Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks

Amirali Boroumand[1] Saugata Ghose[1] Youngsok Kim[2]
Rachata Ausavarungnirun[1] Eric Shiu[3] Rahul Thakur[3] Daehyun Kim[4,3]
Aki Kuusela[3] Allan Knies[3] Parthasarathy Ranganathan[3] Onur Mutlu[5,1]

[1]Carnegie Mellon University [2]Dept. of ECE, Seoul National University [3]Google [4]Samsung Research [5]ETH Zürich

Abstract

We are experiencing an explosive growth in the number of consumer devices, including smartphones, tablets, web-based computers such as Chromebooks, and wearable devices. For this class of devices, energy efficiency is a first-class concern due to the limited battery capacity and thermal power budget. We find that *data movement* is a major contributor to the total system energy and execution time in consumer devices. The energy and performance costs of moving data between the memory system and the compute units are significantly higher than the costs of computation. As a result, addressing data movement is crucial for consumer devices.

In this work, we comprehensively analyze the energy and performance impact of data movement for several widely-used Google consumer workloads: (1) the Chrome web browser; (2) TensorFlow Mobile, Google's machine learning framework; (3) video playback, and (4) video capture, both of which are used in many video services such as YouTube and Google Hangouts. We find that *processing-in-memory* (PIM) can significantly reduce data movement for all of these workloads, by performing part of the computation close to memory. Each workload contains simple primitives and functions that contribute to a significant amount of the overall data movement. We investigate whether these primitives and functions are feasible to implement using PIM, given the limited area and power constraints of consumer devices. Our analysis shows that offloading these primitives to PIM logic, consisting of either simple cores or specialized accelerators, eliminates a large amount of data movement, and significantly reduces total system energy (by an average of 55.4% across the workloads) and execution time (by an average of 54.2%).

CCS Concepts • **Hardware → Power and energy**; **Memory and dense storage**; • **Human-centered computing → Ubiquitous and mobile devices**; • **Computer systems organization** → *Heterogeneous (hybrid) systems*;

Keywords processing-in-memory; data movement; consumer workloads; memory systems; energy efficiency

ACM Reference Format:
A. Boroumand et al. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 16 pages. http://dx.doi.org/10.1145/3173162.3173177

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03…$15.00
http://dx.doi.org/10.1145/3173162.3173177

1 Introduction

Consumer devices, which include smartphones, tablets, web-based computers such as Chromebook [40], and wearable devices, have become increasingly ubiquitous in recent years. There were 2.3 billion smartphone users worldwide in 2017 [32]. Tablets have witnessed similar growth, as 51% of the U.S. population owns a tablet as of 2016, and 1.2 billion people in the world use tablets [32]. There is similar demand for web-based computers like Chromebooks [40], which now account for 58% of all computer shipments to schools in the U.S. [63].

Energy consumption is a first-class concern for consumer devices. The performance requirements of these consumer devices have increased dramatically every year to support emerging applications such as 4K video streaming and recording [154], virtual reality (VR) [87], and augmented reality (AR) [6, 18, 60, 93]. A consumer device integrates many power-hungry components such as powerful CPUs, a GPU, special-purpose accelerators, sensors, and a high-resolution screen. Despite the rapid growth in processing capability, two trends greatly limit the performance of consumer devices. First, lithium-ion battery capacity has only doubled in the last 20 years [26, 123]. Second, the thermal power dissipation of consumer devices has become a severe performance constraint [60]. Therefore, fundamentally energy-efficient design of consumer devices is critical to keep up with increasing user demands and to support a wide range of emerging applications [18, 82, 87, 100–102, 109, 154].

To identify the major sources of energy consumption in consumer devices, we conduct an in-depth analysis of several popular Google consumer workloads, as they account for a significant portion of the applications executed on consumer devices. We analyze (1) Chrome [39], the most commonly-used web browser [108]; (2) TensorFlow Mobile [52], Google's machine learning framework that is used in various services such as Google Translate [49], Google Now [48], and Google Photos [46]; (3) video playback [55] and (4) video capture using the VP9 codec [55], which is used by many video services such as YouTube [53], Skype [97], and Google Hangouts [45]. These workloads are among the most commonly-used applications by consumer device users [25, 28, 33, 68, 113], and they form the core of many Google services (e.g., Gmail [42], YouTube [53], the Android OS [38], Google Search [47]) that each have over a billion monthly active users [54, 117].

We make a **key observation** based on our comprehensive workload analysis: among the many sources of energy consumption in consumer devices (e.g., CPUs, GPU, special purpose accelerators, memory), *data movement* between the main memory system and computation units (e.g., CPUs, GPU, special-purpose accelerators) is a major contributor to the total system energy. For example, when the user scrolls through a Google Docs [44] web page, moving data between memory and computation units causes 77% of the total system energy consumption (Section 4.2.1). This is due to the fact that the energy cost of moving data is orders of magnitude higher than the energy cost of computation [80]. We find that across all of the applications we study, 62.7% of the total system energy, on average, is spent on data movement between main memory and the compute units.

Based on our key observation, we find that we can substantially reduce the total system energy if we greatly mitigate the cost of data movement. One potential way to do so is to execute the data-movement-heavy portions of our applications *close to the data*. Recent advances in 3D-stacked memory technology have enabled cost-effective solutions to realize this idea [71, 75, 90, 94]. 3D-stacked DRAM architectures include a dedicated *logic layer*, that is capable of providing logic functionality, with high-bandwidth low-latency connectivity to DRAM layers. Recent works [3–5, 12, 30, 35, 36, 56, 59, 65, 66, 81, 82, 98, 99, 106, 116, 138, 150, 151, 153] take advantage of the logic layer [71, 75, 90, 94] to perform *processing-in-memory* (PIM), also known as *near-data processing*. PIM allows the CPU to dispatch parts of the application for execution on compute units that are close to DRAM. Offloading computation using PIM has two major benefits. First, it eliminates a significant portion of the data movement between main memory and conventional processors. Second, it can take advantage of the high-bandwidth and low-latency access to the data inside 3D-stacked DRAM.

However, there are challenges against introducing PIM in consumer devices. Consumer devices are extremely stringent in terms of the area and energy budget they can accommodate for any new hardware enhancement. Regardless of the memory technology used to enable PIM (whether it is HMC [71], HBM [75], or some other form of 3D-stacked or compute-capable memory [85, 90, 125, 126]), any additional logic can potentially translate into a significant cost in consumer devices. In fact, unlike prior proposals for PIM in server or desktop environments, consumer devices may not be able to afford the addition of full-blown general-purpose PIM cores [12, 30, 35], GPU PIM cores [65, 116, 153], or sophisticated PIM accelerators [3, 36, 66] to 3D-stacked memory. As a result, a major challenge for enabling PIM in consumer devices is to identify what kind of in-memory logic can both (1) *maximize energy efficiency* and (2) be implemented at *minimum possible cost*.

To investigate the potential benefits of PIM, given the area and energy constraints of consumer devices, we delve further into each consumer workload to understand what underlying functions and characteristics contribute most to data movement. Our analysis leads to a **second key observation**: across all of the consumer workloads we examine, there are often simple functions and primitives (which we refer to as *PIM targets*) that are responsible for a significant fraction of the total data movement. These PIM targets range from simple data reorganization operations, such as tiling and packing, to value interpolation and quantization. For example, as we show in Sections 4 and 5, the data movement cost of data reorganization operations in Chrome and TensorFlow Mobile account for up to 25.7% and 35.3% of the total system energy, respectively.

We find that many of these PIM targets are comprised of simple operations such as *memcopy, memset*, and basic arithmetic and bitwise operations. Such PIM targets are mostly data-intensive and require relatively little and simple computation. For example, texture tiling in Chrome is comprised of *memcopy*, basic arithmetic, and bitwise operations. We find that the PIM targets can be implemented as PIM logic using either (1) a small low-power general-purpose embedded core (which we refer to as a *PIM core*) or (2) a group of small fixed-function accelerators (*PIM accelerators*). Our analysis shows that the area of a PIM core and a PIM accelerator take up no more than 9.4% and 35.4%, respectively, of the area available for PIM logic in an HMC-like [71] 3D-stacked memory architecture (Section 3.2). Thus, the PIM core and PIM accelerator are cost-effective to use in consumer devices.

Our comprehensive experimental evaluation shows that PIM cores are sufficient to eliminate a majority of data movement, due to the computational simplicity and high memory intensity of the PIM targets. On average across all of the consumer workloads that we examine, PIM cores provide a 49.1% energy reduction (up to 59.4%) and a 44.6% performance improvement (up to 2.2x) for a state-of-the-art consumer device. We find that PIM accelerators provide larger benefits, with an average energy reduction of 55.4% (up to 73.5%) and performance improvement of 54.2% (up to 2.5x) for a state-of-the-art consumer device. However, PIM accelerators require custom logic to be implemented for each separate workload. We find that PIM accelerators are especially effective for workloads such as video playback, which already make use of specialized hardware accelerators in consumer devices.

We make the following key contributions in this work:

- We conduct the first comprehensive analysis of important Google consumer workloads, including the Chrome browser [39], TensorFlow Mobile [52], video playback [55], and video capture [55], to identify major sources of energy consumption.
- We observe that data movement between the main memory and conventional computation units is a major contributor to the total system energy consumption in consumer devices. On average, data movement accounts for 62.7% of the total energy consumed by Google consumer workloads.
- We observe that most of the data movement in consumer workloads is generated by simple functions and primitives. These functions and primitives are composed of operations such as *memcopy*, *memset*, basic arithmetic operations, and bitwise operations, all of which can be implemented in hardware at low cost.
- We perform the first detailed analysis of processing-in-memory (PIM) for consumer devices, considering the stringent power and area constraints of such devices. Our evaluation shows that we can design cost-efficient PIM logic that significantly reduces the total system energy and execution time of consumer workloads, by 55.4% and 54.2%, respectively, averaged across all of the consumer workloads we study.

2 Background

Processing-in-memory (PIM) involves embedding logic directly within a memory device, and by offloading some of the computation onto this embedded logic. Early works on PIM [86, 115, 129, 134] add substantial processing logic completely inside a DRAM chip. Later works [29, 79, 95, 112] propose more versatile substrates that increase the flexibility and computational capability available within the DRAM chip. These proposals have limited to no adoption, as they require very costly DRAM–logic integration.

The emergence of 3D-stacked DRAM architectures [71, 75, 90, 94] offers a promising solution. These architectures stack multiple layers of DRAM arrays within a single chip, and use vertical *through-silicon vias* (TSVs) to provide much greater bandwidth between layers than the off-chip bandwidth available between DRAM and the CPUs. Several 3D-stacked DRAM architectures (e.g., HBM [75], HMC [71]) provide a dedicated *logic layer* within the stack that can have low-complexity (due to thermal constraints) logic. Recent works on PIM embed computation units within the logic layer (e.g., [3–5, 12, 30, 35, 36, 56, 59, 65, 66, 81, 82, 98, 99, 106, 116, 138, 150, 151, 153]). However, these previous proposals are *not* designed for the highly-stringent area, power, and thermal constraints of modern commercial consumer devices.

We refer the reader to our survey works [37, 127, 128] for more detail on prior processing-in-memory proposals.

3 Analyzing and Mitigating Data Movement

Our goal in this work is to (1) understand the data movement related bottlenecks in modern consumer workloads, (2) comprehensively analyze the benefits that PIM can provide for such workloads, and (3) investigate the PIM logic that can benefit these workloads while still being feasible to implement given the limited area, power, and thermal budgets of consumer devices.

We start by identifying those portions of the consumer workloads that cause significant data movement and that are best suited for PIM execution. We examine four widely-used Google consumer

workloads: (1) the Chrome web browser [39]; (2) TensorFlow Mobile [52], Google's machine learning framework; (3) video playback, and (4) video capture using the VP9 video codec [55]. These workloads form the core of many Google services (e.g., Gmail [42], YouTube [53], the Android OS [38], Google Search [47]) that each have over a billion monthly active users [54, 117].

In this section, we discuss our characterization methodology (Section 3.1), our approach for identifying PIM targets in each consumer workload (Section 3.2), and the types of PIM logic we use to implement the PIM targets (Section 3.3).

3.1 Workload Analysis Methodology

We perform our workload characterization on a Chromebook [40] with an Intel Celeron (N3060 dual core) SoC [72] and 2GB of DRAM. Our performance and traffic analyses are based on hardware performance counters within the SoC.

We build our energy model based on prior work [114], which sums up the total energy consumed by the CPU cores, DRAM, off-chip interconnects, and all caches. We use hardware performance counters to drive this model. During our characterization, we turn off Wi-Fi and use the lowest brightness for the display to ensure that the majority of the total energy is spent on the SoC and the memory system [113, 114]. We use CACTI-P 6.5 [104] with a 22nm process technology to estimate L1 and L2 cache energy. We estimate the CPU energy based on prior works [114, 143] and scale the energy to accurately fit our Celeron processor. We model the 3D-stacked DRAM energy as the energy consumed per bit, using estimates and models from prior works [74, 124]. We conservatively use the energy of the ARM Cortex-R8 to estimate the energy consumed by a PIM core, and we estimate the PIM accelerator energy based on [1], conservatively assuming that the accelerator is 20x more energy-efficient than the CPU cores.

