HF-LDPC: HLS-friendly QC-LDPC FPGA Decoder with High Throughput and Flexibility

Yifan Zhang[1], Qiang Cao*[1], Shaohua Wang[1], Jie Yao[2], Hong Jiang[3]
[1]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology,
[2]School of Computer Science and Technology, Huazhong University of Science and Technology,
[3]Department of Computer Science and Engineering, University of Texas at Arlington,
[12]{xenon_zhang, caoqiang, M202273497, jackyao}@hust.edu.cn, [3]hong.jiang@uta.edu , *Corresponding Author

***Abstract*—LDPC (Low-Density Parity-Check) codes have become a cornerstone of transforming a noise-filled physical channel into a reliable and high-performance data channel in communication and storage systems. FPGA (Field-Programmable Gate Array) based LDPC hardware, especially for decoding with high complexity, is essential to realizing the high-bandwidth channel prototypes. HLS (High-Level Synthesis) is introduced to speed up the FPGA development of LDPC hardware by automatically compiling high-level abstract behavioral descriptions into RTL-level implementations, but often sub-optimally due to lacking effective low-level descriptions. To overcome this problem, this paper proposes an HLS-friendly QC-LDPC FPGA decoder architecture, HF-LDPC, that employs HLS not only to precisely characterize high-level behaviors but also to effectively optimize low-level RTL implementation, thus achieving both high throughput and flexibility. First, HF-LDPC designs a multi-unit framework with a balanced I/O-computing dataflow to adaptively match code parameters with FPGA configurations. Second, HF-LDPC presents a novel fine-grained task-level pipeline with interleaved updating to eliminate stalls due to data interdependence within each updating task. HF-LDPC also presents several HLS-enhanced approaches. We implement and evaluate HF-LDPC on Xilinx U50, which demonstrates that HF-LDPC outperforms existing implementations by $4\times$ to $84\times$ with the same parameter and linearly scales to up to 116 Gbps actual decoding throughput with high hardware efficiency.**

***Index Terms*—LDPC, FPGA, HLS**

I. Introduction

LDPC (Low-Density Parity-Check) codes approaching the Shannon limit of physical channel have been ubiquitously used for error correction in communication and storage systems. LDPC decoding is both data intensive and compute complex, thus relying on FPGA (Field-Programmable Gate Array) based hardware acceleration to achieve high performance.

LDPC decoders have been extensively studied in terms of code structures, decoding algorithms, and hardware implementations [1] [2]. Most of the existing LDPC decoder designs require manual coding in a Hardware Describe Language (e.g., Verilog) to accurately characterize Register-Transfer Level (RTL) implementation. However, this manual development, even for experts, generally takes a long design cycle with continuous iterations to ensure functional correctness and high quality of result (QoR) [3]. Recently, High-Level Synthesis (HLS) is introduced to automatically compile code written in a high-level programming language, such as C/C++, into RTL implementation, achieving high productivity and flexibility. Using HLS, the designers can focus on precisely describing the algorithmic logic and largely offload laborious circuit-level layout (e.g., timing and routing control) to HLS.

However, it is challenging to use HLS to implement a flexible LDPC decoder with high QoR. The high-level abstract algorithmic logic description without a careful consideration for features of the LDPC code and hardware configuration is often compiled to a sub-optimal implementation. Specifically, it is hard for HLS to precisely describe irregular LDPC with high error correction capability. Furthermore, it is difficult for HLS to fully optimize the small-sized decoding unit with low quantization precision, thus decreasing the overall QoR of the LDPC decoder.

To solve this problem, we propose HF-LDPC, an HLS-friendly QC-LDPC FPGA decoder architecture, to achieve both high throughput and high flexibility. To this end, HF-LDPC designs a dataflow-centric multi-unit framework without binding a specific code parameter with the FPGA configuration. HF-LDPC consists of multiple decoding cores (DCs), where each DC comprises an I/O module and multiple decoding units (DUs) to maintain a balanced data stream. Second, HF-LDPC presents a task-level pipeline with a novel interleaved-update mechanism to exploit intra-/inter-block parallelism of QC-LDPC while eliminating stalls due to the data-update dependence between the check node and value node. Third, HF-LDPC also presents several key HLS-enhanced techniques, such as HBM-aware data frame, DC placement, vectorized data access, ping-pong update, and two-phase addressing, to enable further performance optimization.

The main contributions of this work are as follows:

- We design an HLS-friendly and well-modularized QC-LDPC decoding architecture (HF-LDPC) to ensure both flexibility and scalability.
- We propose a novel fine-grained pipeline with interleaved updating for intra-/inter-block parallelism within the DU, along with several performance-boosting techniques.
- We implement and validate the effectiveness of HF-LDPC on Xilinx U50 terms of performance, flexibility, scalability, and efficiency.

This work was supported in part by the National key research and development program of China under Grant 2018YFA0701800, NSFC No.62172175, Creative Research Group Project of NSFC No.61821003, Key Research and Development Project of Hubei No.2022BAA042, and by the US National Science Foundation grant CNS-2008835 and CCF-2226117.

