In-DRAM Near-Data Approximate Acceleration for GPUs

Amir Yazdanbakhsh Choungki Song[†] Jacob Sacks Pejman Lotfi-Kamran[‡] Hadi Esmaeilzadeh[§] Nam Sung Kim[♭]

Alternative **C**omputing **T**echnologies (**ACT**) Lab

Georgia Institute of Technology [†]University of Wisconsin-Madison [‡]Institute for Research in Fundamental Sciences (IPM)
[§]University of California, San Diego [♭]University of Illinois at Urbana-Champaign

a.yazdanbakhsh@gatech.edu csong38@wisc.edu jsacks@gatech.edu plotfi@ipm.ir hadi@eng.ucsd.edu nskim@illinois.edu

ABSTRACT

GPUs are bottlenecked by the off-chip communication bandwidth and its energy cost; hence near-data acceleration is particularly attractive for GPUs. Integrating the accelerators within DRAM can mitigate these bottlenecks and additionally expose them to the higher internal bandwidth of DRAM. However, such an integration is challenging, as it requires low-overhead accelerators while supporting a diverse set of applications. To enable the integration, this work leverages the approximability of GPU applications and utilizes the neural transformation, which converts diverse regions of code mainly to Multiply-Accumulate (MAC). Furthermore, to preserve the SIMT execution model of GPUs, we also propose a novel approximate MAC unit with a significantly smaller area overhead. As such, this work introduces AXRAM—a novel DRAM architecture—that integrates several approximate MAC units. AXRAM offers this integration without increasing the memory column pitch or modifying the internal architecture of the DRAM banks. Our results with 10 GPGPU benchmarks show that, on average, AXRAM provides 2.6× speedup and 13.3× energy reduction over a baseline GPU with no acceleration. These benefits are achieved while reducing the overall DRAM system power by 26% with an area cost of merely 2.1%.

1 INTRODUCTION

GPUs are one of the leading computing platforms for a diverse range of applications.However, this processing capability is hindered by the *bandwidth wall* [75, 84, 90]. Yet, offering higher bandwidth with either conventional DRAM or HBM is challenging due to package pin and/or power constraints. Such a limitation makes near-data acceleration alluring for GPUs. There are two main options for such an integration: (1) 3D/2.5D stacking [24, 37, 78] and (2) integration within DRAM. The former option may incur a significant cost to expose higher internal bandwidth to 3D/2.5D-stacked accelerators than the external bandwidth exposes to the GPU [6], as the TSVs for standard HBM already consume nearly 20% of each 3D-stacked layer [50].

PACT '18, November 1–4, 2018, Limassol, Cyprus
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5986-3/18/11...$15.00
https://doi.org/10.1145/3243176.3243188

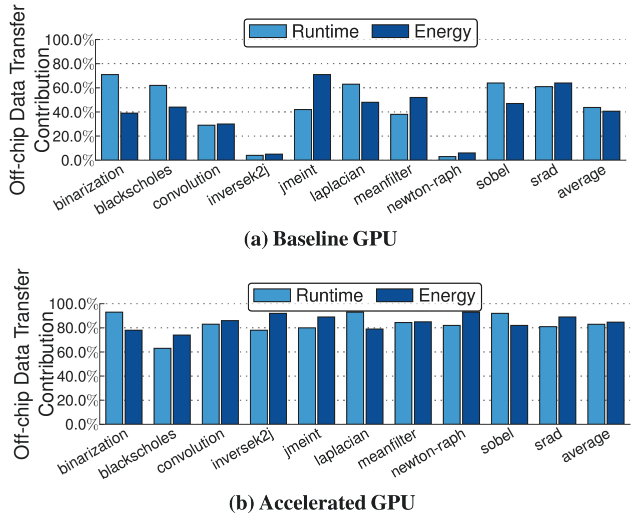


Figure 1: The fraction of total application runtime and energy spent in off-chip data transfer for (a) a baseline GPU and (b) an accelerated GPU [89].

The latter option exposes the accelerators to the higher internal bandwidth of DRAM. Such a tight integration can be attractive if it incurs *little overhead* while enables the acceleration of a *diverse range of applications*. However, integrating many complex accelerators within DRAM is not practical, since DRAM is under tight area, power, and thermal constraints [41, 53–56, 58]. Moreover, even the number of metal layers for routing is limited [49, 66, 67], which severely hinders integrating complex accelerators. Finally, it is highly desirable to avoid changing the innards of DRAM banks, as they have been optimized over decades of engineering.

This work tackles these challenges by exploiting the approximability of many GPU applications. We leverage the neural transformation [5, 22, 29, 61, 89], which can accelerate diverse applications by approximating regions of GPU code and converting them into a neural representation comprised of only *two* types of operations: Multiply-and-Accumulate (MAC) and Look-Up Table (LUT) accesses for calculating the nonlinear function. Hence, the accelerator architecture becomes relatively simple. To further minimize the power and area overhead and enable a low-overhead integration of many in-DRAM accelerators, we further approximate the MAC units. Specifically, these approximate MAC units convert the multiplication into *limited* iterations of shift-add and LUT access operations with early termination by exploiting a unique property of neural transformation, i.e., one of the operands for each MAC operation is fixed. While the accelerators merely comprise simple shift, add, and LUT access operations, they are able to support a wide variety of applications. We attach these simplified units to the wide data lines, which connect the DRAM banks to the global I/O, to avoid altering the banks and memory column pitch. Note that our approach, which significantly simplifies the accelerator design, has merits even when accelerators are placed on logic layers of 3D/2.5D-stacked DRAM. Specifically, package-level power/thermal constraints get more stringent with more stacked-DRAM dies while processors powerful enough to fully exploit high-internal bandwidth will consume high power. Also, the challenges of tying DRAM design to accelerators that only cover few applications may be limiting for DRAM manufacturers. AXRAM tackles this dilemma by introducing a significantly simple and power-efficient design while supporting diverse applications as well as neural networks that are being adopted in various domains. As such, this work defines AXRAM, a novel accelerated DRAM architecture with the following contributions.

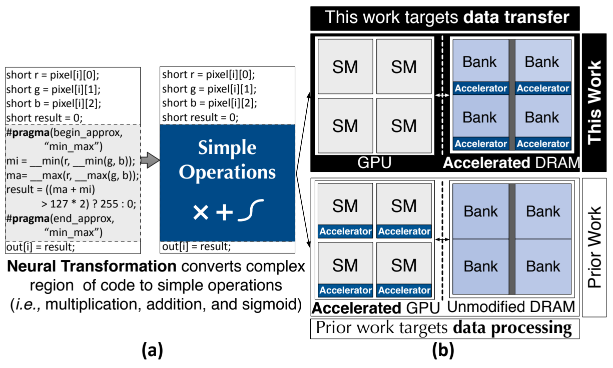
2 OVERVIEW

In this section, we first overview the challenges and opportunities of in-DRAM acceleration for GPUs and how approximation plays an enabling role.

2.1 Challenges and Opportunities

Opportunity to reduce data transfer cost. Off-chip data transfer imposes a significant energy cost relative to data processing. With a 45 nm process, a 32-bit floating-point addition costs about 0.9 pJ, while a 32-bit DRAM memory access costs about 640 pJ [33, 36]. As such, off-chip data transfer consumes over 700× more energy than on-chip data processing. This cost becomes even more pronounced in GPU applications, since they typically stream data and exhibit low temporal locality (*i.e.,* high cache miss rates) [7, 14, 43, 71, 76, 82]. Near-data processing provides an opportunity to cut down this cost. To concretely examine the potential benefits of near-data processing, we conducted a study which teases apart the fraction of runtime and energy consumption spent on off-chip data transfer[1]. As Figure 1a illustrates, on average, applications spend 42% of their runtime and 39% of their energy dissipation on off-chip data transfer on a GPU. In Figure 1b, we further examine this trend with a neurally accelerated GPU (NGPU [89]), to speed up the data processing portion of each thread. The acceleration reduces the data processing time of each thread, in turn increasing the rate of accesses to off-chip memory. This increased rate exacerbates the contribution of data transfer to the application runtime and energy. Moreover, accelerating the GPU further compounds the already significant pressure on the off-chip communication bandwidth [84, 89, 90]. On average, applications spend 83% (85%) of their runtime (energy) on off-chip data transfer on neurally accelerated GPU (NGPU). These results indicate a significant opportunity for near-data processing to address the overhead of off-chip data transfer in GPUs.

