Optimizing Resource Efficiency in Gallager B LDPC Decoders

Dhanyapriya Somasundaram, Hanieh TotonchiAsl
dhanyapriyas@arizona.edu, haniehta@arizona.edu
ECE Department, College of Engineering, University of Arizona, Tucson, AZ

***Abstract*—This project explores the optimization of LDPC (Low-Density Parity-Check) decoders on GPU platforms, addressing the computational challenges and limitations of existing decoding algorithms. By leveraging massively parallel processing capabilities of modern GPUs, the project exploits different optimization schemes to significantly enhance throughput and efficiency. The proposed solution implements a two-stage decoding process utilizing both horizontal processing and a posteriori probability updates, orchestrated through CUDA kernels to exploit fine-grained parallelism. Experimental results demonstrate substantial improvements (12.16x speedup on average) in decoding throughput and reduction in memory latency, proving the effectiveness of the optimized architecture in real-world scenarios.**

***Index Terms*—GPU, LDPC Decoder, CUDA, Parallel Processing, Error-Correction Coding.**

I. Introduction

THE relentless advancement in digital communications demands even more efficient error-correction coding to ensure data integrity across noisy channels. Low-Density Parity-Check (LDPC) codes stand out as one of the most effective solutions for error correction in modern communication systems, used widely from deep-space communication to consumer broadband. Despite their efficacy, LDPC decoders are computationally intensive, requiring significant processing power, which poses challenges in terms of throughput and efficiency, particularly in real-time applications.

The GaB (Gaussian-Belief-propagation-based) algorithm faces several challenges primarily stemming from its computational intensity and memory access patterns. Firstly, the algorithm exhibits high computational intensity, requiring significant computational resources to perform its operations efficiently. Secondly, it encounters a low number of computations relative to the amount of memory access required, leading to inefficiencies in resource utilization. Moreover, there is minimal data reuse between consecutive computations, exacerbating the strain on memory bandwidth. Additionally, the algorithm's reliance on a sparse parity matrix results in a large set of random memory accesses, further impeding performance. Lastly, the iterative nature of the algorithm makes it particularly time-consuming on traditional CPUs, which lack the parallel processing capabilities necessary to expedite these iterative processes.

Current state-of-the-art LDPC decoders leverage parallel processing to some extent, yet they remain limited by suboptimal use of hardware resources, leading to inefficiencies in both power consumption and throughput. These limitations are evident in systems requiring high throughput rates under stringent power constraints, such as mobile devices and portable communications equipment.

This project introduces an integrated optimization approach for LDPC decoders using Graphics Processing Units (GPUs), designed to maximize the throughput while minimizing latency. Our method enhances the parallelism of LDPC decoders by using sparse representation, optimizing memory access patterns, and utilizing the massively parallel processing capabilities of modern GPUs more effectively. Specifically, we implement two main CUDA kernels to handle the decoding tasks, which are adapted to exploit the fine-grained parallelism available in GPUs.

Our results quantify the improvement, showing a throughput increase of up to 12.16x over traditional single-threaded LDPC decoding algorithms while reducing the memory access latency by approximately 48%.

Furthermore, unlike previous works which often focused on optimizing individual aspects of LDPC decoding, our approach provides a comprehensive solution by integrating optimizations at both the algorithmic and architectural levels. This dual focus allows for significant performance gains across various metrics, setting our work apart from existing methodologies.

To ensure a smooth flow of understanding, the remainder of this paper is as follows. Section II provides a sufficient overview of the GaB algorithm and Section III presents the related work. Then, Section IV details the methodology of the proposed solution, and the experimental results are discussed in Section V . Finally, Section VI concludes the paper.

II. An overview of the GaB Algorithm

The Gallager B Algorithm offers a simplified approach to decoding LDPC codes through hard-decision-making. Focusing on binary decisions in each iteration reduces the computational complexity typically associated with soft-decision methods. At its core, the algorithm operates through iterative message passing between variable nodes (VNUs) and check nodes (CNUs) in the Tanner graph structure of the LDPC code (Figure 1), facilitating efficient error correction until predefined criteria, such as a set number of iterations or error thresholds, are met.

The flowchart (Figure 2) provides a systematic representation of the decoding process for Gallager B LDPC decoders. It starts with codeword generation and noise addition, followed

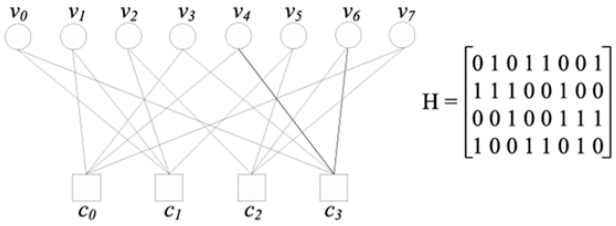

Fig. 1: Tanner graph (left) and its parity check matrix (right).

by iterative updates between variable and check nodes. The process checks for convergence by evaluating if the decoding has met the set criteria, such as the maximum number of iterations or satisfactory error performance. The loop continues until the decoder either achieves the desired fidelity or exhausts allowed iterations, ensuring a robust decoding mechanism adaptable to varying error conditions.