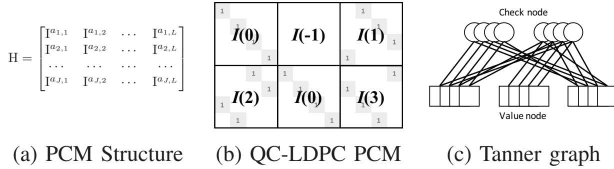
2576-6996/23/$31.00 ©2023 IEEE
DOI 10.1109/ICCD58817.2023.00091

II. Background and Motivation

A. FPGA and High-level Synthesis

FPGA developers traditionally write Hardware Describe Language (e.g. Verilog) code to implement Register-Transfer Level (RTL) hardware, which is generally tedious and error-prone [3]. Recently, High-Level Synthesis (HLS) is advancing to automatically compile an algorithmic-logic behavioral description (e.g., C/C++) into an RTL implementation. Such HLS-based hardware development with high productivity and flexibility has been broadly used in large-scale and new functionality scenarios, e.g., machine learning [4], graph computing [5], and domain-specific accelerators [6]. However, the high-level abstract behavior without detailed low-level descriptions can map to many low-level implementations, most of them are sub-optimal. HLS offers pragmas to precisely characterize the behavior-description to improve the quality of HLS results (QoR) [7], [8].

B. QC-LDPC



(a) PCM Structure (b) QC-LDPC PCM (c) Tanner graph

Fig. 1: An example of QC-LDPC parity check matrix (PCM) and the corresponding tanner graph

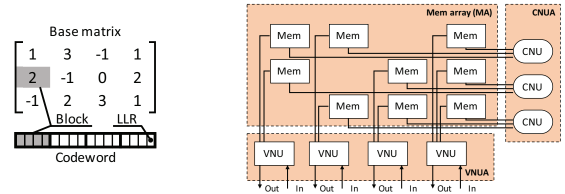
LDPC codes have superior error-correction capability at the cost of computation complexity. LDPC is defined by a parity check matrix (PCM). Each row in PCM corresponds to a check node (CN), each column corresponds to a variable node (VN) and a "1" in the PCM indicates the existence of an edge between VN and CN. All VNs, CNs, and edges constitute a tanner graph. Quasi-Cyclic LDPC (QC-LDPC) [9] codes with well-structured PCM are suitable for hardware implementation without significant error-correction loss, which has been widely used in the communication and storage fields. Fig. 1 gives an example of QC-LDPC and its tanner graph. The PCM of example QC-LDPC consists of 2×3 $I(n)$ sub-matrices. The size of $I(n)$ is $Z \times Z$. An $I(n)$ sub-matrix is an n-step circulant permutation matrix from the unit matrix ($I(0)$). $I(-1)$ represents a zero matrix.

Therefore, a specific base matrix and the size of the sub-matrix (expansion factor) determine a specific PCM and the corresponding QC-LDPC code.

C. QC-LDPC Decoding Algorithm and Structure

The QC-LDPC decoding typically employs the message-passing based min-sum algorithm, which converts the complex posterior probability calculations into the logarithmic form, thus simplifying the computational complexity. Each bit in the codeword is actually represented by the log-likelihood ratio (LLR) value. The decoding process updates LLR messages between CN and VN iteratively, until the stop-condition or a predefined iterative threshold is met. These sub-matrices of PCM are also referred to as blocks [10]. Since the initialization values for each column in the block come from the corresponding LLR at the corresponding positions "1" in the PCM as shown in Fig. 1b. Each block corresponds to a memory block, and block-wise circular shifting is achieved by addressing each block with a different address offset. The memory block stores the received messages after each update.

The QC-LDPC decoding algorithms are classified into the two categories of flood and layered update scheduling. The flood scheduling updates all VNs before updating CNs, and vice versa, which has high error correction performance but low decoding convergence speed. In contrast, the layered scheduling has a faster decoding convergence speed at the cost of error-correction performance without scaling circuits [11]. Moreover, the data dependence among layers causes memory-access contention [12].



(a) Example Check Matrix and Related Codeword (b) A QC-LDPC Decoder Unit

Fig. 2: The Inter-block Parallel Structure of a QC-LDPC Decoder

The decoding parallel algorithms also fall into the intra-block and inter-block categories. The former parallelizes the update of information within a block but serially traverses all blocks in the PCM. They typically require Z serial-update units and parallel shifters. On the contrary, the latter serially processes a block but multiple blocks in parallel, as shown in Fig. 2b. They typically require the numbers of parallel CN and VN update units to be equal to the numbers of rows and columns in the base matrix, achieving high hardware efficiency, but at lower flexibility by binding to a specific code.

D. Motivation

Most of the RTL-based prior works focus on designing a decoding unit to fully exploit hardware resources but are strictly dedicated to a specific QC-LDPC code and FPGA hardware. Existing HLS-based implementations ease development tasks but generally exhibit low QoR. We are motivated to effectively leverage HLS to design scalable and flexible QC-LDPC adapting varied code parameters to FPGA configuration to achieve high decoding performance and hardware efficiency.

III. HF-LDPC Decoder Design

A. HF-LDPC Architecture

We propose an HLS-friendly decoder architecture, HF-LDPC that combines a high-level dataflow-centric multi-unit framework and a low-level interleaved-update task-level pipeline that exploits inter-/intra-block parallelism. HF-LDPC comprises multiple decoding cores (DCs), as shown in Fig. 3.

