consumer GPU products, is the first such GPU architecture that enables high-performance parallel computing applications, written in the C language using the Compute Unified Device Architecture (CUDA) programming model and development tools, and is now considered to be the "most ubiquitous" supercomputing platform.

The NVIDIA GeForce GTX 280 GPU, though retails at mainstream GPU pricing, is supported by the CUDA programming platform. Our performance evaluation of GPU-based network coding in this paper is based on the GTX 280 GPU, with 240 computing cores. Our design and implementation, however, can be used on any existing and future GPU that supports the CUDA programming platform. To facilitate our subsequent discussions in this paper, we refer the reader to [11] and also our previous work [6] for an overview of the Tesla GPU architecture and CUDA.

We now briefly introduce the basics of random network coding. With random linear codes, data to be disseminated is divided into n *blocks* $[b_1, b_2, \ldots, b_n]^T$, where each block b_i has a fixed number of bytes k (referred to as the block size). To code a new coded block x_j in network coding, a network node first independently and randomly chooses a set of coding coefficients $[c_{j1}, c_{j2}, \cdots, c_{jn}]$ in GF(2^8) Galois field, one for each received block (or each original block on the data source). It then produces one coded block x_j of k bytes:

$$x_j = \sum_{i=1}^{n} c_{ji} \cdot b_i \tag{1}$$

A peer decodes as soon as it has received n linearly independent coded blocks $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$. It first forms a $n \times n$ matrix $\mathbf{C}$, using the coefficients of each block b_i. Each row in $\mathbf{C}$ corresponds to the coefficients of one coded block. It then recovers the original blocks $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$ as:

$$\mathbf{b} = \mathbf{C}^{-1}\mathbf{x} \tag{2}$$

In this equation, it first needs to compute the inverse of $\mathbf{C}$, using Gaussian elimination. It then needs to multiply $\mathbf{C}^{-1}$ and $\mathbf{x}$, which takes $n^2 \cdot k$ multiplications of two bytes in GF(2^8). The inversion of $\mathbf{C}$ is only possible when its rows are linearly independent, *i.e.,* $\mathbf{C}$ is full rank.

GF(2^8) operations are routinely used in random linear codes within tight loops. Since addition in GF(2^8) is simply an *XOR* operation, it is important to optimize the implementation of multiplication on GF(2^8). Our baseline implementation takes advantage of the widely-used fast GF multiplication through logarithm and exponential tables similar to the traditional multiplication of large numbers. Fig. 1 shows a C function to multiply using three table references where $\log$ and $\exp$ reflect GF(2^8) logarithmic and exponential tables. Such a baseline implementation requires three memory reads and one addition for each multiplication.

```
byte baseline_gf_multiply(byte x, byte y)
{
    if (x == 0 || y == 0)
        return 0;
    return exp[log[x] + log[y]];
}
```

Figure 1. Table-based multiplication in GF(2^8).

We use *Gauss-Jordan elimination* to implement the decoding process, rather than the more traditional Gaussian elimination. Gauss-Jordan elimination transforms a matrix to its *reduced row-echelon form* (RREF), in which each row contains only zeros until the first nonzero element, which must be 1. The benefit of the reduced row-echelon form is that, once the matrix is reduced to an identity matrix, the result vector on the right of the equation constitutes the solution, without any additional needs of decoding. In addition, if a received coded block is linearly dependent with existing blocks, the Gauss-Jordan elimination process will lead to a row of all zeros, in which case this coded block can be immediately discarded, and there are no explicit linear dependence checks required. The reader is referred to [5] and [6] for more detailed discussions of the decoding process.

4. Network Coding with the GTX 280

In this section, we present the basis of parallel network coding on GPUs, its challenges, and our solutions. Then we evaluate GPU-based network coding on the GTX 280, and compare the results against the best performing results reported in our previous work [6].

4.1. Performance Bottlenecks in Network Coding

Random network coding suffers from two major performance bottlenecks. First, a table-based multiplication in GF(2^8) is a costly operation. Second, the multiplication and addition operations are performed in tight loops over rows of coefficients and coded blocks, each of n and k bytes, respectively. Each row operation is performed through a series of byte-length GF(2^8) operations, since table-based GF(2^8) multiplication is not easily scalable to a higher granularity than the byte level.

To address such a performance bottleneck, we first proposed in [5] to revisit the basics by performing the multiplication on-the-fly using a loop-based approach in Rijndael's finite field, rather than using traditional $\log / \exp$ tables. Although the basic loop-based multiplication takes longer to perform (up to 8 iterations), it lends itself better to a parallel implementation that takes advantage of vector instructions in order to operate on wider chunks of elements from a matrix row at the same time. The loop-based equivalent of the table-based multiplication in Fig. 1 resembles a regular hand multiplication (see Fig. 3 of [6]) by looking into the lower bit of x and adding y at each iteration (addition is equivalent to *XOR* in GF(2^8)).

With such a loop-based approach, we took advantage of SSE2 and AltiVec SIMD (single-instruction, multiple data) vector instruction sets on Intel and IBM PowerPC processor families to perform GF(2^8) multiplication on 16 byte-long units of each row, rather than single-byte units [5]. In [6], we followed the same principle for GPUs but faced two major challenges. First, unlike modern CPU cores with 128-bit registers and execution units, current CUDA-enabled GPUs have plain 32-bit registers and arithmetic units. As a result, we can only go as far as single byte by 4-byte word GF-multiplication. Second, GPU cores lack the sophisticated SIMD instructions that allow test and manipulation of individual bytes of a word. This leads to longer and less efficient code.