Challenges of near-data processing on GPUs. GPUs present unique challenges for near-data processing, as they comprise many cores simultaneously running many threads. To preserve the SIMT execution model of GPUs, we need to integrate many accelerator units near the data. There are two options for where to integrate the accelerator units: (1) the on-chip memory controller or (2) inside the DRAM itself. Option (1) provides the accelerator units with no additional bandwidth, as the on-chip memory controller receives the same bandwidth from memory as the rest of the GPU. Furthermore, placing the accelerator units in the memory controller only circumvents data transfer through the on-chip caches. In addition, integration within the memory controller requires large buffers for holding the accelerators' data, which would impose a significant area overhead. Option (2), which integrates the accelerators in DRAM, reduces the data transfer distance and exploits the high internal bandwidth of the memory. Moreover, integrating the accelerators in DRAM enables us to utilize DRAM as buffers for the accelerators' data. However, this design point can introduce a substantial area and power overhead to the space-limited and power-constrained DRAM. In this work, we integrate the accelerator units in DRAM and leverage the approximability of many GPU applications to significantly simplify the accelerator architecture. These simplifications enable the accelerator to minimize changes to the underlying DRAM architecture and overhead to the DRAM power consumption.



```
short r = pixel[i][0];
short g = pixel[i][1];
short b = pixel[i][2];
short result = 0;
#pragma(begin_approx,
        "min_max")
mi = __min(r, __min(g, b));
ma= __max(r, __max(g, b));
result = ((ma + mi)
        > 127 * 2) ? 255 : 0;
#pragma(end_approx,
        "min_max")
out[i] = result;
```

```
short r = pixel[i][0];
short g = pixel[i][1];
short b = pixel[i][2];
short result = 0;
out[i] = result;
```

Simple Operations
× + ∫
Neural Transformation converts complex region of code to simple operations (*i.e.,* multiplication, addition, and sigmoid)
This work targets **data transfer**
SM
Bank
Accelerator
GPU
Accelerated DRAM
This Work
Accelerated GPU
Unmodified DRAM
Prior Work
Prior work targets **data processing**
(a)
(b)

Figure 2: (a) Neural transformation of a code segment from the binarization **benchmark. (b) Comparison of prior work (bottom diagram) [89] and this work (top diagram).**

2.2 Approximation for Near-Data Processing

Among approximation techniques, the neural transformation [22] is an attractive, yet unexplored, approach for near-data processing in DRAM. The neural transformation converts a code segment into a neural representation comprising only two operations: multiply-and-accumulate (MAC) and sigmoid. Reducing computation to two operations provides an opportunity to significantly simplify the accelerator. This simplified design minimizes changes to the DRAM and can be replicated many times to preserve the GPUs' SIMT execution model.

The neural transformation trains a neural network to replace an approximable region of conventional code [5, 22, 29, 89]. Figure 2a illustrates the transformation of a code segment, where the approximable region is highlighted in gray. An approximable region is a segment that, if approximated, will not lead to any catastrophic failures (*e.g.,* segmentation

[1]Section 8 presents our experimental methodology and settings.

fault). Its approximation will only gracefully degrade of the application output quality. As is customary in approximate computing [11, 69, 77, 88], the programmer *only* annotates the code region(s) that can be safely approximated. The compiler then automatically performs the transformation and replaces the code segment with a neural hardware invocation [89]. As shown in Figure 2b, prior work addresses data processing by integrating neural accelerators within the GPU cores and defines a neurally accelerated architecture for GPUs (NGPU) [89]. This work, on the other hand, develops a neurally accelerated architecture for DRAM, dubbed AxRam, which addresses *off-chip data transfer*. Moving the neural acceleration to DRAM enables AxRam to reduce the data transfer overhead and supply more bandwidth to the accelerators. Moreover, we leverage the approximability of the GPU applications to further simplify the architecture of the accelerator units (Section 5).

3 AxRam EXECUTION FLOW AND ISA

This section discusses the execution flow and instruction set architecture (ISA) extensions which enable the seamless integration of AxRam with the GPU's SIMT execution model. Unlike prior work [5, 22, 29, 61, 89], AxRam is disjoint from the processor core and is instead integrated into DRAM. Hence, the ISA extensions must enable the on-chip memory controller to configure and initiate the in-DRAM accelerator.

3.1 Neural Acceleration of GPU Warps

GPU applications consist of one or more kernels, which are executed by each of the GPU threads. Threads are executed on GPU processing cores called streaming multiprocessors (SMs), which divide the threads into small groups called warps. A warp executes the same instruction of the same kernel in lock-step but with different input data. The neural transformation approximates segments of the GPU kernels and replaces the original instructions of these segments with the computation of a neural network, as shown in Figure 3. A neurally accelerated warp computes the same neural network, one neuron at a time, across all the threads for different inputs. Due to the neural transformation, this computation only consists of MAC and lookup (sigmoid) operations. Specifically, the output y of each neuron is given by $y = sigmoid(\Sigma_i w_i \times in_i)$, where in_i is the input to the neuron and w_i is the weight of the connection. The neural computation portion of the threads are offloaded to the in-DRAM neural accelerator. Instructions which invoke and configure the in-DRAM neural accelerator are added to the GPU's ISA (Section 3.3). These instructions are added by the compiler to the accelerated kernel and are executed by the threads in SIMT mode like other GPU instructions. Thus, the accelerated warp comprises both the normal precise instructions of the unmodified code segments and approximate instructions which communicate with the in-DRAM accelerator. Before explaining these ISA extensions, we provide a high level picture of the execution flow of AxRam.

3.2 Execution Flow with AxRam

Figure 3 illustrates the execution flow of the neurally accelerated warp and communication amongst the GPU, on-chip memory controller, and in-DRAM neural accelerator in one GDDR5 chip. We assume that all data for the neural computation of a given warp is located on one GDDR5 chip. This assumption is enabled by a series of data organization optimizations discussed in Section 6. First, the SM fetches the warp and begins the execution of the precise instructions normally without any in-DRAM acceleration. The warp then reaches the approximable region, which instructs the SM to send an initiation request directly to the on-chip memory controller. Once the initiation request has been sent, the issuing warp goes into halting mode. This is *not* an active warp waiting mechanism but is similar to a load miss in the cache. The core may switch to the execution of another warp while the in-DRAM neural computation proceeds, provided the warp does not have any conflicts with the ongoing in-DRAM computation.

Augmented logic in the on-chip memory controller first sends invalidate signals to the on-chip caches and nullifies dirty data to be modified by the neural computation. The invalidate signals are sufficient to prevent GPU cores from using stale data. As most GPU caches use a write-through policy [80], it is guaranteed that in-DRAM accelerators have access to the most up-to-date data. We explain the cache coherency mechanism of AxRam in Section 7. Then, the on-chip memory controller configures and initiates the in-DRAM accelerators (Figure 3). Specifically, the on-chip memory controller translates the initiation request and instructs the in-DRAM accelerator where the inputs to the neural network are located in memory and to where the accelerator should store its final outputs. Furthermore, the on-chip memory controller blocks any other memory commands to that particular DRAM chip to ensure the atomicity of the in-DRAM neural computation. The on-chip memory controller also does not assign any other neural computations to a GDDR5 chip with an ongoing neural computation. We added a simple on-chip queue per memory controller to keep track of in-flight requests for in-DRAM approximate acceleration. The area overhead of these queues to the GPU die is modest ($\approx$1%). Similar to [24], the on-chip memory controller allows critical memory operations such as *refreshing* to be performed during in-DRAM neural computation.

During neural computation, the in-DRAM accelerator takes full control of accessing and issuing commands to the banks. The in-DRAM accelerator performs the MAC and sigmoid operations (Figure 3). Neural computation for the threads of the neurally accelerated warp is performed in lock-step by the many integrated arithmetic units. Once neural computation is completed, the in-DRAM accelerator writes its results back to the banks in locations dictated by the memory controller. We consider two options for notifying GPU that in-DRAM computation has completed: waiting a fixed number of cycles and polling. The former approach requires pre-determining the execution time of each invocation and exposing that to the compiler. The memory controller would

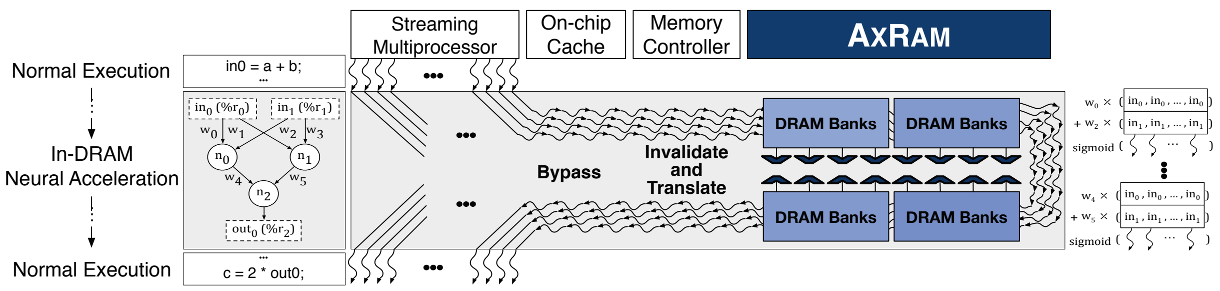


Figure 3: Execution flow of the accelerated GPU code on the in-DRAM accelerator.

then wait for this pre-determined number of cycles before notifying the warp to continue precise execution. However, the execution time of an in-DRAM invocation depends on the neural network topology and the accelerator's DRAM accesses patterns. Anticipating the DRAM's accesses patterns necessitates exposing DRAM microarchitectural parameters to the compile. These details are not always readily available, making this design point less desirable. Instead, we choose the polling approach, in which the accelerator sets the DRAM memory-mapped mode register MR0 [39], similar to [24]. The on-chip memory controller periodically polls this register to determine if the computation has finished. Once it detects that the register has been set, the on-chip memory controller notifies the GPU that the neural computation for the specific warp is finished and the warp can continue precise execution. To enable the controller to properly initiate and configure the in-DRAM accelerator, we need to extend the ISA with instructions that communicate the configuration data.

3.3 ISA Extensions for AxRAM

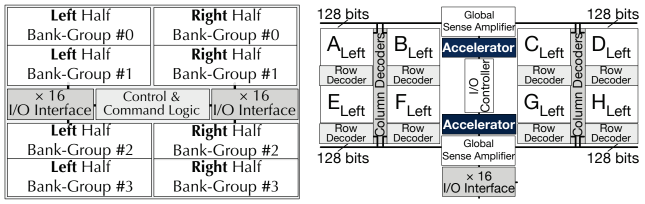
We augment the ISA with three instructionswhich bypass the on-chip caches and communicate directly with the memory controller. The proposed ISA extensions are as follows:

(1) **`config.axram`** `[%start_addr], [%end_addr]` reads the preloaded neural network configuration from the memory region `[%start_addr]` to `[%end_addr]` and sends it to the in-DRAM accelerator. The configuration includes both the weight values and the topology of the neural network.

