Indexed Color History Many-Core Engines for Display Stream Compression Decoders

Shifu Wu and Bevan Baas
Department of Electrical and Computer Engineering
University of California, Davis

***Abstract*—This paper describes and compares 9 many-core designs and software implementations of the Indexed Color History (ICH) module, which is part of VESA Display Stream Compression (DSC) decoders. The smallest design is mapped to only 8 small processors. Other designs use a new algorithm to split the ICH table update process into index update and entry update tasks. This algorithm is implemented with a variety of parallel and optimized architectures to provide a range of throughputs and energy efficiencies utilizing from 9 to 53 processors. The proposed ICH designs deliver frame rates in 1080p (1920×1080) up to 75, 74, and 38 frames per second (fps) in 4:2:0, 4:2:2, and 4:4:4 modes, while dissipating 15 mJ, 16 mJ, and 30 mJ per frame respectively at 1.75 GHz at 1.1 V. Compared to reference designs implemented on an Intel i7-7700HQ, the proposed designs achieve up to 3.4×, 3.9×, and 5.3× higher frame rates, and up to 177×, 193×, and 261× lower energy per frame in 4:2:0, 4:2:2, and 4:4:4 modes respectively.**

I. Introduction

Video applications have been in increasing demand. For example, the global video conferencing market stood at USD 3.02 billion in 2018 and is projected to reach USD 6.37 billion by 2026 [1]. Video compression is one of the key technologies that will drive the growth of video-based applications.

The VESA Display Stream Compression (DSC) [2] standard offers visually-lossless video compression [3] with low hardware cost and low latency which is essential in many real-time applications. While application-specific integrated circuit (ASIC) DSC codecs [4], [5] achieve high performance for real-time encoding and decoding, software DSC codecs on programmable computational platforms offer greater flexibility and scalability.

We present the design and implementation of the critical Indexed Color History (ICH) module for DSC decoders on a programmable many-core computational platform. Through task-level parallelism, the optimized designs are able to perform real-time decoding on 1080p video streams.

II. ICH and the Many-Core Platform

A. *Indexed Color History (ICH)*

In the context of DSC, every frame is partitioned into one or more *slices* which are regions processed independently. DSC supports both RGB and YCbCr pixel format in 4:4:4 mode, in which a pixel contains 3 *components*. In addition, two chroma-subsampled YCbCr formats are also supported: 1) the 4:2:2 format, in which only the even-position pixels contain chroma components, and 2) the 4:2:0 format, which contains only one chroma component (Cb in even-numbered lines and Cr in odd-numbered lines) in the even-position pixels and no chroma components in the odd-position pixels. In 4:2:0 and 4:2:2 formats, two neighboring pixels are packed into one *container pixel*, resulting in approximately twice throughput. A container pixel has 3 components in 4:2:0 format and 4 components in 4:2:2 format.

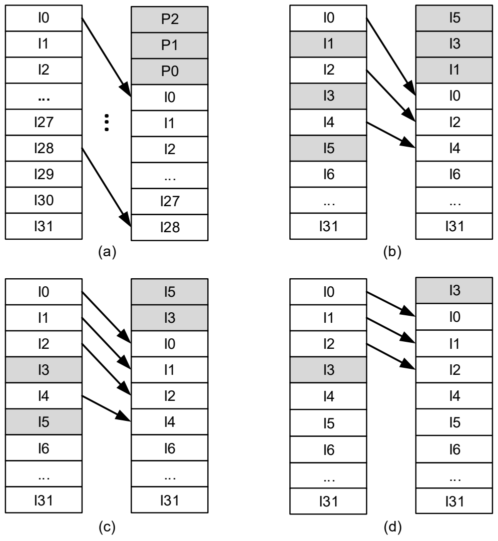

Fig. 1. ICH table update examples: (a) P-mode, (b) ICH-mode with indices (1, 3, 5), (c) ICH-mode with indices (5, 3, 5), and (d) ICH-mode with indices (3, 3, 3).

