# PHD: Parallel Huffman Decoder on FPGA for Extreme Performance and Energy Efficiency

Yunkun Liao[1,2,3], Jingya Wu[1](✉), Wenyan Lu[1,4], Xiaowei Li[1,3], Guihai Yan[1,4](✉)
{liaoyunkun20s, wujingya, luwenyan, lxw, yan}@ict.ac.cn
State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences[1]
University of Chinese Academy of Sciences[2], Zhongguancun Laboratory[3], YUSUR Technology Co., Ltd.[4]
Beijing, China

## ABSTRACT

Huffman decoding is crucial in data compression, and the self-synchronization-based parallel decoding algorithm enables subsequence-level parallelism. This paper introduces PHD, the first accelerator designed for self-synchronization-based parallel Huffman decoding on a Field-Programmable Gate Array (FPGA). Designing PHD poses challenges, including managing fine-grained parallelism, addressing limited on-chip memory, and handling inter-codeword dependency. PHD incorporates bit-level, subsequence-level, and tile-level parallelism, utilizes hybrid memory to store the codebook efficiently, and introduces the ONCE MORE optimization to reduce decoding loop iterations. Experimental results demonstrate that PHD outperforms the state-of-the-art GPU-based baseline regarding latency (9.4X to 12.8X reduction) and energy consumption (12.4X to 18.2X reduction).

## KEYWORDS

Huffman Decoder, Accelerator, FPGA, Parallel Algorithm

## 1 INTRODUCTION

Data compression accounts for 25% of the well-known "datacenter tax" [1]. Offloading data compression from CPU becomes urgent with the continuous growth of data volume. Huffman coding is crucial in many data compression algorithms, such as JPEG and DEFLATE. A profiling[1] conducted on the Lempel-Ziv-Storer-Szymanski compression algorithm reveals that 77.8% and 99% of the processing time are dedicated to Huffman encoding and decoding, respectively. This paper accelerates Huffman decoding, as decoding poses greater challenges for parallelization than encoding.

Acceleration comes from exploiting parallelism in decoding. Huffman decoding translates binary codewords ("01011...") into textual symbols ("ab..."). Parallelizing the decoding of "0" and "1" is feasible by looking up codebook [2, 3], achieving bit-level parallelism. However, parallelizing the decoding of "01" and "011" is hard because the codeword boundary of a decoded symbol depends on its predecessors. For example, the decoding of "011" cannot begin until the decoding of "01" is completed since the starting bit position of "011" is undetermined. Fortunately, most Huffman codes have the self-synchronization property [4]. This property allows for dividing long codewords into multiple short subsequences at the correct bit boundaries, achieving subsequence-level parallelism.

Previous studies have implemented self-synchronization-based parallel decoding on GPU [5–7]. However, it is essential to note that while the GPU-based implementation offers vast threads for parallelizing, it is not a universal solution for all scenarios. In-network computing (NetCompute) is an emerging area that advocates computing within the network. We consider a NetCompute scenario where a data compression accelerator performs compression and decompression for outgoing and incoming network packets [8, 9], reducing the communication cost of distributed neural network training. The compression accelerator wants to invoke the parallel decoding kernels from a GPU if we assume the Huffman decoding is the performance bottleneck. The pertinent codewords should be in the global memory of the GPU to execute the parallel decoding kernels, which involves copying codewords back and forth in the global memory of the GPU. This memory copy process also involves allocating GPU memory for codewords and auxiliary metadata. When profiling the previous GPU-based implementations [5, 7], we find that the overall latency of the memory copy process exceeds the raw decoding latency for a network packet of 4KB. Furthermore, GPU is known to be power-hungry due to off-chip memory access, and an accelerator can optimize the energy efficiency by reducing off-chip DRAM accesses.

Adapting the parallel Huffman decoding algorithm from a GPU to a domain-specific accelerator is highly desirable. Previous Huffman decoding accelerators [2, 3] only utilize bit-level parallelism instead of the self-synchronization property to decode multiple codewords in parallel. Therefore, this work investigates how to design an accelerator for the self-synchronization-based parallel Huffman decoding algorithm.