(2) **`initiate.axram`** `[%start_addr], [%end_addr]` sends the start (`[%start_addr]`) and end (`[%end_addr]`) addresses of a continuous memory region which constitutes the neural network inputs for the warp and then initiates the in-DRAM accelerator.

(3) **`wrt_res.axram`** `[%start_addr], [%end_addr]` informs the in-DRAM accelerator to store the computed value(s) of the neural computation in a continuous memory region defined by the start (`[%start_addr]`) and end (`[%end_addr]`) addresses.

The dimensionality of the different neural network layers is statically identified at compile time and used to configure the in-DRAM accelerator. Thus, the in-DRAM accelerator knows how many neurons to expect per layer, and specifying sufficient memory regions to ensure proper execution. However, this means that input order is important and necessitates a series of data organization optimizations to ensure correct



(a) High-level GDDR5 DRAM **(b) Two Half Bank-Groups**

Figure 4: (a) High-Level GDDR5 DRAM organization. (b) Layout of two half bank-groups (Left Half Bank-Group #0 and Left Half Bank-Group #1) and the accelerators. The black-shaded boxes show the placement of the accelerators.

execution (see Section 6). As with other GPU instructions, these ISA extensions are executed in SIMT mode. That is, each thread in a warp will communicate its input/output data regions to the in-DRAM neural accelerator. Additionally, the weights and the topology of each neural network are embedded by the compiler in the ".data" section of the ELF-formatted CUDA binary code (cubin) [2] during compilation. Along with the CUDA binary code, the weight values and the topology of the trained neural network are copied in a preallocated memory region. Using the **`config.axram`** instruction, the in-DRAM accelerator pre-loads these weights and topology configuration of the trained neural network from memory before starting the neural computation.

4 AxRAM MICROARCHITECTURE

To describe our design, we use a GDDR5 DRAM architecture [48, 66, 67]. Since HBM generally stacks GDDR5-like DRAM [42], our modifications can potentially be extended to such memory architectures. Furthermore, AxRAM is appropriate for these 3D-stacked structures, because, as our evaluations show (see Section 8), our design does not increase the DRAM power consumption due to data transfer. Our main design objectives are to (1) preserve the SIMT execution model while (2) keeping the modifications to the baseline GDDR5 minimal and (3) leveraging the high internal bandwidth of DRAM. AxRAM achieves these goals by integrating many simple arithmetic and sigmoid units into GDDR5. To describe the microarchitecture of AxRAM, we first present an overview of the GDDR5 architecture.

4.1 Background: GDDR5 Architecture

While GDDR5 has a I/O bus width of 32 bits per chip, it has a much higher internal bus width of 256 bits per bank.

This provides an 8× higher bitwidth that would significantly benefit GPUs, which already place significant pressure on the off-chip bandwidth [46, 84, 90]. Furthermore, the bank-group organization of GDDR5 provides intrinsic parallelism which can be leveraged to feed data to a large number of arithmetic units. By exploiting the attribute of the bank-group organization, we can further utilize 1024 bits of internal bus width (32× higher bitwidth than the I/O bus).

Figure 4a shows the GDDR5 DRAM architecture, which consists of four bank-groups, each with four banks. Each bank-group can operate independently, meaning requests to different bank-groups can be interleaved. The bank-groups are organized into upper and lower pairs partitioned by the I/O interface and control and command logic. Moreover, each bank-group contains four banks, which are subdivided into two half-banks. Subdividing the banks splits each bank-group into a left and right half, each with four half-banks. Two upper-left half bank-groups (*i.e.*, Left Half Bank-Group #0 and Left Half Bank-Group #1) are depicted in Figure 4b. In each half bank-group, the four half-banks are split into pairs (*e.g.*, A_{Left} and B_{Left} vs. C_{Left} and D_{Left}) by a global sense amplifier and shared I/O controller. Each half-bank has its own row decoder, while column decoders are shared between the half-bank pairs of the two adjacent half bank-groups. Both the right and left half bank-groups provide a bus width of 128 bits for a total of 256 bits. However, this higher internal bus width is serialized out through the right and left 16-bit I/O interface.

For instance, when the DRAM receives a memory command to access Bank A in Bank-Group #0, both the half-banks, A_{Left} and A_{Right}, process the command in unison to supply the data. For the sake of simplicity, we focus on the left half of Bank-Group #0 shown in Figure 4b. The global row decoder of the half-bank decodes the address and accesses the data. The shared column decoder asserts the column select lines, which drives the data onto a 128-bit global dataline shared between half-banks A_{Left} and B_{Left}. Since the global dataline is shared between the pairs of half-banks, only one may send or receive data at a time. The global sense amplifier then latches the data from the global dataline and drives the data on the bank-group global I/O through the I/O controller. The right and left I/O interfaces then serialize the 256-bit (128-bit each) data on the Bank-Group #0's global I/O before sending them through the 32-bit data I/O pins. By placing the accelerators inside the DRAM, we aim to exploit the higher internal bandwidth instead of relying on the lower bandwidth of the data I/O pins.

We next discuss how AXRAM integrates the accelerators into the GDDR5. Additionally, we describe how the accelerator uses the aforementioned GDDR5 attributes to preserve the SIMT execution model and minimize changes while providing data to all the arithmetic units each cycle.

4.2 In-DRAM Accelerator Integration

To minimize DRAM changes yet benefit from its high internal bandwidth, AXRAM integrates a set of arithmetic and sigmoid units within each half-bank group (Figure 5). These arithmetic and sigmoid units are connected to the half-bank

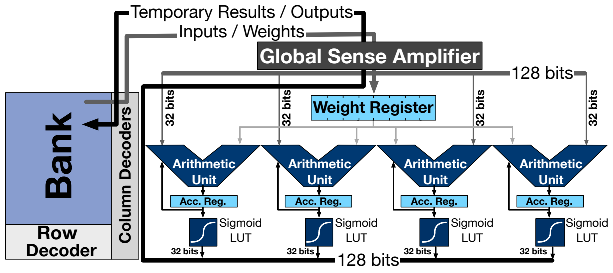


Figure 5: Integration of weight register, arithmetic units, accumulation registers, and sigmoid LUTs.

groups' global sense amplifiers. Below we discuss the design choices, structure, and components of this integration.

Accelerator architecture. As mentioned, the accelerator is a set of arithmetic and sigmoid units. Each pair of arithmetic and sigmoid units is assigned to a thread of the neurally accelerated warp. The sigmoid units are implemented as a read-only LUT synthesized as *combinational logic* to minimize the area overhead. We will further simplify the arithmetic units in Section 5. Here, we discuss how we guarantee SIMT execution of the neurally accelerated warp with these units. Each arithmetic unit can execute one MAC operation each clock cycle with a *32-bit input* and *32-bit weight*. The banks need to be able to feed a 32-bit input to each of the integrated arithmetic units at the same time, such that the arithmetic units can perform the neural computation for all the threads within a warp simultaneously. As mentioned before, each bank-group in the baseline GDDR5 has a 256-bit wide global I/O, 128-bit per each half bank-group. Since each bank group can function independently, the DRAM can provide a total of 1024 bits (32×32 bits) of data from the banks at a time. Thus, we integrate 32 pairs of arithmetic and sigmoid units in each GDDR5 chip, 8 pairs per each bank-group. In Section 6, we describe a data organization which enables us to read and write 1024 bits of data simultaneously.

Unit placement. There are multiple design points for integrating arithmetic units within a GDDR5 chip. To minimize changes to the DRAM architecture, we aim to avoid modifying the underlying mat[2] and bank design. One option is to add arithmetic units close to each half-bank to utilize their high internal bandwidth. However, this would require cutting the global datalines shared between pairs of half-banks (Figure 4b) and adding a separate sense amplifier per half-bank. Therefore, this design point imposes a large area overhead and necessitates significant changes to each GDDR5 chip. Another option is to add arithmetic units in a central manner close to the I/O interface in Figure 4a. Although this option does not suffer from the drawbacks of placing the accelerators close to each half-bank, it requires extensive routing. Because the aforementioned options require such extensive modifications, they are infeasible design points. Instead, AXRAM adds four arithmetic units per half bank-group after the shared sense amplifier within the I/O controller boundary, for a total of eight arithmetic units per bank-group. The accelerators'

[2] A mat constitutes an array of 512×512 DRAM cells. Each mat comes with its own row decoder, datalines, and sense amplifiers.

placement is illustrated in Figure 4b, while the specific accelerator logic layout, including the arithmetic units, is shown in Figure 5. This design choice imposes minimal changes to the DRAM architecture and avoids altering the design of the mats or banks.

Design optimizations. Each of the arithmetic units implements the neural network MAC operations. However, to properly supply and retrieve data from the arithmetic units, we need storage for the (1) inputs, (2) weights, (3) temporary results, and (4) outputs of the network. Generally, neural accelerators use dedicated buffers as storage [22, 89]. However, placing the arithmetic units near the data allows AXRAM to perform a series of design optimizations which minimize the modifications to the baseline GDDR5. As Figure 5 shows, AXRAM is able to instead use the GDDR5 banks as buffers. Input data is read directly from the GDDR5 banks and fed to the arithmetic units for processing. AXRAM leverages the large number of sense amplifiers within the DRAM banks to store temporary results in pre-allocated memory regions during in-DRAM computation. Outputs from the arithmetic units are written directly back to the GDDR5 banks. By not using dedicated buffers, we avoid adding large registers to each GDDR5 chip and reduce the area overhead. We only add dedicated weight registers to supply weights to all the arithmetic units. This enables AXRAM to avoid having to read the weights from the memory banks each cycle and instead utilize the internal buses to supply all the arithmetic units with inputs. Thus, we can simultaneously provide each arithmetic unit with an input and weight each cycle.

