Nuclei: GPU-accelerated Many-core Network Coding

Hassan Shojania, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
{*hassan, bli*}*@eecg.toronto.edu*

Xin Wang
School of Computer Science
Fudan University
xinw@fudan.edu.cn

***Abstract*—While it is a well known result 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. Our previous work has attempted to design a hardware-accelerated and multi-threaded implementation of network coding to fully utilize multi-core CPUs, as well as SSE2 and AltiVec SIMD vector instructions on x86 and PowerPC processors. This paper represents another step forward, and presents the first attempt in the literature to maximize the performance of network coding by taking advantage of not only multi-core CPUs, but also potentially hundreds of computing cores in commodity off-the-shelf Graphics Processing Units (GPU).**

With GPU computing gaining momentum as a result of increased hardware capabilities and improved programmability, our work shows how the GPU, with a design involving thousands of lightweight threads, can boost network coding performance significantly. Many-core GPUs can be deployed as an attractive alternative and complementary solution to multi-core servers, by offering a better price/performance advantage. In fact, multi-core CPUs and many-core GPUs can be deployed and used to perform network coding simultaneously, potentially useful in media streaming servers where hundreds of peers are served concurrently by these dedicated servers. In this paper, we present *Nuclei,* the design and implementation of GPU-based network coding. With *Nuclei,* only one mainstream NVidia 8800 GT GPU outperforms an 8-core Intel Xeon server in most test cases. A combined CPU-GPU encoding scenario achieves coding rates of up to 116 MB/second for a variety of coding settings, which is sufficient to saturate a Gigabit Ethernet interface.

***Index Terms*—Network coding, Many-core GPU computing.**

I. 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. Ahlswede *et al.* [1] and Koetter *et al.* [2] have proved that the cut-set capacity bounds of unicast flows from the source to each of the receivers can be achieved in a multicast session with network coding in directed networks.

In theory, network coding helps to alleviate competition among flows at the bottleneck, thus improving session throughput in general. It has been repeatedly shown that network coding can lead to more robust protocols with less overhead [3], and better utilization of the available bandwidth. Intuitively, it is promising to apply principles of network coding in large-scale content distribution and media streaming systems. Indeed, Wu *et al.* [4] and Gkantsidis *et al.* [5] have both proposed to apply random network coding, first proposed in [6], 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 systems 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 random network coding with random linear codes, especially as the number of blocks to be coded scales up. Since random linear codes are universally adopted in all proposed practical protocols using network coding, we believe that it is important to design and implement random linear codes such that its real-world coding performance is maximized, on modern off-the-shelf hardware platforms. This is particularly important for streaming servers that need to sustain the bandwidth required for hundreds of directly connected peers in a peer-assisted Video-on-Demand (VoD) streaming system, and to saturate the line speed of its Gigabit Ethernet interface.

Our previous work [7] has shown a SIMD-accelerated multi-threaded implementation of network coding, that takes advantage of both multi-core CPUs and SIMD vector instructions on modern processors. This paper represents another substantial step forward, and presents the first attempt in the literature to maximize the performance of network coding by taking advantage of not only multi-core CPUs, but also potentially hundreds of computing cores in commodity off-the-shelf many-core *Graphics Processing Units (GPUs)*. Modern NVIDIA GPUs, for example, are designed with hundreds of specialized *cores*, each with much less complexity than a CPU core, but nevertheless supports operations required for general-purpose high-performance computing. We are motivated by the natural curiosity of whether or not GPUs are able to help improve the performance of network coding. Such an interest is further stimulated by the relative low cost of mainstream GPUs as compared to multi-core CPUs. As an example, the NVIDIA GeForce 8800 GT retails for approximately $1/20$ of the cost of a dual Quad-core Intel CPU setup.

In this paper, based on the NVIDIA CUDA framework, we present *Nuclei*, our design and implementation of GPU-based many-core network coding, across platforms including Windows, Linux and Mac OS X. With *Nuclei*, we have made the following new observations on the feasibility of GPU-based network coding. *First*, although the GeForce 8800 GT, featuring 112 cores, is less capable than the 8-core Intel Xeon

server with respect to its raw computing power, the GPU is designed to better hide memory latency, a critical issue that affects the performance of network coding, particularly in the typical coding range for streaming servers. *Second*, by completely detaching the encoding process from the CPU and hand it over to the GPU, the CPU cores on dedicated streaming servers are set free to perform other CPU-intensive tasks that GPUs are not able to perform. *Third*, due to the specific GPU design that schedules its threads in hardware, the performance of GPU-based network coding is not affected by competing threads and background tasks. In contrast, variations of up to 9% are observed with CPU-based network coding, due to background threads. *Finally*, GPUs are equally capable of both encoding and decoding. Though decoding is much more challenging to be performed on the GPU with less capable performance than the 8-core Intel Xeon server at small block sizes, the GPU decoding performance improves substantially as block sizes become larger, and is on par with 8-core CPUs at a block size of 16 KB.

Nuclei has achieved stellar performance results, making it feasible to saturate Gigabit Ethernet interface by combining 8-core Intel Xeon CPUs and a mainstream NVIDIA GeForce 8800 GT GPU. With respect to encoding, for example, the GeForce 8800 GT by itself is able to achieve an encoding rate of 84 MB/second with 128 blocks, outperforming all eight CPU cores combined. A combined CPU-GPU encoding scenario achieves coding rates of up to 116 MB/second for a variety of coding settings, which is sufficient to saturate the Gigabit Ethernet interface.

The remainder of this paper is organized as follows. Sec. II discusses related work. Sec. III provides an overview of both GPU computing and random network coding. Sec. IV presents our design towards many-core GPU-based network coding. Sec. V evaluates our GPU-based network coding implementation. Finally, Sec. VI concludes the paper.

