2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators

Shu-Ting Wang Hanyang Xu Amin Mamandipoor[†] Rohan Mahapatra
Byung Hoon Ahn Soroush Ghodrati Krishnan Kailas[§]
Mohammad Alian[†] Hadi Esmaeilzadeh
University of California San Diego [†]University of Kansas [§]IBM Research
{shutingwang, hanyang}@ucsd.edu amin.mamandi@ku.edu {rohan, bhahn, soghodra}@ucsd.edu kailas@us.ibm.com
alian@ku.edu hadi@ucsd.edu

***Abstract*—There has been an arms race for devising accelerators for deep learning in recent years. However, real-world applications are not only neural networks but often span across multiple domains, e.g., database queries, compression, encryption, video coding, signal processing, and traditional machine learning, which *may or may not* contain deep learning. The sole focus on this single domain is sub-optimal as it misses the potential to proliferate and promote cross-domain multi-acceleration as there is an opportunity to harness the power of chaining heterogeneous Domain-Specific Architectures (DSAs) in modern datacenter applications. However, there is a catch as the *data motion* overhead can outweigh the benefits from all these chained heterogeneous accelerators. We dub the data restructuring and communication overhead of executing a single application using a chain of accelerators [1] as the *data motion overhead*. In a stark contrast with most works on DSAs that deal with accelerating compute kernels, this work focuses on accelerating data motion within a chain of heterogeneous DSAs in a multi-accelerator datacenter. To that end, this paper introduces Data Motion Acceleration (DMX) for (1) reducing data movement, (2) accelerating data restructuring, and (3) enabling interoperability between heterogeneous accelerators from different domains through a cross-stack hardware-software solution. The results with five end-to-end applications show that utilizing DMX offers up to 8.2×, 13.6×, and 5.2× improvement in latency, throughput, and energy efficiency in a multi-accelerator system, respectively.**

I. Introduction

With the effective end of Dennard Scaling [2], the dark silicon [3–5] phenomenon sparked significant interest in Domain-Specific Architectures (DSAs) or accelerators. With the Cambrian explosion of research on designing accelerators [6–64] and shifts in cloud computing [19, 65–75], it is fitting to consider the current cadence of the datacenter design as a starting point for the large-scale adoption of accelerators. For instance, Amazon Web Service (AWS) [65, 66], Microsoft Azure [67–69, 72], Google Cloud Platform (GCP) [19, 73, 74], and IBM Hybrid Multicloud [76–78] as the four providers of cloud services recently started offering accelerator equipped instances, while Meta started to use accelerators for their internal workloads [75].

To date, these early adoptions rather exclusively focus on the single domain of deep learning. Whereas, the real-world end-to-end applications cross the boundary of multiple domains (e.g., database queries, compression, encryption, video coding, signal processing, and traditional machine learning) and *may or may not* include deep neural networks. Focusing on the single domain of neural networks does not fully take advantage of the potential for acceleration. According to Amdahl's Law [79], after accelerating the neural domain, the overall speedup is and will be limited by the fraction of the code that is not accelerated. To this end, cross-domain multi-acceleration in datacenters seems to be one of the next exciting opportunities for research and development.

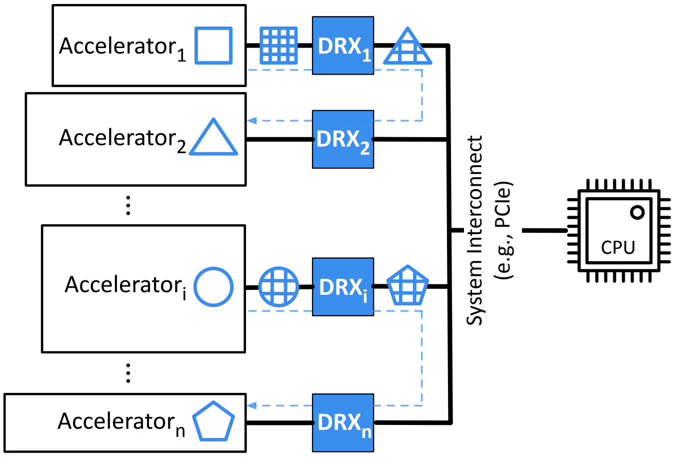


Fig. 1: Multi-accelerator systems using DMX removes the CPU from the critical path of multi-acceleration. CPU, DRX, and accelerators are connected with system interconnect, e.g., PCIe. DMX delivers the performance of a monolithic accelerator while offering the composability and programmablity of the baseline system.

While the research community has explored accelerators across different domains [56–63, 80–84], the adoption of these heterogeneous accelerator in datacenters is challenging. When the application lends itself to cross-domain multi-acceleration, chaining heterogeneous accelerators seems alluring [1], yet it may not be effective. That is because each accelerator generates and consumes data in its own specific format and chaining them requires restructuring inputs and outputs. Normally, the host CPU would be responsible for this restructuring to enable *data motion* from one accelerator to another. First, this setup puts the CPU on the critical path of operand delivery as it has to carry out the data restructuring computations to effectively chain multiple accelerators. Second, the data needs to be copied to/from the CPU across the system interconnect, which causes extra traffic. We dub these data restructuring and communication overheads as the *data motion overhead*, which can hinder multi-acceleration.

To address this pain point, we propose Data Motion Acceleration (DMX) to delegate data restructuring calculations from the CPU to a specialized, yet programmable module near the accelerators. We refer to this module as Data Restructuring Accelerator (DRX) as conceptually illustrated in Figure 1. A group of DRXs acts as a compute-enabled glue that removes the CPU from the critical path of chaining heterogeneous accelerators. As such, our innovation creates the illusion of a monolithic but composable accelerator for the application. The DRXs enable the system to compose any arbitrary chain of the available accelerators or swap them with alternatives or new ones. DRX is the microarchitecture mechanism that materializes the notion of Data Motion Acceleration (DMX).

We evaluate DMX using five end-to-end applications, each of which is composed of kernels from different domains that naturally require data restructuring to use multiple accelerators. We evaluate the benefits of various DMX topologies and configurations compared to a corresponding baseline that uses the same accelerators but, executes data restructuring on the host CPU. On average, Data Motion Acceleration (DMX) provides between 3.4× to 8.2× speedup, 3.0× to 13.6× higher throughput, and 3.8× to 5.2× energy reduction. The significant additional improvements over a baseline that itself maximally speeds up an application with multiple accelerators demonstrate the importance of data motion as heterogeneous accelerators begin to take the stage in datacenters.

II. The Case for Data Motion Acceleration

The current accelerator cards, unlike GPUs, do not have a well-supported system around proprietary interconnection and programming interfaces such as NVLINK and CUDA. Accelerator cards are developed by individual vendors using standard interconnection technologies (i.e., PCIe) and lack a standard interface to inter-operate with each other. The vendors implicitly assume that their accelerator is the only accelerator in the system. Therefore, two DSAs implemented on different accelerator cards rely on a CPU to communicate with each other following these **S**teps: (**S**1) the CPU copies the output of the first accelerator to the system memory, (**S**2) the CPU transforms the output to the second accelerator's input format, (**S**3) the CPU copies the transformed data to the second accelerator's memory, and (**S**4) the CPU fires up the computation on the second accelerator. Note that often the CPU configures a DMA device to copy data from accelerator memory to system memory. The lack of an inter-accelerator communication standard necessitates excessive *data movement* and *data restructuring overhead* for performing non-trivial data restructuring operations on general-purpose cores.

