EURASIP Journal on Wireless Communications and Networking
a SpringerOpen Journal

RESEARCH Open Access

Scheduling parity checks for increased throughput in early-termination, layered decoding of QC-LDPC codes on a stream processor

JaWone A Kennedy and Daniel L Noneaker*

Abstract

A stream processor is a power-efficient, high-level-language programmable option for embedded applications that are computation intensive and admit high levels of data parallelism. Many signal-processing algorithms for communications are well matched to stream-processor architectures, including partially parallel implementations of layered decoding algorithms such as the turbo-decoding message-passing (TDMP) algorithm. Communication among clusters of functional units in the stream processor impose a latency cost during both the message-passing phase and the parity-check phase of the TDMP algorithm with early termination; the inter-cluster communications latency is a significant factor in limiting the throughput of the decoder. We consider two modifications of the schedule for the TDMP algorithm with early termination; each halves the communication required between functional-unit clusters of the stream processor in each iteration. We show that these can provide a substantial increase in the information throughput of the decoder without increasing the probability of error.

Keywords: layered decoding, turbo-decoding message passing (TDMP), stream processor, graphical processing units (GPU)

1 Introduction

Quasi-cyclic (QC) low-density parity-check (LDPC) codes [1] based on circulant permutation submatrices are used for forward error correction in a wide variety of wireless communication systems employing battery-powered devices, including WiMAX (802.16) [2] and Wi-Fi (802.11 n) [3] networks. The embedded processors in the devices must satisfy stringent limits on their power consumption, yet they must also exploit the inherent parallelism in the belief-propagation decoding algorithms [1] used with LDPC codes in order to achieve a high decoder throughput. The twin performance objectives typically result in hardware designs using application specific integrated circuits or field-programmable gate arrays instead of general-purpose digital signal processors (DSPs).

An alternative is the use of a specialized DSP that is optimized for a high level of single-instruction, multiple-data (SIMD) parallelism. Such a processor can provide the high-level-language programmability of a general-purpose DSP but with a much greater computational throughput relative to power dissipation for algorithms that admit high levels of data parallelism. An example of a SIMD-centric DSP is the graphical processing unit (GPU). GPUs are powerful, multi-core processors that provide a combination of task parallelism, thread parallelism, and data parallelism which can be used for high-throughput decoding of LDPC codes [4,5]. GPUs exhibit high power consumption as a result of floating-point processing and support of highly multithreaded task-level parallelism (including the consequent structure of the memory hierarchy [6]), however; thus a GPU is impractical for use in a battery-powered mobile communication device or an unattended sensor node.

An emerging alternative for computationally demanding digital signal processing is a SIMD-optimized DSP designed for embedded systems, exemplified by the *stream processor* (or on-chip stream co-processor) [7]. A stream processor is a high-data-width SIMD architecture with a software-managed, cacheless memory hierarchy that adheres to the stream-processing computing paradigm [7];

* Correspondence: dnoneak@clemson.edu
Holcombe Department of Electrical and Computer Engineering, 305 Fluor Daniel Engineering Innovation Building, Clemson University, Clemson, SC 29634, USA

it is designed for applications that exhibit compute intensity, data parallelism, and producer-consumer locality [6]. A stream processor provides a more efficient trade-off between SIMD parallelism and power consumption than a GPU by supporting only fixed-point arithmetic for single-task, single-threaded execution, emphasizing greater local data-memory density around the functional units, and by omitting some specialized functions found in the GPU. The fixed-point processor may provide two or more data-resolution modes operating on packed-data operands so that lower-resolution arithmetic operations can be used to achieve greater computational throughput.

The stream-processor architecture is well-suited to decoding a QC-LDPC code using the turbo-decoding message-passing (TDMP) algorithm. The TDMP algorithm [8,9] is a rowwise, layered-decoding belief-propagation algorithm [10] that is readily adapted to differing levels of partially parallel computation based on the level of parallelism available in the processor. It achieves performance comparable to the sum-product algorithm (SPA) with about one-half the average number of iterations. The TDMP algorithm requires less memory than the SPA [8], and it permits simpler data management than the SPA in processor architectures organized for high levels of SIMD parallelism [11]. Previous research has addressed LDPC decoding with the SPA [1] using floating-point arithmetic on a GPU [4,5]. Low-power, fixed-point stream processors present different decoder design tradeoffs than GPUs, however, and the TDMP algorithm entails different design considerations than the SPA.

In this article, we investigate alternatives for early-termination decoding of QC-LDPC codes on a stream processor using a form of the TDMP algorithm which achieves good performance with low-resolution, fixed-point arithmetic. (Early termination increases the throughput of the decoder by allowing it to exit the decoding algorithm prior to the maximum number of allowed iterations if the decoded word passes all parity checks.) We consider two algorithms in which the posterior updates and parity checks are integrated for each subset of the check nodes processed in parallel, in contrast with the standard TDMP schedule in which all parity checks for an iteration are performed after all the updates. The decoding algorithms reduce by half the data communications required between the stream processor's functional-unit clusters for each iteration of the decoder using the standard schedule. Termination rules which guarantee a valid decoded word are also specified for each integrated update-and-parity-check decoding algorithm. The probability of error and the throughput achieved with each decoding algorithm is evaluated. It is shown that properly designed integration of the update and parity-check steps results in significantly higher decoder throughput than the standard schedule of the TDMP algorithm without an increase in the decoder's probability of error.

2 The constrained-update offset-min-sum TDMP algorithm

In this section, we describe the form of the TDMP algorithm considered as the benchmark decoding algorithm against which other variants are compared in the article. We consider the algorithm in a system using binary antipodal modulation with transmission over an additive white Gaussian noise channel with double-sided power spectral density $N_0/2$. Coherent demodulation with perfect synchronization is assumed. The sampled channel outputs are assumed to be normalized in amplitude prior to decoding so that the normalized channel outputs are given by

$$r_i = \pm 1 + n_i$$

where $\{n_i\}$ are i.i.d., zero-mean Gaussian random variables with variance $\sigma^2 = N_0/(2E_s)$ and E_s is the received energy per channel symbol. (The corresponding energy per bit of information is denoted by E_b.)