Chrome Web Browser. We analyze Chrome [39] using the Telemetry framework [23], which automates user actions on a set of web pages. We analyze three pages that belong to important Google web services (Google Docs [44], Gmail [42], Google Calendar [43]), two of the top 25 most-accessed web sites [7] (WordPress [149] and Twitter [142]), and one animation-heavy page [23].

TensorFlow Mobile. To analyze TensorFlow Mobile [52], we profile four representative neural networks: VGG-19 [131], ResNet-v2-152 [62], Inception-ResNet-v2 [137], and Residual-GRU [141]. The first three networks are for image classification, and Residual-GRU is for image compression. All four networks have a number of use cases on mobile consumer devices (e.g., grouping similar images, reducing the size of newly-taken photos on the fly).

Video Playback and Video Capture. We evaluate both hardware and software implementations of the VP9 codec [55, 146]. We use publicly-available video frames [152] as inputs to the VP9 decoder and encoder. For the in-house hardware implementation, we use a bit-level C++ model to accurately model all traffic between each component of the hardware and DRAM. The RTL for the commercial VP9 hardware is generated from this C++ model using Calypto Catapult [96].

3.2 Identifying PIM Targets

We use hardware performance counters and our energy model to identify candidate functions that could be PIM targets. A function is a PIM target candidate if (1) it consumes the most energy out of the all functions in the workload, (2) its data movement consumes a significant fraction of the total workload energy, (3) it is memory-intensive (i.e., its last-level cache *misses per kilo instruction*, or MPKI, is greater than 10 [21, 83, 84, 103]), and (4) data movement is the single largest component of the function's energy consumption. We then check if each candidate is amenable to PIM logic implementation using two criteria. First, we discard any PIM targets that incur any performance loss when run on simple PIM logic (i.e., PIM core, PIM accelerator). Second, we discard any PIM targets that require more area than is available in the logic layer of 3D-stacked memory (see Section 3.3). In the rest of this paper, we study only PIM target candidates that pass both of these criteria.

3.3 Implementing PIM Targets in 3D-Stacked DRAM

For each workload, once we have identified a PIM target, we propose PIM logic that can perform the PIM target functionality inside the logic layer of 3D-stacked memory. We propose two types of PIM logic: (1) a general-purpose *PIM core*, where a single PIM core can be used by *any* of our PIM targets; and (2) a fixed-function *PIM accelerator*, where we design custom logic for each PIM target.

For the PIM core, we design a custom 64-bit low-power single-issue core similar in design to the ARM Cortex-R8 [9] (see Section 9 for more details). All of the PIM targets that we evaluate are data-intensive, and the majority of them perform only simple operations (e.g., *memcopy*, basic arithmetic operations, bitwise operations). As a result, we do *not* implement aggressive instruction-level parallelism (ILP) techniques (e.g., sophisticated branch predictors, superscalar execution) in our core. Several of our PIM targets exhibit highly *data-parallel* behavior, leading us to incorporate a SIMD unit that can perform a single operation on multiple pieces of data concurrently. We empirically set the width of our SIMD unit to 4.

For the PIM accelerator for each of our PIM targets, we first design a customized *in-memory logic unit*, which is fixed-function logic that performs a single thread of the PIM target in the logic layer. To exploit the data-parallel nature of the PIM targets, we add multiple copies of the in-memory logic unit, so that multiple threads of execution can be performed concurrently.

We evaluate the performance and energy consumption of a state-of-the-art consumer device that contains either PIM cores or PIM accelerators in 3D-stacked memory (Section 10). In order to assess the feasibility of implementing our PIM cores or PIM accelerators in memory, we estimate the area consumed by both types of PIM logic for a 22 nm process technology. We assume that the 3D-stacked memory contains multiple *vaults* (i.e., vertical slices of 3D-stacked DRAM), and that we add one PIM core or PIM accelerator per vault. Assuming an HMC-like [71] 3D-stacked memory architecture for PIM, there is around 50–60 mm^2 of area available for architects to add new logic into the DRAM logic layer. This translates to an available area of approximately 3.5–4.4 mm^2 *per vault* to implement our PIM logic [30, 36, 74].[1] We find that each PIM core requires less than 0.33 mm^2 of area in the logic layer, conservatively based on the footprint of the ARM Cortex-R8 [9]. This requires no more than 9.4% of the area available per vault to implement PIM logic. The area required for each PIM accelerator depends on the PIM target being implemented, and we report these area numbers in Sections 4–7.

4 Chrome Web Browser

A web browser is one of the most commonly-used applications by consumer device users [28], and is listed as one of the most common applications in many mobile benchmarks [57, 68, 113]. We study Google Chrome, which has claimed the majority of the mobile browsing market share for several years [108], and has over a billion active users [117].

The user perception of the browser speed is based on three factors: (1) page load time, (2) smooth web page scrolling, and (3) quick switching between browser tabs. In our study, we focus on two user interactions that impact these three factors, and govern a user's browsing experience: (1) page scrolling and (2) tab switching. Note that each interaction includes page loading.

[1] As a comparison, a typical system-on-chip (SoC) used in a consumer device has an area of 50–100 mm^2 [132, 136, 139, 147, 148].

4.1 Background

When a web page is downloaded, the rendering engine in Chrome, called Blink [22], parses the HTML to construct a Document Object Model (DOM) tree, which consists of the internal elements of the web page represented as tree nodes. Blink also parses the style rule data from the Cascading Style Sheet (CSS). The DOM tree and style rules enable a visual representation of the page, called the *render tree*. Each render tree node, called a *render object*, requires geometric information of the corresponding element to paint it to the display. The process of calculating the position and size of each render object is called *layout*. Once layout is complete, Chrome uses the Skia library [51] to perform *rasterization*, where a bitmap is generated for each render object by recursively traversing the render tree. The rasterized bitmap (also known as a *texture*) is then sent to the GPU through a process called *texture upload*, after which the GPU performs *compositing*. During compositing, the GPU paints the pixels corresponding to the texture onto the screen.

4.2 Page Scrolling

Scrolling triggers three operations: (1) *layout*, (2) *rasterization*, and (3) *compositing*. All three operations *must* happen within the mobile screen refresh time (60 FPS or 16.7 ms [92, 105]) to avoid frame dropping. Scrolling often forces the browser to recompute the layout for the new dimensions and position of each web page element, which is highly compute-intensive. However, a careful design of web pages can mitigate this cost [135, 140]. The browser also needs to rasterize any new objects that were not displayed previously. Depending on the web page contents and scroll depth, this requires significant computation [91, 140]. The updated rasterized bitmap must then be composited to display the results of the scroll on the screen.

4.2.1 Scrolling Energy Analysis

Figure 1 shows the energy breakdown for scrolling on different web pages. Across all of the pages that we test, a significant portion (41.9%) of page scrolling energy is spent on two data-intensive components: (1) *texture tiling*, where the graphics driver reorganizes the linear bitmap data into a tiled format for the GPU; and (2) *color blitting*, which is invoked by the Skia library [51] during rasterization. The rest of the energy is spent on a variety of other libraries and functions, each of which contributes to less than 1% of the total energy consumption (labeled *Other* in Figure 1).

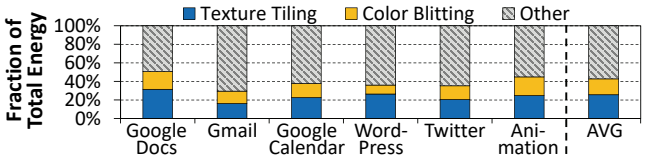

Figure 1. Energy breakdown for page scrolling.

Figure 2 (left) gives more insight into where energy is spent in the system when we scroll through a Google Docs [44] web page. We find that 77% of the total energy consumption is due to data movement. The data movement energy includes the energy consumed by DRAM, the off-chip interconnect, and the on-chip caches. The data movement generated by texture tiling and color blitting alone accounts for 37.7% of the total system energy (right graph in Figure 2). We confirm this by measuring the MPKI issued by the last-level cache (LLC). All of our pages exhibit a high MPKI (21.4 on average), and the majority of LLC misses are generated by texture tiling and color blitting. Texture tiling and color blitting are also the top two contributors to the execution time, accounting for 27.1% of the cycles executed when scrolling through our Google Docs page.

We conclude that texture tiling and color blitting are responsible for a significant portion of the data movement that takes place during web page scrolling.

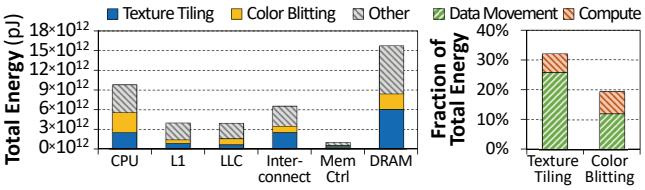

Figure 2. Energy breakdown when scrolling through a Google Docs web page.

4.2.2 Analysis of PIM Effectiveness

In this section, we analyze the suitability of texture tiling and color blitting for PIM execution.

Texture Tiling. Figure 3a illustrates the major steps and associated data movement that take place during the texture tiling process. Texture tiling takes place after rasterization and before compositing. Rasterization generates a linear rasterized bitmap, which is written using a linear access pattern to memory (❶ in the figure). After rasterization, compositing accesses each texture in both the horizontal and vertical directions. To minimize cache misses during compositing, the graphics driver reads the rasterized bitmap (❷) and converts the bitmap into a *tiled* texture layout (❸). For example, the Intel HD Graphics driver breaks down each rasterized bitmap into multiple 4 kB texture tiles [70]. This allows the GPU to perform compositing on only one tile from the bitmap at a time to improve data locality.

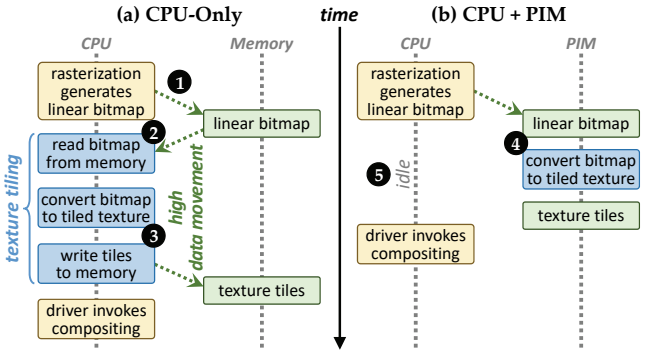

Figure 3. Texture tiling on (a) CPU vs. (b) PIM.

As we observe from Figure 2 (right), 25.7% of the total system energy is spent on data movement generated by texture tiling. In fact, only 18.5% of the total energy consumed by texture tiling is used for computation, and the rest goes to data movement. The majority of the data movement comes from (1) the poor data locality during texture tiling; and (2) the large rasterized bitmap size (e.g., 1024x1024 pixels, which is 4 MB), which typically exceeds the LLC capacity. Due to the high amount of data movement, texture tiling is a good candidate for PIM execution. As shown in Figure 3b, by moving texture tiling to PIM (❹ in the figure), we can free up the CPU (❺) to perform other important compute-intensive tasks in Chrome, such as handling user interaction, executing JavaScript code, or rasterizing other render objects.

We next determine whether texture tiling can be implemented *in a cost-effective manner* using PIM. Our analysis indicates that texture tiling requires only simple primitives: *memcopy*, bitwise operations, and simple arithmetic operations (e.g., addition). These operations can be performed at high performance on our PIM core (see Section 3.3), and are amenable to be implemented as a fixed-function PIM accelerator. Our PIM accelerator for texture tiling consists of multiple *in-memory tiling units*. Each in-memory tiling unit consists of only a simple ALU, and operates on a single 4 kB tile. We empirically decide to use four in-memory tiling units in each PIM accelerator. Using the area estimation approach proposed

in prior work [30], we estimate that the overhead of each PIM accelerator is less than 0.25 mm^2, which requires no more than 7.1% of the area available per vault for PIM logic (see Section 3.3). We conclude that the area required for texture tiling PIM logic is small, making the PIM logic feasible to implement in a consumer device with 3D-stacked memory. We evaluate the energy efficiency and performance of both the PIM core and PIM accelerator for texture tiling in Section 10.1.

An alternate way to reduce the data movement cost of texture tiling is to directly rasterize the content in the GPU, instead of the CPU, by using OpenGL primitives [24]. While GPU rasterization eliminates the need to move textures around the system, the GPU's highly-parallel architecture is not a good fit for rasterizing fonts and other small shapes [73]. We observe this issue in particular for text-intensive pages. We find that when Chrome uses the GPU to rasterize text-intensive pages, the page load time *increases* by up to 24.9%. This is one of the main reasons that the GPU rasterization is *not* enabled by default in Chrome. PIM can significantly reduce the overhead of texture tiling, while still allowing Chrome to exploit the benefits of CPU-based rasterization.