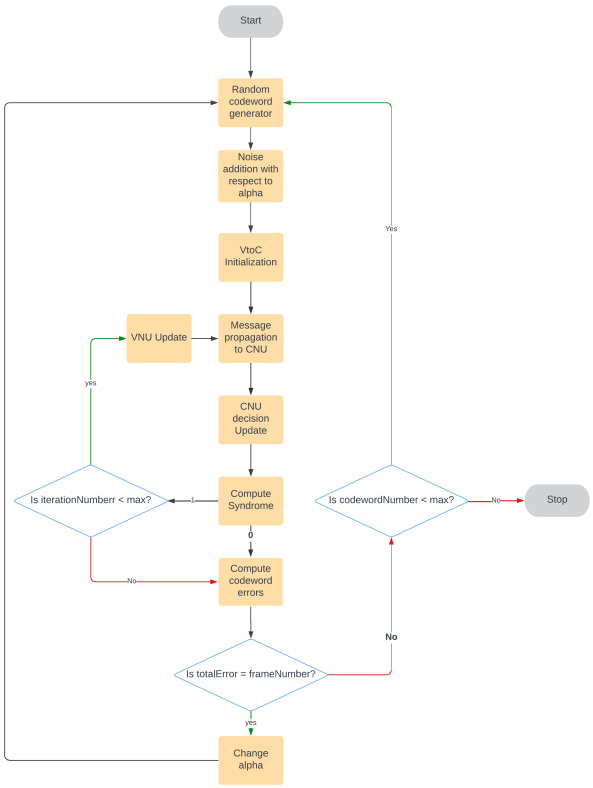

Fig. 2: Flowchart of the GaB Algorithm.

III. Related Work

The development of efficient LDPC decoders has been a focal point of research due to their crucial role in error correction for communications. A variety of strategies have been employed to enhance decoder performance, particularly through hardware optimizations.

[2] focuses on optimizing LDPC decoders on GPU platforms, utilizing the computational power of NVIDIA GTX470 GPU with 448 stream processors and 1280MB of GDDR5 device memory. The decoding process involves two stages: horizontal processing and a posteriori probability (APP) update. Each stage is implemented as a separate computational kernel running on the GPU, while the CPU handles CUDA initialization and memory management. For LDPC codes like 802.11n (1944, 972), multi-codeword decoding schemes (Figure 3) are employed to enhance parallelism, where each codeword is decoded by a thread block. Early termination algorithms are utilized to reduce unnecessary computations once convergence is achieved. Memory access optimization techniques include storing the parity check matrix in constant memory, resulting 40% of memory access and branch instructions reduction, and coalescing device memory access, resulting in 20% throughput improvements.

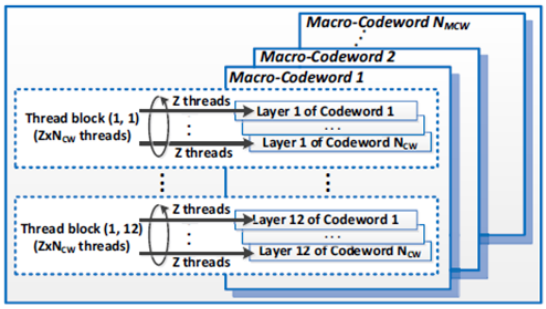

Fig. 3: Multi-codeword parallel decoding algorithm [2].

The experimental setup in [3] for LDPC decoder optimization involves an Intel Core 2 Duo CPU and an NVIDIA GeForce 9800 GTX+ GPU. For shorter code lengths, throughput is limited by the maximum active block per streaming multiprocessor (SM), while longer code lengths are constrained by register count. The microarchitecture of the GPU consists of 16 SMs, each with eight streaming processors running at 1.836GHz. In addition to previous work, a streaming version reduces data transfer time between CPU and GPU, enhancing overall efficiency (Figure 4).

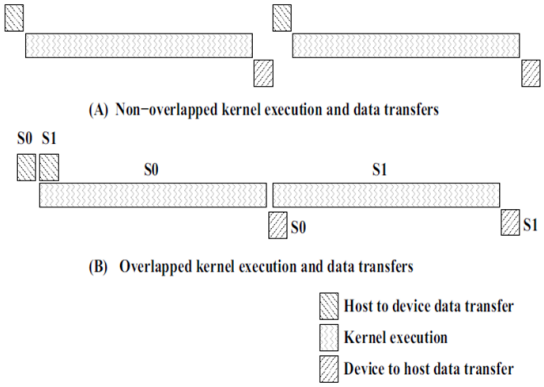

Fig. 4: Non and Multi-streamed execution process [3].

[4] explores parallel software LDPC decoding on GPUs, focusing on optimizing the Min-Sum Algorithm (MSA) to achieve high throughput. In this paper, global memory access is optimized by transferring frequently used data to shared memory, enhancing performance. Data access methods are improved by organizing thread operations to minimize memory latency as shown in Figure 5.

To process the same VNU element of the F frames at the same time, non-contiguous memory access would affect performance. To solve this issue, [5] performed a data interleaving

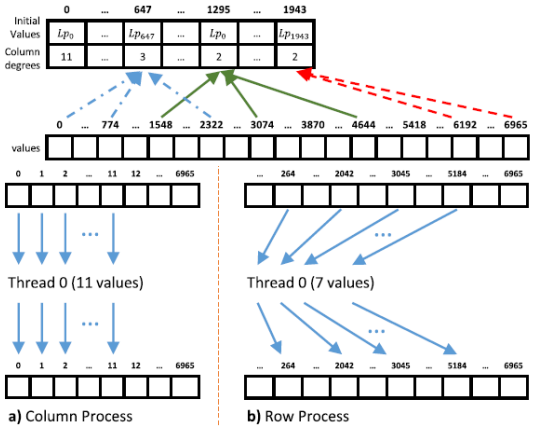

Fig. 5: Memory access pattern [4].

process (Figure 6) before and after the decoding stage to ensure that each set of F frames is reordered to achieve an aligned memory data structure.