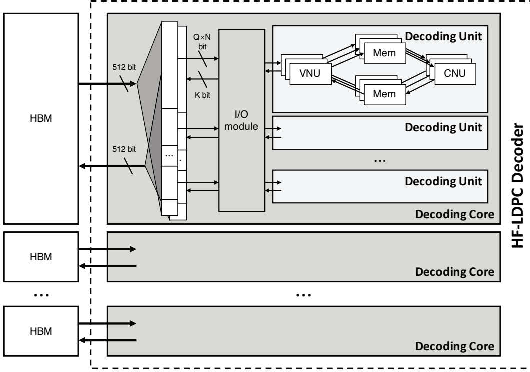


Fig. 3: HF-LDPC Architecture

A DC with a limited scale consists of an input/output (I/O) module and an array of decoding units (DUs), which is connected to an HBM channel. The I/O module reads the codeword from HBM to the DUs and writes decoded data from the DUs to HBM, to meet the data-processing capacity of DUs. HF-LDPC employs the inter-block parallel flood scheduling that decodes all blocks in parallel. We design an interleaved-updating task-level pipeline within a DU to eliminate the stalls due to cyclic data dependence between VNU and CNU in the flood scheduling.

At the high level, we employ HLS to design a dataflow-centric framework, adapting to different quantization precisions and PCM. The codeword stream sequentially flows through the input module, multiply DUs, and output module, which are well described by HLS with the `DATAFLOW` pragma.

Additionally, we also take full advantage of HLS to optimize the RTL-level implementation at the circuit level within well-defined units. For DU, HF-LDPC also leverages HLS to not only accurately characterize intra-/inter-block parallel algorithms, but also to automatically optimize the RTL-level implementation with HLS-enhanced expressions, thus improving hardware efficiency.

B. Multi-unit Decoding Core

Considering that modern FPGAs are evolving with HBM and multi-die layouts, the HF-LDPC design fully utilizes these FPGAs by adopting a multi-DUs structure in each DC so that a DU has a limited scale and fixed iterative-update pipeline. Note that a given LDPC code and the decoding algorithms determine the input/output bit-width of each DU. Therefore, the bit-width of the HBM channel dictates its supportable number of DUs. As a result, DCs are completely independent of one another without data correlation and are entirely placed within an FPGA die.

To well balance data access and data process in terms of both data width and decoding time, the IO module needs to connect HBM and the DUs to form a complete data path. However, this will be challenging for manual RTL-level design to accomplish due to strict cycle-and-circuit level constraints of complex protocols when accessing off-chip memory using high-speed buses in FPGA. Fortunately, HLS can analyze the function interfaces and I/O behaviors of the user code to automatically generate a high-speed bus interface module.

Moreover, HLS' ability to allow users to use the same code with different pragma directives to flexibly adjust microarchitectures makes it possible to decouple the high-level DC from the low-level DU while providing flexible support to tolerate different delay characteristics, interface bit width and internal parallelism of DU.

For example, Xilinx FPGA acceleration cards employ the AXI-MM protocol with XDMA to access HBM. Efficient utilization of high-speed buses requires an appropriate burst transfer length. In HLS, the burst transfer length can be automatically configured by analyzing the user code. The modularized multi-module architecture can be described by HLS accurately and efficiently. For example, HF-LDPC designs a module-level pipeline using the `DATAFLOW` pragma to mask the HBM latency (approximately 70 cycles or more). The high-level pipeline also employs inter-module buffering (e.g., ping-pong or FIFO) without modify the interface of DU.

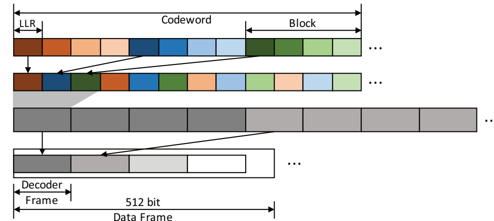


Fig. 4: HBM-aware Data-Frame Format

1) HBM-aware Data-Frame: Note that different code parameters have their corresponding sizes of the codeword and bit-access patterns. When multiple DUs process codewords in parallel, the I/O module needs multiple cycles to read codewords from and write decoded words to the HBM channel respectively. Furthermore, the number of cycles for data-in and data-out should be lower than that of the cycles for DU.

Besides, each DU pipeline needs to read multiple LLRs simultaneously from different blocks in the codeword. This means that the order in which bits are accessed is different from the order of bits in the codeword. This bit-order mismatch needs additional buffers, addressing hardware and extra cycles to re-order the bits.

To avoid the reordering within the critical DU, we define an HBM-aware data-frame matching the HBM width. As shown in Fig. 4, the decoder-frame packs LLRs with the same address offset in different blocks of a single codeword. One or more decoder-frames in a data-frame can be filled up to ensure multiple DUs working in parallel.

For example, suppose a decoder uses a base matrix with a column size of 24 and quantization precisions of 2 bits. In this case, a DU has 48 bits input width. Since the HBM interface width is 512 bits, we can transfer 10 decode-frames in a data-frame, thus inputting data into 10 DUs in parallel.

HF-LDPC offers a flexible and modularized approach to feeding data to multiple DUs with different parameters without

modifying its architecture. Furthermore, due to the match between frame formats and the input order of DU, it does not need additional buffers or scheduling, improving the hardware efficiency.

```
1  Dtype Mem[...][LEN];
2  #pragma HLS ARRAY_RESHAPE ...
3  ...
4     offset = expression(i);
5
6
7
8
9
10    for(i = 0 ~ LEN){
11       Mem[...][offset] = ...;
12    }
13 ...
```

(a) original HLS code

```
1  Dtype Mem[...][LEN];
2  #pragma HLS ARRAY_RESHAPE ...
3  ...
4     offset = expression(i);
5     Dtype proxy[LEN];
6     #... ARRAY_PARTITION var=proxy
7     for(i = 0 ~ LEN){
8        proxy[offset] = ...;
9     }
10    for(i = 0 ~ LEN){
11       Mem[...][i] = proxy[i];
12    }
13 ...
```

(b) optimized HLS code

Fig. 5: Indirect Vectorization

2) Indirect Vectorization: The DU decoding pipeline reads data from different blocks in parallel. To meet the parallel-access requirements of read and write operations, the pragma `array_partition` is commonly used to divide a memory block into multiple partitions with multiple read and write ports. When accessing the same offset addresses in those memory partitions in parallel, the memory can be implemented as vectorized memory block sharing a single addressing circuit.

