Implementation of Random Linear Network Coding Using NVIDIA's CUDA Toolkit

Péter Vingelmann[1] and Frank H.P. Fitzek[2]

[1] Budapest University of Technology and Economics
[2] Aalborg University, Department of Electronic Systems

Abstract. In this paper we describe an efficient GPU-based implementation of random linear network coding using NVIDIA's CUDA toolkit. The implementation takes advantage of the highly parallel nature of modern GPUs. The paper reports speed ups of 500% for encoding and 90% for decoding in comparison with a standard CPU-based implementation.

Keywords: Random Linear Network Coding, GPGPU, CUDA, parallelization.

1 Introduction and Motivation

Network coding is a relatively new research area, as the concept was just introduced in 2000 by Ahlswede, Cai, Li, and Yeung [1]. A large number of research works have been carried out looking into the different aspects of network coding and its potential applications in practical networking systems [7], [9], [10]. The authors in [3] provide an excellent summary of network coding. A brief definition of linear network coding could be: intermediate nodes in a packet network may send out packets that are linear combinations of previously received information.

This approach may provide the following benefits: i.) improved throughput, ii.) high degree of robustness, iii.) lower complexity and iv.) improved security.

As any other coding scheme, network coding can be used to deal with losses. In addition to that, network coding offers the possibility to recode packets at any intermediate node in the network. Traditional coding schemes only work end-to-end, so this is a unique feature of network coding which can be of great help whenever packet flows are intersecting as in fixed or meshed wireless networks. The potential advantages of using network coding are discussed in detail in [4], [6].

As stated beforehand the basic concepts of network coding has been shown in many research works. But only a small number of them considered the actual implementation of network coding together with complexity and resource constraints. Linear network coding requires enhanced computational capabilities and additional memory at the network nodes. The idea would be to utilize cheap computational power to increase network efficiency, as Moore's law suggests that processing power is becoming less and less expensive. However, the computational overhead introduced by network coding operations is not negligible and has become an obstacle to the real deployment of network coding.

T. Doulamis et al.: (Eds.): GridNets 2009, LNICST 25, pp. 131–138, 2010.

In our previous paper [12] we suggested to use the Graphics Processing Unit (GPU) for network coding calculations. We introduced a shader-based solution which yielded reasonably high throughputs. However, other GPU-based alternatives are also worth investigating.

The ideal solution would be to use a standard language and framework for GPU computing which is supported on a wide range of platforms. There exists an open standard for parallel programming of heterogeneous systems, it is called OpenCL (Open Computing Language) [5]. It is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. But right now, OpenCL is nothing more than a specification, since the standard has not been implemented yet (although both major GPU manufacturers, AMD and NVIDIA, have decided to fully support OpenCL in the future).

There is another solution which is currently available: NVIDIA offers a general purpose parallel computing toolkit called Compute Unified Device Architecture (CUDA) [2]. Although this toolkit can provide a very good performance, it has a major disadvantage: it only supports the last few generations of NVIDIA GPUs. This paper will introduce a CUDA-based implementation of Random Linear Network Coding (RLNC).

This work is organized as follows. Section 2 briefly describes the process of network coding. Section 3 provides an overview of NVIDIA's CUDA toolkit. Section 4 presents our CUDA-based implementation. Section 5 contains some measurement results, and Section 6 concludes this paper.

2 Concept of Random Linear Network Coding

The process of Network Coding can be divided into two separate parts, the encoder is responsible for creating coded packets from the original ones and the decoder transforms these packets back to the original format. The data to be sent can be divided into packets and a certain amount of these packets forms a generation. The whole data could be divided into several generations. The generation is a series of packets that are encoded and decoded together.

During encoding, linear combinations of data packets are formed based on random coefficients. All operations are performed over a Galois Field, in our case this is GF(2^8). Let N be the number of packets in a generation, and let L be the size (in bytes) of a single data packet. Each encoded packet contains a header (L bytes) and a payload (N bytes), the aggregate size is $N+L$ bytes. At least N linearly independent encoded packets are necessary to decode all the encoded data at the decoder side. There is a slight chance that the generated random coefficients are not linearly independent, thus the decoding needs additional encoded packets to be completed. The decoding itself can be done using a standard Gaussian elimination.

