ECE569 Project Proposal

CUDA Accelerated Implementation For Low-Density Parity-Check Code Processing

Christopher Brown, Jared Causey

brownca@email.arizona.edu

jaredcausey@email.arizona.edu

Group 3 on D2L

Abstract	Communication between wireless systems comes with a risk of data loss. This loss can be mitigated or removed through the use of Low-Density Parity-Check (LDPC) codes. LDPC codes can be transmitted near the max data channel limit while offering efficient data recovery upon reception. These codes, while effective, can require high computational overhead. As more data is sent over channels, more error encoding is needed. This drastically increases the data size that must be decoded therefore increasing computation time. We propose a parallel implementation of Gallgher's Algorithm B (GaB) to decode these LDPC codes. GaB itself can be divided into parallelized tasks and the Compute Unified Device Architecture (CUDA) provides the optimal framework for this algorithm's execution and demanding throughput.
Benefits	The GPU based implementation strives to provide increased performance. The threading scalability of the GPU means that, should the anticipated performance gains be met, this implementation can be scaled to more powerful GPUs. In addition, this could easily

	scale to large GPU clusters if so desired. This would allow for the wireless communication domain to handle larger datasets. This corresponds to more users on a network with faster data rates without having to sacrifice error correction capabilities.
A maximum of eight key words that describe the project	LDPC, GPU, Error correction, Decoder, Parallel Processing, CUDA, Gallgher

1 IDENTIFICATION AND SIGNIFICANCE OF THE PROBLEM OR OPPORTUNITY

1.1 Statement of Problem

Low-Density Parity-Check (LDPC) codes are predefined lengths of data that are encoded with parity bits throughout the data to ensure the integrity of the data itself. LDPC codes have found extensive use in various wireless communication systems in the last decade. In more recent years, they have become the backbone for efficient communication in the 5 Ghz band. These codes can be transmitted near the Shannon channel limit while still maintaining a high degree of error correcting efficiency. The quick speed of simple parity corrections coupled with the demand for high throughput make a multithreaded solution in a GPU desirable.

In his Doctoral Thesis “Low Density Parity-Check Codes”, Robert Gallager proposed two algorithms to efficiently process LDPC codes. For the purpose of this project, we will focus on enhancing algorithm B (GaB) as it is an extension upon algorithm A. With this algorithm, errors spread throughout a received transmission can be corrected via simultaneous parity checks.

Bottlenecks arise for the algorithm when parity corrections become dependent on the result of other parity corrections. This necessary communication between processes also lends itself to benefitting from a parallel solution that shares a common memory among threads.

As the demand for faster network throughput increases, so does the need for reliable transmission of that data. It is important that we research and improve a solution to this ever growing problem like the GaB algorithm.

Problem Background

Gallagher Algorithm B (GaB) is an optimizable algorithm that is used to decode Low-Density Parity-Check (LDPC) codes. The algorithm is composed of two components: the bit nodes and the check nodes. These two types of nodes interact with each other to correct all potential errors in the received code.

Each bit node represents a single bit in a received LDPC code. Check nodes validate connected bit nodes through a small parity calculation. A bit node's value may change if a check node corrects it. Upon a bit node’s modification, mutually connected check nodes must be notified to recalculate their parity calculations. Once all check node operations are completed the code is deemed correct and the algorithm is complete.

The number of bit nodes is equivalent to the number of bits in a LDPC code. The number of check nodes and their corresponding connections to the bit nodes are determined at the time of encoding, usually by a standard. The connections between check and bit nodes are better presented by an CxV matrix where C represents the number of check nodes and V is the number of bit nodes.

The need to parallelize GaB is realized when a LDPC code has been received with several intertwined parity errors that affect the computation results of multiple check nodes. In a serial implementation of GaB, such a change would take several loop iterations to resolve which becomes more computationally expensive as all check nodes must be resolved before the algorithm is complete.