Three neighboring pixels (container pixels in 4:2:0 and 4:2:2 modes) of the same slice line form a *group*, which is coded using either predictive coding (P-mode) or indexed color history coding (ICH-mode). P-mode codes every group with the quantized residuals, while ICH-mode codes each group with three 5-bit ICH indices.

The ICH module maintains a 32-entry table, which stores one pixel in each location that is addressed by a 5-bit index. The location at address 0 contains the most-recently-used (MRU) pixel. For the first line (and second line in 4:2:0 mode) of each slice, all 32 entries are actual history pixel values, while for non-first line groups the last seven entries point to previous line reconstructed pixels.

The ICH table is updated once per group. In P-mode, the reconstructed pixels of the current group, denoted as P0, P1 and P2 from left to right, enter the top of the ICH table. All other entries are shifted down by 3, as shown in Fig. 1(a). In

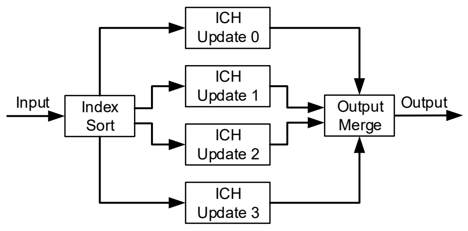

Fig. 2. Dataflow diagram of the *Shift Table Entries* algorithm.

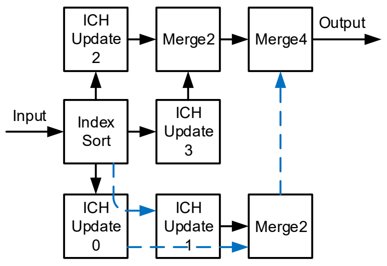

Fig. 3. Processor mapping of the *Shift-Entry* design.

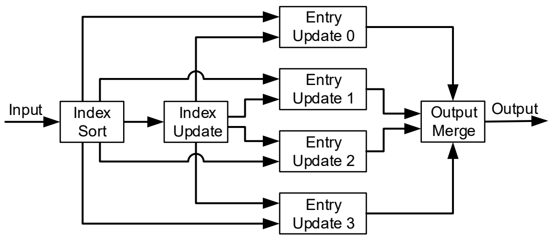

Fig. 4. Dataflow diagram of the *Separate Index and Entry Update* algorithm.

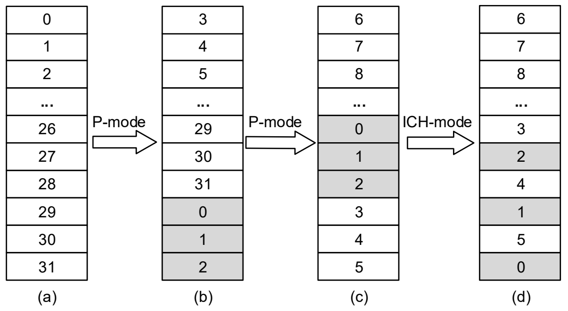

Fig. 5. Example of index table update: (a) initial state, (b) P-mode is selected, (c) P-mode is selected, and (d) ICH-mode is selected with indices (1, 3, 5).

ICH-coded groups, the selected ICH entries are moved to the top of the ICH table. In Fig. 1(b), the selected indices are 1, 3, and 5 for the left, middle and right pixels. Therefore, I5 becomes the MRU, while I3 and I1 are moved to the second and third entries. The original entries I0, I2 and I4 are shifted down by 3, 2, and 1, respectively, while other entries remain unchanged. When the number of unique indices is less than 3, as shown in Fig. 1(b–c), only the last occurrence of a replicated index is counted for an ICH table update.

B. The Many-Core Computational Platform

The KiloCore chip [6] contains 1000 simple RISC-style independent processors connected via a 2-D mesh network, which supports communication between adjacent and distant processors. Each processor occupies 0.055 mm^2 in 32 nm CMOS technology, contains 128×40-bit words of instruction memory and 256×16-bit words of data memory, and operates to a maximum clock frequency of 1.78 GHz [7].

