A Patch Memory System For Image Processing and Computer Vision

Jason Clemons, Chih-Chi Cheng*, Iuri Frosio, Daniel Johnson, and Stephen W. Keckler
NVIDIA, *Qualcomm
Santa Clara, CA

***Abstract*—From self-driving cars to high dynamic range (HDR) imaging, the demand for image-based applications is growing quickly. In mobile systems, these applications place particular strain on performance and energy efficiency. As traditional memory systems are optimized for 1D memory access, they are unable to efficiently exploit the multi-dimensional locality characteristics of image-based applications which often operate on sub-regions of 2D and 3D image data. We have developed a new Patch Memory System (PMEM) tailored to application domains that process 2D and 3D data streams. PMEM supports efficient multidimensional addressing, automatic handling of image boundaries, and efficient caching and prefetching of image data. In addition to an optimized cache, PMEM includes hardware for offloading structured address calculations from processing units. We improve average energy-delay by 26% compared to EVA, a memory system for computer vision applications. Compared to a traditional cache, our results show that PMEM can reduce processor energy by 34% for a selection of CV and IP applications, leading to system performance improvement of up to 32% and energy-delay product improvement of 48–86% on the applications in this study.**

I. Introduction

Image processing (IP) and computer vision (CV) have become increasingly important application domains, driving computational demand in mobile and embedded systems for products such as smart phones, automobiles [1], cameras [2], and augmented reality systems [3]. These application domains require the ability to process large volumes of data, often in real-time, taxing both computation and memory system throughput. While IP/CV applications demand increasingly higher computational capability, they are often employed in embedded or mobile devices where energy efficiency is of key importance, such as the automotive market [4].

IP and CV algorithms continue to evolve at a rapid pace, exhibiting a diverse set of computational characteristics. While the demand for ever-more performance is constant, the changing algorithmic landscape requires IP/CV systems to provide sufficient flexibility and programmability to adapt. As devices and systems add both more cameras and higher resolution imaging sensors, the demand for flexible, high-performance IP/CV processing will continue to increase. For example, emerging automotive platforms can have six or more high-resolution cameras with various fields of view,

978-1-5090-3508-3/16/$31.00

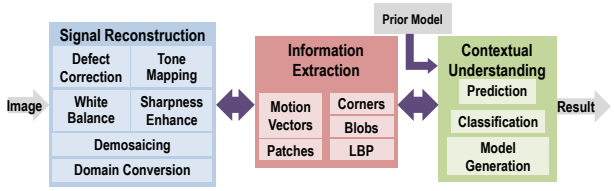


Fig. 1: Schematic of the IP/CV pipeline.

operating under varying weather conditions at both daytime and nighttime. Different situations require different processing algorithms, all within the same system — for instance, pedestrian detection in visible light during daytime [5] or in infrared at night [6]. The diversity of algorithms combined with continuous development of new sensors and requirements means programmability remains fundamental to modern IP/CV processors.

Programmers in this domain require an efficient, high-performance, and flexible programming target to meet these needs. Researchers and engineers have developed architectures targeted at these domains, including application specific processing engines [7], [8], hardwired accelerators [9], and heterogeneous programmable processors [10], [11], [12], [13], [14]. While these designs have generally focused on improving the efficiency of computation, tremendous opportunity lies in customizing the *memory system* that feeds the computation units. For example, in our set of IP/CV workloads, we find that an average of 33% of dynamic instructions are attributable to address computation and indexing — primarily due to accessing regularly structured, multi-dimensional image data in local subregions known as *patches*. A traditional memory system requires multi-instruction sequences to compute address mappings from the (x,y) coordinates in image space to the linear space of traditional memory systems; to handle extrapolation at the image border; and to shift the subregion of interest, or patch, within the image data. This overhead can become a substantial tax on both performance and energy.

Contemporary architectures use standard caches or scratchpads (some with DMA engines) to manage the locality and movement of image data [14], [15]. Both caches and scratchpads are designed on top of linear address space memory systems; neither can exploit optimization opportunities presented by multi-dimensional data structures

TABLE I: Comparison of memory system attributes.

	Cache	Scratch	Prefetch	Texture	PMEM
Easy to use	✓		✓	✓	✓
Predictable		✓			✓
Low latency	✓	✓	✓		✓
2D addressing				✓	✓
Reduced addressing overhead				✓	✓

and access patterns common to IP and CV applications. Caches are unable to effectively exploit 2D image locality. While scratchpads can, they require manual management. Prefetchers can assist caches in capturing 2D locality, but are speculative and unpredictable (undesirable for real-time systems) and may require similar effort as a scratchpad to effectively exploit [16]. Both caches and scratchpads suffer from addressing overheads when indexing multi-dimensional data. GPUs provide Texture Units with caches to exploit 2D image locality and implement graphics-oriented features for filtering, interpolation, and boundary handling. While texture units support 2D addressing using image coordinates, 2D texture lookups are made relative to the image origin, not relative to a local patch of interest. While texture caches take advantage of common block-linear (and similar) memory access patterns to optimize memory layout, they do not implement prefetching; graphics-oriented texture units are optimized for throughput and minimizing memory bandwidth consumed, not low latency.

Table I compares key aspects of these memory system architectures. An efficient, flexible IP/CV memory subsystem should provide the 2D addressing capabilities of texture units; the predictable data availability of scratchpads; the ease of use of caches; and the memory latency reduction of sophisticated prefetching systems.

We developed the *Patch Memory System (PMEM)* to capture the benefits of these memory system features while specifically optimizing for IP/CV applications. PMEM is designed to exploit the attributes of CV and IP algorithms to improve the efficiency of accessing and manipulating 2D and 3D data. Our proposed memory system architecture can be attached to various processing architectures, including CPUs, DSPs, and GPUs. PMEM provides a memory interface abstraction that is both high-performance and an efficient target (in terms of productivity) for algorithm developers. Towards these goals, our Patch Memory System provides the following features:

- **Accelerated 2D and 3D addressing.** PMEM offloads complex address calculations from the programmable processor to dedicated address generation units.
- **Multidimensional data primitives.** Hardware support for manipulating hierarchical data (images, patches, and tensors).
- **Patch-aware caching.** PMEM caches multidimensional image data based on image-space locality for patches being processed.
- **Efficient patch movement operations.** Hardware support for sliding windows and block data transfer.
- **Programmable border handling ("halos").** PMEM provides automatic handling of image and patch borders, eliminating the need for complex conditional code.

Our results show that PMEM can eliminate up to 28% of dynamically executed instructions and 34% of processor energy relative to a system with a conventional cache. By better exploiting structured data locality, PMEM improves application performance by as much as 32% and energy-delay by 48–86% on the applications in this work. We show that PMEM provides better performance than both the computer vision-specific memory system provided by EVA [16] and 2D addressing alone, with energy-delay improvements of 26% and 17% respectively over these approaches. PMEM also provides a memory interface that simplifies programming and can be used to directly support domain-specific image processing languages such as Halide [17]

II. Background

A. Image Processing/CV Pipeline

Image processing (IP) and computer vision (CV) algorithms are used in a large number of applications that exhibit similar structure in their processing. Figure 1 shows a decomposition of a typical IP/CV pipeline into its logical phases. While the primary data flow is down the pipeline from phase to phase, feedback from one phase to a prior one is also possible.

The first phase, *signal reconstruction*, converts the noisy, analog image sensor data into the digital domain. This process involves both performing transformations on the data, such as demosaicing [18] (which converts Bayer pattern data into RGB), and dealing with noise and errors introduced in data capture using techniques such as defective pixel identification [19] and noise reduction [20], [21].