Color Blitting. During rasterization, the Skia library [51] uses high-level functions to draw basic primitives (e.g., lines, text) for each render object. Each of these high-level functions uses a *color blitter*, which converts the basic primitives into the bitmaps that are sent to the GPU. The primary operation of a blitter is to copy a block of pixels from one location to another. Blitting has many uses cases, such as drawing lines and filling paths, performing double buffering, alpha compositing, and combining two images or primitives together.

Color blitting requires simple computation operations, but generates a large amount of data movement. As we see in Figure 2 (right), color blitting accounts for 19.1% of the total system energy used during page scrolling. 63.9% of the energy consumed by color blitting is due to data movement (right graph in Figure 2), primarily due to its streaming access pattern and the large sizes of the bitmaps (e.g., 1024x1024 pixels). Similar to texture tiling, color blitting is a good candidate for PIM execution due to its high amount of data movement.

We next determine whether color blitting can be implemented in a cost-effective manner using PIM. Our analysis reveals that color blitting requires only low-cost computations such as *memset*, simple arithmetic operations to perform alpha blending (e.g., addition and multiplication), and shift operations. These operations can be performed at high performance on our PIM core (see Section 3.3), or we can use a PIM accelerator that consists of the same four in-memory logic units that we design for texture tiling, with different control logic specifically designed to perform color blitting. Thus, we conclude that the area required for color blitting PIM logic is small, and that the PIM logic is feasible to implement in a consumer device with 3D-stacked memory. We evaluate the energy efficiency and performance of both the PIM core and PIM accelerator for color blitting in Section 10.1.

4.3 Tab Switching

Chrome uses a multi-process architecture, where each tab has its own application process for rendering the page displayed in the tab (see Section 4.1), to enhance reliability and security. When a user switches between browser tabs, this triggers two operations: (1) a context switch to the process for the selected tab, and (2) a load operation for the new page. There are two primary concerns during tab switching: (1) how fast a new tab loads and becomes interactive, as this directly affects user satisfaction; and (2) memory consumption. Memory consumption is a major concern for three reasons. First, the average memory footprint of a web page has increased significantly in recent years [67] due to the increased use of images, JavaScript, and video in modern web pages. Second, users tend to open multiple tabs at a time [31, 110], with each tab consuming additional memory. Third, consumer devices typically have a much lower memory capacity than server or desktop systems, and without careful memory management, Chrome could run out of memory on these systems over time.

One potential solution to handle a memory shortage is to kill inactive background tabs, and when the user accesses those tabs again, reload the tab pages from the disk. There are two downsides to this mechanism. First, reloading tabs from the disk (which invokes page faults) and rebuilding the page objects take a relatively long time. Second, such reloading may lead to the loss of some parts of the page (e.g., active remote connections). To avoid these drawbacks, Chrome uses memory compression to reduce each tab's memory footprint. When the available memory is lower than a predetermined threshold, Chrome compresses pages of an inactive tab, with assistance from the OS, and places them into a DRAM-based memory pool, called ZRAM [76]. When the user switches to a previously-inactive tab whose pages were compressed, Chrome loads the data from ZRAM and decompresses it. This allows the browser to avoid expensive I/O operations to disk, and retrieve the web page data much faster, thereby improving overall performance and the browsing experience.

4.3.1 Tab Switching Energy Analysis

To study data movement during tab switching, we perform an experiment where a user (1) opens 50 tabs (picked from the top most-accessed websites [7]), (2) scrolls through each tab for a few seconds, and then (3) switches to the next tab. During this study, we monitor the amount of data that is swapped in and out of ZRAM per second, which we plot in Figure 4. We observe that, in total, 11.7 GB of data is swapped out to ZRAM (left graph in Figure 4), at a rate of up to 201 MB/s. When the CPU performs this swapping, as shown in Figure 5a, it incurs significant overhead, as it must read the inactive page data (❶ in the figure), compress the data (❷), and write the compressed data to ZRAM (❸). We observe a similar behavior for decompression (right graph in Figure 4), with as much as 7.8 GB of data swapped in, at a rate of up to 227 MB/s. Note that during decompression, most of the newly-decompressed page cache lines are not accessed by the CPU, as the tab rendering process needs to read only a small fraction of the decompressed cache lines to display the tab's contents. When switching between 50 tabs, compression and decompression incur 19.6 GB of data movement, and contribute to 18.1% and 14.2% of the total system energy and execution time of tab switching, respectively.

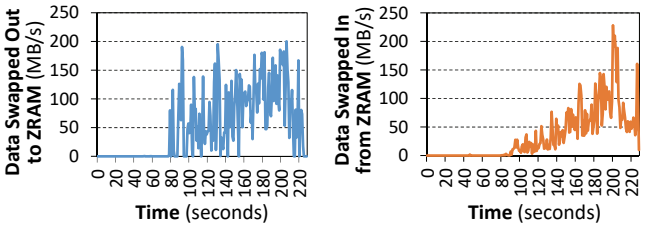

Figure 4. Number of bytes per second swapped out to ZRAM (left) and in from ZRAM (right), while switching between 50 tabs.

4.3.2 Analysis of PIM Effectiveness

Compression and decompression are a good fit for PIM execution, as (1) they cause a large amount of data movement; and (2) compression can be handled in the background, since none of the active tab processes need the data that is being compressed. Figure 5b shows the compression operation once it has been offloaded to PIM. In this case, the CPU only informs the PIM logic about the pages that need to be swapped out, and the PIM logic operates on the uncompressed page data that is already in memory (❹ in the figure). This (1) eliminates the off-chip page data movement

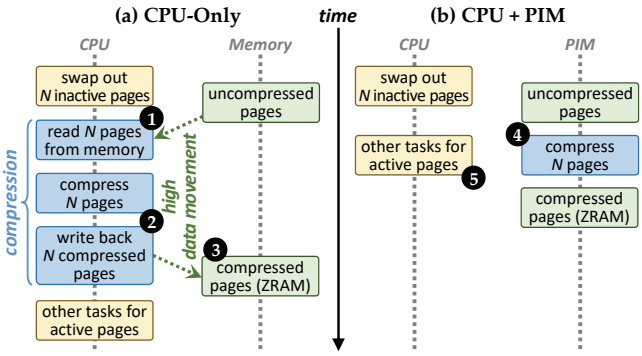

Figure 5. Compression on (a) CPU vs. (b) PIM.

that took place during CPU-based compression, and (2) frees up the CPU to perform other tasks while PIM performs compression in the background (❺). We observe a similar behavior for tab decompression, where the compressed data is kept within DRAM, and only those cache lines used later by the CPU are sent across the off-chip channel, reducing data movement and improving CPU cache utilization.

We find that compression and decompression are good candidates for PIM execution. Chrome's ZRAM uses the LZO compression algorithm [111] which uses simple operations and favors speed over compression ratio [78, 111]. Thus, LZO can execute without performance loss on our PIM core (see Section 3.3). Prior work [156] shows that sophisticated compression algorithms (e.g., LZ77 compression [160] with Huffman encoding [69]) can be implemented efficiently using an accelerator. Thus, we expect that the simpler LZO compression algorithm can be implemented as a PIM accelerator that requires less than 0.25 mm^2 of area [156]. Thus, both the PIM core and PIM accelerator are feasible to implement in-memory compression/decompression.

In-memory compression/decompression can benefit a number of other applications and use cases. For example, many modern operating systems support user-transparent file system compression (e.g., BTRFS [122] and ZFS [11]). Such compression is not yet widely supported in commercial *mobile* operating systems due to energy and performance issues [156]. An in-memory compression unit can enable efficient user-transparent file system compression by eliminating the off-chip data movement cost and largely reducing the decompression latency.

5 TensorFlow Mobile

Machine learning (ML) is emerging as an important core function for consumer devices. Recent works from academia and industry are pushing inference to mobile devices [8, 14, 19, 118, 119], as opposed to performing inference on cloud servers. In this work, we study TensorFlow Mobile [52], a version of Google's TensorFlow ML library that is specifically tailored for mobile and embedded platforms. TensorFlow Mobile enables a variety of tasks, such as image classification, face recognition, and Google Translate's instant visual translation [50], all of which perform inference on consumer devices using a neural network that was pre-trained on cloud servers.

5.1 Background

Inference begins by feeding input data (e.g., an image) to a neural network. A neural network is a directed acyclic graph consisting of multiple layers. Each layer performs a number of calculations and forwards the results to the next layer. Depending on the type of the layer, the calculation can differ for each level. A fully-connected layer performs matrix multiplication (MatMul) on the input data, to extract high-level features. A 2-D convolution layer applies a convolution filter (Conv2D) across the input data, to extract low-level features. The last layer of a neural network is the output layer, which performs classification to generate a prediction based on the input data.

5.2 TensorFlow Mobile Energy Analysis

Figure 6 shows the breakdown of the energy consumed by each function in TensorFlow Mobile, for four different input networks. As convolutional neural networks (CNNs) consist mainly of 2-D convolution layers and fully-connected layers [2], the majority of energy is spent on these two types of layers. However, we find that there are two other functions that consume a significant fraction of the system energy: *packing/unpacking* and *quantization*. Packing and unpacking reorder the elements of matrices to minimize cache misses during matrix multiplication. Quantization converts 32-bit floating point and integer values into 8-bit integers to improve the execution time and energy consumption of inference. These two together account for 39.3% of total system energy on average. The rest of the energy is spent on a variety of other functions such as random sampling, reductions, and simple arithmetic, and each of which contributes to less than 1% of total energy consumption (labeled *Other* in Figure 6).

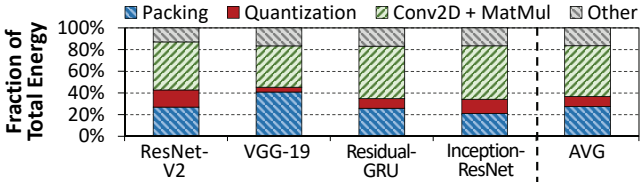

Figure 6. Energy breakdown during inference execution on four input networks.

Further analysis (not shown) reveals that data movement between the CPUs and main memory accounts for the majority of total system energy consumed by TensorFlow Mobile. While Conv2D and MatMul dominate CPU energy consumption, 57.3% of the total system energy is spent on data movement, on average across our four input networks. We find that 54.4% of the data movement energy comes from packing/unpacking and quantization.

Even though the main goal of packing and quantization is to reduce energy consumption and inference latency, our analysis shows that they generate a large amount of data movement, and thus, lose part of the energy savings they aim to achieve. Furthermore, Figure 7 shows a significant portion (27.4% on average) of the execution time is spent on the packing and quantization process. Hence, we focus our analysis on these two functions. We exclude Conv2D and MatMul from our analysis because (1) a majority (67.5%) of their energy is spent on computation; and (2) Conv2D and MatMul require a relatively large and sophisticated amount of PIM logic [36, 81], which may not be cost-effective for consumer devices.

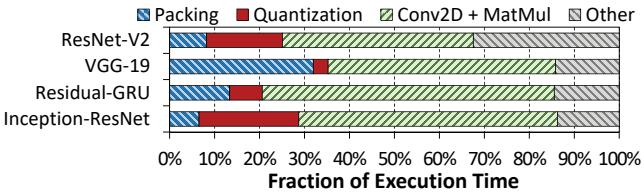

Figure 7. Execution time breakdown of inference.

5.3 Analysis of PIM Effectiveness

Packing. GEneralized Matrix Multiplication (GEMM) is the core building block of neural networks, and is used by both 2-D convolution and fully-connected layers. These two layers account for the majority of TensorFlow Mobile execution time. To implement fast and energy-efficient GEMM, TensorFlow Mobile employs a low-precision, quantized GEMM library called *gemmlowp* [41]. The

gemmlowp library performs GEMM by executing its innermost kernel, an architecture-specific GEMM code portion for small fixed-size matrix chunks, multiple times. First, gemmlowp fetches matrix chunks which fit into the LLC from DRAM. Then, it executes the GEMM kernel on the fetched matrix chunks in a block-wise manner.

To minimize cache misses, gemmlowp employs a process called *packing*, which reorders the matrix chunks based on the memory access pattern of the kernel to make the chunks cache-friendly. After performing GEMM, gemmlowp performs *unpacking*, which converts the result matrix chunk back to its original order.

Packing and unpacking account for up to 40% of the total system energy and 31% of the inference execution time, as shown in Figures 6 and 7, respectively. Due to their unfriendly cache access pattern and the large matrix sizes, packing and unpacking generate a significant amount of data movement. For instance, for VGG-19 [131], 35.3% of the total energy goes to data movement incurred by packing-related functions. On average, we find that data movement is responsible for 82.1% of the total energy consumed during the packing/unpacking process, indicating that packing and unpacking are bottlenecked by data movement.