III. Proposed ICH Implementations

This section presents 2 algorithms and 2 optimization methods for the ICH module. Nine designs are implemented to evaluate these algorithms and methods.

A. The Shift Table Entries Algorithm

Figure 2 shows the dataflow diagram of the *Shift Table Entries* algorithm. It performs ICH table update by shifting the ICH table entries. This algorithm is composed of three tasks:

1) Index Sort: This task sorts the ICH indices and finds the number of unique indices. It then distributes the sorted indices and other inputs for ICH table update.

2) ICH Update: This task updates the ICH table using the approach described in Section II. By exploiting component-level parallelism, this task is further partitioned into 4 parallel subtasks, each of which stores and updates the ICH table of one component.

3) Output Merge: This task merges the decoded pixel values of the 3 or 4 components into a single output stream.

The *Shift-Entry* design is implemented based on this method. Fig. 3 shows the 8-core processor mapping on the many-core platform. Each task/subtask is mapped to one processor, except that the *Output Merge* task is mapped to 3 processors, since each processor has only two input ports and thus can merge only two inputs.

To find the throughput bottleneck, we performed separate simulations on each processor running alone, which can achieve maximum performance since no I/O stalls are caused by other processors. The *ICH Update* processors take 561 cycles on average to process one group, while *Index Sort*, *Merge2* and *Merge4* processors require 90, 6, and 12 cycles, respectively. Therefore, the *ICH Update* processors limit the performance. By running the *Shift-Entry* design on the targeted many-core platform at 1.75 GHz, it achieves 9.1 fps, 9.0 fps, and 4.4 fps in 4:2:0, 4:2:2, and 4:2:4 modes, which is too slow for real-time applications. Due to the data dependency in shifting the ICH table entries, further parallelism on the ICH table update task is not possible with this algorithm.

B. The Separate Index and Entry Update Algorithm

Motivated by the fact that in every group no more than 3 new pixels enter the ICH table, while the majority of ICH entries remain in the table with only their indices changed, we propose the *Separate Index and Entry Update* algorithm which partitions the ICH table update task into index update and entry update tasks. An index table is maintained for ICH indices and an entry table is used to store ICH entries, where the index of each ICH entry is stored in the same location of the index table. The index table is serially updated by looping through the 32 indices, while the entry table is updated by writing 0–3 new pixel values into the appropriate locations,

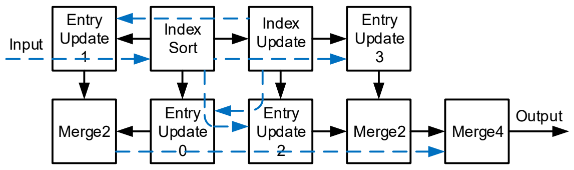

Fig. 6. Processor mapping of the *Split-Index* design.

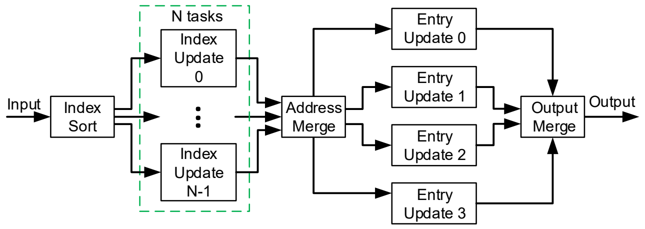

Fig. 7. Dataflow diagram of the *Parallel Index Update* method. For clarity, the connection from index sort to entry update is omitted.

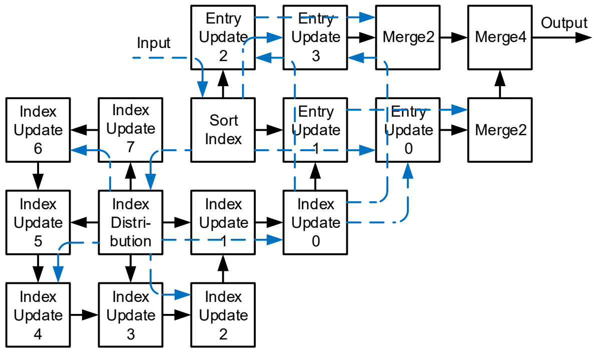

Fig. 8. Processor mapping of the *Parallel-Index-8* design.

the addresses of which are provided by the index update task. The dataflow diagram of this method is shown in Fig. 4.