2.1 The Encoding Mechanism

After reading N number of L sized messages, the encoder is ready to produce encoded packets for this generation. The original data ($N \times L$ bytes) is stored in a matrix of corresponding dimension (matrix B on Figure 1).

Random coefficients are also necessary for the encoding process. Each encoded packet requires N coefficients, i.e. N random bytes. For an entire generation we need $N \times N$ bytes which are also stored in a matrix (matrix C on Figure 1).

The payload of each encoded packet is calculated by multiplying the header as a vector with the data matrix. This operation can be realized with simple array lookups and xor operations (as described later). Basically, encoding is a matrix multiplication performed in the GF domain, as it is depicted on Figure 1.

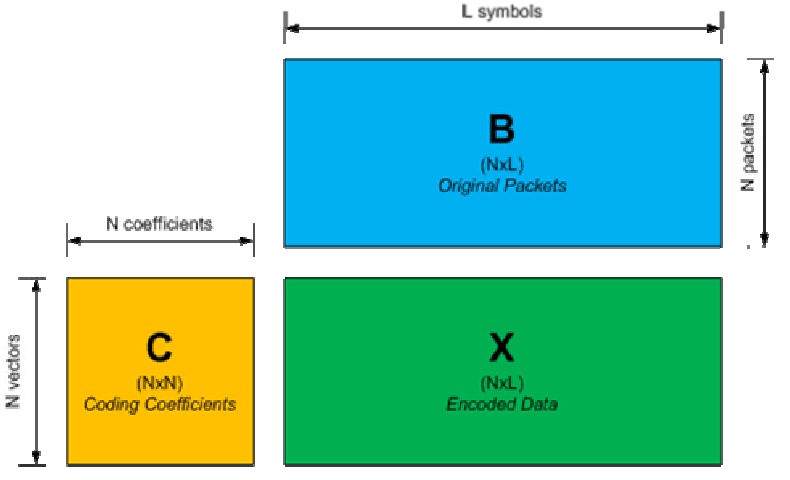

Fig.1. Encoding process in a matrix form

2.2 The Decoding Mechanism

The decoding algorithm used here is called Gauss-Jordan elimination which is basically an on-the-fly version of the standard Gaussian elimination. The encoded packets from the same generation are aggregated together, containing both the header and the payload part. Upon receiving a coded packet, the received data is being interpreted by using the previously received data. The elimination is based on the header part of the coded packet, but the corresponding operations are also performed on the payload part. The decoder stores the received, and partially decoded, data in an $N \times (N+L)$ sized decoding matrix. After the forward substitution part of the elimination each packet which carries new information will have a leading column in the header part with a non-zero pivot element, let's mark this column with K. This row is then normalized by dividing all of its elements by the leading value. After this step the new row can be inserted into the decoding matrix to the corresponding row (row K). The last step is to propagate this row back to the existing non-zero rows. The algorithm stops when the matrix does not have any empty rows, thence the header part forms an echelon form, and the payload part contains the decoded data in order.

3 NVIDIA's CUDA Toolkit

State-of-the-art 3D accelerator cards can be used to perform complex calculations, although they were originally designed to render 3D objects in real-time. A new concept is to use a modified form of a stream processor to allow a General Purpose Graphics Processing Unit (GPGPU) [8]. This concept turns the massive computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power. On recent NVIDIA GPUs, 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 [2]. GPUs have evolved into highly parallel, multi-threaded, multi-core processors. Unlike CPUs, which are originally designed for sequential computing, the design of GPUs is based on the *Single Instruction Multiple Data* (SIMD) architecture. It means that at any given clock cycle, multiple processor cores execute the same instruction, but they can operate on different data. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. This allows a low granularity decomposition of problems by assigning one thread to each data element (such as a pixel in an image or a cell in a grid-based computation). The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state.

