Bulk JPEG Decoding on In-Memory Processors

Joel Nider
University Of British Columbia
joel@ece.ubc.ca

Jackson Dagger
University Of British Columbia
jdagger@student.ubc.ca

Niloo Gharavi
University Of British Columbia
ngharavi@ece.ubc.ca

Daniel Ng
University Of British Columbia
dannsy@student.ubc.ca

Alexandra (Sasha) Fedorova
University Of British Columbia
sasha@ece.ubc.ca

ABSTRACT

JPEG is a common encoding format for digital images. Applications that process large numbers of images can be accelerated by decoding multiple images concurrently. We examine the suitability of using a large array of in-memory processors (PIM) to obtain a high throughput of decoding. The main drawback of PIM processors is that they do not have the same architectural features that are commonly found on CPUs such as floating point, vector units and hardware-managed caches. Despite the lack of features, we demonstrate that it is feasible to build a JPEG decoder for PIM, and evaluate its quality and potential speedup. We show that the quality of decoded images is sufficient for real applications, and there is a significant potential for accelerating image decoding for those applications. We share our experiences in building such a decoder, and the challenges we faced while doing so.

CCS CONCEPTS

• **Computer systems organization** → Parallel architectures; • **Computing methodologies** → **Image compression**; *Parallel computing methodologies.*

KEYWORDS

JPEG, image processing, parallel processing, PIM, in-memory

ACM Reference Format:
Joel Nider, Jackson Dagger, Niloo Gharavi, Daniel Ng, and Alexandra (Sasha) Fedorova. 2022. Bulk JPEG Decoding on In-Memory Processors. In *The 15th ACM International Systems and Storage Conference (SYSTOR '22), June 13–15, 2022, Haifa, Israel.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3534056.3534946

1 INTRODUCTION

In 1992, the Joint Photographic Experts Group (JPEG) produced the ISO/IEC 10918-1 standard for encoding digital images [6]. Today, JPEG is used by web browsers, cell phones, image archives, digital cameras, producing several billion images every day [7]. Many of these images are uploaded to online services such as Flickr [3], SnapChat [15], Instagram [12], Facebook [11] and WhatsApp [13], producing huge online repositories of JPEGs. Processing these JPEG images is a common task in data centers for mining, tagging and machine learning applications. Therefore, acceleration of JPEG decoding is an important goal to save time and energy consumption in these applications.

Many techniques have been explored for accelerating JPEG decoding using various technologies including: vectorized instructions on a general-purpose CPU [10], parallel implementations for GPU [20], FPGA [9] and ASIC [16]. With the recent availability of DRAM enabled with in-memory processors from UPMEM [2], we evaluate the suitability of these new off-the-shelf, general-purpose, in-memory processors (PIM) for this task. For applications that process a lot of images, multiple images can be decoded concurrently to get a boost in throughput. PIM looks like a suitable solution, because JPEG decoding offers data parallelism and PIM is ideally suited for such workloads. The amount of computational resources and internal bandwidth of PIM-enabled memory scales with the amount of memory and hence with data size. Decoding using PIM processors can potentially reduce system cost because DPUs are cheaper than CPUs. In-memory processing architectures are designed to overcome the memory wall; to boost memory-bound workloads. In this case, we show that the massive parallelism of PIM is enough to boost a CPU-bound workload as well.

Processing in memory has been an active research subject for years, but few products have been built. UPMEM PIM

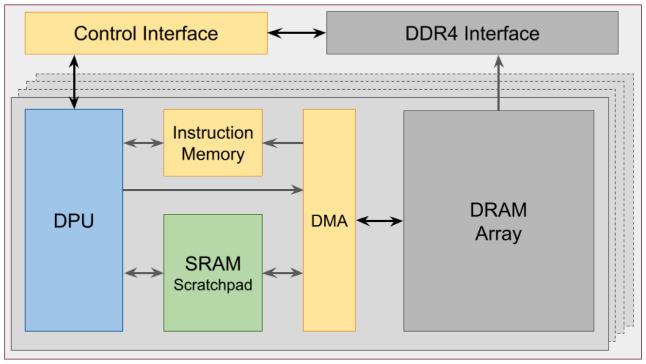

Figure 1: Architecture of in-memory processor from UPMEM

is one commercially available product, Samsung Function-in-Memory (FIM) is another. They took very different approaches: UPMEM integrated general-purpose processors into DRAM, Samsung added a domain-specific processor for AI to HBM (high bandwidth memory) with limited functionality. The jury is still out which approach will be more practical. A domain-specific solution might be more cost-effective, but the availability of a general-purpose processor allows us to explore what kind of an accelerator would be the right one. The results presented in this paper help us understand the properties of an "ideal" application that justifies building a dedicated PIM accelerator.