In the second phase, *information extraction*, algorithms process the image data to identify features or image characteristics. These characteristics can include edges, corners, motion vectors, or image gradients. For example, the FAST corner detector locates primitive geometric features [22] and the BRIEF descriptor summarizes the features using signatures generated by concatenating pixel comparison results in a region around the feature location [23].

In the last phase, *contextual understanding*, the results from information extraction are combined with a model or other information to develop a hypothesis on the content of the image or image stream. This phase typically uses CV procedures or machine learning techniques, such as support vector machines [24] or decision trees [25], to determine characteristics such as the likelihood of the presence of an object or the pose of the camera in the scene.

The signal reconstruction and information extraction phases both operate on multidimensional inputs such as images or video. While signal reconstruction will often result

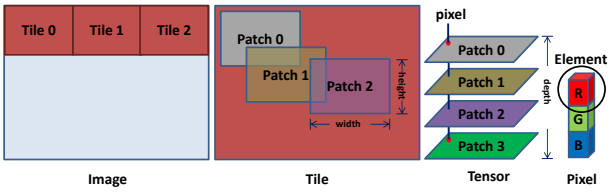


Fig. 2: Image, tile, patch, tensor, pixel and element primitives. In this example, the pixel represents the R, G, and B channels of a color image.

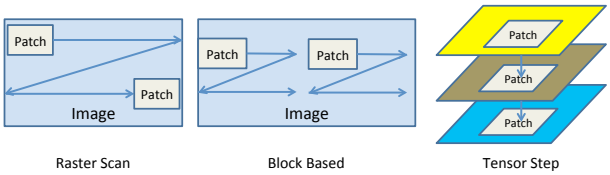


Fig. 3: Common patch access patterns.

in multidimensional output data such as denoised images, information extraction will typically result in a list of locations and a 1D vector signature for each location. These results are commonly fed to a contextual understanding phase for computation. This phase typically outputs sparse information about the original scene, such as the presence of a face or 3D structure of a scene. In this flow, the first two phases operate on dense 2D inputs of multiple megabytes per image (for example, 5.9MB for a 1920×1080 image), while the contextual understanding phase may operate on smaller and sparser data in the range of hundreds of kilobytes.

B. Data Primitives

The common input types for the first and second phases lead to common access patterns and primitives for traversing image data. Figure 2 shows a hierarchical decomposition of the data primitives in IP and CV.

Element: An element is a single scalar value that can be used to represent things like color intensity, luminance, or chrominance.

Pixel: A pixel is a vector of elements representing the characteristics of a 2D or 3D position in an image. Pixels can use a range of formats, including RGB, YUV, or Bayer encoding. For example, an RGB pixel has one element (or channel) each for red, green, and blue color intensity.

Image: An image is a 2D (or 3D) data structure composed of individual pixels that can be addressed using cartesian coordinates. An image is typically treated as having an origin at $(0, 0)$. The image size depends upon its resolution.

Patch: A patch is a 2D (or 3D) subset of an image that is operated on by an IP/CV computation kernel. For example, a 3×3 convolution kernel requires a 2D patch of 9 pixels from the image to compute a single output pixel. Patches are represented with origin coordinates relative to the image origin and patch width and height. Algorithms will often process many overlapping patches which provides the memory system the opportunity to exploit inter-patch locality.

Tensor: A tensor is a set of patches arranged in a stack to form a 3D structure. The tensor can be constructed from patches originating in different images in a video stream or patches coming from different parts of the same image. A 3D pixel address within a tensor is formed by combining the pixel offset location in a patch and the position of the patch in the patch stack. One example application that uses this structure is block matching for denoising [20], [21]. A tensor can be used to aggregate the information from multiple patches to produce a single output patch (or pixel).

Tile: An image can be decomposed into non-overlapping regions that are 2D subsets of the image. While a patch is exposed to IP/CV kernels as a primitive data structure, a tile can be used to represent the unit of data to transfer to and from DRAM. In a conventional processor architecture, a tile could be a single cache line; other architectures can employ larger tiles. In architectures that support software caching in scratchpad memories, tiles may be exposed to the programmer. Other architectures may maintain tiles as a microarchitectural construct. Regardless of tile size, patches may span multiple tiles, thus separating the logical view of memory (patches) from the physical view of memory (tiles).

While the image processing abstractions operate on multidimensional data, image data is stored in DRAM in a linear address space in row-major or column-major order. Consequently, when an algorithm references a pixel, patch, image, tensor, or tile, it must convert multidimensional coordinates into a linear address.

$$i(x, y, c) = i_0 + y \cdot sz_{row} + x \cdot sz_{pixel} + c \cdot sz_{chnl}. \quad (1)$$

Equation 1 shows the computation required to convert an element reference in an image into a linear address. The sz_{row}, sz_{pixel}, and sz_{chnl} are used to translate the given x, y, and channel c coordinates into an offset that is added to the image base address, i_0. For a tensor, the addressing is similar except either the data pointer or the x, y coordinates are different for each patch in the stack. Computing these linear addresses results in significant instruction overhead in conventional processor architectures.

C. Data Access Patterns

Many IP/CV algorithms can operate on different patches in parallel. For example, a 3×3 patch of an image may be required to produce each output pixel during convolution, but all of the output pixels can be computed in parallel. More generally, data access patterns commonly use patches or tensors and move the origin of this region through the input image in algorithm-specific patterns to produce an output. Figure 3 shows a set of common patch access patterns that provide opportunities for hardware acceleration.

Raster: The raster scan order pattern processes patches in a row-major or column-major linear order in the image.

The row-major pattern begins at the top left corner of the image and proceeds by moving through the image to the right until the end of the row. Once the end of the row is reached, the pattern moves to the left most edge of the next row. Patches can be stepped in units of a single pixel or multiple pixels. A variation zig-zags back and forth from row to row to exploit some locality among the adjacent rows. However, for our purposes, these variations are essentially the same. Convolution algorithms commonly access data in a raster pattern.

Blocked: In the blocked pattern, the image is subdivided into 2D regions called blocks. The blocks are processed in a row-major or column-major raster scan order depending on the application. Within each block, the pixel data are typically processed in a raster scan order. This pattern optimizes the reuse of data within a hardware or software controlled cache. The FAST corner detector accesses image data this fashion [22].

Tensor step: The tensor step pattern accesses tensors or a stack of patches. Starting at the patch on the top or bottom of the stack, the same x-y location is accessed on each patch. Stacks of patches can be accessed in a raster or blocked fashion within the image volume. This approach is used to aggregate patches for non-local algorithms such as BM3D denoising [20].

Data dependent: Data dependent patterns do not follow a predetermined path through the image data, but instead depend on the image contents. For example, the base locations of feature signatures are data dependent because the locations of feature points depends on the image content. Algorithms such as BRIEF [23] and BM3D [20] have data dependent access patterns. These algorithms still rely upon patches of image data that can exploit patch-level locality, but the order of patch processing is not predetermined.

D. Image Borders

A final common operation in IP/CV algorithms is computing pixels at image boundaries. IP algorithms that operate on patches of data must account for missing pixels at boundaries to compute the correct results. Missing pixels can be clamped to zero, replicated from adjacent pixels, or mirrored at the image border. Different algorithms implement different strategies for handling border pixels, so a single method cannot be hardwired. While the number of border pixels is small (perimeter) relative to the total number of pixels in the image (area), programmers may employ a significant amount of time and number of lines of code on the special border cases. A means of automatically filling in the missing pixels provides an optimization opportunity.

III. Patch Memory System

To address the needs of accessing 2D and 3D data, we designed the Patch Memory System around a multidimensional addressing scheme. For example, to access the pixel at coordinate $(1, 5)$ the user specifies an image identifier and the coordinates $(1, 5)$. PMEM uses the identifier and coordinates to check if the data is stored in the PMEM caching system. If the data is not in local storage, PMEM translates these specifiers into a linear memory address to retrieve the data from the system memory. To enhance locality, PMEM provides a caching architecture targeted to the structured data from the IP/CV application domain. The addressing and locality functionality leads to the features below.