Our previous work [6] showed that it is possible to use the GPU and still achieve accelerated GF-multiplication using a loop-based approach, despite the lack of CPU-like vector instructions. A number of optimization techniques, including hand-optimization of the PTX assembly code, were used for efficiently employing the 112 cores of 8800 GT. It was reaffirmed that the GPU's advantage over CPUs is their ability to schedule thousands of lightweight threads with almost zero overhead in hardware, to hide stalls in the processing cores due to data dependency and memory access.

4.2. Task Partitioning on the GTX 280

Although current CUDA-enabled GPUs lack the wide registers and arithmetic-logic units (ALUs) of the modern CPUs, they have many processing cores (*e.g.,* 240 for the GTX 280), which run in parallel and can achieve the same functionality as wide execution units. In the following sections, we briefly overview our parallelization scheme of network encoding and decoding. These partitioning schemes are essentially the same as the ones used for 8800 GT in [6] but with some extra fine-tuning for the GTX 280. A good understanding of these schemes is necessary when we present our new techniques in Sec. 5.

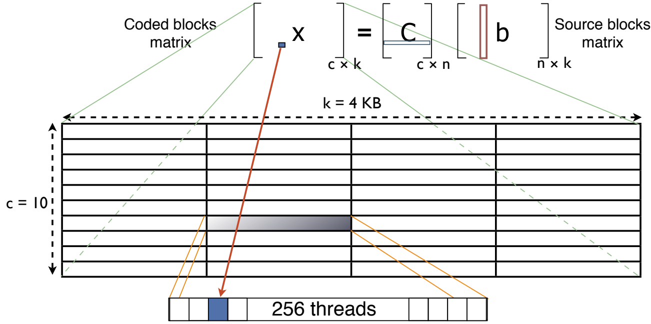

Figure 2. 10,000 GPU threads performing parallel network encoding for 10 coded blocks of each 4 KB.

4.2.1. *Partitioning parallel network encoding*. The process of random network encoding essentially consists of a matrix multiplication in the GF domain, as described in Sec. 3, with single-byte coefficients multiplied in each row of source blocks, and the results *XOR*-ed together to form a coded block. A parallel implementation of network encoding falls in the category of what is known as *embarrassingly parallel* problems, where a parallel implementation is possible with little communication and synchronization among parallel threads. Ignoring memory access to source blocks and coefficients, the performance of loop-based network encoding is only limited by the computational power of the hardware, since the encoding process of multiple coded blocks — and even different section of a coded block — can proceed in parallel by employing a large number of threads.

Fig. 2 shows how generation of 10 coded blocks, each of $k = 4$ KB, is assigned to GPU threads. Each element of the grid reflects a *thread block* consisting of 256 threads. Each thread of a thread block generates a 4-byte word of the coded block by performing the encoding operation according to Eq. (1) on n words from the source blocks using loop-based GF-multiplication. Effectively, each thread block generates 1 KB worth of coded data, *i.e.,* $10,000$ threads are used to generate 10 coded blocks.

With careful assignments of words to threads of each *thread warp*, we take advantage of **(1)** the memory broadcast feature [11] of the GTX 280 for coefficients load, **(2)** the memory coalescing [11] for loading source and storing coded blocks. Since most memory accesses of a thread warp, except coefficient reads, fall next to each other, such partitioning significantly reduces the number of accesses to the GPU memory.

4.2.2. *Partitioning parallel network decoding*. The decoding process has a higher computational complexity than encoding, as Gauss-Jordan elimination involves n^2 row operations on coefficient rows of length n and coded blocks of length k. However, the more critical issue is the smaller degree of parallelization inherent in the decoding process. Gauss-Jordan elimination requires the decoding of each coded block to start only after the decoding of the previous coded blocks is finished. This implies that the decoding process, unlike encoding, lends itself to parallelization only "within" the decoding of the current coded block, and not "across" the decoding of a number of coded blocks.

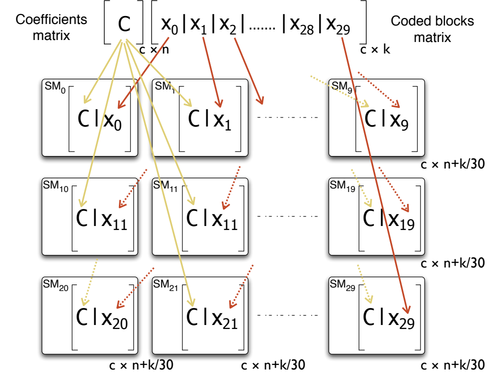

Figure 3. Parallel network decoding on 30 SMs (Stream Multiprocessors [11]) of GTX 280 with duplicate coefficient matrix and partitioned coded blocks matrix.

Such a lower degree of parallelism limits the performance gain of GPU-based decoding much more than the CPU-based implementation, since the GPU needs to run "thousands of threads" to be able to achieve its peak performance. In addition, threaded decoding of each coded block requires a synchronization point, for searching the first non-zero coefficient, which makes the decoding process a *coarse-grained* parallel program [6].

This synchronization point is a major obstacle in deep parallelization of decoding. CUDA's synchronization construct only works "within" threads of a thread block and does not support global synchronization among all GPU threads. To work around the required global synchronization, we divide the data portion of the coded block among all thread blocks, but give each thread block its own "private copy" of the coefficient row. We now can use CUDA's per thread block synchronization to perform the search for the first non-zero coefficient in each thread block. However, we do not wish to consume too much of our computing power on processing redundant coefficients, so we define a thread block to be as large as possible, employing only one thread block per each of the 30 SMs of the GTX 280. This effectively leads to decoding $n + k/30$ bytes of aggregate data by each thread block implying $(n + \frac{k}{30})/4$ GPU threads, each working on a 4-byte column.