We evaluate the only commercially available, general-purpose PIM hardware for JPEG decompression. UPMEM has integrated general-purpose processors, called DRAM Processing Units (DPU), into DRAM DIMMs. DPUs are more akin to embedded microcontroller processors than fully-featured application processors. Due to the physical constraints of integrating DPUs into memory chips, UPMEM has made several design decisions that impact software programming. The lack of features makes programming a challenge and prevents existing applications from being used without modifications.

- DPUs have a small number of 32-bit registers and no hardware support for floating point
- DPUs only have direct access to 64KB of SRAM memory, that must be managed in software through explicit DMA operations on DRAM
- DPUs have a relatively slow clock rate (267 MHz) that does not compare to the GHz scale processors common today
- The threading model requires careful memory management when implementing an algorithm
- Most JPEG images do not contain the necessary information to be decoded with multiple threads

Despite these challenges, it is possible to write a high quality JPEG decoder on this platform.

2 BACKGROUND

2.1 Short overview of PIM architecture

A detailed description of the UPMEM architecture can be found in other publications [2, 5, 14], but the main points are summarized here. Figure 1 shows the high level architecture of an in-memory processor.

A DPU (DRAM processing unit) is an in-order, general-purpose processor that is embedded in the DRAM. There are 8 DPUs in one chip, each with a dedicated 64MB slice of DRAM. As this ratio is fixed, the number of DPUs scales with the memory capacity when more DIMMs are added to a system. This is good news for algorithms that have a lot of independent data that can be processed concurrently. Each DPU has 24 execution contexts (i.e. threads) called tasklets that operate in a round-robin fashion.

Each DPU has a 64KB private SRAM scratchpad buffer, called the *working memory*. The working memory is the only directly addressable memory for use by the DPU and is shared by all tasklets, including static and dynamic data structures as well as the runtime stacks. Any data coming from DRAM is copied with an explicit, blocking DMA instruction. The DMA engine can copy up to 2KB in a single operation and the copy time increases linearly with the size.

Every cycle, a DPU executes an instruction from one of its tasklets. Multiple tasklets are needed to hide DMA latency but only one tasklet can advance at each cycle. When a tasklet is blocked on a DMA operation, other tasklets can still make progress. The more often DMA operations block tasklets, the more tasklets are needed to keep the DPU pipeline filled.

Limited Memory. The most limited resource is the 64KB working RAM. That limits the number of blocks that can be decoded simultaneously. The lack of a communication channel between DPUs means the DRAM is in isolated segments. That puts a hard limit on the size of the image that can be processed by a DPU. The DRAM must hold both the input (encoded) image as well as the decoded output. For that reason, our implementation does not support decoded images that are larger than 40MB. That is still more than enough for a full HD picture (1920x1080) which needs about 6MB.

Floating point. There is no floating point hardware so all algorithms use only integer operations. Float point operations can be emulated in software but that is extremely slow. Libjpeg-turbo [10] (one of the official reference implementations) has forsaken floating point operations during decoding for performance reasons. Comments in their source

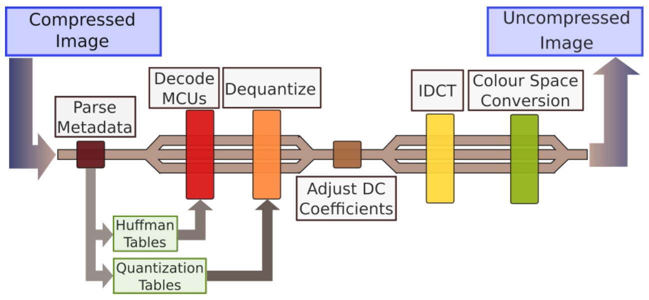

Figure 2: JPEG decoding process pipeline

code claim that the floating point operations do not significantly improve quality, but have a detrimental effect on performance. Even though we have no choice but to use integer operations, this suggests that it will not affect the quality of our implementation.