In the CUDA model, the GPU is regarded as a data-parallel co-processor to the CPU. In CUDA terminology, the GPU is called *device*, whereas the CPU is called *host*. The device can only access the memory located on the device itself. A function executed on the device is called a *kernel*. A kernel is executed in the *Single Program Multiple Data* (SPMD) model, meaning that a user-specified number of threads execute the same program. Threads are organized into thread blocks which can have at most 512 threads. Threads belonging to the same thread block can share data through shared memory and can perform barrier synchronization. Furthermore, thread blocks can be organized into a grid. It is not possible to synchronize blocks within a grid. Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write scalable code. The number of thread blocks in a grid is typically dictated by the size of the data being processed rather than by the number of processors in the system, which it can greatly exceed. The basic scheduling unit in CUDA is called *warp*, which is formed by 32 parallel threads. A multiprocessor unit is only fully utilized if all 32 threads in the warp have the same execution path. If the threads in a warp have different execution paths due to conditional branching, the instructions will be serialized, resulting in long processing time. Threads should be carefully organized to achieve maximum performance.

4 Implementation

The fundamental question is how to realize the Galois Field arithmetics in CUDA. If the field size is a power of 2, then addition and subtraction in the Galois Field are identical with the exclusive OR (XOR) operation, and this can be performed natively

on the GPU. On the other hand, multiplication and division are more complicated over GF(2^8). These operations can be performed procedurally using a loop-based approach. However, it would not be efficient to compute the results every single time, a lot of clock cycles would be wasted this way. The other solution is to pre-calculate the results and store them in tables. The field size is $2^8=256$, so the multiplication and division tables occupy *256x256 bytes = 65 kB* each. These tables can be stored in graphics memory, and they can be bound to CUDA texture references to facilitate fast array look-ups. Two-dimensional texture coordinates are used to pinpoint a specific texture element which is the result of the multiplication or division.

4.1 Encoding

The encoding process can be considered as a highly parallel computation problem, because it essentially consists of a matrix multiplication in the GF domain. A parallel implementation is possible with little or no communication and synchronization among threads. Encoding of multiple coded packets - and even different sections of the same coded packet - can proceed in parallel by using a large number of threads. In CUDA, the GPU can only access the graphics memory, so all the coefficients and original packets have to be transferred from the host to the graphics memory first. Similarly, encoding results residing in graphics memory need to be transferred back to system memory. This imposes an additional overhead, but fortunately CUDA provides very efficient memory management functions.

The most essential design question here is how to partition the encoding task among the threads. We could launch one thread per coded packet (i.e. N threads for a generation), but this approach is simply not parallel enough. We can achieve higher performance with a much finer granularity, with each GPU thread encoding only a single byte of the coded packet, rather than working on an entire packet. This way $N \times L$ threads are necessary for encoding a whole generation. Performance improves significantly because GPUs are designed to create, manage, and execute concurrent threads with (almost) zero scheduling overhead.

The next measure towards further optimization is to process original packet data in 4-byte chunks, rather than byte–by–byte. Thereby we can reduce the number of memory accesses on the GPU. Each thread is responsible for computing a 4-byte chunk (i.e. a 32-bit integer) of the resulting encoded packets. Thus, we need $N \times L/4$ threads for a whole generation. The CUDA-based encoding implementation uses random coefficients generated by the CPU, and transferred to the graphics memory before the start of the encoding process. Using this approach it is possible to compare the CUDA computation results with the reference results computed by an equivalent CPU-based implementation.

4.2 Decoding

The decoding process has a higher computational complexity than encoding. This leads to a reduced decoding performance in general. The efficient parallelization of the decoding process is a real challenge in CUDA. The fundamental problem with the Gauss-Jordan elimination is that the decoding of each coded packet can only start after the decoding of the previous coded packets has finished, i.e. it is essentially a

sequential algorithm. Parallelization is only possible within the decoding of the current coded packet, and not across the whole generation as with the encoding process. The GPU needs to run thousands of threads to be able to achieve its peak performance, consequently the performance gain of GPU-based decoding is limited.