This partitioning scheme is shown in Fig. 3, where SP_i of the GTX 280 performs the decoding on the aggregate $[\mathbf{C}|\mathbf{x}_i]$ matrix where $\mathbf{x}_i$ is the i-th partition of the coded block matrix $\mathbf{x}$. As we shall show in the next section, such parallelized decoding is still not able to fully take advantage of the GTX 280's computation power. We will revisit parallelized decoding in Sec. 5 with a new approach.

4.3. Evaluating the GTX 280

Now we evaluate the GTX 280 running our loop-based implementation of network coding on the GPU. GTX 280

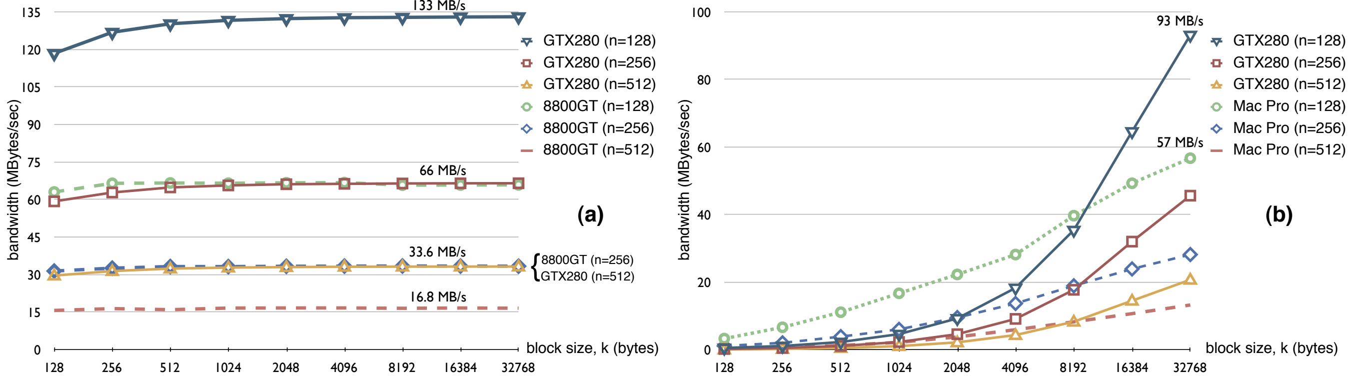

Figure 4. Coding bandwidth of GPU-based and CPU-based (8-threaded with SIMD acceleration) for network **(a)** encoding; and **(b)** decoding processes.

boasts 240 processing cores compared to 112 in the GeForce 8800 GT, but it runs at a slightly lower core frequency of 1458 MHz against 1500 MHz. With almost twice the amount of computing power of the 8800 GT, we can expect better coding performance over the 8800 GT. However, the main questions are that whether the coding bandwidth can scale up linearly, and whether computation and memory can interleave efficiently so that memory latency can be hidden. We use fully dense coding matrices as [6] with non-zero coefficients in all of our evaluations in this paper. The performance will be even higher with sparser matrices.

To evaluate the GTX 280, we use the testbeds in [6] as our benchmarks, namely a 8800 GT GPU and a 8-core Intel Xeon 2.8 GHz Mac Pro server (SIMD accelerated with 8 threads, one thread per CPU core). To have a readable graph, however, we only use the 8800 GT as the encoding benchmark and the Mac Pro as the decoding benchmark. Although the 8800 GT and the Mac Pro essentially achieve similar coding rates at $k = 16$ KB, 8800 GT consistently achieves better encoding performance, especially at smaller block sizes [6]. For decoding, the case is reversed and CPU-based decoding on the Mac Pro system defeats 8800 GT hands down (refer to [6] for a full performance comparison between the two).

We have tested a range of network coding configurations with 128 bytes to 32 KB per block, with 128, 256 and 512 blocks. The coding bandwidth of the GTX 280, in terms of MB per second, is shown in Fig. 4. Note that the encoding (decoding) bandwidth should be read as the total bytes of generated coded blocks (decoded source blocks) within one second with a network coding setup of (n, k).

Fig. 4(a) shows that encoding in GTX 280 achieves a rate almost twice of 8800 GT, a linear speedup, across all coding settings. This is not surprising as the encoding process is an *embarrassingly parallel* operation, and the fact that GTX 280's memory bandwidth is more than double of 8800 GT's (155 GB/s vs. 57.6 GB/s). As a result, the computation power of GPU still remains the performance bottleneck of our loop-based encoding scheme.

At a 133 MB/s encoding rate for the $n = 128$ setting, 4463 million GF-multiplications are executed every second. At an average 7 iterations per GF-multiplication in a random test benchmark as in [6] and each iteration taking an average of 10.5 instructions, the instruction rate will be 329 GIPS (Giga instructions per second). Our GTX 280's theoretical limit of 1080 GFLOPS translates to 360 GIPS. As a result, GF-multiplications alone (not considering the overhead associated with loop traversal, GPU kernel launch, etc.) effectively achieves 91% of the advertised computing power of GTX 280. This confirms that our encoding task partitioning scheme managed to hide memory latency very well. Similar calculations put the memory access rate at 20.9 GB/s (each word of the generated coded data requires $5 \times n + 4$ bytes of read and write), which is substantially lower than the theoretical memory limit of 155 GB/s.

