GPU Acceleration for Particle Filter based LDPC Decoding

Shuang Wang, Lijuan Cui, Samuel Cheng and Robert C. Huck
School of Electrical and Computer Engineering
University of Oklahoma, Tulsa, OK, 74135
Email: {shuangwang, lj.cui, samuel.cheng, rchuck}@ou.edu

Abstract—A parallel belief propagation algorithm based on Particle Filtering (PF) for channel estimation and Low-Density Parity-Check (LDPC) decoding is presented in this paper based on Compute Unified Device Architecture (CUDA). The authors have found that compared with the traditional Belief Propagation (BP) algorithm with fixed estimated noise power, BP algorithm based on PF [1] not only gives a good real-time estimate for the channel noise, but it also achieves a lower decoding error rate. However, the implementation of PF algorithm increased the decoding complexity. As a new hardware and software architecture for addressing and managing computations, CUDA offers parallel data computing using the highly multithreaded coprocessor driven by very high memory bandwidth GPU. The parallel noise adaptive decoding algorithm, based on CUDA, allows variable nodes, factor nodes or particles to be updated simultaneously, thus providing an efficient and fast way for implementing the decoder.

I. INTRODUCTION

As a type of error-correcting code, Low-Density Parity-Check (LDPC) codes were first proposed by Gallager in the early 1960s [2] and revived by Mackay and Neal in 1996 [3]. From that time, LDPC code has raised wide interest in the research community because the performance of LDPC code can make data transmission rates achieve near Shannon limit [3].

LDPC codes can be decoded by using a powerful iterative algorithm known as the Belief Propagation (BP) algorithm [3]. For LDPC decoding over the Additive White Gaussian Noise (AWGN) channel, the knowledge of the estimated noise power is one of the most important parameters to achieve the best performance of the BP algorithm [4]. However, for an unknown or a time-varying AWGN channel, it is very difficult to estimate the noise power without a pilot signal or feedback.

To overcome these problems, in [1], we propose a BP algorithm based on Particle Filtering (PF) for LDPC decoding over an AWGN channel. The proposed algorithm is carried out based on a factor graph [5], which affords great flexibility in modeling systems. We show that the proposed algorithm no longer depends on the initial estimation of noise power σ² and offers a good real-time estimation for channel noise power. For different code rates, our algorithm shows a lower decoding error rate than that of a standard BP algorithm. However, the implementation of the PF algorithm dramatically increased decoding complexity, especially when a large number of particles were used to estimate the channel noise power. A parallel decoding method, by taking advantage of GPU architectures, offers an efficient solution to accelerate this procedure.

In just a few years, GPUs have evolved into flexible platforms for general computing [6]. Initially, GPUs were programmed by low-level languages [7] which restricted its application as computing workhorses. The release of Cg, a high-level programming language for the GPU, facilitated the application of a GPU for a general purpose computation [8]. However, Cg is not user-friendly enough, because it required programmers to have fundamental knowledge of computer graphics for using this high-level programming language. Now that NVIDIA has released the CUDA [6], programmers can write codes for both CPUs and GPUs in a similar way by using the instruction set of CUDA.

In many related fields [9]–[11], CUDA has been verified as a compute-intensive and highly parallel computing workhorse. In [9], a fast 3D tracking of multiple faces using a particle filter based on CUDA was introduced. An important speed boost was observed, especially when a large number of particles are used, which makes the tracker eminently more suitable for real-time processing in a standard PC platform. In [10], an efficient CUDA-based implementation of the BP algorithm was described that sped up stereo image processing and motion tracking calculations. In [11], we presented that CUDA offers a highly parallel architecture and a significant increase in performance compared with the computation on a CPU for LDPC decoding based on the BP algorithm.

In this paper, we propose a parallel BP algorithm based on PF for noise adaptive LDPC decoding by using the CUDA programming model. We show how the parallelism naturally appeared during message-passing and that particle updating can be exploited using the CUDA model, which allows particles, variable nodes, or factor nodes to be updated simultaneously. Thus, a significant increase of performance is observed when the number of particles and the size of the parity check matrix are reasonably large.