Each iteration of the TDMP algorithm is applied to an $M \times N$ parity-check matrix $\mathbf{H}$ of an $(N, N - M)$ LDPC code, with updates of posterior values for variable nodes (each corresponding to a code symbol) performed in a block-sequential manner. Concurrent updates use message passing based on the parity-check equations corresponding to a blocks of rows of $\mathbf{H}$ [9]. Extrinsic messages that are generated from decoding earlier row blocks are used as input prior messages for updates of the posterior values for variables nodes participating in later row blocks. The row-wise update schedule thus differs from the SPA in which all check-node updates occur before the updates of the posterior values are performed for any variable nodes in an iteration.

We consider application of the offset-min-sum approximation [12] to the TDMP algorithm, which yields floating-point decoder performance comparable to the original TDMP algorithm with substantially lower decoding complexity. The use of 8-bit arithmetic can provide up to twice the computational throughput of 16-bit arithmetic in SIMD architectures allowing multiple packed-data arithmetic modes. The use of 8-bit, fixed-point saturating arithmetic can result in a dramatic increase in the probability of error in offset-min-sum TDMP decoding compared with floating-point processing or 16-bit fixed-point processing, however, due to frequent saturation of posterior values in the algorithm if the maximum number of decoding iterations is large [11]. (The offset-min-sum variant of the SPA is much less sensitive to the fixed-point resolution, as seen in [13], for example.) The use of a properly chosen constraint on the magnitude of any

extrinsic update in the algorithm mitigates the effect of saturation, however, and results in performance of the TDMP algorithm with 8-bit processing comparable to the performance obtained with 16-bit processing [11]. Thus we consider the constrained-update, offset-min-sum variant of the TDMP algorithm in this investigation. In the remainder of the article, it is referred to simply as the "TDMP algorithm".

In the following description of the TDMP algorithm, the vector $\underline{\boldsymbol{\lambda}}^i = [\lambda_1^i, ..., \lambda_{c_i}^i]$ represents the extrinsic messages that correspond to the nonzero entries in row i of $\mathbf{H}$, where c_i represents the row weight of row i. The notation I_i denotes a list of the column positions of non-zero entries in row i in $\mathbf{H}$. The vector $\underline{\gamma} = [\gamma_1, ..., \gamma_N]$ represents the N posterior values, one for each code symbol v_i. The subset of the posterior messages corresponding to the non-zero column positions of row i are denoted $\underline{\gamma}(I_i)$.

The algorithm is implemented as follows:

1. Initialize $\underline{\boldsymbol{\lambda}}^i = \underline{\mathbf{0}}$, for $i = 1, ..., M$. Also, initialize the posterior values $\underline{\gamma} = [r_i, ..., r_n]$, where r_i is the real-valued channel output for v_i. Each entry in $\underline{\gamma}$ is applied to a uniform quantizer with quantization interval Δ (and clipping) for subsequent fixed-point processing using a signed, 8-bit representation.

2. Read the extrinsic messages $\underline{\boldsymbol{\lambda}}^i$ and the posterior values $\underline{\gamma}(I_i)$ for row i.

3. Subtract $\underline{\boldsymbol{\lambda}}^i$ from $\underline{\gamma}(I_i)$ to generate prior messages $\underline{\boldsymbol{\rho}} = [\rho_1, ..., \rho_{c_i}] = \underline{\gamma}(I_i) - \underline{\boldsymbol{\lambda}}^i$.

4. Decode the parity-check equation for row i. Define $\underline{\boldsymbol{\alpha}} = [\alpha_1, ..., \alpha_{c_i}]$ and $\underline{\boldsymbol{\beta}} = [\beta_1, ..., \beta_{c_i}]$, where $\alpha_j = \text{sgn}[\rho_j]$ (the sign of ρ_j) and $\beta_j = |\rho_j|$. Set

$$\lambda_j^i = \left(\prod_{k=1, k \neq j}^{c_i} \alpha_k \right) \cdot \max \left(\min_{1 \leq k \leq c_i, k \neq j} \beta_k - \eta, 0 \right)$$

for $j = 1, ..., c_i$, where the *offset* η is a non-negative constant that is an integer multiple of the quantization interval. (The values of Δ and η are chosen jointly to minimize the error probability at some operating point of interest.)

5. Limit the maximum extrinsic updates in Step 4 to

$$\lambda_j^i = \text{sgn} \left[\lambda_j^i \right] \cdot \min \left(\left| \lambda_j^i \right|, \varepsilon \right)$$

where ϵ is a predetermined constant that is used to limit saturation in the posteriors. It is also an integer multiple of the quantization interval.

6. Update the posterior values for the code-symbol positions of I_i as $\underline{\gamma}(I_i) = \underline{\boldsymbol{\rho}} + \underline{\boldsymbol{\lambda}}^i$.

7. Steps 2-6 represent a decoding subiteration for one row of $\mathbf{H}$. Repeat the steps for each row of $\mathbf{H}$.

For *architecture-aware* LDPC codes [8] (such as QC-LDPC codes based on circulant permutation submatrices), large sets of consecutive rows of $\mathbf{H}$ correspond to checks nodes with disjoint sets of variable nodes. Steps 2-6 can be executed in parallel for any such set of rows without altering the logic of the algorithm.

Steps 2-7 represent the *message-passing phase* of one iteration of the TDMP algorithm. It is followed by the *parity-check phase* in which a hard decision is made on each code symbol v_i based upon the sign of γ_i and each independent parity check is tested. If all the parity checks are satisfied, decoding is terminated with a valid decoded code word, and the information bits are recovered by inverse mapping. If not, but the maximum number of iterations has been executed, a known decoding failure occurs. Otherwise, another iteration of the algorithm is performed. We use this schedule of updates and parity checks for the TDMP algorithm (with alternating message-passing and parity-check phases) as a benchmark in the article and refer to it as the TDMP algorithm with the *standard schedule*.