II. Related Work

Ho *et al.* [6] has been the first to propose the concept of *random 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.* [4] and Ghantsidis *et al.* [5], [8] have shown that random network coding is beneficial in bulk content distribution systems. Theoretically, the high computational complexity of random linear codes has been well known: it motivates research on the use of more efficient codes in real-world systems, including traditional Reed-solomon (RS) codes, *fountain codes* [9], and *chunked codes* [10]. While fountain codes are much less computationally intensive as compared to random linear codes, they suffer from a number of drawbacks: (1) coded blocks cannot be recoded without complete decoding, which defeats the original intent of network coding; (2) depending on the code used, there exists a small amount of overhead (about 5% with 10,000 blocks, and may be over 50% with 100 blocks), which decreases the efficiency of using bandwidth; and (3) the decoding process cannot be progressively performed while receiving coded blocks, which may lead to bursty CPU usage when the final blocks are decoded. Alternatively, while Reed-Solomon (RS) codes may also be used to reduce coding complexity, its smaller coded message space makes it difficult for multiple independent encoders to code a shared data source, to be sent to a single receiver.

While there is no doubt that more efficient codes exist, they may not be suitable for random 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 realize the full potential of random network coding in a real-world setting. Our previous work [7] has evaluated the performance of our implementation of network coding that takes advantage of multi-core CPUs and modern SIMD vector instruction sets. Though we have achieved a level of performance in [7] that has not been previously reported at the time, this paper takes another step forward, by evaluating the feasibility of many-core network coding using over a hundred specialized cores on GPUs. We believe that techniques explored in our GPU-based network coding implementation can readily be employed to perform other linear coding schemes operating in the Galois field, such as Reed-Solomon codes.

III. Overview of GPU Computing and Network Coding

Modern GPUs have gradually evolved from specialized engines operating on fixed pixels and vertex data types, into programmable parallel processors with enormous computing power [11]. NVIDIA's Tesla GPU architecture is employed in a wide range of professional and consumer GPU products, and is the first such GPU architecture. On the Tesla architecture, it is possible to develop high-performance parallel computing applications in the C language, using the Compute Unified Device Architecture (CUDA) programming model and development tools [12].

Our performance evaluation in this paper uses the mainstream NVIDIA GeForce 8800 GT GPU with 112 cores, which is supported by the CUDA platform. Our cross-platform implementation in *Nuclei*, however, can be used on any CUDA-supported GPU, ranging from the high-end GeForce GTX 280 with 240 computing cores, to the mobile GeForce 8600M GT, with 32 cores.

A. The Tesla Architecture and CUDA

The Tesla GPU architecture is based on a scalable array of *Scalar Processors (SPs)*. These SPs are grouped into groups of 8, called *Streaming Multiprocessors (SMs)*, and share a number of resources, including *registers* and *shared memory* [13]. On the 8800 GT, for example, there are 112 SPs grouped into 14 SMs, as shown in Fig. 1. Threads too are grouped into *thread blocks*, which allow threads of each block to share data and synchronize among themselves. Each thread block is assigned to a SM and runs on its SPs. After a thread block terminates, new blocks are launched on the vacated SM. The GPU process finally finishes when all thread blocks terminate.

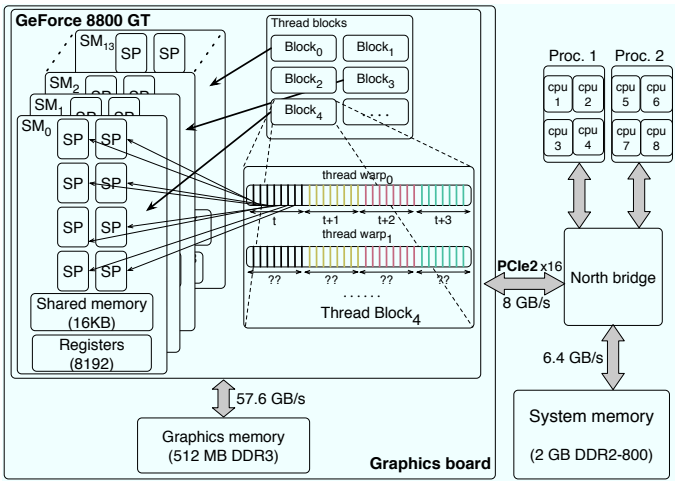

Fig. 1. The architecture of our Mac Pro testbed with the NVIDIA GeForce 8800 GT GPU. The Mac Pro comes with two 2.8 GHz Quad-Core Intel Xeon processors, 2 GB memory, and 12 MB cache per processor.

Each SM manages and executes concurrent threads (of one or more thread blocks) in hardware with zero scheduling overhead. Every 32 threads of a thread block are grouped together in a *thread warp*, running in a synchronized manner with each other, executing the same instruction in a SIMT (Single Instruction, Multiple Thread) fashion. Since there are no more than 8 SPs in a SM, the 32 threads of a thread warp are scheduled eight at a time in 4 cycles. As the example in Fig. 1 shows, the first 8-thread group of warp_0 of the 4^{th} thread block is currently executed at cycle t. Each of the next 8-thread group of warp_0 will execute in the next 3 cycles. However, threads of warp_1 are not ready to be scheduled, *e.g.*, because of a pending memory access.

Since each SP is deeply pipelined, read-after-write dependencies can occur often. Similarly, a memory access causes a thread warp to stall for the next few hundred cycles till the data arrives. The beauty of the Tesla architecture comes from the fact that, as soon as it detects a thread of a thread warp can not proceed, it can switch to another ready warp with zero scheduling overhead [13]. As a result, it is essential to have as many thread as possible so the SPs can execute from other thread warps till the stalled one can be rescheduled. The large thread count, together with the support for many outstanding load requests, helps to hide memory load latency [12].