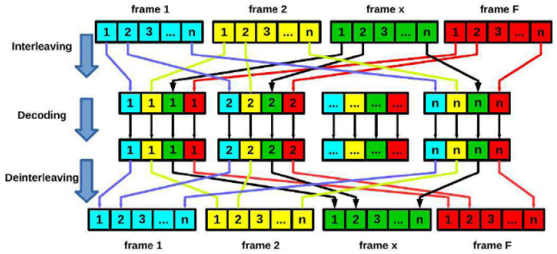

Fig. 6: Data interleaving/deinterleaving process [5].

The distinction of our work lies in its comprehensive optimization, which encompasses both algorithmic efficiency and hardware-specific enhancements. By integrating these two aspects, our approach significantly outperforms existing methods in terms of both throughput and decoding latency, as will be detailed in the subsequent sections of this report.

IV. Methodology

In this section, we detail the methodology adopted for optimizing resource efficiency in Gallager B LDPC decoders. Each version of the decoder introduces specific optimizations, progressively enhancing performance and efficiency. We outline the implementation details, including pseudocode and the associated improvements and drawbacks for each version.

In all versions, the parity check matrix undergoes compression into a sparse matrix representation in the preprocessing phase on the host side. This compression involves two key components: ColumnIndex, which denotes the column index of the non-zero element in each CNU, and Interleaver, which signifies the index number of ColumnIndex in column-wise order, intended for use with each VNU. These steps enable the efficient representation and organization of the matrix data, facilitating subsequent processing stages in the decoding algorithm.

A. ***Version-0:*** *Baseline Implementation of Gallager B LDPC Decoder in C*

This version presents a basic implementation of the Gallager-B LDPC decoder in C. The decoder utilizes the hard decision bit-flipping algorithm for decoding LDPC codes. Figure 7 shows the pseudocode for the main function having the iterative processing.

```
void main() {
  // -------------Iterative decoding phases----------------
  for (All Channel probabilities of error) {
    for (All Sent Codewords) {
        for (All iterations) {
            if(iter==0)
                DataPassGBIter0(...);
            else
                DataPassGB(...);
            CheckPassGB(...);
            APP_GB(...);
            IsCodeword=ComputeSyndrome(...);
            if (IsCodeword)
                break;
        }
        // Compute Statistics
        for (All Variable Nodes)
            if (Decide!=Codeword)
                NbError++;
        NbBitError=NbBitError+NbError;
        // Calculate NbTotalErrors based on IsCodeword
        // Stopping Criterion
        if (NbTotalErrors==NBframes)
            break;
    }
  }
}
```

Fig. 7: Version-0 Main Function.

- **DataPassGBIter0**: Performs the initial data pass from variable nodes to check nodes in the LDPC decoding process for the first iteration.
- **DataPassGB**: Updates the variable node messages based on the received word and messages received from connected check nodes in subsequent iterations.
- **CheckPassGB**: Updates the messages from check nodes to variable nodes based on the received messages from variable nodes.
- **APP_GB**: Makes decisions for variable nodes based on the global value calculated using the received word and messages from connected check nodes.
- **ComputeSyndrome**: Computes the syndrome of the received word based on the parity-check matrix.

While this implementation [1] provides a basic framework for LDPC decoding, it lacks optimization for efficiency and may not be suitable for high-throughput applications.

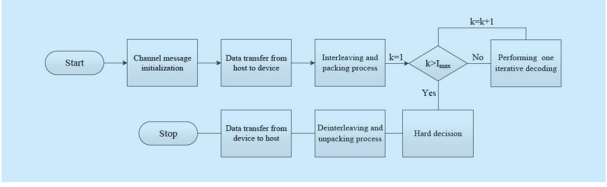

Fig. 8: GPU decoding procedure [6].

*B. **Version-1:** Enhanced LDPC Decoding on GPU with Parallel Processing and Shared Syndrome Reduction*

This version addresses the inefficiency of the baseline implementation by leveraging GPU parallel processing capabilities. By offloading LDPC decoding tasks to the GPU and implementing parallel processing techniques, such as parallel variable-to-check and check-to-variable message updates, the decoder significantly enhances throughput and efficiency for LDPC decoding tasks, making it suitable for high-throughput applications. Figure 8 shows the overall decoding procedure on GPU.

