Distilling the Essence of Raw Video to Reduce Memory Usage and Energy at Edge Devices

Haibo Zhang, Shulin Zhao, Ashutosh Pattnaik, Mahmut T. Kandemir, Anand Sivasubramaniam,
Chita R. Das
The Pennsylvania State University
{huz123,suz53,aup234,mtk2,axs53,cxd12}@psu.edu

ABSTRACT

Video broadcast and streaming are among the most widely used applications for edge devices. Roughly 82% of the mobile internet traffic is made up of video data. This is likely to worsen with the advent of 5G that will open up new opportunities for high resolution videos, virtual and augmented reality-based applications. The raw video data produced and consumed by edge devices is considerably higher than what is transmitted out of them. This leads to huge memory bandwidth and energy requirements from such edge devices. Therefore, optimizing the memory bandwidth and energy consumption needs is imperative for further improvements in energy efficiency of such edge devices. In this paper, we propose two mechanisms for on-the-fly compression and approximation of raw video data that is generated by the image sensors. The first mechanism, *MidVB*, performs lossless compression of the video frames coming out of the sensors and stores the compressed format into the memory. The second mechanism, *Distill*, builds on top of MidVB and further reduces memory consumption by approximating the video frame data. On an average, across 20 raw videos, MidVB and Distill are able to reduce the memory bandwidth by 43% and 72%, respectively, over the raw representation. They outperform a well known memory saving mechanism by 7% and 36%, respectively. Furthermore, MidVB and Distill reduce the energy consumption by 40% and 67%, respectively, over the baseline.

CCS CONCEPTS

• **Human-centered computing** → **Mobile devices**; • **Computer systems organization** → **Architectures**; **Embedded and cyber-physical systems**.

KEYWORDS

SoC, Memory, Video Streaming, Edge Computing

ACM Reference Format:
Haibo Zhang, Shulin Zhao, Ashutosh Pattnaik, Mahmut T. Kandemir, Anand Sivasubramaniam, Chita R. Das. 2019. Distilling the Essence of Raw Video to Reduce Memory Usage and Energy at Edge Devices. In *MICRO '52: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture, October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358298

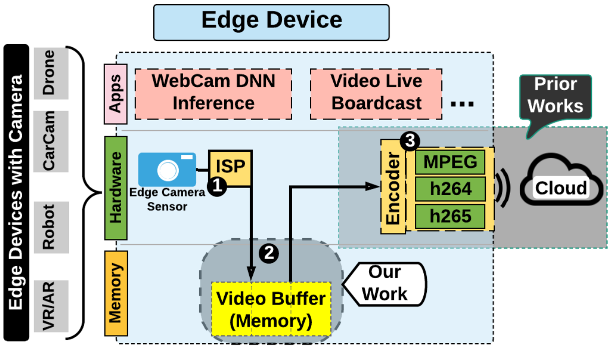


Figure 1: Overview of a video processing application at edge devices.

1 INTRODUCTION

Video has rapidly emerged as one of the dominant applications and use-case drivers in the mobile and Internet-of-Things (IoT) era. On the mobile end, video traffic contributes to nearly 82% of internet traffic [8] with users spending billions of hours streaming up and/or down video, either offline or in real-time. Communication and smaller form factors have led to the proliferation of IoT devices in numerous physical environments. Again video plays an important role in these applications - smart cameras for surveillance, remote imaging on drones/robots, autonomous vehicles, etc. Even though optimizing for video delivery has been a topic under extensive study for the last 3-4 decades, the explosion of video in the context of mobile and IoT edge devices, warrants a revisit. This is specially needed to address the limited resources and energy constraints of these edge devices. Towards this goal, this paper examines the internals of how video is produced and consumed on these edge devices, points out the dominance of the memory system for these purposes, and proposes novel compression and approximation techniques to distill out the bits of non-significance towards significantly boosting memory bandwidth and energy efficiency.

Consider a video processing application, as shown in Fig. 1, where a camera generates raw pixels from its sensors. Based on the resolution and capabilities of these sensors, this data rate could be as high as 1.4GB/s for a 60 FPS at 4K resolution. Lots of prior works over the decades have noted these high data requirements, either for storage or network bandwidth, and have proposed mechanisms to compress such data. Further, standards such as MPEG4 [37], H.264 [74], H.265 [65] and VP9 [19], have evolved over these decades to lower the imposed bandwidth on the transmission network. The video application can be viewed as a pipelined execution,

MICRO '52, October 12–16, 2019, Columbus, OH, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6938-1/19/10...$15.00
https://doi.org/10.1145/3352460.3358298

which first goes through an encoder to compress the data into one of these data formats (e.g., MPEG4), which is then transmitted on a network from the resource-constrained edge device to a back-end server/cloud. The reverse - taking the encoded data and decoding it back into raw data (though this data may have suffered losses due to this process) - takes place at the back-end. Note that, in this paper, we are *not attempting to address any specific inefficiencies in the transmission between the encoding and decoding stages, i.e. the codecs*. Instead, we focus on the edge device itself where the raw pixels from the camera need to be encoded to one of these formats (e.g., MPEG4, H.264) before being sent out on the network.

It is important to understand what happens on the edge device for such encoding operations. The raw pixels are not directly pipelined to the encoder (which is usually an ASIC). There have been prior proposals such as [38], where virtualized channels to flow the data directly have been proposed, but those may not (i) reduce the number of memory accesses, and (ii) may not universally work in all edge-device scenarios (especially very limited resource constrained ones). Instead, memory is used to buffer these raw pixels, which are then read by the encoder, frame after frame, for its computation. We note that to function as a staging area, memory consumes the same amount of energy as the encoder computation itself in the ASIC. This is the target of optimization in this paper - how do we reduce memory usage/bandwidth so as to significantly reduce its energy consumption when serving as a staging area between the camera and the encoder? In the process, we do not want to impact the quality and/or the speed of the video.

One of the most exploited characteristics when encoding video is the repetition of data values which can be compressed or even approximated (e.g. in MPEG4, H.264 and VP9). We use the same rationale for our intention to reduce the bandwidth into and out of the memory before the encoder. At the same time, the technique for leveraging value commonality/locality cannot be as extensive as H.264 (i.e., we cannot make this a recursive problem), since one does not have the luxury of looking at the entire frame's raw data in order to do the compression. For instance, a scheme such as H.264 needs to examine up to four reference frames which can take around 100 MB, and perform quantization/Discrete-Cosine-Transform (DCT)/Motion-Estimation, etc., to come up with the appropriate compression. In fact, it may need several frames' data to perform this effectively. Optimizing the memory buffer using such encoding techniques would thus require us to recursively maintain buffers of several MBs as well (in addition to the extra latency/energy for computations), defeating the entire purpose. Instead, we want a very light-weight compression mechanism that operates with only a few bytes of raw data in real-time and still provide an aggressive compression ratio.

In this work, we focus on exploiting *value similarity* in the frame data to save memory bandwidth, and thus, reduce memory energy. Specifically, within a small spatial region within a frame (denoted as *tile*), the pixel values are highly similar (shown in Fig. 2). Previous works [84, 85] have studied the *value equality* in the decoding buffer, by utilizing a "content cache" to capture the value equality between tiles. As we will show later in this paper, while these techniques are well suited to reduce memory accesses in decoding videos, they do not perform well in the encoding buffer, due to minimal *value equality* at a tile level. Instead we propose a two-step solution to take advantage of the *value similarity* between pixel values inside a tile. We first exploit the compressible value similarity by optimizing the number of bits to represent the tile. Previous work on compression scheme have used the first pixel value as the "base" [53] and have reduced the dynamic value range of the deltas of other pixels to save memory usage. However, as we will discuss later in this paper, selecting the first pixel as the base may not be optimal. We systematically discuss how "base selection" as well as corresponding "value representations" affect bit efficiency. We formulate the base selection problem, and propose to use the middle point (middle point or mid is not the same as median, denoted as **MidVB**) of the tile values, which we prove to have the best bit efficiency than any other base for computing the deltas.

Even if, on the edge device, we are to provide lossless compression between the camera/ISP to the video encoder through the memory, it still goes through a lossy encoding mechanism (such as H.264 [74]) before being sent out over the network to the back-end server from the edge device. So the natural next question we explore is - why not allow lossy compression between the camera/ISP and the encoder as well? It is possible that despite such compression quantization within the encoder may not notice any (or perceptible) difference in the data. With this rationale, we next propose approximating the original frame data before it goes into the memory that the encoder will access. The second proposal, referred to as **Distill**, leverages this feature to further reduce memory bandwidth, through disciplined frame data approximation at minimal computation and energy costs. We find that using the middle value of a tile of value is a good "estimation" of the contents of a given tile, and the deltas (from the original pixel values to middle values) have minimal mean square error from the original tile value. Instead of using a single base (middle point) and its delta to accurately represent the tile, we repeat this process *iteratively* to find "a series of bases", where each iteration reduces the delta range exponentially, to *approximate* the contents of a tile.

To the best of our knowledge, this is the first work that compresses and approximates raw video frame data before being fed to an encoder for reducing the memory bandwidth and energy on the edge device. With memory contributing roughly the same percentage as the encoding computation itself (on the ASIC), this is an important target for optimization. More details about Distill can be found at: `https://huz123.github.io/distill.html`. Specifically, we make the following **major contributions**:

- We identify different types of value similarity in raw and encoded video buffers. We observe that raw video buffers have 99% unique tiles, while encoded videos have only 35% unique tiles.
- We exploit pixel value similarity within a tile. We formally analyze the trade-offs of bit efficiency between the base selection and the representation of deltas. We prove that selecting middle value (or minimal value) of the tile as the base provides better bit efficiency than other values. Using the middle value as its base, **MidVB**, without losing any data, provides around 43% memory bandwidth savings with respect to the raw representation.
- We propose an approximation scheme (**Distill**) which iteratively invokes our MidVB to find a series of bases to approximately represent the original tile contents. We theoretically prove that the proposed scheme converges with a small number of iterations

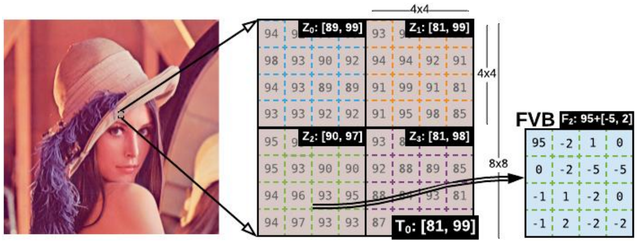


Figure 2: Spatial value similarity in pixels (Image from [73]).

with minimal loss in video quality. Compared to MidVB, Distill further reduces the original memory bandwidth demand, with disciplined quality control. More specifically, it reduces around 72% of memory bandwidth, and consequently saves 67% of the total memory energy with respect to the raw representation.