Comparing to the CPU, the GPU dedicates its die area to a higher number of processing cores. SPs are designed to be simple in-order engines without branch prediction, register renaming, multiple pipelines and many other more advanced but standard features found on modern CPUs. The good news is that, with wider and faster memory interfaces, the GPU has a much higher memory bandwidth at its disposal than the CPU. For example, the DDR3 memory found in the 8800 GT has a peak performance of 57.6 GB/s, compared to a meager 6.4 GB/s of our Mac Pro server based on dual Quad-core 2.8 GHz Intel Xeon CPUs, with its DDR2-800 memory. Further, with a 16-lane bidirectional PCI Express 2.0 interface bus between GPU and CPU, a 8 GB/s bidirectional data rate gives us abundant bandwidth to move data around between the GPU and system memory, with almost no cost for the size of data we cope with to perform network coding.

A CUDA-enabled program, written in C, consists of the *host code* that runs on the CPU, and a *device code* that runs on the GPU. When the CPU invokes a GPU *kernel* (the device code analogous to an application process), the CUDA driver programs the GPU to launch a large number of threads. The host code is compiled by a regular C/C++ compiler. The device code is compiled by NVIDIA's own compiler, generating an assembly language output in a format called Parallel Thread eXecution (PTX). PTX is a "virtual" language, and is not specific to a particular GPU product. However, most of its instructions have machine instruction equivalents in the current CUDA GPUs. Either the PTX intermediate representation or, if the target GPU is known at compile time, the final generated code called *cubin* are embedded in the final application executable. When an application executes, the CUDA runtime generates the machine code, if needed, and loads it to the GPU through the device driver.

B. 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, a node first independently and randomly chooses a set of coding coefficients $[c_{j1}, c_{j2}, \cdots, c_{jn}]$ in $\text{GF}(2^8)$, 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 node 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 $n \times n$ coefficient 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 $\text{GF}(2^8)$. The inversion of $\mathbf{C}$ is only possible when its rows are linearly independent, *i.e.,* $\mathbf{C}$ is full rank.

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

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

Fig. 2. Table-based multiplication in $\text{GF}(2^8)$: baseline implementation.

A network node does not have to wait for all n linearly independent coded blocks before decoding begins. It can instead start to decode as soon as the first coded block is received, and then progressively decode coded blocks as they arrive over the network using Gauss-Jordan elimination.

IV. NUCLEI: GPU-ACCELERATED NETWORK CODING

In this section, we present challenges and solutions involved in the design of *Nuclei*, our implementation of GPU-accelerated network coding.

A. Random Network Coding: Performance Bottleneck

Random network coding suffers from a major performance bottleneck. Byte-length multiplication in GF(2^8) is a costly operation due to three memory accesses when $\log/\exp$ tables are used, and it is performed in tight loops over rows of coefficients and coded blocks, each of n and k bytes, respectively. To address this bottleneck, we first proposed in [7] 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 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. 2 is shown in Fig. 3.

```
byte loop_gf_multiply_byte(byte factor, byte data)
{
   byte result = 0;                                  (1)
   bool overflowing;                                 (2)
   while (factor != 0) {                             (3)
      if ((factor & 1) != 0)                         (4)
         result = result ^ data;                     (5)
      overflowing = data & 0x80;                     (6)
      data = data << 1;                              (7)
      // irreducible polynomial: x^8+x^4+x^3+x^2+1
      if (overflowing == true)                       (8)
         data = data ^ 0x1d;                         (9)
      factor = factor >> 1;                          (10)
   }
   return result;                                    (11)
}
```

Fig. 3. Loop-based byte-length multiplication in GF(2^8).

With such a loop-based approach, we were able to take advantage of SIMD (Single Instruction, Multiple Data) vector instruction sets to perform GF(2^8) multiplication on 16 byte-long units of each row, rather than single-byte units [7]. Rather than using vector instructions provided by the CPU, is it possible to use the GPU and still achieve accelerated multiplication using such a loop-based approach? At first glance, it is a daunting challenge due to the lack of SIMD instructions and wide execution units on the GPU.

B. Loop-based GF(2^8) Multiplication on the GPU

Unlike modern CPU cores with 128-bit registers and execution units, current CUDA-enabled GPUs have plain 32-bit registers and execution units. Nevertheless, they have more than a hundred processing cores (*e.g.*, 112 SPs for the 8800 GT) that run in parallel.

To take full advantage of the available 32-bit arithmetic units in the GPU, we rewrite byte-length multiplication as `loop_gf_multiply_word` to multiply a one-byte coefficient with a 4-byte word. This is not a straightforward task, however. Both Intel SSE2 and PowerPC AltiVec SIMD instructions include special instructions that allow arithmetic and comparison operations on individual bytes of 16-byte registers in parallel. With no similar instruction available on the GPU, we have to emulate such byte-long operations on 4-byte words. The main issues involved are the following: **(1)** To left-shift individual bytes of `data` word without affecting the neighboring bytes (statement (7) in Fig. 3); **(2)** To determine the "overflow" status by examining the top bit of individual bytes of the 32-bit data word (statement (6)); **(3)** To apply the irreducible polynomial byte `0x1d` to each byte of the data word based on the "overflow" states (statements (8) and (9)). We use a series of `if/else` conditions, byte extractions, bitwise operations and bit shifts to address these issues. Even with our best effort to minimize the code complexity, the compiled code results in no fewer than 16 *cubin* instructions on the 8800 GT.

To evaluate the computational performance of our loop-based GF-multiplication, we have designed a benchmark that entails an encoding process that produces $c = 7168$ coded blocks of size 2048 bytes each, where each coded block is a linear combination of 1024 blocks, *i.e.*, $(n = 1024, k = 2048)$. To detach the benchmark from the memory performance of the GPU, we create input coefficients and rows of data on the fly, rather than loading them from graphics memory. To evaluate the worst-case scenario, we consistently use `0xff` as coefficients to make sure that the loop always iterates for the maximum 8 times. Our benchmark is implemented by launching 7168 GPU threads and assigning the computation of each coded block to its own thread. Since a core in the GPU has a 32-bit execution unit, each GPU thread executes `loop_gf_multiply_word` for $n \cdot k/4$ times, and the entire benchmark executes the function for a total of $(c \cdot n \cdot k/4)$ times.