With respect to decoding, as discussed in Sec. 4.2.2 and observed in Fig. 4(b), the decoding performance is generally lower for both GPU and CPU-based schemes because coded blocks are decoded serially. The GTX 280 performs better than 8800 GT in [6], by defeating the Mac Pro for blocks of 8 KB and larger. However, the CPU still performs better than the GTX 280 at smaller block sizes, because the GPU does not have sufficient data (small $k/30$) to launch a sufficient number of threads accordingly. In contrast, CPU's more efficient microarchitecture performs a better job even at small block sizes. As k increases, the performance of both GPU and CPU increases partially due to a lower overhead associated with the decoding of the coefficient matrix, proportional to the n/k ratio. The lack of parallelism in the decoding process prevents the GTX 280 to fulfill its almost twice computing advantage over the 8800 GT. For example, at $n = 128$, the decoding rate of GTX 280 achieves virtually the same performance as the 8800 GT (in a graph not shown here) up to a block size of 1024 bytes. From blocks of 2 KB to 16 KB length, the GTX 280 achieves a modest 5% to 38% gain over 8800 GT.

5. Pushing the Envelope of Network Coding

We now propose a number of new schemes leading to significant performance improvements in GPU-accelerated

network coding. A number of our proposed schemes also improves CPU-based coding.

5.1. Table-based Parallel Encoding

So far, we focused only on loop-based GF-multiplication. Would the table-based approach perform better with the GPU? We already know from [5] that this is not the case for CPU-based coding.

We now migrate the table-based GF-multiplication of Sec. 3 to the GPU. Exponential and logarithm tables are created on the CPU side once and then transferred to the GPU memory. Our experiment here follows the same partitioning scheme of Sec. 4.2.1. However, at each byte-by-word GF-multiplication, a similar table-lookup scheme as Fig. 1 will be invoked four times in a row.

Accessing $\log / \exp$ tables from GPU memory results in very poor performance. In an improved version, we load up the tables from the graphics memory into the on-chip *shared memory* of each GPU's SM at the start of the encoding process. Because all threads of a thread block can access the shared memory seamlessly, we effectively use the shared memory as a "managed L1 cache" shared among the active threads of a thread block with far less access latency than a memory fetch from the graphics memory. However, each SM has only 16 KB of such on-chip memory so the thread blocks have to be sized carefully to let many threads share the same $\log / \exp$ tables.

To improve the initial caching of the $\log / \exp$ tables, each thread of a thread block loads a 4-byte portion of the table in an optimized fashion, ensuring memory coalescing in threads of each thread warp. After the tables are loaded, all threads proceed as a regular encoding thread we employed in Sec. 4.2.1. This method has led to a substantial improvement over our first table-based scheme. Unfortunately, in an experiment at $n = 128$, such a fine-tuned table-based scheme still performs 26% worse than our loop-based approach.

5.1.1. ***Table-based GF-multiplication for streaming servers*.** Before completely deserting table-based GF-multiplication, we have performed a more in-depth investigation into how network coding can be employed in a streaming server. As our performance results from Fig. 4 indicate, the encoding performance is so high that it can be deployed in high-performance live or VoD streaming servers to serve hundreds of downstream peers or clients. As an example, consider the scenario of using a media segment size of 512 KB, with 128 blocks of 4 KB each. With a streaming rate of 768 Kbps that is typical for high quality video streams, each segment contains content that lasts 5.33 seconds, which is an acceptable buffering delay on the client side.

Operating at this setting, the coding bandwidth of 133 MB/s is sufficiently high to saturate a Gigabit Ethernet interface and serve up to 1385 downstream peers. In addition, when a media segment is ready to be encoded and served, it can be transferred and stored in the graphics memory on the GPU. 1024 MB memory on the GTX 280 is able to easily accommodate hundreds of such segments. Then, per request from the downstream peers, many coded blocks will be generated from a GPU-resident segment. For example, serving so many peers in a live video stream requires generating at least $177,333$ coded blocks from every video segment.

Revisiting our baseline table-based GF-multiplication of Fig. 1 and taking into account the fact that thousands of coded blocks are generated from the source blocks of each video segment, we can come up with a more efficient use of table-based GF-multiplication as suggested by the following algorithm: **(1)** As soon as a new video segment becomes available and transferred to the graphics memory, it will be transformed to the GF logarithmic domain by transforming every byte of its content. **(2)** Similarly, as soon as a new coefficient matrix, filled with random numbers, is generated by the GPU, it too will be transformed to the $\log$ domain. **(3)** In the actual encoding process, we execute a simpler GF-multiplication operation based on Fig. 5. Multiplication by 0 is now detected by checking the inputs against `0xff` since $\log(0)$ is equal to `0xff` in Galois field.

```
byte preprocessed_gf_multiply(byte log_x, log_y)
{
    if (log_x == 0xff || log_y == 0xff)
       return 0;
    return exp[log_x + log_y];
}
```

Figure 5. New table-based multiplication in GF(2^8) with inputs already in logarithmic domain.

Note that beside the improvement achieved through preprocessing the source blocks, *i.e.,* the video segment, the same preprocessing of the coefficient matrix is also a significant help. This is because every coded word generated by a thread of the encoding scheme of Fig. 1 fetches and multiplies a full coefficient row of n bytes, so a one-time preprocessing reduces the redundant transforms to the $\log$ domain in the original table-based approach by $k/4$ times.