3 An example of a stream processor

The Storm-1 system-on-a-chip by Stream Processors, Inc., is used as the example of a stream processor in the numerical results. The architecture of the chip (which is no longer in production) is summarized below. (Details are given in [14] and a diagram is shown in Figure 1.) Higher absolute throughput may be obtained with stream processors fabricated in newer process technologies using more recent architectural innovations for low-power, embedded processors with high SIMD parallelism [15-17]. The Storm-1 processor serves as a suitable platform for comparing the relative throughput obtained on a stream processor with various modifications of the TDMP decoding algorithm, however.

The Storm-1 system includes two general-purpose processors and a separate data-processing unit (DPU) with a SIMD architecture. The DPU contains 80 ALUs organized as 16 data-parallel 5-ALU functional-unit clusters referred to as *lanes* that are controlled by a very long instruction word in a Harvard architecture. The Storm-1 processor contains a three-level, non-caching memory hierarchy. A large off-chip memory is accessible to the DPU. Lane register files are high-bandwidth, per-lane, on-chip memory used to stage data for processing by the ALUs. Operand register files within each lane's ALU cluster, addressable only within the cluster, serve as local registers. Data can also be exchanged directly between ALU clusters using an inter-lane cross-bar switch. The accompanying compiler allows an application programmer to exploit available data parallelism and instruction parallelism without the need to explicitly

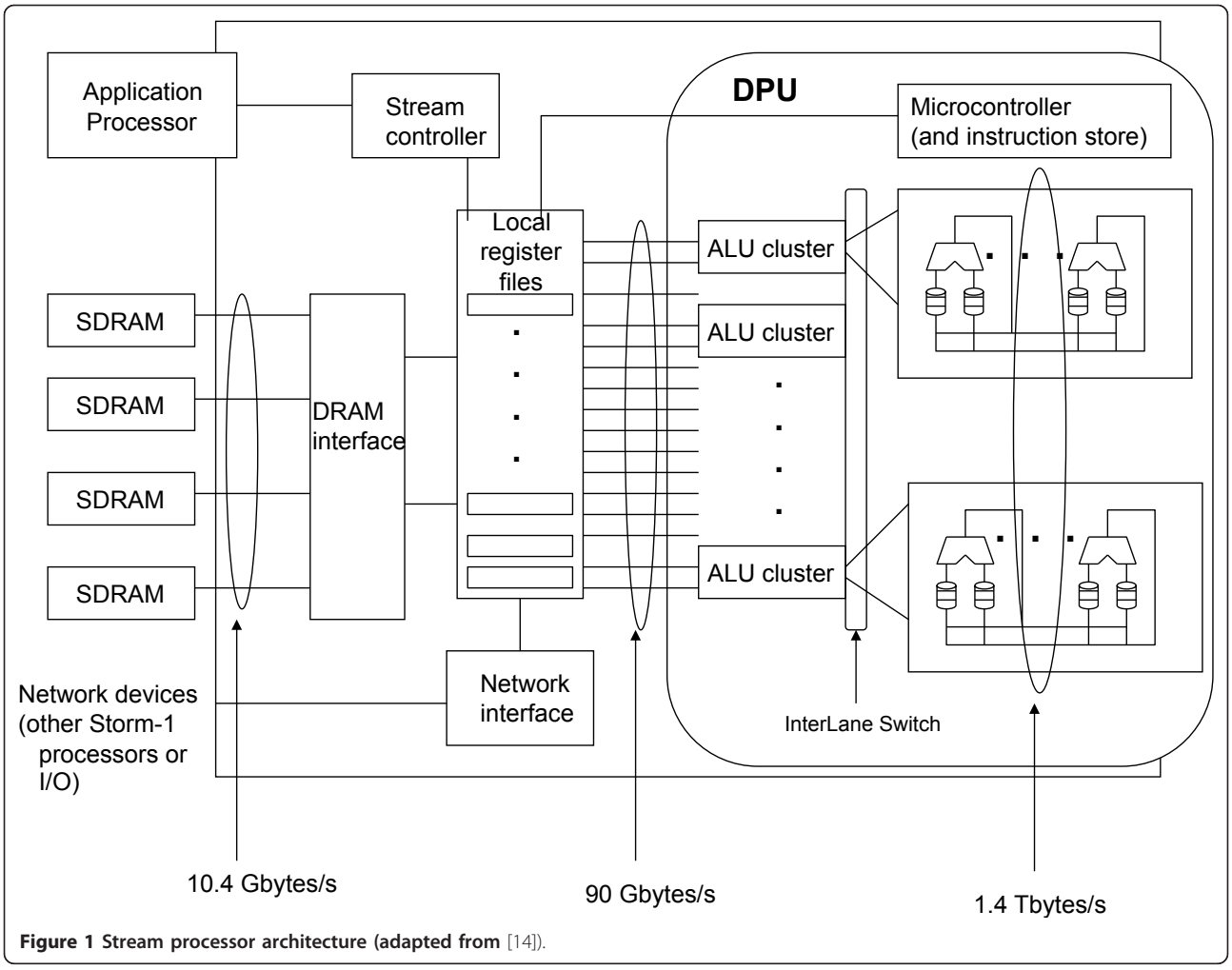

Figure 1 Stream processor architecture (adapted from [14]).

manage the resources of the lanes and the associated memory.

The DPU is designed for the flow of similarly formatted records of a large data set which form a *stream*. The stream is processed by one or more *kernel functions*, each of which is a compute-intensive inner loop that applies parallel processing to a stream that is resident in on-chip memory. The computations in the parallel-processing units are thus restricted to records in the stream as atomic data units and kernel functions as atomic instructions. Control tasks and computations that do not fit well within this stream-processing paradigm are assigned by the compiler to a general-purpose coprocessor.

