LDPC Error Correction for Gbit/s QKD [1]

Alan Mink and Anastase Nakassis
Information Technology Laboratory, National Institute of Standards and Technology,
100 Bureau Dr., Gaithersburg, MD 20899
amink@nist.gov, anakassis@nist.gov

Abstract

Low Density Parity Check (LDPC) error correction is a one-way algorithm that has become popular for quantum key distribution (QKD) post-processing. Graphic processing units (GPUs) provide an interesting attached platform that may deliver high rates of error correction performance for QKD. We present the details of our various LDPC GPU implementations and both error correction and execution throughput performance that each achieves. We also discuss the potential for implementation on a GPU platform to achieve Gbit/s throughput.

Keywords: Quantum key distribution, QKD reconciliation, Entropy, LDPC codes, GPU

1. Introduction

Low Density Parity Check (LDPC) error correction [7, 10] is a one-way algorithm that has become popular for quantum key distribution (QKD) [1] post-processing. The QKD protocol uses an unsecured quantum channel and an unsecured, but authenticated and integrity protected classical channel to establish a shared secret between two parties, Alice and Bob, even in the presence of an eavesdropper, Eve. There are four stages to the QKD protocol. Stage 1 is where a quantum signal is transmitted and measured (i.e., polarization of single photons) over the lossy quantum channel. Stage 2, sifting, is where Alice and Bob exchange information over the classical channel to agree upon a common sequence to work with, but that sequence may have errors. Stage 3, reconciliation, is where Alice and Bob exchange information over the classical channel to correct errors between their common bit sequence without exposing the value of their bits. Stage 4 is where Alice and Bob privacy amplify their now identical bit sequences through the application of a hash function that does not require any communication, yielding a shared secret between Alice and Bob. Some differences between QKD and conventional communications [13] are QKD has a high error rate (1 % to 10 % vs less than 10^{-3}), information is transmitted on the lossy quantum channel and error correction is sent over the error free (e.g., TCP/IP) classical channel, quantum information cannot be re-transmitted and QKD requires efficient error correction (code rates greater than 1/2 and close to the Shannon limit) to enable extraction of some amount of secret bits. Because of these differences, some variations on the error correction code configuration are possible that result in additional efficiencies, although the same error correction algorithms are used.

As researchers delve into Gbit/s QKD key production rates [3], attached processors have become a necessity. Initially Field Programmable Gate Arrays (FPGAs) were the preferred attached platform used to provide real-time and high speed processing [11, 12]. FPGAs have been instrumental in achieving Mbit/s QKD key production rates. But FPGAs are unlikely to scale LDPC algorithms to Gbit/s performance for QKD because they lack sufficient amounts of memory and parallel access to that memory needed for highly parallel QKD implementations. A Gbit/s applications-specific integrated circuit (ASIC) LDPC implementation [14], that contains such memory, has been reported for conventional communication applications. But conventional communication applications generally don't require the error correction performance to be as close to the Shannon limit as QKD does and therefore they can use more structured LDPC codes with lower correction efficiencies.

[1] The identification of any commercial product or trade name does not imply endorsement or recommendation by the National Institute of Standards and Technology. Any software code/algorithm is expressly provided "AS IS." NIST makes no warranty of any kind, express, implied, in fact or arising by operation of law, including, without limitation, the implied warranty of merchantability, fitness for a particular purpose, non-infringement and data accuracy.

Graphic processing units (GPUs) [2] are alternative platforms that provide parallelism through many processing elements and multiple GPUs can be supported by one computer. Although GPUs can have large Common memories that can support many parallel LDPC datasets, that Common memory has a high latency access time while their fast local memories are small. Although QKD privacy amplification dictates large datasets, about one Mbit or more, error correction can be conducted on smaller datasets and the accumulated results of multiple datasets combined into a larger one needed for privacy amplification. LDPC performance of 100 Mbit/s has been reported [5] on a GPU. Such performance is attributed to the use of GPU specific instructions and GPU streaming (similar to threads on a CPU). Newer results demonstrate 200 Mbit/s [6], with no mention of GPU specific instructions and GPU streaming. We present the details of our various LDPC GPU implementations and the performance, error correction and execution throughput, that each achieves. A 200 Mbit/s LDPC GPU performance implies 5 GPUs to attain 1 Gbit/s. The 3rd generation of GPUs is advertised as an order of magnitude faster than the 2nd, implying one GPU (or 2) could achieve Gbit/s performance.

2. GPU Environment

A general purpose GPU is a printed circuit board that is attached to a local computer via a high speed interface such as Peripheral Component Interconnect Express (PCIe). It has k single instruction multiple data (SIMD) multiprocessors (MPs), labeled MPs in Fig 1. Each MP has n processing elements (PEs), labeled PEs in Fig 1, for a total of kn PEs. SIMD means that each MP executes a single instruction (such as a multiply) and each PE would execute that instruction, in parallel and in lock-step, on a different data element (such as different elements of a vector). A GPU has multiple GBytes of Common memory, labeled Slow Common Memory in Fig 1, accessible by all MPs with a large latency (100s of clock cycles). Because of this high memory access latency, significant emphasis is placed on optimizing Common memory access and there is hardware support for combining (coalescing) separate accesses that share the large line width for the cache, while a penalty is paid for disjoint access. The GPU clock frequency is ~1 GHz, translating to a capacity performance on the order of 10^{11} Floating Point Operations per Second (FLOPS) and an overall Common memory bandwidth of a few 100 GB/s via a wide memory interface. Each GPU multiprocessor has a dedicated fast local memory of 64 KB that is shared among its PEs with low latency access (a few clock cycles) in addition to a large cache and a large set of registers also with low latency access. All but the Common memory reside on a single chip. These boards can consume over 200 Watts of power. This power consumption and the number of available PCIe slots are the two factors that limit the number of GPUs per computer.

We had access to two different general purpose GPUs. One is in a Windows 7 environment and is of the 2nd GPU architecture generation with 448 PEs and 3 GBytes of Slow Common Memory. A second GPU is in a Linux environment and is of the 3rd GPU architecture generation with 2688 PEs and 6 GB of Slow Common Memory. Both have 14 MPs. These boards are plugged into server level computers driven by a ~3.3 GHz clock, each server has 10s of GB of memory and the processor chips have 4 cores.

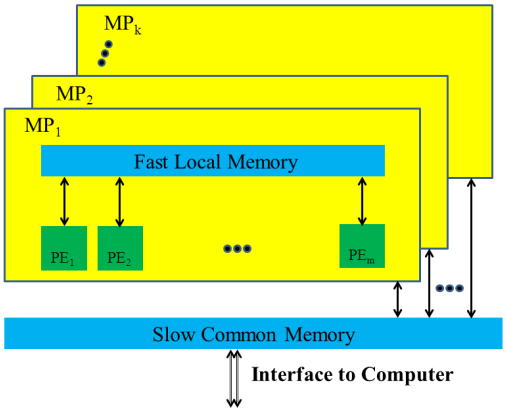


Figure 1. Diagram of a GPU Architecture.

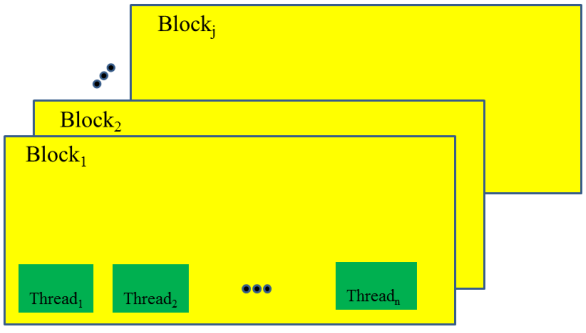


Figure 2. Diagram of a GPU Kernel program.