5.1.2. ***Evaluating table-based parallel encoding*.** We now employ the optimized table-based scheme we presented in Sec. 5.1.1. We design a new partitioning scheme for all preprocessing and encoding stages and assign only a single thread block per each SM. This is intended to reduce the number of loads of $\log / \exp$ tables because each thread block requires a full table at every kernel execution. Unlike CPU caches, CUDA's shared memory is "not persistent" across GPU kernel calls. The performance results are shown in Fig. 6, comparing the optimized table-based scheme against the loop-based scheme both on GTX 280. As it is evident from the graph, the encoding performance has improved by at least 30% across all settings.

Although a rough estimate of executed instructions for this optimized table-based scheme (including the preprocessing

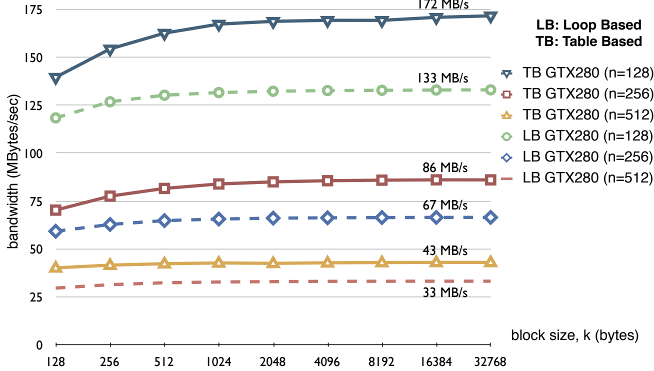

Figure 6. Parallel network encoding using the optimized table-based vs. loop-based scheme on GTX 280.

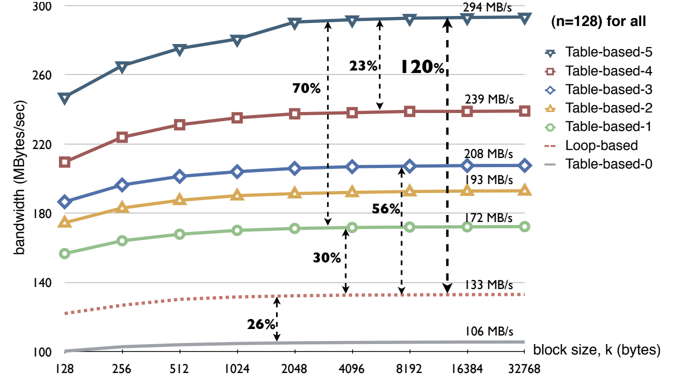

Figure 7. Encoding performance of various schemes for n=128 on GTX 280.

overhead) amounts to around half of the loop-based GF-multiplication, the observed performance improvement is not proportional to the reduction in instruction count. This is mainly attributed to two factors. First, the bandwidth of shared memory is one access (up to 32-bit) per bank in every two cycles [11]. Second, hosting the $\log / \exp$ tables in the shared memory leads to many concurrent byte-length and random accesses to different sections of the tables. These accesses can not be coalesced. Further, bank conflicts will be an issue too. As a result, a 30% performance increase is still a significant gain. At such high encoding bandwidth, now more than 1844 downstream peers can be supported in the streaming server scenario presented in Sec. 5.1.1.

Further, our table-based improvement is not limited to the streaming server applications that generate many coded blocks for every source segment. In an experiment, we produced only n coded blocks for each segment of an array of segments, *e.g.*, a VoD scenario where each client requests a different video segment. The performance degraded only by 0.6% compared to the single-segment case, due to the extra preprocessing. This suggests that our algorithm in Sec. 5.1.1 can be applied to multiple source segments at once.

To be fair to the CPU-based scheme, we also deploy the same optimized table-based approach with preprocessing of the source blocks and coefficients matrix to the logarithmic domain. Not only CPU-based encoding fails to achieve any gain, its bandwidth drops up to 43% from the loop-based SIMD accelerated solution. Obviously, for a CPU-based solution, even the optimized form of the table-based scheme is still not a contender for the loop-based scheme.

This result certainly does not imply that the loop-based encoding on the GPU should be written off altogether. The next generations of CUDA GPUs will likely increase their integer arithmetic units to 64 bits, which potentially can double the performance of loop-based GF-multiplication. Further, our optimized loop-based approach can be applied to similar processor cores as GPU SPs, especially with 32-bit execution units and little or no SIMD support, *i.e.,* the mainstream ARM v6 family used in smartphones.

5.1.3. *Pushing the Envelope: Further optimizations*. Further improvement of coding performance is possible through a number of optimization schemes. *First*, for each byte by word multiplication, we combine four tests of `log_x == 0xff` in Fig. 5 into one because a single coefficient is multiplied into the 4-byte word. *Second*, the tests against `0xff` for individual bytes of each word can be converted to tests against `0x00` by using a *new* log *table* such that a zero input is mapped to `0x00` instead of the original `0xff`. This leads to significant performance increase because the tests against zero can be automatically performed during a register load without the need for extra compare instructions. As a result, branching no longer happens as the compiler will use *predicated instructions* leading to even lower instruction count. Of course, the $\exp$ table also has to be adjusted to compensate for the new $\log$ table. The results of these two schemes are shown as `Table-based-2` and `Table-based-3` respectively in Fig. 7. `Table-based-1` reflects our new table-based scheme in Sec. 5.1.2, while `Table-based-0` shows the original results before any optimization.