We receive each coded packet along with its associated coefficients (i.e. the encoding vector) and decode it partially. After receiving and decoding the N linearly independent coded packets, the decoding process completes and all the original packets are recovered.

For every new coded packet, we can partition the aggregate $N+L$ coefficients and data, such that each byte of the aggregate data is assigned to a thread, leading to a total of $N+L$ threads. Each thread reduces the leading coefficients of the new coded packet through a number of linear combinations. We multiply the previously decoded packets so that their leading coefficient matches the corresponding coefficient in the newly arrived coded packet. This way we can reduce a few of its coefficients to zeros. This multiplication-based approach can be parallelized, because we can reduce each coefficient using only the initial values of the other coefficients.

After completing this reduction step, a global search for the first non-zero coefficient becomes necessary. It is most likely that the position of this pivot element will match the current rank of the decoding matrix, but we cannot be sure about this. Running this global search is a major issue, since CUDA's synchronization construct only works for threads within a single thread block (max. 256 threads), and not among all GPU threads, therefore we are forced to perform this synchronization at the CPU side. This effectively breaks the decoding process into two separate CUDA kernels. Of course, if we want to develop a fully GPU-based solution for decoding, we could omit the CPU-side synchronization by introducing a new CUDA kernel for finding the pivot element. But it could run in only one thread, so there is no real benefit in doing this. Also note that launching another kernel means additional overhead, i.e. lower performance.

After finding the first non-zero coefficient at the CPU side, we launch another CUDA kernel to perform the remaining decoding operations, i.e. the backward substitution. But if we cannot find the pivot element (i.e. all coefficients were reduced to zeros), then the packet was linearly dependent, hence it is not necessary to launch this second kernel. The backward substitution part can be easily parallelized: we can assign a thread to each affected element of the decoding matrix (we should consider only the non-zero rows). This approach leads to a total of $N \times (N+L)$ threads when we process the very last packet of a generation, otherwise this number can be lower.

The first task of this second kernel is to normalize the reduced packet, and subsequently insert it into the decoding matrix. If the actual thread's index points to the correct row, the thread simply inserts the normalized data to that position. Otherwise, it has to perform a linear combination: we multiply the normalized packet with the coefficient at the pivot position in the current row, then this row is xored with this product. Note that each GPU thread affects only one byte of the decoding matrix. Extra care must be taken when we manipulate the coefficients at the pivot position in each row. Because the relative execution order of threads is not known, the initial values of these coefficients (i.e. one column of the decoding matrix) must be saved into a separate array. Thereby we can guarantee correct computation results.

5 Results

The following measurements are done on different GPUs with different generation sizes (*N*). The packet size is always equal to 1024 (*L=1024*). The following numbers indicate the measured throughput values in megabytes/seconds for encoding and decoding separately. Note that these are rounded averages of several measurements.

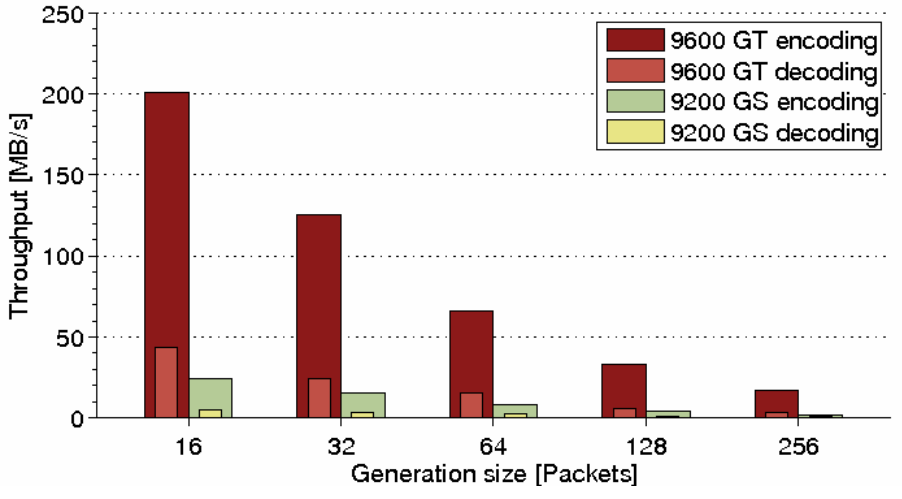

Fig. 2. Measurements performed on NVIDIA GeForce 9600GT and 9200M GS graphics cards