Packing and unpacking are simply pre-processing steps, to prepare data in the right format for the kernel. Ideally, the CPU should execute only the GEMM kernel, and assume that packing and unpacking are already taken care of. PIM can enable such a scenario by performing packing and unpacking without *any* CPU involvement. Our PIM logic packs matrix chunks, and sends the packed chunks to the CPU, which executes the GEMM kernel. Once the GEMM kernel completes, the PIM logic receives the result matrix chunk from the CPU, and unpacks the chunk while the CPU executes the GEMM kernel on a different matrix chunk.

We next determine whether packing and unpacking can be implemented in a cost-effective manner using PIM. Our analysis reveals that packing is a simple data reorganization process and requires simple arithmetic operations to scan over matrices and compute new indices. As a result, packing and unpacking can be performed at high performance on our PIM core (see Section 3.3), or on a PIM accelerator that consists of the same four in-memory logic units that we design for texture tiling (see Section 4.2.2), with different control logic specifically designed to perform packing and unpacking. For packing and unpacking, we can assign each in-memory logic unit to work on one chunk of the matrix. Thus, we conclude that it is feasible to implement packing and unpacking using PIM in a consumer device with 3D-stacked memory.

Quantization. TensorFlow Mobile performs quantization twice for each Conv2D operation. First, quantization is performed on the 32-bit input matrix before Conv2D starts. Then, Conv2D runs, during which gemmlowp generates a 32-bit result matrix.[2] Quantization is performed for the second time on this result matrix (this step is referred to as *re-quantization*). Accordingly, invoking Conv2D more frequently (which occurs when there are more 2-D convolution layers in a network) leads to higher quantization overheads. For example, VGG requires only 19 Conv2D operations, incurring small quantization overheads. On the other hand, ResNet requires 156 Conv2D operations, causing quantization to consume 16.1% of the total system energy and 16.8% of the execution time. The quantization overheads are expected to increase as neural networks get deeper.

Figure 8a shows how TensorFlow quantizes the result matrix using the CPU. First, the entire matrix needs to be scanned to identify the minimum and maximum values of the matrix (❶ in the figure). Then, using the minimum and maximum values, the matrix is scanned a second time to convert each 32-bit element of the matrix into an 8-bit integer (❷). These steps are repeated for re-quantization of the result matrix (❸ and ❹). The majority of the quantization overhead comes from data movement. Because both the input matrix quantization and the result matrix re-quantization need to scan a large matrix twice, they exhibit poor cache locality and incur a large amount of data movement. For example, for the ResNet network, 73.5% of the energy consumed during quantization is spent on data movement, indicating that the computation is relatively cheap (in comparison, only 32.5% of Conv2D/MatMul energy goes to data movement, while the majority goes to MAC computation). 19.8% of the total data movement energy of inference execution comes from quantization. As Figure 8b shows, we can offload both quantization (❺ in the figure) and re-quantization (❻) to PIM to eliminate data movement. This frees up the CPU to focus on GEMM execution, and allows the next Conv2D operation to be performed in parallel with re-quantization (❼).

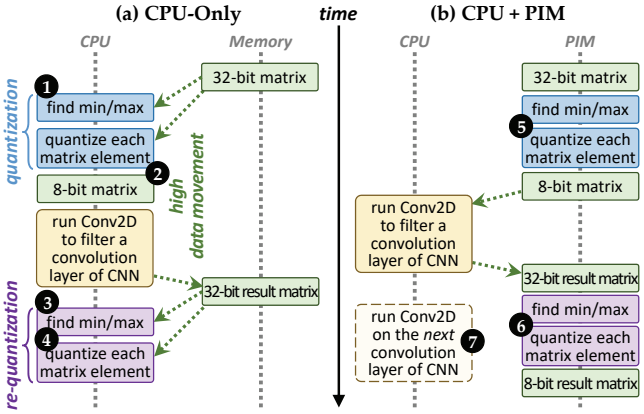

Figure 8. Quantization on (a) CPU vs. (b) PIM.

Quantization itself is a simple data conversion operation that requires shift, addition, and multiplication operations. As a result, quantization can be performed at high performance on our PIM core (see Section 3.3), or on a PIM accelerator that consists of the same four in-memory logic units that we design for texture tiling (see Section 4.2.2), with different control logic specifically designed to perform quantization. For quantization, we can assign each in-memory logic unit to work on a separate Conv2D operation. Thus, we conclude that it is feasible to implement quantization using PIM in a consumer device with 3D-stacked memory.

6 Video Playback

Video playback is among the most heavily- and commonly-used applications among mobile users [155]. Recent reports on mobile traffic [25, 33] show that video dominates consumer device traffic today. Many Google services and products, such as YouTube [53], Google Hangouts [45], and the Chrome web browser [39] rely heavily on video playback. YouTube alone has over a billion users, and the majority of video views on YouTube come from mobile devices [54]. Video playback requires a dedicated decoder, which decompresses and decodes the streaming video data and renders video on the consumer device. In this work, we focus on the VP9 decoder [55], which is an open-source codec widely used by video-based applications, and is supported by most consumer devices [144].

6.1 Background

Figure 9 shows a high-level overview of the VP9 decoder architecture. VP9 processes video one frame at a time, and uses a *reference frame* to decode the current frame. For the current frame, the decoder reads the frame's compressed content (i.e., the input bitstream) from memory (❶ in the figure) and sends the content to the *entropy decoder*. The entropy decoder (❷) extracts (1) *motion vectors*, which predict how objects move relative to the reference

[2]Even though gemmlowp reads 8-bit input matrices, its output matrix uses 32-bit integers, because multiplying two 8-bit integers produces a 16-bit result, and gemmlowp uses a 32-bit integer to store each 16-bit result.

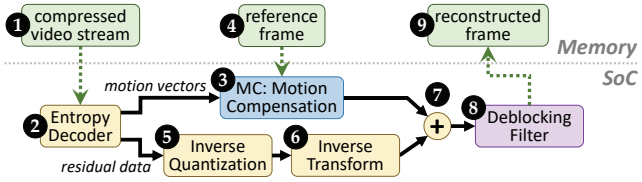

Figure 9. General overview of the VP9 decoder.

frame; and (2) *residual data*, which fills in information not predicted by the motion vectors.

The motion vectors are sent to the *motion compensation* (MC) unit (❸), which applies the object movement predictions to the reference frame (❹). MC predicts movement at a macro-block (i.e., 16x16-pixel) granularity. Each frame is decomposed into macro-blocks, and each motion vector points to a macro-block in the reference frame. VP9 allows motion vectors to have a resolution as small as $\frac{1}{8}$th of a pixel, which means that a motion vector may point to a pixel location with a non-integer value. In such cases, MC performs *sub-pixel interpolation* to estimate the image at a non-integer pixel location. In parallel with MC, the residual data is transformed by the *inverse quantization* (❺) and *inverse transform* (❻) blocks. The resulting information is combined with the macro-block output by MC (❼) to reconstruct the current frame.

Due to block-based prediction, there may be discontinuities at the border between two blocks. The *deblocking filter* (❽) attempts to remove such artifacts by (1) identifying *edge pixels* (i.e., pixels that lie on the border of two blocks) that are discontinuous with their neighbors, and (2) applying a low-pass filter to these pixels. Finally, the reconstructed frame (❾) is written back to the frame buffer in memory.

The VP9 decoder architecture can be implemented in either software or hardware. We provide an extensive analysis of two decoder implementations in this section: (1) libvpx [145], Google's open-source software decoder library, which is used as a reference software implementation for the VP9 codec; and (2) Google's VP9 hardware decoder [146].

6.2 VP9 Software Decoder

6.2.1 Software Decoder Energy Analysis

Figure 10 shows the total system energy consumed when libvpx VP9 decoder is used to play a 4K (3840x2160-pixel) video from Netflix [152]. We find that the majority of energy (53.4%) is spent on MC, which is the most bandwidth-intensive and time-consuming task in the decoder [13, 64]. Prior works [58, 88, 120] make similar observations for other video codecs. Most of the energy consumption in MC is due to sub-pixel interpolation, the most memory-intensive component of MC [13, 64, 77]. We find that sub-pixel interpolation alone consumes 37.5% of the total energy and 41.2% of the execution time spent *by the entire decoder*. Aside from MC, the deblocking filter is another major component of the decoder, consuming 29.7% of the software decoder energy and 24.3% of the total cycle count. Other components, such as the entropy decoder and inverse transform, consume a smaller portion of the energy.

Figure 11 shows the per-function breakdown of the energy consumption of each hardware component when the VP9 software decoder is used to play back 4K video. We find that 63.5% of the

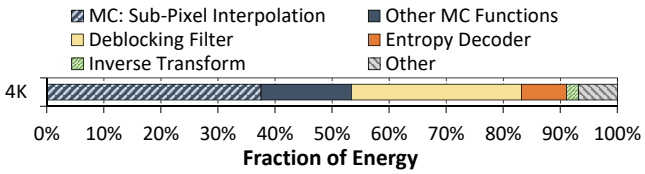

Figure 10. Energy analysis of VP9 software decoder.

total energy is spent on data movement, the majority (80.4%) of which is performed by MC and the deblocking filter. Sub-pixel interpolation is the dominant function of MC, and on its own generates 42.6% of the *total* data movement. We find that the CPU spends the majority of its time and energy stalling (not shown in Figure 11) as it waits for data from memory during MC and deblocking filter execution. The other functions generate much less data movement because their working set fits within the CPU caches. For example, the entropy decoder works on the encoded bit stream and the inverse transform works on the decoded coefficients, and both are small enough to be captured by the caches.

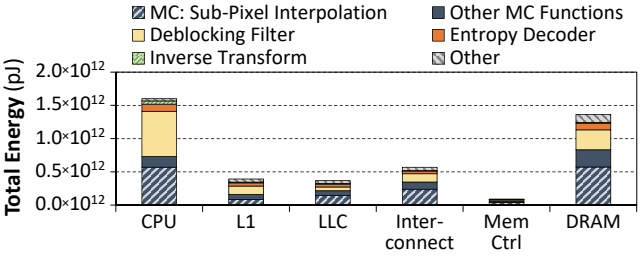

Figure 11. Energy breakdown of VP9 software decoder.

We conclude that video playback generates a large amount of data movement, the majority of which comes from sub-pixel interpolation and the deblocking filter.

6.2.2 Analysis of PIM Effectiveness

Sub-Pixel Interpolation. During sub-pixel interpolation, VP9 uses a combination of multi-tap finite infinite response (FIR) and bi-linear filters to interpolate the value of pixels at non-integer locations [55]. The interpolation process is both compute- and memory-intensive, for two reasons. First, interpolating each sub-pixel value requires multiple pixels to be fetched from memory. VP9 decomposes each frame into 64x64-pixel superblocks, which are then divided further into smaller sub-blocks, as small as 4x4-pixel. In the worst case, where the decoder interpolates pixels in a sub-block at a $\frac{1}{8}$-pixel resolution, the decoder fetches 11x11 pixels from the reference frame. Second, each motion vector can point to *any* location in the reference frame. As a result, there is poor locality, and sub-pixel interpolation does *not* benefit significantly from caching. We confirm this in Figure 11, where we observe that data movement between main memory and the CPU accounts for 65.3% of the total energy consumed by sub-pixel interpolation. As a result, sub-pixel interpolation is a good candidate for PIM execution.

We find that sub-pixel interpolation is amenable for PIM, as it consists mainly of multiplication, addition, and shift operations [55, 64, 77]. As a result, sub-pixel interpolation can be executed with high performance on our PIM core (see Section 3.3), and can be implemented as a fixed-function PIM accelerator. Our PIM accelerator for sub-pixel interpolation is similar in design to the sub-pixel interpolation component of the hardware VP9 decoder. Our analysis shows that our PIM accelerator occupies an area of 0.21 mm^2, which requires no more than 6.0% of the area available per vault for PIM logic (see Section 3.3). Thus, we conclude that it is feasible to implement sub-pixel interpolation using PIM in a consumer device with 3D-stacked memory.

Deblocking Filter. Recall that the deblocking filter works on the edge pixels in each 64x64-pixel superblock, to remove blocking artifacts. Deblocking invokes a low-pass filter on neighboring edge pixels, and iterates through the superblocks in a raster scan order [55]. For each edge between two superblocks, the filter evaluates up to eight pixels on either side of the edge, and if the filter condition is triggered, the filter may modify up to seven pixels on either side.

Like sub-pixel interpolation, the deblocking filter is compute- and memory-intensive, and accounts for a third of the computation

complexity of the entire VP9 decoder [20, 27]. As the filter needs to check the vertical and horizontal edges of each 4x4-pixel block in the frame, its memory access pattern exhibits poor cache locality. Instead, the filter generates a large amount of data movement and produces strictly less output than input, with 71.1% of the movement taking place on the off-chip memory channel. Hence, the deblocking filter is a good fit for PIM.