We choose an FPGA platform for its reconfigurability, but we run into three challenges when adapting the parallel decoding algorithm from GPU to FPGA. Firstly, abundant achievable fine-grained parallelism is essential for FPGA to deliver superior performance despite its lower clock frequency than the GPU counterparts. Our experimental results (Sec. 4.2) show that the state-of-the-art bit-parallel FPGA implementations [3, 10] fail to outperform GPU implementations [5, 7]. Secondly, while an FPGA provides on-chip memory to reduce off-chip memory accesses, the capacity of its on-chip memory is limited. Decoding codebook is required for bit-level parallelism. Storing the entire regular decoding codebook in the on-chip memory proves infeasible for Huffman codes with long code length. Storing the entire codewords in the on-chip memory is also infeasible, and only a fixed-size codeword tile can be stored in

[1]https://github.com/hzxa21/15618-FinalProject



*DAC '24, June 23–27, 2024, San Francisco, CA, USA*

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3655967

the on-chip memory. The GPU decoders can randomly access the entire codewords, but our FPGA decoders are restricted to partial codewords. Thirdly, the self-synchronization property eliminates codeword dependencies between subsequences, but the codeword dependencies still exists within a subsequence. Consequently, the decoding loop within a subsequence cannot be pipelined.

To address the above challenges, we propose PHD, the first self-synchronization-based parallel Huffman decoding accelerator. Our primary technical contributions are summarized below:

- We map the self-synchronization-based parallel Huffman decoding to a custom accelerator called PHD, achieving bit-level and subsequence-level parallelism simultaneously.
- We identify the short codewords as the common cases that should be fast. PHD stores the small primary codebook in on-chip BRAM and the large secondary codebook in off-chip DRAM. Therefore, the frequent codewords hit in the on-chip primary codebook to ensure low query latency. PHD uses ONCE MORE optimization to reduce the decoding loop iteration count by decoding the codeword window once more in one iteration because a codeword window may contain multiple short codewords.
- We observe no dependencies between subsequences from consecutive codeword tiles in the intra-sequence synchronization stage, and the dependencies between consecutive codeword tiles can be solved by resynchronizing in the inter-sequence synchronization stage. PHD uses remember-and-forward technique to solve the challenge of limited on-chip memory for codewords and leverages tile-based pipeline to improve performance.
- We implement, evaluate, and compare PHD against prior GPU-based and FPGA-based baselines. PHD has lower decoding latency and energy consumption than the GPU baselines for the evaluated cases.

## 2 BACKGROUND

This section exemplifies the parallel Huffman decoding.

### 2.1 Huffman Coding

As shown in Fig. 1(a) and (b), Huffman coding encodes frequent symbols ($x$) to shorter codewords ($h(x)$) and less frequent symbols to longer codewords. As shown in Fig. 1(c), Huffman decoding converts the codewords back to the original symbols.

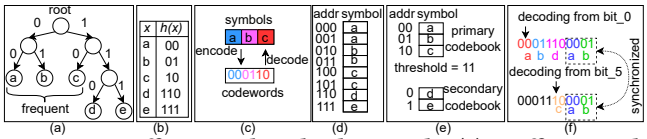



**Figure 1: Huffman coding backgrounds: (a) Huffman coding tree; (b) Huffman coding table; (c) Huffman encoding/decoding; (d) Regular codebook; (e) Compact codebook; (f) Self-synchronization property of Huffman code.**

### 2.2 Regular Codebook and Compact Codebook

Decoding can be implemented by traversing the Huffman coding tree from root to leaf in a bit-serial manner. Based on the Huffman tree in Fig. 1(a), decoding "110" requires three iterations from root to leaf "d". As shown in Fig. 1(d), a regular codebook can be constructed from the Huffman coding table. Based on the regular codebook, decoding "110" only needs to read the sixth ("0b110"=6) element of the regular codebook, which serves as the foundation of bit-parallel decoding. If the maximum codeword length of a Huffman code is $max(|h(x)|)$, the size of its regular codebook is $2^{max(|h(x)|)}$ bytes. For HTTP/2 HPACK Huffman code [11], its $max(|h(x)|)$ is 30, and the size of its regular codebook is 2GB. For widely used canonical Huffman codes, a compact codebook can be used to reduce the codebook size [6]. As shown in Fig. 1(e), a threshold is selected to split the regular codebook into one primary codebook for "a", "b", and "c" and one secondary codebook for "d" and "e". Previous research [6] uses the compact codebook to reduce global memory consumption of GPU, but this study further combines this technique with layered memory hierarchy of accelerator.