We are using C++ and the CUDA C programming environment [2], see Fig. 2. There is a 1-to-1 mapping between the GPU hardware architecture, see Fig. 1, and the CUDA programming architecture in Fig 2. A CUDA program is called a Kernel and is comprised of a number of Blocks. The code for a single Block instance, maps to a MP. Each Block consists of a number of Threads. The code for a single Thread instance, maps to a PE. The number of Blocks is independent of the number of MPs and the number of Threads is independent of the number of PEs. If the number is less than the physical devices, than each can operate in parallel on a separate device. If the number is more than the physical devices, than the CUDA scheduler timeshares them on the available devices. The scheduler's operation is opaque and there is no user interface to control its operation. Thus a user can't assign a Block to a MP or force it's execution to remain on the current MP it is assigned to, although once a Block is assigned to an MP it doesn't seem to move about. There are parameters available to instances of executing GPU code that allow the identification of its Block and Thread identity. Thus, it is common for an entire GPU Kernel program to be written as a single set of instructions, realizing that on execution, instances will be distributed to separate PEs and their data accesses are controlled by their Block and Thread identification. Although there are synchronization primitives between Threads within a given Block, there are no synchronization primitives between Threads of different Blocks. There are atomic primitive operations (operations that are guaranteed to complete uninterrupted) that can be used to build synchronization operations between Blocks, but any such synchronization would need to be conducted through the slow Common memory, incurring significant latency. Such atomic primitives consist of a read-modify-write operation. For example, an atomicADD(x,y) instruction would read word x, add y and write x+y back into the location for x and return the original x value to the calling instruction. During the time of reading x and rewriting x+y, no other instructions could access location x. Although between the issuance of the instruction and the actual access at the Common memory, other accesses can occur. Similarly, during the latency of returning x after the rewrite of x+y, other accesses can also occur. Thus the order of these atomic instructions is unpredictable, but the results are sequentially consistent.

3. GPU Port

We have developed two LDPC algorithm approaches based on the LDPC sum-product belief propagation algorithm [13]. One implementation uses floating point multiply and divide that is a good fit for computer software. The other approach uses logarithm lookup tables with integer addition and subtraction, and is a good fit for hardware such as FPGAs and ASICs. Both implementations use the same supporting data structures. The integer implementation using logarithm lookup tables would not be an efficient port since the lookup tables would use most of the fast local GPU memory, leaving little or no fast memory space for the other lookup tables. The floating-point multiply and divide implementation is a good match for GPU porting, but the data structures require significant revision. Because the GPU's slow Common memory is the critical resource that needs to be optimized, our various implementations, discussed below, migrate to various data structures that attempt that optimization.

Our initial data structures are shown in Fig. 3. They are memory efficient for the allocation of the belief registers (LL_Regs) in that LL_Regs are linearly packed in bit order without any padding. Thus, lookup tables are required to locate each group of LL_Regs, either by checksum or bit. Our initial design attempted to use the same data structures on the GPU, but since they wouldn't fit into a single MP's fast local memory we modified the design to divide these lookup tables and distribute parts of them into different MP's fast memory. So our GPU Kernel program operated on a single codeword (dataset), distributed across a number of Blocks. We did that by assigning a part of these lookup tables to a software Block and then each thread in that Block would operate on a different table entry of that part. This required synchronization between the executing Blocks. Since there were no such synchronization primitives, we built our own out of the atomic read-modify-write GPU CUDA instructions. This design worked, but its execution performance was only slightly better than that on a computer, certainly not an order of magnitude faster as was expected.

Our second design (Mini Lookup – Srow/Scol) was quite different. We abandoned the intent to run very large LDPC matrices, whose lookup table wouldn't fit into the local memory. We focused on assigning separate data sets to separate MPs and on minimizing the lookup table information that would be required to reside in the local fast memory, which is limited to 64 KB. Thus our GPU Kernel program would allocate each dataset to a separate Block. So we laid out the LL_Reg structure to be a full 2 dimensional matrix with the i-th row containing all the LL_Regs for the i-th bit. There would be enough columns in the matrix to hold the longest group of bits and thus potentially some, or all, rows may have empty columns at the end of each row. This eliminated the need for the bit lookup table (LL_index) of Fig 3, since one could now compute the location of any bit LL_Reg group in the matrix.

We next revised the LL_Reg content to store checksum (CS) pointers and other information in addition to the belief values, see Fig 4. This increased the LL_Reg to a 32-bit size. Although our algorithm uses floating-point multiply and divide we only store an 8 or 10 bit integer mapping of that floating-point value along with a sign bit and in the remaining 21 bits we include a CS pointer, flags and other values. Our algorithm uses two different forms of the belief register values, an "a" and an "f" form [13]. These forms are used in the two separate passes of the algorithm and there is a simple conversion between these forms. The "a" form is a positive or negative fraction between 0 and 1.0, while the "f" form is positive only but can be any value greater than 0, other than 1.0. Since the "a" form has a more restricted range, we store the "a" form value as an integer mapping and convert it to a floating-point value when we use it.

We now eliminated the need for the checksum lookup table, CS_list in Fig 3, by placing a CS pointer chain into the extra bits of the current LL_Reg data. We store that CS pointer in two separate fields, which are the two indices, row (bit number) and column, into the LL_Reg array. A NULL pointer indicates the end of a checksum chain. The example shown in Fig 4, indicates that checksum group #3 starts with bit #0, 2nd entry (column), followed by bit #2, 4th entry, followed by bit #4, 2nd entry, followed by bit #n-1, 4th entry, finishing with bit #5, 10th entry. This mean that checksum group #3 consists of bits 0, 2, 4, n-1, and 5. Along with the pointer, we also store Bob's original bit value and the current corrected value into the extra space of the LL_Reg. This allows faster data access for checksum computations and for updating Bob's bit values.

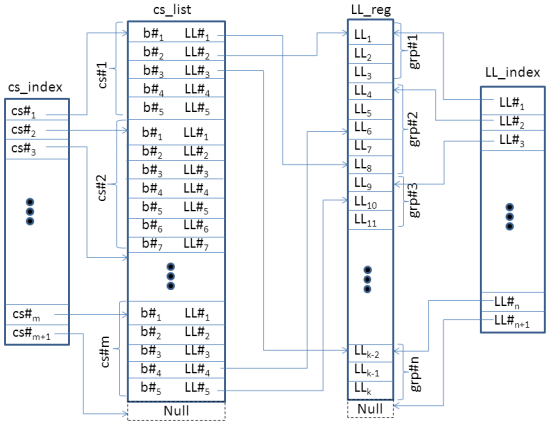


Figure 3. Initial data structure & Lookup Tables.

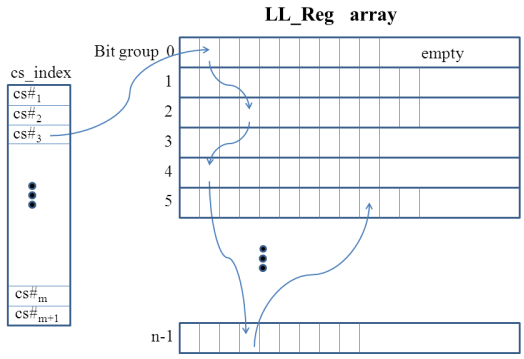


Figure 4. GPU Mini-Lookup data structure.

We could not eliminate the need for the checksum start list, CS_index in Fig. 3, which indicates where the first LL_Reg of a checksum chain is located. This lookup table resides in the fast local memory and there is space in each entry of this table to add the checksum value sent by Alice and another bit indicating whether it matches Bob's original checksum value. These values are needed during the belief propagation algorithm and when checking for successful error correction (convergence). Even with the elimination of all but the CS_index table, we still can't accommodate large matrices. Our limiting factor is the size of this table, which is the number of checksums. That limit is about 18 000 entries, using 3 bytes for each table entry. The number of bits in our matrix is also limited by the size of the pointer field in the LL_Reg. For 10-bit belief value accuracy, we are limited to a 14-bit field that translates into 2^{14} bits. If we shrink our accuracy to 8-bit belief value accuracy, we could increase that to a 16-bit field that translates into 2^{16} bits. Since the largest matrices we have evaluated don't exceed 7 000 checksums or 14 000 bits, our current code uses an 8K checksum limit and a 2^{14} bit limit with a 10-bit belief value accuracy. Our larger matrices would require 26 000 checksums and 54 000 bits. With an 8-bit belief value accuracy, we could accommodate the 54 000 bits, but there isn't sufficient local memory space for 26 000 checksums.

A key decision is whether to allocate the configuration of Fig. 4 by row or column, which results in a significant performance difference. Row allocation (Srow), places a bit group of the LL_Reg array for a single thread in consecutive memory locations. This is not on optimum memory configuration for a GPU since threads execute in parallel. A thread will only access one LL_Reg at a time when processing a bit group, so there is no efficiency in allocating the rows sequentially. A group of threads will simultaneously access separate bit groups (rows) that are not consecutive in

memory. Allocating the array by column (Scol) so the i-th bit of every bit group is now sequential will result in sequential access when a group of threads accesses consecutive bit groups. The first bit of each group to be sequential and each second bit will be sequential, etc. Because of the large latency to the Common memory and the large memory access (cache line) width, the GPU will merge (coalesce) multiple separate memory requests from the parallel executing threads into a single memory request when possible. This results in less memory traffic and increased execution performance. When a group of threads accesses sequential bit groups their combined access will be coalesced, resulting in a significant speed-up for processing bit groups, but we still pay a penalty to process checksums, which are randomly distributed in memory. Thus an allocation by column implementation provides significant speed-up, as much as a factor of two faster.