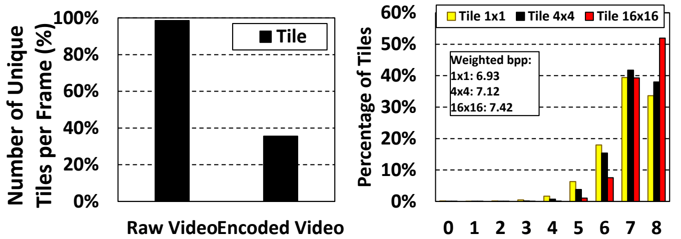
2 BACKGROUND AND MOTIVATION

In this section, we first describe the basics of today's video applications running on mobile/edge devices and the value locality in raw videos. Then, we explain the "redundancy" in the video buffer design, and available "value similarity" in pixels for exploiting the opportunity to reduce the bandwidth demand of the video buffer.

2.1 Value Locality in Raw Videos

A video recording task (such as Snapchat [63], Tiktok [68]) is organized into three separate stages (camera, image signal processor (ISP), and video encoding), as shown in Fig. 1. A video buffer (encoding buffer) (depicted as ❷) is used in the camera pipeline to buffer *raw frames* from ISP (depicted as ❶) to the video encoder (depicted as ❸). The encoder utilizes these buffers to read the raw frames, while the ISP generates new frames.

Although video applications demand high memory bandwidth, they usually exhibit a high value locality across the frames as well as within a frame. To understand this value locality, we consider the Lenna image [73] as an example in Fig. 2, and magnify one small region to view the pixel values of four 4x4 tiles (R channel), namely Z_0, Z_1, Z_2 and Z_3. Each tile shows a very small dynamic range of pixel values, e.g., `[89, 99]` for the first tile, and `[90, 97]` for the third tile. The pixel values have larger variations `[81,99]` for a larger tile (8x8, depicted as T_0). From the control flow perspective, the encoding pipeline is just the reverse of the decoding pipeline. However, the encoding and decoding pipelines exhibit different value localities; specifically, the encoding pipeline has much less value equality compared to the decoding pipeline. In a decoding pipeline, the frame data have already been encoded by an encoder (H.264 [74], H.265 [65], VP9 [19], etc.) and are lossy. They are lossy because the encoder quantizes the original frame data at a macro-block-level (4x4, 8x8, or 16x16). Therefore, the encoded frame has more value equality at a tile-level after the quantization. Compared to the decoding pipeline, the encoding pipeline handles *raw* frame data, which are directly generated by a camera front end (including ISP), and thus have very limited value equality. Fig 3a shows the number of unique tiles per frame in a raw video and in an encoded video, both using a 4K resolution. The number of unique tiles in the raw videos are about 99%, meaning that only 1% of the tiles are exactly same to each other; instead, the number of unique tiles in the encoded video is much less, suggesting that around 65% of the tiles are identical to other tiles. This motivates us to look at the value similarity in the frame data, especially in the encoding pipeline, rather than just exploiting value equality.



(a) Number of unique tiles. (encoding vs decoding)

(b) Bit-width of tiles (1x1, 4x4, 16x16).

Figure 3: (a) Number of unique tiles of raw video and encoded video inputs. (b) Pixel intensity (tile 1x1) of a sequence of tiles (in total 1.8 billion tiles of 1199 frames). 34% of the pixels require 8 bits to fully represent their values. Partitioning frames into tiles: each tile consists of 4x4, or 16x16 pixels. A tile's bit-width is determined by the largest bit-width of the pixels in the tile.

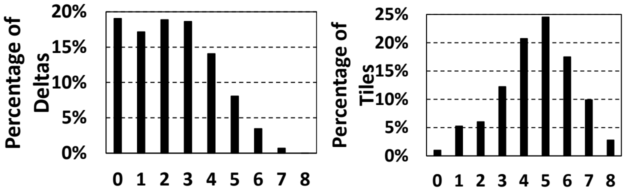
2.2 Reducing Video Buffer Bandwidth

To understand how to reduce the bandwidth demand of the video buffer, we first investigate the characteristics of the pixel values stored in a video buffer, especially in the encoding buffer. In this work, we assume that the pixels are in the RGB color space[1]. Each color channel spans from 0 to 255, and requires 8 bits to fully represent a pixel. However, some pixels do not really need 8 bit-width to represent their values. We collect a set of frames (1199 frames) from a 4K resolution raw video (R3 in Table 3), and plot their pixel value distributions (pixel intensity) in Fig. 3b. We observe that, only 34% of pixels are in the range of `[128, 255]`, which require 8 bits to represent their values. In comparison, around 65% of pixels need only 5 - 7 bits to fully represent their values. Consequently, instead of using 8 bits for each and every pixel, we can potentially use fewer bits (reduced bit-width) to represent some pixels' values, so that the overall amount of data needed for representing a pixel, *bits per pixel (bpp)*, can be reduced.

Leveraging the pixel intensity. One straightforward approach that utilizes the pixel intensity is to add a tag for each pixel, indicating the bit-width required for that pixel. The tag, however, needs 4 bits to indicate the bit-width (`[0,8]`). In the case of Fig. 3b, the average number of bits per pixel is 6.93 for a single pixel (i.e., 1x1 tile). If we tag every pixel with this 4 bit indicator, the total bpp for all frames would require 10.93 bits, worse than the original representation of 8 bits per pixel.

While tagging each individual pixel is quite costly, it could be profitable to use the bit-width tag for a small "pixel region", so that the tag cost can be amortized over the number of pixels in the region. To do so, we partition a frame into a set of "tiles", where each tile contains a square region of pixels (e.g., 4x4 or 16x16). Fig. 3b shows the bit-width distribution of the tiles (4x4, 16x16) . Here, the bit-width of the tile is determined by the *largest* pixel value in a tile. From Fig. 3b, we make the following two *key* observations:

[1] Note that the other color space such as YCrCb [76], HSV [75] are linear transformations of the RGB color space, and as a result, they hold similar properties.



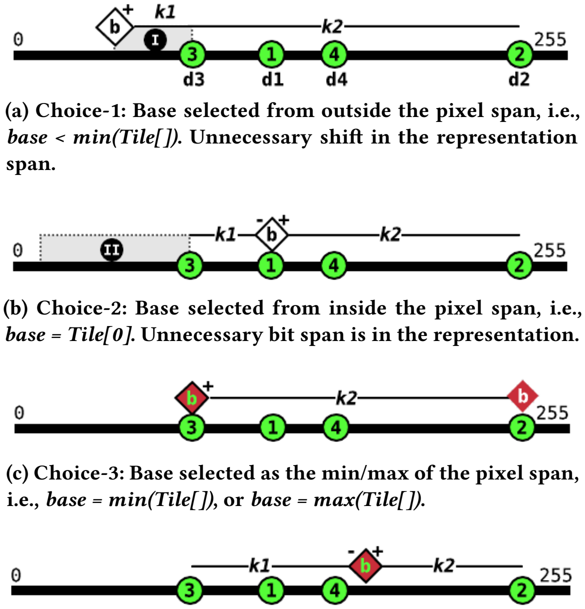
(a) Deltas of a tile. (b) Bit-width of delta in a tile (4x4).

Figure 4: (a) Distribution of the pixel deltas in a tile. Pixel deltas are the differences from the *first pixel* value in the tile. Around 20% of the pixel deltas are 0, indicating that they are the same as the first pixel value. Only 0.01% of the pixels still require 8 bit-width. (b) The bit-width needed for a tile is now determined by the number of bits needed to represent the maximal (minimal) of deltas of the tile.

- **Bpp varies across different tile sizes:** A small tile (4x4) gives a bpp of 7.12 and the bit-widths of small tiles are distributed similarly compared to the original pixel intensity (considered as tile of 1x1). In larger tiles (16x16) however, more tiles require larger bit-widths, increasing bpp to 7.42. We observe that, pixel values have little variations across small-sized tiles, implying rich similarity. We will study the value variation below in detail.
- **Minimizing tag cost saves memory bandwidth:** Each tile is associated with a 4-bit index tag. The additional tag cost amortized by pixels decreases as the tile size increases, e.g., 4/16 = 0.25 bpp for a 4x4 tile and 4/256 = 0.01 bpp for a 16x16 tile. Including the tag cost, the overall cost of transmitting the tiles are 7.37 and 7.43 bits, for the 4x4 and 16x16 tiles, respectively. Compared to the original 8 bpp pixel representation, this tagging scheme effectively saves 8% memory bandwidth for this set of videos.

Exploiting value similarity in tiles. The above observation indicates that there exists value similarity in a tile. To study this value similarity, we set the tile size to 4x4, use the first pixel (left-top) value in the tile as the "base", and represent the difference of the other pixels from this base in this tile as *absolute "deltas"* (distances without signs). Fig. 4a plots the distribution of these deltas, using the same set of frames. The intensity of the deltas concentrates more towards 0, compared to the original pixel intensity, thus exhibiting a very high similarity to the first pixel value (from other pixels). In fact, around 20% of the pixels in a tile are the *same* as the *first pixel value*, and only very few pixels are 8 bits `[128 - 255]` away from the first pixel.