A. Data Restructuring Operations

In this work, we use five end-to-end applications that span multiple domains [82–87] to demonstrate the inefficiencies of cross-domain acceleration in a multi-accelerator system without data motion acceleration. Each of the applications has different kernels – that span both ML and non-ML domains – and data restructuring requirements between the kernels.

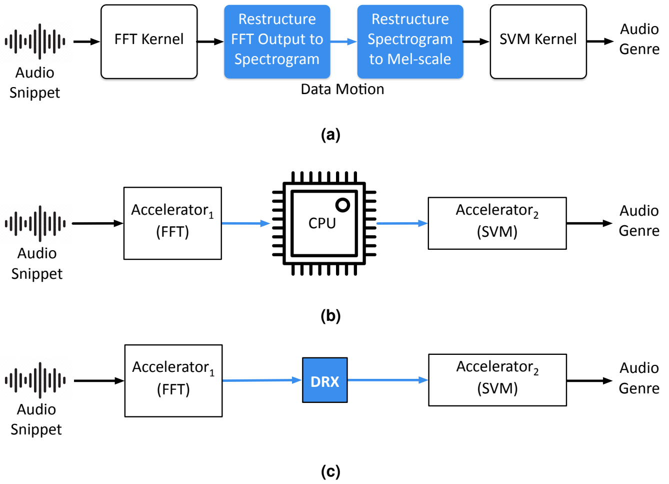


Fig. 2: (a) Data motion stands between two application kernels, i.e., Fast Fourier Transform and Support Vector Machine, of an end-to-end application. (b) Data motion is on CPU and application kernels are on their corresponding accelerators (c) For DMX, data motion is accelerated on DRX and application kernels are on their corresponding accelerators.

Figure 2(a) illustrates the end-to-end pipeline of one of the five applications. As shown, Sound Detection is composed of two domain-specific application kernels: (1) FFT kernel performs short-time Fourier transformation for the input audio snippet, and (2) support vector machine kernel determines the genre of the audio snippet. An intermediate *data motion* step is required for restructuring the output of the FFT kernel to the input format of the support vector machine kernel while copying the data from the output buffer to the input buffer. In this example, data restructuring requires generating a spectrogram from the output of FFT kernels and applying mel scale transformation to the spectrogram. The mel scale transformation maps the spectrogram into mel-frequency bins which are closer to the human-perceivable scale. In a system without data motion acceleration, the CPU should perform these data restructuring operations as shown in Figure 2(b).

B. Data Motion Overheads

Figure 3(a) shows the geometric mean of the runtime breakdown for the five cross-domain representative applications. Refer to Sec.VI for a detailed explanation of the applications. We show the results for co-running up to 15 applications on the server while data restructuring is performed on the CPU. *All-CPU* configuration runs application kernels on the CPU while *Multi-Axl* runs the application kernels on the accelerators. Because each application consists of 2 domain-specific kernels, a 15 application setup runs on 30 accelerators.

Both *All-CPU* and *Multi-Axl* are evaluated on Intel Xeon Platinum 8260L CPU. Application kernels are run on accelerators synthesized on Xilinx UltraScale+ VU9P FPGA (See Sec.VI for the detailed setup). As Figure 3(a) shows, in the *All-CPU* setup, the execution of domain-specific kernels accounts for up to 78.5% and on average 49.1% of the total runtime.

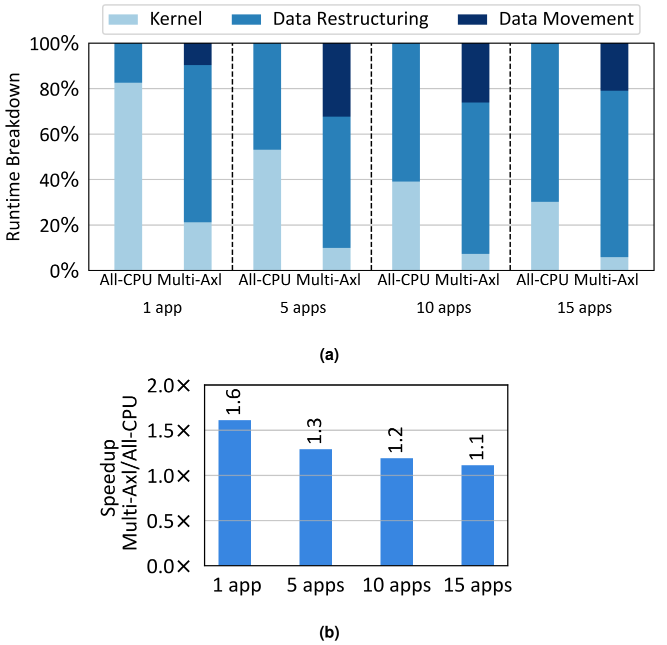


Fig. 3: (a) Runtime breakdown for *All-CPU* configuration that runs application kernels on the CPU and *Multi-Axl* configuration that runs the application kernels on the accelerators. Both configurations perform data restructuring on CPU. (b) *Multi-Axl* speedup is constrained by data motion overhead.

However, the *Multi-Axl* setup reduces the runtime of domain-specific kernels, but at the same time amplifies the ratio of data motion within the end-to-end runtime. The ratios range from 71.3% to 97.1%, showing that data motion becomes the performance bottleneck under multi-acceleration.

Another important observation from Figure 3 is the poor scalability of multi-acceleration in the absence of data motion acceleration, particularly when multiple applications are using multiple accelerators for acceleration at the same time. The transition from a single application to five applications reveals data motion as a critical bottleneck. The limited PCIe bandwidth of CPUs becomes a constraining factor for data moving in and out of the CPU for data restructuring operations, as it cannot directly connect to all accelerators simultaneously. As the number of applications increases further to 10 and 15, the CPU's inability to manage the increased concurrency of data restructuring operations becomes apparent, even with the use of 16 Xeon cores. This bottleneck in data movement and restructuring significantly hinders the overall speedup that could be achieved through multi-acceleration at scale. As illustrated in Figure 3(b), accelerating the application kernel while depending on the CPU for data restructuring results in 1.4× and 1.1× end-to-end speedup for 1 and 10 applications respectively, even though the geometric mean of per accelerator speedup is 6.5×.

These results demonstrate the untapped potential of multi-acceleration with accelerated data motion. This significant performance difference between end-to-end and per-kernel speed-up stems from the following **I**nsights: **(I1)** Using specialized accelerators reduces the runtime of kernels significantly, shifting Amdahl's bottleneck towards data motion. **(I2)** Host CPU engagement imposes inevitable data communication with accelerators, adding the cost of data movement on top of data restructuring. **(I3)** Heterogeneity in the architecture of both accelerators and CPU demands additional arithmetic operations, data type conversions, and layout transformation for data restructuring, further amplifying the cost of data motion. By heeding these insights, this work makes an initial step towards devising the concept of data motion acceleration for next-generation multi-acceleration systems.