Based on Falcao [5], we then proceeded to further revise the design (NoLookup table design - NoS) so that we could use larger matrices. To use larger matrices required the elimination of the remaining lookup table in local memory of Fig. 4. To do this, we used two alternating LL_Reg organizations, each with a separate memory allocation incurring twice the memory space. One would be the current organization by bit groups, while the other would be organized by checksum groups as shown in Fig. 5. Thus when operating on bit groups, we would sequentially access the array organized by bit group but write the results back into the array organized by checksum in a random order. When operating on checksum groups we would sequentially access the array organized by checksum group but write the results back into the array organized by bit group in a random order. To accomplish this required each LL_Reg to contain two pointers, one to its location in each array and resulted in doubling the size of an LL_Reg to 64-bits from 32-bits. Overall the memory requirement was four times that of the LL_Reg array in our previous design. Now all the reads were sequential while all the writes were random. Even though we were accessing 4 times the amount of memory, our throughput was about the same as our previous Scol implementation because we don't have to wait for the random writes to complete since we are guaranteed that each LL_Reg is accessed once and only once per pass. Now our matrix size is no longer restricted by the limited local memory size and with no penalty to error correction or execution performance.

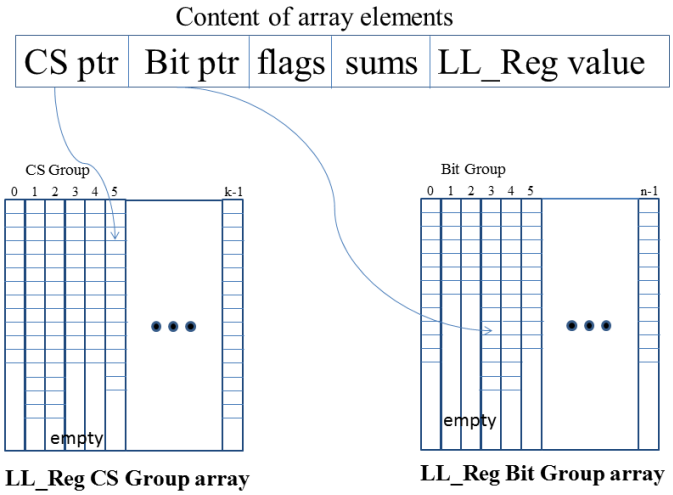


Figure 5. GPU No-Lookup data structure, ping-ponging between dual arrays

We then proceeded to revise our design to use GPU streams also base on [5]. Now a Kernel would run one dataset but we would issue multiple streams of Kernels that should run in parallel. This is similar to the use of Pthreads in the C programming language. The execution performance results were disappointing since they showed little throughput increase over the single Kernel version. Furthermore, although the separate Kernels did appear to overlap their execution but they were not entirely in parallel.

We therefore abandoned our stream design and implemented a multi-dataset (NoM) design based on Falcao [6]. This is similar to our NoLookup table design of Fig. 5, but now we separate the pointers and the data into different arrays as shown in Fig. 6. The upper pair of arrays in Fig. 6 contain a single write pointer and each entry is 32-bits. So the bit ordered array contains the write pointers to the checksum array and the checksum ordered array contains the write pointers to the bit array. Since the pointer arrays are common to all the kernels operating on the code design (LDPC

matrix), there only needs to be one pair in Common memory shared between all the Blocks of a Kernel. The bottom arrays in Fig. 6 contain the dataset information, the LL_reg values and sign, y and y' (Bob's bit value and the corrected value) as well as c and d (Alice's checksum value and a flag if Bob's differs). This can now fit into 16-bit entries. Furthermore, we can group and process multiple datasets together [6] as the 3D array of Fig. 6 shows. So for example, we read a column slice of the CS data array along with a column of write pointers that apply to each of the datasets in the slice. This reduces memory traffic since we read in one set of write pointers for the n datasets of a slice, where we use n=16. This improvement showed a significant execution performance increase, as much as a factor of two faster than the NoLookup table design.

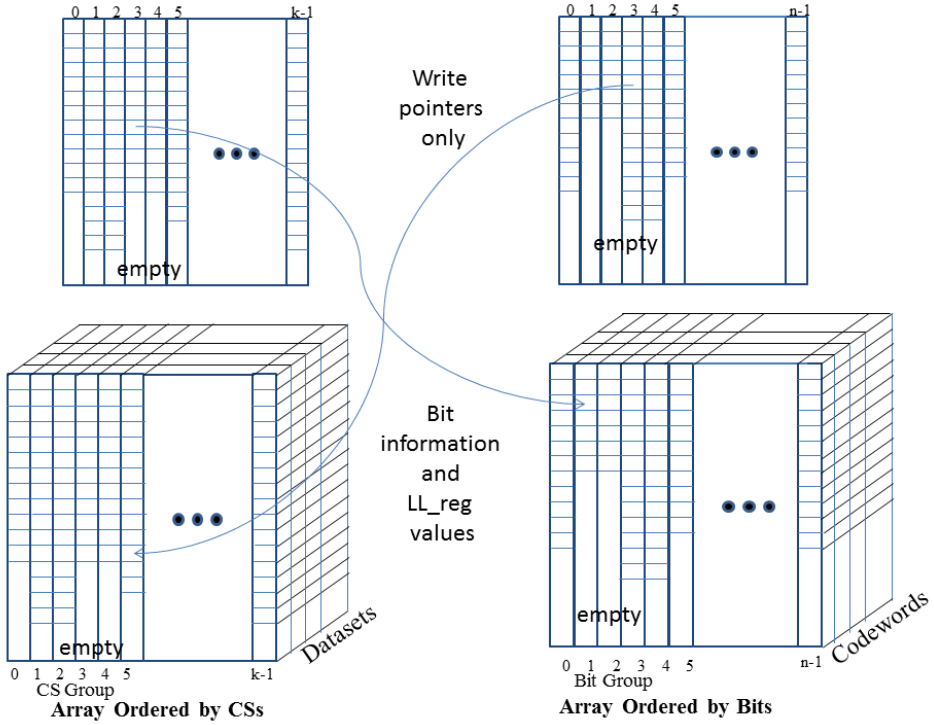


Figure 6. GPU Multi- dataset data structure, ping-ponging between dual 3D arrays

4. Performance Results

We employ a selective set of matrices from standards groups that incorporate LDPC and have published their family of acceptable matrices, such as the IEEE 802.11n Std [8], IEEE 802.16e Std [9] and ETSI DVB Std [4]. Our evaluation of both the error correction performance and computation performance of these matrices for a QKD environment is shown in Tables 2-5. We implemented and executed on two different GPUs, a 2nd generation and a new 3rd generation architecture of the same GPU family. Each GPU is plugged into a Server level computer. The two generation architectures are denoted by a "W" (2nd) and "K" (3rd) in the last letters of their name in the calculation mode column of Tables 2-5 and the number following that represents the number of Threads allocated to each Block. So "NoMW_704" indicates the multi-dataset design run on the 2nd generation GPU architecture using 704 threads per block (the maximum we could fit in that architecture) and "NoMK_1024" indicates the multi-dataset design run on the 3rd generation GPU architecture using 1024 threads per block (the maximum we could fit). While "ScolW_704" and "SrowW_704" refer to the column and row allocation designs of Fig. 4 on the 2nd generation GPU architecture using 704 threads per block. Most of our measurements were for a Kernel with 14 Blocks, one for each MP. But we saw that Falcao [6] got much better throughput by using 100s of Blocks per Kernel. Our results showed up to a 50 % increase in throughput for our multi-dataset implementation by running the maximum number of Blocks that we could, which is denote by entries in Tables 2-5 beginning with a "+". For example, the last line of Table 2 begins with "+488_Blks". That means that for the line above, "NoMW_704", which ran with 14 Blocks the same configuration was run with 488 Blocks, its maximum. The limiting resource (memory) dictated these limits.