The smaller range of the pixel deltas motivates us to represent a tile using these delta values. More specifically, a tile can now be represented as a "base" (the first pixel value in the tile), a set of "delta" values, and a 4-bit tag. In the remainder of this paper, we refer to this representation as **FVB** (First-pixel-Value-Based). The tag is used to indicate the maximum number of bits needed to represent the deltas of the tile, so that these deltas can use this bit-width to represent a tile *without losing any precision.* The previous tagging scheme shown in Fig. 3b can be considered as using *zero* as the base, and is named as **ZVB** (Zero-Value-Based). Fig. 4b shows the *distribution* of the maximum number of bits needed for a tile. FVB can achieve a bpp value of 4.56. The additional data required for the tile are the 8-bit first pixel value, and the 4-bit tag, leading to a bpp value of (8+4)/16 = 0.75. The resulting overall bpp is 5.31, which provides more than 33.7% of memory bandwidth savings.



(a) Choice-1: Base selected from outside the pixel span, i.e., *base < min(Tile[ ])*. Unnecessary shift in the representation span.

(b) Choice-2: Base selected from inside the pixel span, i.e., *base = Tile[0]*. Unnecessary bit span is in the representation.

(c) Choice-3: Base selected as the min/max of the pixel span, i.e., *base = min(Tile[ ])*, or *base = max(Tile[ ])*.

(d) Choice-4: Base selected as the mid of the pixel span, i.e., *base = floor(max+min+1)/2*.

Figure 5: Choices of selecting a base for a given tile.

Can we do better? The base and delta representation compresses pixel data effectively. Given the base and tag costs are constants, the cost of the representation then only depends on the number of bits to represent for the delta values. The next question is whether FVB is good enough? Or, can we find a better set of delta values, to minimize the bit-width of deltas?

3 ISOLATING THE ESSENTIAL BITS

The delta value and its representation are the two key elements in optimizing the tile bit-width. We show that, given any tile that consists of a region of pixels, the selection of its base can impact its delta values, and hence the number of bits needed for representing them. It is also important to consider the representation of the delta values synergistically, while choosing the tile's base. In this section, we discuss these in detail, specifically, the optimal base selections and delta representation, which collectively minimize the number of the bits needed for delta values.

3.1 Base Selection

Consider a pixel tile that contains a sequence of four pixel values, as shown in Fig. 5, distributed in the range of `[0, 255]`. Assuming that pixel-2 (p2) has the largest value among all pixels, pixel-3 (p3) has the lowest value, and other pixel values are in between p2 and p3. Once a base is selected, the deltas (d1 to d4) are also determined. Here, we define "delta" (with "sign") as the distance between the pixel value and the base value. We define k_1 and k_2 as the distance from the lowest pixel value to the base, and the distance from the largest pixel value to the base, respectively. In general, there are *four* choices for the base selection, as discussed below:

Table 1: The tag-encoding table in MidVB, denoted as MidVBTable. This tagging table follows the 2's complement representation.

Tag	Range	#Bit	Tag	Range	#Bit
0000	[0, 0]	0	0101	[-16, 15]	5
0001	[-1, 0]	1	0110	[-32, 31]	6
0010	[-2, 1]	2	0111	[-64, 63]	7
0011	[-4, 3]	3	1000	[-128, 127]	8
0100	[-8, 7]	4			

Choice-1: Base is outside the pixel value range: Fig. 5a shows a scenario, where the base is less than the minimum value in the tile, (this is a theoretical case simply based on pixel values). In this case, the bit-width of this tile is determined by the bit-width of k_2. This bit-width, however, has an unnecessary shift. In fact, there is no need to include region ❶ in these delta representations, as doing so may increase the bit-width of d2 as well as the bit-width of the tile. ZVB is an example of this scenario (except for the cases where the minimum pixel value is zero).

Choice-2: Base is inside the pixel value range (except minimal, mid, and maximal points): Fig. 5b shows a scenario where the base is selected from within the pixel value range. The bit-width of the tile is now determined by the maximum bit-width of deltas (k2 in this case). Note that, these deltas are scattered around the base value in both directions, which means that we need an "additional bit" to represent the sign of delta. The additional sign bit, however, causes unnecessary span of region ❷, making this representation sub-optimal. FVB is an example of this scenario. In fact, any pixel-value-based tagging scheme falls in this scenario.

Choice-3: Base is the minimal value or the maximal value: Fig. 5c shows a scenario, where the base is the minimal or maximal value. If the base is the minimal pixel value, all deltas are larger than or equal to zero; so the sign bit is saved. The bit-width of the tile is only determined by the bit-width of k2. Selecting the maximal as the base value is an equivalent to the minimal case. Using the minimal or maximal as the base, provides a better bit-width than the prior two cases, as it avoids the unnecessary shift and the unnecessary span. We denote this tagging scheme as **MinVB**.

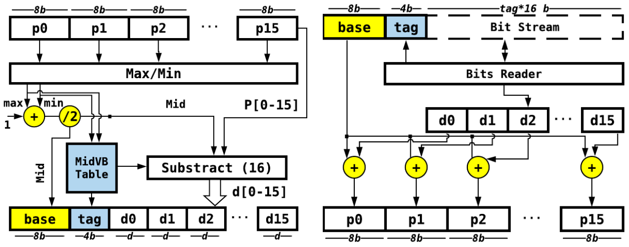
Choice-4: Base is at middle point (Mid): There exists another one possible choice for the base, which is the middle point of the pixel range, as shown in Fig. 5d. *Note that, the middle point of a range can be ambiguous if we define the middle point to be an integer.* If the pixel range ($|r|$) is an even value, there is only one middle point, which is the half-way point of the range. If $|r|$ is an odd value, then the middle point becomes a fractional value. Therefore, we define the middle point (Mid) as follows:

$$mid = floor((max + min + 1)/2). \tag{3.1}$$

By using this definition, we always round to the right integer of the middle point, which produces the 2's complement of the deltas. Fig. 5d shows a tagging scheme, **MidVB**, where the base is at the Mid of the pixel range. In MidVB, where r = max - min is the range of pixel values in a tile, k_1 and k_2 have the following relationship:

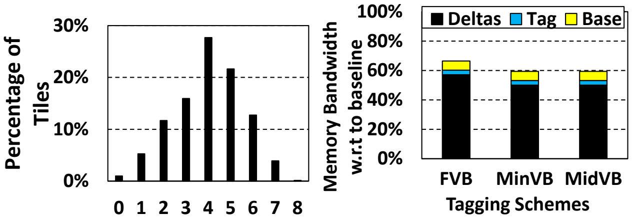
$$k_1 - k_2 = \begin{cases} 0, & \text{if } r \bmod 2 == 0 \\ 1, & \text{if } r \bmod 2 == 1 \end{cases} \tag{3.2}$$

Representation for Delta Values: As is well known, there are three ways to represent integers: *Unsigned Representation (UR), Sign-Magnitude Representation (SR),* and *Two's Complement Representation (TR).* Considering the four choices for base above, UR can only represent MinVB in choice-3, since other categories produce negative values. Compared to SR, TR has only one representation for 0, which saves more bit-width. This is beneficial especially for small range values. Let us consider the pixel range of `[0, 3]` for example: according to Eq. 3.1, Mid is 2, k1 is 2 (power of two, -2 with sign), and k2 is 1 (+1 with sign). In Sign-Magnitude, the bit-width of 2 bit (from k_1) and the additional sign bit together make the total bit-width 3. By adapting *TR* as shown in Table 1, we can reduce the bit-width for representing the range to 2. The following two Lemmas provide the theoretical basis for using the MidVB in 2's complement representation.



(a) MidVB Encoder **(b) MidVB Decoder**

Figure 6: Hardware design of a MidVB encoder and decoder.



(a) Bit-widths needed in MidVB. (b) Memory bandwidth needed.

Figure 7: (a) The bit-widths needed for representing the tile delta range. Pixel deltas *are calculated using Mid.* (b) Memory bandwidth demand (% with respect to 8-bpp baseline) for FVB, MinVB and MidVB, *normalized* to the baseline 8bpp representation.

Lemma 3.1. *To represent any pixel range, MidVB uses the same number of bits in 2's compliment representation, compared to MinVB in Unsigned Representation.*

Lemma 3.2. *In 2's complement Representation, using Mid (min, and max) as the base has the optimal bit-efficiency.*

Due to space limitations, we omit formal proofs of Lemma 3.1 and Lemma 3.2. Lemma 3.1 and Lemma 3.2 reveal that MinVB and MidVB are the optimal tagging schemes, in UR and TR, respectively. *Further, choosing some other value as the base could result in the optimal number of bits for deltas, but they all require different representations for the delta values, which are not UR, SR and TR.* A delta in other representations either needs a specific encoding table to interpret, or a dedicated arithmetic logic to process, leading, in both cases, to additional complexity and higher overhead.

3.2 Evaluation of MinVB and MidVB

We now present the implementation details of MidVB. Algorithm 1 shows the pseudo-code of MidVB encoder (we omit MinVB and MidVB's the similar decoder code here due to space limit.) The

Algorithm 1 MidVB Tagging Scheme.

```
Input T: A tile of pixels.
Output base, bitlen, deltas
procedure $MidVB(T)$
    $maxp \leftarrow T.max, minp \leftarrow T.min, deltas \leftarrow []$
    $base = \texttt{floor}((maxp + minp + 1)/2)$
    $bitlen = MidVBTable(minp - base, maxp - base)$
    for $i$ in $0$ to $Tile.len - 1$ do
        $deltas[i] \leftarrow Tile[i] - base$
    end for
end procedure
```

input to the algorithm is a `Tile` of pixels, and the output contains three parts: the `base` is the middle point; the tag representing `bitlen`, which is the number of bits needed to represent the delta range; and 16 `deltas`, where each delta is stored using `bitlen` bits. Fig. 6 illustrates the hardware encoder and decoder designs of MidVB for a (4x4) tile of 16 pixels.

Hardware Overhead: Using the Synopsys Design Compiler (H-2013.03-SP5-2) and 32nm technology library [67] to synthesize our hardware design, we find that the MidVB consumes only $0.003mm^2$ area. The latency is 2ns and the total power consumption for our MidVB encoder and decoder is 98uW and 360uW, respectively. Thus, the power consumption is negligible to the total system power (≈1-2W [23, 24, 55]).