- **initVNU**: Initializes the variable-to-check message update process by distributing received word values to connected check nodes.
- **VNU**: Implements the variable-to-check message update calculation in parallel.
- **Checkdecide**: Determines the final decision for variable nodes based on check-to-variable messages and received word values.

```
// Implement variable-to-check message update
__global__ void initVNU(int *VtoC, int *receivedWord, int *interleaver) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int offset = columnDegree*idx;
    // Check if thread index is within the range of VarNum
    if (idx < VarNum) {
        for (int i = 0; i < columnDegree; i++) {
            VtoC[interleaver[offset + i]] = receivedWord[idx];
        }
    }
}
// Implement variable-to-check message update
__global__ void VNU(int *VtoC, int *CtoV, int *receivedWord, int *interleaver) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // Check if thread index is within the range of VarNum
    if (idx < VarNum) {
        int i;
        int offset = columnDegree*idx;
        int Global = (1 - 2 * receivedWord[idx]);
        // Perform variable-to-check message update calculation
        for (i = 0; i < columnDegree; i++)
            Global += (-2) * CtoV[interleaver[offset + i]] + 1;
        for (i = 0; i < columnDegree; i++) {
            int buf = Global - ((-2) * CtoV[interleaver[offset + i]] + 1);
            // Store the updated message in the VtoC array
            if (buf < 0)
                VtoC[interleaver[offset + i]] = 1;
            else if (buf > 0)
                VtoC[interleaver[offset + i]] = 0;
            else
                VtoC[interleaver[offset + i]] = receivedWord[idx];
        }
    }
}
// Implement decision making based on check-to-variable messages
__global__ void Checkdecide(int *decide, int *CtoV, int *receivedWord, int *interleaver) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // Check if thread index is within the range of VarNum
    //  if (idx == 0)
    //      decide[0] = 0;
    if (idx < VarNum) {
        int offset = idx * columnDegree;
    //  int idxShifted = idx + 1;
        int Global = (1 - 2 * receivedWord[idx]);
        for (int i = 0; i < columnDegree; i++)
            Global += (-2) * CtoV[interleaver[offset + i]] + 1;

        if  (Global > 0)
            decide[idx] = 0;
        else if (Global < 0)
            decide[idx] = 1;
        else
            decide[idx] = receivedWord[idx];
    }
}
```

Fig. 9: Version-1 VNU Kernels.

- **CNU**: Executes the check-to-variable message update process, ensuring efficient communication between check and variable nodes.
- **ComputeSyndrome1**: Computes syndrome values using the decide matrix by XOR calculation.

```
// Implement check-to-variable message update
__global__ void CNU(int *CtoV, int *VtoC) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // Check if thread index is within the range of CheckNum
    if (idx < CheckNum) {
        int i;
        int signe = 0;
        int offset = rowDegree*idx;
        // Calculate signe
        for (i = 0; i < rowDegree; i++) {
            signe ^= VtoC[offset + i];
        }
        // Update CtoV
        for (i = 0; i < rowDegree; i++) {
            CtoV[offset + i] = signe ^ VtoC[offset + i];
        }
    }
}
// Implement syndrome computation using the decide matrix
__global__ void ComputeSyndrome1(int *decide, int *columnIndex, int* syndrome) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < CheckNum) {
        syndrome[idx] = decide[columnIndex[idx*rowDegree]]^
                        decide[columnIndex[idx*rowDegree+1]]^
                        decide[columnIndex[idx*rowDegree+2]]^
                        decide[columnIndex[idx*rowDegree+3]]^
                        decide[columnIndex[idx*rowDegree+4]]^
                        decide[columnIndex[idx*rowDegree+5]]^
                        decide[columnIndex[idx*rowDegree+6]]^
                        decide[columnIndex[idx*rowDegree+7]];
    }
}
// Implement final syndrome ORing calculated syndrome matrix
__global__ void ComputeSyndrome2_1(int* syndromeOut, int* syndromeIn) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int tid  = threadIdx.x;
    // in each block, we have log(block dimension) steps to calculate the sum
    for (int stride = 1; stride < blockDim.x; stride *= 2)
    {
        // just some specific threads are involved in calculation
        if (idx % (2*stride) == 0)
            syndromeIn[idx] |= syndromeIn[idx + stride];
    }
    // write back the last partial sum to dout added in the next phase kernel run
    if (tid == 0)
        syndromeOut[blockIdx.x] = syndromeIn[idx];
}
// Implement syndrome computation using the decide matrix
__global__ void ComputeSyndrome2(int* syndromeOut, int* syndromeIn) {
    // Determine global thread index
    int idx = blockIdx.x * 2*blockDim.x + threadIdx.x;
    int tid  = threadIdx.x;
    // shared memory for syndrome calculation of each thread
    extern __shared__ int shared_syndrome[];
    // pre-calculation to half # of threads
    shared_syndrome[tid] = syndromeIn[idx] | syndromeIn[idx+blockDim.x];
    //  shared_syndrome[tid] = decide[columnIndex[idx]];
    __syncthreads();
    for (int stride = blockDim.x/2; stride > 0; stride >>= 1)
    {
        if (tid < stride)
            shared_syndrome[tid] |= shared_syndrome[tid + stride];
        __syncthreads();
    }
    if (tid < 32) {
        shared_syndrome[tid] |= shared_syndrome[tid + 32];
        shared_syndrome[tid] |= shared_syndrome[tid + 16];
        shared_syndrome[tid] |= shared_syndrome[tid + 8];
        shared_syndrome[tid] |= shared_syndrome[tid + 4];
        shared_syndrome[tid] |= shared_syndrome[tid + 2];
        shared_syndrome[tid] |= shared_syndrome[tid + 1];
    }
    if (tid == 0)
        syndromeOut[blockIdx.x] = shared_syndrome[tid];
}
```

Fig. 10: Version-1 CNU Kernels.