There are no entries for the "Srow" or "Scol" designs for the ETSI DVB matrices run on the GPU since the fast local memory for our GPU wasn't large enough to hold the needed mapping tables and placing them in slow Common memory would significantly impact execution rates. The large ETSI matrices would require up to 26 000 checksums and 54 000 bits. With an 8-bit or 10-bit belief value accuracy, we could accommodate the 54 000 bits, but there isn't sufficient local memory space for 26 000 checksums. Each column of Tables 2-5 is the mean QBER of the generated data for which the matrix was being evaluated and the actual errors vary around that mean with a normal distribution. Each entry of Tables 2-5 is the result of at least 1000 datasets and as many as 750 000. Each datasets is randomly chosen with errors randomly distributed throughout the datasets. Each table entry consists of three values, the number of attempts that failed to converge (i.e., failed to succeed in correction) per 1000 (i.e., units of 0.1 %), the average number of algorithm iterations to converge and the data processing rate in Mbit/s. We used 10-bit belief value accuracy since it provided better performance, mostly for the 5/6 rate matrices, but would have no impact on the throughput values. We limited the maximum number of passes on a dataset to 31.

The failure rate is never zero; there is always some probability that the algorithm will fail to converge. Thus a zero for failure to converge just means a low failure rate (below 0.1 %) and indicates a successful operation point for QKD, but not necessarily a good operating point in terms of coding efficiency and secure key ratio. Scanning Tables 2-5 from left to right, we look for the transition from low failure rate to high failure rate, which indicates where a given matrix is no longer effective.

The asymptotic number of secret bits, **K**, that can be extracted from the number of sifted bits, **N**, is based on the following equation:

$$K_{avg} = (1\text{-}f) * \{(N\text{-}V) * [S(X|E) - g*h(p_{est})] - C\}$$

where **V** is the number of sifted bits sacrificed (if any) to estimate the QBER, $\mathbf{p_{est}}$, for error correction, **f** is the failure rate (≤ 1.0) of the error correction scheme, **h** is the Shannon entropy, **g** is an efficiency factor (**g**≥1) such that (N-V)*$\mathbf{g*h(p_{est})}$ equals the amount of information exchanged over the classical channel to accomplish error correction, **S(X|E)** represent Eve's ignorance for Alice's typical qbit value **X** and Eve's corresponding measurement **E** of that qbit and **C** is a fixed number of bits used for hash signatures to verify error correction, to (possibly) report the number of errors corrected so that a better estimate of the QBER can be obtained and maybe a few other small fixed values for safety margins, etc. For N< ~1 Mbit, there is a correction factor that reduces **K** further and for N< ~10 Kbits, **K** falls below zero.

Table 1. QKD Entropy Calculations.

QBER	h(p)	1-2h(p)	g*h(p)	1-h(p)-g*h(p)
1.0%	0.080793	0.838414	0.2	0.719206864
2.0%	0.141441	0.717119	0.2	0.658559457
3.0%	0.194392	0.611216	0.333	0.472608142
4.0%	0.242292	0.515416	0.333	0.424707811
5.0%	0.286397	0.427206	0.5	0.213603043
6.0%	0.327445	0.34511	0.5	0.172555081
7.0%	0.365924	0.268153	0.5	0.134076349
8.0%	0.402179	0.195642	0.5	0.09782081
9.0%	0.43647	0.12706	0.666	-0.102469817
10.0%	0.468996	0.062009	0.666	-0.134995594
11.0%	0.499916	0.000168	0.666	-0.165915958

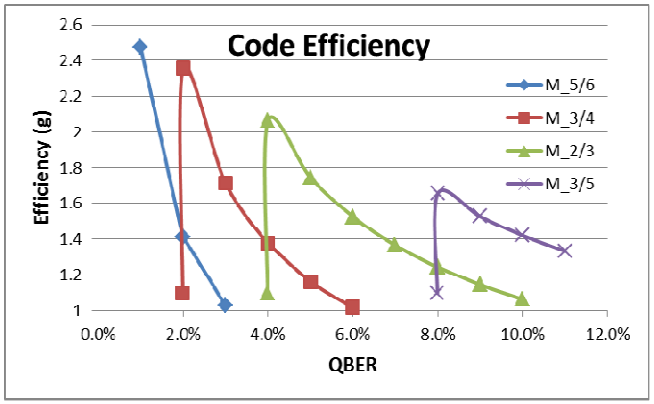


Figure 7. Error Correction Efficiencies for our LDPC Matrices.

For the BB84 (the original QKD) protocol, $\mathbf{S(X|E) = 1 - h(QBER) \geq 1 - h(p_{sft})}$ with great likelihood, where $\mathbf{p_{sft}}$ is an upper likelihood bound of the QBER. As a result, a simplified (ignores f, V and C) inexact version of this equation is commonly shown as:

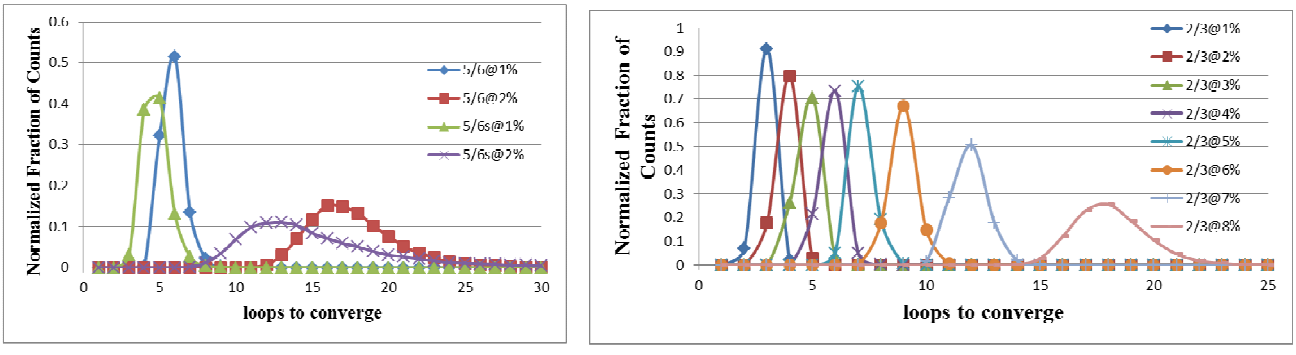
$$K/N = [S(X|E) - g*h(\mathbf{p_{est}})] = 1 - h(\mathbf{p_{sft}}) - g*h(\mathbf{p_{est}}),$$

or some even assume perfect error correction (g=1) and that the QBER is known, present the following:

$$K/N = 1 - 2*h(QBER).$$

From Table 1, we see that for perfect codes (the 3rd column) we can theoretically extract secret bits through 11 %. But for our non-perfect LDPC codes used here, the "g" value is too high to extract secrets above 8 %, as shown by the negative numbers in the last column. Above 8 % we would need to use our 3/5 rate matrices (rate is the information bits divided by the information bits plus the error correction bits and defines the matrix size ratio), with a g*h(p) value of 0.666 per bit, since our 2/3 rate matrices have significant failures above 8 %. Fig. 7 shows the "g" efficiencies for our LDPC matrices, where "M_5/6" is the curve for a 5/6 rate code (or matrix), "M_3/4" is for a 3/4 rate code, etc. The efficiencies decrease (get better) as the QBER increase because the entropy of a matrix is fixed, and as the QBER increases, the entropy increases. Also as the matrix rate decrease, their starting efficiency improves. This is important since decreasing matrix rates implies fewer secret bits can be extracted due to error correction leakage. On the other end of the scale, at 1 % QBER, our efficiency is poor, g ~1.4 or 140 % above the Shannon limit. For QBER ≤ 1 % it may be better to use a more efficient matrix, possibly a 9/10 rate matrix. We did not explore such matrices since this is an unusual operating point for QKD and the entropy is low enough that significant secret extraction already occurs.

From the Tables 2-5 we see that the ETSI DVB matrices provide the best QKD performance. Fig. 8 shows the convergence histogram characteristics of some of the ETSI matrices. These histograms show the distribution of how many passes (loops) the algorithm takes before it converges and corrects the data. The x axis is the number of passes to converge and the y axis is the percent of time the algorithm accomplishes that for the given matrix at the specified QBER. For example, we see that the 5/6 DVB matrix at a QBER 0f 1 % in Fig. 8a converges after 5 passes a little more than 30 % of the time, after 6 passes a little more than 50 % of the time and after 7 passes a little more than 10 % of the time. What isn't obvious from these plots is that some of these matrices have long tails and occasionally take many more passes to converge, although these occurrences are much less than 1 % of the time. Fig. 8a is the histogram for the 5/6 DVB (5/6) and sDVB (5/6s) matrices at a QBER of 1 % and 2 %. We see that the 5/6 DVB curves are sharper with higher peaks, while the 5/6 sDVB curves are flatter with lower peaks. The sharper the curve for a given QBER the more uniformly the algorithm will converge at that QBER. From Fig. 8b for the DVB 2/3 matrix, we see that as the QBER increases the histogram curve shifts right (more passes to converge) and flattens (more variability in the number of passes needed to converge). This presents a possible strategy to use when one is operating at a QBER that demonstrates a high peak and thus low variability to converge within a fixed number of passes. We could eliminate the test for convergence and just do a fixed number of passes. For example, the 5/6@1 % curve at the left of Fig .8a converges within 10 passes 99.9+ % of the time. Similarly the 2/3@7 % curve near the right of Fig. 8b converges within 15 passes 99.9+ % of the time and likewise for all the curves to its left. By fixing the passes rather than implementing the convergence test we can gain some throughput increase by not having to take the time at the end of each passes to do that test, which includes a pair of synchronization primitives between the Threads of a Block. This may boost our throughput a few percent, but it won't be a significant gain. Furthermore, when these curves start to flatten indicating the QBER is approaching the limit of this code, as shown on the far right side of Figs. 8a and 8b, the convergence becomes more variable with a long tail. In this area, foregoing the convergence test may cost more than it saves.