For our *third* optimization effort, we evaluate the performance of *texture memory* for storing the $\exp$ table. We basically run the same algorithm except that we access the $\exp$ table resident in the texture memory (texture memory accesses are cached). According to the `Table-based-4` graph in Fig. 7, this leads to another 15% performance improvement. Unfortunately there is very little public information on the structure of the texture cache (*e.g.,* cache line size and latency) and how concurrent accesses to texture caches are resolved. This improvement is mainly due to the locality of accesses to the $\exp$ table and also the smaller number of instructions needed for address calculation in texture memory accesses compared to shared memory's address calculation. Also, we suspect that the texture cache controller can combine multiple pending requests to a cache line even if initiated from different SMs (in GTX 280, every three SMs use a shared texture cache).

In table-based GF-multiplication, each thread of a thread warp accesses a different byte from the $\exp$ table, in general. Except our last scheme, which used the texture memory,

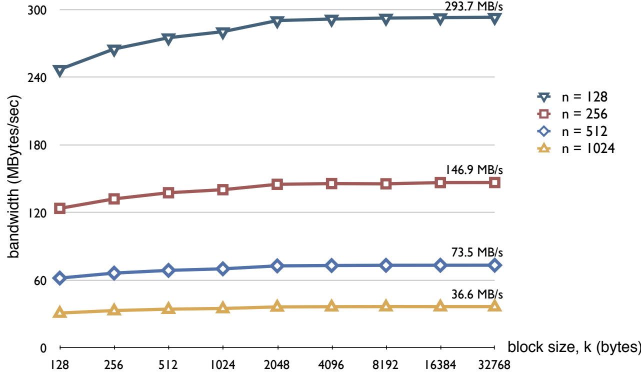

Figure 8. Highly optimized encoding on GTX 280.

we store the exp table in the shared memory. The shared memory has 16 banks, each with a 4-byte width, to serve 16 concurrent requests issued from threads of a half-warp every two cycles. Because these accesses are to random locations in the exp table, albeit with some locality, stalls due to bank conflicts are common. In average, around 3 conflicts happen within each 16 parallel requests, which need to be resolved through serially issued accesses. Our *fourth* and final optimization effort attempts to eliminate the bank conflicts by employing multiple copies of the exp table, so each thread accesses its private copy of the table. Ideally, we need 16 such exp tables which should easily fit in the shared memory of 16 KB size. However, banks are interleaved within the address space of the shared memory, so mapping each thread's access to its private exp table requires a number of extra instructions for address calculation. It turns out that this extra overhead defeats any gain achieved by conflict-free access.

To alleviate address mapping for referencing the table, we store each element as a word rather than a byte. However, with word-length elements, we can only fit 8 copies of exp tables in the share memory (each table has 512 elements). Although this scheme can not fully eliminate bank conflicts, it reduces the conflict probability significantly. Fitting eight tables does not turn out to be easy as the shared memory is also used for other essential tasks, *e.g.,* passing parameters to the GPU kernel. Also, we optimize address calculation to minimize the number of instructions. The result, shown in the `Table-based-5` graph, improves our previous scheme by another 23% up to 293.7 MB/s, which is is 2.2 times of our loop-based scheme. Such an encoding rate can serve more than 3050 peers, way up from 1385 peers, at the streaming setting we discussed in Sec. 5.1.1. Our estimates show that the encoding performance would be around 330 to 340 MB/s for a fully conflict-free deployment if the share memory size was at least 32 KB.

Fig. 8 summarizes our best encoding performances across various coding settings. Now even encoding at $n = 1024$ achieves rates in order of a few tens of MB/s. As a final note, memory accesses of the encoding process are almost perfectly hidden within the computations. A benchmark that generates dummy input data (source blocks and coefficients) on the fly, instead of accessing the graphics memory, performs better by only 0.5%. This confirms that access to the graphics memory has almost no negative effects in the performance of our encoding algorithm.

5.2. Parallel Multi-Segment Decoding

As pointed out in Sec. 4.3, due to the lack of parallelism in our decoding scenario, our GPU-based network decoding had difficulty to scale up well and to fully take advantage of the available GPU computing power. Coded blocks have to be decoded one by one till a segment is fully decoded. Only then the decoding of the next segment starts. However, this is not the only application scenario to use network decoding. Avalanche [3], which uses network coding in bulk content distribution, gathers a large number of coded blocks over a period of time and performs decoding offline. Even in peer-to-peer VoD applications where the downstream bandwidth varies significantly over time, a peer might receive multiple video segments at the same time to take advantage of the available bandwidth at the moment.

The degree of parallelism in the decoding process increases linearly with the number of available segments, since coded blocks from different segments can be processed in parallel. Then we will have more coded blocks to work on, *i.e.,* more GPU threads to launch, leading to better masking of memory latency and data dependency in GPU cores.

Let us revisit our task partitioning scheme in Fig. 3, and assume that there exist coded blocks from 30 different segments. Then an ideal solution will decode each segment fully by a dedicated SM so there will be no longer the need to duplicate the coefficient matrix $\mathbf{C}$ at every SM. Each SM works on a full coded block matrix $\mathbf{x}$. However, a new problem arises as the original thread assignment scheme will no longer work. We assigned one thread for the decoding of every 4-byte column of the aggregate $[\mathbf{C}|\mathbf{x}]$. For example, at $(n = 128,\ k = 4096)$, 1056 thread was required, far beyond the limits of a single thread block. Using multiple thread blocks for a segment is not an option either, as it will cause synchronization problems we discussed in Sec. 4.2.2. Further, each thread would have to work on multiple columns, leading to less efficient code and many load and stores to memory.