This paper is structured as follows. In Section II, we explain the BP algorithm based on PF in factor graph. Then, the corresponding parallel algorithms are described in Section III. Finally, in Section IV we present simulation results and in Section V we draw the concluding remarks.
II. BELIEF PROPAGATION ALGORITHM BASED ON PARTICLE FILTER

In this section, we introduced a novel model using the BP algorithm based on particle filtering in the factor graph for LDPC decoding, see Fig. 1. In region 1, a BP algorithm based on PF is used to estimate the real time noise power, while the standard BP algorithm is used in region 2 and region 3. In the following section, we will explain our algorithm in detail.

A. Particle Filter

Particle filters, also known as Sequential Monte Carlo methods, are used to estimate posterior probability distribution of an interest object by sampling a list of random particles with associated weights [12]. In the following, we explain how we apply PF in our model to estimate noise power $\sigma^2$ sequentially over an AWGN channel. In our model, each variable node in region 1 is assigned a PF which follows the same rule.

1) First we initialize the list of particles from the prior distribution. For each variable node $i$ in region 1, each particle $k$ is assumed to have a value $\sigma^2_k = \tilde{\sigma}$ for estimating the noise and a uniform weight $\frac{1}{N_p}$, where $N_p$ is the number of the particles.

2) Then we compute the importance weights $\omega_i^{(k)}$ at each iteration. In our model this weight is equal to the belief of each particle, where the belief is obtained by the BP algorithm, which will be presented in section II-B.

3) Since the variance of the importance weights increase stochastically over time, a selection (resampling) stage is needed to eliminate particles with negligible weights and to concentrate on particles with large weights. The future particles in domains of higher posterior probability entail improved estimates. Here we use the Systematic Resampling (SR) [12], [13] algorithm. SR first calculates the cumulative sum of the particle weights $C^{(k)}(i) = \sum_{i=1}^{k} \omega_i^{(i)}$ and updated uniform number $U^{(k)} = U^{(k-1)} + \frac{1}{N_p}$, $k = 1, ..., N_p$, where $U^{(0)}$ is obtained by drawing from the uniform distribution $u[0, \frac{1}{N_p}]$. Then SR compares $C^{(k)}$ and $U^{(k)}$ to determine the number of replications for particle $k$ by computing the number of time $U^{(k)}$ in the range $[C^{(k-1)}, C^{(k)}]$.

4) Next we use the Random Walk (RW) Algorithm. After the resampling step, particles congregate round the values with large weights. In order to perform the further estimation, particles should be dispersed around values with large weights. RW is implemented by adding a Gaussian random variable $N(0, \sigma^2)$ with zero mean and variance $\sigma^2$ on the current value $\sigma^2_i$ of each new particle generated in step 3.

5) Then we update the weight by resetting to a uniform weight $\frac{1}{N_p}$ for each particle.

6) Now we iterate steps 2 to 5 when updating each variable node.

B. Belief Propagation Algorithm Based On Particle Filter in Factor graph

A factor graph [5] is a bipartite graph that expresses the factorization structure of a function. The factor graph of our novel model, shown in Fig. 1, contains 3 different regions. In these regions, all square nodes are factor nodes which are indexed by $\{f_A, f_B, ..., f_M\}$, $\{f_1, f_2, ..., f_N\}$, $\{f_{1,2}, f_{3,4}, ..., f_{N-1,N}\}$, and all circle nodes are variable nodes which are indexed by $\{x_1, x_2, ..., x_N\}$, $\{\sigma_1, \sigma_2, ..., \sigma_N\}$.

For the belief propagation based on factor graph, let $m_{n-i}(x_i)$ denote the message sent from factor node $a$ to variable node $i$, and let $m_{i-a}(x_i)$ denote the message sent from variable node $i$ to factor node $a$. In this paper, the proportionality symbol $\propto$ indicates that these messages must be normalized so that they sum to one. Since we considered the AWGN channel, we assumed that the initial value of the message sent from the factor nodes in region 2 to variable nodes in region 3 following (1)

$$m_{a-i}(x_i) \propto f(x_i) = \frac{1}{\sqrt{2\pi\tilde{\sigma}^2}}e^{-\frac{(y_i-x_i)^2}{2\tilde{\sigma}^2}}$$

where $\tilde{\sigma}$ is a initial estimated value for $\sigma$, while all other initial values of messages are equal to 1. First, the messages sent from variable nodes were updated. Then we performed PF. Finally, the messages sent from factor nodes were updated.