- **ComputeSyndrome2_1**: Performs parallel reduction OR computation.
- **ComputeSyndrome2**: Further optimizes reduction OR computation by utilizing and pre-processing in shared memory, enhancing overall efficiency, referred to as Version 1.5 in the rest of the paper.

While this version significantly enhances LDPC decoding efficiency through GPU parallel processing, it may still face limitations in handling extremely large scenarios with high

levels of parallelism, requiring further optimization for scalability and adaptability to diverse hardware configurations.

C. ***Version-2:*** *Optimizing LDPC Decoders through Constant Memory Utilization on GPU*

In Version-2, we tackle the scalability limitation of Version-1 by optimizing memory access patterns and reducing redundant computations. By utilizing constant memory, we minimize memory access latency and enhance overall efficiency, thus improving scalability and adaptability to diverse hardware configurations. By storing frequently accessed, read-only data such as the columnIndex and interleaver arrays in constant memory, this version ensures that all threads can access this data quickly and simultaneously without the overhead of global memory fetches. This approach significantly improves the decoder performance by leveraging the fast, cached access provided by constant memory in GPU architectures.

```
// Define constant memory for columnIndex and Interleaver
__constant__ int  columnIndex[CheckNum*rowDegree];
__constant__ int  interleaver[CheckNum*rowDegree];
// copy data to device
CUDA_CHECK(cudaMemcpyToSymbol(interleaver, hostInterleaver, BranchNum*sizeof(int)));
CUDA_CHECK(cudaMemcpyToSymbol(columnIndex, hostColumnIndex, CheckNum*rowDegree*sizeof(int)));
```

Fig. 11: Version-2 Constant memory exploitation.

Despite the optimizations introduced in Version-2, the constant memory size limitation (64KB per kernel) may restrict the size of arrays that can be stored, potentially impacting performance for larger LDPC codes. Additionally, while constant memory usage improves efficiency for read-only data, it may not fully address performance challenges in scenarios with extremely large LDPC codes or high levels of parallelism, requiring additional optimizations for optimal scalability and adaptability.

```
void main() {
    ...
    //// Declare an array of CUDA streams
    cudaStream_t streams[StreamNum];
    //// Specify stream size based on the input length
    unsigned int StreamSizeVar = VarNum/StreamNum;
    unsigned int StreamSizeCheck = CheckNum/StreamNum;
    ////  Create streams
    for (All streams)
    cudaStreamCreate(&streams[j]);
    ...
    for (All streams) {
      cudaMemcpyAsync(deviceDecide+(j*StreamSizeVar), hostDecide+(j*StreamSizeVar),
        StreamSizeVar*sizeof(unsigned int), cudaMemcpyHostToDevice, streams[j]);
      ...
    }
    ...
    for (All Channel probabilities of error) {
        for (All Sent Codewords) {
            ...
            for (All iterations) {
                cudaMemset(..., 0, ...*sizeof(unsigned int));
                ...
                for (All streams) {
                //// call kernels with new splitted data
                //// Synchronize all streams before continuing
                for (int i = 0; i < StreamNum; i++)
                    cudaStreamSynchronize(streams[i]);
                cudaDeviceSynchronize();
                }
                ...
            }
            ...
        }
    }
}
```

Fig. 12: Version-3 Main Function.

D. ***Version-3:*** *Advanced GPU Acceleration for LDPC Decoders Using Shared Memory, Local Variables, and Streaming*

In Version-3, we addressed the drawbacks of Version-2 by introducing shared memory to reduce memory read accesses and improve kernel performance. Additionally, we optimized data movement by exploiting asynchronous streams, enhancing the overall efficiency and throughput of the LDPC decoder.

```
__global__ void initVNU(int *VtoC, int *receivedWord, int size) {
    ...
    if (idx < size) {
        int writeData = receivedWord[idx];
        for (int i = 0; i < columnDegree; i++) {
            VtoC[interleaver[offset + i]] = writeData;
        }
    }
}
__global__ void VNU(int *VtoC, int *CtoV, int *receivedWord, int size) {
    ...
    int writeData = receivedWord[idx];
    // shared memory
    extern __shared__ int shared_CtoV[];
    if (idx < size) {
        for (i = 0; i < columnDegree; i++)
            shared_CtoV[offset + i] = (-2) * CtoV[interleaver[offset + i]] + 1;
        //__syncthreads();
        int Global = (1 - 2 * writeData);
        for (i = 0; i < columnDegree; i++)
            Global += shared_CtoV[offset + i];
        for (i = 0; i < columnDegree; i++) {
            int buf = Global - shared_CtoV[offset + i];
            ...
                VtoC[interleaver[offset + i]] = writeData;
        }
    }
}
__global__ void Checkdecide(int *decide, int *CtoV, int *receivedWord, int size) {
    ...
    // shared memory
    extern __shared__ int shared_CtoV[];
    // Check if thread index is within the range of VarNum
    if (idx < size) {
        int writeData = receivedWord[idx];
        for (i = 0; i < columnDegree; i++)
            shared_CtoV[offset+i] = (-2) * CtoV[interleaver[offset + i]] + 1;
        //  __syncthreads();
        int Global = (1 - 2 * writeData);

        for (int i = 0; i < columnDegree; i++)
            Global += shared_CtoV[offset+i];
        ...
            decide[idx] = writeData;
    }
}
```

Fig. 13: Version-3 VNU Kernels.