We find that the deblocking filter is amenable for PIM, as it is a simple low-pass filter that requires only arithmetic and bitwise operations. As a result, the deblocking filter can be executed with high performance on our PIM core (see Section 3.3), and can be implemented as a fixed-function PIM accelerator that is is similar in design to the deblocking filter component of the hardware VP9 decoder. Our analysis shows that our PIM accelerator occupies an area of 0.12 mm^2, which requires no more than 3.4% of the area available per vault for PIM logic (see Section 3.3). Thus, we conclude that it is feasible to implement the deblocking filter using PIM in a consumer device with 3D-stacked memory.

6.3 VP9 Hardware Decoder

For high-resolution (e.g., 4K) video playback, a hardware decoder is essential. It enables fast video decoding at much lower energy consumption than a software decoder. In this section, we present our analysis on the VP9 hardware decoder, which is used in a large fraction of consumer devices [144].

Unlike the software decoder, the VP9 hardware decoder employs several techniques to hide memory latency, such as prefetching and parallelization. For example, the hardware decoder can work on a batch of motion vectors simultaneously, which allows the decoder to fetch the reference pixels for the entire batch at once. In addition, the hardware decoder does not make use of the CPU caches, and instead sends requests directly to DRAM, which allows the decoder to avoid cache lookup latencies and cache coherence overheads. Thus, there is little room to improve the memory latency for the hardware decoder. However, we find that despite these optimizations, there is still a significant amount of off-chip data movement generated by the hardware decoder, which consumes significant memory bandwidth and system energy.

6.3.1 Hardware Decoder Energy Analysis

Figure 12 shows the breakdown of off-chip traffic when decoding one frame of a 4K video and an HD (720x1280-pixel) video. For each resolution, we show the traffic both with and without lossless frame compression. We make five key observations from the analysis. First, the majority of the traffic (up to 75.5% for HD and 59.6% for 4K) comes from reading the reference frame data from DRAM during MC. This data movement is responsible for 71.3% and 69.2% of total hardware decoder energy, respectively. Note that while frame compression reduces the reference frame traffic, reference frame data movement still accounts for a significant portion (62.2% for HD and 48.8% for 4K) of the off-chip traffic. Second, similar to our observations for the software decoder, the bulk of this data movement is due to the extra reference pixels required by sub-pixel interpolation. For every pixel of the current frame, the decoder reads 2.9 reference frame pixels from DRAM. Third, data movement increases significantly as the video resolution increases. Our analysis indicates that decoding one 4K frame requires 4.6x the data movement of a single HD frame. Fourth, the reconstructed frame data output by the decoder is the second-biggest contributor to the off-chip data movement, generating 22.2% of the total traffic. Fifth, unlike the software decoder, the deblocking filter does not generate much off-chip data movement in the hardware decoder, as the hardware employs large SRAM buffers (875 kB) to cache the reference pixels read during MC, so that the pixels can be reused during deblocking.

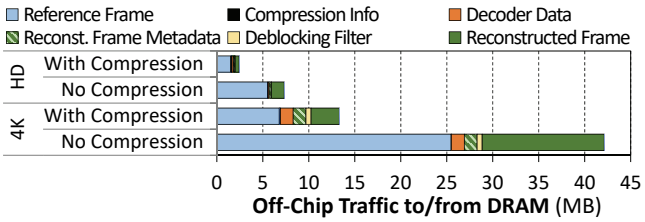

Figure 12. Off-chip traffic breakdown of VP9 hardware decoder.

6.3.2 Analysis of PIM Effectiveness

Similar to our observations for the software decoder (see Section 6.2), the MC unit (sub-pixel interpolation, in particular) is responsible for the majority of off-chip data movement in the hardware VP9 decoder. The MC unit is a good fit for PIM execution, because we can eliminate the need to move reference frames from memory to the on-chip decoder, thus saving significant energy. Figure 13 shows a high-level overview of our proposed architecture, and how a modified hardware decoder interacts with memory and with the *in-memory* MC unit. In this architecture, the entropy decoder (❶ in the figure) sends the motion vectors to memory, where the MC unit now resides. The in-memory MC unit (❷) generates macro-blocks using the motion vectors and the reference frame (❸) fetched from memory. The macro-blocks generated by MC are then combined with residual data and sent to the deblocking filter (❹). To avoid sending the macro-blocks back to the decoder (which would eliminate part of the energy savings of not moving reference frame data), we also made the design choice of moving the deblocking filter into memory (❹). While the deblocking filter itself does *not* generate significant traffic, by placing it in PIM, we can avoid moving the reconstructed frame (❺) back and forth on the main memory bus unnecessarily.

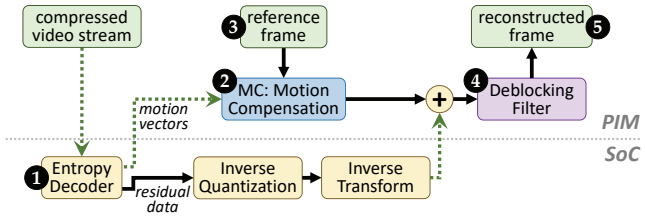

Figure 13. Modified VP9 decoder with in-memory MC.

Overall, moving the MC unit and deblocking filter reduces data movement significantly, as only the compressed input bitstream, the motion vectors, and the residual data need to move between the decoder hardware components and the memory. Another benefit of this approach is that we can use the same in-memory components for software VP9 decoding as well. In total, the area used by a PIM accelerator that performs MC and the deblocking filter is 0.33 mm^2, which requires no more than 9.4% of the area available per vault for PIM logic (see Section 3.3). We conclude that it is effective to perform part of VP9 hardware decoding in memory.

7 Video Capture

Video capture is used in many consumer applications, such as video conferencing (e.g., Google Hangouts [45], Skype [97]), video streaming (e.g., YouTube [53], Instagram [34]), and recording local video from a mobile device camera. Video-capture-based applications account for a significant portion of mobile traffic [25, 33, 155]. It is expected that video will constitute 78% of all mobile data traffic by 2021 [25]. To effectively capture, store, and transfer video content over the network, a video encoder is required. The video encoder handles the majority of the computation for video capture. In this work, we focus on the VP9 encoder, as it is the main compression technology used by Google Hangouts [45], YouTube [53], and most

web-based video applications in the Chrome browser [39], and is supported by many mobile devices [144].

7.1 Background

Figure 14 shows a high-level overview of the VP9 encoder architecture. The encoder fetches the current frame (❶ in the figure) and three reference frames (❷) from DRAM. Then, the current frame is sliced into 16x16-pixel macro-blocks. Each macro-block is passed into the *intra-prediction* (❸) and *motion estimation* (ME) (❹) units. Intra-prediction predicts the value of an entire macro-block by measuring its similarity to adjacent blocks within the frame. ME, also known as *inter-prediction*, compresses the frame by exploiting temporal redundancy between adjacent frames. Then, ME finds the motion vector, which encodes the spatial offset of each macro-block relative to the reference frames. If successful, ME replaces the macro-block with the motion vector. The *mode decision unit* (❺) examines the outputs of intra-prediction and ME for each macro-block, and chooses the best prediction for the block. The macro-block prediction is then (1) transformed into the frequency domain using a discrete cosine (DCT) *transform* (❻), and (2) simultaneously sent to the motion compensation (MC) unit (❼) to reconstruct the currently-encoded frame, which will be used as a reference frame (❽) to encode *future* frames. The encoder then uses quantization (❾) to compress the DCT coefficients of each macro-block. Finally, the *entropy coder* step (❿) uses *run-length coding* and *variable-length coding* to reduce the number of bits required to represent the DCT coefficients.

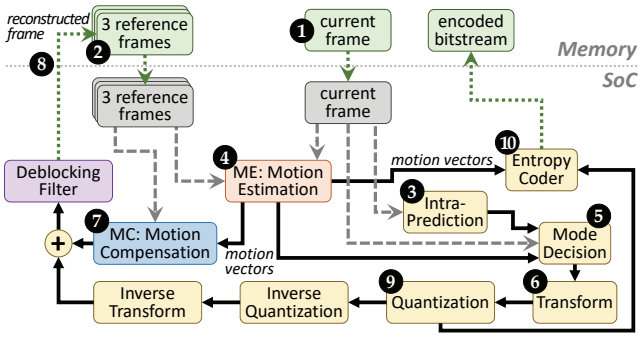

Figure 14. General overview of the VP9 encoder.

As is the case for the VP9 decoder, the encoder can be implemented in either software or hardware. We analyze both libvpx [145], Google's open source software encoder library, and Google's VP9 hardware encoder [146].

7.2 VP9 Software Encoder

7.2.1 Software Encoder Energy Analysis

Figure 15 shows our energy analysis during the real-time encoding of an HD video. We find that a significant portion of the system energy (39.6%) is spent on ME. ME is the most memory-intensive unit in the encoder [133], accounting for 43.1% of the total encoding cycles. The next major contributor to total system energy is the deblocking filter, which we discuss in in Section 6.2. While intra-prediction, transform, and quantization are other major contributors to the energy consumption, none of them individually account for more than 9% of the total energy. The remaining energy, labeled *Other* in the figure, is spent on decoding the encoded frame, and behaves the same way as the software decoding we discuss in Section 6.2.

We find that 59.1% of encoder energy goes to data movement. The majority of this data movement is generated by ME, and accounts for 21.3% of the total system energy. Other major contributors to data movement are the MC and deblocking filter used to decode the encoded frames (see Section 6.2).

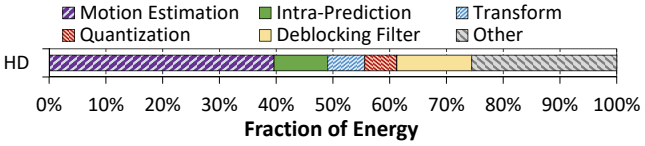

Figure 15. Energy analysis of VP9 software encoder.

7.2.2 Analysis of PIM Effectiveness

In libvpx, ME uses the diamond search algorithm [157] to locate matching objects in reference frames, using the sum of absolute differences (SAD) to evaluate matches. ME is highly memory-intensive and requires high memory bandwidth. 54.7% of ME's energy consumption is spent on data movement (not shown). For each macro-block, ME needs to check for matching objects in *three* reference frames, which leads to a large amount of data movement. Hence, ME is a good candidate for PIM execution.

While ME is more compute- and memory-intensive than the other PIM targets we examine, it primarily calculates SAD, which requires only simple arithmetic operations. As a result, ME can be executed with high performance on our PIM core (see Section 3.3), and can be implemented as a fixed-function PIM accelerator that is similar in design to the ME component of the hardware VP9 decoder. Our analysis shows that our PIM accelerator occupies an area of 1.24 mm^2, which requires no more than 35.4% of the area available per vault for PIM logic (see Section 3.3). Thus, we conclude that it is feasible to implement ME using PIM in a consumer device with 3D-stacked memory.

7.3 VP9 Hardware Encoder

In this section, we present our analysis on the VP9 hardware encoder. While software encoding offers more flexibility, hardware encoding enables much faster encoding in an energy-efficient manner. Similar to the hardware video decoder (Section 6.3), the hardware video encoder employs prefetching to hide memory latency. Unlike the decoder, the encoder's memory access pattern is highly predictable, as the search window for each reference frame is predefined. As a result, the encoder (specifically the ME unit) successfully hides latency. However, energy consumption and the cost of data movement remain as major challenges.

7.3.1 Hardware Encoder Energy Analysis

Figure 16 shows the breakdown of off-chip traffic for encoding a single frame of 4K and HD video, both with and without lossless frame compression. Similar to our findings for the software encoder, the majority of the off-chip traffic is due to the reference frame pixels fetched by ME (shown as *Reference Frame* in the figure), which accounts for 65.1% of the entire encoder's data movement for HD video. The traffic grows significantly when we move to 4K video, as the encoder requires 4.3x the number of pixels read during HD video encoding. While frame compression can reduce the amount of data transferred by 59.7%, a significant portion of encoder energy is still spent on reference frame traffic.

The two other major contributors to off-chip traffic are (1) the current frame after encoding, and (2) the reconstructed frame, which is used as a future reference frame. The current frame generates 14.2% of the traffic when frame compression is disabled, but since

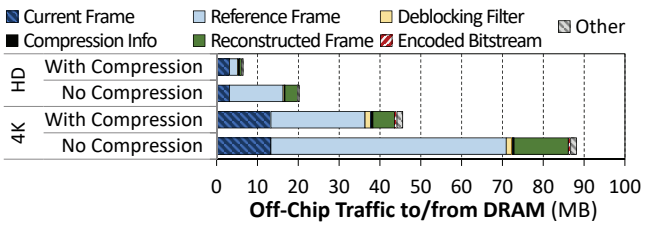

Figure 16. Off-chip traffic breakdown of VP9 hardware encoder.

compression cannot be applied to the encoded version of the current frame, the total data movement for the current frame takes up to 31.9% of the traffic when compression is enabled. The reconstructed frame consumes on average 12.4% of the traffic across the two resolutions.