C. DMX: Accelerating the Data Motion

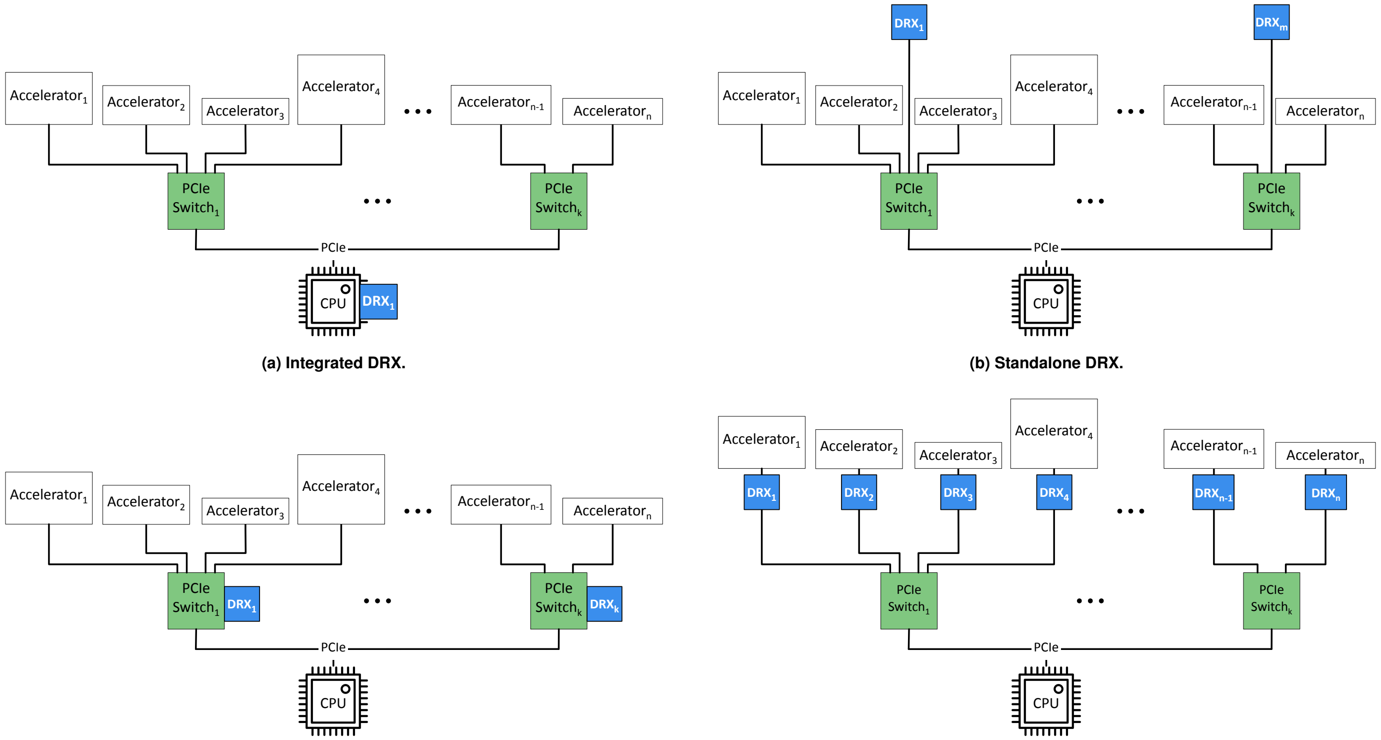
DMX accelerates data restructuring and bypasses CPU for data movement between accelerators via integrating a purposefully-built Data Restructuring Accelerator (DRX) into the system (Figure 2(c)). Realizing DMX requires synergistic design considerations at the following levels:

- **DRX Placement.** An important design decision in DMX is the location of the DRX. The placement of DRX impacts the data movement and the overall system design. We consider four different placements for DRX: integration on the CPU, standalone PCIe-attached card, per accelerator bump-in-the-wire placement, and integration on the PCIe switches.
- **Specialized Hardware Acceleration.** We need to design DRX to be programmable and support a range of data restructuring operations. As Figure 3(a) shows, data restructuring accounts for 57.7%∼73.2% of end-to-end runtime, therefore efficient execution of data restructuring is critical for multi-acceleration.
- **System Integration and Programmability.** To minimize data movement, the CPU should be removed from the critical path of accelerator-to-accelerator communication. However, the control plane should run on the CPU, otherwise, it requires a completely new programming interface that stifles interoperability of DMX across arbitrary accelerators. In Sec.V we explain a programming interface built atop an existing programming framework and show how DMX offloads the data motion to the hardware without changing the CPU-centric control plane.

In Sec.III we explore various placements for DRX and study their trade-offs. Our results show a tight integration of DRX and accelerators in a bump-in-the-wire fashion minimizes the data movement and delivers the optimal performance and energy efficiency at scale. Next, we demystify the data restructuring operations in Sec.IV and introduce a programmable accelerator specialized for the data restructuring domain. Lastly in Sec.V, we discuss the runtime and drivers that coordinate the offload of data restructuring to bump-in-the-wire DRX while still running the control plane on the CPU.

III. DRX Placement

The key design considerations in designing DMX are the placement of DRX and interconnection between DRX, accelerator, and CPU in the system. Since DMX is to enable interoperability between accelerators designed by different vendors, DRX's interconnect should be standard and well adopted. As such, the current incarnation of DMX considers



(a) Integrated DRX.

(b) Standalone DRX.

(c) PCIe-Integrated DRX.

(d) Bump-in-the-Wire DRX.

Fig. 4: DRX placement. Number of DRX units in Standalone placement (b) is configurable, and the illustration represents just one possible configuration.

PCIe as the standard interconnect to connect accelerators to CPU and DRX. PCIe is a well-established standard of interconnect and serves as the basis for future interconnects such as CXL [88].

The placement of DRX ideally should (1) scale with the capacity of associated accelerators, (2) avoid being the bandwidth bottleneck when accelerators transfer/receive data from it, and (3) minimize data movement as data movement is the main performance and energy bottleneck in today and future system [89].

Integrated DRX into CPU. This configuration considers integrating DRX with the CPU as illustrated in Figure 4(a). The integrated accelerators become more common recently as Intel Sapphire Rapids, IBM z15, POWER9, and Telum offer them in their CPU products [90–92]. Integrated accelerators are efficient in performing computation on the data that is on the CPU chip. However, integrated accelerators are going to eat up the already limited CPU power budget [3, 4]. Such power and thermal constraints limit the performance of integrated accelerators on the CPU.

DMX considers a fixed power budget for an integrated accelerator and design an Integrated DRX to operate within this power limit [91, 93]. This fixed power budget limits the performance of DRX. As we will show in Sec. VII, Integrated DRX becomes the performance bottleneck when scaling the number of accelerators to more than 8. Although integrating DRX using die-to-die interconnects like UCIe could alleviate the affect, integrated DRX still become the performance bottleneck with excessive data movement [94–96]. Moreover, Integrated DRX has the same data movement as the baseline CPU without DRX. Such design requires all accelerators to send their data to the CPU which makes the PCIe link connecting the CPU to the accelerators the bandwidth bottleneck when multiple accelerators use DRX at the same time. Such data movement is also the main source of system energy consumption.

Standalone DRX as a PCIe card. This configuration considers implementing DRX as a standalone PCIe card that is installed just like any other accelerator on a PCIe slot. Without using an external power supply cable, the performance of a single Standalone DRX PCIe card is limited by the PCIe power supply standard, which is 25 Watts. Nevertheless, as illustrated in Figure 4(b), installing multiple Standalone DRX cards can scale DRX performance with the number of accelerators. However, this Standalone DRX still incurs bandwidth oversubscription as the PCIe link to a shared, Standalone DRX card can become the bottleneck.