The message update rules are detailed as follows:

1) Update variable nodes in region 3 following (2)

$$m_{i-a}(x_i) \propto \prod_{c \in N(i) \setminus a} m_{c-i}(x_i)$$

2) In the PF algorithm, the messages are represented by $N_p$ particles. Hence, updating variable nodes in region 1 needs to update information for each particle. Using the updating equation (3)

$$m_{i-a}(\sigma^2_i) \propto \prod_{c \in N(i) \setminus a} m_{c-i}(\sigma^2_c)$$

where $N(i) \setminus a$ denotes the set of neighbors of node $i$ except for node $a$; $k$ denotes the $k$-th particle and
$k \in [1, N_p]$, $\sigma^k_i$ is the value of particle $k$ for variable node $i$.

3) Compute the belief of each variable node $i$ in region 3 being $x_i$

$$b_i(x_i) \propto \prod_{a \in N(i)} m_{a \rightarrow i}(x_i) \quad (4)$$

where $x_i$ is equal to $-1$ or $1$.

4) Compute the belief of each particle for each variable node $i$ in region 1 being $\sigma^k_i$

$$b(\sigma^k_i) \propto \prod_{a \in N(i)} m_{a \rightarrow i}(\sigma^k_i) \quad (5)$$

In the model, the belief $b(\sigma^k_i)$ of each particle $k$ is corresponding to the weight ($\omega^k_i$) described in the second step of PF.

5) Update factor node in region 3 follows (6)

$$m_{a \rightarrow i}(x_i) \propto \sum_{x_a \setminus x_i} \left( f_a(x_a) \prod_{j \in N(a) \setminus i} m_{j \rightarrow a}(x_j) \right), \quad (6)$$

where $f_a(x_a) = \begin{cases} 1 & \text{if even number of 1's in arguments} \\ 0 & \text{otherwise} \end{cases}$

6) Update factor node in region 1, which means to update each particle according to (7)

$$m_{a \rightarrow i}(\sigma^k_i) \propto \sum_{\sigma_{i+1}} \left( f_a(\sigma^k_{i+1}, \sigma^k_i) m_{i+1 \rightarrow a}(\sigma^k_{i+1}) \right),$$

$$m_{a \rightarrow i+1}(\sigma^k_{i+1}) \propto \sum_{\sigma_i} \left( f_a(\sigma^k_{i+1}, \sigma^k_i) m_{i \rightarrow a}(\sigma^k_i) \right), \quad (7)$$

where $f_a(\sigma^k_{i+1}, \sigma^k_i) = e^{-(\sigma^k_{i+1} - \sigma^k_i)^2 / x}$ and $\sigma_i$ or $\sigma_{i+1}$ means all the particles in the variable nodes $i$ or $i+1$ in region 1. In region 1, it is assumed that each factor node only connects two variable nodes $\sigma_i$ and $\sigma_{i+1}$, $i \in [1, N-1]$, see Fig. 1.

7) Update factor node in region 2.

Message from region 2 to region 1 follows (8)

$$m_{a \rightarrow i}(\sigma^k_i) \propto \sum_{x_j \in [0,1]} \left( f_a(x_j, \sigma^k_i) m_{j \rightarrow a}(x_j) \right) \quad (8)$$

where $f_a(x_j, \sigma^k_i) = \frac{1}{\sigma^k_i} e^{-\frac{(y_j - x_j)^2}{2}}$, $j$ means the $j$-th variable node in region 3 and $y_i$ is the input codeword.

Message from region 2 to region 3 follows (9)

$$m_{a \rightarrow i}(x_i) \propto \sum_{k \in [1, N_p]} \left( f_a(x_i, \sigma^k_j) m_{j \rightarrow a}(\sigma^k_j) \right), \quad (9)$$

where $f_a(x_i, \sigma^k_j) = \frac{1}{\sigma^k_j} e^{-\frac{(y_i - x_i)^2}{2}}$, and $j$ means the $j$-th variable node in region 1 and $y_i$ is the input codeword.