2.2 JPEG decoding overview

Decoded data is grouped into blocks of 8 x 8 pixels called Minimum Coded Units (MCUs). For example, a 1920 x 1080 grayscale image has 32400 MCUs, and a coloured image is 97200 MCUs (3× larger) since it has 3 colour channels. Figure 2 shows the decoding stages with serial steps on a single line and parallel steps on parallel lines. JPEG decoding begins with a serial step of reading the image metadata to discover global parameters such as the Huffman tables, and the quantization tables. The file contains multiple markers such as Start Of Image (indicates that this is a JPEG file), Start Of Frame (holds information about image width and height), and Start Of Scan (where the encoded image data starts). After extracting the information from the markers, the image data can be decoded. The Huffman codes are derived from the Huffman tables that are read from the markers. Using the Huffman codes, each MCU is decoded bit by bit.

After decoding, MCUs can be dequantized independently. During dequantization, each pixel in the MCU is multiplied by its corresponding value in the quantization table. The dequantized MCUs then undergo inverse discrete cosine transform (IDCT). This is described further in section 3.3. Adjusting the DC coefficients of all MCUs must be performed serially, since the coefficient of block N is based on the adjusted value of block N-1 (i.e. differential encoding). Lastly, the 3 colour channels must be converted from the YCbCr colour space to RGB values before they can be used by the application, or saved to a file.

2.3 Parallelization

The JPEG standard specifies an optional feature called a 'restart marker' to parallelize decoding. When included, the restart marker appears periodically throughout the file, to mark the beginning of a new scan line. The markers make it easy to know the index of the MCU being decoded, so every scan line can be decoded by a different thread without major effort of coalescing the results at the end. Unfortunately, most JPEG images in the wild do not contain the restart marker, which complicates parallelization efforts. At the same time, most JPEG images in the wild are encoded using Huffman trees. Klein and Wiseman [8] explain how most Huffman codes are self-synchronizing and show how this property can be exploited for parallel JPEG decoding. Their theorem states that Huffman codes can be decoded starting from a random location (not necessarily at a code boundary) and will eventually read valid codes, given the block of data is sufficiently large, and certain (rare) properties of the Huffman codes do not hold. While this behaviour is not guaranteed, we find (as they did) that it is quite reliable for JPEG images encoded in the standard way.

3 DESIGN

The parallel portions of the decoding process are divided among multiple tasklets, where each tasklet handles the dequantization, inverse DCT, and YCbCr to RGB conversion steps. Dequantization and conversion of YCbCr to RGB are simple steps, working on each pixel individually with simple operations. The inverse DCT is the most computationally intensive step, detailed in section 3.3. After colour conversion, raw data can be written to a file for validation of whether decoding was performed correctly, and further preprocessing such as scaling and cropping can be carried out.

3.1 Working memory

The 64KB of working memory is the most constrained resource but sufficient for decoding a JPEG. It is used for the runtime stack of each tasklet (1KB) as well as program data. During decoding, some global data structures must remain in memory. The four quantization tables are a fixed size of 256 bytes each (64 32-bit integers) for a total of 1KB. The four Huffman tables (2 for DC components and 2 for AC components) are 1298 bytes each, for a total of 5192 bytes. If each of the 24 tasklets decodes a single MCU at a time (64 bytes of decoded data), we need about 32KB in total.