However, when accessing multiple elements in the vectorized memory block, HLS can mistakenly detect the existence of data dependency and must serialize the multiple accesses. To avoid this misunderstanding, we temporarily interpret the vector as separate scalars, which is referred to in this paper as *vector proxy*.

Specifically, as shown in Fig. 5a, the pseudocode reads data from a vectorized memory block. Line 1 and 2 together represent a vectorized array, which is implemented by HLS as a vector memory block with a width of LEN. Due to the complex expression of line 4, the accesses to the vector memory block in Line 11 may be analyzed by HLS as I/O contention (can happen whether the $offset$ is on the left or right side of $=$).

Therefore, we introduce Indirect Vectorization, for which the code can be modified as shown in Fig. 5b. Lines 5 and 6 define a vector proxy of the same width as the vector memory block with `array_partition`. Line 8 rewrites the original assignment to the vector memory block to assign to the vector proxy. Line 11 further assigns the elements of the vector proxy to the vector memory block one by one.

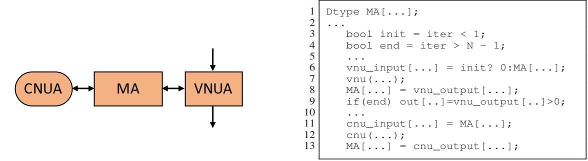
After this code modification, HLS does not mistakenly detect data dependency and correctly implements a vectorized memory block with parallel accessing. Since the additional array only performs assignment operations, it may only add a small number of registers or is expressed as a wire connection, resulting in small or no increase in hardware consumption.

3) DC Hardware Allocation: When the design of the DC is determined, we consider scaling multiple DCs to efficiently utilize the resources on the FPGA for high actual decoding throughput.

Specifically, each DC is connected to an independent HBM to avoid memory channel contention. Additionally, all DCs are evenly placed in different Super Logic Region (SLR) areas to avoid high circuit delays caused by crossing SLRs and circuit congestion that may occur prematurely due to uneven SLR utilization during automatic placement. This approach can keep a high implementation frequency of the FPGA.

C. Decoding Unit

Within the decoding unit, we employ a task-level pipeline with interleaved updating to eliminate the stalls caused by flood scheduling. Additionally, we design a two-stage addressing approach to effectively jointly exploit inter-block parallelism with intra-block parallelism.



(a) Decoder Structure (b) HLS code

Fig. 6: The Decoding Process of Iterative Update

1) Interleaved Update: In DU, groups of CNUs and VNUs are referred to as CNU array (CNUA) and VNU array (VNUA) respectively. The memory block, referred to as memory array (MA), stores messages passed between CNUA and VNUA, as shown in Fig. 6a, the corresponding HLS code is shown in Fig. 6b.

In the flood scheduling, the LLRs are updated by VNUA and CNUA iteratively. There exists data dependence between the CN and VN updates within the decoding pipeline, causing stalls. To eliminate such data dependence, CNUA and VNUA update two codewords simultaneously in an interleaved manner.

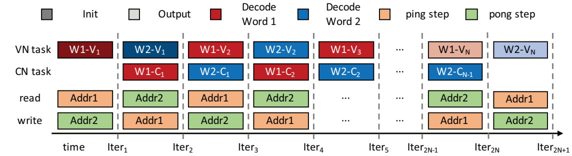
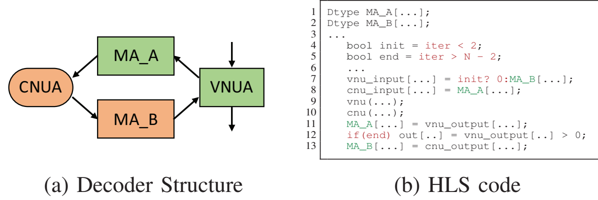


Fig. 7: The Decoding Pipeline of Interleaved Update

Specifically, as shown in Fig. 7, first, the number of iterations is extended to $K * N + N - 1$ according to the number of cyclic dependency tasks K, and the original number of iterations N. For example, if there are only VN and CN updating tasks involved in the cyclic dependency, then $K = 2$. And the number of iterations is extended to $2N + 1$. In terms of scheduling, pipeline initialization is performed during iterations 1 to 2, normal iterations occur from iteration 2 to $2N$, and pipeline output from iteration $2N - 1$ to $2N + 1$.

We design the decoder structure as shown in Fig. 8a, and the corresponding HLS code is shown in Fig. 8b. MA is divided into MA_A and MA_B in Lines 1 and 2. The read interface of VNUA is connected to MA_B in Line 7, while its write interface is connected to MA_A in Line 11. Similarly, the read interface of CNUA is connected to MA_A in Line 8, and its write interface is connected to MA_B in Line 13. The



(a) Decoder Structure (b) HLS code

Fig. 8: The Decoding Process of Interleaved Update

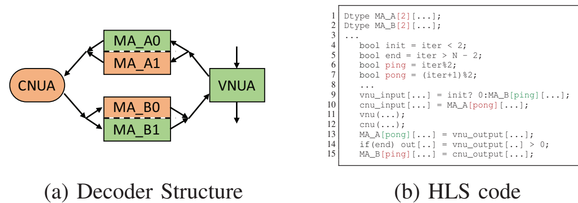
data inter-dependency is eliminated. Furthermore, to obtain correct pipeline initialization and output, the extra scheduling statements in Lines 4 and 5 are necessary to redirect the output and input of modules in Lines 8 and 12.