Analysis by the compiler establishes the stream allocation and run-time data transfers into on-chip memory for kernels and streams based on dependencies associated with execution. Parallel processing occurs within kernels only; consequently, single structured instruction flow is preserved. Kernels are managed by the compiler, and they can form pipelines which share intermediate stream results. Data reference by a kernel is limited to the data records it is processing and other locally retained constants and variables. The compile-time analysis imposes a tight control on the use of the memory hierarchy, hiding the access latency to external memory for many tasks.

The simplicity of the stream-processing application programming model is achieved at the cost of restrictions in the computation model and strict management of the memory hierarchy. The resources of the parallel-processing architecture are utilized most efficiently if the application has characteristics consistent with these constraints. Specifically, the application should exhibit compute intensity, data-parallelism, and data locality [6]. Many signal-processing algorithms exhibit these characteristics.

The Storm-1 processor provides 8-bit packed-data and 16-bit packed-data modes of saturating fixed-point arithmetic using 32-bit data registers. The production

device was fabricated in a 130 nm process. It employs a nominal DPU clock rate of 800 MHz, resulting in an aggregate computation rate of 512 8-bit fixed-point GOPS (billion operations per second) or 256 16-bit fixed-point GOPS [14]. An evaluation board which operates at a lower clock rate is used (along with an emulator) to obtain the decoder performance measurements discussed in subsequent sections, but the results are given for the clock rate of the production device. The programming language used for application development for the Storm-1 processor is based on ANSI "C" with enhancements that define the kernel functions and streams as well as several compiler directives (pragmas).

4 Implementing the offset-min-sum TDMP algorithm on a stream processor

The TDMP algorithm is well suited to implementation on the stream processor if it is used with LDPC codes in which large groups of consecutive row updates can be performed in parallel without data conflicts. For example, a QC-LDPC code with circulant permutation submatrices permits a degree of parallelism at least as great as the row dimension of the submatrices. Each block of parallel row updates forms one subiteration of a decoder iteration. In this circumstance, the algorithm is characterized by a high level of available data parallelism and a high level of data locality. The compute intensity is only moderate, but data exchanges are limited to the high-speed inter-lane crossbar switch.

A single compiler directive for the Storm-1 processor can specify up to 64 concurrent data-parallel computations (four per lane) by using its 8-bit packed-data mode. This allows simultaneous processing of four check nodes per lane which yields a total of 64 concurrent row updates across the 16 lanes of the processor. Thus the row-update data parallelism available with the compiler and the architecture in 8-bit mode is fully exploited if the dimension of the permutation submatrices is an integer multiple of 64.

The five ALU's per lane also provide the compiler with the opportunity for instruction-level parallelism in the calculations associated with each row update, including parallel updates of as many as five posterior values for each row in Steps 2-6 of the algorithm. Since most of the QC-LDPC codes of interest have a parity-check matrix with a weight of five or more for most rows, an assignment of four concurrent row updates to each lane should result in a high utilization of the processor's 320 ALUs in 8-bit mode. (The diagnostic tools provided with the Storm-1 evaluation board do not allow measurement of the VLIW packing ratio or the ALU utilization to assess the average level of instruction-level parallelism achieved by the decoding algorithm, however.)

As a result of the match between the available parallelism in the LDPC code's parity-check matrix and the SIMD parallelism of the processor, straightforward programming of the TDMP decoding algorithm results in an implementation in which extrinsic messages and updates for a given row of the parity-check matrix are managed by the same lane of the processor throughout the decoding of a code word. Consequently, information defining the variable-node participation in a given row is loaded once into the operand register files of the responsible lane and retained for the duration of decoding. The structure of the parity-check matrix permits its representation in a form amenable to efficient loading and storing, which we exploit (and which is exploited elsewhere in implementations on GPUs [5,18,19]).

The posterior value for each variable node must be communicated to another lane after it is updated in a subiteration, in general, where the recipient lane is the one that requires the value in the most immediate future subiteration. Most of the latency for these data transfers can be hidden during the message-passing phase (Steps 2-7) of the TDMP algorithm due to the computation required during the phase. The same posterior transfers must occur during the parity-check phase, however, and the lower computational load in that phase exposes most of the latency as a decoding delay. As shown in the next section, the exposed inter-lane communication latency can be a significant factor in limiting the throughput of the TDMP decoder using the standard schedule consisting of separate message-passing and parity-check phases in each iteration.

Imperfect regularity in the data structures employed by the TDMP decoder can result in conditional execution and branching which markedly reduces the average utilization of the processor's resources. These problems arise if the LDPC code is irregular in the row weights of the parity-check matrix, such as the WiMAX codes we consider in our examples. (The same observation is noted in [5] regarding decoding irregular LDPC codes on a GPU.) We address this problem by adding "dummy" variable-node positions to the representation of the parity-check matrix and corresponding dummy circulant permutation submatrices to each row-block containing fewer than the maximum number of non-zero submatrices. The apparent row-weights of the code are consequently "regularized", which eliminates the need for conditional execution of instructions in key parts of the code. Each dummy variable node is initialized with a prior value set to the largest possible magnitude of a binary zero in its signed, 8-bit representation. No two row-blocks contain non-zero dummy circulant submatrices in the same column positions, so each dummy variable node participates in message passing for only one row. The technique

reduces the computation time per iteration at the cost of a modest increase in the size of the representation of the parity-check matrix. Simulations show it has no measurable detrimental effect on the error probability or the convergence behavior of the decoding algorithm.

The TDMP decoder is implemented as a single kernel function on the Storm-1 processor. Each input stream corresponds to the vector of quantized channel samples for one code word, and each output stream corresponds to one detected code word. The instruction kernel and the information defining the code's parity-check matrix are loaded from the global memory into the DPU only once, at the start of decoding; both are retained in the DPU throughout the processing of many consecutive streams (i.e., while decoding many code words). In our implementation, the transfer of the channel samples to the DPU's lane register files from the general-purpose processor occurs while the DPU is otherwise idle. The resulting latency is a negligible fraction of the decoding time for a code word, however.