7.3.2 Analysis of PIM Effectiveness

Similar to the software encoder, ME is responsible for the majority of data movement in the hardware encoder, and is again a good candidate for PIM execution, which would eliminate the need to move three reference frames to the on-chip encoder. Figure 17 shows the high-level architecture of a modified VP9 encoder with in-memory ME (which is the same as the PIM accelerator in Section 7.2.2). Once in-memory ME (❶ in the figure) generates the motion vector for a macro-block, it sends the vector back to the mode decision unit (❷) in the on-chip accelerator. Note that we also offload MC (❸) and the deblocking filter (❹) to PIM, similar to the hardware decoder (see Section 6.3.2), since they use reference frames (❺) together with the motion vectors output by ME to reconstruct the encoded frame. This eliminates data movement during frame reconstruction. We conclude that data movement can be reduced significantly by implementing the ME, MC, and deblocking filter components of the VP9 hardware encoder in memory.

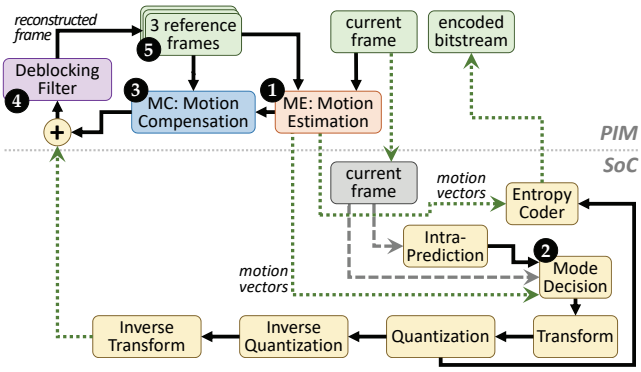

Figure 17. Modified VP9 encoder with in-memory ME.

8 System Integration

We briefly discuss two system challenges to implementing PIM in the consumer device context.

8.1 Software Interface for Efficient Offloading

We use a simple interface to offload the PIM targets. We identify PIM targets using two macros, which mark the beginning and end of the code segment that should be offloaded to PIM. The compiler converts these macros into instructions that we add to the ISA, which trigger and end PIM kernel execution. Examples are provides in a related work [37].

8.2 Coherence Between CPUs and PIM Logic

A major system challenge to implementing PIM is CPU-PIM communication and coherence. The majority of the PIM targets we identified are minimal functions/primitives that are interleaved with other parts of the workloads in a fine-grained manner. As a result, to fully benefit from offloading them to PIM, we need efficient communication between the CPU and PIM logic to allow coordination, ordering, and synchronization between different parts of the workloads. We find that coherence is required to (1) enable efficient synchronization, (2) retain programmability, and (3) retain the benefits' of using PIM. As a result, we employ a simple fine-grained coherence technique, which uses a local PIM-side directory in the logic layer to maintain coherence between PIM cores (or PIM accelerators), and to enable low-overhead fine-grained coherence between PIM logic and the CPUs. The CPU-side directory acts as the main coherence point for the system, interfacing with both the processor caches and the PIM-side directory. Our design can benefit further from other approaches to coherence like LazyPIM [12, 37].

9 Methodology

We evaluate our proposed PIM architectures using the gem5 [10] full-system simulator. Table 1 lists our system configuration.

SoC	4 OoO cores, 8-wide issue; *L1 I/D Caches*: 64 kB private, 4-way assoc.; *L2 Cache:* 2 MB shared, 8-way assoc.; *Coherence*: MESI
PIM Core	1 core per vault, 1-wide issue, 4-wide SIMD unit, *L1 I/D Caches*: 32 kB private, 4-way assoc.
3D-Stacked Memory	2 GB cube, 16 vaults per cube; *Internal Bandwidth:* 256 GB/s; *Off-Chip Channel Bandwidth:* 32 GB/s
Baseline Memory	LPDDR3, 2 GB, FR-FCFS scheduler

Table 1. Evaluated system configuration.

We organize our PIM architecture such that each vault in 3D-stacked memory has its own PIM logic. The PIM logic for each vault consists of either a PIM core or a PIM accelerator. We assume that the PIM core is ISA-compatible with the main processor cores in the SoC, and is a simple general-purpose core (e.g., 1-wide issue, no prefetcher) that has only a 32KB private L1 cache. Each PIM accelerator has a small 32kB buffer to hold its working set of data. Like prior works [3, 4, 12, 35, 65, 66], we use the memory bandwidth available to the logic layer of 3D-stacked memory as the memory bandwidth to the PIM core/accelerator. We model our 3D-stacked memory similar to HBM [75] and use the bandwidth available in the logic layer of HBM (256GB/s), which is 8x more than the bandwidth available (32GB/s) to the off-chip CPU cores.

To evaluate how PIM benefits each of the PIM targets that we identified, we study each target in isolation, by emulating each target separately, constructing a microbenchmark for the component, and then analyzing the benefit of PIM in our simulator. We use Verilog RTL [96] to estimate the area overhead of the VP9 hardware encoder and decoder.

Chrome Browser. For texture tiling, we precisely emulate `glTextImage2d()` for OpenGL from the Intel i965 graphics driver, using 512x512-pixel RGBA tiles as the input set. For color blitting, we construct a microbenchmark that closely follows the color blitting implementation in Skia, and use randomly-generated bitmaps (ranging from 32x32 to 1024x1024 pixels) as inputs. For compression and decompression, we use LZO [111] from the latest upstream Linux version. We generate the input data by starting Chrome on our Chromebook, opening 50 tabs, navigating through them, and then dumping the entire contents of the Chromebook's main memory to a file.

TensorFlow Mobile. For packing, we modify the *gemmlowp* library [41] to perform only matrix packing, by disabling the quantized matrix multiplication and result matrix unpacking routines. We evaluate in-memory quantization using a microbenchmark that invokes TensorFlow Mobile's post-Conv2D/MatMul requantization routines, and we use the result matrix sizes of GEMMs to reflect real-world usage.

Video Playback and Capture. For sub-pixel interpolation and the deblocking filter, we construct microbenchmarks that closely follow their implementations in *libvpx*, and use 100 frames from a 4K Netflix video [152] as an input. For motion estimation, we develop a microbenchmark that uses the diamond search algorithm [157] to perform block matching using three reference frames, and use 10 frames from an HD video [152] as an input. For the hardware VP9 evaluation, we use the off-chip traffic analysis (Figures 12 and 16) together with in-memory ME and in-memory MC traffic analysis from our simulator to model the energy cost of data movement.

10 Evaluation

In this section, we examine how our four consumer workloads benefit from PIM. We show results normalized to a processor-only baseline (*CPU-Only*), and compare to PIM execution using PIM cores (*PIM-Core*) or fixed-function PIM accelerators (*PIM-Acc*).

10.1 Chrome Browser

Figure 18 shows the energy consumption and runtime of CPU-Only, PIM-Core, and PIM-Acc across different browser kernels, normalized to CPU-only. We make three key observations on energy from the left graph in Figure 18. First, we find that offloading these kernels to PIM reduces the energy consumption, on average across all of our evaluated web pages, by 51.3% when we use PIM-Core, and by 61.0% when we use PIM-Acc. Second, we find that the majority of this reduction comes from eliminating data movement. For example, eliminating data movement in texture tiling contributes to 77.7% of the total energy reduction. Third, PIM-Acc provides 18.9% more energy reduction over PIM-Core, though the energy reduction of PIM-Acc is limited by the highly-memory-intensive behavior of these browser kernels. This is because while computation on PIM-Acc is more efficient than computation using PIM-Core, data movement accounts for the majority of energy consumption in the kernels, limiting the impact of more efficient computation on overall energy consumption.

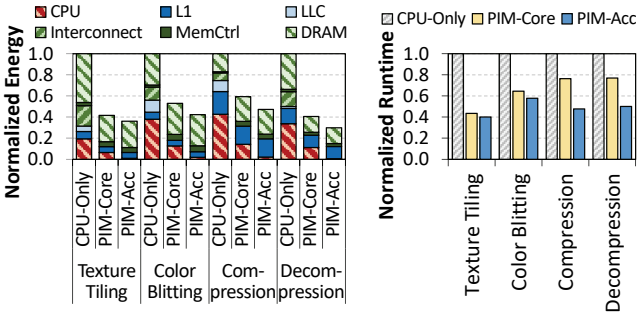

Figure 18. Energy (left) and runtime (right) for all browser kernels, normalized to CPU-Only, for kernel inputs listed in Section 9.

We make five observations on performance from the right graph in Figure 18. First, PIM-Core and PIM-Acc significantly outperform CPU-Only across all kernels, improving performance by 1.6x and 2.0x, on average across all of our evaluated web pages. Second, we observe that the browser kernels benefit from the higher bandwidth and lower access latency of 3D-stacked memory (due to their data-intensive nature), leading to performance improvement. Third, this performance benefit increases as the working set size increases. For example, our analysis (not shown) shows that the speedup increases by 31.2% when the texture size increases from 256x256 to 512x512 pixels. Fourth, the performance improvement of PIM-Acc over PIM-Core for texture tiling and color blitting is limited by the low computational complexity of these two kernels. Fifth, compression and decompression benefit significantly from using PIM-Acc over PIM-Core in terms of performance, because these kernels are more compute-intensive and less data-intensive compared to texture tiling and color blitting.

10.2 TensorFlow Mobile

Figure 19 (left) shows the energy consumption of CPU-Only, PIM-Core, and PIM-Acc for the four most time- and energy-consuming GEMM operations for each input neural network in packing and quantization, normalized to CPU-Only. We make three key observations. First, PIM-Core and PIM-Acc decrease the total energy consumption by 50.9% and 54.9%, on average across all four input networks, compared to CPU-Only. Second, the majority of the energy savings comes from the reduction in data movement, as the

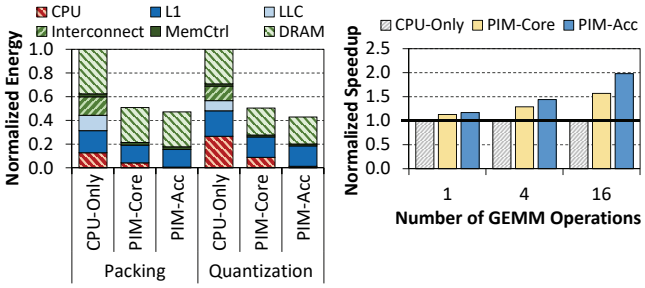

Figure 19. Energy (left) and performance (right) for TensorFlow Mobile kernels, for four neural networks [62, 131, 137, 141].

computation energy accounts for a negligible portion of the total energy consumption. For instance, 82.6% of the energy reduction for packing is due to the reduced data movement. Third, we find that the data-intensive nature of these kernels and their low computational complexity limit the energy benefits PIM-Acc provides over PIM-Core.

Figure 19 (right) shows the total execution time of CPU-Only, PIM-Core and PIM-Acc as we vary the number of GEMM operations performed. For CPU-Only, we evaluate a scenario where the CPU performs packing, GEMM operations, quantization, and unpacking. To evaluate PIM-Core and PIM-Acc, we assume that packing and quantization are handled by the PIM logic, and the CPU performs GEMM operations. We find that as the number of GEMM operations increases, PIM-Core and PIM-Acc provide greater performance improvements over CPU-Only. For example, for one GEMM operation, PIM-Core and PIM-Acc achieve speedups of 13.1% and 17.2%, respectively. For 16 GEMM operations, the speedups of PIM-Core and PIM-Acc increase to 57.2% and 98.1%, respectively, over CPU-Only. These improvements are the result of PIM logic (1) exploiting the higher bandwidth and lower latency of 3D-stacked memory, and (2) enabling the CPU to perform GEMM in parallel while the PIM logic handles packing and quantization. For example, offloading packing to the PIM core speeds up the operation by 32.1%. Similar to our observation for the browser kernels (Section 10.1), as these kernels are not compute-intensive, the performance improvement of PIM-Acc over PIM-Core is limited.

10.3 Video Playback and Capture

10.3.1 VP9 Software

Figure 20 (left) shows the energy consumption of CPU-Only, PIM-Core, and PIM-Acc across different video kernels, normalized to CPU-Only. We make three observations from the figure. First, all of the kernels significantly benefit from PIM-Core and PIM-Acc, with an average energy reduction of 46.8% and 66.6%, respectively, across our input videos. Second, the energy reductions are a result of the memory-bound behavior of these video kernels. Moving the kernels to PIM logic (1) eliminates a majority of the data movement, and (2) allows the kernels to benefit from the lower power consumed by PIM logic than by the CPU without losing performance. For example, offloading sub-pixel interpolation and the deblocking filter to PIM-Core reduces their total energy consumption by 49.9% and 50.1%, respectively. Of those reductions, 34.6% and 51.9% come from the reduced energy consumption of PIM-Core compared to the SoC CPU, while the rest of the reduction comes from decreasing data movement. Third, we find that PIM-Acc provides, on average, a 42.7% additional energy reduction than PIM-Core across all of the video kernels.