After modifying the code in this way, since there is no longer a read-after-write operation as shown in Fig. 6b in Lines 11 and 8, the dependency is eliminated, so that HLS can schedule CNUA and VNUA for parallel operations.

2) Ping-pong updating: In fact, the update operations take multiple cycles, and CNU and VNU have different traversal orders for their updates. An incorrect update order causes erroneous decoding results.

For example, when updating position x, CNU reads an LLR of the first codeword from MA_A and writes it to MA_B, thus overwriting the corresponding LLR of the second codeword required by VNU. When updating position x, VNU should have read the LLR of the second codeword from MA_B, but VNU will read a wrong LLR overwritten by CNU.

To avoid such order-induced errors, it is necessary to ensure that the memory address ranges for CN and VN operations do not overlap during the update process. Considering that CN and VN operations are always parallel, we have designed a ping-pong update approach, as shown in Fig. 7, to achieve the correct decoding process with minimal hardware overhead.



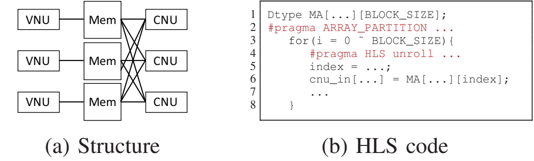
(a) Decoder Structure (b) HLS code

Fig. 9: The Decoding Process of Ping-pong Updating

Specifically, as shown in Fig. 9b we introduce two single-bit control variables as ping and pong, which always have different values that are switched at each iteration. In terms of code, the values of ping and pong are calculated based on the current iteration count in Lines 6 and 7. The stride of the array representing MA is increased to 2 in Lines 1 and 2. The expressions for the read and write operations related to MA are modified to set ping and pong to fill the stride for addressing offset in read and write operations in Lines 9, 10, 13 and 15.

This approach guarantees the correctness of results by only adding one bit for addressing and doubling the memory depth.

3) Two-stage Addressing: Since the number of CNUs and VNUs in inter-block parallel decoders is usually much smaller than that of the intra-block parallel decoders, the proportion of decoding components is limited. Even if the parallel CNU and parallel VNU are more efficient, the entire decoder remains inefficient. To address this issue, we attempt to combine intra-block parallelism with inter-block parallelism to increase the number of decoding components, improving the efficiency of the HF-LDPC decoder.

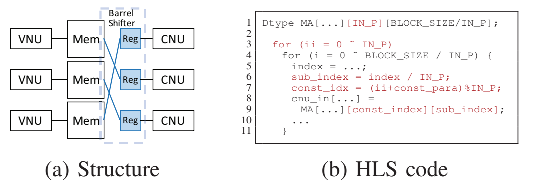


(a) Structure (b) HLS code

Fig. 10: The Potential IO Competition of Intra-block Parallel

Intuitively, as shown in Fig. 10a, increasing the intra-block parallelism only adds appropriate pragma `unroll` in the serial loops to increase the number of node update components as Line 4, and splits the corresponding memory block using `array_partition` pragma as Line 2 to satisfy the I/O requirements of the decoding components. However, when implementing this optimization, HLS is unable to generate an expected pipeline design with the initiation interval (II) being equal to 1, resulting in a suboptimal pipeline schedule and decreasing the decoding throughput.

After analyzing the HLS logs and code behaviors, we find that VNU and CNU in each iteration have different address offsets when using `array_partition`. Therefore, HLS interprets this as VNU and CNU accessing the same sub-memories, as shown in Fig. 10a. HLS is unable to determine whether there exists I/O competition, thus can not give a $II = 1$ pipeline scheduling.



(a) Structure (b) HLS code

Fig. 11: The Two-stage Addressing for Intra-block Parallel

However, due to the cyclic characteristic of sub-matrices in QC-LDPC, such I/O competition will not occur actually. When partitioning each memory block in MA, circular shifting can be achieved by a two-stage addressing process, as shown in Fig. 11a. Therefore, we guide HLS to explicitly partition the memory blocks using the two-stage addressing approach.

We modify the original HLS code as shown in Fig.10b. Specifically, we introduce an additional dimension to the arrays, which allows for explicit data layout, and then decomposes the original addressing variable into a constant address and a variable address offset. This explicitly informs HLS that a barrel shifter in combination with address offsetting should be used to access the multiple sub-memories.

In this way, at the cost of increasing one pipeline stage, we ensure that HLS can achieve an optimal pipeline design while exploiting the intra-block parallelism.

IV. Experiment and Evaluation

A. Experimental Configuration

TABLE I: Hardware Resources and Memory of FPGA Platform A-U50-P00G-PQ-G

Platform information	Value
User budget (LUT)	750k
Max clock frequency (MHz)	500
HBM2 total capacity (GB)	8
HBM2 bandwidth (GB/s)	201

We use the Xilinx Vitis 2020.2 hardware development platform to implement HF-LDPC on Xilinx Alveo U50 data center accelerator card (A-U50-P00G-PQ-G) with an official firmware (Xilinx_u50_gen3x16_xdma_201920_3). This platform integrates Vivado HLS, Vivado, XRT middleware, and a semi-automatic configuration generator. FPGA is connected to the host via PCIe3.0.

The key parameters are listed in Table I. All the actual throughputs are measured running on FPGA instead of calculated by delay and frequency (throughput).