5 Alternatives for early termination of the offset-min-sum TDMP algorithm on a stream processor

Early termination can be achieved with the TDMP algorithm using the standard schedule, in which each iteration consists of a phase with message passing for all rows of the parity-check matrix followed by a phase with parity checks for all rows of the matrix. The algorithm guarantees that either the decoder produces a valid code word at termination or declares a decoder failure after the maximum allowed number of iterations. In this section, we consider three alternatives to the standard schedule for the offset-min-sum TDMP algorithm with early termination. Two of them are alternatives of practical interest that provide the same correctness guarantee as the standard schedule. The other alternative uses a naive approach that does not guarantee a valid decoded code word. The decoder with the standard schedule and the decoder using the naive approach provide two benchmarks against which we evaluate the performance of the two practical alternatives to the standard schedule.

The performance of the algorithms is illustrated for four WiMAX-standard codes [2], each of which is a QC-LDPC code composed of circulant permutation submatrices of size 64. The block length of each code is 1536, and the rates of the codes are 1/2, 2/3, 3/4, and 5/6. (The rate-2/3 code and the rate-3/4 code are constructed using base-matrix option A in [2].)

The performance of TDMP decoding is evaluated for each code with the Storm-1 processor using its 8-bit mode for saturating, fixed-point arithmetic. The parameters of the decoding algorithm for each code are chosen to provide nearly-optimal performance with the standard schedule. In each instance, a quantization interval of $\Delta = 0.125$, an offset of $\eta = 0.125$, and an extrinsic-update magnitude constraint of $\epsilon = 2.5$ yield the lowest probability of error across signal-to-noise ratios of interest. A maximum of twenty decoder iterations is allowed for any code word, which is sufficient to provide most of the performance achievable with TDMP decoding for the four codes. Decoder execution times are evaluated with the processor decoding a sequence of consecutive code words as would be required in a practical communication system.

The decoding time per code word using the standard schedule is determined by the time required for the three components of decoding. The first component is the set-up and completion that includes the time required to initialize the DPU before the first iteration and the time required to recover decoded data after the last iteration. The other two components represent the time for the message-passing phase of the TDMP algorithm per iteration and the time for the parity-check phase per iteration. The decoding time for a code word using the standard schedule is thus given by the set-up-and-completion time plus the number of decoding iterations multiplied by the sum of the message-passing time and the parity-check time for an iteration.

The first-row entry in Table 1 shows the processing times for the three decoding components for a code word of the rate-1/2 WiMAX code. (The entries in Table 1 are obtained from the cycle-accurate emulator of the Storm-1 processor and verified by decoder implementation on the processor.) The set-up and completion time is 2.13 μs. The message-passing phase for one iteration requires 2.05 μs, while the parity-check phase for one iteration requires 1.28 μs. Other than inter-lane communication of information during each phase, only a few operations are required per row in the parity-check phase, whereas many more operations are required in the message-passing phase. Yet the parity-check phase requires 62% of the time required for the message-passing phase, revealing that the processing time of the separate parity-check phase is dominated by the latency of exposed inter-lane communications.

The processing time per iteration can be reduced substantially if some means is employed to eliminate the inter-lane communications associated with performing parity checks. Consider an alternative approach to testing the correctness of code-symbol polarities in which the parity checks for a given block of rows are incorporated within the corresponding message-passing subiterations. At the end of each subiteration of the message-passing phase, the posteriors updated during the subiteration are used to determine if the corresponding subset of parity checks are satisfied. The current posterior values required for each parity check are already located in the stream-processor lane in which the parity check is performed since the updates of the same values have just been completed in the same lane. Thus the need for

Table 1 Processing time of decoding components for the rate-1/2 WiMAX code

Decoding algorithm	Set-up & completion	Message passing Only (per iteration)	Parity checks only (per iteration)	Integrated message passing & parity checks (per iteration)
Standard schedule	2.13 μs	2.05 μs	1.28 μs	-
Naive IPC	2.13 μs	-	-	2.13 μs
IPC/confirmation	2.13 μs	-	1.28 μs	2.13 μs
IPC/stability check	2.13 μs	-	-	2.15 μs

separate inter-lane communications for parity checks is eliminated. We refer to this non-standard schedule of parity checks as the *integrated parity check (IPC)*. The decoding time for the IPC is shown in the second-row entry in Table 1. The processing time for the integrated message-passing-and-parity-check phase of an iteration is only .08 μs greater than the message-passing phase alone in the standard schedule.

First consider a decoder in which the IPC is used and decoding is terminated after an iteration in which the integrated parity checks are satisfied during all message-update subit-erations. (There is no separate parity-check phase and thus no processing time allocated to such a phase.) The elimination of the parity-check phase in each iteration results in a substantial reduction in the processing time per iteration compared with decoding using the standard schedule. The algorithm does not guarantee that a valid code word is decoded at termination, however; thus, we refer to it as the *naive IPC*.

The probability of error with TDMP decoding is much higher if the naive IPC is used than if the standard schedule is used. This is illustrated in Figure 2 in which the probability of code-word error is shown for both decoders (as well as two others discussed below) for the rate-1/2 WiMAX code. Decoding with the standard schedule results in a probability of code-word error of 10^{-3} at a signal-to-noise ratio of 1.97 dB, whereas acceptable performance is not achievable at a reasonable signal-to-noise ratio for decoding with the naive IPC. (Moreover, most code-word errors yield known decoder failures with the standard schedule, whereas invalid decoded code words are not recognized as such by the decoder with the naive IPC.) Similar results are observed when comparing the probability of bit error for the two decoders and when comparing the probability of error for the other three example codes. Thus decoding with the naive IPC is not of practical interest.