Figure 20 (right) shows the runtime of CPU-Only, PIM-Core, and PIM-Acc on our video kernels. We observe from the figure that PIM-Core and PIM-Acc improve performance over CPU-Only by 23.6% and 70.2%, respectively, averaged across all input videos. While PIM-Core provides a modest speed up (12.6%) over CPU-Only for the motion estimation kernel, PIM-Acc significantly outperforms

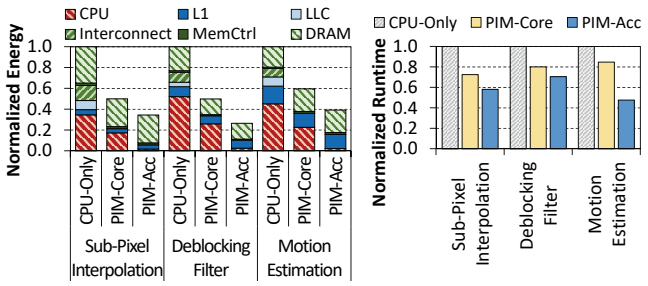

Figure 20. Energy (left) and runtime (right) for all video kernels, normalized to CPU-Only, using 100 4K frames [152] for decoding and 10 HD [152] frames for encoding.

CPU-Only and improves performance by 2.1x. The reason is that motion estimation kernel has relatively high computational demand, compared to the other PIM targets we evaluate, and thus, benefits much more from executing on PIM logic.

10.3.2 VP9 Hardware

Figure 21 shows the total energy consumption for the decoder (left) and encoder (right), for three configurations: VP9 only (which uses the baseline VP9 hardware accelerator [146] in the SoC), VP9 with PIM-Core, and VP9 with PIM-Acc. For each configuration, we show results without and with lossless frame compression.

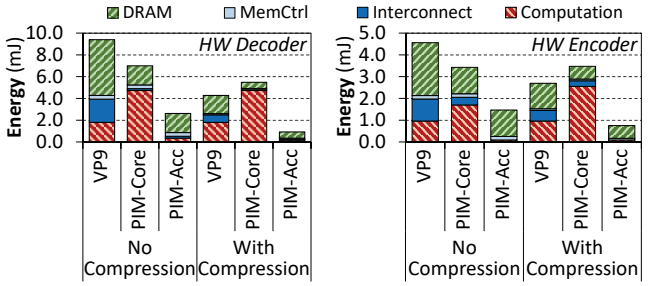

Figure 21. Total energy for VP9 hardware decoder (left) and VP9 hardware encoder (right), based on an analysis using 4K and HD video [152].

We make four key observations from the figure. First, we find that for both the VP9 decoder and encoder, off-chip data movement is the major source of energy consumption, consuming 69.2% and 71.5% of the total decoder and encoder energy, respectively. Second, we find that computation using the VP9 hardware is an order of magnitude more energy-efficient than computation using PIM-Core. As a result, when compression is enabled, PIM-Core actually consumes 63.4% *more* energy than the VP9 baseline. Third, we find that PIM-Acc reduces energy when compared to the VP9 hardware baseline, by 75.1% for decoding and by 69.8% for encoding. This is because PIM-Acc embeds parts of the VP9 hardware itself in memory, retaining the computational energy efficiency of the baseline accelerator while reducing the amount of data movement that takes place. Fourth, we find that PIM-Acc without compression uses less energy than the VP9 hardware baseline with compression. This indicates that the PIM logic is more effective at reducing data movement than compression alone. We achieve the greatest energy reduction by combining PIM-Acc with compression.

11 Other Related Work

To our knowledge, this is the first work to (1) conduct a comprehensive analysis of Google's consumer device workloads to identify major sources of energy consumption, with a focus on data movement; and (2) analyze and evaluate how processing-in-memory (PIM) benefits consumer devices, given the stringent power and area constraints of such devices. Section 2 briefly discusses related work on PIM. In this section, we discuss other related works.

Prior works [15, 57, 68, 113, 121] study the general performance and energy profile of mobile applications. Narancic et al. [107] study the memory system behavior of a mobile device. Shingari et al. [130] characterize memory interference for mobile workloads. None of these works analyze (1) the sources of energy consumption and data movement in consumer workloads, or (2) the benefits of PIM for modern consumer workloads. One prior work [114] measures the data movement cost of emerging mobile workloads. However, the work does not identify or analyze the sources of data movement in consumer workloads, and does not propose any mechanism to reduce data movement.

A number of prior works select a single consumer application and propose techniques to improve the application. Many prior works [16, 17, 158, 159] use a variety of techniques, such as microarchitecture support and language extensions, to improve browsing energy and performance. None of these works (1) conduct a detailed analysis of important user interactions (page scrolling and tab switching) when browsing real-world web pages, or (2) identify sources of energy consumption and data movement during these interactions. Other works [13, 20, 27, 77, 88, 89, 155] attempt to improve the efficiency of motion compensation, motion estimation, and the deblocking filter in different video codec formats using various techniques, ranging from software optimization to hardware acceleration. None of these works (1) thoroughly analyze the data movement cost for a video codec, or (2) use PIM to improve the energy and performance of the video codec. Several works [19, 36, 61, 81, 119] focus on accelerating inference. While these mechanisms can speed up inference on mobile TensorFlow, they do *not* address the data movement issues related to packing and quantization. Tetris [36] and Neurocube [81] attempt to push the *entire* inference execution into PIM. While Tetris and Neurocube eliminate data movement during inference, their PIM logic incurs high area and energy overheads, which may not be practical to implement in consumer devices given the *limited* area and power budgets. In contrast, our work identifies the *minimal* functions of inference that benefit from PIM, allowing us to greatly reduce data movement without requiring a large area or energy overhead, and making our PIM logic feasible to implement in consumer devices.

12 Conclusion

Energy is a first-class design constraint in consumer devices. In this paper, we analyze several widely-used Google consumer workloads, and find that data movement contributes to a significant portion (62.7%) of their total energy consumption. Our analysis reveals that the majority of this data movement comes from a number of simple functions and primitives that are good candidates to be executed on low-power processing-in-memory (PIM) logic. We comprehensively study the energy and performance benefit of PIM to address the data movement cost on Google consumer workloads. Our evaluation shows that offloading simple functions from these consumer workloads to PIM logic, consisting of either simple cores or specialized accelerators, reduces system energy consumption by 55.4% and execution time by 54.2%, on average across all of our workloads. We conclude that reducing data movement via processing-in-memory is a promising approach to improve both the performance and energy efficiency of modern consumer devices with tight area, power, and energy budgets.

Acknowledgments

This research started at Google, during Amirali Boroumand's extended internship and Onur Mutlu's stay as a visiting researcher. We thank our industrial partners, especially Google, Huawei, Intel, and VMware, for their generous support. This research was partially supported by the Semiconductor Research Corporation.

References

[1] D. Abts, "Lost in the Bermuda Triangle: Complexity, Energy, and Performance," in *WCED*, 2006.
[2] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, "Fathom: Reference Workloads for Modern Deep Learning Methods," in *IISWC*, 2016.
[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[4] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[5] B. Akin, F. Franchetti, and J. C. Hoe, "Data Reorganization in Memory Using 3D-Stacked DRAM," in *ISCA*, 2015.
[6] A. Al-Shuwaili and O. Simeone, "Energy-Efficient Resource Allocation for Mobile Edge Computing-Based Augmented Reality Applications," *IEEE Wireless Communications Letters*, 2017.
[7] Alexa Internet, Inc., "Website Traffic, Statistics and Analytics," http://www.alexa.com/siteinfo/.
[8] M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava, "RSTensorFlow: GPU Enabled TensorFlow for Deep Learning on Commodity Android Devices," in *EMDL*, 2017.
[9] ARM Holdings PLC, "ARM Cortex-R8," https://developer.arm.com/products/processors/cortex-r/cortex-r8.
[10] N. Binkert, B. Beckman, A. Saidi, G. Black, and A. Basu, "The gem5 Simulator," *Comp. Arch. News*, 2011.
[11] J. Bonwick and B. Moore, "ZFS: The Last Word in File Systems," https://csde.washington.edu/~mbw/OLD/UNIX/zfs_lite.pdf, 2007.
[12] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE CAL*, 2017.
[13] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "HEVC Complexity and Implementation Analysis," *IEEE CSVT*, 2012.
[14] Q. Cao, N. Balasubramanian, and A. Balasubramanian, "MobiRNN: Efficient Recurrent Neural Network Execution on Mobile GPU," in *EMDL*, 2017.
[15] A. Carroll and G. Heiser, "An Analysis of Power Consumption in a Smartphone," in *USENIX ATC*, 2010.
[16] G. Chadha, S. Mahlke, and S. Narayanasamy, "EFetch: Optimizing Instruction Fetch for Event-Driven Web Applications," in *PACT*, 2014.
[17] G. Chadha, S. Mahlke, and S. Narayanasamy, "Accelerating Asynchronous Programs Through Event Sneak Peek," in *ISCA*, 2015.
[18] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui, "Mobile Augmented Reality Survey: From Where We Are to Where We Go," *IEEE Access*, 2017.
[19] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," *JSSC*, 2017.
[20] J.-A. Choi and Y.-S. Ho, "Deblocking Filter Algorithm with Low Complexity for H.264 Video Coding," in *PCM*, 2008.
[21] C. Chou, P. Nair, and M. K. Qureshi, "Reducing Refresh Power in Mobile Devices with Morphable ECC," in *DSN*, 2015.
[22] Chromium Project, "Blink Rendering Engine," https://www.chromium.org/blink.
[23] Chromium Project, "Catapult: Telemetry," https://chromium.googlesource.com/catapult/+/HEAD/telemetry/README.md.
[24] Chromium Project, "GPU Rasterization in Chromium," https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome, 2014.
[25] Cisco Systems, Inc., "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html, 2017.
[26] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *MobiSys*, 2010.
[27] H. Deng, X. Zhu, and Z. Chen, "An Efficient Implementation for H.264 Decoder," in *ICCSIT*, 2010.
[28] T. M. T. Do, J. Blom, and D. Gatica-Perez, "Smartphone Usage in the Wild: A Large-Scale Analysis of Applications and Context," in *ICMI*, 2011.
[29] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, "The Architecture of the DIVA Processing-in-memory Chip," in *ICS*, 2002.
[30] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The Mondrian Data Engine," in *ISCA*, 2017.
[31] P. Dubroy and R. Balakrishnan, "A Study of Tabbed Browsing Among Mozilla Firefox Users," in *CHI*, 2010.
[32] eMarketer, Inc., "Slowing Growth Ahead for Worldwide Internet Audience," https://www.emarketer.com/article/slowing-growth-ahead-worldwide-internet-audience/1014045?soc1001, 2016.
[33] Ericsson, Inc., "Ericsson Mobility Report: On the Pulse of the Networked Society," https://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf, 2015.
[34] Facebook, Inc., "Instagram," https://www.instagram.com/.
[35] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[36] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *ASPLOS*, 2017.
[37] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions," arxiv:1802.00320 [cs.AR], 2018.
[38] Google LLC, "Android," https://www.android.com/.
[39] Google LLC, "Chrome Browser," https://www.google.com/chrome/browser/.
[40] Google LLC, "Chromebook," https://www.google.com/chromebook/.
[41] Google LLC, "gemmlowp: a small self-contained low-precision GEMM library," https://github.com/google/gemmlowp.
[42] Google LLC, "Gmail," https://www.google.com/gmail/.
[43] Google LLC, "Google Calendar," https://calendar.google.com/.
[44] Google LLC, "Google Docs," https://docs.google.com/.
[45] Google LLC, "Google Hangouts," https://hangouts.google.com/.
[46] Google LLC, "Google Photos," https://photos.google.com/.
[47] Google LLC, "Google Search," https://www.google.com/.
[48] Google LLC, "Google Search: About Google App," https://www.google.com/search/about/.
[49] Google LLC, "Google Translate," https://translate.google.com/.
[50] Google LLC, "Google Translate App," https://translate.google.com/intl/en/about/.
[51] Google LLC, "Skia Graphics Library," https://skia.org/.
[52] Google LLC, "TensorFlow: Mobile," https://www.tensorflow.org/mobile/.
[53] Google LLC, "YouTube," https://www.youtube.com/.
[54] Google LLC, "YouTube for Press," https://www.youtube.com/yt/about/press/.
[55] A. Grange, P. de Rivaz, and J. Hunt, "VP9 Bitstream & Decoding Process Specification," http://storage.googleapis.com/downloads.webmproject.org/docs/vp9/vp9-bitstream-specification-v0.6-20160331-draft.pdf.
[56] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti, "3D-Stacked Memory-Side Acceleration: Accelerator and System Design," in *WoNDP*, 2014.
[57] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-System Analysis and Characterization of Interactive Smartphone Applications," in *IISWC*, 2011.
[58] H. Habli, J. Lilius, and J. Ersfolk, "Analysis of Memory Access Optimization for Motion Compensation Frames in MPEG-4," in *SOC*, 2009.
[59] R. Hadidi, L. Nai, H. Kim, and H. Kim, "CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-in-Memory," *ACM TACO*, 2017.
[60] M. Halpern, Y. Zhu, and V. J. Reddi, "Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction," in *HPCA*, 2016.
[61] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and B. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *ISCA*, 2016.
[62] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," in *ECCV*, 2016.
[63] B. Heater, "As Chromebook Sales Soar in Schools, Apple and Microsoft Fight Back," https://techcrunch.com/2017/04/27/as-chromebook-sales-soar-in-schools-apple-and-microsoft-fight-back/, 2017.
[64] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro, "H.264/AVC Baseline Profile Decoder Complexity Analysis," *CSVT*, 2003.
[65] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Conner, N. Vijaykumar, O. Mutlu, and S. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[66] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[67] "HTTP Archive," http://httparchive.org/.
[68] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A Mobile Benchmark Suite for Architectural Simulators," in *ISPASS*, 2014.
[69] D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, 1952.
[70] D. Hwang, "Native One-Copy Texture Uploads," https://01.org/chromium/2016/native-one-copy-texture-uploads-for-chrome-OS, 2016.
[71] Hybrid Memory Cube Consortium, "HMC Specification 2.0," 2014.
[72] Intel Corp., "Intel Celeron Processor N3060," https://ark.intel.com/products/91832/Intel-Celeron-Processor-N3060-2M-Cache-up-to-2_48-GHz.
[73] Intel Corp., "Software vs. GPU Rasterization in Chromium," https://software.intel.com/en-us/articles/software-vs-gpu-rasterization-in-chromium.
[74] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *VLSIT*, 2012.
[75] JEDEC Solid State Technology Assn., "JESD235: High Bandwidth Memory (HBM) DRAM," 2013.
[76] S. Jennings, "Transparent Memory Compression in Linux," https://events.static.linuxfound.org/sites/events/files/slides/tmc_sjennings_linuxcon2013.pdf, 2013.
[77] E. Kalali and I. Hamzaoglu, "A Low Energy HEVC Sub-Pixel Interpolation Hardware," in *ICIP*, 2014.
[78] J. Kane and Q. Yang, "Compression Speed Enhancements to LZO for Multi-Core Systems," in *SBAC-PAD*, 2012.