The execution of the benchmark takes 2509 ms (milliseconds) to complete, which shows a much better performance than byte-length GF-multiplication, which takes 7186 ms to complete.

C. Further Optimizations of Word-length GF-multiplication

In a CUDA kernel, all 32 threads of a *thread warp* execute the same instruction in a synchronized manner. When reaching an `if/else` condition, even if only a single thread takes one path against the others, the overall warp execution time will be delayed as if *both paths* are executed. When generating the word-length polynomial mask pattern in our `loop_gf_multiply_word` implementation, with each byte having `0x1d` if individual bytes of the `data` word is about to overflow, we resorted to a series of conditions. It is apparent that, to further improve performance, we need to avoid these conditions as much as possible.

As shown in statements (5), (6), and (7a) of Fig. 4, we can generate the polynomial mask pattern by shifting the overflow state of bits to the first bit of each byte and then multiply the entire word by `0x1d`. With this improvement, the number of *cubin* instructions decreases from 16 to 14, and our benchmark now takes 2373 ms, a 5.7% improvement of performance.

```
word loop_gf_multiply_word(byte factor, word data)
{
   word PrimPolyMask, result = 0;                          (1)
   while (factor != 0) {                                   (2)
      if ((factor & 1) != 0)                               (3)
         result = result ^ data;                           (4)
      // creating the irreducible polynomial mask
      PrimPolyMask = data & 0x80808080;                    (5)
      PrimPolyMask = PrimPolyMask >> 7;                    (6)

(7a)  PrimPolyMask = PrimPolyMask * 0x1d;
(7b)  { PrimPolyMask = __mul24(PrimPolyMask, 0x1d);
        if (data &0x80000000)
          PrimPolyMask = PrimPolyMask + 0x1d000000;

      // clear top-bit of bytes before shift
      data = data & 0x7f7f7f7f;                            (8)
      data = data << 1;                                    (9)

      data = data ^ PrimPolyMask;                          (10)
      factor = factor >> 1;                                (11)
   }
   return result;                                          (12)
}
```

Fig. 4. Loop-based GF(2^8) word-length multiplication for a CUDA-enabled GPU.

Since current CUDA-enabled GPUs do not have native 32-bit multiplication, the compiler translates the multiplication of statement (7a) to 4 instructions using 24-bit multiplication and shifting. However, we can rewrite statement (7a) as (7b) to take advantage of the 24-bit multiplication directly. Now the compiled code is reduced to 12 *cubin* instructions and the benchmark executes in 2067 ms.

At a final attempt of our optimization effort, we sidestep the compiler and optimize the PTX virtual assembly file directly. This approach successfully removes another *cubin* instruction, now down to 11 instructions, which improves the benchmark's execution time to 1905 ms, a 32% improvement over our initial implementation. This is a very interesting result, since a "back-of-the-envelope" calculation suggests that the total computing power of all 112 cores on the 8800 GT is almost used to the best possible. The number of cycles taken by executing a single `loop_gf_multiply_word` can be derived as:

$$\begin{aligned}\text{Cycles}_{\text{GF}-\text{mul}} &= \text{total cycles / total computed words}\\ &= (\text{time}\cdot\text{cores}\cdot\text{freq})/(c\cdot n\cdot k/4)\\ &= (1.905\cdot 112\cdot 1.5\text{ GHz})/(7168\cdot 1024\cdot 2048/4)\\ &= 85.16\text{ cycles}\end{aligned}$$

Since our benchmark fixed the number of loop iterations to 8, each iteration takes 10.65 cycles. At first glance, it seems surprising that the result is less than the execution time required for 11 instructions. A closer examination reveals that the conditional addition in statement (7b) executes once every other time, resulting in an effective 10.5 cycles in an ideal execution.

Although both CUDA and PTX virtual instructions support 64-bit data types, a `GF-multiply` implementation for double-word `data` can not achieve better performance. This is due to the fact that the current CUDA-enabled GPUs have 32-bit integer engines, and 64-bit operations are emulated through a series of 32-bit operations.

D. CPU vs. GPU: Estimating the Computing Power

Having our highly optimized GPU-based implementation of `GF-multiply`, we are now ready to compare its performance against a SIMD-accelerated CPU-based implementation [7]. Our corresponding benchmark for the CPU, ($c = 7168, n = 1024, k = 2048$), divides each source block of k bytes into 8 partitions, each processed by a dedicated thread, one per CPU core on our 8-core Mac Pro server. The benchmark finishes execution in 1672 ms, reflecting that the 8-core Mac Pro system outperforms the 112-core 8800 GT with a 13.9% margin.

TABLE I
CPU VS. GPU: THEORETICAL COMPUTING PERFORMANCE

	Mac Pro 8-core Intel	8800 GT
f: core freq. (GHz)	2.8	1.5
n: number of cores	8	112
w: multiply data width (bytes)	16	4
ic: instruction count/iteration	12	11
Per core superscalar factor	3	1
cc_1: cycle count/iteration	$12/3 = 4$	11
Est. throughput (cycles/inst.)	$7\cdot 0.33 + 5\cdot 0.5 = 0.40$	1
cc_2: cycle count/iteration	$12/(1/0.4) = 4.8$	11
cc_3: cycle count/iteration	5	10.5
$\text{Performance}_{\frac{\text{CPU}}{\text{GPU}}}$	$\frac{f_{\text{CPU}}}{f_{\text{GPU}}}\cdot\frac{n_{\text{CPU}}}{n_{\text{GPU}}}\cdot\frac{w_{\text{CPU}}}{w_{\text{GPU}}}\cdot\frac{cc_{\text{GPU}}}{cc_{\text{CPU}}}$	