Image to linear addressing. While addressing in the patch memory system takes place in a multidimensional image domain, traditional memory systems use a linear address space. Transactions with DRAM must be translated between the image domain and a linear memory space. This transform is handled by the PMEM memory interface using image metadata and facilitates transfers in and out of the PMEM.

Border extrapolation. The patch memory system handles border extrapolation automatically, including clamp to value, mirroring, and extending the edge pixel values as defined in [26]. This hardware support helps eliminate border handling code, streamlining CV and IP kernels.

2D caching. To exploit spatial locality, PMEM caches 2D image regions, with the unit of transfer to/from DRAM being a 2D tile instead of a 1D cache line. A patch is mapped to a set of tiles which are loaded into the PMEM. The caching of tiles is a form of prefetching that leverages the 2D locality of the application.

A. Image and Patch State

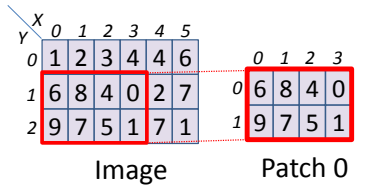
The patch memory system includes the six key primitives described in Section II-B. The image, patch, and tensor are exposed to the programmer via an image table and patch table which store the metadata required to describe the 2D/3D data structures.

Image table. The programmer defines images that will be accessed by PMEM, described by a set of attributes used when accessing the image data in DRAM space. Image attributes include height, width, number of channels, distance between rows in memory (row_step), distance between pixels of the same row in memory (col_step), and the distance between the channels in a given pixel ($channel\_step$). The image table also encodes the mode for border extrapolation and the clamp value when the clamp mode is used. These attributes are stored for each image in a table that can be modified by the programmer to support multiple image sizes and formats. In general, we expect most applications to operate simultaneously on a small number of images. BRIEF and Convolution use a single image, while algorithms such as HDR can operate on up to 16 images [27]. We expect that 32 entries in the image table will be sufficient for most IP/CV algorithms.

Patch table. The programmer also defines the patches to be accessed within the image. Patch attributes include an index into the image table for the image that encompasses

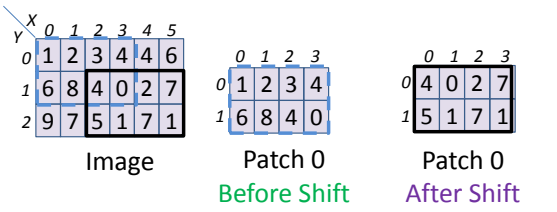
TABLE II: Patch Memory System operations.

Operation	Description
`patch_ld patch_id`	Load patch/tensor from memory into cache using patch table state
`patch_st patch_id`	Store patch/tensor from cache into memory using patch table state
`patch_sft patch_id, delta_x, delta_y`	Shift patch/tensor in image space using (x,y) stride
`pxlp_ld patch_id, R_dst, x, y`	Load pixel from patch at (x,y) from patch origin
`pxli_ld image_id, R_dst, x, y`	Load pixel from image at (x,y) from image origin
`pxlp_st patch_id, R_src, x, y`	Store pixel to patch at (x,y) from patch origin
`pxli_st image_id, R_src, x, y`	Store pixel to image at (x,y) from image origin
`vecp_ld patch_id, R_dst, x, y`	Load vector from patch at (x,y) from patch origin
`veci_ld image_id, R_dst, x, y`	Load vector from image at (x,y) from image origin
`vecp_st patch_id, R_src, x, y`	Store vector to patch at (x,y) from patch origin
`veci_st image_id, R_src, x, y`	Store vector to image at (x,y) from image origin



pxli_ld (0, 0, 0) Returns 1
pxlp_ld (0, 0, 0) Returns 6
veci_ld (0, 0, 1) Returns {6,8,4,0}
vecp_ld (0, 0, 1) Returns {9,7,5,1}

(a) Addressing Example.



patch_sft (0,2,1)

Patch Table Entry	ID	Origin	Height	Width	...
Before	0	**(0,0)**	2	4	...
After	0	***(2,1)***	2	4	...

(b) Patch Shift Example.

Fig. 4: Examples of PMEM addressing.

the patch, the current patch origin within the image, patch height, and patch width. To support tensor data types up to a depth of four, the patch table entry replicates the origin and parent image index fields four times. To optimize border handling, the patch table entry can also hold the image height, image width, and extrapolation information. Like the image table, the patch table stores patch attributes and is accessible by the programmer. To provide at least one patch per image, the patch table has at least 32 entries.

B. PMEM ISA

PMEM extends the ISA of the accompanying compute engine with instructions specific to images and patches. These instructions trigger the multidimensional address calculation and caching within the PMEM hardware. Table II lists the operations provided by the patch memory system ISA.

Patch loads and stores. The patch load instruction specifies the region of the image to prefetch into the patch memory system. This prefetch operation allows the system to ensure that later pixel and vector loads and stores from the patch used by the compute kernel can be serviced by the PMEM cache. The hardware enforces an interlock between `patch ld/st` operations and earlier/later memory operations to ensure proper memory access semantics. Prior to executing a `patch ld/st` instruction, the programmer will have stored the patch metadata in the patch table. Thus `patch ld/st` instructions need only to specify a patch identifier. Patch `st` instructions evict patch data from the PMEM cache and if dirty write it back to the next level of the memory hierarchy. Tensor loads and stores use the `patch ld/st` instructions but leverage the additional tensor metadata in the patch table entry.

Patch shifts. The patch shift instruction moves the region of interest (patch) within the image space, as specified by the `patch id`, horizontal shift step, and vertical shift step. This operation updates the patch table entry and can initiate memory transfers to fill in data to make the new patch complete. This instruction has an implicit memory barrier for accesses to the patch.

Pixel loads and stores. PMEM supports accessing individual elements and loading them into registers or storing them into an image or patch. The pixel load instruction comes in two flavors, one each for accessing data in patches and images. These instructions specify a patch or image identifier, a register target, and (x,y) coordinates. The PMEM hardware computes the proper address using the patch or image origin and the coordinates. For example, an image pixel load with coordinates $(1, 1)$ will return the image pixel $(1, 1)$. For a patch with origin $(2, 3)$, a patch pixel load with coordinates $(1, 1)$ will deliver the data from image pixel $(3, 4)$, accessing the PMEM cache when possible. Misses in the PMEM cache will cause the pixel to be fetched directly from the next level of the memory hierarchy. Stores have a similar format.

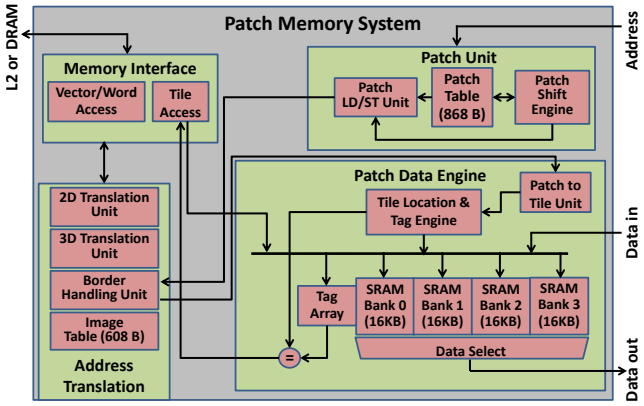


Fig. 5: The Patch Memory System (PMEM).

Vector loads and stores. PMEM implements vector load and store operations that are very similar to pixel loads and stores. In our design, the vector registers are four adjacent 32-bit registers.