Weight register. Since all threads within a warp perform the computation of the same neuron in lock-step, the weights are the *same* among all the threads for a given neural network. Therefore, AXRAM can use *one weight* at a time and share it among the arithmetic units within a half-bank group. We add a weight register (shown in Figure 5) per half bank-group, or for each group of four arithmetic units. As shown in Figure 5, the weights are pre-loaded into the weight register before the computation starts. If the number of weights exceeds the capacity of the register, the next set of weights are loaded after the first set has been depleted. This weight register has 8×32-bit entries per each half bank-group. Since each half bank-group can provide 128 bits of data at a time, the weight register should have at least four entries to fully utilize the provided bandwidth. We increase the number of weight register entries to allow computation to move forward while the next set of weights are loaded and avoid unnecessary stalls.

GDDR5 timing constraints. Adding arithmetic units to the half bank-groups increases the load to the half bank-groups' global I/Os.The only timing constraint affected by the increased load is the column access latency (`tCL`). To estimate the timing impact of `tCL` by `HSPICE` simulation, we measure the increase in load due to the accelerator on the GIOs after the placement and routing. Based on our evaluation, the extra loading on the half bank-groups' global I/Os increases the `tCL` by ≈ 20 `ps`. This increase is 0.16% of the typical value for `tCL`, which is around 12.5 `ns` to 15 `ns` [39, 67], and is less than the guardband which accounts for various design variations [40]. Thus, the 20 `ps` increase has virtually no effect on the timing of GDDR5.

Connection between DRAM banks and arithmetic units. The internal half bank-groups' global I/Os need to support two different modes: (1) normal mode and (2) in-DRAM acceleration mode. When the accelerator performs the computation, the half bank-group's global I/Os are connected to the arithmetic units to transmit input data. Once the computation of a neuron completes, the arithmetic unit inputs arithmetic units are disconnected from the half bank-group's global I/Os. The arithmetic units outputs are then connected to the global datalines through the global I/Os for storing the computed data into the memory banks. We use a series of pass transistors to control the connection between the inputs and outputs of the arithmetic units and the GDDR5 half bank-groups. Supporting a direct connection between the arithmetic units and the GDDR5 banks also requires additional routing paths in the DRAM. To enable the in-DRAM accelerator to gain access of the GDDR5 chip, we also modify the internal address/command bus. In normal mode, the on-chip memory controller has the full access of the address/command bus. However, in in-DRAM acceleration mode, the accelerator gains access to the address/command bus. A set of pass transistors supports this functionality in memory as well. We evaluate the overhead of pass transistors and routing paths in Section 8. To orchestrate the flow of data in the banks to and from the in-DRAM accelerator, we add an in-DRAM controller. Furthermore, we augment the on-chip memory controller with additional logic to translate the ISA extensions and properly initiate and configure the in-DRAM accelerator.

4.3 Interfacing the GPU with AXRAM

Memory controller. We extend the on-chip memory controllers to send invalidation signals to the on-chip caches upon receiving AXRAM instructions. Moreover, we extend the on-chip memory controller to translate the AXRAM instructions (Section 3) to a sequence of special memory instructions. These memory instructions (1) configure the in-DRAM accelerator and (2) initiate the in-DRAM neural computation. The on-chip memory controller is augmented with customized address mapping logic to perform this translation. Upon receiving AXRAM instructions, the implemented address mapping logic inside each on-chip memory controller sends a series of special memory commands to the in-DRAM accelerator to configure and initiate the in-DRAM acceleration. We also add a one-bit flag inside each memory controller to keep track of the status of its corresponding GDDR5 chip. During in-DRAM neural computation, the flag is set so that the memory controller knows not to issue any further memory commands to the memory chip.

However, the memory controller may regain the ownership of the memory chip for performing mandatory memory operations such as refreshing [62]. Similar to prior work [24], the memory controller sends a `suspend` command to the in-DRAM controller if the GDDR5 chip is in neural computation mode. Upon receiving the `suspend` command, the in-DRAM control unit stores any temporary results in the

DRAM and stops computation. Once the refresh period finishes, the memory controller instructs the in-DRAM controller to continue the suspended neural computation.

In-DRAM controller. Previous work [32] has proposed integrating an on-DIMM controller and a handful of specialized microcontrollers in memory to accelerator associative computing. However, since the neural network does not require a complicated controller [22, 89], we instead add a simple control unit inside each GDDR5 chip. This in-DRAM controller (1) marshals data and weights between memory banks and the in-DRAM accelerator and (2) governs the sequence of neural network operations. Specifically, it fetches input data from the banks and sends them to the arithmetic units, reads weights from memory and loads them into the weight buffers, and stores temporary results and neural output(s) into the banks. When the in-DRAM controller receives instructions from the on-chip memory controller, it gains full control of the internal DRAM buses. As discussed, the memory controller only re-gains ownership of the internal DRAM buses when neural computation completes and for performing mandatory memory operations such as random refreshing.

5 ARITHMETIC UNITS SIMPLIFICATION

There exist two options for the arithmetic units. The first option is to use floating-point arithmetic units to perform the neural computation. Another option is to use fixed-point arithmetic units for energy gains and a smaller area overhead. We propose a third option to approximate the arithmetic units to further reduce the area overhead and keep the impact on the overall DRAM system power low.

These simplified arithmetic units break down the MAC operations into iterations of add and shift operations. More iterations of this shift-add unit offers higher precision at the expense of the throughput of the unit. Since the weights W_i remain constant after training a neural network, the shift amounts can be pre-determined based off the bit indices of ones within the 32-bit weight value, starting with the most significant one. Figure 6a shows an implementation of this simplified shift-add unit. X_i represents the input of a neuron and W_{ij} is the shift amount for the ith weight in its jth iteration. The weight register stores these predetermined shift amounts. Since the shift amounts are indices of bits within a 32-bit weight value, the maximum shift amount is 32, which can be represented by a 5 bit value. Thus, each 32-bit entry in the weight register can hold a total of five shift amounts.

Figure 6 shows the design using an example in which $W_i = 01011010_2 90_{10}$ and $X_i = 01111101_2\ 125_{10}$. Multiple iterations of the simplified shift-add unit execution are shown in Figure 6b and 6c. The W_{ij} shift amount can be pre-determined by obtaining the bit index of the j_{th} leading one of W_i. In this example, the most significant one in W_i is in the *sixth* bit position, meaning X_i is shifted by $W_{00} = 6_{10} = 110_2$. The result is then accumulated to the sum, which is initialized to zero. The *first* iteration (Figure 6b) yields 8000_{10}, which achieves 71% accuracy to the actual sum 11250_{10}. More iterations leads to higher accuracy at the cost of higher energy consumption. The *second* (Figure 6c), *third*, and *fourth* iterations achieve

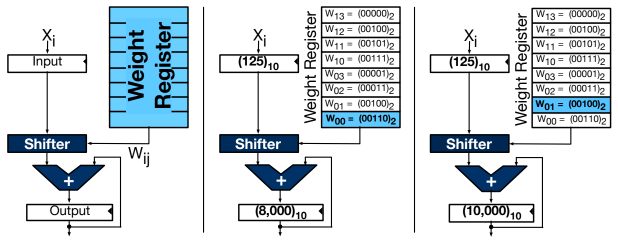


Figure 6: (a) Example of the simplified shift-add unit with pre-loaded shift amounts. (b-c) Two iterations of the shift-add unit.

89%, 98%, and 100% (*e.g.* zero accuracy loss) accuracy, respectively. We evaluate the trade-offs between different arithmetic units for in-DRAM neural acceleration in Section 8.

6 DATA ORGANIZATION FOR AXRAM

The AXRAM architecture (Section 4) leverages bank-group level parallelism to supply all arithmetic units with inputs simultaneously. For this design to comply with the SIMT execution model, we require data to be laid out in a specific order on a single GDDR5 chip. Recent work [4, 35, 38] has shown the benefits of data organization in improving the efficiency of near-data processing for certain applications. A neural network execution has consistent and predictable memory access patterns [15, 27]. Similar to recent work [38], we leverage the predictability of the memory access patterns in neural network execution to perform a series of data organization optimizations to fully utilize the inherent *bank-group* and *bank-level* memory parallelism in memory. Since the weights of the network are shared amongst all the threads and loaded into the weight register before in-DRAM neural computation, we only need to ensure that the input data is properly placed in memory.

Data partitioning. We logically divide a warp into *four* partitions, each with *eight* threads. The data for all the eight threads of each partition is allocated within each bank-group. That is, the data for the first partition of threads (*e.g.* thread$_{0-7}$) is allocated to the first bank-group. Similarly, the data for *thread*$_{8-15}$, *thread*$_{16-23}$, *thread*$_{24-31}$ is allocated to the *second*, *third*, and *fourth* bank-group, respectively. If there is shared data between warps, we replicate it during the data partitioning. On average, the overhead of duplicated data is ≈2% in terms of storage.

Data shuffling. Within a partition, the data has to be organized in such a way that we can read and write all the data for the 32 arithmetic units at a time and efficiently utilize the bank-level parallelism. Specifically, AXRAM requires two constraints to be met for the data layout: (1) the row and column addresses of a given neuron's inputs for all the 32 threads have to be the same across the bank-groups and (2) the addresses of a neuron's inputs for each thread in a given partition have to be consecutive. As such, similar to address mapping in baseline GPU [13], in AXRAM data for different neurons for a given partition is distributed among the banks to enable interleaving requests to different banks on the chip.