We evaluate MidVB, MinVB and FVB using the same set of frames (R3 from Table 3, evaluation methodology is discussed in Sec. 5) and show the results in Fig. 7. The key observations are:

- **More tiles are represented using less bit-width:** Fig 7a shows the bit-width needed for representing the tile delta range for MidVB. Compared to FVB (Fig. 4b), we observe that the bit-width needed for the tile delta range shifts more towards 0.
- **MidVB/MinVB save memory bandwidth:** Fig. 7b shows the normalized bandwidth compared to the 8-bpp baseline. We compare MidVB with MinVB and FVB with respect to deltas, base and tag bits. We observe 18% reduction in the number of bits for delta values for both MinVB and MidVB, compared to FVB. MinVB and MidVB have the same bit efficiency as per Lemma 3.1. With the base and tag, the overall savings, compared to FVB, is around 12%, in terms of memory bandwidth. Note that, FVB reduces memory bandwidth consumption by 34% while MidVB and MinVB reduce it by 40% over the baseline.

4 DISTILLING THE ESSENTIAL BITS

As discussed in Sec. 3, further reduction in memory bandwidth cannot be achieved by any other tagging mechanisms. Therefore, to save additional memory bandwidth, we propose to approximate the video buffer data. Hence, we develop an approximation scheme, called **Distill** that optimizes the following three parameters: *bandwidth*, *application QoS control*, and *implementation overhead.*

Bandwidth: Sec. 3.2 presented a compression scheme (MidVB) that is tile-level optimal. Any sophisticated compression techniques such as PNG [5] will only provide marginal benefits as shown in Fig. 13. Pixel values are amenable to approximation with little or no impact on the later stages of the video processing pipeline. Potentially, by using approximation on the video buffer data, fewer bits are needed to represent a tile, reducing bandwidth consumption. However, we need to answer the question: *which bits can be dropped while minimizing the impact on video quality.*

Application QoS Control: For video applications, maintaining the required QoS is essential for satisfying user experience. There are two main perspectives of QoS, namely, "frame rate" and "quality of the frame". While saving memory bandwidth can potentially improve the frame rate, this should not deteriorate the video quality. In this paper, we use Peak Signal-to-Noise Ratio (PSNR) as our QoS Control metric. PSNR [22, 34] has been widely used as a quality metric and can be calculated as:

$$PSNR = 10 \cdot log_{10}(\frac{R^2}{MSE}), \tag{4.1}$$

where R is the maximum signal variation (255 in 8-bit image), and Mean Squared Error (MSE [33]) is defined as:

$$MSE = \frac{\sum_{[i,j]\in T}[I(i,j) - \hat{I}(i,j)]^2}{w \cdot h}, \tag{4.2}$$

where I is the original image; $\hat{I}$ is the approximated image; w and h are the width and height of a region T in the images (I and $\hat{I}$), respectively and (i,j) are the coordinates of the pixel in the image.

Implementation Overhead: A typical video application requires a frame rate of at least 60FPS, implying the processing and transmission of a frame need to be done in 16.66ms. Thus, both encoding/approximation and decoding of a tile need to be fast to maintain the required frame rate. Thus, high-compression codecs (H.264 [74], H.265 [65], etc.) schemes are not suitable for bandwidth saving optimization in mobile platforms due to their high computational demands.

The proposed Distill scheme considers the above three metrics and uses MidVB as the basic building block for significant bandwidth savings with negligible hardware and computation overheads, while maintaining QoS and video quality.

4.1 Approximation through Distilling

At a high level, the Distill design starts with the base and delta values generated by the MidVB mechanism for a tile and iteratively computes new base and delta values and based on the PSNR/MSE degradation and bandwidth demand, it discards the delta values obtained in the last iteration. Ignoring these delta values leads to approximation of the original video frame, but reduces memory bandwidth consumption.

Let us consider an example as shown in Fig. 8 to better understand the workings of Distill. Fig. 8 takes a tile from the Lenna image as shown in Fig. 2. In the first iteration ("I_0"), we represent the tile ⓿ in the form of base (b_0) ❶ and deltas (d) ❷ using MidVB, where the Mid is 94 and the deltas are in the range of `[-4,3]`. This dynamic range of deltas is no longer compressible, as we have proved in Lemma 3.2. To reduce memory bandwidth consumption further, we need to approximate the tile. Therefore, if we choose to drop the deltas now, and we represent the entire tile with a single value of 94 (this costs only 7 bits to represent 94), the approximated tile may be too lossy (MSE = 3.6).

Instead of dropping the "I_0" deltas ❷, we iteratively compute further bases and deltas to reduce the quality degradation. Note that, the deltas ❷ now contain both positive and negative values

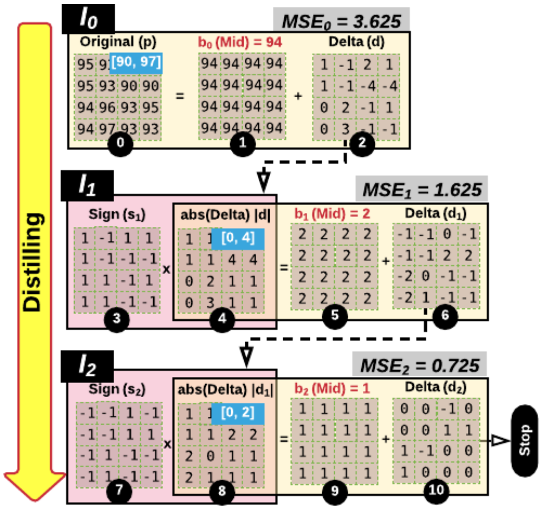


Figure 8: We show how Distill works on an example tile from Fig. 2. With bases b_0 and b_1, the peak error per pixel in this tile is within 2. Now, at "I_2", the representation cost is just 7 (bits for representing 94) + 2 (bits for representing 2) + 1 (bits for representing 1) + 32 (bits for sign) = 42 bits, reducing 86 bits from original 128 bits.

and cannot be used with MidVB. Intuitively, we can use *the absolute values* of the deltas for further iterations using MidVB. To do so, we extract the sign tile ("s_1") ❸, and apply MidVB on the absolute delta tile ❹. We now have a new base (b_1) ❺ and new deltas ❻ to represent the tile ❹ which along with the sign tile ❸ can regenerate the previous delta tile ❷. After "I_1", by using b0, b1 and the sign tile and dropping the delta tile ❻ at this iteration, we find a better approximation for the original tile than just using b_0 (MSE = 1.625). We further repeat this procedure to reduce the approximation of the tile, with more bases and more sign tiles. We stop iterating to find the new bases when the deltas of a given iteration have a range of [-1,1] which leads to a maximal error of 1 from the original pixel. Therefore, in Fig. 8, it takes 3 iterations to get the deltas within the range of [-1,1] (MSE = 0.725). The Algorithm 2 describes the Distill procedure.

Originally, a tile requires 128 bits (16 pixels x 8 bits). With MidVB, this requirement goes down to 55 bits with full precision leading a reduction of 57% in memory usage. Furthermore, by using Distill, with 3 iterations and dropping the last iteration's delta values ❿, we only need to store 42 bits (7 bits (b_0) + 2 bits (b_1) + 1 bit (b_2) + (16 bits/sign tile x 2 iterations)), leading to further savings in memory bandwidth of 24% over MidVB. The reconstruction of the tile can be simply performed by using Distill in reverse. Specifically, the approximated tile is ❾×❼×❸+❺×❸+❶.

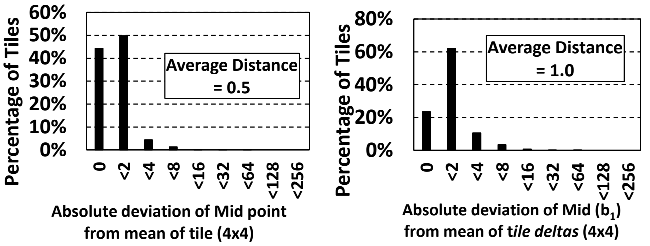
4.1.1 *Why Distill works?*

We now provide a theoretical discussion of how Distill controls QoS, guarantees convergence, and reconstructs the original pixel values.

Maintaining Quality: In our example, at "I_0", we use a single value to approximate the entire tile. Lemma 4.1 provides the optimal single value selection to minimize MSE. We also show the choice for our base selection (❶, ❺, ❾) to be very close to this optimal selection.

Algorithm 2 Distill Encoding.

```
Input Tile
Output header, bases, signs
procedure DistillVB(Tile, ErrCtrl)
    $s \leftarrow [1] \cdot len(Tile)$
    while max(|vals|) > ErrCtrl and max(|vals|) > 1 do
        $header[\lceil log(\mathbf{max}(|\text{vals}|))\rceil] \leftarrow 1$
        $base, tag, vals \leftarrow MidVB(|vals|)$
        $s = s * sign(vals)$
        $signs.push(s)$
        $bases.push(base)$
    end while
end procedure
```



(a) Distribution of distances from Mid to the mean of a tile.

(b) Distribution of distances from Mid to the mean of *tile deltas*.

Figure 9: Mid as a good estimation to mean.

Lemma 4.1. *If we use a scalar value x to approximate a pixel region, the value x^* that minimizes the MSE is the mean of all the pixel values in the tile.*

Proof.

$$MSE = \frac{\sum_{[i,j]\in T}[I(i,j) - x]^2}{w \cdot h}. \tag{4.3}$$

In order to minimize MSE and since I(i,j) is non-negative, we equate the derivative of MSE with respect to x to be equal to 0. By using algebraic manipulation to find the value of x, we have,

$$x^* = x = \sum_{w,h} I(i,j)/(w \cdot h), \tag{4.4}$$

where x is the mean of all the pixel values in a tile. □

Note that, as discussed in Sec 3, we already have a hardware unit to compute MidVB, and we would like to reuse it for Distill as well. For a small range of pixels (within a tile), we empirically find that, Mid is a good approximation of the mean. Fig. 9a quantifies the distribution of the distance of Mid values from the mean values for all the tiles. We see that, around 94% of Mid is within the range of 1 pixel from the mean, and the mean distance from the Mid to the mean is 0.5 (within one pixel value). Thus, using Mid for b0 in "I_0" as shown in Fig. 8 is a good "approximation" for the base value that results in minimal MSE and higher PSNR.