### 2.3 Parallel Huffman Decoding

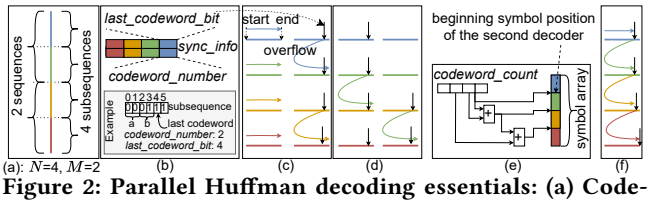



**Figure 2: Parallel Huffman decoding essentials: (a) Codewords splitting; (b) Synchronization information; (c) Phase_1; (d) Phase_2; (e) Phase_3; (f) Phase_4. Different colors stand for different subsequences and corresponding decoders.**

Decoding codewords cannot be directly parallelized because the codeword bit boundaries of continuous symbols are correlated. Only if we know the bit boundaries of different symbols in advance, we can split the codewords into multiple small subsequences and decode these subsequences in parallel. Previous research has pointed out that most Huffman codes have the self-synchronization property [4], as shown in Fig. 1(f). One decoder decodes the codewords from the first bit while the other decoder decodes from the sixth bit, ultimately achieving synchronization between the two decoders at the last two symbols.

Parallel Huffman decoding algorithm has been proposed [5–7, 12] based on the self-synchronization property. The size of each subsequence is $W$ bits and the original codewords are divided into $N$ subsequences as shown in Fig. 2(a). Furthermore, $B$ subsequences are grouped into one sequence. Thus, there are $M$ sequences ($M = \frac{N}{B}$). Each subsequence is associated with its synchronization information (*sync_info*), including *last_codeword_bit* and *codeword_number*. The *last_codeword_bit* and the *codeword_number* represent the position of the first bit of last possible codeword and the number of unbroken symbols in the respective subsequence, respectively. The parallel decoding algorithm includes four phases:

- Phase_1 applies intra-sequence synchronization. Shown in Fig. 2(c), $N$ decoders first decode their allocated subsequences and initialize their *sync_info* concurrently. Then, each decoder continually overflows from the *last_codeword_bit* of the subsequence in the last round iteration to the neighbor subsequence. The decoder stops when the neighbor subsequence is out of the sequence boundary. If the *last_codeword_bit* in a subsequence detected by the current decoder matches the one detected by the other decoder of the same sequence, the current decoder achieves synchronization point and also stops.

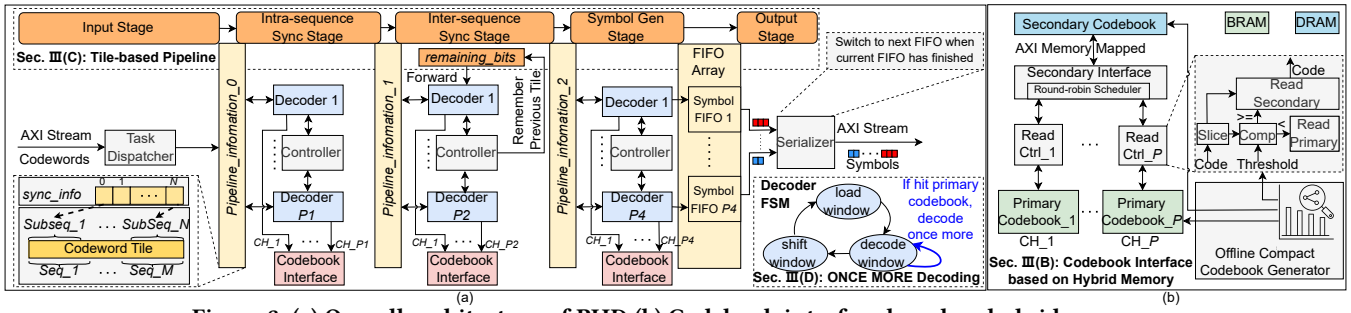



**Figure 3: (a) Overall architecture of PHD (b) Codebook interface based on hybrid memory**

- Phase_2 applies inter-sequence synchronization. Although the decoders achieve synchronization in their sequence group, the first decoder of each sequence starts from a possible wrong bit position in the whole codewords, excluding the first sequence. As shown in Fig. 2(d), there are $(M-1)$ decoders overflowing from the last subsequence of a previous sequence to the last subsequence of its neighbor sequence until all the decoders achieve synchronization.
- Phase_3 applies prefix summation. After Phase_1 and Phase_2 finish, the *sync_info* of each subsequence reveals the right bit boundary and the number of symbols of each subsequence. As shown in Fig. 2(e), adopting prefix summation determines the beginning byte position of each subsequence in the output symbol array.
- Phase_4 applies final symbol generation. There are $N$ decoders that decode their subsequence concurrently according to the bit boundary specified by its *sync_info* and the *sync_info* of previous subsequence. The generated symbols are concurrently written into the output array according to the positions determined by Phase_3.

The decoding behavior of Phase_1 and Phase_2 is the computation overhead to identify the correct bit boundaries for Phase_4. However, the parallelisms in Phase_1, Phase_2, and Phase_4 reduce the overall latency at the expense of multiple decoders.