The validity of the decoded code word is guaranteed if the naive IPC is supplemented by a standard parity-check phase which is employed beginning with the first iteration in which all the integrated parity checks are satisfied. This modified IPC is referred to as the *IPC with confirmation*. Alternatively, the same validity guarantee is achieved if the naive IPC is modified by adding *stability checks* in each subiteration. At the end of the subiteration, the lane performing the updates for a given row of the parity-check matrix determines if the updates have changed the sign of the posterior value for any variable node participating in the corresponding parity check. (I. e., it determines if the update has changed the tentative hard decision for any corresponding code symbol.)

The stability checks add .02 μs to the processing time for the message-passing phase of an iteration compared with the naive IPC and the IPC with confirmation, as shown in Table 1. If both the integrated parity checks and the stability checks are satisfied for each message-update subiteration in an iteration, the hard decisions at the end of the iteration are guaranteed to correspond to a valid code word (without the requirement of a separate parity-check phase). This modified IPC is referred to as the *IPC with stability check*. The stability-check technique has been pointed out previously in [10,20], though not in the context of implementation on a stream processor. (It is referred to as an "on-the-fly convergence check"

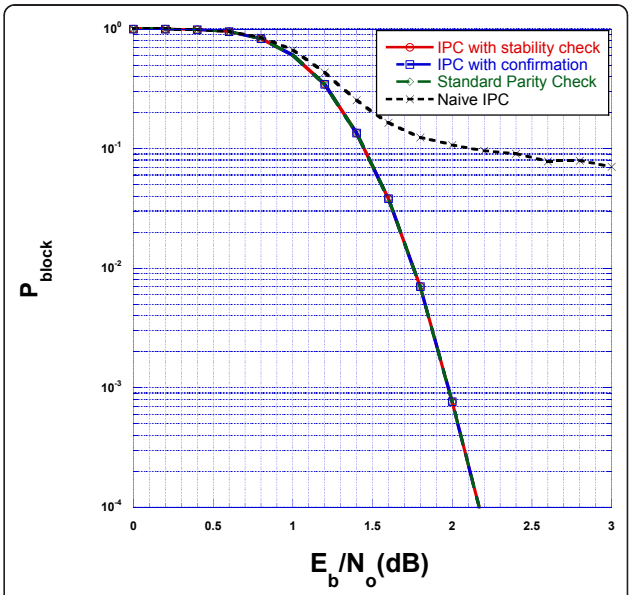

Figure 2 Probability of code-word error for the rate-1/2 WiMAX code.

in the latter article.) No evaluation of its impact on the decoding time is provided in either article.

As seen in Figure 2, both the IPC with confirmation and the IPC with stability check result in the same probability of code-word error as the standard schedule for the rate-1/2 WiMAX code. (For all three algorithms, almost all code-word errors are known decoder failures.) The probability of code-word error with each schedule is 10^{-3} if the signal-to-noise ratio is 1.97 dB, and it is 10^{-4} if the signal-to-noise ratio is 2.18 dB. Agreement of the error probabilities for the three schedules is also observed when considering the probability of bit error and when comparing the performance for the other three WiMAX codes.

The effect of each parity-check schedule on the decoder's information throughput is shown in Table 2 for the rate-1/2 WiMAX code. The throughput is measured as the average number of decoded *information* bits per second. Decoding *without* early termination is considered for several values of the number of decoding iterations, I. (As noted above, the naive IPC is a decoder with a high error probability that provides a throughput benchmark for the practical IPC algorithms.) The standard schedule yields a decoder throughput of 87.4 Mbps and 11.2 Mbps respectively, for two and 20 iterations per code word. This represents 27% and 35% less decoder throughput, respectively, than with the naive IPC. The IPC with confirmation achieves a throughput of 100.2 Mbps with two decoder iterations and 16.7 Mbps with 20 decoder iterations. The throughputs are 17% less and 3% less than with the naive IPC for two and 20 iterations, respectively. (The IPC with confirmation is assumed to employ the parity-check phase only during the Ith iteration in the results shown in Table 2.) The IPC with stability check yields nearly the same throughput as the naive IPC for a given number of decoder iterations. (The difference is no more than 1% in each case shown in Table 2.)

The results in Table 2 do not reflect the fact that the different schedules result in a different average number of iterations if early termination is employed. Nor do they reflect the fact that the IPC with confirmation may initiate the parity-check phase in an iteration prior to the terminating iteration. The average number of decoding iterations for each of the three practical schedules with early termination is shown in Figure 3 as a function of the channel quality (measured by the probability of code-word error achievable by the decoder). For a channel which results in a probability of code-word error of 10^{-4} with each of the three schedules, the IPC with stability check requires an average of 6.1 iterations, whereas the standard schedule requires an average of only 5.1 iterations. Over the range of channel quality of practical interest, in fact, the average number of iterations required with the same two schedules differs consistently by about one iteration.

Table 2 Information throughput without early termination for the rate-1/2 WiMAX code

Decoding algorithm	$I = 2$	$I = 6$	$I = 10$	$I = 20$
Naive IPC (Mbps)	120.4	51.6	32.9	17.2
IPC w/stability check (Mbps)	119.6	51.2	32.6	17.1
IPC w/confirmation (Mbps)	100.2	47.5	31.1	16.7
Standard schedule (Mbps)	87.4	34.7	21.7	11.2

An average of 5.4 iterations are required by the IPC with confirmation for the channel which results in a probability of code-word error of 10^{-4}. This schedule requires approximately 0.2-0.3 iterations more than is required by the standard schedule on average for any channel quality of practical interest. It follows from the definition of the algorithms that the naive IPC must result in a lower average number of iterations and higher throughput than either the IPC with confirmation or the IPC with stability check, but any other comparison of the throughputs of the schedules requires evaluation by execution on the processor or simulation in conjunction with Table 1. The average number of iterations required with each schedule approaches one asymptotically with improving channel quality, but the IPC with stability check approaches its limiting behavior more slowly than either the IPC with confirmation or the standard schedule.