Despite these optimizations, Version-3 may still face challenges with large-scale LDPC decoding due to limitations in shared memory size and kernel execution efficiency, especially in scenarios with extremely high throughput requirements. Further optimization and fine-tuning may be necessary to address these concerns and achieve optimal performance.

E. *Change Index Order Optimization*

For the last optimization technique, we try changing the order of data to provide coalesced memory access. As Figure 15 illustrates, the ChangeOrderC and RestoreOrder kernels implement matrix transposition techniques to optimize memory access patterns for GPU processing. The ChangeOrderC kernel rearranges the input vector according to a specified index mapping, effectively performing a transposition of matrix segments. Conversely, the RestoreOrder kernel reverses this process, restoring the original order of elements to ensure data integrity for further computations.

These operations enhance memory efficiency and access speed, crucial for the performance of parallel computations in LDPC decoding on GPUs. However, captured execution

```
__global__ void CNU(int *CtoV, int *VtoC, int size) {
    ...
    // shared memory
    extern __shared__ int shared_VtoC[];

    // Check if thread index is within the range of CheckNum
    if (idx < size) {
        for (i = 0; i < rowDegree; i++)
            shared_VtoC[offset+i] = VtoC[offset + i];
        //__syncthreads();
        ...
        for (i = 0; i < rowDegree; i++)
            signe ^= shared_VtoC[offset+i];
        // Update CtoV
        for (i = 0; i < rowDegree; i++)
            CtoV[offset + i] = signe ^ shared_VtoC[offset+i];
    }
}
__global__ void ComputeSyndrome1(int *decide, int* syndrome, int size) {
    ...
    // shared memory
    extern __shared__ int shared_Decide[];

    if (idx < size) {
        for (int i = 0; i < rowDegree; i++)
            shared_Decide[offset+i] = decide[columnIndex[offset+i]];
        __syncthreads();
        syndrome[idx] = shared_Decide[offset]^shared_Decide[offset+1]^
                        shared_Decide[offset+2]^shared_Decide[offset+3]^
                        shared_Decide[offset+4]^shared_Decide[offset+5]^
                        shared_Decide[offset+6]^shared_Decide[offset+7];
    }
}
```

Fig. 14: Version-3 CNU Kernels.

time shows that the latency is increased by applying this optimization technique. Thus, we didn't include this method in our implementations.

```
__global__ void ChangeOrderC(int* vectorOut, int* vectorIn, int* index, int num, int degree) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < num) {
        for (int i = 0; i < degree; i++)
            vectorOut[idx+num*i] = vectorIn[index[idx*degree+i]];
    }
}
__global__ void RestoreOrder(int* vectorOut, int* vectorIn, int* index, int num, int degree) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < num) {
        for (int i = 0; i < degree; i++)
            vectorOut[index[idx*degree+i]] = vectorIn[idx+num*i];
    }
}
__global__ void VNUChanged(int *VtoC, int *CtoV, int *receivedWord) {
    // Determine global thread index
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    // Check if thread index is within the range of VarNum
    if (idx < VarNum) {
        int i;
        int Global = (1 - 2 * receivedWord[idx]);
        // Perform variable-to-check message update calculation
        for (i = 0; i < columnDegree; i++) {
            Global += (-2) * CtoV[idx+VarNum*i] + 1;
        }
        for (i = 0; i < columnDegree; i++) {
            int buf = Global - ((-2) * CtoV[idx+VarNum*i] + 1);
            // Store the updated message in the VtoC array
            if (buf < 0)
                VtoC[idx+VarNum*i] = 1;
            else if (buf > 0)
                VtoC[idx+VarNum*i] = 0;
            else
                VtoC[idx+VarNum*i] = receivedWord[idx];
        }
    }
}
```

Fig. 15: Changing order.

V. Evaluation Results

A. Functionality Verification

The evaluation of our optimized LDPC decoders on GPUs was conducted using a detailed experimental setup designed to assess various performance metrics, thereby validating the effectiveness of our methodologies.

- **Experimental Testbed:** All experiments were conducted on a Tesla P100-PCIE GPU, which includes 16 GB of global memory. This high-performance GPU is built on NVIDIA's Fermi architecture, known for its robust processing capabilities and efficiency in handling parallel computations. The CUDA toolkit version 11.0 was utilized for developing and executing all CUDA kernels.
- **Performance Metrics:** The primary metrics used for evaluating the decoders were: (1) Throughput (Mbps) which measures the amount of data processed per second, which is crucial for real-time decoding applications, and (2) Decoding Latency (ms): The time taken to decode a single codeword, important for applications requiring low response times.
- **Data Collection Strategy:** Data was generated using the Monte Carlo method, ensuring randomness and diversity in test cases. Errors were applied to the generated data based on different alpha values, simulating various channel conditions and error rates. The generated data were stored in their corresponding files to be executed in all versions for fair comparison.
- **Benchmarks and Parameters:** The benchmark used for evaluation was the regular LDPC codes with (N, dv, dc, R) configurations of (1296, 4, 8, 0.5). After analysis, which will be explained in the next subsection, the block dimension, number of codewords, and number of streams were selected as 64, 1000G, and 1, respectively.

```
module load cuda11/11.0
gcc -O gabc GaBR_Final.c -lm
nvcc -G -g -o gab1 GaB1_Final.cu
nvcc -G -g -o gab2 GaB2_Final.cu
nvcc -G -g -o gab3 GaB3_Final.cu
./gabc
./gab1
./gab2
./gab3
#Parameter Analysis
#for blockDim in 32 64 128 256 512 1024 2048
#do
#   for NbMonteCarlo in 1 10 100 1000 10000
#   do
#       for StreamNum in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
#       do
#           ./gab3 $blockDim $NbMonteCarlo $StreamNum
#       done
#   done
#done
```

Fig. 16: Script used for compilation and program run.