# 3 PHD DESIGN

## 3.1 Overall Architecture

**Top-level Interface** Figure 3(a) illustrates the overall architecture of PHD. The input codewords and output symbols are transmitted by the standard AXI Stream interfaces. The ready-valid protocol of AXI Stream provides flow control capability for PHD, considering that PHD cannot consistently accept a new beat of codewords each clock cycle. Moreover, standard AXI Stream interface facilitates seamless integration of PHD with other modules for developers.

**Datapath** In Fig. 3(a), we partition the datapath of PHD into five stages based on the parallel decoding algorithm mentioned in Sec. 2.3. The stages are connected with pipeline register arrays (i.e., *pipeline_information*) or FIFOs.

(1) The input stage initializes the *pipeline_information_0* and dispatches a new task. We assume each subsequence has $W$ bits. The *pipeline_information* can only hold $N$ subsequences due to the limited on-chip memory, defined as the codeword tile. Furthermore, the tile has $M$ sequences, each containing $B$ subsequences. It is important to note that the codeword tile may be partially filled. Therefore, the task metadata includes the count of valid subsequences, valid sequences, and the final valid bit position within the last valid subsequence.

(2) The intra-sequence synchronization, inter-sequence synchronization, and symbol generation stages implement Phase_1, Phase_2, and Phase_4, respectively. There are *P1*, *P2* and *P4* decoders in Phase_1, Phase_2 and Phase_4. Constrained by the codeword tile organization, *P1*, *P2*, and *P4* are less or equal to $N$, $M$, and $N$, respectively. We omit Phase_3 in the datapath of PHD because Phase_3 is redundant for a streaming accelerator. Specifically, while the output symbols are stored in addressed memory for CPU- and GPU-based implementations, the output symbols of PHD are transmitted via a stream. There is no address associate with stream data. Consequently, calculating the starting address of output symbol streams for each subsequence becomes meaningless. Instead, we arrange the symbol FIFOs between the decoders of the symbol generation stage and the serializer.

(3) The final serializer reads the symbol FIFOs in turn and outputs symbols in AXI Stream. The serializer is required to maintain the correct order of symbols because the finish time of different decoders may be out of order. Therefore, the serializer switches to the next symbol FIFO until all the expected symbols of the current symbol FIFO have been outputted. The number of expected symbols comes from the *sync_info* of *pipeline_information_2*.

## 3.2 Codebook Interface based on Hybrid Memory

Codebook is the foundation of bit-parallel Huffman decoding. The codebook interface should well balance the on-chip memory consumption and access latency. Cramming the entire regular codebook into BRAM is not feasible due to the limited on-chip BRAM capacity in FPGA devices. For instance, the maximum length of Huffman code exceeds 30 bits in the case of HTTP/2 HPACK [11], yielding a codebook size of 2GB according to Sec. 2.2. Such capacity requirement is approximately two orders of magnitude larger than what typical FPGAs can accommodate. As a result, only a portion of the codebook can be cached in on-chip BRAM, while the remainder needs to be stored in off-chip DRAM.

Can we design a BRAM-friendly codebook interface with tolerable performance loss? Our insight is that the symbols of the smaller codeword length are more common in Huffman code. For example, a symbol with a code length less than or equal to eight has a 99% chance of appearing if the symbols obey the HPACK Huffman code distribution. We propose our codebook interface based on the

compact codebook with hybrid memory based on the principle of making common cases fast.

As shown in Fig. 3(b), our solution includes the offline compact codebook generator and the codebook interface module. The offline compact codebook generator computes the partition threshold value for a given Huffman code according to the symbol coverage rate and the BRAM constraint. The binary length of the threshold value defines the depth of the primary codebook and primary code length. The symbol coverage rate is the statistical percentage of symbols the primary codebook can cover. The primary and secondary codebooks are generated after the threshold is determined.

The codebook interface module has $P$ separate channels to support concurrent queries for the primary codebook (BRAM) but one shared channel for the secondary codebook (DRAM). $P$ copies of the primary codebook are stored in BRAM to assure low query latency for most symbols. First, the read controller slices the query code to obtain a prefix code based on the primary code length when one query comes. Then, the read controller checks whether the query hit the primary codebook by comparing the value of the prefix code with the threshold value. Last, the read controller uses the prefix code to query the primary codebook if the value of prefix code is less than the threshold. Otherwise, the read controller sends the query to the secondary interface. Querying the secondary codebook is the uncommon case. Thus, we adopt a design that $P$ queries share one copy of secondary codebook in a round-robin order. This shared design is resource-friendly because only one AXI master interface is required for reading DRAM. Although this shared design increases the latency for querying the symbols in secondary codebook, the uncommon cases only slightly slow the overall decoding.