The decoder's information throughput using each schedule with early termination is shown as a function of the signal-to-noise ratio in Figure 4 for the rate-1/2 WiMAX code. Both the IPC with confirmation and the IPC with stability check result in an information throughput of nearly 47 Mbps if the signal-to-noise

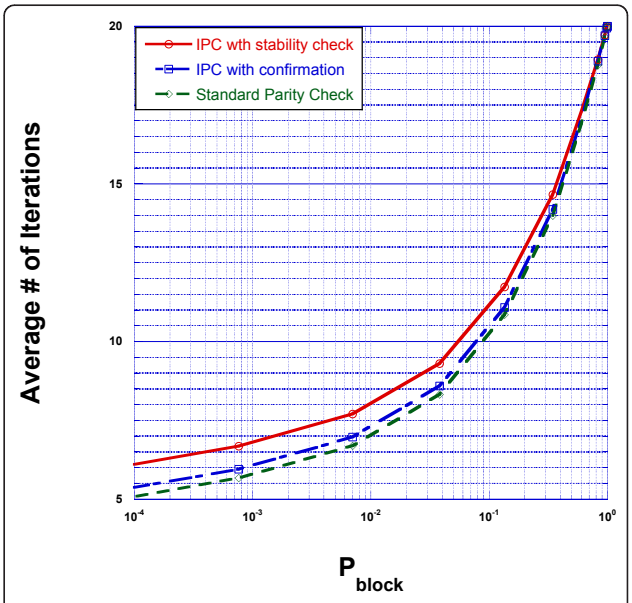

Figure 3 Average number of decoding iterations with early termination for the rate-1/2 WiMAX code.

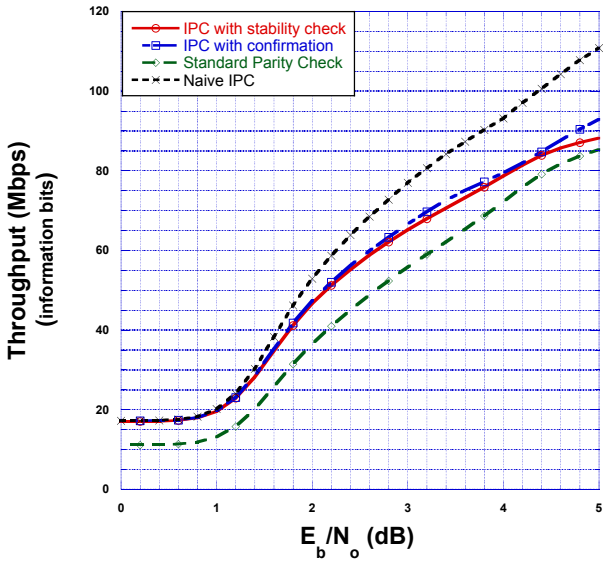

Figure 4 Information throughput with early-termination decoding for the rate-1/2 WiMAX code.

ratio is 2 dB, which is 28% greater than the throughput achieved with the standard schedule. The throughput is 12% less than the throughput with the naive IPC, however, which reflects the extra processing cost of guaranteeing a valid decoded code word at termination. The throughput improvement from using either the IPC with confirmation or the IPC with stability check in place of the standard schedule is reduced to about 21% if the signal-to-noise ratio is 2.8 dB, but the advantage of either over the standard schedule is substantial for signal-to-noise ratios of practical interest.

The IPC with confirmation provides a slightly greater throughput than the IPC with stability check with the rate-1/2 WiMAX code for signal-to-noise ratios between 1 dB and 5 dB. In the limit as the signal-to-noise ratio approaches infinity (in which each decoding attempt requires only one iteration), however, the throughput for both the standard schedule and the IPC with confirmation is approximately 140 Mbps, whereas the limiting throughput for both the naive IPC and the IPC with stability check is nearly 180 Mbps. (Note that accurate simulated or emulated results for the average number of iterations can be obtained using a feasible number of sample outcomes at a higher signal-to-noise ratio than is feasible for accurately measuring the probability of code-word error.)

The information throughput for decoders using the standard schedule, the IPC with confirmation, and the IPC with stability check is shown in Table 3 for each of the four example WiMAX codes and two choices of the achievable probability of code-word error. Both the IPC with confirmation and the IPC with stability check provide much greater throughput than the standard schedule for the different rate WiMAX codes. The IPC with confirmation achieves 8.3%-30.0% greater throughput than the standard schedule over the four codes and the two channel conditions, whereas the throughput resulting with the IPC with stability check is 12.2%-26.0% greater than that obtained with the standard schedule. The largest percentage throughput improvements over the standard schedule are achieved with the lowest-rate code in the lower-quality channel, and the smallest percentage improvements occur with the highest-rate code and the higher-quality channel. The IPC with confirmation yields a slightly greater throughput than the IPC with stability check for the rate-1/2 and rate-3/4 codes, the two schedules result in the same throughput for the rate-2/3 code, and the IPC with stability check produces a slightly greater throughput than the IPC with confirmation for the rate-5/6 code.

Another characteristic of the decoder implementation which is of interest is the memory occupied by its executable code. The DPU in the Storm-1 processor includes an instruction memory of 2,048 384-bit instruction words. In our implementation, the rate-1/2 decoder using the IPC with stability check occupies only 21.2% of the instruction memory, whereas the rate-1/2 decoders using the standard schedule and the IPC with confirmation occupy 24.2% and 25.1% of the instruction memory, respectively. The decoder using the IPC with stability check requires 23.0%, 28.4%, and 76.4% of the instruction memory for the codes of rates 2/3, 3/4, and 5/6, respectively. Over all four codes, 7%-21% more instruction memory is required if the standard schedule is used than if the IPC with stability check is used. Similarly, 10%-28% more instruction memory is required if the IPC with confirmation is used than if the IPC with stability check is used. The greater instruction-memory efficiency of the decoder using the IPC with stability check is the result of the absence of a separate parity-check phase in the schedule.