(a) (b)

Figure 8. Convergence Histogram of (a) ETSI DVB and sDVB 5/6 Rate Matrices and (b) ETSI DVB 2/3 Rate Matrix.

Our results show almost two orders of magnitude increase in execution speed for our multi-dataset GPU implementation compared to a server implementation, while the error correction performance was mostly the same. This is a skewed comparison since we were only running a single code thread on the CPU. If we ran a multi-thread code on a dual CPU, each with 4 cores, then it would more likely be about an order of magnitude improvement. Nevertheless, we were able to achieve up to 196 Mbit/s error correction rates. This compares well with Falcao's [5] 109 Mbit/s originally reported performance and close to his newer achievements of 200+ Mbit/s [6].

Unfortunately neither of these implementations are practical for QKD application. Falcao uses a 1/2 rate matrix, which eliminates any possible secret extraction since g*h(p) = 1.0. Furthermore, he uses a regular LDPC matrix, which are known not to provide as good error correction as irregular matrices, and he also uses a minimum-sum algorithm that doesn't provide as good results as sum-product algorithms. For Falcao's application, conventional communications, these are irrelevant concerns. But QKD operates at relatively high error rates and, therefore, there are concerns about the effectiveness of the matrices and the LDPC algorithm. As we've seen, 8-bit vs. 10-bit precision has an impact on the failure rate of some of our LDPC matrices. Regular matrices allow for efficient implementations since the location of bits and checksums can be easily calculated and the lengths are fixed. This allows for fixed implementations that don't require variable length loops and conditionals, fixed length loops can be "unrolled". This results in in-line GPU code, identical for each thread – perfect for a SIMD architecture like a GPU. In addition, our implementation includes algorithm convergence tests and thus incorporates both Alice's and Bob's information and uses synchronization primitive and atomic operations to accomplish this. These primitives, although efficient between threads of a block, do slow down execution. Another feature that Falcao's implementation doesn't require.

Table 2. QKD LDPC Evaluation for 5/6 Rate Matrices.

5/6 Rate Matrices			Failures/Avg iterations/Mb/s - per 1000 samples (Max 31 iterations for Convergence)								
alculation Mo	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%	11%
ETSI DVB, Rate 5/6 (54000 bits, 10800 Chksums)											
CPU_S	1/7.5/0.96	28/21.1/0.31	1000/31.0/0.20								
NoSW_704	0/5.9/17.38	4/17.8/5.92	1000/31.0/3.82								
NoSK_1024	0/5.8/33.47	5/17.7/11.53	1000/31.0/8.22								
NoMW_704	0/5.9/44.02	4/17.8/15.27	1000/31.0/11.21								
+70_Blk	0/5.9/46.77	4/17.9/16.00	1000/31.0/11.30								
NoMK_1024	0/5.9/70.43	4/17.8/24.91	1000/31.0/17.2								
+140_Blk	0/5.9/77.06	4/17.8/26.12	1000/31.0/17.24								
ETSI sDVB, Rate 5/6 (13320 bits, 2880 Chksums)											
CPU_S	2/6.0/1.67	67/17.8/0.47	1000/31.0/0.27								
SrowW_512	0/3.5/22.10	57/12.9/6.35	1000/31.0/4.73								
SeqW_704	0/4.8/45.51	57/16.0/12.40	1000/31.0/10.39								
NoSW_704	0/4.8/43.72	57/16.0/12.21	1000/31.0/9.49								
ScolK_1024	0/4.7/53.23	47/15.6/15.78	1000/31.0/13.80								
NoSK_1024	0/4.7/60.43	45/15.6/15.93	1000/31.0/13.77								
NoMW_704	0/4.8/58.67	56/15.9/19.83	1000/31.0/13.60								
+224_Blk	0/4.8/73.64	56/15.9/21.88	1000/31.0/13.81								
NoMK_1024	0/4.8/79.96	54/15.9/31.27	1000/31.0/20.99								
+448_Blk	0/4.8/117.81	56/15.9/34.76	999/31.0/21.18								
IEEE 11n, Rate 5/6 (1620 bits, 324 Chksums)											
CPU_S	1/4.6/1.92	419/19.4/0.38	988/30.9/0.25								
SrowW_512	12/5.1/22.04	441/20.1/6.89	989/30.9/4.52								
SeqW_704	1/3.5/29.83	406/18.6/7.57	984/30.8/6.71								
NoSW_704	1/3.5/32.19	406/18.6/8.11	984/30.8/7.32								
ScolK_1024	2/3.5/36.09	404/18.4/9.23	982/30.7/8.68								
NoSK_1024	2/3.5/37.02	404/18.4/9.62	982/30.7/8.91								
NoMW_704	1/3.5/43.75	407/18.6/12.27	984/30.8/9.90								
+448_Blk	1/3.5/65.27	413/18.7/14.24	985/30.8/9.99								
NoMK_1024	1/3.5/61.14	404/18.6/16.80	985/30.8/13.35								
+448_Blk	1/3.5/92.78	409/18.7/19.74	985/30.8/13.45								
IEEE 16e, Rate 5/6 (1920 bits, 384 Chksums)											
CPU_S	1/4.6/1.83	417/20.0/0.37	989/30.9/0.24								
SrowW_512	4/5.0/25.05	443/20.7/7.27	990/30.9/4.93								
SeqW_704	0/3.6/35.72	393/18.7/8.41	991/30.9/7.42								
NoSW_704	0/3.6/39.06	393/18.7/9.15	991/30.9/8.21								
ScolK_1024	0/3.6/41.81	412/19.0/10.00	987/30.8/9.35								
NoSK_1024	0/3.6/42.58	412/19.0/10.42	987/30.8/9.65								
+448_Blk	0/3.6/73.90	397/18.8/15.72	991/30.9/11.08								
NoMW_704	0/3.6/55.48	398/18.8/13.72	991/30.9/11.00								
+448_Blk	0/3.6/102.07	397/18.7/21.47	991/30.9/14.59								

Although our peak performance of 196 Mbit/s is about the same as Falcao, that performance of ours is on a 3rd generation GPU. Our peak 2nd generation GPU performance, the same platform as Falcao, is 134 Mbit/s. Furthermore, that peak performance does not represent a reasonable QKD operating point. That peak performance is for a 2/3 rate matrix at a 1 % QBER. When at 1 % QBER, we should be operating with a 5/6 or even a 9/10 LDPC matrix. From Tables 2-5, our reasonable QKD performance is more in the range of 27 Mbit/s to 117 Mbit/s for our 3rd generation GPU and 17 Mbit/s to 73 Mbit/s for our 2nd generation GPU. We also note that the performance increase we see due to moving from the 2nd generation GPU to the new 3rd generation GPU is up to about 50 %, not the order of magnitude that we expected.

Table 3. QKD LDPC Evaluation for 3/4 Rate Matrices.

