Implementation of Random Linear Network Coding on OpenGL-enabled Graphics Cards

Péter Vingelmann*, Péter Zanaty*, Frank H. P. Fitzek† and Hassan Charaf*
*Budapest University of Technology and Economics
†Aalborg University, Department of Electronic Systems

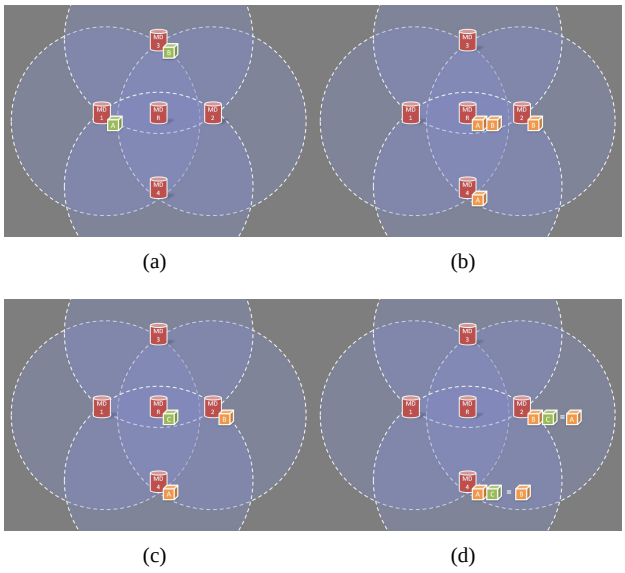
***Abstract*— This paper describes the implementation of network coding on OpenGL-enabled graphics cards. Network coding is an interesting approach to increase the capacity and robustness in multi-hop networks. The current problem is to implement random linear network coding on mobile devices which are limited in computational power, energy, and memory. Some mobile devices are equipped with a 3D graphics accelerator, which could be used to do most of the RLNC related calculations. Such a cross-over have already been used in computationally demanding research tasks as in physics or medicine. As a first step the paper focuses on the implementation of RLNC using the OpenGL library and NVidia's Cg toolkit on desktop PCs and laptops. Several measurement results show that the implementation on the graphics accelerator is outperforming the CPU by a significant margin. The OpenGL implementation performs relatively better with larger generation sizes due to the parallel nature of GPUs. Therefore the paper shows an appealing solution for the future to perform network coding on mobile devices.**

I. Introduction and motivation

Network coding is getting more and more attention lately. After the introduction of the concept by Ahlswede in 2002 [1], a large number of research works has been carried out looking into the different aspects of network coding. [2] The majority of research papers in the field deals with the construction of network codes [13], [9] and the application of network coding to different communication scenarios [7], [11], namely fixed networks, meshed wireless network, underwater communication, and many more. The idea behind network coding is to take several original packets and combine them into one coded packet, which has the length of one of the original packets. If a sender is combining N packets, the receiver needs to receive at least N coded packets successfully to decode the original packets. Each coded packet is holding the so-called encoding vector to give the receiver the knowledge which packets have been combined. Thus, as any other coding scheme, network coding can be used to deal with erasures. In addition to that, network coding offers the possibility to recode packets inside the network. In contrast to other coding schemes that only work end to end, each node has the possibility to recombine coded packets into a new coded packet. This feature is of great help whenever packet flows are intersecting as in fixed or meshed wireless networks.

In order to understand this feature we give the following example. Consider the communication topology of a wireless meshed network with five nodes given in Figure 1(a): MD1 is sending packets to MD2 using MD R as a relay due to the fact that a direct communication between MD1 and MD2 is not possible. The communication of MD1 and MD2 is part of an ongoing stream A, whereas the sender and source are not depicted here. The same holds for MD3 and MD4 working on stream B. Without network coding the relay would easily become the bottleneck forwarding packets of stream A and B. In case both streams are working at the capacity limit, the data rate after the relay would be half of the original one. Using network coding, we would apply some sort of inter flow coding for each stream. That means packets of stream A are coded mostly probably already at the sender and decoded at the receiver. The same applies for stream B. The interflow coding assures the robustness for the given stream. On top of the inter flow coding, we will apply an intra flow coding. Due to the given scenario, MD2 can overhear packets of MD3 and MD4 is doing the same with MD1. Whenever MD1 and MD3 are transmitting packets of their given stream A and B, the relay MD R will probably receive both packets. Also MD 2 and MD 3 will receive packets even though those packets do not belong to their stream (see Figure 1(b)). The relay MD R will take both packets and code these together into a new coded packet referred to as packet C. The coding could be a simple XOR of both packets (see Figure 1(c)). Packet C will be broadcast to MD2 and MD4. At each receiver the packet C will be XORed again with the packet A or B. E.g. MD4 has packet A before (which did not belong to stream B), which will become packet B after the XORing with packet C (see Figure 1(d)). At MD2 the coded packet allows to retrieve packet A. Thus, with one packet transmission, the relay satisfied two nodes at the same time.