[79] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD*, 2012.
[80] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, 2011.
[81] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ISCA*, 2016.
[82] J. S. Kim, D. Senol, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies," *BMC Genomics*, 2018.
[83] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.
[84] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[85] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE CAL*, 2015.
[86] P. M. Kogge, "EXECUBE: A New Architecture for Scaleable MPPs," in *ICPP*, 1994.
[87] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai, "Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices," in *MobiCom*, 2017.
[88] M. J. Langroodi, J. Peters, and S. Shirmohammadi, "Decoder-Complexity-Aware Encoding of Motion Compensation for Multiple Heterogeneous Receivers," *TOMM*, 2015.
[89] C. Lee and Y. Yu, "Design of a Motion Compensation Unit for H.264 Decoder Using 2-Dimensional Circular Register Files," in *ISOCC*, 2008.
[90] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," *ACM TACO*, 2016.
[91] P. Lewis, "Avoiding Unnecessary Paints," https://www.html5rocks.com/en/tutorials/speed/unnecessary-paints/, 2013.
[92] T. Li, C. An, X. Xiao, A. T. Campbell, and X. Zhou, "Real-Time Screen-Camera Communication Behind Any Scene," in *MobiSys*, 2015.
[93] F. Liu, P. Shu, H. Jin, L. Ding, J. Yu, D. Niu, and B. Li, "Gearing Resource-Poor Mobile Devices with Powerful Clouds: Architectures, Challenges, and Applications," *IEEE Wireless Communications*, 2013.
[94] G. H. Loh, "3D-Stacked Memory Architectures for Multi-Core Processors," in *ISCA*, 2008.
[95] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," in *ISCA*, 2000.
[96] Mentor Graphics Corp., "Catapult High-Level Synthesis," https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.
[97] Microsoft Corp., "Skype," https://www.skype.com/.
[98] A. Mirhosseini, A. Agrawal, and J. Torrellas, "Survive: Pointer-Based In-DRAM Incremental Checkpointing for Low-Cost Data Persistence and Rollback-Recovery," *IEEE CAL*, 2017.
[99] N. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot, "Sort vs. Hash Join Revisited for Near-Memory Execution," in *ASBD*, 2007.
[100] B. Moatamed, Arjun, F. Shahmohammadi, R. Ramezani, A. Naeim, and M. Sarrafzadeh, "Low-Cost Indoor Health Monitoring System," in *BSN*, 2016.
[101] A. Mosenia, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, "CABA: Continuous Authentication Based on BioAura," *IEEE TC*, 2017.
[102] A. Mosenia, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, "Wearable Medical Sensor-Based System Design: A Survey," *MSCS*, 2017.
[103] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *MICRO*, 2011.
[104] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO*, 2007.
[105] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "VIP: Virtualizing IP Chains on Handheld Platforms," in *ISCA*, 2015.
[106] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in *HPCA*, 2017.
[107] G. Narancic, P. Judd, D. Wu, I. Atta, M. Elnacouzi, J. Zebchuk, J. Albericio, N. E. Jerger, A. Moshovos, K. Kutulakos, and S. Gadelrab, "Evaluating the Memory System Behavior of Smartphone Workloads," in *SAMOS*, 2014.
[108] Net Applications, "Market Share Statistics for Internet Technologies," https://www.netmarketshare.com/.
[109] A. M. Nia, M. Mozaffari-Kermani, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, "Energy-Efficient Long-term Continuous Personal Health Monitoring," *MSCS*, 2015.
[110] Nielsen Norman Group, "Page Parking: Millennials' Multi-Tab Mania," https://www.nngroup.com/articles/multi-tab-page-parking/.
[111] M. F. X. J. Oberhumer, "LZO Real-Time Data Compression Library," http://www.oberhumer.com/opensource/lzo/, 2018.
[112] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA*, 1998.
[113] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, Energy Characterizations and Architectural Implications of an Emerging Mobile Platform Benchmark Suite – MobileBench," in *IISWC*, 2013.
[114] D. Pandiyan and C.-J. Wu, "Quantifying the Energy Cost of Data Movement for Emerging Smartphone Workloads on Mobile Platforms," in *IISWC*, 2014.
[115] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A Case for Intelligent RAM," *IEEE Micro*, 1997.
[116] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.
[117] B. Popper, "Google Services Monthly Active Users," https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users, 2017.
[118] Qualcomm Technologies, Inc., "Snapdragon 835 Mobile Platform," https://www.qualcomm.com/products/snapdragon/processors/835.
[119] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *ISCA*, 2016.
[120] J. Ren and N. Kehtarnavaz, "Comparison of Power Consumption for Motion Compensation and Deblocking Filters in High Definition Video Coding," in *ISCE*, 2007.
[121] P. V. Rengasamy, H. Zhang, N. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Characterizing Diverse Handheld Apps for Customized Hardware Acceleration," in *IISWC*, 2017.
[122] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM TOS*, 2013.
[123] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen, "Revisiting Network Energy Efficiency of Mobile Apps: Performance in the Wild," in *IMC*, 2015.
[124] F. Ross, "Migrating to LPDDR3: An Overview of LPDDR3 Commands, Operations, and Functions," in *JEDEC LPDDR3 Symposium*, 2012.
[125] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.
[126] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.
[127] V. Seshadri and O. Mutlu, "The Processing Using Memory Paradigm: In-DRAM Bulk Copy, Initialization, Bitwise AND and OR," arXiv:1610.09603 [cs:AR], 2016.
[128] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers, Volume 106*, 2017.
[129] D. E. Shaw, S. J. Stolfo, H. Ibrahim, B. Hillyer, G. Wiederhold, and J. A. Andrews, "The NON-VON Database Machine: A Brief Overview," *IEEE DEB*, 1981.
[130] D. Shingari, A. Arunkumar, and C.-J. Wu, "Characterization and Throttling-Based Mitigation of Memory Interference for Heterogeneous Smartphones," in *IISWC*, 2015.
[131] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015.
[132] R. Smith, "Apple's A9 SoC Is Dual Sourced From Samsung & TSMC," https://www.anandtech.com/show/9665/apples-a9-soc-is-dual-sourced-from-samsung-tsmc, 2015.
[133] J. Stankowski, D. Karwowski, K. Klimaszewski, K. Wegner, O. Stankiewicz, and T. Grajek, "Analysis of the Complexity of the HEVC Motion Estimation," in *IWSSIP*, 2016.
[134] H. S. Stone, "A Logic-in-Memory Computer," *IEEE TC*, 1970.
[135] R. Sukale, "What Are Reflows and Repaints and How to Avoid Them," http://javascript.tutorialhorizon.com/2015/06/06/what-are-reflows-and-repaints-and-how-to-avoid-them/, 2015.
[136] S. Sutardja, "The Future of IC Design Innovation," in *ISSCC*, 2015.
[137] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," in *AAAI*, 2017.
[138] X. Tang, O. Kislal, M. Kandemir, and M. Karakoy, "Data Movement Aware Computation Partitioning," in *MICRO*, 2017.
[139] TechInsights, "Samsung Galaxy S6," http://www.techinsights.com/about-techinsights/overview/blog/inside-the-samsung-galaxy-s6/.
[140] R. Thompson, "Improve Rendering Performance with Dev Tools," ttps://inviqa.com/blog/improve-rendering-performance-dev-tools, 2014.
[141] G. Toderici, D. Vincent, N. Johnston, S. J. Hwang, D. Minnen, J. Shor, and M. Covell, "Full Resolution Image Compression with Recurrent Neural Networks," in *CVPR*, 2017.
[142] Twitter, Inc., "Twitter," https://www.twitter.com/.
[143] E. Vasilakis, "An Instruction Level Energy Characterization of ARM Processors," Foundation of Research and Technology Hellas, Inst. of Computer Science, Tech. Rep. FORTH-ICS/TR-450, 2015.
[144] WebM Project, "Hardware: SoCs Supporting VP8/VP9," http://wiki.webmproject.org/hardware/socs.
[145] WebM Project, "WebM Repositories — libvpx: VP8/VP9 Codec SDK," https://www.webmproject.org/code/.
[146] WebM Project, "WebM Video Hardware RTLs," https://www.webmproject.org/hardware/.

[147] S. Wegner, A. Cowsky, C. Davis, D. James, D. Yang, R. Fontaine, and J. Morrison, "Apple iPhone 7 Teardown," http://www.techinsights.com/about-techinsights/overview/blog/apple-iphone-7-teardown/, 2016.
[148] A. Wei, "Qualcomm Snapdragon 835 First to 10 nm," http://www.techinsights.com/about-techinsights/overview/blog/qualcomm-snapdragon-835-first-to-10-nm/, 2017.
[149] WordPress Foundation, "WordPress," https://www.wordpress.com/.
[150] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the Wall: Near-Data Processing for Databases," in *DaMoN*, 2015.
[151] C. Xie, S. L. Song, J. Wang, W. Zhang, and X. Fu, "Processing-in-Memory Enabled Graphics Processors for 3D Rendering," in *HPCA*, 2017.
[152] Xiph.Org Foundation, "Derf Video Test Collection," https://media.xiph.org/video/derf/.
[153] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.
[154] H. Zhang, P. V. Rengasamy, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Race-To-Sleep + Content Caching + Display Caching: A Recipe for Energy-eficient Video Streaming on Handhelds," in *MICRO*, 2017.
[155] H. Zhang, P. V. Rengasamy, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Race-to-Sleep + Content Caching + Display Caching: A Recipe for Energy-Efficient Video Streaming on Handhelds," in *MICRO*, 2017.
[156] X. Zhang, J. Li, H. Wang, D. Xiong, J. Qu, H. Shin, J. P. Kim, and T. Zhang, "Realizing Transparent OS/Apps Compression in Mobile Devices at Zero Latency Overhead," *IEEE TC*, 2017.
[157] S. Zhu and K.-K. Ma, "A New Diamond Search Algorithm for Fast Block Matching Motion Estimation," in *ICICS*, 1997.
[158] Y. Zhu and V. J. Reddi, "WebCore: Architectural Support for Mobile Web Browsing," in *ISCA*, 2014.
[159] Y. Zhu and V. J. Reddi, "GreenWeb: Language Extensions for Energy-Efficient Mobile Web Computing," in *PLDI*, 2016.
[160] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *TIT*, 1977.