Compared to Integrated DRX, a Standalone DRX has the potential to reduce the data movement if DMX implements a point-to-point PCIe connection between DRX card and accelerator cards. This way, a Standalone DRX can localize the communication under the PCIe switch to which other accelerator cards are installed.

PCIe-Integrated DRX. This configuration integrates DRX onto a PCIe switch (Shown in Figure 4(c)). Compared to a Standalone DRX, A PCIe-Integrated DRX saves a round-trip between DRX and the PCIe switch. However, PCIe-

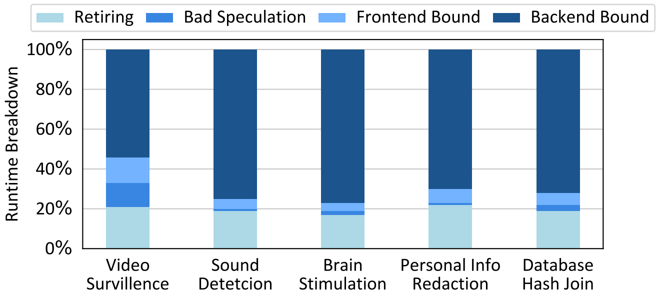


Fig. 5: Top-down breakdown of stall cycles for data restructuring operations.

Integrated DRX requires DRX to operate at the aggregated rate of all downstream PCIe ports, which adds considerable hardware complexity. Also, computation on switches only permits limited memory usage and a limited number of instructions per packet [97–100]. This configuration requires significant engineering effort to redesign the PCIe hardware and related software stack.

Bump-in-the-Wire DRX. Lastly, we introduce a Bump-in-the-Wire DRX configuration inspired by Catapult [67] that connects an exclusive DRX to each accelerator (Figure 4(d)).

Bump-in-the-Wire configuration avoids overprovisioning of PCIe links and DRX resources for a multi-accelerator system and enables DMX to scale with the hardware resources compared with the other configurations. More importantly, Bump-in-the-Wire DRX placement reduces the data movement to a minimum when accelerators communicate with each other. Coupled with a programmable DRX that enables offloading of any data restructuring operation (c.f., Sec.IV), Bump-in-the-Wire DRX serves as an option to build future scalable multi-accelerator systems.

IV. Data Restructuring Accelerator (DRX) Design

As discussed in Sec.II, the CPU is not an optimal place to perform data restructuring operations. In this section, we first analyze different data restructuring operations by profiling their execution on the CPU. This analysis guides us in devising a programmable accelerator specialized for the data restructuring domain. Refer to Sec.VI for more information on the experimental setup.

A. Data Restructuring Characterization

Figure 5 shows the top-down [101] breakdown of stall cycles for data restructuring operations. We characterize data restructuring operations with the top-down analysis of Intel VTune [102] on an Intel Xeon Gold 6242R processor. The processor has the same microarchitecture as our testbed setup on AWS (See Sec.VI for details). Across different data restructuring operations, we see at most 12.5% Bad Speculation Bound and 14% Front-End Bound cycles. A deeper analysis of *Video Surveillance* reveals that this distinct behavior is linked to a higher number of branch instructions, resulting in a relatively larger number of cycles spent on branch re-steer and uOp cache switches. On the other hand, the Back-End Bound cycles range from 53% to up to 77.6% of total cycles. The culprit for Back-End Bound cycles is both the unavailability of functional units and misses in the data cache. 23.2% of Back-End Bound cycles are Core-Bound and 46% are Memory-Bound.

The profiling shows that data restructuring operations have low L1I cache Misses Per Kilo Instructions (MPKI). The average L1I MPKI for data restructuring is 2.3. As a reference, our measurements for online services from CloudSuite [103] report an average of 7.8 L1I MPKI. Such low L1I MPKI suggests a small instruction working set for data restructuring operations that fit inside the L1I cache of the core.

The profiling also shows 50∼215 L1D MPKI and 25∼109 L2 MPKI for data restructuring operations. Such high data cache MPKI is due to the streaming data access pattern of large batches of data that are being restructured before passing to a destination accelerator. The size of each data batch is between 6∼16 MBs, which clearly does not fit in the 1MB L2 cache. Such a mismatch between dataset size and cache capacity results in cache thrashing and high data cache MPKI. For reference, the L2 MPKI of CloudSuite online services is less than 3. A small staging buffer and a simple next-line prefetcher can remove the need for large and deep cache hierarchies for data restructuring operations.

The profiling results show that all data restructuring operations have a high degree of vector unit utilization. The data restructuring kernels use 100% of available vector unit capacity which is 256 bits wide AVX-256 on our servers. We also observe a high number of ephemeral threads that are spawned by the Intel Math Kernel Library while restructuring the data. The number of threads that are spawned while running the data restructuring operations is between 130 to 140. These threads operate on the data in parallel and illustrate the high data-level parallelism and inefficiency of CPUs in executing the data restructuring operations.

B. DRX Hardware Architecture

We use the above insights to design a programmable DRX that specializes in the data restructuring domain. The main observations driving DRX design are the abundance of data-level parallelism, streaming access pattern, and non-trivial operations of data restructuring. Figure 6 overviews the architecture of DRX hardware.

DRX uses a decoupled access-execute architecture that consists of a programmable front-end specialized for walking over multi-dimensional data structures, and a configurable number of interleaved vector processing units dubbed Restructuring Engine (RE) in the same pipeline. It also includes a Transposition Engine for data transposition operations and a programmable Off-chip Data Access Engine for off-chip load/store which also houses a DMA engine that initiates data movement with other accelerators. For evaluation, we configure the DRX to contain 128 lanes of RE, a 64KB instruction cache, a 64KB data scratchpad, and 8GB of DDR4 DRAM. A DDR4 3200 memory channel sustains ∼25GBps, therefore DRX implements a single DDR4 channel to match the bandwidth of an x8 PCIe Gen 4 link.

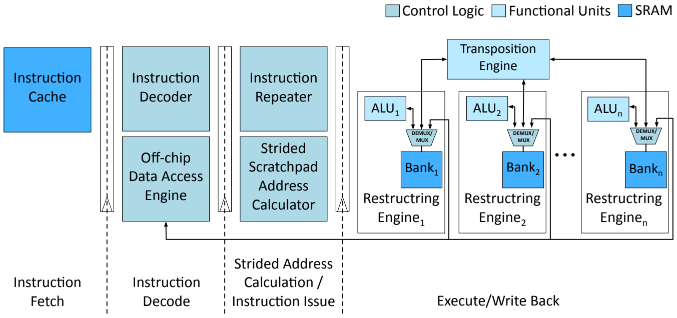


Fig. 6: DRX Hardware Architecture.

DRX ISA. The DRX ISA and hardware architecture are optimized based on the observation that data restructuring workloads consist of known-shape, pre-located multidimensional arrays. Such arrays can be indexed using a set of loops. As shown in Figure 7, the DRX ISA includes specialized loop, compute, off-chip memory access, and synchronization instructions for vector operations while preserving the option for scalar operations, enabling serial tasks like pointer dereferencing.