B. Overall Performance

In our evaluation, the HF-LDPC prototype is compared against the state-of-the-art (SOTA) decoders with their best-performing versions, as listed in Table. II. Furthermore, for a fair comparison, we have implemented multiple versions of our HF-LDPC decoder with the same hardware parameters and QC-LDPC parameters as the SOTA decoders.

Table. II demonstrates that HF-LDPC achieves very high actual throughput, surpassing the performance of SOTA decoders by a factor ranging from $4\times$ to $84\times$ with the same set of parameters. Moreover, the efficiency (throughput or actual throughput divided by hardware consumption) of HF-LDPC also exceeds existing HLS-based solutions, even surpassing RTL-based solutions.

These results indicate that HF-LDPC can efficiently utilize the hardware resources of modern FPGAs to achieve high actual decoding throughput.

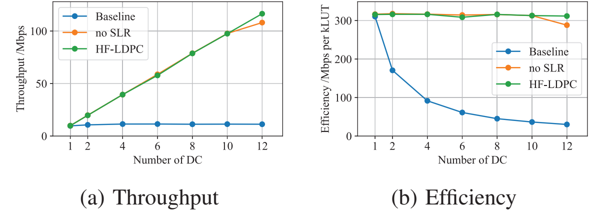
C. HF-LDPC Decoder Scalability

1) HF-LDPC Decoder DC Hardware Allocation: In this experiment, we fix a specific set of parameters to evaluate the optimizations in a multi-DC decoder.

Fig. 12 illustrates the effects of different allocation strategies: no specified HBM channel allocation and DC placement

TABLE II: Decoder Performance Comparison

Method	Technology	Code Rate	Quant /bit	Iter.	Throughput /Gbps	Efficiency /Mbps per kLUT
HF-LDPC	HLS(C++)	1/2	2	5	116.51	311.65
HF-LDPC	HLS(C++)	1/2	6	10	16.97	47.75
HF-LDPC	HLS(C++)	3/4	6	10	11.87	39.26
HF-LDPC	HLS(C++)	3/4	6	10	16.70	36.69
[13]	RTL	3/4	6	10	0.85	35.20
[14]	RTL	7/8	6	14	2.00	35.00
[15]	RTL	1/2	3	20	3.36	42.84
[16]	HLS(C++)	1/2	6	10	0.19	7.68
[17]	HLS(Labview)	1/2	6	5	0.28	14.95



(a) Throughput (b) Efficiency

Fig. 12: Multi-DC HF-LDPC Decoder

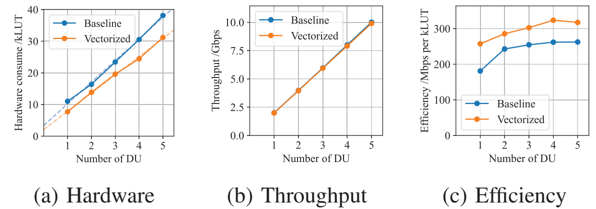
(*baseline*), only specifying HBM channel allocation (*no SLR*), and simultaneously specifying HBM channel allocation and DC placement (*HF-LDPC*). As the number of DCs increases, the actual throughput of baseline remains almost unchanged due to IO bus contention. The actual throughput of *no SLR* exhibits nearly linear growth with the number of DCs, reaching up to $11\times$ actual throughput. This indicates that the memory channel allocation is efficient.

However, at the maximum number of DCs, there exists an actual throughput decrease for *no SLR*. This is because when no specific SLR is specified, the default placement strategy prioritizes placing the DCs in a particular SLR near the HBM channel. This can lead to circuit congestion in that SLR, thus resulting in a decrease in implementation frequency and reduce actual throughput. By specifying the SLR for each DC, *HF-LDPC* successfully improved the implementation frequency, and the actual throughput increases from 108.06Gbps to 116.51Gbps, achieving an $11.8\times$ actual throughput improvement over the baseline.

The experiments demonstrate that the DC hardware allocation of HF-LDPC enables linear scalability.

2) Decoding Core: In this experiment, we fix a specific set of parameters to evaluate the optimization effect of multi-DU DC and vectorized buffer.

All decoders in this experiment consist of one DC with a different number of DUs. Each DU applies all optimizations.



(a) Hardware (b) Throughput (c) Efficiency

Fig. 13: Multi-DU Decoding Core

Fig. 13a and Fig. 13b show that the actual throughput of DC scales linearly with the number of DUs. The hardware consumption of DC increases by $2.83\times$ while the actual throughput improves by $4.95\times$, thus resulting in a reduction of 18.2% in hardware consumption compared to the baseline.

Furthermore, when linearly fitting on the data in Fig. 13a, the intercept of the fitted line as the hardware consumption of the IO module in the DC is small, only 6.22% in 5-unit DC. This indicates that our designed IO module efficiently supports the scalability of the DUs, shown in Fig. 13c.

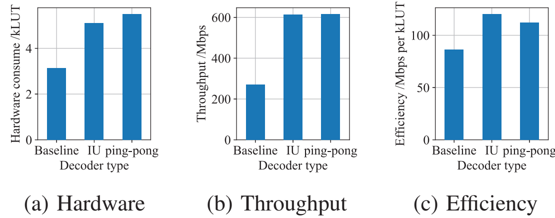
In conclusion, we leverage HLS to design an efficient DC

while improving the Quality of Results (QoR).

D. Decoder Unit

In this experiment, we evaluate several key optimizations applied in the decoder unit. The QC-LDPC PCM in the experiments uses a code rate of 1/2 in the 802.16e standard with an expansion factor of 64.