The drawback is that the compact codebook can only be constructed for canonical Huffman code. However, most Huffman codes are canonical for data compression [6]. Satyabrata [3], the state-of-the-art FPGA-based canonical Huffman decoding accelerator, uses more comparison logic to save the codebook storage. Our codebook interface requires only one comparison per query, while Satyabrata requires $max(|h(x)|)$ comparisons per query.

## 3.3 ONCE MORE Decoding

The decoders of PHD in different stages perform similar operations by traversing their codeword subsequences and querying the codebook interface to translate codewords into symbols. As shown in Fig. 4(a), previous GPU- and FPGA-based implementations adopt the naive decoding approach. This approach involves using a decoding window of a fixed width, which is set to be greater than or equal to the maximum code length ($max(|h(x)|)$) of the Huffman code being used. The decoding process consists of a loop that iteratively decodes subsequences. In each iteration, there are three main steps: loading the codeword bits into the empty slots of the decoding window (load_window()), slicing the decoding window to obtain the prefix code with a length of $max(|h(x)|)$, querying the codebook interface to retrieve the corresponding symbol (dec_window()), and shifting out the used bits from the decoding window based on the code length of the returned symbol (shift_window()).

Naive decoding is inefficient. As shown in Fig. 4(b), the overall loop latency of naive decoding is the product of loop iteration count and per-iteration latency. One possible optimization is pipelining

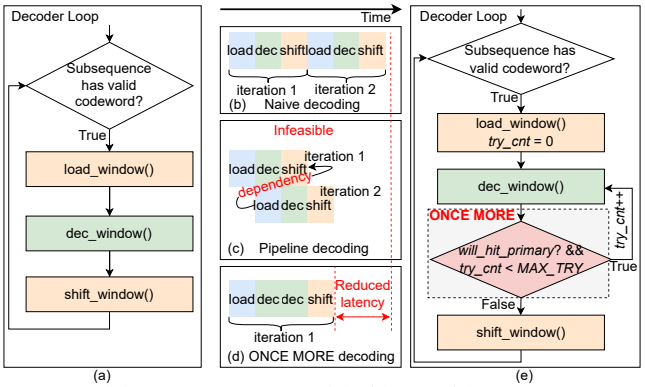



**Figure 4: (a) Naive decoding. (b), (c) and (d) Execution diagram of naive, pipeline and ONCE MORE decoding. (e) ONCE MORE decoding.**

the load_window(), dec_window() and shift_window(), as shown in Fig. 4(c). However, pipeline optimization is infeasible because there exists an inevitable dependency between iterations. Our insight is that the decoding window has a high probability of containing multiple symbols whose code length is less than the primary code length. As shown in Fig. 4(d), our ONCE MORE optimization tries once more decoding in each iteration to decrease the loop iteration count, thus reducing overall loop latency.

The critical problems encountered in ONCE MORE optimization are the criteria of trying decoding and the maximum count of trying decoding (*MAX_TRY*). As shown in Fig. 4(e), ONCE MORE optimization only tries another decoding when the number of remaining valid bits in the decoding window is less than the primary code length, and possible decoding hits the primary codebook. This criterion guarantees that the remaining bits of the decoding window contain at least one complete codeword. The ONCE MORE optimization does not try another decoding when the possible decoding does not hit the primary codebook because it cannot infer that the remaining valid bits of the decoding window are enough for the symbols of secondary codebook. An incorrect decoding result is returned if we query the codebook interface using the decoding window with invalid bits. The ratio between the primary code length and the width of the decoding window first decides the upper bound of *MAX_TRY*. For example, the *MAX_TRY* is less or equal to 3 ($\lceil \frac{32}{8} \rceil - 1$) if the primary code length is 8 bits and the width of the decoding window is 32 bits. The target clock frequency then decides the *MAX_TRY* because the critical path grows longer as the number of consecutive attempts increases.

## 3.4 Tile-based Pipeline

Partitioning variable-length codeword stream into multiple fixed-capacity codeword tiles in PHD poses a challenge for maintaining bit boundary alignment. Previous GPU-based implementations do not encounter this challenge because all the codewords and *sync_infos* are stored in global memory and accessed by addresses. However, we need to split the input codeword stream into multiple codeword tiles due to the on-chip memory capacity constraint of PHD. The codewords are loaded from the input stream to the codeword tile of *pipeline_information_0*. Then, the codeword tile flows along the decoding stages. Only the first subsequence of the first codeword tile starts from the correct bit boundaries. The first subsequence of the remaining codeword tiles may not start from

the correct bit boundaries. There are dependencies between consecutive tiles. Ignoring the dependencies between consecutive tiles leads to incorrect decoding results of the remaining codeword tiles.