As a solution, we propose to calculate $\mathbf{C}^{-1}$ first by doing Gauss-Jordan elimination for the aggregate matrix $[\mathbf{C}|\mathbf{I}]$. Then a regular multiplication in Galois field, similar to the encoding process of Eq. 1, will restore the original segment. Although the GPU will not be fully used in the first stage of such a decoding process (due to the small coefficients matrix and the serial nature of row operations for matrix inversion), it will be fully utilized in the multiplication stage because of its high degree of parallelism.

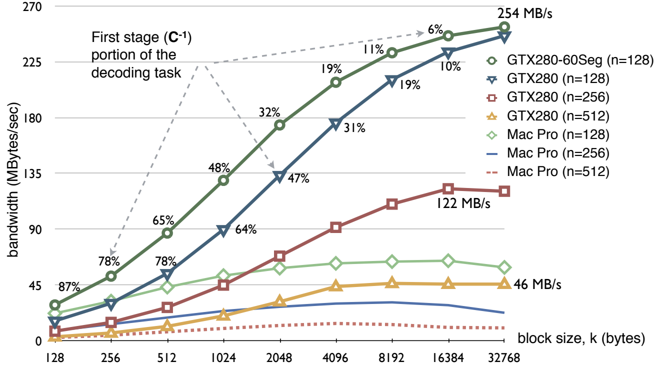

Figure 9. Parallel multi-segment decoding on GTX 280 and Mac Pro.

After implementing the new scheme in CUDA, we have also developed a parallel segment decoding scheme for the CPU. For our 8-core Mac Pro system, we operate on 8 segments in parallel at a time, with each segment being processed by a CPU thread. The performance comparison is given in Fig. 9. As it is clear from the graph, GTX 280 outperforms the Mac Pro for all configurations with block sizes more than 256 bytes by a ratio between 1.3 and 5.3. By comparing Fig. 9 against Fig. 4, we notice that GPU-based multi-segment decoding achieves far better gains than the CPU-based multi-segment decoding when they are compared against their single-segment results. At $(n = 128, k = 16384)$, for example, the GTX 280 gains by a factor of 3.6, while the Mac Pro only gains by a factor of 1.3. As a result, multi-segment decoding should be the preferred scheme whenever the application scenario allows.

Another disadvantage of CPU-based decoding can be observed when the block size increases. The Mac Pro's decoding bandwidth starts dropping at block sizes of 8 KB for $n = 512$, at 16 KB for $n = 256$, and at 32 KB for $n = 128$. This is due to the fact that the data set increases substantially in the multi-segment scheme. The overall coded blocks being decoded become so large (4 MB per segment and 32 MB for the 8 active segments) that data accesses become memory-bound (the total Level 2 cache of all 8 cores is 24 MB).

Finally, our GPU-based multi-segment decoding can benefit from issuing more than one segment to each SM, *i.e.,* operating on 60 segments instead of 30 in parallel. This is due to the increased GPU utilization in the first stage of decoding, *i.e.,* calculation of $\mathbf{C}^{-1}$. By assigning two matrix inversions from separate segments to each SM, now two matrix inversions can proceed in parallel, improving the GPU utilization in the first stage (utilization in the second stage does not change much as forward multiplication is already a highly parallel task). The result is shown by the `GTX280-60Seg-n128` graph in Fig. 9 for $n = 128$, which clearly defeats the decoding performance of 30 segments, by up to a factor of 1.4. The graph is annotated by the first stage's workload share in the overall decoding task. It is obviously reduced compared to the 30-segment case, reflecting better utilization (for example from 64% to 48% at $k = 1024$). As the block size increases, the workload ratio of the first stage decreases and the overall decoding rate gets closer to its encoding counterpart.

At this point, GPU-based decoding defeats the Mac Pro's decoding bandwidth across the board by a factor of 1.3 to 4.2. The advantage over single-segment GPU-based decoding in Fig. 6 is between a factor of 2.7 and 48.8. Higher gains are achieved at smaller block sizes, where single-segment GPU decoding did not perform very well. In more practical block sizes, 1024 bytes and more, the decoding gain is between 2.7 and 27.6.

5.3. Revisiting CPU-based Encoding

For generating encoded blocks on a multi-core system, our CPU-based scheme in [5] and [6] partitioned the encoding task of "each coded block" among different threads, one thread per core, running in parallel. This ensured that a new encoded block is generated as fast as possible to achieve a maximum degree of parallelism at the coded block level. However, in a streaming server that generates a large number of encoded blocks for its downstream nodes (as in the GPU-based encoding scheme described in Sec. 5.1.1), the goal is to generate many coded blocks fast, buffer them and serve the downstream from the buffer over a longer period. In this case, each CPU's thread can encode fully coded blocks rather than partial blocks.

Fig. 10 compares the CPU-based encoding performance of the new full-block encode scheme against the original scheme, showing much better performance for small block sizes, apparently due to better performance of the memory prefetcher loading a longer sequence of data, *i.e.,* full blocks. However, both schemes essentially achieve the same encoding rate as the block size grows. Of course, the new scheme is the preferred one for a CPU-based streaming server, as it achieves a more consistent encoding bandwidth across a range of block sizes.

Note that the new partitioning scheme essentially has the same overall computational complexity as the original one. The difference is in the usage scenario and system deployment. The original rationale behind partitioning the encoding process of each block in [5] was: (1) a new coded block should be *generated* as soon possible *on demand*; and (2) better caching performance is to be achieved by partitioning the data set among CPU cores. However, a more powerful server like our 8-core Mac Pro enjoys a much larger aggregate cache. In a streaming server deployment, the coded blocks can be generated in advance and buffered. This replaces *on-demand generation* in the original scheme with *on-demand delivery*.