As these first examples are only addressing the advantages of network coding in a nutshell, the interested readers are referred to [3], [5]. As stated beforehand the basic concepts of network coding has been shown in many research works. But only a small number of research works has been carried out in the actual implementation of network coding considering the complexity and resource constraints. The implication of network coding on the computational power, the memory and energy consumption is of special interests if network coding is performed on mobile devices, such as commercial mobile phones [12] or small wireless sensors [6]. The energy consumption is of special interest on battery driven devices to optimize their operational time. The energy consumption related to network coding can be divided into two different fields.



(a) (b)

(c) (d)

Fig. 1. Different network topologies

- The first field is on the protocol design for network coding. As the coded packets are sent over the wireless medium, the strategies describing which packets to be sent, received or just overheard have a large impact on the energy consumption [4].
- The second part deals with the way how the actual encoding and decoding process is implemented or where it is carried out. Different forms of network coding such as XOR coding or random linear network coding (RLNC) have different impact on the hardware performing the actual coding.

The first implementation of network coding on mobile phones was presented in [12] and extended in [4], where XOR was used in [12] and RLNC in [4]. Another work has implemented the COPE algorithm [10] on the N810 tablet. All three works have performed the actual network coding on the CPU of the wireless devices. Doing simple XOR network coding, such as COPE [7] is doing, has nearly no impact on the energy consumption regarding the coding. Here the most energy was used for the wireless communication. One XOR combination of two packets of size 1000 byte results in the energy consumption of 191 nJ [12]. The transmission of a packet with the same length, over the IEEE802.11 network using the standard chipset on Nokia N95 devices, consumed 2.31 mJ. These numbers look quite promising for the usage of network coding on mobile devices. Nevertheless, most applications of network coding are not using the simple XOR mechanism, but they rather use the RNLC approach. This approach has some advantages as it can operate even without any feedback information. On the other side the encoding and decoding process is demanding a lot of computational power. One of the first implementations of network coding on the S60 platform (Nokia N95) yielded just a couple of hundreds of kbit/s of encoding speed. The general way to tackle this problem is to introduce special hardware doing the coding job. This has been done for the voice codec and the video codec on the mobile phone. As a thumb rule the energy consumption on dedicated hardware is a tenth of what a software solution would take. Unfortunately there is no dedicated network coding hardware on the mobile phones, yet.

Therefore this paper will look into the possibility to use the 3D graphics accelerator (GPU) for the number crunching needed by network coding. 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. Their fixed-function pipeline is now replaced by a programmable one, so developers can write short programs called shaders that run on the GPU. As a matter of fact, graphics card manufacturers encourage developers to make use of the GPU's computational potential. For example, the GPU can be used for advanced physics simulations as it is described in [8]. Two software interfaces are necessary to use the programmable pipeline:

1) A regular software interface to the GPU (OpenGL or Direct3D)
2) A shading language to write shaders (DirectX HLSL, GLSL, NVidia Cg toolkit)

As a very first step network coding is implemented on PC platform using the OpenGL API and NVidia CG toolkit (as a shading language). Doing so allows us to determine and compare the coding speed for the CPU and the GPU on the PC. Later this solution can be ported to OpenGL-capable mobile devices.

II. Concept of Random Linear Network Coding

The process of Network Coding can be divided into two separate parts, the encoder is responsible to create a coded message from the original one and the decoder transforms these messages back to the original format.