1) Interleaved Update: In this experiment, with a specific set of parameters, we evaluate the optimization effect of the interleaved update mechanism in the decoder. All decoders in this experiment consist of one DC, one DU per DC, and no intra-block parallelism applied in each DU. The compared items include the *Baseline* without the interleaved update, the *IU* with only interleaved update, and the *ping-pong* with both interleaved update and ping-pong scheduling.



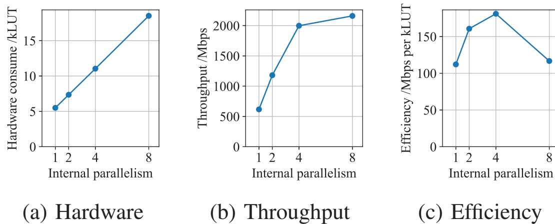
(a) Hardware (b) Throughput (c) Efficiency

Fig. 14: Interleaved Update

Fig. 14a and Fig. 14b demonstrate that the interleaved update mechanism roughly doubles the actual throughput. And the ping-pong addresses the issue of data overlap during decoding at the cost of a small hardware consumption. Fig. 14c illustrates that the interleaved update mechanism improves the efficiency by less than $2\times$. Considering that modern RTL toolchains often incorporate dozens of optimization steps, even if the baseline design is not ideal, it may be optimized during the FPGA implementation phase by the RTL toolchain, thus enhancing the efficiency of the baseline.

However, our experimental results indicate that relying solely on the automatic optimizations provided by HLS and RTL toolchains is insufficient. A well-considered coarse-grained dataflow design can further enhance the Quality of Results (QoR) of HLS designs.

2) Intra-block parallelism: After applying the two-stage addressing, we can adjust the internal parallelism of the DU and utilize HLS for automatic pipeline redesign. Therefore, in this experiment, we designed DUs with different internal parallelism. All decoders in this experiment consist of one DC, one DU per DC, and each DU applies interleaved update and ping-pong scheduling.



(a) Hardware (b) Throughput (c) Efficiency

Fig. 15: Two-stage Addressing

Fig. 15a shows that intra-block parallelism is linearly scaling with hardware. When the internal parallelism (the numbers of CNUAs and VNUAs) is low, the update units without two-stage addressing only consume a small portion of the hardware resource. With two-stage addressing, HLS successfully improves the internal parallelism and efficiency of DU, as depicted in Fig. 15c.

Note that, as shown in Fig. 15b, when the internal parallelism exceeds 4, the actual throughput of the decoder reaches the peak, due to a good speed-matching between the I/O module and the update units in the DC. When the internal parallelism surpasses the number of decoder's iterations, the I/O module becomes the bottleneck.

Therefore, for a decoding unit with 5 iterations, internal parallelism of 4 achieves an optimal design balance. The optimal balance point may vary with different numbers of iterations.

E. Decoder Unit Flexibility

To assess flexibility, we implemented three sets of decoders with different parameters. The first set consists of baseline decoders without any optimization, the second set of decoders use interleaved update only, and the third set of decoders further incorporates the two-stage addressing. Each set consists of 15 decoders, representing various combinations of code rates and quantization precisions to demonstrate the flexibility of our approach. Each decoder is a complete HF-LDPC decoder with only one DU.

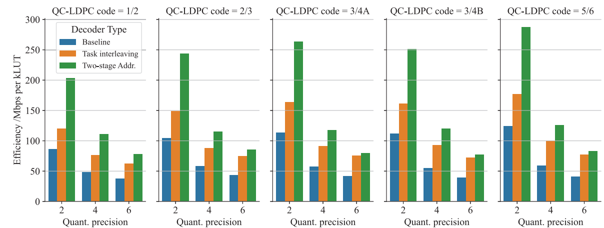


Fig. 16: Decoder Unit Flexibility

Focusing on the green bars in Fig. 16, we can observe that HF-LDPC not only adapts to different decoder parameters but also achieves similar efficiency with similar parameters, which demonstrates flexibility. When comparing all the decoders as a whole, low quantization precision decoders primarily improve efficiency through two-stage addressing, while high quantization precision decoders primarily improve efficiency through interleaved update.

This is because when the quantization precision is lower, the scale of the update units is smaller, and MA and the interface modules in the HF-LDPC decoder consume most of the hardware resources. However, when the quantization precision is higher, the update units are larger, and the hardware efficiency primarily comes from increased internal parallelism.

The result shows that the proposed DU of HF-LDPC is an effective and general optimization, and improves efficiency by $1.9\times$ to $2.4\times$.

V. Related Work

A. Interleaving-based QC-LDPC decoder

Amaricai et al. [18] and Kumawat et al. [19] use a layered algorithm, with each layer performing a specific update. It is close to a fully-parallel algorithm and achieves high throughput. However, adjusting QC-LDPC codes also change the number of layers, hence the pipeline needs to be redesigned. Furthermore, when there are a large number of LLRs that need to be updated in parallel in the CNU, the implementation frequency becomes significantly low. In some cases, it may not even be possible to complete the routing.

Milicevic et al. [11] use the flooding algorithm, but only one type of update is performed per iteration. The number of iterations can only be a multiple of the number of interleaved sets, causing a lower actual throughput.

Weiner et al. [20] use the flooding algorithm, where each row of the base matrix corresponds to a specific task. Similar to the layered interleaved approach, it requires a redesign of the pipeline when changing the QC-LDPC code.