Note that the GeForce 9600GT can be considered a middle-class GPU, and the 9200M GS belongs to the low class. If we compare the results with the other implementations presented in our previous paper [12], then we may notice the significantly higher encoding throughputs. In some cases the CUDA implementation outperforms the CPU implementation by an order of magnitude, and it is twice as fast as the shader-based solution. The encoding algorithm is relatively simple, thereby its CUDA implementation can be considered straightforward, and we can obtain near-optimal encoding performance. On the other hand, the decoding throughput values are not significantly higher than those of the shader-based solution. This indicates that the current implementation is sub-optimal, so finding an optimal decoding solution for the GPU remains an open question for the future.

6 Conclusion

We introduced a CUDA-based implementation of Random Linear Network Coding which yields reasonably high throughput values. In fact, the numbers look promising compared to our previous implementations. We intend to port one of our GPU-based implementations onto mobile devices as soon as possible. At the moment, the GPUs on these devices lack certain capabilities required to run our application.

From a theoretical point of view, there is another interesting idea which might lead to simpler (and faster) implementations. The authors in [11] suggest that it might be beneficial to use a smaller Galois Field such as GF(2), despite the fact that the probability of generating linearly dependent packets increases significantly. On the other hand, the computational complexity decreases dramatically. Combining this idea with a GPU-based implementation can lead to very high throughput values in the future.

References

1. Ahlswede, R., Cai, N., Li, S.-Y.R., Yeung, R.W.: Network information flow. IEEE Transactions on Information Theory 46(4), 1204–1216 (2000)
2. NVIDIA Corporation. NVIDIA CUDA: Programming Guide, Version 2.0 (July 2008)
3. Fragouli, C., Le Boudec, J.-Y., Widmer, J.: Network coding: an instant primer. SIGCOMM Comput. Commun. Rev. 36(1), 63–68 (2006)
4. Fragouli, C., Soljanin, E.: Network Coding Applications. Now Publishers Inc. (January 2008)
5. Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems (February 2009)
6. Ho, T., Lun, D.: Network Coding: An Introduction. Cambridge University Press, Cambridge (2008)
7. Katti, S., Rahul, H., Hu, W., Katabi, D., Médard, M., Crowcroft, J.: Xors in the air: practical wireless network coding. SIGCOMM Comput. Commun. Rev. 36(4), 243–254 (2006)
8. Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., Lefohn, A.: GPGPU: general purpose computation on graphics hardware. In: SIGGRAPH 2004: ACM SIGGRAPH 2004 Course Notes. ACM Press, New York (2004)
9. Matsumoto, R.: Construction algorithm for network error-correcting codes attaining the singleton bound E 90-A(9), 1729 (2007)
10. Park, J.-S., Gerla, M., Lun, D.S., Yi, Y., Medard, M.: Codecast: a network-coding-based ad hoc multicast protocol. IEEE Wireless Communications 13(5), 76–81 (2006)
11. Heide, J., Pedersen, M.V., Fitzek, F.H.P.: Torben Larsen: Network Coding for Mobile Devices – Systematic Binary Random Rateless Codes. In: The IEEE International Conference on Communications (ICC), Dresden, Germany, June 14-18 (2009)
12. Vingelmann, P., Zanaty, P., Fitzek, F.H.P., Charaf, H.: Implementation of Random Linear Network Coding on OpenGL-enabled Graphics Cards. In: European Wireless 2009, Aalborg, Denmark, May 17-20, pp. 118–123 (2009)