Figure 4a shows example pixel and vector load instructions to illustrate the mapping of coordinates in image and patch spaces. The patch in Figure 4a has an origin at $(0, 1)$ causing a pixel patch load from coordinate $(0, 0)$ to return a value of 6. The patch origin coordinates are added to patch coordinates to produce the image coordinates to be accessed. Vector loads fetch data starting at the given coordinate. Figure 4b shows the effect of a patch shift. In this example, the patch starts with an origin of $(0, 0)$ represented by pixels in the blue bounding box. The patch shift instruction moves the origin by 2 pixels in the x-dimension and 1 pixel in the y-dimension, resulting in a new patch origin of $(2, 1)$ represented by pixels in the black bounding box.

IV. PMEM Microarchitecture

Figure 5 shows the core components of the PMEM architecture, including the Patch Unit, the Patch Data Engine, the Address Translation Unit, and the Memory Interface. Together, these units implement the features described in Section III.

A. Patch Unit

The Patch Unit maintains the patch table and directs the execution of PMEM data requests. The Patch Unit consists of three sub-units: the patch ld/st unit, the patch shift engine, and the patch table.

Patch LD/ST. This unit coordinates memory requests sent to PMEM. If the request accesses a patch, this unit will query the patch table to obtain the required metadata and forward the request to the Border Handing Unit. This unit can also bypass the Patch Data Engine and send a request directly to the memory interface. The patch ld/st unit interfaces with the patch shift engine when a shift instruction is issued.

Patch shift engine. The patch shift engine handles traversal of patches and tensors through the image space by manipulating their origins. A `patch_sft` instruction dictates the distance in image space that the patch is moved. The unit computes the new patch location and sends a command to the Patch Data Engine to initiate fetching of data to fill in the shifted patch in the cache.

TABLE III: Patch table entries.

Entry	*Use*	*Bits*
Image Id×4	Id for image source	32
Width	Patch Width	6
Height	Patch Height	6
Channels	Number of Channels	3
Row Step	Dist to next row in pixels	12
Column Step	Dist to next column in bytes	5
Channel Step	Dist to next channel in bytes	3
Parent Height	Parent Height	15
Parent Width	Parent Width	15
Origin In Image (x,y)×4	Origin in image coordinates	120
	Total	217

Patch table. Table III shows the metadata associated with each patch entry. Using the information in the table, address calculations for intra-patch accesses need only a few bits instead of the full bit-width of a memory address. The table itself is a small RAM array with 217-bit entries; with a total of 32 entries, the patch table is 868 bytes. The origin and the parent image entries in the table have four entries each to support the tensor primitive.

B. Patch Data Engine

The Patch Data Engine manages the caching of 2D data. This unit has two major components, the patch-to-tile conversion unit and the tile cache. Together they provide the physical caching support for the image data. For multicore systems the Patch Data Engine would be shared but each core would have a private Patch Unit.

Patch-to-tile conversion. This hardware unit converts the coordinates in a patch to coordinates in a tile. The hardware first translates the patch space coordinates into image space coordinates by adding the patch (x,y) coordinates to the (x,y) origin of the patch. These image coordinates are converted to the tile coordinate space, called tile indices, based on the fixed dimensions of the tiles. We limit the tile size to a power of 2 to allow the hardware to use the low order bits from the image (x,y) coordinates as the (x,y) coordinates within the tile; the higher order bits are used as a tag to identify the tile. We also limit the maximum patch size to be twice the tile size to ensure that any patch covers at most nine tiles. As a result, we can determine which tiles are needed to supply data to the patch by computing the tile identifiers for each of the corners of the patch. The unit can then request up to the nine unique tiles necessary to cover the patch.

Cache unit. The cache unit stores data at the granularity of the fixed-size tiles. Figure 6 shows how the tag and index are formed from the coordinates passed to the Patch Data Engine. The patch address is converted to image coordinates and an image_id using the state in the patch table. The tile

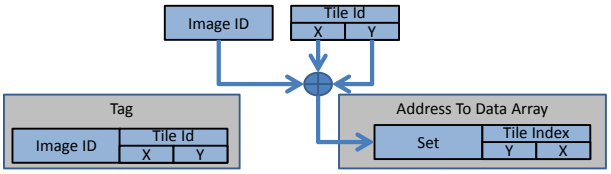


Fig. 6: Tile cache tag and address generation.

x-index and tile y-index are formed using the lower 4 bits of the image (x,y) coordinates (16×16 tiles). The tile x-id and tile y-id are formed using the next 10 bits of the (x,y) coordinates (max 16K×16K images). The tag is formed by concatenating the image_id, tile x-id, and tile y-id. The set is chosen by xor-ing the tile x-id, tile y-id and the image_id together. The result is concatenated with the tile y-index and tile x-index to produce the address for the cache data array. The use of the image_id allows for patches with the same origin in different images to map to different sets. Storing data based on the granularity of tiles reduces the tag array size as compared to a traditional cache. Furthermore, the cache unit uses bank swizzling similar to [28], [29] so that it can provide both row and column vector access to 2D data. Ordinary coherence mechanisms can be used to provide cache coherence between the Patch Data Engine and other caches in a multi-cache or multiprocessor system.

C. Address Translation

The Address Translation Unit translates between the multidimensional address space of PMEM and the linear address space of DRAM. This unit has multiple subunits, including an image table, a 2D/3D translation unit, and a border handling unit.

Image table. The image table is a small RAM array with each entry encoding the fields listed in Table IV. Each entry is 152 bits and corresponds to a single image. This table is accessed to obtain the parameters for Equation 1 needed to translate from the image space to linear DRAM memory. This table is shared by all the compute elements attached to the patch memory system. With a total of 32 entries, the image table is 608 bytes.

2D address translation. The 2D translation unit uses the information in the image table to perform the translation to linear space. This translation can be performed with one shift, two multiplies, and three small integer adds, as defined by Equation 1. To simplify the hardware, we assume the channel step is a power of two since data types are power of two bytes in length.

3D address translation. The 3D translation is for tensors and is similar to the 2D component except that the base image origin is chosen based on the tensor's third (z) component of the address. This unit generates up to N addresses to be fetched to support gathering the data for a given tensor access. For this work, we assume N equals four to match the maximum tensor depth that we consider.

TABLE IV: Image table for tracking image data.

Entry	***Use***	***Bits***
Width	Image Width	14
Height	Image Height	14
Channels	Number of Channels	3
Row Step	Dist to next row in pixels	14
Col Step	Dist to next column in bytes	5
Channel Step	Dist to next channel in bytes	3
Image Origin	Data ptr to image origin	64
Border Method	Border Extrapolation Method	3
Border Clamp Value	Border Clamp Value	32
	Total	152

Border handling. Border handling is performed before sending a request to the data fetch engine using information from the patch table. The patch table contains the image dimensions which the border handling unit uses to determine whether the access is within the image boundaries. The hardware uses the image coordinates from the patch unit and performs comparisons to ensure that the (x,y) coordinates are between zero and the image dimensions. Depending on the border handling method, the unit will either modify the requested address before forwarding to the data fetch engine or return the data itself. For example, if the border method is *clamp* then the request is handled by retrieving the clamp value from the image table. If the method is *replicate border* then the address is modified to the closest edge pixel and sent through the normal data fetch path.

D. Memory Interface

The memory interface communicates with the memory hierarchy outside of the PMEM. Specifically, it fetches required data at the granularity of the external memory hierarchy (e.g. cache line) and uses the memory translation unit to compute the proper addresses. PMEM could be integrated into a system as an alternative L1 with an L2 supplying data or as the only cache accessing data.