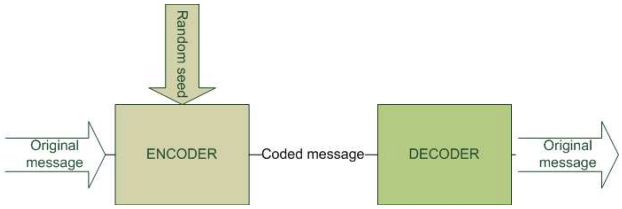


Fig. 2. Overview of Network Coding.

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 more generations. The generation is a series of packets that are encoded and decoded together. During the encoding, linear combinations of data packets are formed based on random coefficients. Let N be the number of packets in a generation, and let P be the size of a single data packet. The header acts as the random coefficient matrix, and each encoded packet has the size of N+P bytes.

The encoded packets from the same generation are aggregated together, containing both the header data, and the encoded payload. When N encoded packets are received the encoded data could be decoded. There is a slight chance that the generated random coefficients are not linearly independent,

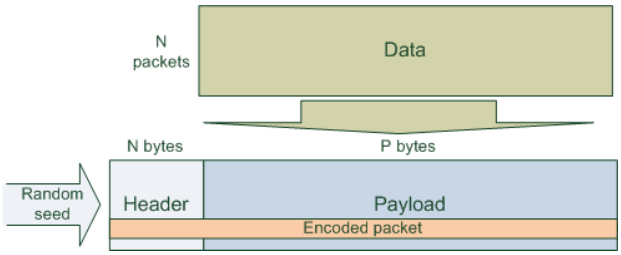


Fig. 3. A generation.

thus the decoding needs additional encoded data packets to be completed. The decoding itself can be done using a standard Gaussian elimination.

A. OpenGL

OpenGL (Open Graphics Library) is a standard specification defining a cross-language cross-platform API for writing applications that produce 2D and 3D computer graphics. The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, and flight simulation. It is also used in video games, where it competes with Direct3D on Microsoft Windows platforms.

OpenGL serves two main purposes:

1) To hide the complexities of interfacing with different 3D accelerators, by presenting the programmer with a single, uniform API.
2) To hide the differing capabilities of hardware platforms, by requiring that all implementations support the full OpenGL feature set (using software emulation if necessary).

OpenGL's basic operation is to accept primitives such as points, lines and polygons, and convert them into pixels. This is done by a graphics pipeline known as the OpenGL state machine. Most OpenGL commands either issue primitives to the graphics pipeline, or configure how the pipeline processes these primitives. Prior to the introduction of OpenGL 2.0, each stage of the pipeline performed a fixed function and was configurable only within tight limits. OpenGL 2.0 offers several stages that are fully programmable using a shading language. OpenGL is a low-level, procedural API, requiring the programmer to dictate the exact steps required to render a scene. OpenGL's low-level design requires programmers to have a good knowledge of the graphics pipeline, but also gives a certain amount of freedom to implement novel rendering algorithms on the GPU. Many modern 3D accelerators provide functionality far above the fixed-function pipeline, but these new features are generally enhancements of this basic pipeline rather than radical revisions of it.

B. Shaders

A shader is a short program, a set of software instructions, which is originally used to calculate rendering effects on graphics hardware with a high degree of flexibility. Shaders are used to program the graphics processing unit (GPU) programmable rendering pipeline, which has mostly superseded the fixed-function pipeline that allowed only common geometry transformation and pixel shading functions; with shaders, customized calculations can be performed. As Graphics Processing Units evolved, major graphics software libraries such as OpenGL and Direct3D began to exhibit enhanced ability to program these new GPUs by defining special shading functions in their API.

1) Types of shaders: The Direct3D and OpenGL graphic libraries use three types of shaders.