As Table I shows, this is not a surprising result after comparing a number of performance metrics of the CPU with those of the GPU. In our comparison, we can observe that each GPU core is an in-order processing unit with a single Arithmetic Logic Unit (ALU), while each CPU core has three ALUs, each capable of processing a SSE2 instruction in any cycle [14]. This dramatically offsets the higher number of GPU cores, despite the fact that GPU cores run at almost half of the CPU speed. In our first estimate, we assume a coarse throughput advantage of 3 (equal to the superscalar factor) for CPU cores. The performance advantage of CPU-based over GPU-based implementations, $\text{Performance}_{\frac{\text{CPU}}{\text{GPU}}}$, can be computed through estimating the cycle count per iteration. We now progress through three alternative estimates of cycle count per iteration, from coarser to finer granularities. The first estimate cc_1 results in a performance advantage of 1.47, where the CPU is faster. Our second estimate, cc_2, takes a closer examination into CPU instructions and comes up with an average throughput based on individual instructions, which results in a performance advantage of 1.22. Our final refinement considers the data dependency of instructions in the CPU-based implementation, and results in a performance advantage of 1.12, quite close to our measurement results.

This confirms our measurement results that the 8-core Intel Xeon system is expected to have a higher computing power compared to the 112-core 8800 GT. However, as we shall soon see in the next section, memory performance will substantially affect the performance of a CPU-based implementation.

E. Many-core Network Encoding on the GPU

The process of random network encoding essentially consists of a matrix multiplication in the GF domain, and can be considered as a *embarrassingly parallel* computation problem, where a parallel implementation is possible with little or no communication and synchronization among threads. Without considering memory access to source blocks and coefficients, the performance of network encoding is only limited by the hardware's computational power, since the encoding process of multiple coded blocks — and even different section of a coded block — can proceed in parallel by using a large number of threads.

We now complete the picture by considering memory access in a complete process of network encoding. As we shall see, achieving a high speedup can not be taken for granted and requires careful task partitioning.

1) ***Partitioning for many-core network encoding****:* Our synthetic benchmark does not consider memory access. In CUDA, GPUs can only access their graphics memory, so the coefficients and source blocks have to be transferred from the host to the graphics memory first. Similarly, encoding results residing in the graphics memory need to be transferred back to system memory. With 8 GB/s on the PCI Express 2.0 interface, transfer times to and from graphics memory are negligible.

We use the same $(c = 7168, n = 1024, k = 2048)$ setup for our new benchmark with memory access. To ensure a fair comparison with previous tests, we fill the coefficient matrix with all `0xff` to ensure a maximum load. The encoding benchmark on the GPU now takes 5306 ms, reflecting a substantial increase from 1905 ms, suggesting poor memory performance.

After attempting other alternatives, we have settled on a partitioning mechanism with a much finer granularity, with each GPU thread encoding only a "single word" of the coded block, rather than a full block. With careful assignments of words to threads of each warp, we can now take advantage of memory coalescing [13], so most memory accesses of a thread warp fall next to each other, significantly reducing the number of memory accesses by the memory controller on the GPU. With such fine-granularity partitioning, 512×7168 threads have been launched, much higher than the original 7168 threads. However, unlike CPU threads, GPU threads are very lightweight as GPUs are designed to switch to new threads seamlessly in hardware, in order to hide memory latency. Using this new partitioning scheme, our encoding process now takes 3016 ms, a 76% improvement over our original coarse partitioning.

Our next measure towards further optimization reads coefficients in 4-byte chunks, rather than byte by byte, and then caches them for use in the next four multiplications. This reduces read requests for coefficients to $1/4$ of the original approach, and the execution time is reduced to 2098 ms, now over 2.5 times better than our original partitioning.

On the 8-core Mac Pro, the same network encoding benchmark takes 2250 ms, which implies that the GPU performance defeats 8-core CPUs when memory access is considered. reflecting the GPU's better ability to hide memory access latency. While the CPU-based implementation suffers 35% due to memory access, the GPU performance only degrades 10%. This clearly demonstrates the GPU's superior ability to hide memory access latency, due to seamless hardware switching across a large number of threads.

Finally, we use actual random coefficients instead of `0xff`. The CPU takes 2157 ms, while the GPU, being constrained by computation and not memory access, decreases significantly to 1751 ms, showing a 23% advantage over the CPU. All considered, the GPU performs 23% better than the 8-core CPU with memory access.

2) ***Mixed CPU-GPU network encoding****:* When even better performance is required, both CPU and GPU can perform network encoding in parallel. There are two main approaches for task partitioning between CPU and GPU, both having fundamentally the same computation load. Either one divides the working set by partitioning each source block into two (*e.g.,* half each), and assigns each partition to CPU and GPU; or one assigns the encoding tasks of some coded blocks entirely to the GPU, and the remainder to the CPU.

We choose to set up the same $(c = 7168, n = 1024, k = 2048)$ experiment based on the second approach. First, we divide a generation of 7168 coded blocks evenly between the 8-core CPU and GPU, 3584 coded blocks by each. The encoding process takes 1094 ms, reflecting a speedup of 1.99 over a similar CPU only execution. Since the GPU has a better encoding performance, we may consider increasing the GPU's share of coded blocks. Assigning 54% of coded blocks to the GPU improves the encoding performance to 1000 ms, a 2.17 speedup.

3) ***Generating random coefficients in the GPU****:* So far, our GPU-based encoding implementation uses random coefficients generated by the CPU, and transferred to the graphics memory before the start of the encoding process. Although neither the process of generating random coefficients nor the transfer times take a long time to execute, migrating the task of generating random coefficients to the GPU makes the design simpler, as it fully detaches the CPU from the encoding process. In this case, the application using network coding simply requests the GPU for a number of coded blocks.

Our GPU-based random number generation runs hundreds of generators in parallel, each generating the random sequence for a coded block through a random seed. It takes no longer than 1.61 ms for 7168×1024 random coefficients, which is ten times faster than using the CPU.

F. Many-core Network Decoding on the GPU

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. Compared to encoding, this leads to a reduced coding performance in general. However, the more critical challenge is the smaller degree of parallelization 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 the encoding process, lends itself to parallelization only *within* the decoding of the current coded block, and not *across* a number of coded blocks.