5.4. Miscellaneous Improvements

We now briefly review a number of miscellaneous issues we explored to further improve the coding performance.

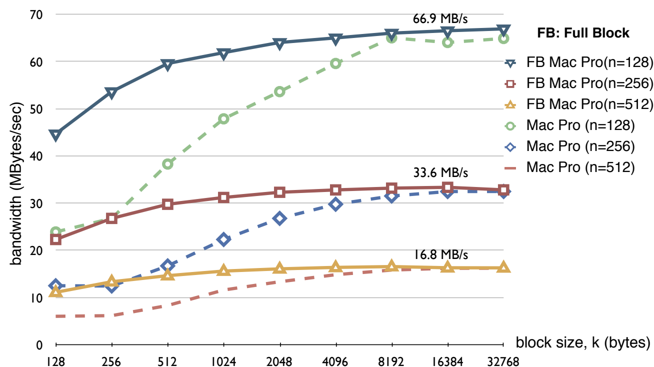

Figure 10. Parallel CPU-based encoding: full-block encoding vs. partitioned-block encoding.

5.4.1. *Network encoding with both GPU and CPU*. Due to the high degree of parallelism in the network encoding process, encoding can be employed by GPU and CPU in parallel, achieving encoding rates in proximity to the sum of the individual bandwidths on the GPU and the CPU. However, the GTX 280 encoding rate is around 4.3 times that of a CPU-based solution on our 8-core Mac Pro server. This suggests an extra GTX 280 GPU, priced around US$300 at the time of this writing, leads to not only a much cleaner solution relieving CPU from heavy computation, but also a much better price/performance ratio.

5.4.2. *Speeding up the decoding process with specialized instructions*. As discussed in Sec. 4.2.2, the decoding process requires a search for the first non-zero coefficient of the current row. However, the search has to wait till all GPU threads working on the current coefficient row synchronize after some initial work [5]. The search process can be accelerated if each thread reports its leading non-zero coefficient through an atomic minimum operation, `atomicMin`. GTX 280 is the first CUDA-enabled GPU that allows `atomicMin` operate on its shared memory, where we buffer the intermediate results. This optimization improves the decoding performance by around 0.6%.

5.4.3. *Aggressive caching for decoding*. Our decoding scheme aggressively caches various data structures on the shared memory to reduce accesses to the GPU memory. A more aggressive caching scheme would try to cache the entire coefficient matrix, because elements of **C** are the most frequent data referenced during the decoding process. However, having only 16 KB of on-chip shared memory per SM limits such scheme to coding configurations with $n = 128$ and less. With a number of creative techniques, we deployed full caching of the coefficients matrix beside caching of other essential data structures. The results show between 0.5% to 3.4% improvement in the decoding process. The smaller block sizes, *e.g.,* less than 1024, enjoy the most substantial gain, since the processing of the coefficient matrix consumes a larger share of their execution time.

6. Concluding Remarks

This paper presents the best implementation of random network coding using GPUs reported in the literature. We presented a highly optimized table-based encoding scheme for the GPU that outperforms the loop-based algorithm on the same GPU by a factor of 2.2 across the board. The encoding advantage over a 8-core Mac Pro server is at least a factor of 4.3. Our multi-segment decoding scheme outperforms its 8-core Mac Pro counterpart by a factor between 1.3 and 4.2, significantly closing the gap with the encoding performance especially at large block sizes.

With a encoding rate of 294 MB/s at 128 blocks, more than 3000 downstream peers can be served at a streaming rate of 768 Kbps. Such a computation bandwidth can easily saturate two Gigabit Ethernet interfaces, making the GPUs the prime choice for streaming servers. With a much better memory performance than CPUs, a rapidly increasing number of cores, and much better performance/price ratio, GPUs are able to bring high-performance network coding to real-world applications.

References

[1] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung, “Network Information Flow,” *IEEE Trans. on Info. Theory*, July 2000.

[2] P. Chou, Y. Wu, and K. Jain, “Practical Network Coding,” in *Proc. of Allerton Conference on Communication, Control, and Computing*, October 2003.

[3] C. Gkantsidis and P. Rodriguez, “Network Coding for Large Scale Content Distribution,” in *Proc. of IEEE INFOCOM 2005*, March 2005.

[4] T. Ho, R. Koetter, M. Medard, D. Karger, and M. Effros, “The Benefits of Coding over Routing in a Randomized Setting,” in *Proc. of ISIT 2003*, June-July 2003.

[5] H. Shojania and B. Li, “Parallelized Network Coding With Hardware Acceleration,” in *Proc. of the 15th IEEE International Workshop on Quality of Service (IWQoS)*, 2007.

[6] H. Shojania, B. Li, and X. Wang, “Nuclei: Graphics-accelerated Many-core Network Coding,” in *Proc. of IEEE INFOCOM 2009*, August 2009.

[7] C. Gkantsidis, J. Miller, and P. Rodriguez, “Comprehensive View of a Live Network Coding P2P System,” in *ACM Internet Measurement Conference (IMC 2006)*, 2006.

[8] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, “Efficient Erasure Correcting Codes,” *IEEE Trans. Info. Theory*, vol. 47, no. 2, pp. 569–584, February 2001.

[9] P. Maymounkov, N. Harvey, and D. Lun, “Methods for Efficient Network Coding,” in *Proc. of 44th Annual Allerton Conference on Comm., Control, and Computing*, Sep. 2006.

[10] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architecture,” in *IEEE MICRO*, March-April 2008, vol. 28.

[11] NVIDIA Corporation, *NVIDIA CUDA: Programming Guide, Version 2.0*, July 2008.