Memory management APIs. We adopt a memory model similar to the AMD's Accelerated Processing Units [10] to provide a single physical memory space divided into two separate and non-overlapping logical memory spaces for the GPU and in-DRAM neural accelerator respectively. The separation between the GPU and in-DRAM accelerator data and the proposed data partitioning and shuffling schemes are performed on-the-fly when the host transfers the data to the GPU memory during kernel initialization using customized memory management APIs. We use an approach similar to prior work [25, 45] and modify the CUDA driver API (*e.g.* `cuMemCopyHtoD()`, `cuMemCopyDtoH()`) to implement the proposed data organization optimizations (*e.g.* data partitioning and shuffling) for in-DRAM neural acceleration. The overhead of performing the proposed data organization is amortized over the long CUDA kernel execution time and is accounted for in Section 8.

7 MEMORY MODEL

Virtual memory. Modern GPUs support simple virtual memory [16, 60, 72, 73, 83, 86]. AxRam instructions use virtual addresses similar to CUDA instructions. Once a GPU core issues an AxRam instruction, the virtual addresses are translated to physical addresses through TLBs/page tables placed in the on-chip memory controllers, similar to other CUDA instructions [72, 73]. Then, the physical addresses are sent to the memory for in-DRAM neural computation. Virtual address support in AxRam instructions expels the need to modify the underlying GPU virtual memory management system.

To fully utilize the inherent parallelism in memory (explained in Section 6), AxRam requires the data to be allocated in a consecutive memory region. Most CUDA-enabled GPUs do not support on-demand paging [9, 85]. Thus, all the virtual memory locations are backed by actual physical memory before the kernel initialization. To guarantee that a contiguous virtual memory is translated to a consecutive physical memory, we use our proposed custom memory management API to copy the allocated data to consecutive physical pages before the kernel execution. Additionally, AxRam may be extended to HSA-enabled GPUs [34]. One potential solution is to raise a page fault exception if the data for an in-DRAM invocation is not in the memory. The in-DRAM accelerator will then stall until all the demanded pages are loaded into the memory. Exploring the challenges and opportunities for integrating in-memory accelerators to HSA-enabled GPUs is outside the scope of this paper.

Cache coherency. We adopt a similar technique as [38] to guarantee the cache coherency in AxRam. The AxRam instructions bypasses the on-chip caches and communicate directly with on-chip memory controller. A GPU core always pushes all of its memory update traffic to memory before issuing any of the AxRam instructions. Sending memory update traffic along with write-through policy used in most GPUs [80] ensure that the in-DRAM accelerators have access to the most up-to-date data. `wrt_res.axram` is the only AxRam instruction that updates the data in memory. Upon receiving this instruction and in order to guarantee cache coherency, the on-chip memory controller sends a series of invalidate signals to on-chip caches and nullify any cache block that will be updated by the offloaded in-DRAM computation. The invalidate signals ensure that GPU cores never consume stale data. On average, it takes ten cycles to invalidate all the cache lines related to one neural execution. Based on our evaluation, the overhead of sending the invalidate signals to guarantee cache coherency is, on average, only 1.9%.

Memory consistency. The neural transformation does *not* introduce additional memory accesses to the approximable region. Therefore, there is no need to alter the applications. AxRam simply maintains the same memory consistency model as the baseline GPU.

8 EVALUATION AND METHODOLOGY

We evaluate AxRam with our simplified shift-add units (AxRam-SHA), fixed-point arithmetic units (AxRam-FxP), and floating-point arithmetic units (AxRam-FP).

8.1 Evaluation and Methodology

Applications and datasets. As Table 1 shows, we use a diverse set of benchmarks from the AxBench suite [87] to evaluate AxRam. AxBench comprises a combination of memory- (blackscholes, jmeint, and srad) and compute-intensive applications and comes with annotated source code, the compiler for neural transformation, separate training and test data sets, and quality measurement toolsets [87].

Neural networks and quality metric. Table 1 shows the neural network topology automatically discovered by the AxBench compiler [87] which replaces the annotated code region. We use application-specific quality metrics (Table 1) provided by AxBench [87] to assess the output quality of each application after in-DRAM acceleration. This quality loss is due to accumulated errors from repeated execution of the approximated region.

Cycle-level microarchitectural simulation. We use the `GPGPU-Sim 3.2.2` cycle-level microarchitectural simulator [7] modified with our AxRam ISA extensions with the latest configuration which closely models an `NVIDIA GTX 480` chipset with a Fermi architecture.[3] For the memory timing, this configuration models the GDDR5 timing from `Hynix` [39]. Additionally, we augmented the simulator to model the microarchitectural modifications in the GPU, the memory controller, and the GDDR5 for in-DRAM neural acceleration. The overheads of the extra instructions and logics in AxRam, on-chip memory controller invalidate signals, and the data partitioning and shuffling are faithfully modeled in our simulations. For all the baseline simulations that do not include any approximation or acceleration, we use a plain version of `GPGPU-Sim`. Table 2 summarizes the microarchitectural parameters of the GPU and GDDR5 DRAM. In all the experiments, we run the applications to completion.

Circuit and synthesis. We use the `Synopsys Design Compiler (J-2014.09-SP3)` with a `NanGate 45nm` library [1] for synthesis and energy analysis of our architecture. Additionally, we use `Cadence SoC Encounter (14.2)` for

[3]NVIDIA GTX 480 is the latest configuration in GPGPU-Sim as of time of submission.

Table 1: Applications (from AxBench [87]), quality metrics, train and evaluation datasets, and neural network configurations.

Applications			Input Dataset		Neural Network Topology
Name	Domain	Quality Metric	Training	Evaluation	
binarization	Image Processing	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	3→4→2→1
blackschole	Finance	Avg. Rel. Error	8,192 options	262,144 options	6→8→1
convolution	Machine Learning	Avg. Rel. Error	8,192 data points	262,144 data points	17→2→1
inversek2j	Robotics	Avg. Rel. Error	8,192 2D coordinates	262,144 2D coordinates	2→16→3
jmeint	3D Gaming	Miss Rate	8,192 3D coordinates	262,144 3D coordinates	18→8→2
laplacian	Image Processing	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	9→2→1
meanfilter	Machine Vision	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	7→4→1
newton-raph	Numerical Analysis	Avg. Rel. Error	8,192 cubic equations	262,144 cubic equations	5→2→1
sobel	Image Processing	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	9→4→1
srad	Medical Imaging	Image Diff.	Three 512 × 512 pixel images	Twenty 512 × 512 pixel images	5→4→1

Table 2: Major GPU, GDDR5, and in-DRAM neural accelerator microarchitectural parameters.

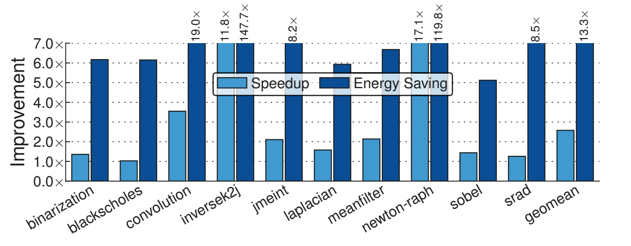
System Overview	15 SMs, 32 Threads/Warp, 6 × 32-bit P2P Memory Channels
Shader Core	1.4 GHz, 1,538 Threads (48 Warps), 32,768 registers, GTO Scheduler [84] Two Schedulers / SM
L1 Data Cache	16 KB, 128B Cache Line, 4-Way Associative, LRU Replacement Policy [44] Write Policy: Write-Evict (hit), Write No-Allocate (Miss)
Shared Memory	48 KB, 32 Banks
Interconnect	1 Crossbar/Direction (15 SMs, 6 MCs), 1.4 GHz
L2 Cache	768 KB, 128B Cache Line, 16-Way Associative, LRU Replacement Policy [44] Write Policy: Write-Evict (hit), Write No-Allocate (Miss)
Memory Model	6 × GDDR5 Memory Controllers (MCs), Double Data Rate ×32 mode 64 Columns, 4K Rows, 256 Bits/Column, 16 Banks/MC, 4 Bankgroups 2KB Row Buffer/Bank, Open Row Policy, FR-FCFS Scheduling [81, 82] 177.4 GB/Sec Off-Chip Bandwidth
GDDR5 Timing [40]	t_{WCK} = 3,696 MHz, t_{CK} = 1,848 MHz, t_{CL} = 12, t_{RP} = 12, t_{RC} = 40 t_{RAS} = 28, t_{RCD} = 12, t_{RRD} = 6, t_{CDLR} = 5, t_{WR} = 12, t_{CCD} = 2, t_{CCDL} = 3 t_{RTPL} = 2, t_{FAW} = 23, t_{32AW} = 184
GDDR5 Energy	RD/WR without I/O = 12.5 pJ/bit [40], Activation = 22.5 pJ/bit [40] DRAM I/O Energy = 2 pJ/bit, Off-Chip I/O Energy = 18 pJ/bit [70, 95]
Arithmetic Unit Energy [26, 34]	32-bit Floating-Point MAC = 0.14 nJ, 32-bit Fixed-Point MAC = 0.030 nJ 32-bit Approximate MAC = 0.0045 nJ, 32-bit Register Access = 0.9 pJ

placement and routing. As DRAM technology has only three metal layers, naïvely taking the area numbers from the `Synopsys Design Compiler` underestimates the area. To account for this, we restrict the number of metal layers to three in `Cadence SoC Encounter` for I/O pins and routing. We measure and report the area overhead of the added hardware components after the placement and routing stage with three metal layers. Similarly, for the added registers, we extract the area after the placement and routing stage while restricting the number of metal layers to three. With this infrastructure, we analyze the proposed arithmetic units, in-DRAM controllers, routing multiplexers, bypass transistors, and sigmoid LUTs.