3/4 Rate Matrices			Failures/Avg iterations/Mb/s - per 1000 samples (Max 31 iterations for Convergence)								
alculation Mo	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%	11%
ETSI DVB, Rate 3/4 (48600 bits, 16200 Chksums)											
CPU_S	0/4.7/1.59	0/6.6/1.06	0/9.3/0.71	0/14.9/0.42	996/31.0/0.19						
NoSW_704	0/3.6/26.20	0/5.4/18.07	0/8.1/12.60	0/13.6/7.70	991/31.0/3.66						
NoSK_1024	0/3.6/52.98	0/5.5/35.50	0/8.1/26.28	0/13.5/16.00	987/31.0/7.80						
NoMW_704	0/3.6/78.05	0/5.4/54.08	0/8.1/37.33	0/13.6/22.58	990/31.0/11.48						
+70_Blk	0/3.6/82.90	0/5.4/56.55	0/8.1/39.02	0/13.6/23.52	989/31.0/11.63						
NoMK_1024	0/3.6/126.72	0/5.4/87.58	0/8.1/60.63	0/13.6/36.55	988/31.0/17.85						
+140_Blk	0/3.6/135.05	0/5.4/91.38	0/8.1/62.67	0/13.6/37.38	988/31.0/17.89						
ETSI sDVB, Rate 3/4 (11880 bits, 4320 Chksums)											
CPU_S	0/4.0/2.64	0/5.6/1.70	0/7.9/1.11	0/12.2/0.66	556/27.9/0.27						
SrowW_512	0/2.2/33.09	0/3.2/23.18	0/5.0/16.17	0/8.5/9.79	544/24.8/4.67						
SeqW_704	0/3.0/79.13	0/4.5/55.12	0/6.8/38.13	0/11.0/22.55	544/27.2/10.88						
NoSW_704	0/3.0/72.42	0/4.5/50.28	0/6.8/34.94	0/11.0/21.00	544/27.2/9.98						
ScolK_1024	0/3.0/104.72	0/4.5/71.33	0/6.8/44.55	0/11.0/27.94	550/27.1/14.10						
NoSK_1024	0/3.0/102.41	0/4.5/69.88	0/6.8/43.86	0/11.0/26.92	550/27.1/13.97						
NoMW_704	0/3.0/118.15	0/4.5/80.67	0/6.8/55.46	0/11.0/33.14	543/27.2/15.0						
+224_Blk	0/3.0/126.23	0/4.5/86.15	0/6.8/58.77	0/11.0/35.59	542/27.2/15.66						
NoMK_1024	0/3.0/159.05	0/4.5/105.85	0/6.7/72.72	0/11.0/43.29	546/27.2/22.36						
	0/3.0/191.06	0/4.5/129.22	0/6.7/87.64	0/11.0/52.83	543/27.2/22.51						
IEEE 11n, Rate 3/4 (1458 bits, 486 Chksums)											
CPU_S	0/3.2/2.53	0/4.6/1.61	3/7.1/0.97	205/15.9/0.40	815/28.4/0.22						
SrowW_512	0/3.3/31.71	1/4.7/24.36	3/7.2/17.02	202/16.0/8.67	815/28.4/4.96						
SeqW_704	0/2.2/58.45	0/3.6/38.19	5/6.1/18.23	212/15.2/7.72	818/28.2/6.71						
NoSW_704	0/2.2/61.27	0/3.6/40.12	5/6.1/19.14	212/15.2/8.04	818/28.2/7.09						
ScolK_1024	0/2.2/70.00	3/3.7/38.23	20/6.4/17.36	225/15.4/9.05	832/28.5/8.48						
NoSK_1024	0/2.2/69.49	0/3.6/46.93	5/6.1/21.97	197/14.9/9.24	823/28.3/8.52						
NoMW_704	0/2.2/97.87	0/3.6/61.25	4/6.1/26.70	212/15.2/13.6	821/28.3/9.88						
+448_Blk	0/2.2/112.45	0/3.6/70.07	4/6.2/37.97	218/15.3/15.82	824/28.3/10.33						
NoMK_1024	0/2.2/138.06	0/3.6/86.53	4/6.1/38.99	215/15.2/19.17	818/28.2/13.33						
+448_Blk	0/2.2/138.14	0/3.6/97.76	4/6.1/54.56	216/15.3/22.45	822/28.3/14.01						
IEEE 16e, Rate 3/4 A (1728 bits, 576 Chksums)											
CPU_S	0/3.2/2.61	0/4.6/1.60	9/7.8/0.86	365/19.4/0.33	941/30.2/0.21						
SrowW_512	0/3.2/28.85	0/4.6/21.10	8/7.8/14.03	364/19.5/6.05	941/30.2/3.99						
SeqW_704	0/2.2/68.28	0/3.6/43.18	9/6.7/17.34	368/19.1/8.36	939/30.2/7.40						
NoSW_704	0/2.2/71.16	0/3.6/45.26	9/6.7/18.22	368/19.1/8.72	939/30.2/7.89						
ScolK_1024											
NoSK_1024	0/2.2/77.76	0/3.6/50.06	11/6.7/19.63	365/18.9/9.97	937/30.1/9.37						
NoMW_704	0/2.2/110.12	0/3.6/67.17	9/6.7/25.93	373/19.2/13.22	940/30.2/10.78						
+448_Blk	0/2.2/126.05	0/3.6/77.83	9/6.7/37.13	370/19.1/15.06	939/30.2/11.01						
NoMK_1024	0/2.2/156.10	0/3.6/94.58	9/6.7/37.97	367/19.1/18.18	937/30.1/14.59						
+448_Blk	0/2.2/164.75	0/3.6/108.23	9/6.7/53.29	368/19.1/21.00	939/30.2/14.88						
IEEE 16e, Rate 3/4 B (1728 bits, 576 Chksums)											
CPU_S	0/3.3/2.37	0/4.7/1.52	2/7.1/0.93	208/16.2/0.38	851/29.0/0.21						
SrowW_512	0/3.4/26.39	0/4.7/20.42	2/7.2/14.01	212/16.4/6.78	851/29.1/3.90						
SeqW_704	0/2.2/63.64	0/3.6/41.65	3/6.2/20.53	219/15.6/8.25	858/28.9/7.14						
NoSW_704	0/2.2/66.99	0/3.6/43.99	3/6.2/21.74	219/15.6/8.68	858/28.9/7.68						
ScolK_1024											
NoSK_1024	0/2.2/71.57	0/3.6/48.40	2/6.1/25.09	200/15.2/9.59	847/28.8/8.97						
+448_Blk	0/2.2/119.22	0/3.6/75.48	2/6.1/42.39	214/15.6/16.95	860/29.0/11.02						
NoMK_1024	0/2.2/143.50	0/3.6/91.57	2/6.1/42.66	212/15.5/19.72	858/29.0/13.91						
+448_Blk	0/2.2/161.00	0/3.6/101.68	2/6.1/57.77	216/15.6/23.14	860/29.0/14.50						

Table 4. QKD LDPC Evaluation for 2/3 Rate Matrices.