## III. CUDA Implementation Of Parallel Belief Propagation Algorithm Using Particle Filter

### A. General Framework of CUDA

CUDA is a new hardware and software architecture for parallel computing on the nVidia GPU. In CUDA, the GPU operates as a coprocessor to the main CPU (host), which means data-parallel, computation-intensive portions of applications running on the host are off-loaded onto the GPU (device). Both the host and device maintain their own DRAM, referred to as host-memory and device memory, respectively. In order to execute a kernel on a device, one needs to allocate memory on the device and then copy the pertinent data from the host memory to the allocated device memory. Also, the resultant data on the device needs to be transferred back to host memory and the memory that is no longer in use needs to be freed up.

In CUDA, the device memory can be categorized into read-write (R/W) per-thread registers, R/W per-block shared memory, R/W per-grid global memory, read-only per-grid constant memory, and texture memory. Registers, shared memory (organized in 16 banks), and constant memory (cached) can be accessed at higher speed than the global memory. Thus these memories can be used to reduce the number of accesses to the global memory. However, the limited size of these special memories restricts the number of thread that can simultaneously reside in the Stream Multiprocess (SM). The more special memories each thread requires, the fewer the number of threads that can be invoked in each SM, which may result in performance degradation if there are insufficient active threads in SMs [6]. Moreover, if two addresses of a shared memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized, which decreases the effective bandwidth by a factor equal to number of bank conflicts. Thus the request of shared memory needs to be scheduled efficiently to get the maximum performance.

The execution of a CUDA program starts with host execution. The batch of threads that executes a kernel is moved to a device, which is organized as a grid of thread blocks. A
block of threads can cooperate by sharing data through the fast shared memory and synchronizing executions efficiently. Each thread is identified by its thread ID in each block, and each block is identified by its block ID in each grid. The thread ID is arranged sequentially. A grid of thread blocks is executed on the device by scheduling blocks for execution on the SMs. Once a block is assigned to an SM, it is further divided into 32-threads units called Warps, which are the unit of thread scheduling in SMs. The maximum number of threads per block is 512 and the warp size is 32 threads. Moreover, each SM can accommodate up to 8 active blocks, 24 active warps or 768 active threads, which become the restrictions. Thus programmers need to organize their kernel call well to reach the maximum performance. A kernel will be terminated when all threads in this kernel have finished their execution. Then the execution of program continues on the host until the next kernel is invoked.

B. Parallel Belief Propagation using Particle Filter on CUDA

In our CUDA implementation of the parallel BP algorithm using PF, a thread is assigned to a variable node, a factor node, or a particle. The workflow of the algorithm is illustrated in Fig. 2 and is summarized as follows:

1) Allocate and initialize memory in global memory. Then copy the data required for GPU computation from the host memory in the CPU to global memory in the GPU, so that all threads can access the data in the global memory.

2) The parallel update of variable nodes or factor nodes in region 3 is performed by assigning a thread for each node (see Fig. 3). The block size in the actual execution is 128, so that enough active threads (768 threads per SM) can be issued to reach the maximum performance. Here, the products of the incoming message in (1) and (6) are performed on the 1D shared memory arrays to reduce memory latency.