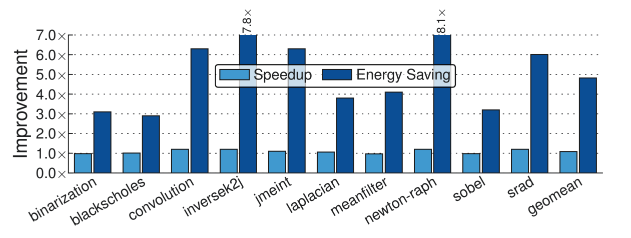
Energy modeling. To measure the energy numbers, we use `GPUWattch` [51]. We also modified the `GPGP-Sim` to generate an event log of the in-DRAM neural accelerator and all the other added microarchitectural components. We use the collected event logs to measure the energy of the in-DRAM neural acceleration. Our energy evaluations use a `NanGate 45nm` [1] process node and 1.4GHz clock frequency for the shader core (see Table 2). In-DRAM AxRam changes are modeled using `McPAT` [52] and `CACTI 6.5` [63].

8.2 Experimental Results

Performance and energy benefits with AxRam-Sha. Figure 7 shows the whole application speedup and energy reduction when all the warps undergo approximation, normalized to a baseline GPU with no acceleration and an accelerated GPU (NGPU) [22], respectively. The highest speedups are in inversek2j and newton-raph, where a large portion of their execution time is spent in the approximable region. The speedup with AxRam-Sha compared to NGPU is modest,



(a) Baseline GPU



(b) Accelerated GPU (NGPU)

Figure 7: AxRam-Sha whole application speedup and energy reduction compared to (a) baseline GPU and (b) an accelerated GPU (NGPU) [89].

because in AxRam-Sha, we use up to four iterations of shift and add operations. On average, AxRam-Sha provides 2.6× (1.1×) speedup compared to GPU (NGPU).

Figure 7 also shows the energy reduction benefit of using AxRam-Sha normalized to a baseline GPU and NGPU, respectively. The maximum energy reduction are in applications–inversek2j and newton-raph–with the highest contribution of off-chip data transfer to the whole application energy (cf. Figure 1). The off-chip data transfer contribution in jmeint is also high (90%). However, this application has a large neural network topology (cf. Table 1) which leads to a higher number of accesses to the DRAM banks to read and write temporary data, diminishing the energy reduction. On average, the studied benchmarks enjoy 13.3× (4.8×) energy reduction compared to a baseline GPU (NGPU).

Energy reduction breakdown. Figure 8 shows the energy breakdown of the DRAM system, data transfer, and data computation for AxRam-Sha, normalized to NGPU [89]. The first bar shows the breakdown of energy consumption in NGPU [89], while the second bar shows the breakdown of energy consumption in AxRam-Sha normalized to NGPU [89]. As the first bar shows, the NPGU [89] significantly reduces the contribution of the data computation in the overall system energy. Therefore, the contribution of the other

Table 3: Area overhead of the added major hardware components.

Hardware Units	Area (mm^2)	
	8 Metal Layers	3 Metal Layers
AxRAM-SHA (32 × 32-bit Approximate MACs)	0.09	0.15
AxRAM-FxP (32 × 32-bit Fixed-Point MACs)	0.40	0.76
AxRAM-FP (32 × 32-bit Floating-Point MACs)	0.54	0.97
64 × 32-bit Weight Registers	0.03	0.06
32 × Sigmoid LUT ROMs	0.19	0.34
In-DRAM Controller	0.23	0.40

main parts (e.g., data transfer and DRAM system) increases. The second bar illustrates how AXRAM-SHA significantly reduces the contribution of data transfer between the GPU cores and memory to the overall energy consumption of the system. On average, AXRAM-SHA reduces the energy consumption of data transfer by a factor of 18.5 ×. AXRAM-SHA also reduces the average energy consumption of the DRAM system by a factor of 2.5 × due to (1) decreased I/O activity and (2) a higher row-buffer hit rate. Based on our evaluation, the proposed data organization improves the row-buffer hit rate by 2.6×. Finally, the use of simplified shift-add units reduces the average contribution of data computation to the whole application energy consumption by a factor of 1.7 × compared to NGPU. These results elucidate how AXRAM reduces the overall energy consumption compared to a neurally accelerated GPU (NGPU) [22].

Design overheads. Table 3 shows the area overhead of the major hardware components added to ea ch DRAM chips. We implement the added hardware units in `Verilog` and synthesize them with `Design Compiler` using the `NanGate 45 nm` library. Similar to other DRAM architecture research papers [12, 81], we use two or three generation older logic technology to have conservative estimations. Then, we use `Cadence SoC Encounter` to perform the placement and routing on the synthesized designs using only three metal layers, similar to the baseline DRAM layout, for both routing and I/O pins. We increase the area up to a point where no placement and routing violations are identified by `Cadence SoC Encounter`. We also obtain the area overhead numbers with 8 metal layers. On average, the area overhead with three metal layers is ≈1.9× higher than with eight metal layers (Table 3). In total (including extra routing for power distribution and clock network), AXRAM-SHA consumes 1.28mm^2 (2.1%) per each GDDR5 chip with a 61.6mm^2 area [66, 67]. AXRAM-FXP and AXRAM-FP impose 2.0× and 2.4× higher area overhead compared to AXRAM-SHA. Recent work [6, 24] has proposed the integration of CGRA-style [28] accelerators atop commodity DRAM, either through TSVs or to the global I/Os. Based on our evaluation, such an integration on each DRAM chip incurs ≈47.8% area overhead. This large area overhead makes such integration an inefficient design point in GPUs. In contrast, our work leverages approximation to integrate many simplified shift-add units inside each GDDR5 chip to enable in-DRAM acceleration.

Quality loss. Figure 9 shows the quality loss of AXRAM-SHA, AXRAM-FXP, and AXRAM-FP. The quality loss is compared with that of the original precise application executed on a baseline GPU with no acceleration and an unmodified DRAM. Using fixed-point arithmetic units in

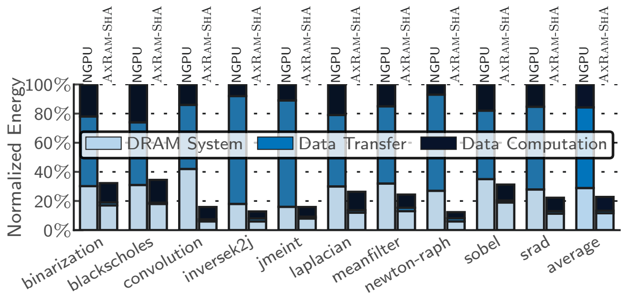


Figure 8: Breakdown of AXRAM-SHA's energy consumption between DRAM system, data transfer, and data computation normalized to NGPU [89].

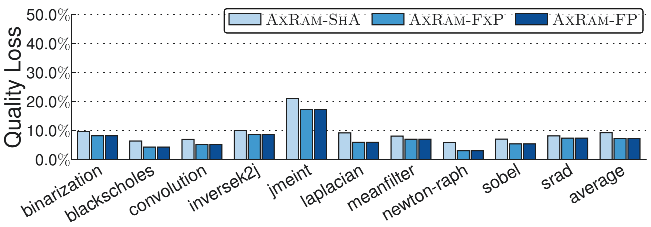
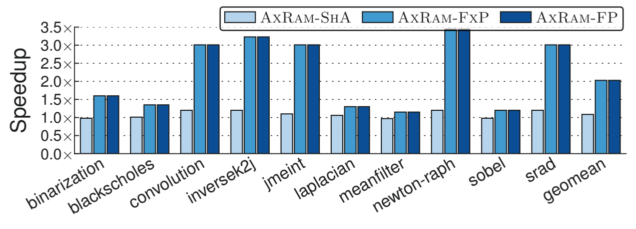


Figure 9: Whole application quality loss with AXRAM-SHA, AXRAM-FXP, and AXRAM-FP compared to a baseline GPU.

AXRAM-FXP has negligible impact on the quality loss compared to using floating-point arithmetic units in AXRAM-FP, commensurating with other work [5, 61]. Furthermore, the quality loss due to AXRAM-FP and AXRAM-FXP are the same as with NGPU. To achieve an acceptable output quality in AXRAM-SHA, we use up to four iterations of shifts and adds operations. On average, using AXRAM-SHA increases the output quality loss by 2.1% compared to the two other AXRAM microarchitectures.

Sensitivity study of AXRAM with different arithmetic units. Figure 10a compares the whole application speedup with AXRAM-SHA, AXRAM-FXP, and AXRAM-FP normalized to NGPU. Since AXRAM-SHA performs multiple iterations of shifts and adds for each MAC operations its average speedup is less than the other two AXRAM microarchitectures. AXRAM-SHA, with multiple iterations per each multiply-accumulate operation, still provides a 1.1× speedup on average. We see the same speedup across the evaluated applications for AXRAM-FP and AXRAM-FXP, which both take the same number of cycles to compute an in-DRAM neural accelerator invocation. On average, AXRAM-FP and AXRAM-FXP provide 2.0× speedup for the evaluated benchmarks. Figure 10b shows the whole application energy reduction of the three AXRAM options normalized to NGPU. On average, AXRAM-SHA achieves 4.8× energy reduction, which is 1.6× and 1.2× more than that of AXRAM-FP and AXRAM-FXP, respectively. AXRAM-SHA achieves a higher energy reduction by simplifying the integrated arithmetic units and trading off the speedup and output quality loss.