One alternative approach to avoid multiple-tile scenarios is to store the entire codewords in off-chip DRAM instead of BRAM. First, this solution leads to performance degradation since accessing off-chip DRAM is considerably slower than on-chip memory. Second, this solution reduces achievable codeword reading parallelism due to limited DRAM channels and increases the resource consumption of AXI memory interfaces.

We observe no dependencies between codeword tiles at the subsequence level in the intra-sequence synchronization stage. The tile-level dependency appears until inter-sequence synchronization stage. Hence, we can remember and forward the remaining bits of the previous tile to the subsequent tile in the inter-sequence synchronization stage. This forms the basis of our tile-based pipeline dataflow for multiple-tile scenarios. As the example shown in Fig. 5, the input codeword stream consists of two tiles named Tile_0 and Tile_1. The Tile_0 is first dispatched into PHD. After inter-sequence synchronization of Tile_0 finishes, the remaining valid bits of the last subsequence of Tile_0 are remembered in the *remaining_bits* register. When the inter-sequence synchronization of Tile_1 starts, the *remaining_bits* is forwarded to the first decoder in this stage. The first decoder leverages the remaining_bits to resynchronize the first sequence of Tile_1.

PHD pipelines decoding stages of Tile_0 and Tile_1, provided there are no structure hazards. The benefit of pipelined execution is illustrated in Fig. 5. The intra-sequence synchronization of Tile_1 can be immediately launched after the intra-sequence synchronization of Tile_0 is completed. This tile-level parallelism significantly enhances performance in scenarios where the input codeword stream consists of multiple tiles.

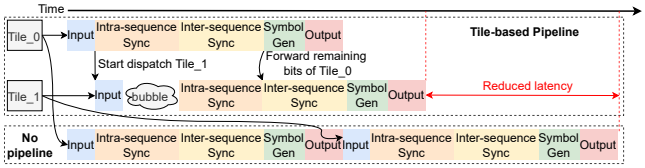



**Figure 5: Tile-based pipeline dataflow**

# 4 EXPERIMENTAL EVALUATION

## 4.1 Methodology

We implement PHD using the Xilinx High-Level Synthesis C++ language. The Xilinx Vitis 2022.1 is used for cycle-accurate simulations and Verilog generation. The Xilinx Vivado 2022.1 is used to assess timing, resource utilization, and power consumption. The Xilinx U200 board is selected as the FPGA target. Finally, the clock frequency is set to 250 MHz.

We select Huffman code for HTTP/2 HPACK [11] as our case study. The test symbols are generated randomly according to HPACK Huffman code distribution. The primary code length is 8 bits and the threshold is 254. Both the input and output stream interfaces are configured to be 512 bits wide. We set the default size and number of subsequences to 512 bits and 64, respectively. As a result, the default codeword tile size is set to 4K bytes. The decoding window width is set to 32 bits. The default design parameters of *P1*, *P2*, and *P4* are 64, 8 and 16 according to our design space exploration results. The default configuration utilizes 1052 BRAM_18Ks (24.4%), 227992 FFs (9.6%), and 918358 LUTs (77.7%).

We also implement three FPGA baselines. The first FPGA baseline implements bit-parallel decoding based on a compact codebook to emulate the decoding behavior of Vitis Library [10] and Satyabrata [3]. It is noted that Satyabrata is the state-of-the-art FPGA solution, which utilizes LUT-optimized indexing to implement bit-parallel decoding. Our bit-parallel FPGA baseline does not implement the LUT-optimized indexing of Satyabrata, because our compact codebook design already obtains smaller average codebook access latency than the LUT-optimized indexing. The second and third FPGA baselines implement PHD separately without the ONCE MORE and tile-based pipeline optimization, respectively.

On CPU and GPU platforms, we implement three baselines. The first baseline, LS-HPACK[2], involves using a performance-optimized HPACK library on a CPU platform equipped with an Intel Core-i5 12400 processor. For the GPU platform, we select two baselines: Weißenberger [5] and Rivera [7]. Both Weißenberger and Rivera are running on NVIDIA GeForce RTX 4090 GPU. When the size of symbols is 4K bytes, there are 64 available decoding threads in the four algorithm phases for Weißenberger and Rivera. As the size of symbols increases, the implementations of Weißenberger and Rivera increase the number of CUDA threads. Specifically, Weißenberger and Rivera have 128, 256, 448 and 832 parallel decoding threads when the size of symbols are 8K, 16K, 32K and 64K bytes.