Such a lesser degree of parallelization 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 at least one synchronization point, which makes the decoding process a *coarse-grained* parallel program.

We have tested a number of GPU-based decoding schemes and their performance for the $(c = 1024, n = 1024, k = 2048)$ setup. Not surprisingly, our best-performing scheme only achieves 82% of the CPU-based decoding implementation at this setup, as presented in the following.

*1) **GPU-based decoding with CPU assistance**:* In a progressive decoding application scenario, we receive each coded block along with its associated coefficients and decode it partially. After receiving and decoding the n-th coded block, the decoding process completes and all n source blocks are recovered if no linear dependence has been encountered. In our first scheme, for every new coded block, we partition the aggregate $n + k$ coefficients and data — *i.e.,* a row of the aggregate $[\mathbf{C}|\mathbf{x}]$ matrix from Eq. (2) — such that each 4-byte word of the aggregate data is assigned to a thread, leading to a total of $(n + k)/4$ threads.

Each thread reduces the leading coefficients of the new coded block through a number of linear combinations. However, it can not do further work as a global search for the first non-zero coefficient has now become necessary. Since CUDA's synchronization construct only works for threads within a *single thread block*, and not among all GPU threads, we are forced to perform this synchronization at the CPU side. This effectively breaks the decoding process into two GPU kernel processes. After finding the first non-zero coefficient at the CPU side, we launch another GPU kernel to perform the remainder of the decoding operations for the current block, with each GPU thread performing a series of linear combinations for a 4-byte column of the aggregate $[\mathbf{C}|\mathbf{x}]$.

Although this scheme perfectly divides each aggregate row among threads, it suffers from launching an extra GPU kernel to perform synchronization at the CPU side. The decoding performance of 1024 coded blocks with a $(n = 1024, k = 2048)$ setup achieves 82% of the CPU-based performance (2484 ms against 2031 ms).

*2) **Full GPU-based decoding**:* In an attempt to avoid CPU-assisted synchronization and the extra GPU kernel call, 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 can now use CUDA's synchronization construct within each thread block to perform the search for the first non-zero coefficient. However, we do not wish to consume an excessive amount of 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 14 SMs of the 8800 GT. This leads to each thread block effectively decoding $n+k/14$ bytes of aggregate data through $(n+\frac{k}{14})/4$ GPU threads, each thread working on a 4-byte column. Unfortunately, even after applying a number of data caching optimizations in the per-SM shared memory, the GPU-only decoding is not able to perform better than 6073 ms, lagging far behind our CPU-assisted decoding approach.

So far, in both of our GPU-based and CPU-based schemes, we considered progressive decoding in a sense that each coded block is decoded individually, *i.e.,* right after receiving the block from network, ending up with a total of n GPU kernel or CPU calls. From a measurement point of view, this implies that we have to synchronize all execution threads, either CPU or GPU, right after the decoding of each new coded block finishes. For many practical application scenarios, *e.g.,* $(n = 128, k = 4096)$, the complete decoding of n coded blocks takes only around $10-20$ ms. This suggests that we can buffer the n coded blocks from the network, and then start decoding.

Such a decoding approach helps both CPU-based and GPU-based decoding. First, more of the internal structures (used across the decoding of several coded blocks) can remain in the cache. Second, the threads can be executed longer without being interrupted as individual coded blocks are decoded. With such an approach, the performance of the GPU-based implementation improves to 2175 ms, but our 8-threaded CPU-based scheme also performs better, now at 1688 ms, with GPU-based decoding achieving only 78% of the CPU performance.

V. Performance Evaluation

In this section, we evaluate the performance of *Nuclei*, our design and implementation of GPU-accelerated many-core network coding. We use fully dense coding matrices with non-zero coefficients in our evaluation. The performance will be even higher with sparser matrices.

A. Coding Bandwidth

As we evaluate the performance of *Nuclei*, we have tested a range of 128 bytes to 16 KB per block, with 128, 256 and 512 blocks. When compared to our baseline SIMD-accelerated CPU-based implementation with 8 threads (one per CPU core), the coding bandwidth of *Nuclei*, in MB per second, is shown in Fig. 5. The encoding (decoding) bandwidth should be interpreted as the total number of bytes that are produced (or decoded) per second.

Fig. 5(a) shows that GPU encoding in *Nuclei* achieves its peak performance across almost all coding settings. We are able to make a number of observations from these results. As the number of blocks n doubles from 128 to 256 and again to 512, the encoding bandwidth halves first from 66.9 MB/s down to 33.8 MB/s, and again to 16.8 MB/s. This is due to the fact that generating a coded word requires n GF multiplications. For 128 blocks, the encoding of each word requires reading 128 words of source data and writing one word of coded data, in addition to the reading of coefficients. As such, our coding bandwidth of 66.9 MB/s results in a memory access rate of 10.8 GB/s, which is far below the 57.6 GB/s theoretical limit.

These results have confirmed that our encoding performance is only limited by the computation limits of the 8800 GT. For a number of executed instructions to achieve an encoding bandwidth of 66.9 MB/s at 128 blocks, it is equivalent to an instruction rate of 151 GIPS (Giga instructions per second), which is 90% of the advertised theoretical limits of 504 GFLOPS (*i.e.,* 168 GIPS). This represents a surprisingly high performance level, confirming that our partitioning scheme performed very well in hiding the memory latency. It also confirms that the GPU can execute concurrent threads *with zero scheduling overhead* in hardware as claimed, and can perfectly hide register read-after-write latencies when a sufficient number of parallel threads exists within each SM. Since the GPU is only limited by its computation power, its peak performance has been achieved across all (n, k) settings. In comparison, the CPU-based encoding performance follows the same trend as discussed in [7], but at higher coding rates due to more CPU cores. As the block size increases, the CPU