- Vertex shaders are run once for each vertex given to the graphics processor. The purpose is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen (as well as a depth value for the Z-buffer). Vertex shaders can manipulate properties such as position, color, and texture coordinate, but cannot create new vertices. The output of the vertex shader goes to the next stage in the pipeline, which is either a geometry shader if present or the rasterizer otherwise.
- Geometry shaders can add and remove vertices from a mesh. Geometry shaders can be used to generate geometry procedurally or to add volumetric detail to existing meshes that would be too costly to process on the CPU. If geometry shaders are being used, the output is then sent to the rasterizer.
- Pixel shaders, also known as fragment shaders, calculate the color of individual pixels. The input to this stage comes from the rasterizer, which fills in the polygons being sent through the graphics pipeline. Pixel shaders are typically used for scene lighting and related effects such as bump mapping and color toning. (Direct3D uses the term "pixel shader," while OpenGL uses the term "fragment shader." The latter is arguably more correct, as there is not a one-to-one relationship between calls to the pixel shader and pixels on the screen. The most common reason for this is that pixel shaders are often called many times per pixel for every object that is in the corresponding space, even if it is occluded; the Z-buffer sorts this out later.)

2) Parallel processing: Shaders are written to apply transformations to a large set of elements at a time, for example, to each pixel in an area of the screen, or for every vertex of a model. This is well suited to parallel processing, and most modern GPUs have a multi-core design to facilitate this, vastly improving efficiency of processing.

A shading language and a corresponding framework is necessary to write a shader, compile it and execute it on the GPU. There are several alternatives available:

1) HLSL
2) GLSL
3) NVIDIA's CG toolkit

Probably the most popular one is Microsoft's HLSL, which is only for Direct3D. Although OpenGL has its own shading language called GLSL, we decided not to use it, because

NVidia's CG toolkit offers a better alternative in terms of performance and platform-independence.

C. NVidia CG toolkit

Cg or C for Graphics is a high-level shading language developed by Nvidia for programming vertex and pixel shaders. It is very similar to Microsoft's HLSL. Cg is based on the C programming language and although they share the same syntax, some features of C were modified and new data types were added to make Cg more suitable for programming graphics processing units. Note that this language is only suitable for GPU programming and it does not replace a general programming language. The most appealing feature of the CG toolkit is its cross-platform nature. It works on the following operating systems: Windows (Vista, XP, 2000), MacOS X, Linux, Solaris, and it is compatible with both OpenGL and Direct3D.

III. Implementation

We have implemented the same algorithm both on the CPU and the GPU. Our main goal was to prove that the GPU is capable of encoding and decoding data using Random Linear Network Coding. Moreover, we wanted to compare the performance of these implementations. Note that the Galois Field(2^8) is used in both implementations.

A. Reference implementation on the CPU

The C programming language is used to implement the following algorithms on the CPU. The field arithmetic on the Galois Field(2^8) is realized with two static byte arrays, one for multiplication and one for division. Addition and subtraction are the same operation in this field, and are identical with the exclusive OR (XOR) operation on the CPU.

B. The encoding mechanism

After reading N number of P sized messages, the encoder is ready to produce encoded messages for this generation. This $N * P$ bytes of data chunk is stored in a matrix of corresponding dimensions. For the coding it needs a random seed. In this paper we are not going into details about the connection between the entropy of this seed and the quality of the network coding. From this seed the generator gets an N byte long header for each encoded packet, and calculates the payload by multiplying the header as a vector with the data matrix. This operation is realized with simple array lookups and xor operations. The cost of getting a coded message is $O(NP)$. At least N encoded message should be delivered to the decoder side to be able to decode the message. Therefore the total computing cost of transmitting $N * P$ bytes of data is $O(N^2P)$, and the transmitted overload is $N * N$ bytes.

C. The decoding mechanism

Upon receiving a coded message, the received data is being interpreted by using the previous data. Basically the decoding algorithm is a step by step Gaussian elimination over the Galois Field with back propagation. The elimination is based on the header part of the coded message, but the corresponding operations are also done on the payload part. The decoder stores the received, and partially decoded, data in an $N*(N+P)$ size matrix. After the forward substitution part of the elimination each message which carries new information will have a leading column in the header part with a nonzero pivot element, let's mark this column with L. 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 L). The last step is to propagate this row back to the existing nonzero 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 of contains the decoded data in order. The cost of encoding a row is $O(N(N+P))$ for the forward substitution, $O(N)$ for the pivot search, $O(N+P)$ for the normalization and $O(N(N+P))$ for the backward substitution, therefore totally $O(N(N+P))$ for each row. The total computational cost is then $O(N^3+N^2P)$.