For all subsequent iterations after the first, we extract a sign tile (s) and generate a base (b). To show the efficacy of our choices of sign (s) and base (b), we represent MSE in terms[2] of s, b and d, and find the optimal values for s^* and b^* for minimizing MSE.

[2] We assume s is a 1-bit vector representing either "+1" or "-1", b is any positive scalar value and d is delta values that we found in Lemma 4.1.

We represent a tile of pixels, $\vec{p}$, in image I, in the form of:

$$\vec{p} = \vec{1} \cdot b_0 + \vec{d} = \vec{1} \cdot b_0 + (\vec{s} \cdot b_1 + \vec{d_1}), \tag{4.5}$$

where b_0 is the base value found using MidVB and d_1 is the second-order deltas. Note that, s and b_1 are the independent variables for which we need find the optimal values to minimize MSE.

Now, we approximate the pixel values by discarding the second order deltas and representing the approximated pixel values with just the first base b_0, b_1 and s,

$$\vec{\hat{p}} = \vec{1} \cdot b_0 + \vec{s} \cdot b_1, \tag{4.6}$$

We can now represent MSE_1 of $\vec{\hat{p}}$ as:

$$MSE_1 = \frac{\sum_{i \in w \cdot h} (d[i] - s[i] \cdot b_1)^2}{w \cdot h} \tag{4.7}$$

Recall that, we need to find $\vec{s}$ and b_1 that together minimize MSE_1 (Eq. 4.7). We first consider the selection of $\vec{s}$. Therefore, we need to find the number of '+1's and '-1's in $\vec{s}$. To do this, we partition $\vec{d}$ into two parts, where $\vec{d_+}$ are the *non-negative values* including zeros, and $\vec{d_-}$ are the negative values. Let the number of *non-negative values* in $\vec{d}$ be m_+ and the number of the negative values be m_-. We reorder the non-negative values to be the first m_+ elements in $\vec{d}$, followed by the negative values. Let us also denote the number of '+1' in $\vec{s}$ as k. Note that, if $d[i]$ or b_1 is 0, $s[i]$ does not affect MSE_1, and in both cases, we set $s[i]$ as '+1'. Further, we reorder the '+1's as the first k elements in the $\vec{s}$, followed by '−1's.

$$\vec{s} = [\overbrace{+1, \cdots, +1}^{k}, \overbrace{-1, \cdots, -1}^{n-k}] \tag{4.8}$$

$$\vec{d} = [\overbrace{d_+, \cdots, d_+}^{m_+}, \overbrace{d_-, \cdots, d_-}^{m_-}] \tag{4.9}$$

LEMMA 4.2. *Based on Eq. 4.7, MSE_1 is minimal when $k = m_+$.*

PROOF. We prove this by contradiction. Assume we flip $s[m_+]$ to '+1'. Since the corresponding $d[m_+]$ is a negative value, this makes the k_{th} term in the MSE_1 larger than or equal to the original term which increases MSE_1:

$$(d[m_+] - b_1)^2 \geq (d[m_+] + b_1)^2, d[m_+] < 0. \tag{4.10}$$

Also, if we flip $s[m_+ - 1]$ to −1, since the corresponding $d[m_+ - 1]$ is a non-negative value, this makes the m_{+th} term in the MSE_1 larger than or equal to the original term,

$$(d[m_+ - 1] + b_1)^2 \geq (d[m_+ - 1] - b_1)^2, d[m_+ - 1] > 0, \tag{4.11}$$

which also increases MSE_1. Therefore, MSE_1 is minimized only when k is m_+. □

Lemma 4.2 shows, while d[i] is negative, its corresponding s[i] is -1, and similarly, when d[i] is positive, s[i] is +1. Under this notion, we can simplify Eq. 4.7 into:

$$MSE_1 = \frac{\sum_{i \in w \cdot h} (|d[i]| - b_1)^2}{w \cdot h}. \tag{4.12}$$

Now, based on the selection of s, and from Eq. 4.12 and Lemma 4.1, we conclude that the optimal b_1 is the mean of $|d|$, where $|d|$ is the absolute values of $\vec{d}$. Recall that, similar to the scenario of b_0, we reuse MidVB rather than use mean for the iterative base computations. Fig. 9b shows that mid_1 is a good approximation of the optimal b_1.

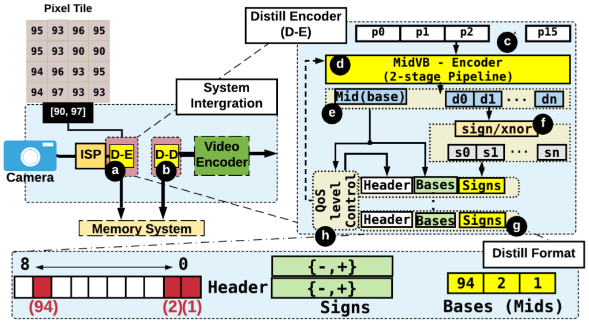


Figure 10: Details of the Distill encoder shown with a (4X4) tile.

Convergence Guarantee: Distill is an iterative procedure, and therefore, it requires a termination condition. Note that, $\vec{\hat{p}}$ (Eq. 4.6 - approximation using two Mids) has lower MSE than $\vec{1} * b_0$ (approximation using only Mid_0). This is because by selecting Mid_1, it bounds the second-order deltas (of any pixel range) into a smaller range (`[-64, 63]`), compared to the range of first-order deltas (`[-128, 127]`). At each iteration, the range of the output of MidVB narrows down from the input. This is because by selecting Mid as the base, it essentially halves the range of the original input range, until it narrows to a range of ±1. Further iteration will not affect the output range, and therefore, Distill stops at this iteration(10) in Fig. 8). We denote the output values of the last iteration with range `[-1, 1]` as the *residual bits* ($\vec{rd}$). With a starting range of `[0, 255]`, Distill can at most iterate for 8 times, which guarantees an upper bound. In practice, it converges much faster. Fig. 11 shows the number of iterations needed for various tiles to converge. On an average, 4 Distill iterations are needed for a raw video (R3) with over 1.8 billion tiles to completely converge all deltas to a range of `[-1, 1]`.

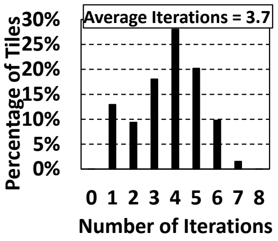


Figure 11: Iteration count for convergence.

Reversibility: To recover the original pixel values from the *full* Distill outputs, we perform the following computation:

$$\vec{p} = \sum_{i=0}^{l} \vec{\hat{s}}_i \cdot mid_i + rd, \tag{4.13}$$

where l is number of iterations for Distill to terminate. For faster computation during decoding, we transform sign s_i in to $\vec{\hat{s}}_i$ (line 6 in Algorithm 2) by computing *xnor* of all the previous values of s from {1,i-1} which can then be directly used during decoding.

4.2 Implementation Details of Distill

We discuss the implementation details of Distill. Due to space constraints, we discuss the Distill encoder design only.

Input and Outputs: We reuse MidVB interface (input and outputs) and each iteration of Distill uses the output of MidVB as the input of each Distill iteration. Initially, the input of Distill are the original pixel values. The output of each iteration includes Mid (used as base), its associated signs, and the new deltas. As $\vec{p}$ is the original non-negative pixel values, $\vec{s}_0$ then defaults to $\vec{1}$, which we can safely omit in transmission. To summarize, the outputs of Distill are $\vec{mid}$, a vector of sign bits vector ($\vec{s}$), and residual bits $\vec{rd}$.

Distill Format: We propose the Distill tile format, as shown in Fig. 10. There are three regions in our Distill format:

- **Header:** This is a 9 bit-map to indicate base information, the number of 1's in the bitmap indicates the number of bases, and each 1's position in the bitmap indicates the number of bits needed for the base. From this header field, one can infer the number of bits used for this tile.
- **Bases:** Bases are stored as unsigned values using the exact bits required to represent them.
- **Signs:** Each sign array is affiliated to one base. Note that, the sign array for the first base is all 1s. The total bits for the signs field is also inferred from the header.

Here, we opt to not carry the residual bits and therefore, introduce approximation. We use the lossless MidVB to replace Distill format (Distill-0), if lossless is required.

Integrating Distill Encoder (DE) and Distill Decoder (DD): In Fig. 10, we show the additional hardware required to support Distill. We attach a Distill Encoder ⓐ after the ISP outputs the raw video frames, but before the video buffer and a Distill Decoder ⓑ after the video buffer but before the video encoder. DE compresses and approximates the video buffer data and stores the Distilled format in the video buffer, while DD decodes the Distilled Format and feeds the approximated video frame data into the encoder.

Distill Hardware Design: The raw pixel data is fed to the DE as shown in ⓒ (Fig. 10). We reuse the MidVB ⓓ hardware from Fig. 6 for the base computation. The outputs of the MidVB ⓓ are the base and the deltas ⓔ. The signs of the deltas are *xnor-ed* with the previous iteration's signs ⓕ, and along with the base are stored in the video buffer. The tiles are stored in-order. This is because, at the DD side, the order of reading tiles needs to be the same as the original stream. However, in a scenario where a tile A (which is followed by tile B) may need more MidVB iterations (based on the quality control ⓗ, line-3 in Algorithm 2) during which time the tile B may arrive. In this case, we store B temporarily in the buffer ⓖ, until A finishes all its iterations. Note that we provide enough buffers in DE to account for the highest number of iterations to be performed by Distill while ensuring that the overheads do not show up in the critical path. For video R3, Distill-3 needs 2.1 iterations (4.2ns) to compress and approximate the tile, on average.

5 EVALUATION

In this section, we first detail the evaluation platform and workloads. Then, we present the experimental results of MidVB and compare with other memory saving algorithms such as content-cache (CC) (we use a 2KB 4-way set-associative LRU cache [84]), First Value Based Compression (FVB in Sec. 3) [53] and PNG. We evaluate the effects of Distill towards memory bandwidth and video quality (PSNR). Furthermore, we compare Distill against three other approximation-based memory saving schemes. To understand the impact that MidVB[3] has on encoded video, we perform sensitivity analysis to show memory bandwidth benefits.