For fair comparison with PHD, the measured decoding latency of the GPU baselines only includes raw decoding latency, excluding latency of the memory copy process. We are unable to measure the decoding latency of Rivera for all symbol sizes due to issues with its opensource implementation. Specifically, the implementation gets stuck when the size of symbols exceeds 16K bytes. We use nvidia-smi tool to monitor the GPU power consumption. The standby power is obtained from the power output of nvidia-smi when there is no load on the GPU. We calculate the average power output of nvidia-smi during the execution of the GPU baselines to obtain the runtime power. We collect data from nvidia-smi at intervals of $1\mu s$.

## 4.2 Latency and Energy

The decoding latency comparison between PHD and the other baselines is illustrated in Fig. 6. We omit the bit-parallel FPGA and LS-HPACK baselines to make comparisons of other baseline results easier to distinguish in Fig. 6. Compared with LS-HPACK, PHD achieves a latency reductions ranging from 24.5X to 60.6X. PHD shows its superiority compared with GPU baselines, delivering latency reduction of 5.6X to 25.4X and 9.4X to 12.8X over Weißenberger and Rivera, respectively. The performance improvement of PHD decreases with the increasing size of symbols because PHD only has fixed number of parallel decoders in each stage. However, the parallel decoder number of Weißenberger and Rivera scale with the size of symbols. Thus, when dealing with huge data volume, GPU-based implementations outperform PHD due to the larger parallelism offered numerous GPU CUDA cores. We conclude the weakness of PHD under huge data volume to the low logic density of FPGA, not to the architecture of PHD.

PHD achieves a substantial latency reduction ranging from 18.5X to 57.5X compared with previous bit-parallel FPGA baseline. This is

[2]https://github.com/litespeedtech/ls-hpack

attributed to the subsequence-level parallelism, tile-based pipeline, and ONCE MORE optimization techniques employed in PHD. However, PHD is resource-hungry because the number of decoders consumed by the PHD is 88 times the bit-parallel FPGA baseline.

The ONCE MORE optimization tries at most three times in current configuration, contributing latency reductions of 3.0X to 3.3X. The tile-based pipeline contributes latency reductions of 1.0X (only one tile) to 2.3X. As the codeword volume increase, the tile-based pipeline improves the performance of PHD even more.

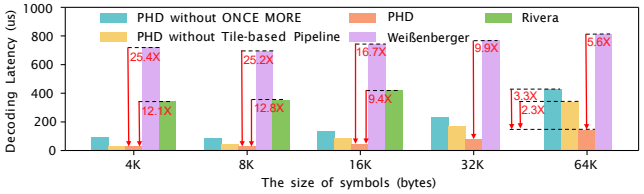



**Figure 6: Decoding latency**

Figure 7 shows the energy consumption results for the five sizes of symbols. Energy consumption is the integral of power and decoding latency. The on-chip power of PHD is obtained from the implementation report generated by Vivado. We compute the GPU decoding power by comparing the runtime power with the standby power of the GPU. We observe energy reduction when comparing PHD to GPU baselines. PHD reduces energy by 8.3X to 29.6X over Weißenberger and 12.4X to 18.2X over Rivera. We credit the energy reduction of PHD with reducing off-chip memory access for codebooks and codewords. We find that increasing the CUDA threads of the GPU baselines can decrease the latency when the codeword volume increases but the runtime power also increases.

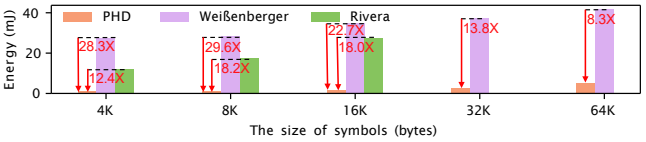


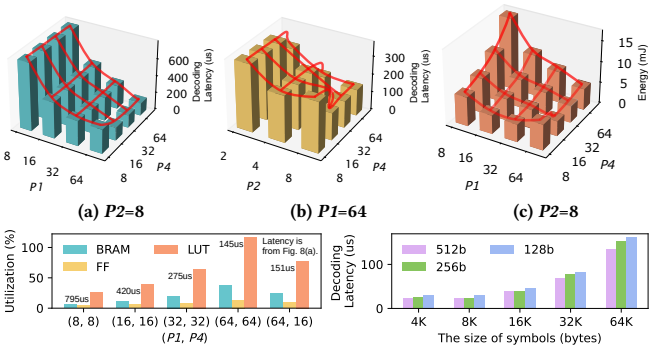

**Figure 7: Energy consumption**

## 4.3 Design Space Exploration

The design parameters of *P1*, *P2*, and *P4* control the subsequence-level parallelism in intra-sequence synchronization, inter-sequence synchronization, and symbol generation stages. The design parameter of subsequence size controls the workload capacity of each decoder. Figure 8 shows the design space exploration results regarding decoding latency, energy, and resource utilization. Large *P1* and *P2* parallelism are crucial for overall performance, while a medium value for *P4* can decrease energy consumption and resource utilization with minimal performance loss. Smaller subsequence size leads to a little longer latency due to higher decoder setup latency.