The DRX ISA significantly departs from traditional SIMD semantics, offering optimizations for memory, loops, and data packing. For memory optimization, DRX employs software-managed on-chip scratchpads instead of vector register files and the conventional cache hierarchy found in common SIMD ISAs. Memory instructions configure the Off-chip Data Access Engine to fetch data directly from DRAM to the on-chip scratchpads. For loop optimization, DRX utilizes hardware loops within an Instruction Repeater unit to reduce branch instruction overhead. Loop instructions configure the Instruction Repeater based on the dimensions of the kernel's multidimensional arrays. For data packing optimization, the DRX compiler partitions the kernel's multidimensional arrays across the REs, eliminating the need for pack/unpack instructions.

During the vector execution, loop instructions first configure the Off-chip Data Access Engine and Strided Scratchpad Address Calculator with sets of <Base, Stride, Iteration> configurations that correspond to the input/output loop dimensions and data location. After the Off-chip Data Access Engine loads the data to scratchpad banks, compute instruction is issued with scratchpad addresses calculated by the Instruction Repeater by traversing the dimensions of multidimensional arrays based on the configurations in the Strided Scratchpad Address Calculator. This data access scheme significantly reduces memory and address calculation overhead and is applied to all operations on multidimensional arrays such as data transformation, memory access, and compute operations. Finally, synchronization instructions are issued at the start and the end of the instruction stream to ensure proper program order. For scalar execution, DRX turns off all but one REs and operates as a scalar in-order CPU.

DRX compiler. Inspired from prior works [104–106] in other domains, DRX compiler compiles high-level data restructuring kernels into DRX instructions based on the DRX ISA. The DRX compiler takes two inputs: a high-level representation of the data restructuring kernel and an architecture configuration file that defines the DRX hardware configurations such as the number of REs and on-chip scratchpad size. The compiler first maps the data restructuring kernel to the intermediate representation of the kernel operations. It then optimizes tiling and relaxes dependency on the intermediate representation based on the hardware configuration and the dimension of multidimensional arrays. Finally, it generates instructions based on DRX ISA from the optimized intermediate representation. Figure 8 shows a sample of the DRX kernel.

	2 bits	4 bits	26 bits
Loop	Operaton	Function	Loop Dims, Base, Iter, Stride
Compute	Operaton	Function	Dest Addr, Src1 Addr, Src2 Addr
Off-chip Memory	Operaton	Function	Base/Tile Control, Req Size
Synchronization	Operaton	Function	Instruction Group, Start/Done

Fig. 7: DRX instruction types.

V. System Integration and Programmability

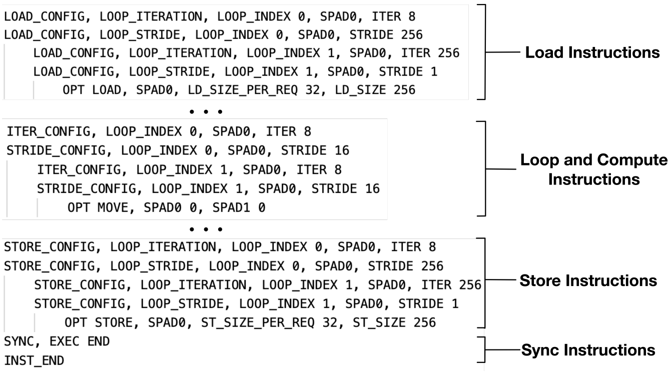
In this section we discuss the system integration and programmability of DMX with Bump-in-the-Wire DRX placement. The system integration of other DRX placements share many similarities with Bump-in-the-Wire DRX.

Programming model. DMX implements an OpenCL-style programming model that has a host program on the CPU and kernels on accelerators or DRX. Application kernels are executed on accelerators while data restructuring kernels are executed on DRX. Because DMX runs the control plane on the CPU, it does not compromise the programmer's productivity and does not incur any additional accelerator orchestration overhead compared to the baseline multi-acceleration system.

The host program creates an execution context for each instance of the application kernel or data restructuring kernel. The context includes (1) the hardware – e.g. the accelerator or DRX– involved in the applications, (2) application or data restructuring kernels, and (3) a per accelerator *command queue* that is mapped to the global host address space. The command queue is used for buffering the output of the application kernels and the restructured input of the next application kernel before being transferred to the destination.

The host program uses user-level OpenCL API to create the execution context. It also uses the API to interact with the accelerators and DRXs through their own *command queue* on each device. The command queue accepts commands to enqueue kernels for execution, transfer data, or synchronize memory buffers. The execution of a command can be blocking or non-blocking. Blocking execution does not return to the host program before the current command completes. Non-blocking execution, on the other hand, requires a detailed description of the dependency between kernels and data restructuring programs. For a single command queue, the queued commands are executed in the order they are enqueued.

The application kernels execute domain-specific kernels of the end-to-end application on different accelerators. The data restructuring kernels perform the required data restructuring



```
LOAD_CONFIG, LOOP_ITERATION, LOOP_INDEX 0, SPAD0, ITER 8
LOAD_CONFIG, LOOP_STRIDE, LOOP_INDEX 0, SPAD0, STRIDE 256
    LOAD_CONFIG, LOOP_ITERATION, LOOP_INDEX 1, SPAD0, ITER 256
    LOAD_CONFIG, LOOP_STRIDE, LOOP_INDEX 1, SPAD0, STRIDE 1
        OPT LOAD, SPAD0, LD_SIZE_PER_REQ 32, LD_SIZE 256
                          • • •
ITER_CONFIG, LOOP_INDEX 0, SPAD0, ITER 8
STRIDE_CONFIG, LOOP_INDEX 0, SPAD0, STRIDE 16
    ITER_CONFIG, LOOP_INDEX 1, SPAD0, ITER 8
    STRIDE_CONFIG, LOOP_INDEX 1, SPAD0, STRIDE 16
        OPT MOVE, SPAD0 0, SPAD1 0
                          • • •
STORE_CONFIG, LOOP_ITERATION, LOOP_INDEX 0, SPAD0, ITER 8
STORE_CONFIG, LOOP_STRIDE, LOOP_INDEX 0, SPAD0, STRIDE 256
    STORE_CONFIG, LOOP_ITERATION, LOOP_INDEX 1, SPAD0, ITER 256
    STORE_CONFIG, LOOP_STRIDE, LOOP_INDEX 1, SPAD0, STRIDE 1
        OPT STORE, SPAD0, ST_SIZE_PER_REQ 32, ST_SIZE 256
SYNC, EXEC END
INST_END
```

Fig. 8: Sample DRX kernel.

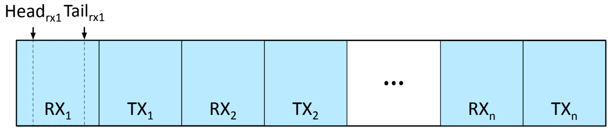


Fig. 9: RX/TX data queue pair architecture in Bump-in-the-Wire DRX. DRX uses the data queue as a circular buffer with head and tail pointers. The output of the accelerator that is destined for $Accelerator_i$ is enqueued in RX_i before being restructured and stored in TX_i for transmission to $Accelerator_i$. Current DRX implementation supports up to a total $n = 40$ accelerators.

operations when two accelerators are communicating. The host program executes the serial portion of the application and runs a daemon to orchestrate the execution of application and data restructuring kernels running on accelerators and DRXs, respectively. The data restructuring kernels are shipped to DRXs that understand the exact input and output format of each accelerator. The data restructuring kernels are engaged to ensure that properly structured input/output data is moved directly between accelerators and DRX.

