GPU Accelerated Decoding of High Performance Error Correcting Codes

Andrew D. Copeland, Nicholas B. Chang, and Stephen Leung
MIT Lincoln Laboratory, Lexington, MA 02420
{andrew.copeland, nchang, sleung}@ll.mit.edu*

Introduction

In this abstract we present a GPU-based implementation of a powerful class of error correcting codes known as low-density parity-check (LDPC) codes. LDPC codes can be used to achieve near-capacity performance in noisy channels. Unfortunately, this high performance comes with very high computational complexity. For a particular multiple input multiple output (MIMO) wireless communication system targeting a 1 GB/s data rate, a real time implementation would require custom hardware (ASIC or FPGA), while our optimized MATLAB-based version can take over nine days to decode a half second of recorded data. The real time solution is of course desireable, but it would take significant development to get such a device ready. Furthermore with a hardware-based decoder it is difficult to change the particular code and even more difficult to modify it to an alternate application with different parameters.

Using an algorithm ported to a single NVIDIA GTX 280 GPU, the decoding time is reduced from 9 days to 22 minutes and 32 seconds (a 582x speedup). This can be further reduced to 5 minutes and 37 seconds (a 2336x speedup) by breaking up the data across 4 GPU system containing two NVIDIA GTX 295 cards. The implementation is general and can be called from within MATLAB via a MEX function.

Algorithm Description

LDPC codes are block codes defined by a sparse parity-check matrix **H** which essentially represents a sparse bipartite graph connecting check and symbol nodes [1,2]. Binary LDPC codes were first shown to achieve remarkable performance [1]. It was later discovered [3] that coding performance could be enhanced by using LDPC codes defined over the Galois field $GF(q)$. An LDPC code over $GF(2^m)$ groups every m bits into elements of this field. As m increases, the performance of $GF(2^m)$ codes improves; however, this improvement comes at the expense of higher decoding complexity.

The LDPC decoder [4] is a belief propagation algorithm that iteratively estimates the posteriori probabilities $\{P(c_j = k|r_j)\}$, where c_j is the jth transmitted GF(q) symbol, k is an element of GF(q), and z_j is the receiver observation. Given a set of prior probabilities $\{P(c_j = k)\}$ on the symbols, each iteration computes probability vectors R_{ij} and Q_{ij}, defined below for all checks $\{i\}$ and symbols $\{j\}$. The length of R_{ij} or Q_{ij} is equal to the finite field size. Each decoding iteration of this message-passing algorithm involves two stages. The basic structure of the LDPC decoder is depicted in Figure 1.

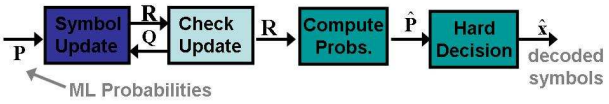

Figure 1: LDPC decoder basic structure

During the first stage, also known as the *symbol update*, the following probabilities are computed:

$$Q_{ij} = [P(c_j = 0|\{z_i\}_{i\in M(j)-i}), \cdots , P(c_j = q-1|\{z_i\}_{i\in M(j)-i})] \ , \quad (1)$$

where $M(j)$ denotes the set of check nodes connected to symbol j. The processing required to compute Q_{ij} is depicted in Figure 2.

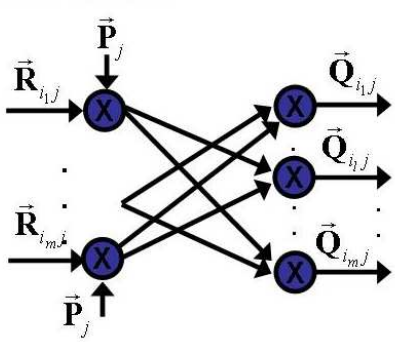

Figure 2: Symbol update stage of LDPC decoder. Computed for each symbol j and its connected checks $i_1 \dots i_m$.

The second stage, called the *check update*, computes the following probabilities:

$$R_{ij} = [P(z_j = 0|c_j = 0, \{c_n\}_{n\in N(i)-j}), \cdots , P(z_j = 0|c_j = q-1, \{c_n\}_{i\in N(i)-j})] \ , \quad (2)$$

where $N(i)$ denotes the set of symbol nodes connected to check i and z_j denotes the value of the jth syndrome. The processing required to compute R_{ij} is depicted in Figure 3.

After the check update, the probability vectors R_{ij} and the prior probabilities are used to compute the posteriori probabilities as given in Figure 4. The posteriori probabilities are used to make a hard decision on the transmitted codeword. This hard decision candidate codeword is then multiplied by the parity matrix to determine the syndromes; if the syndromes are all zero, then decoding halts. Otherwise, decoding continues iterating between symbol and check updates until the syndromes are all zero.