6 Conclusions

The data-level parallelism of a low-power, embedded stream processor can be used to accelerate the decoding of a QC-LDPC code using variants of the TDMP layered belief-propagation algorithm. The communications among functional-unit clusters of the processor is a significant factor in the decoding time using the standard approach of alternating message-passing and parity-check phases for TDMP decoding. An alternative approach in which the parity checks are integrated with the message-passing phase reduces the exposed communication latency within the processor. Either a separate parity-check confirmation phase or a check for stability of hard decisions throughout an iteration of the

Table 3 Information throughput for the WiMAX code of each rate

P_{block}	10^{-4}				10^{-3}			
Rate	**1/2**	**2/3**	**3/4**	**5/6**	**1/2**	**2/3**	**3/4**	**5/6**
IPC w/								
stability check (Mbps) IPC w/	50.4	88.9	100.0	146.0	46.0	79.7	91.7	131.6
confirmation (Mbps) Standard	51.3	88.9	101.0	140.9	46.8	79.7	93.0	128.9
schedule (Mbps)	40.3	74.4	83.3	129.3	36.0	64.4	79.0	111.4

message-passing phase can be used to ensure a valid code word at termination of the modified algorithm. Both provide a substantial increase in the decoder throughput over the standard TDMP schedule at no cost in the error probability for decoding with early termination. The algorithm employing stability checks permits the most efficient use of instruction memory due to the absence of a separate parity-check phase.

Competing interests

The authors declare that they have no competing interests.

Received: 16 May 2011 Accepted: 12 April 2012 Published: 12 April 2012

References

1. S Lin, DJ Costello Jr, *Error Control Coding*, 2nd edn. (Pearson Prentice Hall, Upper Saddle River, NJ, 2004)
2. IEEE Computer Society and IEEE Microwave Theory and Techniques Society, *IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Broadband Wireless Access Systems*. IEEE Std. 802.16-2009 (29 May 2009)
3. IEEE Computer Society, *IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks – Specific requirements Part 11: Wireless LAN Medium Access Control (MAC), and Physical Layer (PHY) Specifications. Amendment 5: Enhancements for Higher Throughput*. IEEE Std. 802.11n-2009 (29 Oct. 2009)
4. G Falcão, L Sousa, V Silva, Massive parallel LDPC decoding on GPU, in *ACM Symp Princ Prac Parallel Prog*, Salt Lake City, UT, 83–90 (2008)
5. J Ji, J Cho, W Sung, Memory access optimized implementation of cyclic and quasi-cyclic LDPC codes on a GPGPU. J Sig Proc Syst. **64**(1), 149–159 (2011). doi:10.1007/s11265-010-0547-9
6. UJ Kapasi, S Rixner, WJ Dally, B Khailany, JH Ahn, P Mattson, JD Owens, Programmable stream processors. IEEE Comput. **36**(8), 54–62 (2003). doi:10.1109/MC.2003.1220582
7. WJ Dally, UJ Kapasi, B Khailany, JH Ahn, A Das, Stream processors: programmability with efficiency. ACM Queue. **2**(1), 52–62 (2004). doi:10.1145/984458.984486
8. MM Mansour, NR Shanbhag, High-throughput LDPC decoders. IEEE Trans Very Large Scale Integration Syst. **11**(6), 976–996 (2003)
9. MM Mansour, A turbo-decoding message-passing algorithm for sparse parity-check matrix codes. IEEE Trans Signal Process. **54**(11), 4376–4392 (2006)
10. DE Hocevar, A reduced complexity decoder architecture via layerer decoding of LDPC codes, in *Proc IEEE Workshop Signal Proc Systs*, Austin, TX, 107–112 (2004)
11. JA Kennedy, DL Noneaker, Decoding of a quasi-cyclic LDPC code on a stream processor, in *Proc IEEE Military Commun Conf*, San Jose, CA, 2062–2067 (2010)
12. J Chen, MPC Fossier, Density evolution for two improved BP-based decoding algorithms of LDPC codes. IEEE Commun Lett. **6**(5), 208–210 (2002)
13. J Zhao, F Farkeshvari, AH Banihashemi, On implementation of min-sum algorithm and its modifications for decoding low-density parity-check (LDPC) codes. IEEE Trans Commun. **53**(4), 549–554 (2005). doi:10.1109/TCOMM.2004.836563
14. BK Khailany, T Williams, J Lin, EP Long, M Rygh, DW Tovey, WJ Dally, A programmable 512 GOPS stream processor for signal, image, and video processing. IEEE J Solid-State Circ. **43**(1), 202–213 (2008)
15. WJ Dally, J Balfour, D Black-Shaffer, J Chen, RC Harting, V Parikh, J Park, D Sheffield, Efficient embedded computing. IEEE Comput. **41**(7), 27–32 (2008)
16. E Krimer, R Pawlowski, M Erez, P Chiang, Synctium: A near-threshold stream processor for energy-constrained parallel applications. IEEE Comput Architecture Lett. **9**(1), 21–24 (2010)
17. M Woh, S Seo, S Mahlke, T Mudge, AnySP: Anytime anywhere anyway signal processing. IEEE Micro. **41**(1), 81–91 (2010)
18. KK Abburi, A scalable LDPC decoder on GPU, in *IEEE Intl Conf VLSI Design*, Chennai, India, 183–187 (2011)
19. G Wang, M Wu, Y Sun, JR Cavallaro, A massively parallel implementation of QC-LDPC decoder on GPU, in *IEEE Symp Application Specific Processors*, San Diego, CA, 82–85 (2011)
20. K Gunnam, G Choi, W Wang, M Yeary, Parallel VLSI Architecture for Layered Decoding. Texas A&M Technical Report (May 2007)

doi:10.1186/1687-1499-2012-141
Cite this article as: Kennedy and Noneaker: **Scheduling parity checks for increased throughput in early-termination, layered decoding of QC-LDPC codes on a stream processor.** *EURASIP Journal on Wireless Communications and Networking* 2012 **2012**:141.