Vector/word unit. The vector/word unit fetches vector and scalar word data from the next level of the memory system. This unit coalesces across and within requests to generate the fewest external references needed. In this paper, we assume a vector length of four, but supporting different vector lengths requires few changes to the architecture. Non-unit stride vector accesses may require multiple references to gather the data from the next level of the memory hierarchy.

Tile access unit. The access unit fetches tiles from the next level of the memory hierarchy. This unit accepts the address of the tile origin and the image dimensions from the Address Translation Unit. It then generates the external memory hierarchy requests to gather the data to fill the tile. Each tile request is translated into multiple fetches from the next level of the memory hierarchy based on the data access granularity. Since the tile size is fixed, the hardware to perform the gathers is inexpensive, requiring a simple set of adders to increment the starting address based on the cache

```
1 Image input = load("inputImage.png");
2 Image output (inputImage.getSize());
3 Image coeffs = getCoeffientsImage(5,5);
4 for (int oy = 0; oy < height; oy++){
5  for (int ox = 0; ox < width; ox++){
6    accum = 0;
7    patch = input.data+(oy-2)*input.rowStep +ox-2;
8    for (int y = 0; y < coeff_height; y++){
9      coeffRow = coeffs.data + y*coeffs.rowStep;
10      imgRow = patch + y* input.rowStep;
11      for (int x = 0; x < coeff_width; x++){
12        if(oy < 2|| oy >= input.height -2
           || ox < 2 || ox >= input.width-2){
13          imgX = ox-2;
14          imgY = oy-2;
15          pix = ((imgX >=0 && imgY >= 0) &&
                 (imgX+4 <= input.width-1) &&
                 (imgY+4 <= input.height-1)) ?
                  imgRowPtr[x]:0);
16        }else{
17           pix = *(imgRow+x);
18          }
19         coeff = *(coeffRow+x);
20      }
21       accum += pix * coeff
22    }
23  }
24  *(output.data + oy*output.rowStep + ox)= accum;
25 }
```

(a) Conventional Version.

```
1 Image inputImage = load("inputImage.png");
2 Image outputImage(inputImage.getSize());
3 Image coefficients = getCoeffsImage(5,5);
4 Patch inputPatch (inputImage,-2,-2, 5,5);
5 Patch coeffPatch (coefficients,0,0,5,5);
6 patchld inputPatch
7 patchld coeffPatch
8 for (int oy = 0; oy < height; oy++){
9   for (int ox = 0; ox < width; ox++){
10    accum = 0;
11     for (int y = 0; y < coeff_height; y++){
12       for (int x = 0; x < coeff_width; x++){
13         pxlp_ld inputPatch, pix, x,y;
14         pxlp_ld coeffsPatch, coeff,x,y;
15         accum += pix * coeff
16         }
17       }
18       pxli_st outputImage, accum ,ox,oy;
19       patch_sft inputPatch, 1,0;
20     }
21     patch_sft inputPatch,-width,1;
22 }
```

(b) PMEM Version.

Fig. 7: The pseudo-code for a 5×5 convolution.

line size and image row step. This unit sends the returned data to the tile cache in the Patch Data Engine. Fetching at the tile granularity promotes a regular access pattern that can be exploited by a memory controller for good DRAM efficiency. Specifically, PMEM can employ tile sizes and tile fetch orderings to maximize hits per activate when accessing DRAM.

V. Example: 5×5 Convolution

To illustrate how the Patch Memory System operates, this section first shows how PMEM simplifies code implementing algorithmic kernels. It then describes how that code exercises the elements of the PMEM hardware. We use a 5×5 image convolution to illustrate the key aspects of the system.

A. Code Comparison

Figure 7a shows the pseudo-code for performing a 5×5 convolution on a traditional cache-based memory system, with code for address calculations and border checking. Figure 7b shows the corresponding code for a system with PMEM. The PMEM code is more compact and better matches the addressing in the image processing space, while the conventional memory system code requires address pointer calculations and special handling of the computations at the border (Figure 7a, lines 9 and 15). The pointer calculations are replaced by the semantics of patches in PMEM which allow for the code to specify an image (`image_id`) and (x,y) offsets to access pixels and the `patch_sft` instructions (lines 19 and 21) that adjusts the position of the patch. Finally, the PMEM code does not require the complex conditionals for border checking, as the hardware performs the necessary extrapolations automatically.

B. Patch Memory System Execution

The different lines of pseudo-code in Figure 7b correspond to activations of the hardware units in Figure 5. The code begins in lines 1–5 by creating the images and patches, which allocates and fills in entries in the image and patch tables; this operation can be performed when the images and patches are instantiated, hiding the complexity from the application programmer. The code then uses PMEM to load the initial patches at lines 6–7. The code then steps through the output image computing each pixel as it progresses. Each pixel and coefficient is fetched using 2D addressing which engages PMEM (lines 13–14). The patch ld/st unit handles the memory requests by accessing the patch table and sending the required data to the border handling unit. The border handling unit modifies the request based on the border extrapolation mode or returns the pixel value if such an extrapolation mode is set. This unit then forwards the request on to the patch data engine, which either returns the data if it is present in the patch memory system, or requests that the memory interface fetch the data from the memory hierarchy. Line 15 accumulates the product of the pixel value and the weight to compute the convolution. Line 18 stores the resulting summation to the corresponding pixel position in the output image using PMEM. After storing the output pixel, Line 19 shifts the patch over by one pixel position

along the horizontal axis of the image. At the end of the row, Line 21 moves the patch to the beginning of the next row using the patch shift operation.

VI. Experimental Framework

Simulation framework. The PMEM simulator includes a PMEM functional model, performance model, and energy model, coupled to a SIMD vector processor simulator. We avoid developing a compiler by implementing the benchmarks in standard C++ and augmenting the code with API calls for the patch memory primitives. All of the vector instructions and the load/store instructions are included in the functional API. Program control including loops and conditionals are implemented in the base C++ application code and we annotate these statements to account for performance and energy.

The simulator tabulates hardware events to feed to the energy model. To estimate the energy and area of the system, we developed RTL models for components of the PMEM and used an industry product design flow including Synopsys tools and memory generators to synthesize the design in 28 nm technology. We used our tools to provide estimates of energy for various operations and combined these energy numbers with the hardware event counters to estimate system energy.

Architecture configurations. Table V shows the parameters used for evaluating PMEM. We compare PMEM with three other memory systems: a baseline 64KB, 4-way data cache architecture (Cache); the baseline cache augmented with 2D addressing (2D Addr); and a prefetch-based computer vision architecture (EVA) [16]. 2D Addr provides 2D addressing instructions but lacks other PMEM features. For EVA, we implemented loads with the ability to prefetch neighboring tiles based on register values as done in [16]. PMEM is configured as a 64 KB, 4-way set-associative tile cache with tiles of 16×16 pixels, and an LRU replacement policy. Pixel elements are 32 bits each.

While our Patch Memory System can be paired with a variety of computational engines, this evaluation uses an in-order RISC scalar core with a 4-wide SIMD engine, bypassing, interlocks, and a 1GHz clock rate. The use of in-order compute pipeline is similar to modern CV systems [14], [15]. We model a cache bandwidth of 128 bits per cycle and a fixed cache miss latency of 40 cycles. This latency is similar to that of an SOC-wide L2 cache experiencing some contention.

Benchmarks. Table VI lists the benchmarks used to evaluate PMEM. Each of the benchmarks employs block-based computation which subdivides the image space into 2D regions and processes them in an order conducive to memory locality [17]. This approach naturally exploits the features of PMEM. The first three benchmarks are from signal reconstruction: (1) a 5×5 convolution, used in many image processing applications for FIR filters; (2) the block matching portion of BM3D denoising, which matches a target patch to other patches within an image [20]; and (3) Debayer, which converts an image from the Bayer sensor pattern to RGB [18]. The second three are from the information extraction domain: (4) FAST, which detects corners in an image using contrast [30], [22]; (5) BRIEF, which computes a feature signature using comparisons of pixel intensities within a region around a feature point [23]; and (6) Gaussian Mixture Model, which uses up to five Gaussian models at each pixel to perform foreground and background segmentation [31]. We use 32-bit floating-point data throughout the benchmarks.