Table 2: System Configuration.

Parameter		Value
SoC		Google Android Emulator [20] GemDroid [7]

DRAM		2GB, 2 channels; 1 rank per channel;
		8 Banks per rank; 800Mhz;
		tCL,tRP,tRCD = 12,18,18 ns

Video Encoder [82]		H.264 (HighProfile) FPS = 60, GOP = 30 Constant bitrate = 68Mbps, 2 B-Frames 4k = 3840x2160; 2k = 1920x1080
Video Decoder		4k/2k resolution, 60FPS, H.264/H.265
Distill Auxiliaries	Encoder	MidVB-e: 2ns; 0.3mW
	Decoder	MidVB-d: 2ns; 0.06mW
	Buffer	1KB SRAM buffer; ~ 0.5mW

Table 3: Workload videos.

Key	Video Name	Description	#Frames
R0	Boat [40]	Boat View Video	300
R1	Crosswalk [41]	Street Video	300
R2	RitualDance [44]	Dancing Video	600
R3	Driving [42]	Driving Video	1199
R4	Fountain [47]	Building View Video	1199
R5	PierSeaside [43]	Sky View Video	1199
R6	RollerCoaster [45]	Flag View Video	1199
R7	Tango [46]	Dancing Video	294
R8	TunnelFlag [48]	Tunnel View Video	600
R9	WindAndNature [49]	Nature View Game	1199
R10	Portrait [39]	Face Closeup Video	1199
R11	Bosphorus [14]	Bridge View Video	600
R12	HoneyBee [10]	Animal Video	600
R13	Jockey [13]	Sports Video	600
R14	ReadySetGo [12]	Horse Racing Video	600
R15	ShakeNDry [11]	Animal Video	300
R16	YachtRide [15]	Luxury Yacht Video	600
R17	DOTA2 [69]	Video Game	3602
R18	Hearthstone [70]	Video Game	3602
R19	StarCraft [71]	Video Game	3602

5.1 Workloads and Experimental Setup

Workloads: We evaluate our designs on twenty raw videos as listed in Table 3. To cover a variety of video scenes, we include a set of indoor and outdoor videos (R0-R9, R11, R16), sports videos (R13-R14) and gaming videos (R17-19).

Experimental Platform: We use FFmpeg [18] to extract original frames from the raw videos. We instrument the publicly available GemDroid simulator [7] to feed the extracted raw data into the video buffer based on the camera and video IP traces. We use DRAMSim2 [58] to model the video buffer and capture its timing and energy consumption. The video buffer is modeled as an LPDDR3 DRAM [35] (commonly used in edge devices). The MidVB and Distill hardware units have been modeled accurately as described in Table 2. In total, we evaluated more than 22700 frames to show the efficacy of our schemes.

Metrics: To quantify the performance of various schemes, we use *memory bandwidth*, *PSNR* (video quality), and *energy consumption* for our evaluation. For PSNR, we compare the video frame of the *H.264* decoded frame (we use this as the baseline for video quality measurements) data against the *Distill+H.264* decoded frame data.

5.2 Results

We present the results in four parts: (i) memory bandwidth consumption with MidVB, (ii) memory bandwidth and energy consumption with Distill, (iii) quantitative comparison of Distill with prior approximation schemes, and (iv) a sensitivity study over 10 encoded videos (33000 frames).

Memory bandwidth savings with MidVB: Fig. 13 shows the memory bandwidth consumption when utilizing our lossless memory saving scheme MidVB with respect to the original 8-bpp representation. It also compares MidVB with other schemes such as CC,

[3] We refer to MidVB as Distill-0 and use them interchangeably.

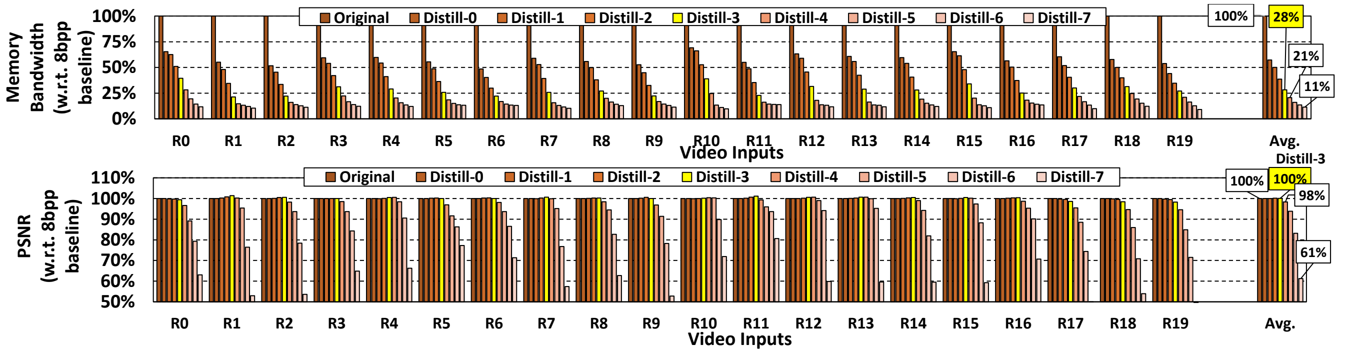


Figure 12: Overall memory bandwidth utilization (lower the better) and PSNR (higher the better) results of Distill. Original is 8bpp pixel representation. MidVB represents Distill-0. Distill-i ensures the maximal error of a pixel within 2^{i-1}. Distill-3 saves 72% bandwidth.

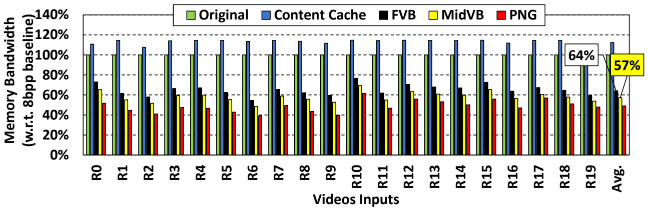


Figure 13: Memory bandwidth consumption for various lossless memory saving schemes.

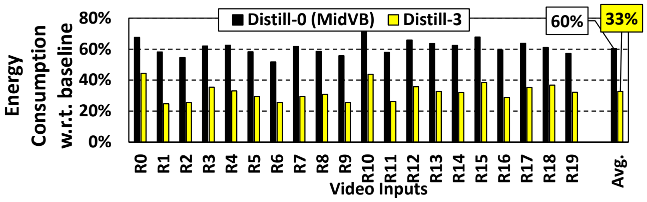


Figure 14: Overall energy results of Distill-0 (MidVB) and Distill-3.

FVB and PNG. On an average, MidVB reduces memory bandwidth consumption by 43%, FVB by 36% and PNG by 51%. Note that, CC does not work well in all raw videos. This is because content cache can only exploit value equality in the decoding pipeline, however, raw videos have very limited value equality, as shown in Fig. 3a. Thus, CC performs worse than the baseline (112%), due to its metadata overhead. MidVB shows uniformly better bit-efficiency compared to FVB in all raw video inputs (Lemma 3.2). Compared to FVB, MidVB provides 11% improvement in bit efficiency, on an average, which translates to 7% memory bandwidth savings. Note that, PNG outperforms MidVB by 8%, which shows the upper bound of exploiting value similarity. However, PNG requires orders of magnitude more computations than MidVB which makes it *unsuitable* for real-time scenarios.

Bandwidth and energy consumption with Distill: We present the bandwidth consumption, video quality and energy consumption of Distill for all twenty raw videos in Fig.12. We present eight levels of Distill (0-7), to quantify the benefits at different quality levels, where MidVB represents Distill-0 (lossless). Distill-i is the i_{th} level of the Distill, e.g., Distill-1 approximates the original tile by ensuring the absolute error of each pixel is within the range of 1. Distill-3 ensures the max errors of each pixel are within the range of 4. Fig. 12 shows the impact of Distill in terms of bandwidth consumption and PSNR. We observe that, on an average, with Distill-0 to Distill-3, the quality of the video output does not degrade; while reducing the memory bandwidth by 72%. Further, distillation causes the PSNR to drop sharply. Distill-7 provides the highest level of approximation and reduces memory bandwidth by 89%, but degrades PSNR by 39%. This is due to the fact that increasing Distill level (4 and beyond) increases the tolerance of approximation at an exponential rate which leads to more quality degradation. We select Distill-3 to be our design choice for the rest of our evaluations. Fig. 14 shows the energy consumption of Distill-3. On an average, energy consumption reduces by 67% with respect to 8-bpp baseline. Distill provides 27% more energy reduction compared to MidVB. Note that we gain energy savings not only because of bandwidth savings, but also due to the reduction in number of memory accesses and memory pages used. The data is condensed and stored in the video buffer in Distill format (Fig. 10).

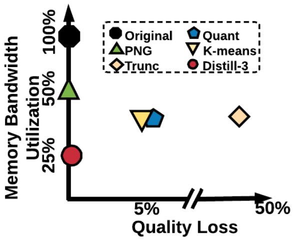


Figure 15: Overall comparison.

Comparison with other approximation techniques: Fig. 15 shows, on an average, how Distill-3 stacks up against three other approximation-based memory saving schemes in terms of memory bandwidth and quality loss for across all the twenty raw videos. For fair comparison, we tune the three schemes to reduce the bandwidth consumption by a similar amount to that of Distill-3 and compare their video quality (PSNR).

- **Truncation Bits (Trunc):** This scheme discards the least-significant bits of a pixel and sends only the most-significant three bits to the video encoder. The memory bandwidth consumption reduces by 62.5%, while the PSNR drops by 38%.
- **Dynamic Color Quantization (Quant) [6]:**This scheme provides a way to quantize the dynamic value range (minimal and maximal of the tile). We tune this scheme to provide bandwidth savings of 56%, while incurring a loss of 5% in PSNR.
- **Color Quantization using k-means (KMeans) [59]:**This scheme provides a k-means approach to approximate the image by finding representative colors of an image. We configure the scheme to use k-means to find 1536 of most representative pixels (512 pixels x 3 channels) of the raw frame image. This provides 62.5% bandwidth savings, while incurring a PSNR loss of 4.5%.