D. Basic Operations on the GPU

Porting such a complex algorithm to the GPU is not a straightforward task, because the usual arithmetic operations and control structures are not available. It is very important to notice that shaders don't work in a sequential manner, pixels are rendered in parallel by the currently activated shaders. Directly reading and writing memory in a shader is not possible, the only way to store and retrieve data is by using textures. From our point of view, textures behave as two-dimensional arrays of bytes. It is possible to read certain elements of this array by performing texture lookup operations. Two dimensional vectors (called texture coordinates) are used to specify which array element is to be read. These texture coordinates are always floating-point numbers, so extra care must be taken to ensure that they are rounded to the correct integer indices. The most serious limitation is that no data can be written into a texture in a shader. OpenGL only offers functions to copy data from normal memory to a texture and from the framebuffer to a texture.

Another important limitation is the lack of basic arithmetic operations in shader code. For example, there is no bitwise XOR operation, so it must be simulated by a texture lookup. A 256x256-sized texture is necessary for this: the first texture coordinate will be the first operand of the XOR operation, and the second coordinate will be the other operand. These coordinates identify a texture element which is the result of the operation. Notice that any two-operand function can be simulated like this, if its domain and range fits certain limitations.

The generation size (N) cannot be an arbitrary value, because both texture dimensions must be powers of 2. Moreover, $N+P$ should also be a power of 2, since it is length of an encoded packet.

E. Encoder on the GPU

The encoder implementation is relatively simple, it uses only one shader to perform the necessary calculations. Its task

is to generate a texture whose rows are linear combinations of the rows of the original image. An NxN random matrix contains the coefficients to form the linear combinations. This random matrix is generated at each application startup.

Multiplications are performed over GF(2^8), and this operation is implemented by a texture lookup similarly to the XOR function described above. Again a 256x256 texture contains the results for the multiplication, and operands are specified by two texture coordinates. The XOR texture is also used here to perform addition over the Galois Field.

The encoded image is rendered row-by-row as GL_LINES primitives. Every row is rendered exactly N times using the encoder shader. This shader performs the following steps:

1) Get the multiplication coefficient from the random matrix
2) Multiply it with the next row of the original image
3) Xor this product with the previous results for this row

Of course, every row is rendered onto the framebuffer and the resulting pixels must be copied back to the result texture after completing Step 3. So these steps are done N times for each row. After this, the next row of the random matrix is selected to calculate the next line of the result texture.

It means NxN line primitives are rendered to encode the whole image, and it would mean NxN OpenGL calls on the CPU. Fortunately OpenGL offers the possibility to precompile these API calls into a display list. All these instructions can be executed by calling this pre-compiled display list. It means that the whole image can be encoded by a single API call on the CPU. But before calling this function, the shader and its parameters must be set up properly. The resulting encoded texture serves as the input for the decoder.

F. Decoder on the GPU

The decoder is composed of 3 different shaders that implement the 3 consecutive phases of the Gaussian elimination:

1) phase: Forward substitution: reduce the new data row by the existing rows
2) phase: Find the pivot point in the reduced row
3) phase: Backward substitute this normalized row into the existing rows

An Nx(N+P) sized texture represents the internal structure of the decoder, and this texture is propagated through all 3 phases. The end result after processing N linearly independent packets should be an echelon form in left NxN segment and the original image in the right NxP segment.

The 1st phase introduces a new operation: division over the Galois Field. It is implemented using the same technique that is used for xoring and multiplication. This is also a row-by-row rendering, where each row signifies a new step in the reduction of the newly arrived packet. The algorithm enumerates all basis vectors that are represented in the decoder matrix, so the new packet is normalized and xored with the corresponding basis vector. Note that these operations are performed both on the header and the payload part of the packet. OpenGL API calls in this phase are compiled into a display list, thus it can be executed by a single call on the CPU.

The task in the 2nd phase is to find the pivot element (if any) in the reduced new packet. It is implemented using a simple shader which enumerates all elements of this packet, and stores the position and value of the first non-zero element into a special pixel. In this phase rendering is performed point-by-point, because data elements must be processed in a sequential manner. The pivot point position and value is used in the next phase.