3.2 Tasklet synchronization

To improve decoding time, the image is processed by multiple tasklets. Each tasklet is assigned an equal length range of image data to decode. Since the encoded MCUs are variable length, there is no simple way to determine where one MCU ends and another begins. Therefore, it is very likely that the tasklets will begin decoding in the middle of an MCU, and thus produce MCUs which are incorrect. This means that each tasklet will decode a small number of MCUs incorrectly and all subsequent MCUs will be decoded correctly after the Huffman codes synchronize. The key is to identify when the tasklets start decoding MCUs correctly. We detect an incorrectly decoded MCU by looking at the range of known values from Huffman codes, such as the field length and values of the AC and DC coefficients. Any out-of-range values causes decoding to abort and restart as the next MCU. Each tasklet records the byte index of the first 128 MCUs that are decoded successfully. Rather than having a tasklet rewind the stream to an earlier point to retry failed decoding, we depend on the previous tasklet to compensate by continuing to decode beyond its assigned range. To illustrate the point, assume that tasklet *N* has completed decoding its assigned range and the first few MCUs are decoded incorrectly. We rely on tasklet *N-1* to continue decoding beyond its range into tasklet *N*'s assigned range. As tasklet *N-1* decodes MCUs, it compares the index of the first compressed byte of the MCU with the index recorded by tasklet *N*. When three consecutive indices match, the decoded data has been synchronized. After all tasklets have finished decoding, all the correctly decoded MCUs are concatenated into a contiguous memory block.

From previous work, we know the optimal number of tasklets depends on the ratio of data movement time to processing time. Gomez-Luna et al. [5] explain how the DPU pipeline means instructions in the same thread should be dispatched at least 11 cycles apart to fully utilize the DPUs. Gomez-Luna et al. further experimentally confirmed that at least 11 tasklets were needed to fully saturate the DPU pipeline when measuring arithmetic throughput or MRAM throughput. Furthermore, they recommend using more than 11 tasklets in real-world workloads to ensure the pipeline remains fully saturated. Experimentally, we determine the optimum number of tasklets for JPEG decoding to be anywhere between 11 and 15, the maximum number of decoding tasklets that can be supported by the DPUs' WRAM. We speculate that using more than 11 tasklets means that the pipeline remains fully saturated when a few tasks are waiting on transfers between MRAM and WRAM which may take up to 1100 cycles in our case.

3.3 Inverse DCT with integers

The Discrete Cosine Transform (DCT) maps a finite sequence of data points from a spatial representation into the frequency domain. Basically, each value in an MCU represents the sum of cosine functions oscillating at different frequencies. This sum has to be transformed back to a finite sequence of data points for each MCU to obtain the original values of the 3 color channels for each pixel. The IDCT function performs this transform.

A multi-dimensional DCT can be reduced to 1D transformation along all dimensions. In JPEG decoding, the rows will be transformed and then the DCT is run on the columns. The resulting matrix is the two dimensional DCT. The IDCT step of decoding depends on the cosine function. Since JPEG encoding uses 8x8 blocks, there are only 64 cosine values that are needed and they are constant across all MCU blocks. The AAN algorithm (Arai, Agui, and Nakajima [1]) is the fastest known algorithm to transform a 8x8 block. AAN simplifies 8-point one-dimensional DCT to 13 multiplications and 16 additions/subtractions so that all cosine values can be stored in 13 constants. Expensive floating point multiplications, that are not supported by the DPUs, are approximated by integer addition and shift operations [22].

4 EVALUATION

We evaluate using the tiny-imagenet-200 dataset [21], which is widely used for machine learning tasks. In addition, we have tested many random JPEGs that we find on the Internet to explore the limits of the decoder. We are not aiming for a production-ready implementation and have omitted features that are not necessary for decoding the majority of JPEG images that we tested. Since our implementation does not closely follow the reference implementation (libjpeg-turbo), it is important to evaluate the quality of the decoded image compared to the reference implementation by using two accepted indicators: SSIM (structural similarity index) [19] and PSNR (peak signal-to-noise ratio). While this will give us concrete numbers, we also want to know qualitatively if the decoded images are still good enough for real applications. We do so by comparing the results of image classification using ResNet18 as detailed below. Since our current implementation has not yet been optimized for speed we cannot fully evaluate the performance, so we address this by looking at some extrapolations from our measurements.

4.1 Information Loss

SSIM. Structural similarity index is a full-reference image metric which means it compares a reference image with a distorted image. While the uncompressed original image is generally used as the reference, we do not have access to the original images. Instead, we use the same compressed image decoded by the reference implementation as the reference image to compare to the image as decoded by our implementation. Unlike Mean-Squared Error or PSNR, which are also full-reference metrics, SSIM attempts to take into account the differences in the structure of each image. SSIM factors in the relation between nearby pixels to estimate perceptual error. Figure 3 shows the SSIM, with a median of 0.996. For comparison, a mean SSIM of 0.95 was found to be the approximate threshold for which a distortion is imperceptible to non-expert human observers [4].