Off-chip bandwidth utilization. In Figure 11 we compare the off-chip bandwidth of AXRAM-SHA with a baseline GPU with no acceleration and an accelerated GPU (NGPU) [89]. NGPU can accelerate the data processing part of GPU applications, but it increases the off-chip bandwidth


(a) Speedup

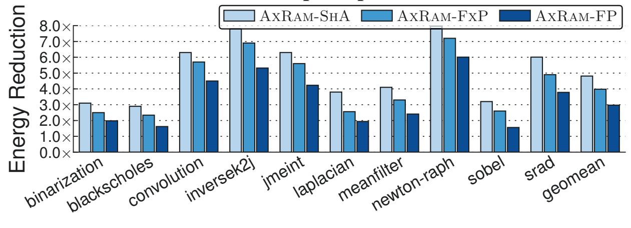

(b) Energy Reduction

Figure 10: AxRam whole application (a) speedup and (b) energy reduction with the different microarchitectural options compared to a neurally accelerated GPU (NGPU [89]).

utilization by 2.2×. However, AxRam-Sha significantly can reduce the off-chip bandwidth pressure by performing the neural computation in DRAM. This effectively eliminates most of the data transfer of the approximable region between GPU cores and DRAM. Yet, there is still a small amount of communication between the GPU cores and memory for initializing the in-DRAM execution and transferring the control messages. On average, AxRam-Sha can effectively reduce the off-chip bandwidth by a factor of 7.3× (16×) compared to NGPU (baseline GPU).

DRAM power. In this work we aim to offset that the increase in power due to integrating the arithmetic units with the decrease in overall DRAM power due to the reduction in memory I/O activity and increased row-buffer hit rate. To determine if AxRam is able to remain power neutral within DRAM, we analyze DRAM power consumption with three AxRam options in Figure 12. The reduction in the data communication and the increase in row-buffer hit rate for all the three AxRam options is the same (see Figure 11). However, as we simplify the arithmetic units, the contribution of the in-DRAM accelerators to the overall DRAM power decreases. AxRam-FP and AxRam-FxP increase the overall DRAM system power consumption by 70% and 5% on average, respectively. On the other hand, AxRam-Sha with its simplified shift-add units effectively *decreases* the average overall DRAM power consumption by 26%.

9 RELATED WORK

AxRam is fundamentally different from the prior studies on PIM in two major ways: (1) instead of using 3D/2.5D-stacked technology, we build on conventional graphics DRAM devices and (2) we study the unexplored area of tightly integrating approximate accelerators in memory. AxRam represents a convergence of two main bodies of research, approximate computing and processing in memory.

Neural acceleration. Several architectures have been proposed for neural acceleration [5, 8, 19, 20, 22, 29, 30, 59,

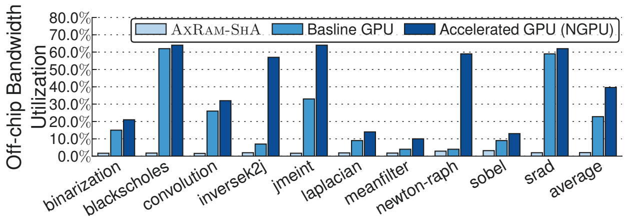


Figure 11: Off-chip memory bandwidth consumption for AxRam-Sha, a baseline GPU, and an accelerated GPU (NGPU) [89].

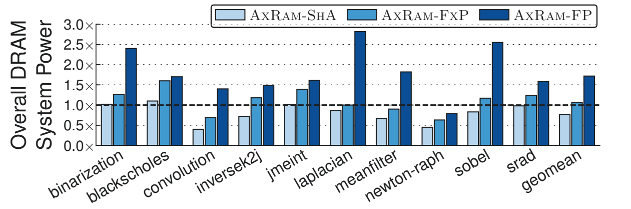


Figure 12: AxRam average DRAM system power with the different microarchitectural options normalized to a baseline GPU.

61, 89]. For example, prior work tightly integrated such neural accelerators with GPUs for significant improvement in performance and energy efficiency [89], but the improvement quickly diminishes due to limited off-chip DRAM bandwidth. In contrast, we leverage the simplicity of the neural accelerator architecture to tightly integrate them with conventional DRAM devices. This makes in-DRAM acceleration more practical and improves the gains from neural acceleration by overcoming the off-chip memory bandwidth wall. Prior to this work, the benefits, limits, and challenges of tight integration of neural accelerators in the conventional graphics DRAM devices was unexplored.

Processing in memory. Traditional PIM systems [17, 18, 21, 44, 57, 68, 70] integrate logic and memory onto a single die to enable lower data access latency and higher memory bandwidth, but they suffer from high manufacturing cost and low yield. Recently, a wealth of architectures for PIM have been proposed, ranging from fully programmable to fixed-function, using 3D/2.5D stacking technologies [3, 6, 23, 24, 26, 27, 31, 38, 47, 64, 65, 74, 79, 91, 92].

10 CONCLUSION

PIM and approximate computing are two promising approaches for higher performance and energy efficiency. This work developed AxRam, a low-overhead accelerated memory architecture that represents the confluence of these two approaches. AxRam delivers 1.1× speedup and 4.8× higher energy efficiency over even an accelerated GPU with with less than 2.1% added area to each DRAM chip.

11 ACKNOWLEDGMENTS

Amir Yazdanbakhsh is partly supported by a Microsoft Research PhD Fellowship. This work was in part supported by NSF awards CNS#1703812, ECCS#1609823, CCF#1553192, Air Force Office of Scientific Research (AFOSR) Young Investigator Program (YIP) award #FA9550-17-1-0274, NSF-1705047, Samsung Electronics, and gifts from Google, Microsoft, Xilinx, and Qualcomm.

REFERENCES

[1] 2015. NanGate FreePDK45 Open Cell Library. http://www.nangate.com. (2015). http://www.nangate.com/?page_id=2325

[2] 2015. NVIDIA Corporation. CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide. (2015). http://docs.nvidia.com/cuda/cuda-c-programming-guide

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA*.

[4] Berkin Akin, Franz Franchetti, and James C Hoe. 2015. Data Reorganization in Memory using 3D-stacked DRAM. In *ISCA*.

[5] Renée St Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-Purpose Code Acceleration with Limited-Precision Analog Computation. In *ISCA*.

[6] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. 2016. Chameleon: Versatile and Practical Near-DRAM Acceleration Architecture for Large Memory Systems. In *MICRO*.

[7] A Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA Workloads using a Detailed GPU Simulator. In *ISPASS*.

[8] Bilel Belhadj, Antoine Joubert, Zheng Li, Rodolphe Héliot, and Olivier Temam. 2013. Continuous Real-World Inputs Can Open Up Alternative Accelerator Designs. In *ISCA*.

[9] Tarun Beri, Sorav Bansal, and Subodh Kumar. 2015. A Scheduling and Runtime Framework for a Cluster of Heterogeneous Machines with Multiple Accelerators. In *IPDPS*.

[10] Pierre Boudier and Graham Sellers. 2011. Memory System on Fusion APUs. *AMD Fusion developer summit* (2011).

[11] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware. In *OOPSLA*.

[12] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu. 2016. Low-Cost Inter-Linked Subarrays (LISA): Enabling fast inter-subarray data movement in DRAM. In *HPCA*.

[13] Niladrish Chatterjee, Mike O'Connor, Gabriel H. Loh, Nuwan Jayasena, and Rajeev Balasubramonian. 2014. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *SC*.

[14] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *MICRO*.

[15] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*.

[16] Radoslav Danilak. 2009. System and Method for Hardware-based GPU Paging to System Memory. US7623134 B1.

[17] Michael F Deering, Stephen A Schlapp, and Michael G Lavelle. 1994. FBRAM: A New Form of Memory Optimized for 3D Graphics. In *SIGGRAPH*.

[18] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, et al. 2002. The Architecture of the DIVA Processing-in-Memory Chip. In *Supercomputing*.

[19] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna Palem, Olivier Temam, and Chengyong Wu. 2014. Leveraging the Error Resilience of Machine-Learning Applications for Designing Highly Energy Efficient Accelerators. In *ASP-DAC*.

[20] Schuyler Eldridge, Amos Waterland, Margo Seltzer, Jonathan Appavoo, and Ajay Joshi. 2015. Towards General-Purpose Neural Network Computing. In *PACT*.

[21] Duncan G Elliott, W Martin Snelgrove, and Michael Stumm. 1992. Computational RAM: A Memory-SIMD Hybrid and its Application to DSP. In *Custom Integrated Circuits Conference*, Vol. 30.

[22] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *MICRO*.

[23] A. Farmahini-Farahani, Jung Ho Ahn, K. Morrow, and Nam Sung Kim. 2015. DRAMA: An Architecture for Accelerated Processing Near Memory. *CAL* 14, 1 (2015).

[24] A. Farmahini-Farahani, Jung Ho Ahn, K. Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *HPCA*.

[25] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. 2013. Data Transfer Matters for GPU Computing. In *ICPADS*.

[26] M. Gao and Ch. Kozyrakis. 2016. HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In *HPCA*.

[27] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. (2017).

[28] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 32, 5 (2012), 38–51.

[29] Beayna Grigorian, Nazanin Farahpour, and Glenn Reinman. 2015. BRAINIAC: Bringing Reliable Accuracy Into Neurally-Implemented Approximate Computing. In *HPCA*.

[30] Beayna Grigorian and Glenn Reinman. 2014. Accelerating Divergent Applications on SIMD Architectures using Neural Networks. In *ICCD*.

[31] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T-M. Low, L. Pileggi, J. Hoe, and F. Franchetti. 2014. 3D-Stacked Memory-Side Acceleration: Accelerator and System Design. In *WoNDP*.

[32] Qing Guo, Xiaochen Guo, Ravi Patel, Engin Ipek, and Eby G. Friedman. 2013. AC-DIMM: Associative Computing with STT-MRAM. In *ISCA*.

[33] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding. In *ICLR*.

[34] Mark Harris. 2016. Inside Pascal: Nvidia's Newest Computing Platform. https://devblogs.nvidia.com/parallelforall/inside-pascal/. (2016). https://devblogs.nvidia.com/parallelforall/inside-pascal/

[35] Syed Minhaj Hassan, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2015. Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore. In *MEMSYS*.