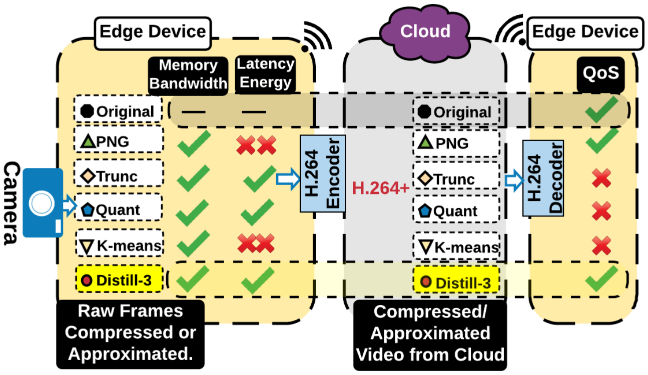


Figure 16: Distill vs other contemporary memory optimizations.

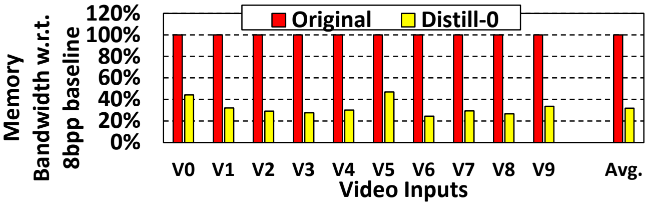


Figure 17: Memory bandwidth consumption of encoded videos with Distill-0 (MidVB).

Fig. 15 further quantifies the memory savings and QoS requirements of various schemes. Fig. 16 compares Distill with other approximation schemes. Distill-3 achieves the best tradeoffs between memory bandwidth savings, latency, energy consumption as well as the end-user QoS requirement.

Sensitivity Study: Distill-0, which is lossless, can also be applied to encoded videos (already lossy). Fig. 17 shows that, on average, Distill-0 saves 68% memory bandwidth with respect to the 8-bpp representation. Distill can also support random access (if needed) to each macroblock by using an additional address for each of the compressed macroblock (including 3 channels). An address to point to the macroblock needs 28 bits (25bits ($log(3840x2160x3)$) + 3 bits) for indexing. Therefore, the additional memory demand to support random access is 7.3% (28/(16*8*3)).

6 RELATED WORK

Value Based Optimizations: Many prior studies have explored value based optimizations in CPUs, GPUs, Memory, I/O Buses, Caches and register domains [4, 29, 31, 32, 60, 79, 80, 85, 86]. Lee et al. [29] observed that multiple data elements within a single cache line/sector are often similar to one another and exploited this characteristic to encode DRAM transfers such that there is only one reference copy of the data, with remaining similar data items being encoded predominantly as 0 values. Liu et al. [32] present an approach to optimizing the on-chip interconnect power by exploiting the value locality in data transfers between processors. Yang et al. [79] showed that, by exploiting the characteristic of memory reference locality, switching activity on the address bus can be reduced by as much as 66%. Zhang et al. [84] propose content caching to exploit tile-level value equality in encoded frames. These prior efforts do not exploit value similarity for approximating data. MidVB and Distill are able to exploit value similarity to efficiently compress and approximate the data.

Compression and Approximation Algorithms: There is a wide body of work on compression and approximation of data that employs base selection, compression, approximation and quantization [1, 9, 16, 17, 25, 27, 28, 36, 52–54, 61, 72, 77, 78, 90]. KMeans clustering [28] provides the least quality loss but has huge computational overheads which makes it impractical for real-time applications.

Miguel et al. [36] designed an approximate cache that associates the tags of multiple similar blocks onto a single data block. Thereby, increasing the effective cache capacity. Jevdjic et al. [25] proposed VideoApp, an efficient compression mechanism for already compressed and encrypted videos. Kim et al. [27] develop a bit-plane compression algorithm that targets memory blocks with homogeneously-typed data. Phekimenko et al. [53] proposed a cache compression technique, called Base-Delta-Immediate, which is the foundation of ZVB and FVB (Sec. 2). Prior works do not relate the intrinsic value similarity in the video frames to the degradation in video quality that accompanies the approximation. [3] targets only the decoding pipeline and not the encoding pipeline. To the best of our knowledge, this is the first work to control the compression and approximation of raw (pre-encoded) video frames based on the video quality while minimizing overheads.

Energy Optimization Techniques: Multiple energy optimization techniques have been studied on edge devices [2, 21, 26, 30, 50, 51, 56, 57, 62, 64, 66, 81, 83, 87]. Several proposals by Zhu et al. [88, 89], Nachiappan et al. [38] and Yedlapalli et al. [81] have proposed mobile-SoC energy optimization through scheduling and virtualization. However, they do not reduce the number of memory accesses. Note that all these energy saving techniques discussed above are orthogonal to the proposed scheme in this work.

7 CONCLUSIONS

Improving the efficiency of memory usage is critical for supporting growing class of video applications in edge devices with limited resources and energy constraints. In this paper, we present two complementary techniques to reduce the memory bandwidth and energy consumption of camera-based edge devices. We exploit the concept of value locality, prevalent in video frame data in designing our first scheme, MidVB, to achieve lossless compression on raw video frames. MidVB is able to reduce the memory bandwidth and energy consumption by 43% and 40%, respectively. The second scheme, Distill, further reduces the memory footprint of the raw video frames by approximating the non-essential bits. It achieves this by intelligently iterating over MidVB output and discards the least essential bits from the raw videos, with minimal loss in video quality. This helps Distill to significantly reduce memory bandwidth and energy consumption by 72% and 67%, respectively. Due to the immense popularity of camera-based applications such as VR, AR, autonomous driving, drones, etc., our schemes can have considerable impact on these edge devices.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants #1763681, #1629915, #1629129, #1317560, #1526750, and a DARPA/SRC JUMP grant. We would also like to thank Jack Sampson for his feedback on this paper.

REFERENCES

[1] A. R. Alameldeen and D. A. Wood. 2004. Adaptive Cache Compression for High-performance Processors. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 212–223.

[2] Antonios Antoniadis, Chien-Chung Huang, and Sebastian Ott. 2015. A Fully Polynomial-time Approximation Scheme for Speed Scaling with Sleep State. In *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’15).* 1102–1113.

[3] ARM. 2013. Arm Frame Buffer Compression (AFBC) . ”https://developer.arm.com/architectures/media-architectures/afbc”.

[4] Saisanthosh Balakrishnan and Gurindar S. Sohi. 2003. Exploiting Value Locality in Physical Register Files *(MICRO 36).* 265–.

[5] T. Boutell. 1997. PNG (Portable Network Graphics) Specification Version 1.0.

[6] M. Budagavi and M. Zhou. 2008. Video Coding using Compressed Reference Frames. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing.* 1165–1168.

[7] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2014. GemDroid: A Framework to Evaluate Mobile Platforms *(SIGMETRICS).* 355–366.

[8] Cisco. 2019. Internet of Things At a Glance. ”https://bit.ly/2vpGRxp”.

[9] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019), 640–653.

[10] Digiturk. 2013. Bee Harvesting Blue Flowers Video. ”https://bit.ly/2CFy0vQ”. [Online; accessed March-2019].

[11] Digiturk. 2013. Dog Drying Himself Video. ”https://bit.ly/2FvZSmC”. [Online; accessed March-2019].

[12] Digiturk. 2013. Horse Racing Track Video. ”https://bit.ly/2Wtqmw5”. [Online; accessed March-2019].

[13] Digiturk. 2013. Horse Racing Video. ”https://bit.ly/2FBym8G”. [Online; accessed March-2019].

[14] Digiturk. 2013. Luxury Yacht Video. ”https://bit.ly/2UYQBdd”. [Online; accessed March-2019].

[15] Digiturk. 2013. Wavy Water Video. ”https://bit.ly/2TxXVex”. [Online; accessed March-2019].

[16] Wei Ding and Bede Liu. 1996. Rate Control of MPEG Video Coding and Recording by Rate-quantization Modeling. *IEEE Trans. Cir. and Sys. for Video Technol.* (1996), 12–20.

[17] Magnus Ekman and Per Stenstrom. 2005. A Robust Main-Memory Compression Scheme *(ISCA ’05).* 74–85.

[18] FFmpeg. 2019. FFmpeg - A Complete, Cross-platform Solution to Record, Convert and Stream Audio and Video. ”https://ffmpeg.org/”.

[19] Google. 2016. VP9 Bitstream & Decoding Process Specification. ”https://bit.ly/23vYrrw”.

[20] Google. 2019. Run Apps on the Android Emulator. ”https://bit.ly/2k618T2”.

[21] E. G. Hallnor and S. K. Reinhardt. 2005. A Unified Compressed Memory Hierarchy. In *11th International Symposium on High-Performance Computer Architecture.* 201–212.

[22] A. Hore and D. Ziou. 2010. Image Quality Metrics: PSNR vs. SSIM. In *2010 20th International Conference on Pattern Recognition.* 2366–2369.

[23] Intel. 2018. Intel Edison Compute Module Hardware Guide. ”https://intel.ly/2uTPkJr”.

[24] Intel. 2018. Intel Joule Module Expansion Board Hardware Guide. ”https://intel.ly/2LNWyoS”.

[25] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S. Malvar. 2017. Approximate Storage of Compressed and Encrypted Videos *(ASPLOS).* 361–373.

[26] O. Kayiran, A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. 2016. μC-States: Fine-grained GPU datapath power management. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT).* 17–30. https://doi.org/10.1145/2967938.2967941

[27] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures *(ISCA).* 329–340.

[28] K. Krishna and M. Narasimha Murty. 1999. Genetic K-means algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* (1999), 433–439.

[29] D. Lee, M. O’Connor, and N. Chatterjee. 2018. Reducing Data Transfer Energy by Exploiting Similarity within a Data Transaction. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 40–51.