TABLE V: Memory system configurations.

Element	Parameter	Value
Baseline Cache and 2D Addr	Size	64 KB
	Associativity	4-way
	Line Size	64 Bytes
PMEM and EVA	Size	64 KB
	Associativity	4-way
	Tile Size	16x16

TABLE VI: Benchmarks.

Benchmark	Processing Phase	Input
5x5 Convolution	Signal Reconstruction	1920x1080 Image
Block Match [20]	Signal Reconstruction	768x512 Image
Debayer [18]	Signal Reconstruction	1920x1080 Image
FAST [30], [22]	Information Extraction	1920x1080 Image
BRIEF [23]	Information Extraction	1920x1080 Image
Gaussian Mixture Model [31]	Information Extraction	640x480 Video Frame

VII. Evaluation

A. Instruction Count

Figure 8 shows the fraction of instructions that are eliminated due to hardware support for 2D addressing and border handling. 2D Addr eliminates 7%–28% of total instructions issued. In certain applications such as Conv, Block, and BRIEF, the additional support for border handling in PMEM eliminates another 5–10% of instructions. The EVA system sees a small increase in total instruction count from loading the registers for the prefetch operations. PMEM shows the most benefit for FAST, reducing instruction count by 28% by optimizing the address computation for each pixel in the pattern used for determining the presence of a corner. The smallest improvement is in Debayer, where instruction count was reduced by only 9%. While we were able to reduce instructions attributable to addressing computations by 30% and to eliminate the border handling instructions, these operations only constitute about 30% of total instructions in our applications.

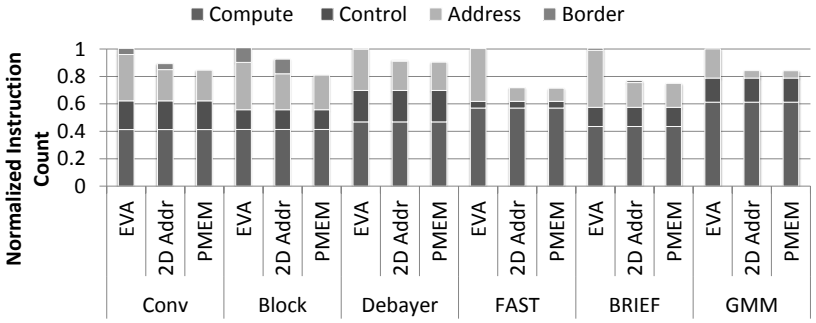


Fig. 8: Instruction count reduction (normalized to conventional cached memory) for EVA, 2D Addressing, and PMEM.

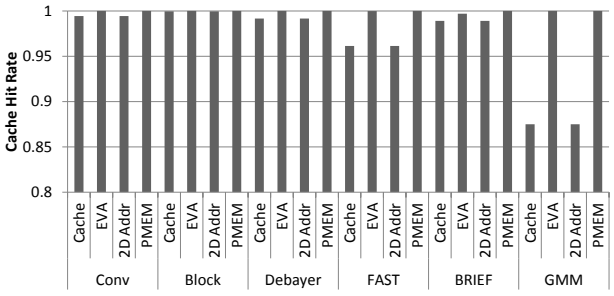


Fig. 9: Cache hit rate for the conventional memory system, EVA, 2D Addressing, and PMEM.

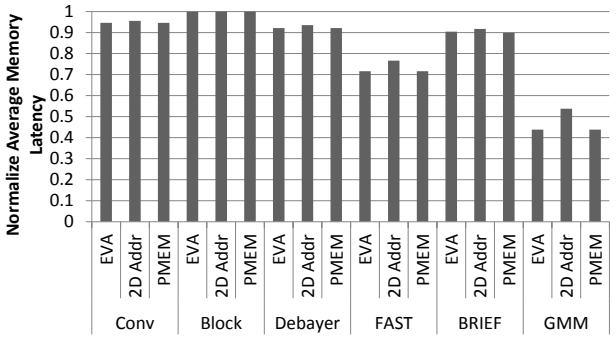


Fig. 10: The average memory latency normalized to traditional memory system.

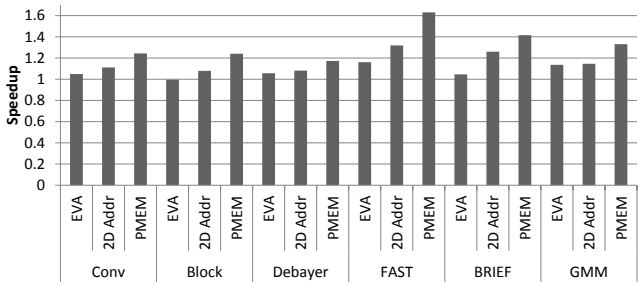


Fig. 11: Speedup for EVA, 2D Addressing, and PMEM normalized to traditional cache architecture.

B. Memory Behavior

Figure 9 shows the cache hit rate of our four memory configurations. PMEM improves cache hit rate by exploiting 2D locality while EVA improves hit rates via 2D prefetching. In both architectures, locality is captured in both dimensions and leads to improved efficiency. While the increase is not large in terms of percentage points, Figure 10 shows that cache hit rate improvements translate directly into reductions in average memory latency. The GMM benchmark shows the largest reduction in the number of cache misses. The benchmark steps through multiple planes of data for the Gaussian models at each pixel. While this access pattern causes conflicts in a traditional cache, PMEM avoids the conflicts by mapping the same tile in two images to different cache sets. With a cache hit rate of 99%, PMEM approaches the ideal memory system where data is fetched only once from memory.

C. Performance

Figure 11 shows the relative performance of EVA, 2D Addr, and PMEM, normalized to the baseline cache architecture. The 2D prefetching of EVA improves performance by an average of 7.3% over the baseline by reducing the number of cache misses. The addressing capabilities of 2D Addr help it improve by an average of 16.5% over the baseline by optimizing the common addressing mode. PMEM exceeds both, improving performance for all of the applications, on average by 34%; no applications see performance degradation. This is due to PMEM reducing cache misses with less overhead than EVA and 2D addressing with border handling. Frame processing latency when using PMEM is thus much better than with a conventional cache, enabling higher frame rates, higher resolutions, and more complex algorithms.

D. Energy

Figure 12 compares the system energy for the four memory architectures, normalized to that of a system using a conventional cache. The bars are subdivided into ALU computation energy (Compute), memory system (cache and DRAM) access energy (Mem), address calculation operations (Addressing), and border handling operations (Border). The category *Other* includes energy from instruction fetch, control, and static energy. EVA, 2D Addr, and PMEM all

TABLE VII: Area costs normalized to conventional cache.

Unit	Normalized Area
Conventional Cache	
Cache data arrays	75.5%
Tag and control logic	24.5%
Total	100%
PMEM	
Patch Unit	6.0%
Address Translation	6.8%
Cache data arrays	75.5%
Tag and control logic	21.2%
Total	109.6%

demonstrate improved energy efficiency over the baseline. The majority of PMEM's efficiency gain comes from reduced address computation overhead for multidimensional data although a significant amount can be attributed to locality as well. PMEM saves energy by replacing multiple addressing instructions with application specific instructions, thus reducing the instruction fetch and decode energy for accessing a given pixel as well as the dynamic energy. The additional performance of EVA, 2D Addr, and PMEM decrease leakage energy by increasing the cache hit rate and reducing the time spent waiting for data. PMEM has a specific advantage over the other architectures for Conv and Block by eliminating the border handling overheads. The computations for addressing are optimized to take advantage of the limited range of image sizes, enabling use of smaller, more efficient, data paths.