1.2 Current Level of Technology

There are currently both serial and parallel implementations of LDPC decoders. The serial implementation provided to us also applies the GaB algorithm to a provided dataset. However, the serial implementation doesn't take advantage of threading via the CPU which greatly lengthens its runtime for codes affected by more noise.

An FPGA implementation was developed which showed vast improvements in runtime compared to the serial implementation. At an "Alpha Environment" level of 0.01 the FPGA implementation saw over a 147x speedup over the serial implementation. The "Alpha Environment" value is used to simulate the amount of noise affecting the transmission. A lower value such as 0.01 represents a very error prone transmission environment.

While we originally planned to write the CUDA implementation of GaB from scratch, we have found another LDPC decoder that will serve as a good baseline to start optimizing from. This implementation uses the min-sum algorithm in the check nodes to determine hard bit values for the bit nodes. Our focus for this project will be to implement GaB and optimize the decode process by improving memory access speeds. [1][2]

1.3 Limitations of Current Technology

Some implementations of LDPC decoders are done utilizing FPGAs. While these can offer greater performance over a CPU, they do not have the parallelization opportunities of a GPU. In addition, some of the serial CPU implementations do not utilize the full threading capability of a CPU. This leaves large room for performance gain and algorithmic simplification via a GPU based implementation.

As society's need for higher network speeds increases, it is important to stay ahead of those demands. The amount of wireless devices transmitting and receiving signals is ever growing, which leads to an error prone environment ridden with noise. Both of these issues coupled together push us towards a GPU solution capable of decoding and correcting LDPC codes.

1.4 Proposed Solution

We plan to streamline both the throughput of data and the check node computation of the GaB algorithm through parallelization efforts in the CUDA architecture. To quickly receive the LDPC codes into the device, we will implement data streaming to the GPU. Streaming will allow the host to consistently send sequential data over to the GPU to be decoded. This will streamline the movement of LDPC codes and the corrected results between the host machine and GPU. As the GPU processes one set of data, the next set will be copied to the GPU's global memory.

In addition to streaming data, data will be operated on in shared memory. Since the largest LDPC code in commercial use is at most 4 kB long, there is ample storage per thread block to store a code in shared memory. [3]

With LDPC, each bit node corresponds to an equivalent bit in memory. The check nodes can be represented as a thread in a single thread block. Because of this, a single GPU thread block will be used to execute a single LDPC code. Another possible optimization, would be to keep the CxV matrix stored in constant memory and use it to route check nodes to their corresponding bit nodes. We believe we can implement some form of tiling to quickly access the code bits in global memory and move them to shared memory, but we are stuck trying to make sure redundant read and write operations do not occur between check nodes that use the same bit nodes.

1.4.1 Background

The Gallager B LDPC algorithm was originally proposed in 1963. At the time, the world lacked sufficiently powerful hardware to make the algorithm practical. In the late 1990s, the algorithm was rediscovered and quickly found use in space based communications. The increasing prevalence of wireless devices has made the Gallager B algorithm especially important. It's ability to operate near the Shannon channel limit means that error correction can be done with little effect to overall data rates.

1.4.2 Statement of Solution

For streaming, there are several things to consider. The performance advantage of streaming data comes for when the GPU takes long to decode a section of data. If this decoding time is longer than the time it takes to copy the next set of data to the device, then streaming will provide a performance boost. However, if the GPU can decode data faster than the host can copy data over, streaming no longer becomes a viable option.

The matrix used to coordinate which V and C nodes are connected will be placed in constant memory. This will completely remove the need to stream this data to the GPU. This information will be available for all threads to access.

All data will be operated on out of shared memory. Shared memory allows for faster fead times than global memory does. However it is critical to ensure that reading from the global memory into shared memory is done efficiently. For this, the global memory reads will take advantage of memory coalescing. This means that subsequent data is stored next to each other in memory. This allows the full utilization of the GPU's memory bandwidths.