Figure 5 shows an example of the index table update. The ith word is initialized to i. In this example, the first two groups are P-mode coded, thus the index values of 29, 30, and 31 are updated to 0, 1, and 2, respectively, while the remaining 29 indices are increased by 3, as shown in Fig. 5(b–c). The third group uses ICH-mode coding with indices (1, 3, 5). Therefore, the ICH indices 5, 3, and 1 are updated to 0, 1, and 2, respectively. Index 0 is increased to 3, since it is smaller than all three selected indices. Index 2 and 4 are smaller than two and one of the selected indices, thus they are increased by 2 and 1, respectively. All other indices remain unchanged.

The 9-core *Split-Index* design is implemented using this algorithm, as shown in Fig. 6. The simulation on processors running alone shows that the *Index Update* processor takes 1,154 cycles to process one group, while 68 cycles are needed for the *Entry Update* processor. Therefore, to increase throughput, it is essential to speed up the index update task.

C. Design Optimization I — Parallel Index Update

Unlike shifting ICH table entries, there is no data dependency in the update process between ICH indices. Therefore, the index table update task can be partitioned into multiple parallel subtasks, the output of which are merged later. We use N, a number evenly divisible by 32, to denote the number of index update subtasks. Every subtask maintains a table of $32/N$ indices, resulting in N times speed up compared to the non-parallel index update task. Since one subtask does not contain all indices, the three ICH outputs can contain 0–3 valid addresses. In addition, for every ICH address, there is one and only one valid output in the N subtasks. By setting the invalid outputs to 0, the ICH addresses of the subtasks are merged together using bitwise OR operations.

Figure 7 shows the dataflow of this method. For evaluation purpose, we implemented 5 designs: *Parallel-Index-2*, *Parallel-Index-4*, *Parallel-Index-8*, *Parallel-Index-16*, *Parallel-Index-32*, which correspond to N = 2, 4, 8, 16, and 32, respectively. Each index update subtask is mapped to a different processor. For $N \leq 8$, the ICH addresses are merged in one of the index update processors, which saves an extra merging processor. *Parallel-Index-2* and *Parallel-Index-4* are mapped to 10 and 12 processors, respectively. Fig. 8 shows the processor mapping of *Parallel-Index-8*. Since the index sort task of *Parallel-Index-8* needs 12 output connections (4 for entry update and 8 for index update) but every processor has 8 output ports, it is mapped into 2 processors, where one processor distributes data to the index update processors, resulting in a total of 17 processors. For *Parallel-Index-16*, 4 processors are used for the index sort task, where 1 processor distributes data to entry update processors, 1 processor sorts the indices, and 2 processors distribute data to the index update processors. The results of every 8 index update processors are merged first, which are then merged using an additional processor. As a result, in total 28 processors are required. *Parallel-Index-32* is mapped to 52 processors, out of which 4 processors are used to distribute data to the 32 index update processors and 7 processors merge the results.

D. Design Optimization II — I/O Stall Reduction

During the time that the entry processors are waiting for the ICH addresses, their other input FIFOs keep accumulating data from the index sort processors. On the other hand, increasing the number of parallel index update processors reduces the average execution delay, which means the index sort processors write to the entry update processors faster. In *Parallel-Index-16* and *Parallel-Index-32*, stalls occur in the index sort processors due to full input FIFOs of the entry update processors, which degrades the overall throughput. Two designs *Parallel-Buffer-16* and *Parallel-Buffer-32* are implemented by using a buffer processor between index sort processors and entry update processors in *Parallel-Index-16* and *Parallel-Index-32*, resulting in stall reductions of 6% and 51%, and throughput increases of 2.2% and 8.1%, respectively.