E. Area

To examine the area costs of PMEM, we implemented the key components in Verilog RTL and synthesized them using Synopsys tools to a 28nm library. Table VII shows the relative area for key components of the baseline cache (data arrays and tag/control logic) and PMEM, normalized to the total area of the baseline. The tag and control area decreases because the tag array is smaller due to using tiles which are larger than traditional cache lines. This also leads to decreased tag array access energy, thus improving system energy. As expected, cache memory array dominates the area in both the conventional cache and PMEM. The additional hardware to support multidimensional addressing accounts for 11.7% of the area of the PMEM system. Compared to a conventional cache, we consume only 9.6% more area to support all of the PMEM system features.

F. Discussion

The Patch Memory System provides both performance and energy improvements for image processing applications. In particular, the 2D addressing instructions and the automatic boundary handling delivers substantial reduction in cycle count by using specialized hardware to perform these computations. The use of specialized hardware also leads to more efficient computations by using data paths optimized for this application space. PMEM also reduces energy by eliminating the need to fetch instructions for these common operations. While a 2D addressing memory system (2D Addr) shows the benefits of hardware support of 2D addressing, the lack of border handling or 2D cache locality leads to less performance improvement on average than PMEM. Some benchmarks use more of the specialized hardware than others. For example, block matching, BRIEF, and convolution show significant utilization of the border handling hardware, while GMM does not use this hardware at all. Such logic would be amenable to standard clock gating strategies to further improve energy efficiency.

Providing support for addressing computations increases the energy per hit compared to a traditional cache, but this increase is offset by the smaller tag array. Patches work as a form of prefetching based on explicit information from the programmer and eliminate both the speculation that is inherent in hardware prefetchers and the explicit prefetch instructions used in typical software prefetching. These capabilities provide PMEM with the benefits of prefetching without the typical costs. While EVA provides similar capability, the programmer must manipulate specific registers when performing a special load. The patch table allows PMEM to elide prefetch control operations, decreasing overall instruction count. The instruction count for PMEM is further reduced due to 2D addressing support. PMEM also delivers productivity benefits due to the hardware/ISA matching common programming primitives for the target problem domain.

Overall, PMEM improves the energy delay product (EDP) by 14–52% (average improvement of 42%) compared to a system with a conventional cache. In contrast, EVA and the 2D addressing scheme provide an average EDP improvement of 18% and 25%, respectively.

VIII. Related Work

Prefetching. Prefetching techniques can be applied to improve performance for workloads with dense, regular, or otherwise predictable access patterns. Simple prefetchers such as stride prefetchers can effectively capture one-dimensional locality [32]. However, hardware prefetching approaches operate speculatively and outside the programmer's control. Previous work has shown that stride prefetchers have problems stabilizing when accessing data within a 2D window [16]. Algorithms such as BRIEF can have random steps within a patch causing further problems for stride prefetchers. Two-dimensional spatial prefetchers have been proposed, targeted specifically at image data [16], [33]. The EVA memory system includes a cache with a flexible prefetcher capable of handling both 1D and 2D memory access patterns [16]. EVA requires the programmer manually insert specialized load instructions that prefetch tiles. Our results show that the additional 2D addressing, border handling, and patch-based memory operations lead to improved performance of PMEM over EVA. Larabi et al. propose using a 2D tracker to prefetch data for image

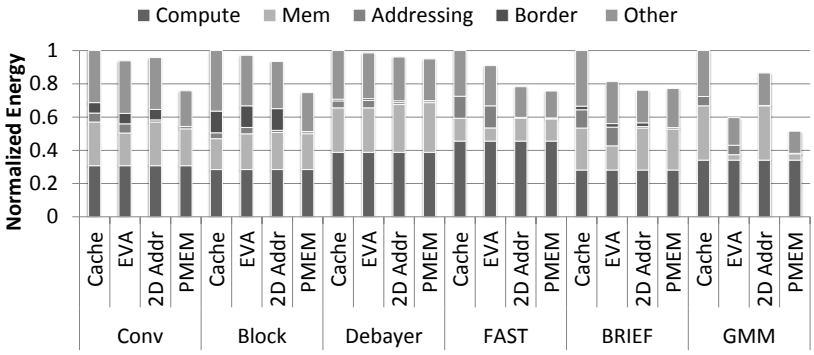


Fig. 12: System energy (normalized to conventional cached memory) for EVA, 2D Addressing, and PMEM.

processing [34]. However, this tracker will not be as accurate as getting the correct region to fetch from the programmer.

Texture Units. Graphics processors have been used to run computer vision applications to achieve high performance [35], [36]. However, there is still room for improved efficiency when operating on image-based workloads. GPUs employ specialized texture memory units to accelerate common image access and interpolation options [37]. Texture units offer special modes for clamping at boundaries, using interpolated floating-point coordinates, and various filtering and sampling modes. They also offer support for access patterns with 2D locality [38]. However, the 2D locality is typically limited to small texel sizes [37]. Texture units do not provide a patch equivalent, so they cannot exploit the benefits of subregion addressing or prefetching that PMEM can. We expect texture units to perform slightly better than the 2D addressing model we studied here.

Scratchpad and Specialized Memory Systems. DSPs commonly deploy scratchpads for various workloads [39]. While scratchpads can improve locality and latency, they require the programmer to manage the local storage using explicit load or DMA operations. Furthermore, scratchpads do not natively provide 2D addressing, caching, or prefetching without elaborate software support. Scatter-gather memory systems have also been used for IP/CV workloads. However, they require expensive generation of linear 1D addresses and do not fully take into account the 2D/3D nature of data structures, leading to lower expected performance than PMEM. Prior work has also examined specialized programmable architectures targeted at common imaging operations. The convolution engine [8] is optimized for the data access patterns and locality observed in convolution stencil computations. The engine employs a set of special 1D and 2D shift registers to manage data transfers. Compared to a scratchpad or architectures with limited configurability, PMEM provides a more traditional cache interface with scratchpad-like performance. PMEM concepts such as 2D addressing and border handling could be applied to scratchpad memory systems. Specialization for memory access patterns can also be incorporated into memory controllers to optimize for DRAM characteristics. PPMC provides such a specialized memory controller for accelerators and is complementary to PMEM [40]; PPMC could be integrated into the memory interface of PMEM.

IX. Conclusions

Patch Memory (PMEM) provides three main features to improve locality and energy efficiency for image processing and computer vision algorithms: 2D and 3D addressing, 2D caching and prefetching of data to exploit locality, and automatic border extrapolation. PMEM provides special instructions based on data types such as patches and images that are common to many image-based applications. These primitives make program code more compact and better matched to the application space. Our results show that PMEM can reduce processor energy by 34%, increase performance by 32% and improve energy-delay product by 48–86% on the applications in this work.

Along with performance and energy improvements, PMEM delivers productivity benefits due to matching the underlying hardware to common programming primitives for the problem domain. Such specialization enables both improvements in efficiency and flexibility for developers in a domain with rapidly evolving applications and algorithms.

Acknowledgment

This research was developed, in part, with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] NVIDIA, "Advanced Driver Assistance Systems (ADAS)," http://www.nvidia.com/object/advanced-driver-assistance-systems.html.

[2] R. Ng, M. Levoy, M. Brédif, G. Duval, M. Horowitz, and P. Hanrahan, "Light Field Photography with a Hand-held Plenoptic Camera," Stanford University, Computer Science Department, Tech. Rep. CSTR 2005-02, 2005.