We are unsure of the exact speedup compared to the FPGA implementation because we do not know exactly how it was implemented. We only know how the FPGA implementation stacks up against the serial implementation. If we assume GaB was parallelized on an FPGA, then we can expect the CUDA implementation to be at least as fast as the FPGA implementation. We anticipate that faster memory accesses and the use of constant memory will improve upon the FPGA version referenced previously. The combination of shared and constant memory will reduce the number of global memory accesses needed. Streaming data will ensure the GPU does not have overhead to wait for the next group of LDPC codes to become available for processing.

1.4.3 How Solution Overcomes Current Technology

Our solution will offer significant performance improvements over serial and nominal improvements over FPGA implementations. These runtime improvements allow for more room to operate on larger data sets with little sacrifice to runtime speeds.

The benefit of our solution is it will provide a balance between optimal accuracy and runtimes. Our solution has the potential to offer sufficient runtime speeds despite the large datasets. This makes Viterbi decoders generally more portable as our solution eliminates the need for specialized hardware and instead utilizes GPUs. Additionally, a software solution allows the decoder to be easily tweaked and configured.

The GPU implementation can take advantage of GPUs threading capabilities to provide faster runtimes over current sequential implementations. The performance gain from the GPU allows the Viterbi algorithm to be scaled up to handle larger datasets. The GPU implementation will keep the runtime of these larger datasets to a minimum. This is vital for ensuring no unnecessary slowdowns are added to communications.

Utilizing a heavily parallelized GPU implementation will allow convolutional codes to be decoded with the same accuracy as traditional implementations but at a fraction of the runtime cost. Lower runtime costs also leave more room for increasing the amount of error correction without causing significant bottlenecks to data rates. This additional error correction would allow for data to be recovered over increasingly noisy and unreliable communication channels. The GPU implementation ensures that this extra reliability does not hamper data rates.

1.4.4 Validation and Evaluation Strategy

Validation will focus on two things: computation time for decoding, and the resulting Bit Error Rate (BER). The decoding computation time will prioritize several sections. Individual decoders will be measured to calculate the efficiency of each decoder kernel. In addition to this, the overall computation time for decoding will be done. This means that, for a particular dataset, the time to completely decode the entire dataset will be determined. Additional timings may be done on individual algorithm components for optimization purposes. These timings can be useful to identify any bottlenecks in the algorithm implementation itself. This will be particularly useful for determining the effectiveness of the streaming optimization strategy. As mentioned earlier, streaming may not be an effective strategy if the decoder can process faster than the data can be transferred to the device. Benchmarking the decoder algorithm will determine if our implementation falls within this case.

When transmitting data over a channel, bits will be altered or changed due to channel noise, interference, or other errors. The bit error rate is the number of bit errors over time. Determining the BER of our LDPC implementation is critical to verifying its effectiveness. This will determine not only how the algorithm fundamentally works, but also how well it works.

The computation time and BER will provide a clear picture of our solution's effectiveness as well as prove the usefulness of this algorithm on a GPU. The computation time along with the BER will showcase the algorithm can provide the necessary error correction without sacrificing performance.

2 TECHNICAL OBJECTIVES

The technical objectives of this project is to develop a highly parallelized GPU implementation of the Gallager B LDPC decoding algorithm. This proposal leverages the GPU’s threading scalability as well as several optimization techniques to ensure high performance. The initial effort will be centered around building off the serial implementation to form a parallelized GPU implementation. From there, the individual optimization strategies discussed earlier will be applied. Simultaneously, these strategies will be benchmarked to validate their effectiveness.