Driver support for DMX. At a high level, DMX enumerates both accelerators and DRXs as PCIe devices connected to the CPU. Each DRX unit has a driver to initialize the command queues, exchange the start and end pointers of the queue to other DRXs at the start, and orchestrate data restructuring operations. The drivers use GEM [107, 108] for command executions and memory-related operations. DRX driver executes commands and reads/writes/maps operations using ioctl syscall. For setting up point-to-point DMA between DRX and accelerators, the drivers use dma-buf API [109]. The vendor-specific accelerator drivers should support point-to-point DMA in order to work with DMX. By default, we operate accelerators and DRXs in interrupt mode for sending notifications to the CPU. The interrupt handling of the drivers utilizes interrupt coalescing for the bursty arrival of interrupts. If the arrival rate of interrupts exceeds a certain threshold, the drivers switch to polling. This design is similar to Linux NAPI design [110].

Although Bump-in-the-Wire DRX is attached to each accelerator, each DRX unit should be able to set up a point-to-point connection with all the other accelerators and DRXs in the system. The memory address space of each DRX is statically partitioned between all the accelerators as well as DRXs in the system to implement two pairs of RX/TX *data queues* per accelerator on each DRX: one pair of queues for direct DRX-accelerator communication and another pair of queues for DRX-DRX communication.

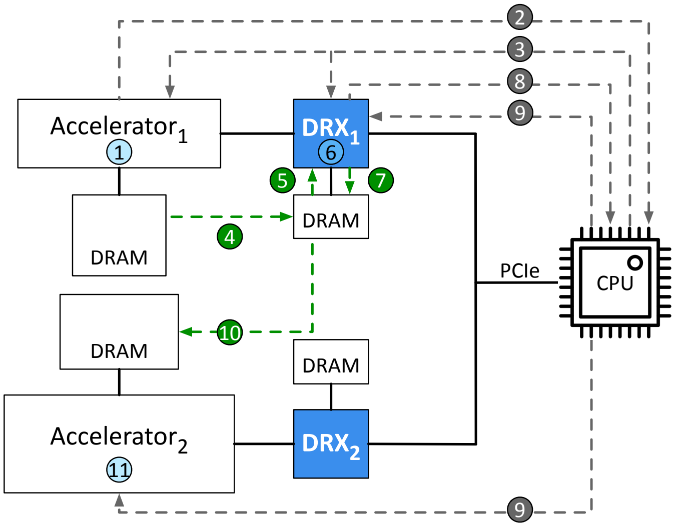


Fig. 10: Point-to-point DMA workflow involves two accelerators and the sending side DRX. The DMA bypasses the receiving side DRX. DMX supports other communication patterns such as broadcast and multicast among DRXs and between DRXs and accelerators.

The number of accelerators is determined at PCIe enumeration time when it discovers connected accelerators that need data restructuring. We provision 8GB of memory space for implementing data queues on each DRX. The size of each data queue pair is 100MB. This will enable DMX to support up to 40 accelerators on a server. DRX driver maintains a head and tail pointer for each data queue to keep track of the data that is enqueued for restructuring. RX and TX data queues on a DRX are shown in Figure 9. A point-to-point DMA moves data between data queue pairs and accelerator memory.

GEM allocates and frees data buffers opaquely because it is agnostic to the data content in the buffer. The allocated data buffers are referred to by their handle, which is equivalent to a file descriptor.

End-to-end data motion acceleration. Figure 10 shows the interactions between accelerators, CPU, and Bump-in-the-Wire DRX when $Accelerator_1$ tries to communicate with $Accelerator_2$. Although Figure 10 depicts the accelerator and its DRX as separate chips with separate DRAM modules, DRX can be integrated into the accelerator chip and share its physical DRAM modules. YesWhen $Accelerator_1$ completes kernel execution in step ①, it raises an interrupt to the CPU in step ②. The driver of $Accelerator_1$ captures the interrupt and setup a point-to-point DMA between $Accelerator_1$ and the TX data queue corresponding to $Accelerator_2$ on DRX_1. DRX_1's driver shares the offset of RX_2 data queue (i.e., RX data queue corresponding to $Accelerator_2$) in step ③ with $Accelerator_1$. This enables the $Accelerator_1$ to access and write to the RX_2 data queue on DRX_1. A DRX driver then configures $Accelerator_1$ to perform a point-to-point DMA and move data from $Accelerator_1$'s memory to the next available

Benchmark	Kernel 1	Kernel 1 Accelerator	Data Restructuring	Kernel 2	Kernel 2 Accelerator	Input Dimension
Video Surveillance [84]	H.264 Codec	Xilinx Video Codec Unit [111]	Mul, MaxPool, Reshape, Cast	Object Detection	DNN Accelerator [13]	(960, 540, 3)
Sound Detection [85]	FFT	Xilinx Vitis DSP Library [112]	Pow, Add, Mul, Div, Log10, Cast	Support Vector Machine	Xilinx Vitis Data Analytics Library [113]	(8192, 768)
Brain Stimulation [86]	FFT	Xilinx Vitis DSP Library [112]	Pow, Div, Mul, Cast	Proximal Policy Optimization	DNN Accelerator [13]	(256, 1024, 8)
Personal Information Redaction [87]	AES-GCM	Xilinx Vitis Security Library [114]	Concat, Flatten	Regular Expression	Xilinx Vitis Data Analytics Library [113]	(4, 2048, 768)
Database Hash Join [82]	Gzip	Xilinx Vitis Data [115] Compression Library	Concat, Reshape, Cast	Hash Join	Xilinx Vitis Database Library [116]	(4, 1024, 512)

TABLE I: End-to-end benchmarks

buffer in RX_2 data queue on DRX_1 in step ④. The DRX processing unit on DRX_1 reads the output on $Accelerator_1$'s memory from RX_2 data queue, performs data restructuring, and writes the output to the next available buffer in TX_2 data queue as shown in step ⑤ to ⑦. In step ⑧, DRX_1 raises an interrupt to the CPU to notify the DRX_1 driver about the completion of data restructuring. Next, a point-to-point DMA is configured between DRX_1 and $Accelerator_2$ in step ⑨. In step ⑩, point-to-point DMA between DRX_1 and $Accelerator_2$ passes through an internal PCIe multiplexer without invoking DRX_2 because it does not need further data restructuring on it. In step ⑪, $Accelerator_2$ runs the kernel on its DRAM.

One-to-many and many-to-one data movement. Supporting broadcast and multicast between the accelerator chain is necessary for load balancing as well as efficient collective communication implementation. The workflow of such movement patterns is similar to that of Figure 10, except that for one-to-many, the source DRX transfers the restructured output of the source accelerator to multiple accelerators (or DRXs) using multiple back-to-back point-to-point DMA transfers. Variations of many-to-one data movement can be used to implement reduction collectives by setting up direct data transfer from multiple source DRXs to a single destination DRX that also performs the reduction operation. The DMX support for broadcast and multicast facilitates the efficient implementation of various collective operations.