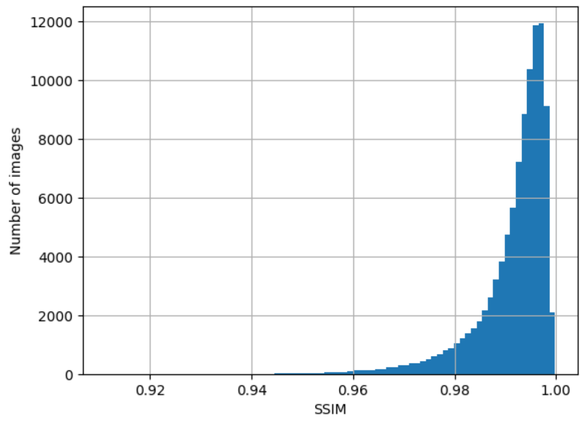

Figure 3: SSIM of tiny-imagenet images decoded on DPU vs CPU

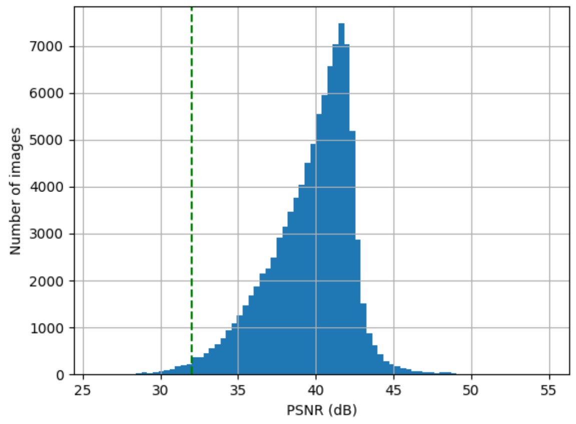

Figure 4: PSNR of tiny-imagenet images decoded on DPU vs CPU with line marking 32 dB quality threshold

PSNR. *Peak signal-to-noise ratio* quantifies the noise in an image by finding the difference between an original image and a processed (i.e. compressed and then decompressed) image. Since we do not have access to the original uncompressed images, we use PSNR to measure the difference introduced by decompression on PIM as compared to the same image decompressed on a CPU. Therefore, the metric does not indicate the quality of PIM decompression per-se, but rather how different the decompression is from that of a CPU implementation. Figure 3 shows the distribution of noise measurements. PSNR is measured in dB (decibels) on a logarithmic scale. Perfectly matching images would have infinitely large ratios, so larger numbers indicate higher similarity. Figure 4 shows a median of approximately 41 dB. This indicates a strong similarity between the outputs of the two decoders. For comparison, 32 dB is enough to be considered good for JPEG transmission across a wireless channel [18].

	Inference	
	CPU	DPU
CPU Model	31.63% (0.06%)	31.88% (0.18%)
DPU Model	31.70% (0.10%)	32.21% (0.11%)

Table 1: Accuracy of image classification using ResNet18. The model was trained on images decoded by libjpeg-turbo (CPU) and our implementation (DPU), and cross-validated by inference on images from both decoders.

4.2 Real Applications

SSIM and PSNR are good mathematical methods for summing up the differences between two images, but it does not tell us much about utility of the image itself. We are most interested to know if the results are useful for image processing applications. We use ResNet18, a popular neural network to experiment with image classification. It operates in two phases: teaching the network to recognize objects (training), and then classifying objects in images that the network has not yet seen (inference). Both phases rely on the quality of the images for good results. We train ResNet18 on tiny-imagenet-200 decompressed with both implementations. Table 1 shows the inference accuracy is not significantly different between the two tested decoders. The accuracy we see with a stock (untuned) network is approximately 10% lower than the best (41%) reported by students at Standford, who used a highly tuned network [21].

4.3 Performance

Our implementation is a proof-of-concept to explore the feasibility of implementing a complex piece of code such as a JPEG decoder in PIM. We did not invest the effort required for an optimized implementation and so we cannot report the performance in as much detail as we would like. However, we did some experiments to show system characteristics such as scalability to get some intuition regarding the performance.