## 4.4 Discussion of ASIC-based PHD

Limited by the intrinsic low logic density of FPGA, the maximum subsequence-level parallelism of PHD is much lower than GPU-based baselines. The design of PHD can be migrated from FPGA to ASIC for higher parallelism. We calculate the ASIC area of PHD according the area data of PIFO project (16 nm) [13]. Based on the relative resource usage between FPGA-based PHD and FPGA-based PIFO, the estimated ASIC area of PHD can be calculated as 5.49 $mm^2$ (16 nm) and 0.54 $mm^2$ (5 nm), while the area of NVIDIA 4090 is 609 $mm^2$ at 5 nm. The maximum parallelism of ASIC-based PHD




**(d) *P2*=8, the configuration of (64, 64) is infeasible due to LUT capacity.** **(e) Under different subsequence sizes, (*P1*, *P2*, *P4*) = (64, 8, 16)**

**Figure 8: Design space exploration results. The sizes of symbols in (a), (b), (c) and (d) are all 64K bytes.**

increases by 1127X than FPGA-based PHD if PHD has the same ASIC area as NVIDIA 4090. ASIC-based PHD can also have higher clock frequency to scale the processing speed.

# 5 CONCLUSION

We presented PHD, the first accelerator targeting at self-synchronization-based parallel Huffman decoding. PHD realizes bit-level, subsequence-level and tile-level parallelism. In order to solve the limited on-chip storage for regular codebook, PHD implements a compact codebook interface based on hybrid memory. Finally, PHD proposes ONCE MORE optimization to accelerate subsequence decoding. Experimental results demonstrate the benefits of PHD compared with previous FPGA-based and GPU-based solutions.

# ACKNOWLEDGMENTS

This paper is supported in part by the National Natural Science Foundation of China (NSFC) under grant No. 62002340, 61872336, and No.62090020, in part by the Strategic Priority Research Program of the CAS under Grant No. XDB44030100, and in part by Youth Innovation Promotion Association CAS No. Y201923. Part of this work is supported by the internship program of YUSUR Technology Co., Ltd. The corresponding author is Guihai Yan and Jingya Wu.

# REFERENCES

[1] Kanev, Svilen, et al. "Profiling a warehouse-scale computer." In ACM/IEEE ISCA 2015.
[2] Z. Aspar, et al, "Parallel Huffman decoder with an optimized look up table option on FPGA." In IEEE TENCON, 2000.
[3] Sarangi, Satyabrata, et al. "Energy-efficient canonical Huffman decoders on many-core processor arrays and FPGAs." Integration 88 (2023): 156-165.
[4] Ferguson, Thomas, et al. "Self-synchronizing huffman codes (corresp.)." IEEE Transactions on Information Theory 30.4 (1984): 687-693.
[5] Weißenberger, André, et al. "Massively parallel Huffman decoding on GPUs." In ACM ICPP, 2018.
[6] Yamamoto, Naoya, et al. "Huffman coding with gap arrays for GPU acceleration." In ACM ICPP, 2020.
[7] Rivera, Cody, et al. "Optimizing huffman decoding for error-bounded lossy compression on gpus." In IEEE IPDPS, 2022.
[8] Li, Youjie, et al. "A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks." In IEEE MICRO, 2018.
[9] Ren, Qingqing, et al. "A High-performance FPGA-based Accelerator for Gradient Compression." In IEEE DCC, 2022.
[10] AMD Xilinx. "Vitis Accelerated Libraries." https://www.xilinx.com/products/design-tools/vitis/vitis-libraries.html.
[11] Peon, Roberto, et al. HPACK: Header compression for HTTP/2. No. rfc7541. 2015.
[12] Klein, Shmuel Tomi, et al. "Parallel Huffman decoding with applications to JPEG files." The Computer Journal 46.5 (2003): 487-497.
[13] Sivaraman, Anirudh, et al. "Programmable packet scheduling at line rate." Proceedings of the 2016 ACM SIGCOMM Conference. 2016.