*This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government

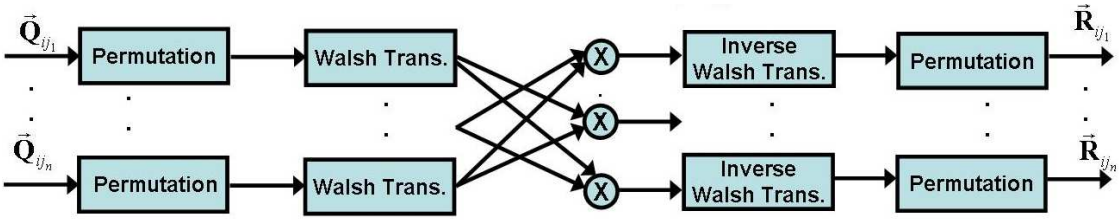

Figure 3: Check update stage of LDPC decoder. Computed for each check i and its connected symbols $j_1 \dots j_n$.

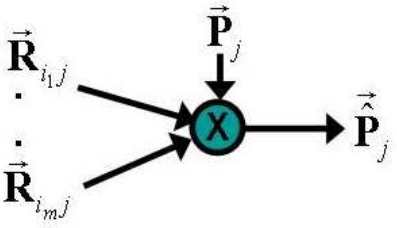

Figure 4: Computations of posteriori probabilities. Computed for each symbol j and its connected checks $i_1 \dots i_m$.

Parallelization

NVIDIA's CUDA programming language extends the C programming language to provide a convenient way to launch and schedule processing jobs from a single host CPU call known as a kernel. Each kernel consists of a grid that distributes the processing across the available multiprocessors and a thread block that coordinates the processing within each multiprocessor. Each thread block is run until completion, at which time the next thread block in the grid is executed. When the list of elements in the grid is exhausted the kernel returns to the host CPU [5]. On the GPU, threads can coordinate their effort with the other threads in a block but are unable to coordinate with threads in other blocks. Algorithms ported to the GPU, must similarly be decomposed into blocks that are independent of all the other blocks.

The most computationally intensive portion of the algorithm is the check update depicted in Figure 3. Because the computation for each check is independent of the computation for all other checks, we assign each check i to a separate block. Within a block the threads are assigned to one of the q possible values the symbol c_j can take. To setup for each of the permutations and the Walsh transforms, the probabilities are loaded into the multiprocessor's fast shared memory. Through shared memory, each thread can access each of the probabilities held by the other threads. The permutations proceed by each thread reading the probability it needs from shared memory. The Walsh transform proceeds through a simple multi-threaded radix-4 algorithm.

Because some of the computation is duplicated, we combine the symbol update, the computations of posteriori probabilities, and the hard decision. With this step, each symbol j is mapped to a particular block and every thread is assigned to one of the q possible values the symbol c_j can take. Here shared memory is used to normalize the probability of each symbol and to compute the index corresponding to the most likely value of c_j.

Results

The $GF(q)$ parity check matrix $\mathbf{H}$ used for benchmarking the algorithm is of size 4000x6000 and contains 12000 nonzero elements. The benchmark runs for 15 iterations before a valid codeword is determined. The time taken to compute a single codeword is 33.8 ms which corresponds to a 582X speedup versus the 19.68s MATLAB takes. The multi-GPU setup can decode 4 frames in 33.7 ms. The decoding of a single codeword takes 922 MFLOPS. The GPU implementation achieves 26.3 GFLOPS a second out of a possible 930 GFLOPS a second the GTX 280 is capable of during the decode process. The check update takes 78% of the computation time and achieves a throughput of 36.2 GFLOPS per second. The overall performance of the algorithm is limited by both memory accesses and integer operations that often take longer to compute than floating point operations.

The single GPU results were from a single NVIDIA GTX 280 on a system containing a 3.00GHz Intel Core 2 X9650 with 8GB of 800MHz DDR2 memory. The multi-GPU performance results used two NVIDIA GTX 295s on a system containing a 3.20GHz Intel Core i7 965 with 12GB of 1330MHz DDR3 memory.

References

[1] Gallager, R.G., Low-Density Parity-Check Codes, Massachusetts Institute of Technology ScD dissertation, 1960.

[2] Mackay, D.J.C, "Good error correcting codes based on very sparse matrices", IEEE Transactions on Information Theory, Volume 45, No 2, pp.399-431

[3] Davey, M.C. and Mackay D.J.C. "Low density parity check codes over GF(q)", IEEE Communications Letters, Volume 2, No. 6, pp.165-167

[4] Davey, M.C., Error Correction Using Low-Density Parity Check Codes, University of Cambridge PhD dissertation, 1999

[5] NVIDIA Corporation, "NVIDIA CUDA Programming Guide", Version 2.1, December 2008.