The last phase is backward propagation. The 3rd shader is responsible for putting the new packet to the right place (based on its pivot point) in the decoder matrix, and for reducing the existing rows with the new one. If there is no pivot point in the new packet (i.e. all elements are zero), then this shader will have no effect at all. The new packet has to be normalized, i.e. divided by its pivot point value before insertion to the decoder matrix. Rendering is done row-by-row: every row of the decoder matrix is xored with the normalized packet multiplied by the corresponding element in the actual row. The end result of backward propagation is the echelon form on the left, and the original data on the right after N linearly independent packets have been processed.

G. GUI

We have developed a simple GUI to see the actual results of the computations. This GUI uses both the CPU and the GPU implementation to encode and decode the same data. Thereby it is possible to compare the resulting images and verify the GPU algorithm's correctness. The bottom row shows calculation results on the CPU, whereas the top row shows the results of the same calculations performed on the GPU.

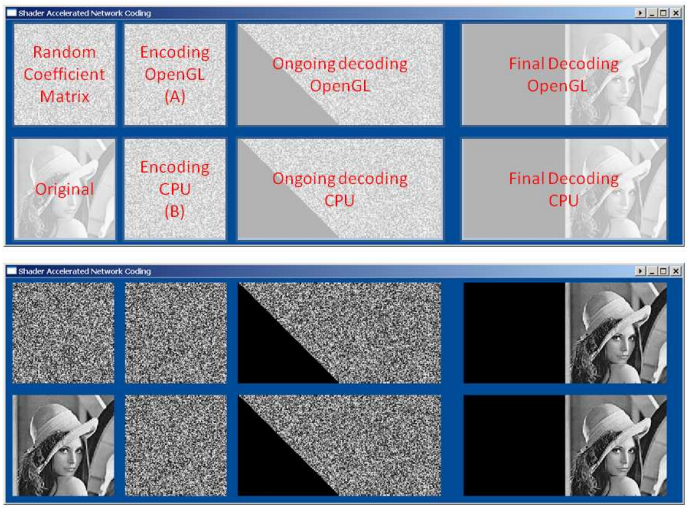


Fig. 4. Graphical User Interface.

Figure 4 shows the graphical user interface, and labels what data is visible on which part of the screen.

IV. Results

The following measurements are done on different platforms with different generation sizes (N). For simplicity, the packet size is always equal to 1024 ($P = 1024$). 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.

A. CPU implementation

The CPU implementation runs only on one thread, so it utilizes only a single CPU core even if there are more available.

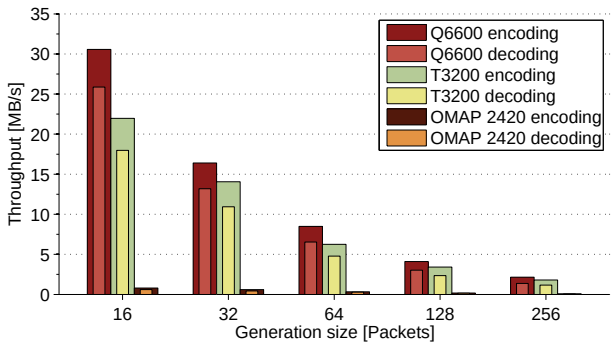


Fig. 5. CPU measurements performed on Intel Core2 Quad CPU Q6600 2.40 GHz, Intel Pentium Dual CPU T3200 2.00 GHz, and Texas Instruments OMAP 2420 400 MHz processors

The tendency is the same on all platforms: the throughput is approximately a first-order function of the generation size. This observation is in accordance with the cost formulae presented previously, since the cost of encoding or decoding a single packet ($O(NP)$ and $O(N(N+P))$ respectively) theoretically determines the achievable throughputs. Decoding performance is about 15-30% lower than encoding performance. This fact can also be justified theoretically by the same cost formulae, noticing that $N+P$ is only slightly larger than P itself, as $N << P$ in most cases.

B. GPU implementation

Note that the times for shader execution and memory transfers are measured on the GPUs, but shader initialization and parameter setup are not considered here.