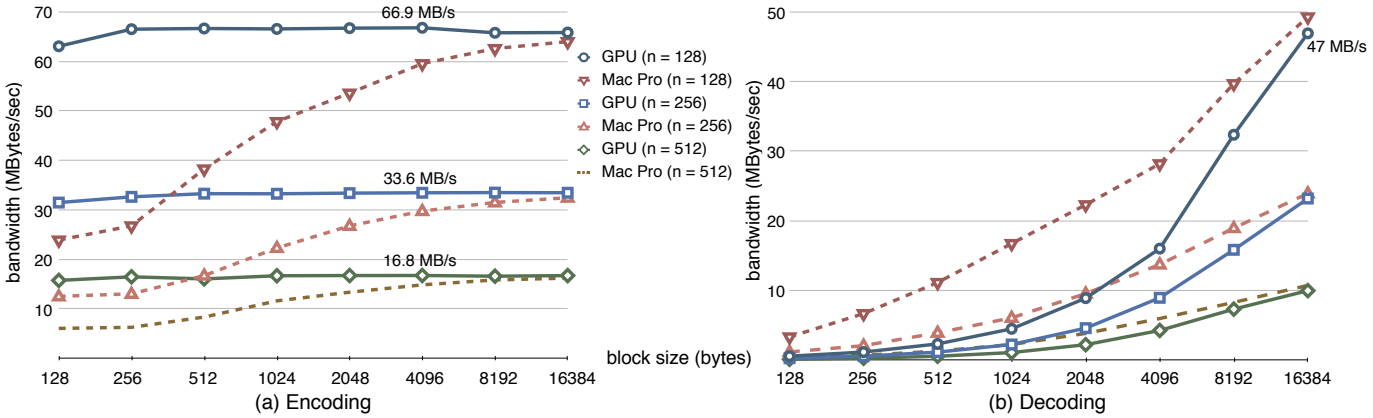

Fig. 5. Coding bandwidth of GPU-based and CPU-based (8-threaded with SIMD acceleration) for the **(a)** encoding; and **(b)** decoding processes.

cache performance has improved, leading to better memory performance and higher encoding rates.

With respect to decoding, the decoding performance shown in Fig. 5(b) is generally lower with both the GPU and the CPU, because each coded block has to be decoded serially. The CPU performs better than the GPU across the board, especially at smaller block sizes, since the GPU does not have sufficient data (small $k/14$) to launch a sufficient number of threads to achieve an acceptable performance gain, and the CPU's computation power is not affected by the block size, unlike its memory performance. On the GPU, as an example at $k = 128$, each of the 14 SMs decodes only $k/14 = 9.14$ bytes of data on average. At $n = 128$, the decoding time remains around 31 ms even if we increase the block size from 128 to 2048 bytes, because latencies of the computation pipeline and memory accesses can not be hidden when there is so little useful computation to be performed. As k increases, though the CPU's performance has improved due to a higher k/n ratio and improved memory performance, the GPU has quickly caught up as soon as it has sufficient data to process and to launch a sufficient number of threads.

B. Network Coding with Both GPU and CPU

In order to evaluate the maximum achievable performance, we now explore the limits of encoding bandwidth when we employ both the CPU and GPU in parallel. We partition the encoding process by generating α portion of the required coded blocks by GPU, and the remainder of the coded blocks by CPU. Because the coding performance of GPU and CPU are not equal, and with the GPU outperforming CPU, we need to configure α optimally. Fig. 6 shows our results. A peak encoding bandwidth of 116.7 MB/s has been achieved at $n = 128$, which can offer sufficient packet payload to saturate the Gigabit Ethernet interface at servers.

C. Revisiting Table-Based Network Coding

We have mainly focused on loop-based GF-multiplication on the GPU so far in this paper. A table-based implementation creates the log/exp tables once at the host side and transfers them to the graphics memory. To achieve higher performance as these tables are heavily accessed, the GPU threads can be better designed by first loading the tables from the graphics memory into the *shared memory*, effectively using it as a *managed L1 cache*. Afterwards, each GPU thread performs byte-by-word GF-multiplication, similar to our loop-based approach. Still, our experiments have shown that such a fine-tuned design performs 30% worse than our loop-based approach.

It is, however, possible to optimize table-based GF-multiplication further, which is explored in details in our ongoing work. The basic idea is the following. In our partitioning, each GPU thread, which calculates a word-length worth of a coded block, accesses a full coefficient row of $\mathbf{C}$ and a full column of original blocks $\mathbf{b}$. However, many other threads use the same row and column in their own coding processes that leads to many redundant conversions to the log domain. By first preprocessing $\mathbf{C}$ and $\mathbf{b}$ and transferring them fully to the log domain, the number of table accesses can be effectively reduced by $2/3$. Our new optimized table-based encoding can improve the performance by 25% over the loop-based approach, as demonstrated in Fig. 7.

Finally, we have also discovered that the performance of optimized table-based GF-multiplication can be further improved by manipulating the tables to more optimally exploit the GPU hardware of SP cores, and by improving the access pattern to the shared memory. With these performance improvements, we are able to achieve encoding rates up to 293 MB/second with $n = 128$ blocks, in our experiments on the high-end NVIDIA GeForce GTX 280 GPU. At this performance level, there is no need to involve the CPU, *e.g.*, in a combined encoding scheme with the GPU, to satisfy most real-world performance needs.

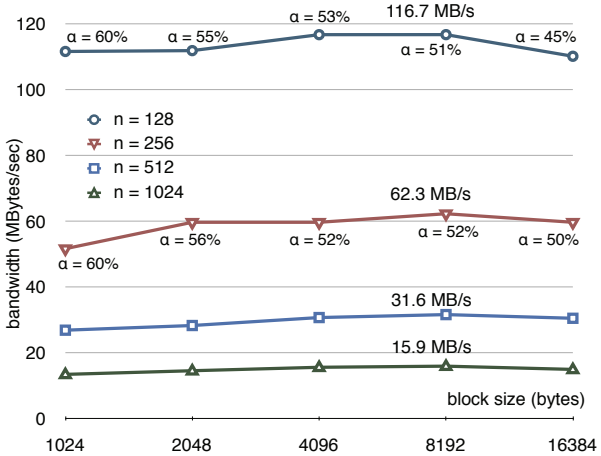

Fig. 6. Encoding bandwidth with the 8-core CPU and the 112-core 8800 GT GPU combined.