- **Validation Approach:** To ensure the reliability and reproducibility of our experimental results, the following validation approach was meticulously designed. This structured methodology allows other researchers or engineers to replicate our setup and confirm the effectiveness of our optimization strategies for Gallager B LDPC decoders using GPUs. Both the baseline version (Version 0) and the optimized versions (Versions 1, 2, and 3) of the LDPC decoder were implemented. Each version was tested under identical conditions to measure and compare improvements accurately. Care was taken to ensure that

the only differences between the versions were those related to the specific optimizations being tested. Figure 16 shows the .sh script used for compilation, program run, and parameter analysis.

B. Experimental analysis and comparison

The results of our experiments are presented through a series of graphs and tables, each designed to clearly illustrate the performance impacts of our optimizations.

- **Throughput Analysis:** A line graph depicting throughput achieved by each decoder version with varying numbers of codewords processed simultaneously. The x-axis represents the number of codewords, and the y-axis is the throughput in Mbps. Figure 17 demonstrates a clear upward trend in throughput as the number of codewords increases.

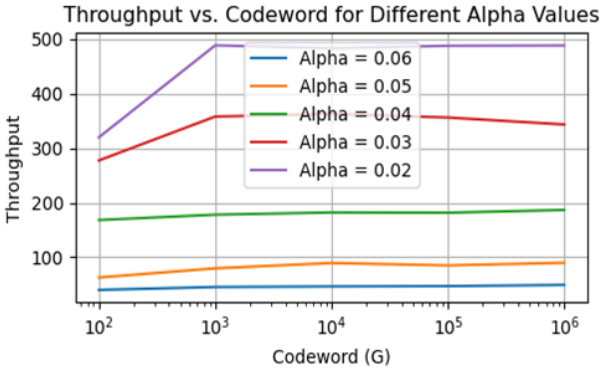

Fig. 17: Throughput versus Codeword.

Key observations include:

  - **Decreasing Alpha Enhances Throughput:** As Alpha increases, the throughput decreases. The graph shows that with an Alpha of 0.02, throughput rapidly increases as codeword size increases from 100G to 1000G, and then stabilizes. For higher Alphas (0.04, 0.05, 0.06), throughput remains relatively flat across all codeword sizes.
  - **Saturation Point:** The throughput reaches a saturation point where further increases in codeword size do not yield significant gains in throughput. This could indicate the limitations of the system or algorithm at higher data volumes.

- **Decoding Latency Comparison:** Figure 18 compares the decoding latency across different block dimensions for different channel probability of error.
  Key observations include:
  - **Decreasing Trend:** The execution time decreases sharply as the block dimension increases from 32 to 512, particularly noticeable for Alpha = 0.02. This suggests that larger block dimensions allow for more efficient parallel processing, reducing execution time.
  - **Alpha Sensitivity:** The effect of increasing block dimensions appears to be more pronounced at lower Alpha values. At higher Alphas (0.04, 0.05, 0.06),

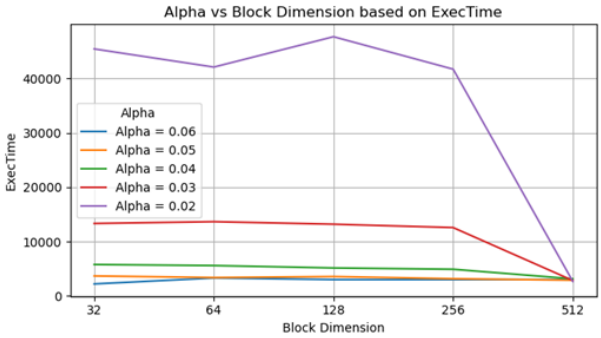

Fig. 18: Execution time versus Block Dimension.

the execution time decreases modestly and flattens out, indicating less sensitivity to block dimension changes.
  - **Optimal Block Dimension:** While increasing the number of threads per block can potentially improve performance by utilizing more parallelism, it also introduces challenges related to resource usage, synchronization, and data integrity, which can impact the correctness of error correction functionality and lead to undetected errors. There is an optimal block dimension (around 64 to 128) where execution time minimizes for higher Alpha values, the error correction functionality is correct, and the statistics remain the same. This optimal point represents the best balance between parallel processing efficiency and memory and thread handling overhead management.

In summary, balancing throughput and execution time in LDPC decoding on GPUs requires careful consideration of parameters such as Alpha and block dimension. Lower Alpha values can enhance throughput, but optimizing execution time necessitates thoughtful adjustment of block dimensions. System designers aiming to scale up LDPC decoding systems on GPUs must account for these parameters, as they significantly influence performance and efficiency. Consequently, achieving optimal performance requires a nuanced understanding of how Alpha and block dimensions interact to balance throughput and execution time effectively.