IV. Results and Analysis

The 9 proposed ICH engine designs are evaluated in a cycle-accurate C++ simulator of the KiloCore chip [8]. The simulations are performed at 1.75 GHz with a supply voltage of 1.1 V. Results are obtained from simulating two 1080p frames—a noise image and an image of mandrills. Two reference designs *i7-Shift-Entry* and *i7-Split-Index* implemented

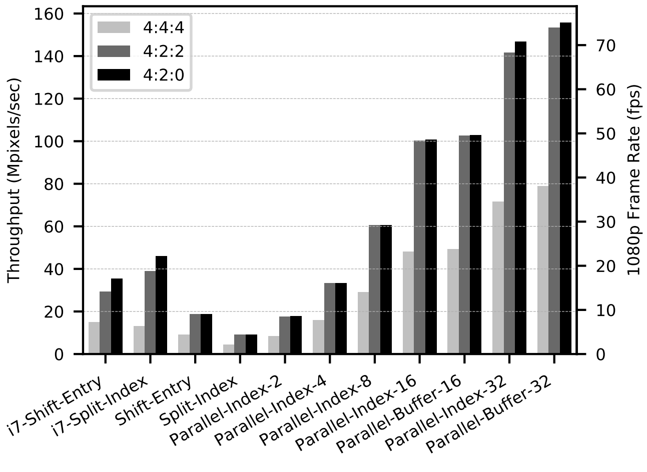

Fig. 9. Throughput and frame rates of the many-core and i7 designs.

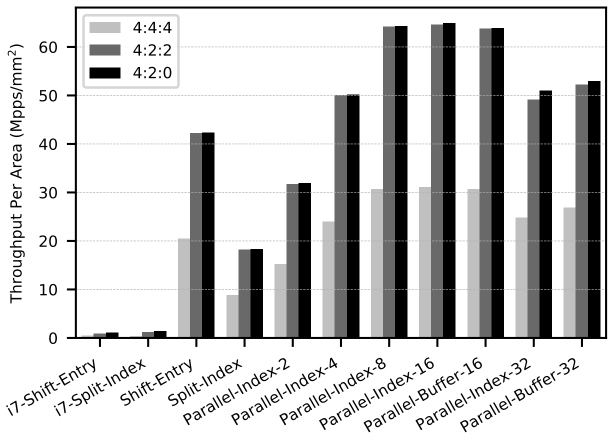

Fig. 10. Throughput per chip area of the many-core and i7 designs.

with similar methods are evaluated on an i7-7700HQ processor using the same test images.

Figure 9 shows the average throughput and the frame rates. The throughput of 4:2:0 and 4:2:2 modes is around twice of 4:4:4 mode in the same design, since a container pixel is processed in the same way as a 4:4:4 pixel. Increasing the number of parallel index update processors results in a near linear increase in throughput. *Parallel-Buffer-32* has the highest throughput of 155.6 million pixels per second (Mpps) in 4:2:0 mode, 153.3 Mpps in 4:2:2 mode and 78.9 Mpps in 4:4:4 mode, which are 75 fps, 74 fps, and 38 fps at 1080p, and **3.4**×, **3.9**×, and **5.3**× higher than the i7 reference designs.

To fairly compare performance, the throughput is normalized by the total chip area used, as shown in Fig. 10. For the i7 reference designs, only the area of 4 cores is counted, which we estimate as 32 mm^2 based on a publicly-available die photo. Our highest normalized throughputs of 64.9 Mpps/mm^2, 64.6 Mpps/mm^2, and 31.0 Mpps/mm^2 in 4:2:0, 4:2:2, and 4:4:4 modes are achieved in *Parallel-Index-16*, and are **45**×, **53**×, and **66**× higher throughput per chip area than the i7 reference designs.