3) For parallel update of variable nodes in region 1, we will invoke a kernel function on a grid of thread blocks with size $N \times N_p$ to compute all particles in all variable nodes in region 1, see Fig. 4. Because there are $B \times B$ threads per block, the number of thread blocks needed to update all variable nodes and particles is $\frac{N}{B} \times \frac{N_p}{B}$. The four small figures on the right hand side of Fig. 4, show the sequential processing of the particle filter algorithm for each particle. Thus, in our CUDA model, each thread is in charge of a calculation of the particle filter algorithm for a given particle and a variable node. In our actual execution, $B$ is equal to 16 (for total 256 threads per block) to satisfy the performance restriction described in section III-A. For the update of factor nodes in region 1, we use the same strategy, except keeping the $N$-th thread and its corresponding $N_p$ sub-threads in an idle status, since there are only $N - 1$ factor nodes in region 1. Here, 2D shared memory arrays with size $B \times (B + 1)$ are used to accelerate memory access, where we add one float to the end of each row (using $B \times (B+1)$ instead of $B \times B$) to avoid bank conflicts [6]. Due to the page limitation, we only present an example to explain how the factor nodes are updated in region 1. Suppose we use 16 particles and the block size $B$ is equal to 16 too. Firstly, we allocate two 2D shared memory arrays with size $B \times (B+1)$, named “OutMsg”, “InMsg” and allocate one 2D shared memory array with size $(B+1) \times (B+1)$, named “pArray”. According to (7), we update the message $m_{\sigma_i}^j$ first. All the incoming messages $m_{i+1-a} \sigma_i^k$ are load into shared memory “InMsg” and all the particle values are loaded into shared memory “pArray”. Then the sum-product results, obtained by (7), are stored in “OutMsg”. Next, we copy the data of “OutMsg” into “InMsg”, since we no longer need “InMsg”. Then we perform reduction on “InMsg” to calculate the normalization base. Here “InMsg” is reused to reduce the usage of shared memory. Finally, we can obtain the normalized “OutMsg” $\frac{“OutMsg”}{“InMsg”}$. For the update of the message $m_{\sigma_i^k}$ in (7), we use the same way described above.

4) The update of factor nodes in region 2 is separated into two parts. For the update of message from region 2 to region 1, we use the parallel strategy as step 3), while for the update of message from region 2 to region 3, we perform the parallel strategy as step 2).

5) The updates of beliefs for each variable node in region
and $N$ same, however, a big gap existed for the run time of different number of particles, GPU run times were almost the same as the CPU run time. For the number of factor nodes was equal to 1024. But the GPU run time was drastically shorter than the CPU run time. For the simulation. Moreover the numbers of particles ranging from 8 to 256 were used during the experiment. The number of check nodes in the following experiments varied from 128 to 8192. Then, over the same AWGN channel, as shown in Fig. 5 (left). Here, 16 particles were used in each variable node. Moreover, LDPC codes were randomly generated by a parity check matrix with 10,240 variable nodes and 7,008 check nodes. The result verified that our proposed model can generate a good estimation of a complex time-varying noise power. Then, over the same AWGN channel, the decoding performance (in terms of bit error rate (BER)) of standard BP and that of PF based BP algorithm were also compared in Fig. 5 (right). Even though we used a codeword with a code rate of 0.1, the standard BP algorithm still could not decode any codeword without errors. However, our new algorithm showed good performance in decoding codewords over this kind of AWGN channel.

Next we investigated the performance of a CUDA based parallel algorithm. The number of check nodes in the following experiments varied from 128 to 8192, where the number of variable nodes for simulation was always 2 times bigger than the corresponding number of factor nodes. Moreover the numbers of particles ranging from 8 to 256 were used during the simulation.

From Fig. 6, we can see that the run time for CPU or GPU increased as the number of factor nodes increased, except that the number of factor nodes was equal to 1024. But the GPU run time was drastically shorter than the CPU run time. For the different number of particles, GPU run times were almost the same, however, a big gap existed for the run time of $N_p = 16$ and $N_p = 32$ on the CPU. Fig. 7 showed how the performance varied as the number of factor nodes changed, when given $N_p = 16$ and $N_p = 32$. It appeared that a larger number of particles and factor nodes tend to result in better performance (in terms of GFLOP/S). Moreover, a performance step was also observed when the number of factor nodes was equal to 1024. The above results indicated that sufficient numbers of threads were required to achieve a good performance. Fig. 8 showed how the performance varies as the number of particles...
For the noise adaptive decoding of LDPC codes, CUDA offers a highly parallel architecture and significant increase of performance in contrast with traditional computation on a CPU. With CUDA, we do not need specific hardware design knowledge to accomplish parallel channel estimation and decoding. Finally, we conclude that GPU based parallel programming is a very efficient way for the intensive noise adaptive LDPC decoding.

**REFERENCES**