The execution times recorded in Table I suggest several performance characteristics and trends across different versions and values of alpha (channel error probability). Here's a detailed root cause analysis:

- **Version Scalability and Efficiency:** Version-0 (C implementation) exhibits the longest execution times, which drastically increase as alpha decreases. This pattern suggests a higher computational demand due to increased error rates and the iterative nature of the LDPC decoding process, which becomes more intense as errors increase. CUDA Versions (1 through 3) show a significant reduction in execution time compared to Version-0. This improvement is primarily due to leveraging the parallel processing capabilities of GPUs, which handle simultaneous operations more efficiently than sequential CPU processing.

- **Impact of Alpha:** As alpha decreases, indicating a lower error probability, the execution time increases for all versions. This trend is expected as a lower error probability often requires more iterations to achieve convergence in LDPC decoding, particularly under stringent error-correction requirements.
- **Optimization Efficacy:** Version-1 shows a substantial decrease in execution time compared to the C implementation, benefiting from basic parallel processing on the GPU. Version-1.5 and Version-2 further decrease the time by optimizing memory access patterns and utilizing constant memory. However, Version-1.5 occasionally shows longer execution times than Version-1, which could be attributed to overheads from additional memory management or less optimal memory access patterns in certain contexts. Version-3, which integrates shared memory, local variables, and streaming, provides the best performance in most cases, demonstrating the effectiveness of these advanced GPU features in reducing computational overhead and memory latency.
- **GPU Memory and Parallelism:** The use of constant and shared memory in Versions 2 and 3 significantly impacts performance by minimizing the latency of frequent memory accesses and maximizing the efficiency of data transfers within the GPU architecture. However, the effectiveness can vary depending on the complexity of decoding tasks at different alpha levels, affecting memory usage and computational demands.

TABLE I: Execution time(s) for different versions and alphas

α/Ver	C	CUDA 1	CUDA 1.5	CUDA 2	CUDA 3
0.06	35.528	2.146	3.187	2.257	1.861
0.05	44.619	2.896	2.944	2.780	2.249
0.04	53.901	5.669	3.912	3.937	3.031
0.03	98.797	15.965	10.725	9.626	6.817
0.02	240.589	53.329	28.827	27.133	24.983
SpeedUp	1.00	5.92	9.55	10.77	12.16

Table II highlights the performance advantages of using GPUs over CPUs for LDPC decoding.

- **Throughput and Execution Time:** The GPU outperforms the CPU significantly in terms of throughput (approximately 10 times higher) and execution time (more than 12 times faster). This discrepancy stems from the parallel processing capabilities of GPUs, which excel in handling the multiple independent calculations required for LDPC decoding.
- **Error Handling Capability:** Both platforms maintain the same level of error handling as indicated by NbUndec(Dmin) and other error-related metrics, suggesting that the optimizations do not compromise the decoding accuracy.
- **Efficiency Under Load:** The GPU maintains high efficiency even under increased computational loads, as indicated by consistent performance metrics across the number of tested frames and error rates.

TABLE II: Performance Comparison between CPU (Throughput = 4.09) and GPU (Throughput = 44.00) with Alpha = 0.06 and BlockDimension = 64

Device	Undec (Dmin)	IterAv (Max)	Tested	Fer (FER)	Er (BER)
CPU	0(100K)	95.04(78)	112	100(0.89)	29309(0.20)
GPU	0(100K)	95.04(78)	112	100(0.89)	29309(0.20)

VI. Conclusion

This study successfully demonstrated the potential of optimizing Gallager B LDPC decoders using GPU-accelerated architectures, yielding significant improvements in throughput and efficiency. By implementing various optimization strategies, such as leveraging constant memory and enhancing parallel processing capabilities, we achieved a substantial increase in decoding speed and a reduction in memory latency. The impact of these optimizations is evident in the increased performance metrics, making GPU-accelerated LDPC decoders viable for high-throughput applications that demand real-time processing.

Looking forward, the project could be expanded by integrating multi-codeword decoding algorithms, which allow multiple codewords to be processed simultaneously, further increasing throughput and reducing latency. Additionally, exploring adaptive decoding algorithms that dynamically adjust based on the noise level and other real-time feedback from communication channels could further optimize decoder performance under varying conditions. These advancements would provide a more robust and efficient decoding framework for modern communication systems.

References

[1] B. Unal, A. Akoglu, F. Ghaffari, and B. Vasić, "Hardware Implementation and Performance Analysis of Resource Efficient Probabilistic Hard Decision LDPC Decoders," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 3074-3084, Sept. 2018.

[2] G. Wang, M. Wu, Y. Sun and J. R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *IEEE 9th Symposium on Application Specific Processors (SASP), San Diego, CA, USA*, pp. 82-85, 2011.

[3] K. K. Abburi, "A Scalable LDPC Decoder on GPU," in *24th International Conference on VLSI Design, Chennai, India*, 2011.

[4] S. Keskin and T. Kocak, "GPU-Based Gigabit LDPC Decoder," in *IEEE Communications Letters*, vol. 21, no. 8, pp. 1703-1706, Aug. 2017.

[5] R. Amiri and H. Mehrpouyan, "Multi-Stream LDPC Decoder on GPU of Mobile Devices," in *IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA*, 2019.

[6] Z. Liu, R. Liu and L. Zhao, "GPU-based non-binary LDPC decoder with weighted bit-reliability based algorithm," in *China Communications*, vol. 17, no. 5, pp. 78-88, May 2020.