Figure 5 shows the throughput (in files per second), as the number of images is increased exponentially. Each DPU processes a single image, so the number of DPUs increases along with the number of images, up to the maximum (576 DPUs across 9 ranks). The first segment ramps up quickly, up to 8 ranks (512 DPUs) as the incoming images fill the

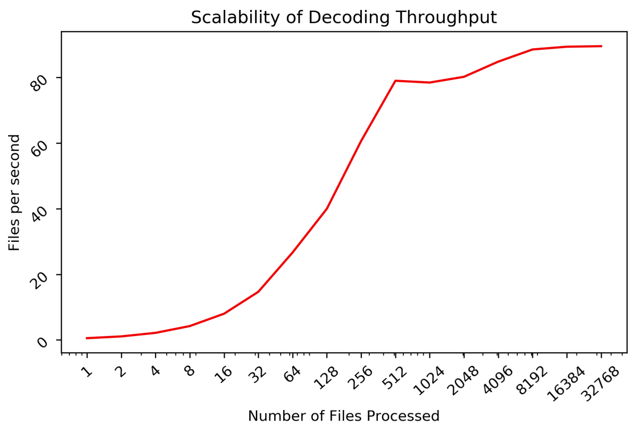

Figure 5: Processing scalability as the number of DPUs increases with the number of files (log scale)

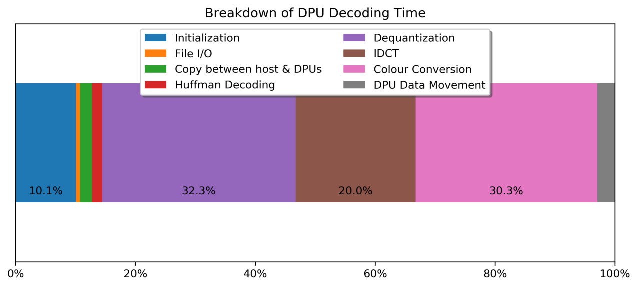

Figure 6: Timing details of JPEG decoding on a DPU. Over 85% of the time is spent on the DPUs. Dequantization (30.3%), inverse DCT (20.0%) and YCrRb-to-RGB colour conversion (30.3%) take the majority of the time.

idle hardware. At 9 ranks, the performance levels off as new work entering must wait until previous batches complete, and are written to disk. After that point, there is another smaller performance increase as all of the DPUs are used more than once, and concurrent operation of the ranks starts to compensate for stragglers. The performance then reaches a steady state of 90 files per second at which point the system is compute-bound (all of the DPUs are operating at maximum capacity). This highlights the advantage of having no communication or sharing between the DPUs; they operate nearly completely independently. Adding more DPUs to the system would continue to increase throughput until the I/O device or memory bus becomes saturated.

Our DPU implementation with a single tasklet decodes a JPEG approximately 200× slower than ImageMagick [17] that uses libjpeg-turbo, [10] (using SIMD acceleration) on an Intel Xeon 4110 CPU core. Figure 6 shows a detailed breakdown of the time spent. The initialization of the SDK has a fixed overhead that is significant when decoding a single image, in this particular case, 10% of the time. The majority of the time is spent on dequantization, inverse DCT and colour conversion operations. These rely heavily on multiplication which is quite costly in the DPU. Since many of the multiplications are by constants (such as the size of an MCU), it is likely that we can make significant performance improvements by replacing them with adds and shifts.

Additional tasklets has the most significant performance improvements for larger images. The additional effort of decoding MCUs that were decoded incorrectly for small images (64x64) outweighs the benefit. Larger images (500x375 or 640x480) can benefit from multiple tasklets.

With a large number (10000) of small images (64x64 pixels), our implementation on 200 DPUs (2.5 ranks) with a single tasklet has approximately the same throughput as a single Xeon 4110 CPU core. In this case, the SDK initialization time is less significant, but our rudimentary implementation of disk I/O becomes a factor.

4.4 Floating Point

As we have shown, the lack of floating point hardware in the DPU does not hinder JPEG decoding but there are other cases to consider. Once images are decoded, image processing libraries perform a range of operations including arbitrary rotations and arbitrary scaling that do depend on floating point. These operations require high precision interpolation of data, and multiplication with fractional values that can be best performed in floating point. These can be accepted as future challenges for others to face.