2/3 Rate Matrices			Failures/Avg iterations/Mb/s - per 1000 samples (Max 31 iterations for Convergence)								
alculation Mo	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%	11%
ETSI DVB, Rate 2/3 (43200 bits, 21600 Chksums)											
CPU_S	0/4.0/1.90	0/4.9/1.46	0/5.8/1.18	0/6.9/0.95	0/8.2/0.77	0/10.1/0.61	0/13.0/0.46	0/19.3/0.30	994/30.9/0.18		
NoSW_704	0/3.0/31.63	0/3.8/25.21	0/4.8/20.81	0/5.8/17.34	0/7.1/14.32	0/9.0/11.63	0/11.9/8.92	0/18.1/5.89	990/31.0/3.71		
NoSK_1024	0/3.0/68.19	0/3.9/53.23	0/4.8/43.68	0/5.8/36.39	0/7.1/29.74	0/9.0/24.12	0/11.9/18.44	0/18.0/11.92	990/31.0/7.76		
NoMW_704	0/3.0/99.06	0/3.9/78.13	0/4.8/64.33	0/5.8/53.64	0/7.2/43.74	0/9.0/35.69	0/11.9/27.34	0/18.1/17.49	990/31.0/11.89		
+56_Blk	0/3.0/103.10	0/3.9/80.86	0/4.8/66.63	0/5.8/55.14	0/7.2/45.34	0/9.0/36.74	0/11.9/28.15	0/18.1/18.10	990/31.0/12.01		
NoMK_1024	0/3.0/150.28	0/3.9/121.46	0/4.8/100.01	0/5.8/83.33	0/7.2/68.81	0/9.0/55.59	0/11.9/42.28	0/18.1/27.26	989/31.0/17.81		
+140_Blk	0/3.0/162.66	0/3.9/126.38	0/4.8/103.17	0/5.8/85.33	0/7.2/70.30	0/9.0/56.47	0/11.9/42.89	0/18.1/27.70	990/31.0/17.69		
ETSI sDVB, Rate 2/3 (10800 bits, 5400 Chksums)											
CPU_S	0/3.5/2.70	0/4.4/1.94	0/5.4/1.49	0/6.4/1.17	0/7.8/0.89	0/9.6/0.70	0/12.6/0.51	7/19.2/0.32	873/30.5/0.20		
SrowW_512	0/1.7/33.03	0/2.3/25.38	0/3.2/20.70	0/3.7/17.22	0/4.7/14.08	0/6.0/11.26	0/8.2/8.38	10/13.4/5.02	863/29.1/3.68		
SeqW_704	0/2.5/73.67	0/3.4/57.07	0/4.3/46.62	0/5.4/38.64	0/6.7/31.29	0/8.5/24.89	0/11.5/18.24	10/18.0/10.53	863/30.3/8.20		
NoSW_704	0/2.5/60.34	0/3.4/45.93	0/4.3/37.33	0/5.4/30.90	0/6.7/25.26	0/8.5/20.10	0/11.5/14.91	10/18.0/8.94	863/30.3/6.33		
ScolK_1024											
NoSK_1024	0/2.5/92.71	0/3.4/71.59	0/4.3/57.83	0/5.4/48.99	0/6.7/40.29	0/8.5/31.77	0/11.5/23.16	10/18.0/13.12	847/30.2/10.67		
NoMW_704	0/2.5/111.87	0/3.4/85.23	0/4.3/69.00	0/5.4/56.82	0/6.7/46.50	0/8.5/36.85	0/11.5/27.06	10/18.0/16.06	863/30.3/11.74		
+224_Blk	0/2.5/121.30	0/3.4/92.18	0/4.3/73.92	0/5.4/60.33	0/6.7/48.77	0/8.5/38.55	0/11.5/28.51	11/18.0/17.35	859/30.3/11.93		
NoMK_1024	0/2.5/145.47	0/3.4/108.80	0/4.3/87.69	0/5.4/71.85	0/6.7/58.29	0/8.5/46.16	0/11.5/34.04	13/18.0/23.71	855/30.3/16.76		
+448_Blk	0/2.5/178.09	0/3.4/134.91	0/4.3/107.42	0/5.4/87.30	0/6.7/70.60	0/8.5/55.89	0/11.5/41.38	11/18.0/25.59	857/30.3/16.93		
IEEE 11n, Rate 2/3 (1296 bits, 648 Chksums)											
CPU_S	0/2.9/2.68	0/3.5/2.03	0/4.3/1.57	0/5.1/1.24	1/6.5/0.93	10/8.9/0.65	142/15.0/0.37	621/25.3/0.22	944/30.4/0.18		
SrowW_512	0/2.9/28.64	0/3.6/23.83	0/4.3/21.24	0/5.2/17.85	0/6.5/15.04	8/8.9/11.26	140/15.1/7.07	615/25.3/4.28	942/30.3/3.58		
ScolW_704	0/1.8/68.57	0/2.5/54.32	0/3.3/43.01	0/4.2/34.12	0/5.5/24.59	13/8.1/12.97	168/14.7/7.35	607/24.8/6.57	935/30.2/6.10		
NoSW_704	0/1.8/70.70	0/2.5/55.72	0/3.3/44.22	0/4.2/35.25	0/5.5/25.45	13/8.1/13.41	168/14.7/7.56	607/24.8/6.82	935/30.2/6.47		
ScolK_1024											
NoSK_1024	0/1.8/76.24	0/2.5/60.57	0/3.3/49.83	0/4.2/39.70	0/5.5/28.32	15/8.1/13.61	175/14.8/8.01	620/25.0/7.55	935/30.1/7.29		
NoMW_704	0/1.8/117.07	0/2.5/88.50	0/3.3/70.40	0/4.2/55.05	0/5.5/38.58	12/8.1/21.18	166/14.6/13.90	605/24.8/10.34	935/30.2/9.60		
+448_Blk	0/1.8/132.06	0/2.5/99.65	0/3.3/78.23	0/4.2/61.58	0/5.5/46.09	12/8.1/28.33	166/14.6/16.14	605/24.8/11.20	935/30.2/9.80		
NoMK_1024	0/1.8/150.70	0/2.5/113.27	0/3.3/90.37	0/4.2/70.57	0/5.5/50.50	12/8.0/28.29	162/14.5/17.63	600/24.7/12.75	933/30.2/11.76		
+448_Blk	0/1.8/163.09	0/2.5/123.99	0/3.3/98.55	0/4.2/77.57	0/5.5/58.43	12/8.0/37.20	164/14.6/20.84	605/24.8/13.92	933/30.2/12.01		
IEEE 16e, Rate 2/3 A (1536 bits, 768 Chksums)											
CPU_S	0/2.8/2.93	0/3.6/2.17	0/4.3/1.71	0/5.2/1.34	0/6.5/1.02	3/8.8/0.72	102/14.6/0.42	543/24.4/0.25	917/30.2/0.20		
SrowW_512	0/2.9/32.7	0/3.6/27.68	0/4.4/23.25	0/5.3/20.51	0/6.6/16.70	3/8.9/12.95	101/14.6/8.37	538/24.5/5.09	915/30.2/4.13		
ScolW_704	0/1.9/73.15	0/2.6/58.08	0/3.3/46.14	0/4.3/36.58	0/5.6/27.24	4/7.9/16.04	109/13.8/8.37	539/24.1/7.55	922/30.1/7.18		
NoSW_704	0/1.9/73.14	0/2.6/57.96	0/3.3/46.24	0/4.3/36.85	0/5.6/27.57	4/7.9/16.33	109/13.8/8.54	539/24.1/7.70	922/30.1/7.42		
SeqK_1024											
NoSK_1024	0/1.8/96.36	0/2.5/78.41	0/3.3/61.47	0/4.3/49.91	0/5.6/35.98	5/8.0/20.28	116/14.0/10.53	552/24.2/9.64	917/30.0/9.20		
NoMW_704											
NoMK_1024											
IEEE 16e, Rate 2/3 B (1536 bits, 768 Chksums)											
CPU_S	0/2.8/3.08	0/3.4/2.24	0/4.3/1.70	0/5.5/1.25	12/8.0/0.80	159/14.3/0.43	566/24.0/0.25	921/30.0/0.20	997/31.0/0.20		
SrowW_512	0/2.8/32.31	0/3.4/29.02	0/4.3/23.64	0/5.5/19.66	8/8.0/14.86	154/14.2/8.79	558/24.0/6.46	993/30.9/10.82	997/31.0/4.12		
ScolW_704	0/1.8/78.74	0/2.4/58.22	0/3.3/42.96	0/4.5/29.64	14/7.0/14.52	151/13.3/8.30	569/23.7/7.62	927/30.0/7.26	996/31.0/7.11		
NoSW_704	0/1.8/77.57	0/2.4/56.77	0/3.3/42.37	0/4.5/29.39	14/7.0/14.51	151/13.3/8.28	569/23.7/7.62	927/30.0/7.35	996/31.0/7.23		
NoSK_1024	0/1.7/101.19	0/2.4/75.55	0/3.3/59.25	0/4.5/40.15	15/7.0/19.20	155/13.3/10.7	586/23.7/9.97	910/29.7/9.42	995/31.0/9.02		
NoMW_704	0/1.8/117.59	0/2.4/87.52	0/3.3/63.21	0/4.5/40.15	13/7.0/21.95	152/13.3/14.2	569/23.7/10.25	926/30.0/9.39	996/31.0/9.37		
+448_Blk	0/1.8/134.76	0/2.4/99.14	0/3.3/73.30	0/4.5/52.30	13/7.0/30.77	151/13.3/16.9	569/23.7/11.33	923/29.9/9.61	996/31.0/9.42		
NoMK_1024	0/1.8/186.05	0/2.4/126.21	0/3.3/92.36	0/4.5/58.23	15/7.0/31.88	150/13.2/20.4	566/23.5/16.49	918/29.9/15.03	996/31.0/14.97		
+448_Blk	0/1.8/196.67	0/2.4/155.68	0/3.3/122.46	0/4.5/87.58	13/7.0/51.61	151/13.2/28.2	569/23.6/18.38	922/29.9/15.40	996/31.0/15.07		

Table 5. QKD LDPC Evaluation for 3/5 Rate Matrices.