B. HLS-based QC-LDPC decoder

Mhaske et al. [21], [22] propose a layered decoder based on LabVIEW-based HLS. This HLS approach represents the hardware design using circuit diagrams, which has a larger semantic gap than mainstream HLS based on software programming languages. Additionally, due to the insufficient exploitation of inter-block and intra-block parallelism, the throughput per unit of hardware is generally lower.

Wang et al. [16] implement a performance-balanced general-purpose QC-LDPC decoder design using Vivado HLS, which is similar to the expert design in 2007 [10]. However, this work only adopts inter-block parallelism and does not eliminate the dependency between VNU and CNU in the flooding decoding algorithm. As a result, the throughput per unit of hardware is relatively low.

VI. Conclusion

By leveraging HLS ability and enhancing detailed hardware behavior descriptions, we present an HLS-friendly QC-LDPC decoding architecture to improve QoR of HLS, achieving efficiency comparable to RTL solutions while maintaining the flexibility of HLS. With key optimization methods, HF-LDPC's actual throughput performance surpasses SOTA decoders with similar parameters by $4\times$ to $84\times$, up to 116 Gbps.

Acknowledgment

This work was supported in part by the National key research and development program of China under Grant 2018YFA0701800, NSFC No.62172175, Creative Research Group Project of NSFC No.61821003, Key Research and Development Project of Hubei No.2022BAA042, and by the US National Science Foundation grant CNS-2008835 and CCF-2226117.

References

[1] K. K. Gunnam, G. S. Choi, W. Wang, E. Kim, and M. B. Yeary, "Decoding of Quasi-cyclic LDPC Codes Using an On-the-Fly Computation," Oct. 2006, pp. 1192–1199.

[2] E. A. Papatheofanous, D. Reisis, and K. Nikitopoulos, "LDPC Hardware Acceleration in 5G Open Radio Access Network Platforms," *IEEE Access*, vol. 9, pp. 152 960–152 971, 2021.

[3] R. Nane, V.-M. Sima *et al.*, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[4] P. Goswami, M. Shahshahani, and D. Bhatia, "MLSBench: A Synthesizable Dataset of HLS Designs to Support ML Based Design Flows," ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 2020, p. 312.

[5] X. Chen, H. Tan *et al.*, "ThunderGP: HLS-based Graph Processing Framework on FPGAs," ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 69–80.

[6] J. Li, Y. Chi, and J. Cong, "HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration," ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 51–57.

[7] Y.-k. Choi and J. Cong, "HLS-based optimization and design space exploration for applications with variable loop bounds." San Diego California: ACM, Nov. 2018, pp. 1–8.

[8] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," ser. DAC '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 1–6.

[9] R. Townsend and E. Weldon, "Self-orthogonal quasi-cyclic codes," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 183–195, Apr. 1967.

[10] Z. Wang and Z. Cui, "A Memory Efficient Partially Parallel Decoder Architecture for Quasi-Cyclic LDPC Codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 4, pp. 483–488, Apr. 2007.

[11] M. Milicevic and P. G. Gulak, "A Multi-Gb/s Frame-Interleaved LDPC Decoder With Path-Unrolled Message Passing in 28-nm CMOS," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 10, pp. 1908–1921, Oct. 2018.

[12] K. Zhang, X. Huang, and Z. Wang, "High-throughput layered decoder implementation for quasi-cyclic LDPC codes," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 6, pp. 985–994, Aug. 2009.

[13] V. L. Petrović, M. M. Marković *et al.*, "Flexible High Throughput QC-LDPC Decoder With Perfect Pipeline Conflicts Resolution and Efficient Hardware Utilization," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 5454–5467, Dec. 2020.

[14] T. Xie, B. Li, M. Yang, and Z. Yan, "Memory compact high-speed QC-LDPC decoder," Oct. 2017, pp. 1–5.

[15] S. Nimara, O. Boncalo, A. Amaricai, and M. Popa, "FPGA architecture of multi-codeword LDPC decoder with efficient BRAM utilization," Apr. 2016, pp. 1–4.

[16] B. Wang, J. Kang, and Y. Zhu, "Performance Balanced General Decoder Design for QC-LDPC Codes Using Vivado HLS," Jun. 2021, pp. 26–30.

[17] Y. Delomier, B. Le Gal, J. Crenne, and C. Jego, "Model-Based Design of Flexible and Efficient LDPC Decoders on FPGA Devices," *Journal of Signal Processing Systems*, vol. 92, no. 7, pp. 727–745, Jul. 2020.

[18] A. Amaricai, D. Stein, and O. Boncalo, "Generalized Very High Throughput Unrolled LDPC Layered Decoder," Nov. 2020, pp. 1–4.

[19] S. Kumawat, R. Shrestha *et al.*, "High-Throughput LDPC-Decoder Architecture Using Efficient Comparison Techniques," *IEEE Transactions on Circuits and Systems I*, vol. 62, no. 5, pp. 1421–1430, May 2015.

[20] M. Weiner, B. Nikolic, and Z. Zhang, "LDPC decoder architecture for high-data rate personal-area networks." Rio de Janeiro, Brazil: IEEE, May 2011, pp. 1784–1787.

[21] S. Mhaske, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "FPGA-Based Channel Coding Architectures for 5G Wireless Using High-Level Synthesis," *International Journal of Reconfigurable Computing*, vol. 2017, pp. 1–23, 2017.

[22] S. Mhaske, D. Uliana, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "A 2.48Gb/s FPGA-based QC-LDPC decoder: An algorithmic compiler implementation," Sep. 2015, pp. 88–93.