[3] Microsoft, "Microsoft HoloLens," https://www.microsoft.com/microsoft-hololens/en-us.

[4] F. Stein, "The Challenge of Putting Vision Algorithms into a Car," in *Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, June 2012, pp. 89–94.

[5] P. Dollar, C. Wojek, B. Schiele, and P. Perona, "Pedestrian Detection: An Evaluation of the State of the Art," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 4, pp. 743–761, April 2012.

[6] S. Krotosky and M. Trivedi, "On Color-, Infrared-, and Multimodal-Stereo Approaches to Pedestrian Detection," *IEEE Transactions on Intelligent Transportation Systems*, vol. 8, no. 4, pp. 619–629, December 2007.

[7] R. Rithe, P. Raina, N. Ickes, S. Tenneti, and A. Chandrakasan, "Reconfigurable Processor for Energy-Efficient Computational Photography," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 48, no. 11, pp. 2908–2919, November 2013.

[8] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing," in *International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 24–35.

[9] T. Ohmaru, T. Nakagawa, S. Maeda, Y. Okamoto, M. Kozuma, S. Yoneda, H. Inoue, Y. Kurokawa, T. Ikeda, Y. Ieda, N. Yamade, H. Miyairi, M. Ikeda, and S. Yamazaki, "25.3μW at 60fps 240$\times$160-pixel Vision Sensor for Motion Capturing with In-pixel Non-volatile Analog Memory Using Crystalline Oxide Semiconductor FET," in *International Solid-State Circuits Conference (ISSCC)*, February 2015.

[10] J. Tanabe, S. Toru, Y. Yamada, T. Watanabe, M. Okumura, M. Nishiyama, T. Nomura, K. Oma, N. Sato, M. Banno, H. Hayashi, and T. Miyamori, "A 1.9TOPS and 564GOPS/W Heterogeneous Multicore SoC with Color-based Object Classification Accelerator for Image-recognition Applications," in *International Solid-State Circuits Conference (ISSCC)*, February 2015.

[11] I. Hong, K. Bong, D. Shin, S. Park, K. Lee, Y. Kim, and H.-J. Yoo, "A 2.71nJ/pixel 3D-stacked Gaze-activated Object-recognition System for Low-power Mobile HMD Applications," in *International Solid-State Circuits Conference (ISSCC)*, February 2015.

[12] T. Kurafuji, M. Haraguchi, M. Nakajima, T. Nishijima, T. Tanizaki, H. Yamasaki, T. Sugimura, Y. Imai, M. Ishizaki, T. Kumaki, K. Murata, K. Yoshida, E. Shimomura, H. Noda, Y. Okuno, S. Kamijo, T. Koide, H. Mattausch, and K. Arimoto, "A Scalable Massively Parallel Processor for Real-Time Image Processing," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 46, no. 10, pp. 2363–2373, October 2011.

[13] M. Demler, "Synopsys Embeds Vision Processing," *Microprocessor Report*, April 2015.

[14] T. R. Halfhill, "Ceva Sharpens Computer Vision," *Microprocessor Report*, April 2015.

[15] *TMS320C64x+ DSP Cache User's Guide (Rev. B)*, Texas Instruments. [Online]. Available: http://www.ti.com/litv/pdf/spru862b

[16] J. Clemons, A. Pellegrini, S. Savarese, and T. Austin, "EVA: An Efficient Vision Architecture for Mobile Systems," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, September 2013, pp. 1–10.

[17] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," in *Conference on Programming Language Design and Implementation (PLDI)*, June 2013, pp. 519–530.

[18] H. Malvar, L.-W. He, and R. Cutler, "High-quality Linear Interpolation for Demosaicing of Bayer-patterned Color Images," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2004.

[19] I. Frosio and N. Borghese, "Statistical Based Impulsive Noise Removal in Digital Radiography," *IEEE Transactions on Medical Imaging*, vol. 28, no. 1, pp. 3–16, January 2009.

[20] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian, "Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering," *IEEE Transactions on Image Processing*, vol. 16, no. 8, pp. 2080–2095, August 2007.

[21] G. Yu and G. Sapiro, "DCT Image Denoising: A Simple and Effective Image Denoising Algorithm," *Image Processing On Line*, vol. 1, 2011.

[22] E. Rosten and T. Drummond, "Machine Learning for High-speed Corner Detection," in *Proceedings of the European Conference on Computer Vision (ECCV)*, May 2006, pp. 430–443.

[23] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF: Binary Robust Independent Elementary Features," in *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2010, pp. 778–792.

[24] T. Joachims, "Making Large-Scale SVM Learning Practical," in *Advances in Kernel Methods - Support Vector Learning*, B. Schölkopf, C. Burges, and A. Smola, Eds. Cambridge, MA: MIT Press, 1999, ch. 11, pp. 169–184.

[25] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, October 2001.

[26] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 3rd ed. Upper Saddle River, NJ: Prentice-Hall, Inc., 2006.

[27] D. Jacobs, O. Gallo, and K. Pulli, "Dynamic Image Stacks," in *Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, June 2014.

[28] M. Bojnordi, N. Sedaghati-Mokhtari, O. Fatemi, and M. Hashemi, "An Efficient Self-Transposing Memory Structure for 32-bit Video Processors," in *Asia Pacific Conference on Circuits and Systems (APCCAS)*, December 2006, pp. 1438–1441.

[29] T.-C. Chen, Y.-H. Chen, S.-F. Tsai, S.-Y. Chien, and L.-G. Chen, "Fast Algorithm and Architecture Design of Low-Power Integer Motion Estimation for H.264/AVC," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 5, pp. 568–577, May 2007.

[30] E. Rosten and T. Drummond, "Fusing Points and Lines for High Performance Tracking," in *IEEE International Conference on Computer Vision*, October 2005, pp. 1508–1511.

[31] C. Stauffer and W. Grimson, "Adaptive Background Mixture Models for Real-time Tracking," in *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, June 1999.

[32] J.-L. Baer and T.-F. Chen, "Effective Hardware-Based Data Prefetching for High-Performance Processors," *IEEE Transactions on Computers.*, vol. 44, no. 5, pp. 609–623, May 1995.

[33] N. Zhou, F. Qiao, and H. Yang, "A Hybrid Cache Architecture with 2D-based Prefetching Scheme for Image and Video Processing," in *International Conference on Communications and Signal Processing (ICCSP)*, April 2013, pp. 1092–1096.

[34] Z. Larabi, Y. Mathieu, and S. Mancini, "High Efficiency Reconfigurable Cache for Image Processing," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, July 2009, pp. 226–232.

[35] J.-P. Farrugia and P. Horain, "GPUCV: A Framework for Image Processing Acceleration with Graphics Processors," in *International Conference on Multimedia and Expo (ICME)*, July 2006, pp. 585–588.

[36] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating Computer Vision Algorithms Using OpenCL Framework on the Mobile GPU - A Case Study," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2013, pp. 2629–2633.

[37] H. Igehy, M. Eldridge, and K. Proudfoot, "Prefetching in a Texture Cache Architecture," in *SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, 1998.

[38] N. Wilt, *The CUDA Handbook: A Comprehensive Buide to GPU Programming*, 1st ed. Addison-Wesley Professional, 2013.

[39] S. Gilani, N. S. Kim, and M. Schulte, "Scratchpad Memory Optimizations for Digital Signal Processing Applications," in *Design, Automation Test in Europe (DATE), 2011*, March 2011, pp. 1–6.

[40] T. Hussain, M. Shafiq, M. Pericàs, N. Navarro, and E. Ayguadé, "PPMC: A Programmable Pattern Based Memory Controller," in *International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, March 2012, pp. 89–101.