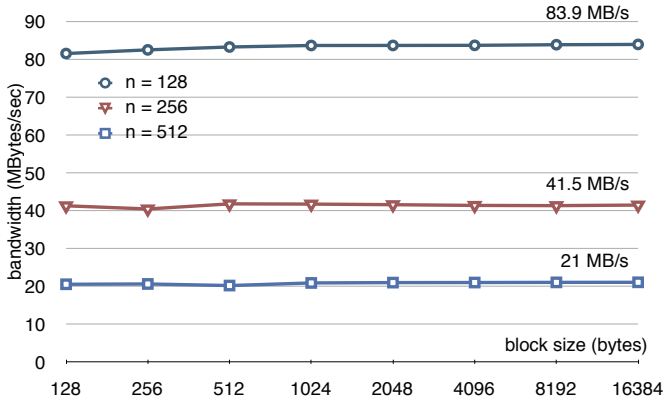

Fig. 7. Encoding bandwidth of the optimized table-based approach on the 8800 GT GPU.

This result certainly does not imply that the loop-based encoding on 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. In contrast, it is less likely to observe a substantial performance improvement of the shared memory on GPU SMs in the future.

D. Feasibility of using Network Coding on Streaming Servers

As we have just shown, the performance of *Nuclei* makes it feasible for it to be deployed in high-performance streaming servers using network coding, with hundreds of clients served concurrently. As an example, consider the scenario of using a media segment size of 512 KB, with 128 blocks of 4 KB each, corresponding to a $(n = 128, k = 4096)$ setting. 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. With *Nuclei* operating at this setting, the coding bandwidth is sufficiently high to serve up to 870 clients with the mainstream 8800 GT GPU alone. 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. Even with the modest memory capacity of 512 MB on the 8800 GT, hundreds of such segments can be accommodated.

The layout of segments and buffers in both graphics and system memory are shown in Fig. 8. The GPU keeps two other buffers to manage random seeds and their associated random coefficients. Coded blocks are directly loaded from the graphics memory as they are produced. The only involvement of the server CPU in network coding is to transfer new source segments to the GPU for encoding, to request a number of coded blocks from a particular segment in the GPU, and when they are ready, transfer them back to its buffers in the system memory. As such, the CPUs are relieved to perform other CPU-intensive tasks, such as media encoding.

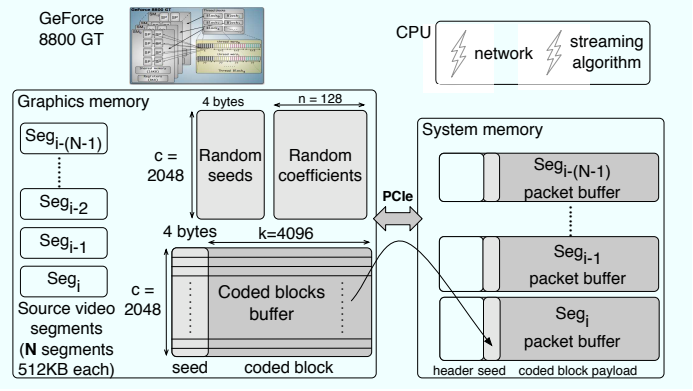

Fig. 8. GPU-accelerated network coding in a dedicated streaming server.

VI. Conclusion

This paper presents *Nuclei*, a high-performance design and implementation of many-core network coding using GPUs. *Nuclei* is able to achieve 90% of the advertised theoretical limits (504 GFLOPS) of the NVIDIA GeForce 8800 GT, across a wide range of network coding configurations. We have provided an in-depth comparison of GPU-accelerated network coding against a CPU-based implementation. We have shown that many-core GPUs can be deployed as an attractive alternative solution to multi-core servers, by offering a higher encoding performance level at a much lower cost. Furthermore, multi-core CPUs and many-core GPUs can be used to perform network coding simultaneously, which achieves a level of coding performance sufficient to saturate a Gigabit Ethernet interface and to serve in media streaming servers. With a much better memory performance than CPUs and a rapidly increasing number of cores, GPUs are very promising to bring network coding to reality.

References

[1] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung, "Network Information Flow," *IEEE Trans. on Information Theory*, vol. 46, July 2000.

[2] R. Koetter and M. Medard, "An Algebraic Approach to Network Coding," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 782–795, October 2003.

[3] C. Fragouli, J.-Y. Le Boudec, and J. Widmer, "Network Coding: An Instant Primer," *ACM SIGCOMM Computer Communication Review*, vol. 36, January 2006.

[4] P. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Proc. of Allerton Conference on Comm., Control, and Computing*, October 2003.

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

[6] T. Ho, R. Koetter, M. Medard, D. Karger, and M. Effros, "The Benefits of Coding over Routing in a Randomized Setting," in *Proc. of International Symposium on Information Theory (ISIT 2003)*, 2003.

[7] H. Shojania and B. Li, "Parallelized Network Coding With Hardware Acceleration," in *Proc. of the 15th IEEE International Workshop on Quality of Service (IWQoS)*, Chicago, IL, June 20-22 2007.

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

[9] 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.

[10] P. Maymounkov, N. Harvey, and D. Lun, "Methods for Efficient Network Coding," in *Proc. of 44th Annual Allerton Conference on Communication, Control, and Computing*, September 2006.

[11] E. Lindholm and J. Nickolls *et al.*, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," in *IEEE MICRO*, March-April 2008.

[12] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, 2008.

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

[14] Intel Corporation, *Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 1: Basic Architecture*, April 2008.