2.1 Project Milestones and Tasks

2.1.1 Milestone 1: Obtain and analyze available LDP datasets

- Investigate the viability of available datasets mentioned here [4].
  - Lead: Christopher Brown
- Identify alternative datasets should the current one not be viable
  - Lead: Jared Causey

2.1.2 Milestone 2: Implement working LDPC decoding algorithm on GPU

- Write kernel code to perform LDPC decoding on a GPU
  - Lead: Christopher Brown

2.1.3 Milestone 3: Implement Optimization Strategies

- Implement streaming based optimization
  - Lead: Jared Causey
- Implement constant memory optimization
  - Lead: Christopher Brown
- Implement shared memory optimization
  - Lead: Jared Causey

2.1.4 Milestone 4: Validate optimization strategies have desired effect

- Validate streaming optimization
  - Lead: Christopher Brown
- Validate constant memory optimization
  - Lead: Jared Causey
- Validate shared memory optimization
  - Lead: Christopher Brown
- Validate overall performance
  - Lead: Jared Causey

2.2 Work Plan Schedule

The initial work will be done to verify publicly available datasets for their viability. NExt, the initial GPU implementation will be created. This implementation will be verified as working correct and provide a baseline from which to optimize on. Next the different validation strategies we discussed will be applied. These strategies will then be benchmark to verify their effectiveness. Final benchmarking will then be performed on the entire implementation to identify any unforeseen performance bottlenecks and to verify we achieve high degrees of accuracy.

Table : Task Schedule

Task	**Description**	**Weeks**					
		1	**2**	**3**	**4**	**5**	**6**
1	Validate Available datasets	X					
2	Implement Generic GaB Algorithm		X	X			
3	Apply optimization strategies			X	X		
4	Benchmark and Tweak Individual optimization strategies				X	X	X
5	Final benchmarking and optimizations					X	X
6	Write Final Report				X	X	X

References

[1] Wang, Guohui, et al. "A Massively Parallel Implementation of QC-LDPC Decoder on GPU." *IEEE Xplore*, https://ieeexplore.ieee.org/abstract/document/5941084.

[2] Wang, Guohui, et al. "Robertwgh/CuLDPC: CUDA Implementation of LDPC Decoding Algorithm." *GitHub*, https://github.com/robertwgh/cuLDPC.

[3] "IEEE p802.3ca 50g-Epon Task Force." *IEEE P802.3ca 100G-EPON Task Force*, https://www.ieee802.org/3/ca/public/meeting_archive/2017/11/.

[4] Hui, Dennis. "Channel Coding in 5G New Radio: Tutorial Overview and Performance Comparison with 4G LTE." *IEEE DataPort*, IEEE, 8 Nov. 2018, https://ieee-dataport.org/documents/channel-coding-5g-new-radio-tutorial-overview-and-performance-comparison-4g-lte.

IEEE Paper Summaries:

A Scalable LDPC Decoder on GPU:

https://ieeexplore.ieee.org/abstract/document/5718799?casa_token=wVYp9ovbt2sAAAAA:JRJyUR8RIacbjmNbZSNJG6jCZJc-M5day6-ZHwIJUU2A-7BYi_lyZeMwVK6J9dShBpmhTKa4nQ

Summary:

The paper utilizes a layered decoding algorithm (turbo decoding message passing TDMP) for LDPC codes which is based on the Two-phase message-passing (TPMP) LDPC decoding algorithm. The TDMP algorithm devices check nodes into subgroups that are processed sequentially. The paper claims a 2x speed up over TPMP. This paper uses the min-sum algorithm in their check nodes. For this paper, they decode multiple codewords simultaneously and each codeword is processed by a block. The layered decoding algorithm offers a better ratio of arithmetic operations to memory accesses. Global memory reads were reduced by representing messages with 8 bit fixed point precision. This allowed 4 codewords to be contained within a single 32-bit word that the kernel code then unpacked. The authors also utilized streaming and found, through experimentation, that 10% of their total LDPC decoding time is used for data transfer. The authors also utilized shared memory to ensure all memory reads by the threads could be coalesced. Overall the authors achieved a peak throughput of 160 Mbps.