[36] Mark Horowitz. [n. d.]. Energy Table for 45nm Process. ([n. d.]).

[37] Rui Hou, Lixin Zhang, Michael C Huang, Kun Wang, Hubertus Franke, Yi Ge, and Xiaotao Chang. 2011. Efficient Data Streaming with On-chip Accelerators: Opportunities and Challenges. In *HPCA*.

[38] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*.

[39] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0. 2015. (2015).

[40] Krzysztof Iniewski. 2010. *CMOS Processors and Memories*. Springer Science & Business Media.

[41] Jayesh Iyer, Corinne L Hall, Jerry Shi, and Yuchen Huang. 2006. System Memory Power and Thermal Management in Platforms Build on Intel Centrino Duo Technology. *Intel Technology Journal* 10, 2 (2006).

[42] JEDEC. October 2013. High Bandwidth Memory DRAM. http://www.jedec.org/standards-documents/docs/jesd235. (October 2013).

[43] Hadi Jooybar, Wilson W.L. Fung, Mike O'Connor, Joseph Devietti, and Tor M. Aamodt. 2013. GPUDet: A Deterministic GPU Architecture. In *ASPLOS*.

[44] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. 2012. FlexRAM: Toward an Advanced Intelligent Memory System. In *ICCD*.

[45] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX*.

[46] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (2011), 7–17.

[47] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. 2016. NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *ISCA*.

[48] K.W. Kim. 2004. Apparatus for Pipe Latch Control Circuit in Synchronous Memory Device. (2004). https://www.google.com/patents/US6724684 US6724684 B2.

[49] K. Koo, S. Ok, Y. Kang, S. Kim, C. Song, H. Lee, H. Kim, Y. Kim, J. Lee, S. Oak, Y. Lee, J. Lee, J. Lee, H. Lee, J. Jang, J. Jung, B. Choi, Y. Kim, Y. Hur, Y. Kim, B. Chung, and Y. Kim. 2012. A 1.2V 38nm 2.4Gb/s/pin 2Gb DDR4 SDRAM with Bank Group and x4 Half-Page Architecture. In *ISSCC*. 40–41.

[50] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong. 2014. 25.2 A 1.2V 8Gb 8-Channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods using 29nm Process and TSV. In *ISSCC*.

[51] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*.

[52] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*.

[53] Jiang Lin, Hongzhong Zheng, Zhichun Zhu, Howard David, and Zhao Zhang. 2007. Thermal Modeling and Management of DRAM Memory Systems. In *ISCA*.

[54] Jiang Lin, Hongzhong Zheng, Zhichun Zhu, Eugene Gorbatov, Howard David, and Zhao Zhang. 2008. Software Thermal Management of DRAM Memory for Multicore Systems. *SIGMETRICS* 36, 1 (2008), 337–348.

[55] J. Lin, H. Zheng, Z. Zhu, and Z. Zhang. 2013. Thermal Modeling and Management of DRAM Systems. *IEEE Trans. Comput.* 62, 10 (2013), 2069–2082.

[56] Song Liu, Brian Leung, Alexander Neckar, Seda Ogrenci Memik, Gokhan Memik, and Nikos Hardavellas. 2011. Hardware/Software Techniques for DRAM Thermal Management. In *HPCA*.

[57] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J Dally, and Mark Horowitz. 2000. Smart Memories: A Modular Reconfigurable Architecture. In *ISCA*.

[58] K Man. [n. d.]. Bensley FB-DIMM Performance/Thermal Management. In *Intel Developer Forum*.

[59] Lawrence McAfee and Kunle Olukotun. 2015. EMEURO: A Framework for Generating Multi-Purpose Accelerators via Deep Learning. In *CGO*.

[60] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU Memory Hierarchy through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 99 (2016).

[61] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration. In *HPCA*.

[62] Janani Mukundan, Hillery Hunter, Kyu-hyoun Kim, Jeffrey Stuecheli, and José F. Martínez. 2013. Understanding and Mitigating Refresh Overheads in High-density DDR4 DRAM Systems. In *ISCA*.

[63] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *MICRO*.

[64] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *HPCA*.

[65] R. Nair, S.F. Antao, C. Bertolli, P. Bose, J.R. Brunheroto, T. Chen, C. Cher, C.H.A. Costa, J. Doi, C. Evangelinos, B.M. Fleischer, T.W. Fox, D.S. Gallo, L. Grinberg, J.A. Gunnels, A.C. Jacob, P. Jacob, H.M. Jacobson, T. Karkhanis, C. Kim, J.H. Moreno, J.K. O'Brien, M. Ohmacht, Y. Park, D.A. Prener, B.S. Rosenburg, K.D. Ryu, O. Sallenave, M.J. Serrano, P.D.M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems. *IBM Journal of Research and Development* 59, 2/3 (2015).

[66] T. Y. Oh, Y. S. Sohn, S. J. Bae, M. S. Park, J. H. Lim, Y. K. Cho, D. H. Kim, D. M. Kim, H. R. Kim, H. J. Kim, J. H. Kim, J. K. Kim, Y. S. Kim, B. C. Kim, S. H. Kwak, J. H. Lee, J. Y. Lee, C. H. Shin, Y. Yang, B. S. Cho, S. Y. Bang, H. J. Yang, Y. R. Choi, G. S. Moon, C. G. Park, S. W. Hwang, J. D. Lim, K. I. Park, J. S. Choi, and Y. H. Jun. 2011. A 7 Gb/s/pin 1 Gbit GDDR5 SDRAM With 2.5 ns Bank to Bank Active Time and No Bank Group Restriction. *JSSC* 46, 1 (2011), 107–118.

[67] T. Y. Oh, Y. S. Sohn, S. J. Bae, M. S. Park, J. H. Lim, Y. K. Cho, D. H. Kim, D. M. Kim, H. R. Kim, H. J. Kim, J. H. Kim, J. K. Kim, Y. S. Kim, B. C. Kim, S. H. Kwak, J. H. Lee, J. Y. Lee, C. H. Shin, Y. S. Yang, B. S. Cho, S. Y. Bang, H. J. Yang, Y. R. Choi, G. S. Moon, C. G. Park, S. W. Hwang, J. D. Lim, K. I. Park, J. S. Choi, and Y. H. Jun. [n. d.]. A 7Gb/s/pin GDDR5 SDRAM with 2.5ns Bank-to-Bank Active Time and no Bank-group Restriction. In *ISSCC'10*.

[68] M. Oskin, F.T. Chong, and T. Sherwood. 1998. Active Pages: a Computation Model for Intelligent Memory. In *ISCA*.

[69] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. FlexJava: Language Support for Safe and Modular Approximate Programming. In *FSE*.

[70] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A Case for Intelligent RAM. *Micro, IEEE* 17, 2 (1997).

[71] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ACM SIGARCH Computer Architecture News*, Vol. 42. 743–758.

[72] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ASPLOS*.

[73] Jason Power, Mark D Hill, and David A Wood. 2014. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *HPCA*.

[74] S.H. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and Feifei Li. 2014. Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. *Micro, IEEE* 34, 4 (2014).

[75] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. 2009. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *ISCA*.

[76] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *MICRO*.

[77] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *PLDI*.

[78] Richard Sampson, Ming Yang, Siyuan Wei, Chaitali Chakrabarti, and Thomas F Wenisch. 2013. Sonic Millip3De: A Massively Parallel 3D-stacked Accelerator for 3D Ultrasound. In *HPCA*.

[79] C. Shelor, K. Kavi, and Adavally S. 2015. Dataflow based Near Data Processing using Coarse Grain Reconfigurable Logic. In *WoNDP*.

[80] Inderjit Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. 2013. Cache Coherence for GPU Architectures. In *HPCA*.

[81] Young Hoon Son, O. Seongil, Yuhwan Ro, Jae W. Lee, and Jung Ho Ahn. 2013. Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations. In *ISCA*.

[82] Yingying Tian, Sooraj Puthoor, Joseph L Greathouse, Bradford M Beckmann, and Daniel A Jiménez. 2015. Adaptive GPU Cache Bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*.

[83] Peter C Tong, Sonny S Yeoh, Kevin J Kranzusch, Gary D Lorensen, Kaymann L Woo, Ashish Kishen Kaul, Colyn S Case, Stefan A Gottschalk, and Dennis K Ma. 2008. Dedicated Mechanism for Page Mapping in a GPU. US20080028181 A1.

[84] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Onur Mutlu, Chita Das, Mahmut Kandemir, and Todd C. Mowry. 2015. A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Efficient Data Compression. In *ISCA*.

[85] Nicholas Wilt. 2013. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education.

[86] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU Microarchitecture

through Microbenchmarking. In *ISPASS*.

[87] Amir Yazdanbakhsh, Divya Mahajan, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2016. AxBench: A Multi-Platform Benchmark Suite for Approximate Computing: Acceleration for GPU Throughput Processors. *IEEE Design and Test* (2016).

[88] Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar, Sindhuja Sethuraman, Kartik Ramkrishnan, Nishanthi Ravindran, Rudra Jariwala, Abbas Rahimi, Hadi Esmaeilzadeh, and Kia Bazargan. 2015. Axilog: Language Support for Approximate Hardware Design. In *DATE*.

[89] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kamran, and Hadi Esmaeilzadeh. 2015. Neural Acceleration for GPU Throughput Processors. In *MICRO*.

[90] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. 2015. RFVP: Rollback-Free Value Prediction with Safe to Approximate Loads. In *TACO*.

[91] D.P. Zhang, N. Jayasena, A. Lyashevsky, J.L. Greathouse, L.F. Xu, and M. Ignatowski. 2014. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *HPDC*.

[92] Qiuling Zhu, T. Graf, H.E. Sumbul, L. Pileggi, and F. Franchetti. 2013. Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware. In *HPEC*.