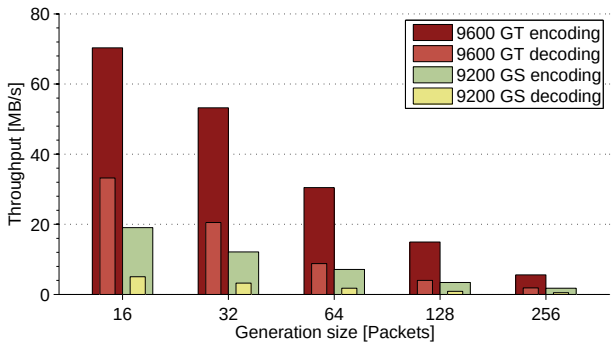


Fig. 6. GPU measurements performed on NVIDIA GeForce 9600GT and 9200M GS graphics cards

It is important to notice that shader execution times don't increase linearly with the generation size. It is relatively more efficient to encode and decode large generations than small ones. This fact can be explained if we consider the highly parallel nature of the GPU. As the generation size increases the number of rendered pixels increases as well resulting in a higher degree of parallelism.

The second observation is that the decoding throughputs are only fractions of the encoding values (always less than 50%). This is because the decoding algorithm cannot be fully parallelized. Hence, we cannot take advantage of the full potential of the GPU.

V. Conclusion

Our work shows that it is possible to implement Random Linear Network Coding on state-of-the-art graphics cards. We intend to port this implementation onto OpenGL-enabled mobile devices. These devices also have a programmable GPU pipeline, but their capabilities are limited compared to graphics cards in the PC. Although these results seem very promising, a more native approach like NVIDIA's CUDA toolkit could yield much better results. Unfortunately, none of the state-of-the-art mobile devices support CUDA.

References

[1] R. Ahlswede, Ning Cai, S.-Y.R. Li, and R.W. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, Jul 2000.

[2] Christina Fragouli, Jean-Yves Le Boudec, and Jörg Widmer. Network coding: an instant primer. *SIGCOMM Comput. Commun. Rev.*, 36(1):63–68, 2006.

[3] Christina Fragouli and Emina Soljanin. *Network Coding Applications*. Now Publishers Inc, January 2008.

[4] Janus Heide, Morten V. Pedersen, Frank H.P. Fitzek, and Torben Larsen. Cautious view on network coding - from theory to practice. *Journal of Communications and Networks (JCN)*, 2009.

[5] Tracey Ho and Desmond Lun. *Network Coding: An Introduction*. Cambridge University Press, 2008.

[6] R. Jacobsen, K. Jakobsen, P. Ingtoft, T. Madsen, and F.H.P. Fitzek. Practical Evaluation of Partial Network Coding in Wireless Sensor Networks. In *4th International Mobile Multimedia Communications Conference (MobiMedia 2008)*, Oulu, Finland, July 2008. ICTS/ACM.

[7] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. Xors in the air: practical wireless network coding. *SIGCOMM Comput. Commun. Rev.*, 36(4):243–254, 2006.

[8] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, New York, NY, USA, 2004. ACM Press.

[9] Ryutaroh Matsumoto. Construction algorithm for network error-correcting codes attaining the singleton bound. *VOL.E 90-A*, 9:1729, 2007.

[10] K.F. Nielsen, T.K. Madsen, and F.H.P. Fitzek. Network coding opportunities for wireless grids formed by mobile devices. *The Second International Conference on Networks for Grid Applications, ICST*, 2008.

[11] Joon-Sang Park, M. Gerla, D.S. Lun, Yunjung Yi, and M. Medard. Codecast: a network-coding-based ad hoc multicast protocol. *Wireless Communications, IEEE*, 13(5):76–81, October 2006.

[12] M.V. Pedersen, F.H.P. Fitzek, and Torben Larsen. Implementation and performance evaluation of network coding for cooperative mobile devices. *Communications Workshops, 2008. ICC Workshops '08. IEEE International Conference on*, pages 91–96, May 2008.

[13] Peter Sanders, Sebastian Egner, and Ludo Tolhuizen. Polynomial time algorithms for network information flow. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 286–294, New York, NY, USA, 2003. ACM.