VI. Experimental Methodology

Benchmarks. We create five diverse cross-domain and end-to-end applications inspired by real-world scenarios. Table I lists the five benchmark applications, their cross-domain kernels and corresponding accelerators, the data restructuring operations needed to chain the kernels, and the dimensions of the input data. Each application is a pipeline of two kernels, where the first kernel outputs intermediate data, which requires restructuring before it can be processed by the second kernel. The Video Surveillance decodes input video streams into video frames and passes them to an object detection kernel [117]. Sound Detection performs Fast Fourier Transform (FFT) on audio snippets and use the transformed snippets to determine the genre of input audio [85]. Brain Stimulation receives electromagnetic input signal generated from a brain simulation model, processes it with FFT and data restructuring operations before outputting the data to reinforcement learning kernel [86]. Personal Information Redaction decrypts privacy-sensitive text and uses a regular expression kernel to detect personally identifiable information and redact them from the text with blanks [87]. Database Hash Join decompresses database tables and hash joins the tables [82, 83]. To exercise the system performance with respect to resource contention on interconnect bandwidth and compute for data restructuring, we use 1, 5, 10, to 15 concurrent running applications for the benchmarks.

DRX hardware implementation. We implement DRX using Verilog in RTL and synthesize it on Xilinx UltraScale+ VU9P FPGA using Xilinx Vivado 2022.2. The synthesized design achieves an operating frequency of 250 MHz. We also synthesize an ASIC version of DRX using Synopsys Design Compiler R-2020.09-SP4 with the FreePDK 15nm standard cell library [118]. The ASIC implementation achieves a 1 GHz operating frequency.

Baseline FPGA-based multi-acceleration system. Beside DRX, we also synthesize application kernels discussed earlier in this section on FPGA to implement a baseline multi-acceleration system without data motion acceleration (i.e., that uses CPU for performing data motion). This setup consists of multiple AWS Xilinx UltraScale+ VU9P FPGAs [119] connected through PCIe x16 to Intel Xeon Platinum 8260L CPUs operating at 2.4 GHz with 64 GB of memory and hyperthreading disabled.

We implement the application kernels on the FPGA using the following methods: hard-IP blocks, High-Level Synthesis (HLS), or Register-Transfer Level (RTL) implementation. For the video codec kernel, we use a pre-existing hard-IP available on the VT1 instance of AWS [120]. We use Xilinx's Vitis libraries [121], which provide HLS implementations, for kernels such as FFT, support vector machine, AES-GCM, Gzip decompression, regular expression, and database hash join. We use the RTL implementation from open-sourced accelerators [13] for the remaining kernels that use deep neural networks such as object detection and proximal policy optimization. We synthesize both the HLS and RTL implementations on the FPGAs operating at 250 MHz clock frequency.

In this FPGA multi-acceleration implementation the host CPU runs the control plane (refer to Sec.V) and performs the data restructuring operations while the FPGAs accelerate the application kernels.

Performance evaluation. We use the FPGA setup to collect cycle-level latency of executing end-to-end applications on a

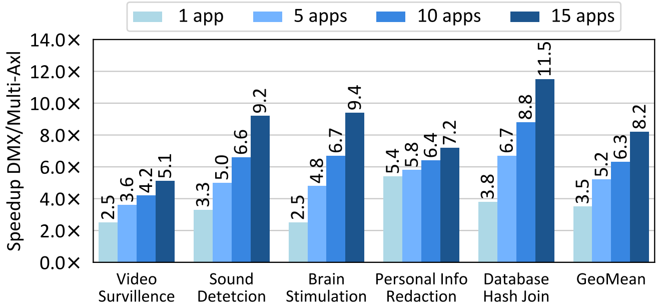
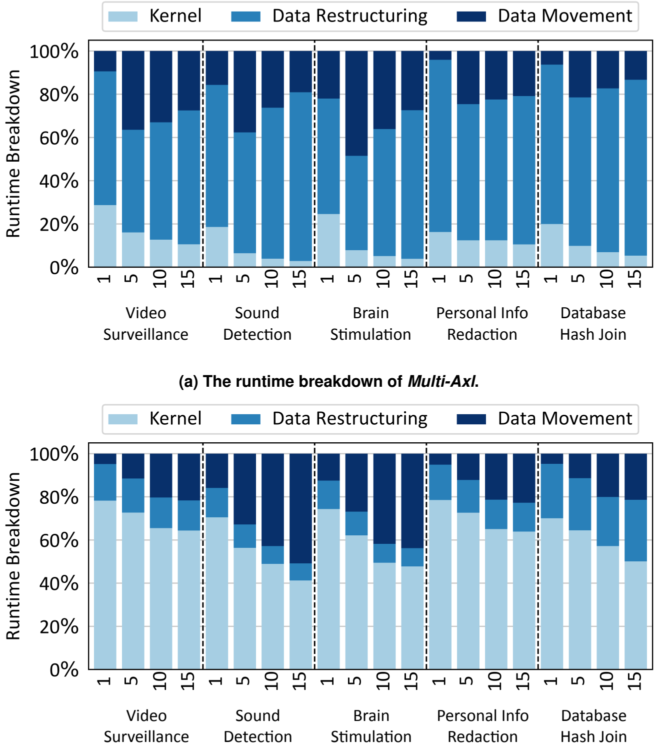


Fig. 11: DMX speedup over *Multi-Axl* configuration that uses CPU for data motion between accelerators. DMX performance scales with the number of concurrent applications by using Bump-in-the-Wire DRX placement.



(a) The runtime breakdown of *Multi-Axl*.

(b) The runtime breakdown of DMX.

Fig. 12: The latency breakdown of the *Multi-Axl* baseline and DMX. DMX shrinks data restructuring ratio from 64.1% to 14.1% in average.

baseline without data motion acceleration (we refer to this baseline as *Multi-Axl* configuration in Sec.VII). We then scale the performance of FPGA acceleration using scaling factors based on ASIC implementation and clock frequency (250 MHz to 1GHz). We develop an end-to-end system emulation infrastructure to compare the performance of various DMX configurations with a multi-acceleration baseline without DMX. The input to the emulation setup are cycle-level latency numbers for executing application kernels, data restructuring on the CPU or DRX, communication over PCIe, and software stack overheads for interrupt and polling.

Energy evaluation. We measure the energy of the CPU using Intel RAPL [122]. We use the post-synthesis power of the FPGA and multiply it by the execution time of the kernels to estimate the energy consumption for the accelerators. We also include the energy consumption of the PCIe switch [123] and the energy for data transfer over PCIe [124].

VII. Experimental Results

A. End-to-end Performance Improvement

Speedup. Figure 11 compares the end-to-end execution time of cross-domain applications withiout (*Multi-Axl*) and with DMX. Note that uses Bump-in-the-Wire DRX placement. On average, accelerating the data motion provides 3.5× to 8.2× speedup for running one to 15 concurrent applications. The higher the number of accelerators in use, the greater the data motion between the accelerators. Therefore, as DRX accelerates the data restructuring portion of the end-to-end application, the speedup grows as the number of concurrent applications increases. DMX yields less end-to-end speedup for Video Surveillance because the accelerator used for Video Surveillance provides less speedup compared to the other benchmarks. The speedup of DMX is more pronounced for Database Hash Join because the data restructuring takes up the majority of the runtime for this benchmark which is significantly being accelerated by DRX.