[30] Kangmin Lee, Se-Joong Lee, and Hoi-Jun Yoo. 2004. SILENT: Serialized Low Energy Transmission Coding for On-chip Interconnection Networks. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design (ICCAD ’04).* 448–451.

[31] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. 2018. Semantic-Aware Virtual Reality Video Streaming. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys ’18).* 21:1–21:7.

[32] C. Liu, , and M. Kandemir. 2004. Optimizing Bus Energy Consumption of On-chip Multiprocessors Using Frequent Values. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings.* 340–347.

[33] Mathworks. 2019. Compute peak signal-to-noise ratio (PSNR) between images - Simulink. ”https://www.mathworks.com/help/vision/ref/psnr.html”.

[34] MathWorks. 2019. Peak Signal-to-Noise Ratio (PSNR) - MATLAB psnr. ”https://bit.ly/2ORHZTE”.

[35] Micron. 2019. Micron EDF8164A1MA Specs. ”https://bit.ly/2KitR7t”.

[36] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. 2015. Doppelgänger: A Cache for Approximate Computing. In *MICRO.* 50–61.

[37] MPEG. 2019. The Moving Picture Experts Group. ”https://bit.ly/2OOchql”.

[38] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das. 2015. VIP: Virtualizing IP chains on handheld platforms. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA).* 655–667.

[39] NETFLIX. 2013. Digital Video Sequence ”Beauty”. ”https://bit.ly/2uKQgzu”. [Online; accessed March-2019].

[40] NETFLIX. 2015. Digital Video Sequence ”Boat”. ”https://bit.ly/2CEsQQN”. [Online; accessed March-2019].

[41] NETFLIX. 2015. Digital Video Sequence ”Crosswalk”. ”https://bit.ly/2Ytdsjx”. [Online; accessed March-2019].

[42] NETFLIX. 2015. Digital Video Sequence ”Driving POV”. ”https://bit.ly/2U0sHBS”. [Online; accessed March-2019].

[43] NETFLIX. 2015. Digital Video Sequence ”Pier Seaside”. ”https://bit.ly/2WqXEvS”. [Online; accessed March-2019].

[44] NETFLIX. 2015. Digital Video Sequence ”Ritual Dance”. ”https://bit.ly/2FDsPyk”. [Online; accessed March-2019].

[45] NETFLIX. 2015. Digital Video Sequence ”Roller Coaster”. ”https://bit.ly/2UW4Uzt”. [Online; accessed March-2019].

[46] NETFLIX. 2015. Digital Video Sequence ”Tango”. ”https://bit.ly/2utBpJB”. [Online; accessed March-2019].

[47] NETFLIX. 2015. Digital Video Sequence ”Toddler Fountain”. ”https://bit.ly/2HFBPVX”. [Online; accessed March-2019].

[48] NETFLIX. 2015. Digital Video Sequence ”Tunnel Flag”. ”https://bit.ly/2Hyten0”. [Online; accessed March-2019].

[49] NETFLIX. 2015. Digital Video Sequence ”Wind and Nature”. ”https://bit.ly/2JJr1rN”. [Online; accessed March-2019].

[50] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT ’16).* ACM, New York, NY, USA, 31–44. https://doi.org/10.1145/2967938.2967940

[51] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2019. Opportunistic Computing in GPU Architectures. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA ’19).* ACM, New York, NY, USA, 210–223. https://doi.org/10.1145/3307650.3322212

[52] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. 2016. A Case for Toggle-aware Compression for GPU Systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 188–200.

[53] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT ’12).* 377–388.

[54] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2013. Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 172–184.

[55] Raspberry Pi. 2018. Rasperry Pi 3 Model B. ”https://bit.ly/1WTq1N4”.

[56] P. V. Rengasamy, H. Zhang, N. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. 2017. Characterizing diverse handheld apps for customized hardware acceleration. In *2017 IEEE International Symposium on Workload Characterization (IISWC).* 187–196. https://doi.org/10.1109/IISWC.2017.8167776

[57] P. V. Rengasamy, H. Zhang, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. 2018. CritICs Critiquing Criticality in Mobile Apps. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 867–880. https://doi.org/10.1109/MICRO.2018.00075

[58] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* (2011), 16–19.

[59] Scikit. 2019. Color Quantization using K-Means. ”https://bit.ly/2FKhxHe”. [Online; accessed March-2019].

[60] H. Seol, W. Shin, J. Jang, J. Choi, J. Suh, and L. Kim. 2016. Energy Efficient Data Encoding in DRAM Channels Exploiting Data Value Similarity. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 719–730.

[61] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. 2014. MemZip: Exploring Unconventional Benefits from Memory Compression. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 638–649.

[62] A. Sharifi, W. Ding, D. Guttman, H. Zhao, X. Tang, M. Kandemir, and C. Das. 2017. DEMM: A Dynamic Energy-Saving Mechanism for Multicore Memories. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 210–220.

[63] Snapchat. 2019. Snapchat - the Fastest Way to Share A Moment. ”https://www.snapchat.com/l/en-gb/”.

[64] Yanwei Song and Engin Ipek. 2015. More is Less: Improving the Energy Efficiency of Data Movement via Opportunistic Use of Sparse Codes. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. 242–254.

[65] G. J. Sullivan, J. Ohm, W. Han, and T. Wiegand. 2012. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology* (2012), 1649–1668.

[66] D. C. Suresh, B. Agrawal, J. Yang, and W. A. Najjar. 2009. Tunable and Energy Efficient Bus Encoding Techniques. *IEEE Trans. Comput.* (2009), 1049–1062.

[67] Synopsys. 2019. Design Compiler: RTL Synthesis. ”https://bit.ly/2D2XC5X”.

[68] Tiktok. 2019. Tiktok: Real Short Videos. ”https://www.tiktok.com/”. [Online; accessed March-2019].

[69] Twitch. 2015. Game Video Sequence ”DOTA2”. ”https://bit.ly/2UhX7i6”. [Online; accessed March-2019].

[70] Twitch. 2015. Game Video Sequence ”Hearthstone”. ”https://bit.ly/2uv4BA0”. [Online; accessed March-2019].

[71] Twitch. 2015. Game Video Sequence ”STARCRAFT”. ”https://bit.ly/2Owufxy”. [Online; accessed March-2019].

[72] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. 2015. A Case for Core-assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA ’15)*. 41–53.

[73] Mike Wakin. 2019. Standard Test Images. ”https://www.ece.rice.edu/ wakin/images/”. [Online; accessed March-2019].

[74] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. 2003. Overview of the H.264/AVC Video Coding Standard. *IEEE Transactions on Circuits and Systems for Video Technology* (2003), 560–576.

[75] Wikipedia. 2019. HSL and HSV. ”https://bit.ly/1L6R7zM”.

[76] Wikipedia. 2019. YCrCb. ”https://bit.ly/2i2u3u3”.

[77] Haichuan Yang, Yuhao Zhu, and Ji Liu. 2018. ECC: Energy-Constrained Deep Neural Network Compression via a Bilinear Regression Model. *CoRR* (2018). http://arxiv.org/abs/1812.01803

[78] Haichuan Yang, Yuhao Zhu, and Ji Liu. 2019. Energy-Constrained Compression for Deep Neural Networks via Weighted Sparse Projection and Layer Input Masking. In *International Conference on Learning Representations*.

[79] Jun Yang, Rajiv Gupta, and Chuanjun Zhang. 2004. Frequent Value Encoding for Low Power Data Buses. *ACM Trans. Des. Autom. Electron. Syst.* (2004), 354–384.

[80] Jun Yang, Youtao Zhang, and Rajiv Gupta. 2000. Frequent Value Compression in Data Caches *(MICRO 33)*. 258–265.

[81] P. Yedlapalli, N. C. Nachiappan, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. 2014. Short-Circuiting Memory Traffic in Handheld Platforms. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 166–177.

[82] Youtube. 2019. Recommended Upload Encoding Settings. ”https://bit.ly/1dM6dVq”. [Online; accessed March-2019].

[83] Haibo Zhang, Prasanna Venkatesh Rengasamy, Nachiappan Chidambaram Nachiappan, Shulin Zhao, Anand Sivasubramaniam, Mahmut T. Kandemir, and Chita R. Das. 2018. FLOSS: FLOw Sensitive Scheduling on Mobile Platforms. In *Proceedings of the 55th Annual Design Automation Conference (DAC ’18)*. ACM, New York, NY, USA, Article 173, 6 pages. https://doi.org/10.1145/3195970.3196052

[84] Haibo Zhang, Prasanna Venkatesh Rengasamy, Shulin Zhao, Nachiappan Chidambaram Nachiappan, Anand Sivasubramaniam, Mahmut T. Kandemir, Ravi Iyer, and Chita R. Das. 2017. Race-to-sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds *(MICRO-50 ’17)*. 517–531.

[85] Youtao Zhang, Jun Yang, and Rajiv Gupta. 2000. Frequent Value Locality and Value-centric Data Cache Design. *SIGPLAN Not.* (2000), 150–159.

[86] H. Zhao, L. Xue, P. Chi, and J. Zhao. 2017. Approximate Image Storage with Multi-level Cell STT-MRAM Main Memory. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 268–275.

[87] Shulin Zhao, Prasanna Venkatesh Rengasamy, Haibo Zhang, Sandeepa Bhuyan, Nachiappan Chidambaram Nachiappan, Anand Sivasubramaniam, Mahmut Kandemir, and Chita R. Das. 2019. Understanding Energy Efficiency in IoT App Executions. In *In Proceedings of The 39th IEEE International Conference on Distributed Computing and Systems (ICDCS)*.

[88] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-performance and Energy-efficient Mobile Web Browsing on Big/Little Systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA ’13)*. 13–24.

[89] Yuhao Zhu and Vijay Janapa Reddi. 2017. Optimizing General-Purpose CPUs for Energy-Efficient Mobile Web Computing. *ACM Trans. Comput. Syst.* (2017), 1:1–1:31.

[90] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018. Euphrates: Algorithm-SoC Co-design for Low-power Mobile Continuous Vision. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA ’18)*. 547–560.