Figure 11 shows the energy consumption to process one 1080p frame. For the i7 designs, we assume the power consumption is half of TDP [9]. After scaling to 32 nm using data from Holt [10], *i7-Shift-Entry* consumes 2,551 mJ, 3,075 mJ, and 6,029 mJ per frame, and *i7-Split-Index* consumes 1,956 mJ, 2,306 mJ, and 6,934 mJ per frame in 4:2:0, 4:2:2, and 4:4:4 modes. Since 4:4:4 frames contain twice the number of groups compared to 4:2:0 and 4:2:2 frames, 4:4:4 mode consumes ∼2× energy per frame compared to 4:2:0 and 4:2:2 modes. *Parallel-Index-8* is the most energy efficient, featuring 11 mJ, 12 mJ and 23 mJ in 4:2:0, 4:2:2, and 4:4:4 modes, which are **177**×, **193**×, and **261**× lower energy per frame compared to the most energy efficient i7 design.

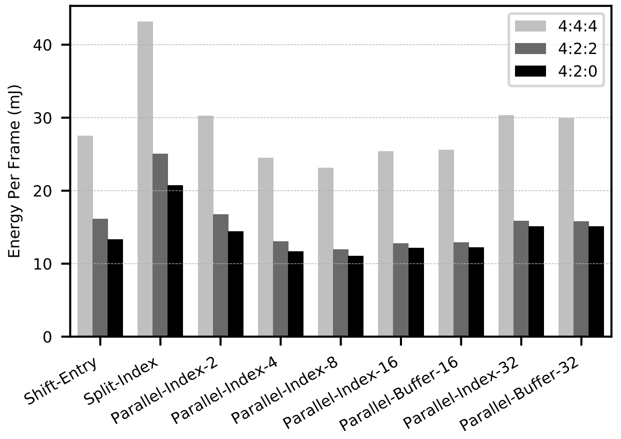

Fig. 11. Energy per 1080p frame of the 9 many-core designs. The i7 reference designs are not shown because they range from 1,956–6,934 mJ and are therefore far off scale.

V. Conclusion

We have presented two algorithms and two optimization methods for the DSC ICH module on many-core platforms. Nine designs were implemented and evaluated on a simple RISC-style processor array. Results show large increases in throughput and massive increases in energy efficiency compared to a 4-core Intel i7. These ICH designs can be integrated into complete DSC decoder designs on many-core platforms.

References

[1] Fortune, “Video conferencing market size, share & ind. analysis, by type, by appl., by enterp. size and reg. forecast, 2019-2026,” Dec 2019.

[2] VESA, “VESA Display Stream Compression (DSC) standard v1.2a,” Jan 2017. [Online]. Available: http://vesa.org

[3] F. G. Walls *et al.*, “VESA Display Stream Compression for television and cinema applications,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 4, pp. 460–470, Dec 2016.

[4] S. Wu and B. M. Baas, “A low-cost slice interleaving DSC decoder architecture for real-time 8K video decoding,” in *IEEE International Midwest Symposium on Circuits and Systems*, Aug. 2018.

[5] S. Wu *et al.*, “Display stream compression encoder architectures for real-time 4K and 8K video encoding,” in *IEEE Asilomar Conference on Signals, Sysems and Computers*, Oct. 2018.

[6] B. Bohnenstiehl *et al.*, “KiloCore: A 32 nm 1000-processor array,” in *IEEE HotChips Symposium on High-Performance Chips*, Aug. 2016.

[7] B. Bohnenstiehl, A. Stillmaker *et al.*, “Kilocore: A 32-nm 1000-processor computational array,” *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 52, no. 4, pp. 891–902, Apr. 2017.

[8] B. Bohnenstiehl *et al.*, “KiloCore: A fine-grained 1,000-processor array for task parallel applications,” *IEEE Micro*, pp. 63–69, Mar. 2017.

[9] M. Butler, “Bulldozer a new approach to multithreaded compute performance,” in *IEEE Hot Chips Symposium*, 2010, pp. 1–17.

[10] W. M. Holt, “Moore’s law: A path going forward,” in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, 2016, pp. 8–13.