3/5 Rate Matrices			Failures/Avg iterations/Mb/s - per 1000 samples (Max 31 iterations for Convergence)								
alculation Mo	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%	11%
ETSI DVB, Rate 3/5 (38880 bits, 25920 Chksums)											
CPU_S	0/3.4/1.61	0/4.0/1.28	0/4.6/1.07	0/5.1/0.93	0/5.9/0.78	0/6.5/0.68	0/7.5/0.57	0/9.0/0.46	0/11.6/0.35	61/21.2/0.18	1000/31.0/0.13
NoSW_704	0/2.3/23.57	0/3.0/19.69	0/3.6/17.21	0/4.1/15.22	0/4.8/13.42	0/5.5/11.81	0/6.5/10.18	0/8.0/8.45	0/10.6/6.41	68/20.1/3.27	1000/31.0/2.38
NoSK_1024	0/2.4/48.35	0/3.0/38.92	0/3.6/34.25	0/4.1/29.65	0/4.8/26.81	0/5.5/23.74	0/6.5/21.43	0/8.0/17.60	0/10.6/13.32	59/20.0/6.30	1000/31.0/5.00
NoMW_704	0/2.3/75.29	0/3.0/63.34	0/3.6/55.63	0/4.1/48.04	0/4.8/42.92	0/5.5/38.13	0/6.5/32.70	0/8.0/26.80	0/10.6/20.03	68/20.1/9.08	999/31.0/7.83
NoMK_1024	0/2.3/121.96	0/3.0/100.43	0/3.6/88.39	0/4.1/78.52	0/4.8/67.94	0/5.5/60.50	0/6.5/51.73	0/8.0/42.49	0/10.6/31.78	66/20.1/15.31	999/31.0/11.86
ETSI sDVB, Rate 3/5 (9720 bits, 6480 Chksums)											
CPU_S	0/3.5/2.70	0/4.4/1.93	0/5.4/1.49	0/6.4/1.17	0/7.8/0.91	0/9.6/0.70	0/12.6/0.51	7/19.2/0.32	873/30.5/0.20	1000/31.0/0.20	1000/31.0/0.20
SrowW_512	0/1.7/33.04	0/2.2/25.40	0/2.8/20.70	0/3.6/17.21	0/4.8/14.07	0/6.1/11.26	0/8.2/8.39	10/12.2/5.02	863/29.2/3.68	1000/31.0/3.62	1000/31.0/3.62
ScolW_704	0/2.1/50.37	0/2.9/43.78	0/3.2/36.85	0/3.9/33.39	0/4.4/29.13	0/5.2/25.16	0/6.2/21.51	0/7.6/17.33	0/10.5/11.60	253/21.2/5.56	983/30.9/5.16
NoSW_704	0/2.1/38.49	0/2.9/31.99	0/3.2/27.38	0/3.9/24.21	0/4.4/21.13	0/5.2/18.25	0/6.2/15.52	0/7.6/12.61	0/10.5/8.75	253/21.2/4.35	983/30.9/3.52
SeqK_1024	0/2.1/71.78	0/2.9/63.34	0/3.2/51.81	0/3.9/47.93	0/4.4/40.84	0/5.2/35.35	0/6.2/29.24	1/7.6/23.32	4/10.5/15.40	257/21.1/7.52	984/30.9/7.20
NoSK_1024	0/2.1/66.51	0/2.9/58.95	0/3.2/48.50	0/3.9/45.10	0/4.4/38.51	0/5.2/33.51	0/6.2/28.62	0/7.6/23.21	0/10.4/15.65	250/21.0/7.23	984/30.9/6.96
NoMW_704	0/2.1/82.53	0/2.9/66.81	0/3.2/58.50	0/3.9/51.00	0/4.4/44.97	0/5.2/38.76	0/6.2/32.98	0/7.6/26.47	0/10.5/16.96	252/21.2/8.92	982/30.9/7.63
NoMK_1024	0/2.1/116.61	0/2.9/88.63	0/3.2/79.47	0/3.9/67.40	0/4.4/59.89	0/5.2/51.71	0/6.2/43.57	0/7.6/34.89	0/10.4/25.49	248/21.1/13.85	982/30.9/11.42

5. Conclusion

LDPC error correction is a one-way algorithm that has become popular for QKD post-processing. GPUs provide an interesting attached platform that we hoped could deliver Gbit/s error correction performance for QKD. We present the details of our various LDPC GPU implementations along with both error correction and execution throughput performance that each achieves. Although our revisions have increased our top GPU LDPC throughput from about 30 Mbit/s to 196 Mbit/s, the reasonable QKD operating range is 17 Mbit/s to 73 Mbit/s on our 2nd generation GPU and is 27 Mbit/s to 117 Mbit/s on our 3rd generation GPU. The major performance increases were due to memory optimizations (data read coalescing and multiple Blocks and datasets), while the new 3rd generation GPU provided up to about an additional 50 % performance increase even though there are 6 times more processing elements (192 vs 32) per GPU multiprocessor. GPU streaming did not provide any throughput increase for our implementation, possibly due to the overhead incurred from the increase in the CPU pthreads required to support GPU streaming and partly due to the overhead incurred by spawning multiple kernels. Also the fact that for streaming, the Kernels' execute times are overlapped and not completely in parallel. Specific GPU instructions have less impact on performance for newer architectures, especially when data items are accessed only once and not reused from caches dedicated to those instructions, as is the case for LDPC. Thus, contrary to previous thoughts, the potential for GPUs to provide Gbit/s error correction, although possible, seems less feasible – requiring up to 37 3rd generation GPUs or up to 58 2nd generation GPUs vs. the more palatable 10 GPUs previously speculated.

Acknowledgement

We gratefully acknowledge Microway, Inc. for providing access to a Nvidia Kepler-accelerated compute cluster. We would also like to acknowledge Joao Andrade and Gabriel Falcao of the University of Coimbra, Portugal for their discussions on this topic.

References

[1] C. H. Bennett and G. Brassard, "Quantum Cryptography: Public key distribution and coin tossing", Proc of the IEEE Intern'l Conf. on Computers, Systems, and Signal Processing, Bangalore, India (1984).

[2] Nvidia Corp., "CUDA C Programming Guide Version 3.2", <https://developer.nvidia.com/cuda-toolkit-32-downloads> (22 Oct. 2010).

[3] DARPA-BAA-12-42, "Quiness: Macroscopic Quantum Communications" (15 May 2012). <https://www.fbo.gov/index?s=opportunity&mode=form&id=6a3a61d577305f71d9be268925c4b201&tab=core&_cview=0>

[4] ETSI EN 302 307 V1.2.1 (2009-08), "Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2)", Sect 5.3 and Annex B (2009).

[5] G. Falcao, V. Silva, and L. Sousa, "How GPUs can outperform ASICs for fast LDPC decoding", Proc. of the 23rd intern'l conf. on Supercomputing, ACM, pp 390–399 (2009).

[6] G. Falcao, V. Silva, L. Sousa and J. Andrade, "Portable LDPC Decoding on Multicores Using OpenCL", IEEE Signal Processing Magazine, pp 81–87 (July 2012).

[7] R. Gallager, "Information Theory and Reliable Communication", Wiley, New York (1968).

[8] IEEE Std 802.11n 2009, "Wireless LAN Medium Access Control & Physical Layer Specification", Part 11, Amendment 5, Annex R (Oct. 2009).

[9] IEEE Std 802.16e 2005, "Air Interface for Fixed and Mobile Broadband Wireless Access Systems", Part 16, Amendment 2, Annex H (Feb. 2006).

[10] D. MacKay, "Good Error-Correcting Codes Based on Very Sparse Matrices", IEEE Trans Information Theory, Vol. 45, N0. 2, pp 399-431 (Mar 1999).

[11] A. Mink, "Custom Hardware to Eliminate Bottlenecks in QKD Throughput Performance", Proc. SPIE Optic East: Next-Generation Photonic Sensor Technologies Symposium, Boston, MA, (9-12 Sept. 2007).

[12] A. Mink, J. C. Bienfang, R. Carpenter, L. Ma, B. Hershman, A. Restelli and X. Tang, "Programmable instrumentation & gigahertz signaling for single-photon quantum communication systems", New Journal of Physics (043001-045025), # 045016 (April 2009).

<http://www.iop.org/EJ/article/1367-2630/11/4/045016/njp9_4_045016.pdf?request-id=fc24ca32-c620-4e46-a997-49df06baf1d5>

[13] A. Mink and A. Nakassis,"LDPC for QKD Reconciliation", The Computing Science and Technology Intern'l Journal, Vol. 2, No. 2, ISSN (Print) 2162-0660, ISSN (Online) 2162-0687 (June 2012). <http://www.researchpub.org/journal/cstij/archives.html>

[14] Z. Zhang, V. Anantharam, M. Wainwright and B. Nikolic. "A 47 Gbit/s LDPC Decoder with Improved Low Error Rate Performance", Symposium on VLSI Circuits, Kyoto, Japan, pp. 286-287 (June 2009).