High Throughput LDPC Decoder on GPU:

https://ieeexplore.ieee.org/abstract/document/6715256?casa_token=M7JYnnjUt4YAAAAA:0u4cJMaSRGLRiGsf9lMmmRfnBciNBUMz_a8CFL04dPJh4-nQPKGAusv7MY8-zxRXhh62ggz3jQ

Summary:

The authors utilized a Sum Product Algorithm for their check nodes. The authors reduced global memory access and utilized more memory coalescing by mapping a set of codewords to a block. The set size was kept the same as the GPU warp size. This made it easy for threads to share results for different stages. The parity-check matrix is stored as an array. The threads in a warp process the same sections of different codewords so those threads only need a single element of the parity-check array. Codewords are also stored as 8-bit fixed-points so that multiple codewords are loaded simultaneously. The authors were able to achieve a throughput of over 550 Mbps. This achievement was done when running 10 iterations and larger iteration counts (e.g., 20, 30, etc. iterations) saw reduced throughput. The authors also mention their implementation is unsuitable for real-time applications due to the decoding latency for a single codeword being too high.

High Throughput Low Latency LDPC Decoding on GPU for SDR Systems

https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6737137&casa_token=RmLu4uHlDTkAAAAA:mp9hlYe3u2ZHKsoRKng0hHjgx_tECVk7xIAw2QJ5FW7LWdMFC7E8RQ5NhiWZrn2C4wWlvJDa1w&tag=1

Summary:

Goal of the paper is high throughput with low latency for SDR systems. The authors utilized two-min decoding algorithms, fully coalesced memory access and data/thread alignment. They also utilized asynchronous memory data transfer and multi-stream kernel executions. Messages are represented as 32-bit floats. The memory transfers between host and device is done using page-locked memory to get direct memory access (DMA). This yielded a 15% throughput improvement. Coalesced reads are performed to move data into shared memory where the threads can read/write from in a coalesced way. This makes all device memory accesses coalesced. The authors utilized “dummy” threads to keep memory access as 128-byte aligned. This came with a waste of thread resources but offered a 20% throughput improvement. Multiple streams were used to improve the latency. Codewords were divided up across multiple streams. This meant each stream was only decoding a small number of codewords. The resulting tests showed this design as achieving latency between .207 ms and 1.266 ms with one test run having a peak throughput of 316.07 Mbps.

A massively parallel implementation of QC-LDPC decoder on GPU

https://ieeexplore.ieee.org/abstract/document/5941084

Summary:

This paper utilizes the sum-product algorithm for check node processing. LAtency was reduced by utilizing multi-codeword parallel decoding. This combines multiple codewords into a macro-codeword that is decoded by a thread block. An additional kernel was developed for early termination. This sought to avoid any more computations as the decoder already converged on the correct codeword. The authors also utilized constant memory and memory coalescing. The authors were able to achieve throughputs of around 100.3 Mbps with their implementation on WiFi LDPC codes.

Massively LDPC Decoding on Multicore Architectures

https://ieeexplore-ieee-org.ezproxy2.library.arizona.edu/document/5445086

Summary:

The authors used the sum-product algorithm for check node processing. They tested parallelizing LDPC codes on multiple architectures, but we are mainly focusing on the CUDA. They used memory coalescing and various thread sizes to do their tests. Results showed that throughput increased for larger H matrices. They were also able to obtain faster results by limiting the data resolution for each piece of data from 16 bits to 8 bits. They would also store the H matrix in constant memory however for very large H matrices this was not possible as they ran out of constant memory storage. They were able to achieve throughputs near 40 Mbps on their CUDA implementation which far exceeded their other implementations.

A High Throughput Efficient Approach for Decoding LDPC Codes onto GPU Devices

https://ieeexplore.ieee.org/abstract/document/6762823

Summary:

BER is comparable to previously publish LDPC implementations, however decoding convergence is sped up twice as fast. They were able to speed up the node convergence by explicitly layering/staggering how the check nodes would operate. Overall speed up factors range from 2x to 18x depending on the matrix.