5 CONCLUSION

We show that it is feasible to build a JPEG decoder on PIM hardware that produces high quality output. The hardware architecture matches well with the requirements of the algorithm. The major limitations are the size of the decoded file and lack of features in the encoded files to support parallel decompression.

As in many systems, this is a case of gaining throughput at the cost of higher latency, when compared with JPEG decoding on a CPU. PIM is suitable for use as a JPEG decoder, in cases where the datasets are large enough to overcome the high latency with concurrent processing of many files. It is noted however, that specialized hardware that could perform the computationally intensive portions of the algorithm (inverse DCT) more efficiently would likely have a huge impact. While there is much to be said about the flexibility of a general purpose processor in memory, some common computational tasks readily expose its limits.

REFERENCES

[1] Yukihiro Arai, Takeshi Agui, and Masayuki Nakajima. 1988. A Fast DCT-SQ Scheme for Images. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 71 (1988), 1095–1097.

[2] F. Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. HOTCHIPS, 1–24. https://doi.org/10.1109/HOTCHIPS.2019.8875680

[3] Flickr. 2022. photos on Flickr. https://www.flickr.com/photos/tags/photos/

[4] Jeremy R. Flynn, Steve Ward, Julian Abich, and David Poole. 2013. Image Quality Assessment Using the SSIM and the Just Noticeable Difference Paradigm. In *Engineering Psychology and Cognitive Ergonomics. Understanding Human Cognition*, Don Harris (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–30.

[5] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. arXiv:2105.03814 [cs.AR]

[6] Joint Picture Experts Group. [n.d.]. Overview of JPEG 1. https://jpeg.org/jpeg/index.html

[7] Joint Picture Experts Group. 2015. JPEG Privacy & Security Abstract and Executive Summary. https://jpeg.org/items/20150910_privacy_security_summary.html

[8] S. T. Klein and Y. Wiseman. 2003. Parallel Huffman Decoding with Applications to JPEG Files. *Comput. J.* 46, 5 (2003), 487–497. https://doi.org/10.1093/comjnl/46.5.487

[9] George Gabriel Kyrtsakas. 2017. "An FPGA Implementation of a Custom JPEG Image Decoder SoC Module". *Electronic Theses and Dissertations* 5945 (2017).

[10] Miyasaka Masaru. [n.d.]. libjpeg-turbo. https://libjpeg-turbo.org/Main/HomePage

[11] Meta Inc. 2022. Connect with friends and the world around you on Facebook. https://www.facebook.com/

[12] Meta Inc. 2022. Instagram. https://www.instagram.com/

[13] Meta Inc. 2022. Simple. Secure. Reliable messaging. https://www.whatsapp.com/

[14] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, and Alexandra Fedorova. 2021. A Case Study of Processing-in-Memory in off-the-Shelf Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 117–130. https://www.usenix.org/conference/atc21/presentation/nider

[15] Snap Inc. 2022. The fastest way to share a moment. https://www.snapchat.com/

[16] Sung-Hsien Sun and Shie-Jue Lee. 2003. A JPEG Chip for Image Compression and Decompression. *VLSI Signal Processing* 35 (08 2003), 43–60. https://doi.org/10.1023/A:1023383820503

[17] The ImageMagick Development Team. [n.d.]. *ImageMagick*. https://imagemagick.org

[18] N. Thomos, N.V. Boulgouris, and M.G. Strintzis. 2006. Optimized transmission of JPEG2000 streams over wireless channels. *IEEE Transactions on Image Processing* 15, 1 (2006), 54–67. https://doi.org/10.1109/TIP.2005.860338

[19] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

[20] André Weißenberger and Bertil Schmidt. 2021. Accelerating JPEG Decompression on GPUs. arXiv:2111.09219 [cs.DC]

[21] Jiayu Wu, Qixiang Zhang, and Guoxi Xu. [n.d.]. Tiny ImageNet Challenge. ([n. d.]). http://cs231n.stanford.edu/reports/2017/pdfs/930.pdf

[22] Pingping Zhu, Jianguo Liu, Shengkui K. Dai, and Guoyou Wang. 2009. Scaled AAN for Fixed-Point Multiplier-Free IDCT. *EURASIP Journal on Advances in Signal Processing* 2009 (2009), 1–9.