To better understand the sources of benefits, Figure 12(a) and Figure 12(b) report the runtime breakdown for *Multi-Axl* baseline and DMX across the three main runtime components: accelerated kernels time, data restructuring, and data movement time between CPU and accelerator for *Multi-Axl* and between accelerators for DMX. Kernel execution latencies are the same for both *Multi-Axl* and DMX. However, after we apply DMX (Figure 12(b)), the kernel execution takes up larger portion of the runtime breakdown compared to the baseline (Figure 12(a)).

As shown in Figure 12(a), data restructuring accounts for the largest portion of the end-to-end runtime for the baseline. Data restructuring is on average 66.8%, 55.7%, 64.7%, and 71.7% of multi-acceleration end-to-end latency for 1, 5, 10, and 15 concurrent applications, respectively. Using DRX significantly accelerates data restructuring and shrinks data restructuring overhead to 17.0%, 15.3%, 13.5%, and 7.2% of DMX end-to-end latency for 1, 5, 10, and 15 concurrent applications, respectively, as shown in Figure 12(b). Increasing the number of concurrent applications requires more accelerators, meaning more computation for data restructuring operations between accelerators. Furthermore, the data movement in the baseline system increases due to the bandwidth bottleneck caused by multiple accelerators sharing the PCIe switch's upstream bandwidth. On the contrary, DMX accompanies each accelerator with its own local DRX and therefore avoids bandwidth contention on shared PCIe links.

Throughput improvement. Although the end-to-end execution latency of each request is important, in a real world setup, an application receives back to back requests that need to be processed in the cross-domain application pipeline. Therefore, assuming that each application consists of three pipeline stages

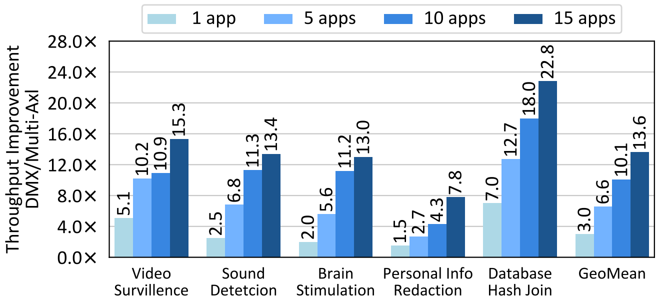


Fig. 13: DMX throughput improvement over *Multi-Axl*. DMX resolves the throughput bottleneck of data restructuring and shifts the throughput bottleneck to the accelerated kernel.

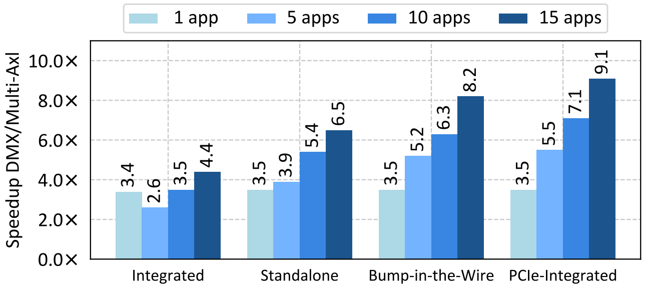


Fig. 14: Comparison of end-to-end latency speedup with different DRX placements: Integrated DRX integrates a shared DRX on the CPU. Standalone DRX implements DRX as a standalone PCIe card shared by accelerators. Bump-in-the-Wire DRX is an exclusive DRX to each accelerator. PCIe-Integrated DRX integrates shared DRXs with PCIe switches connecting accelerators.

(first kernel, data motion, and second kernel as shown in Figure 2), the throughput of an application is determined by the latency of the slowest stage. We compare the throughput of *Multi-Axl* baseline and DMX assuming continuous arrival of requests for each application.

Figure 13 shows the throughput improvement of DMX over the multi-acceleration baseline. On average, DMX achieves from 3.0× to 13.6× throughput improvements when running one to 15 concurrent applications, respectively. Data restructuring is the slowest stage of the application pipeline in the *Multi-Axl* baseline as demonstrated in Figure12(a). Hence it is the throughput bottleneck for all benchmarks, especially as the number of concurrent applications increases. DMX leverages DRX to address this bottleneck and shifts the throughput bottleneck to the accelerated kernel. Personal Info Redaction shows relatively low improvement on the throughput as its throughput is limited by its regular expression kernel accelerator. Data movement is not the throughput bottleneck for the *Multi-Axl* baseline because the PCIe bandwidth never gets saturated due to the poor throughput of data restructuring operations on the CPU.

B. DRX Placement Analysis

One of the critical design decisions in DMX is the location of the DRX in the system: Integrated, Standalone, Bump-in-the-Wire, PCIe-Integrated. This is because the placement of DRX impacts the data movement and the overall system design.

Speedup with different DRX placements. Figure 14 compares the latency speedup between Integrated DRX, Standalone DRX, Bump-in-the-Wire DRX, and PCIe-Integrated DRX. The figure reports the average speedup across the five benchmarks for one to 15 concurrent applications. For all setups from one through 15 concurrent applications, the results show that the speedups compared to the *Multi-Axl* baseline are in the following order: Integrated $\leq$ Standalone $\leq$ Bump-in-the-Wire $\leq$ PCIe-Integrated.

Integrated DRX shows 4.4× speedup with 15 concurrent applications compared to the baseline where data restructuring is performed on the CPU. However, when running more than one application in Integrated DRX, the concurrent applications contend for the shared DRX computation resources on the CPU and the PCIe bandwidth to access the shared DRX. The upstream port of the PCIe switch connecting to the CPU uses a single link (8 lanes) while the downstream ports connecting to accelerators use multiple links. Also, a PCIe transaction pays 110 ns or more port-to-port latency tax to get through a PCIe switch [123]. Despite the significant overhead from the contended PCIe links, Integrated DRX's speedup relative to the baseline increases as we add more accelerators. This demonstrates the benefits of using DRX instead of general-purpose CPU cores for data restructuring operations.

Standalone DRX shows 3% and 48% improvements compared to the Integrated for one and 15 concurrent applications, respectively. In the Integrated DRX, we have a single DRX that is integrated to the CPU for the entire system. On the other hand, the Standalone configuration scales the number of DRX with the number of concurrent applications by inserting more DRX PCIe cards. Therefore, the speedup compared to Integrated DRX can be attributed to the larger number of DRX in the system.

Bump-in-the-Wire DRX achieves 33%, 17%, and 26% higher speedup for 5, 10, and 15 concurrent applications compared with Standalone DRX. Bump-in-the-Wire DRX keeps its point-to-point DMA traffic between accelerators and DRX under the same PCIe multiplexer so the accelerators do not need to contend for PCIe bandwidth as in Standalone DRX placement on the CPU.

PCIe-Integrated DRX shows the highest speedup. The improvement of PCIe-Integrated DRX against Bump-in-the-Wire DRX comes from the saving of a round-trip between the source DRX and the source PCIe multiplexer and a pass-through of the destination PCIe multiplexer. However, it is important to note that the integration of DRX with a PCIe switch requires in-depth modification to make the PCIe switch programmable and process data at the line rate. In other words, despite the luring benefits, the prohibitive level of engineering effort to achieve it makes the Bump-in-the-Wire a reasonable choice of DMX design that can achieve significant speedup with relatively affordable engineering efforts.

Energy reduction with different DRX placements. Figure 15 shows system-wide energy reduction provided by different DRX placements compared to *Multi-Axl* baseline. Integrated DRX provides 3.4×, 3.9×, 4.0×, and 4.0× of energy reduction.

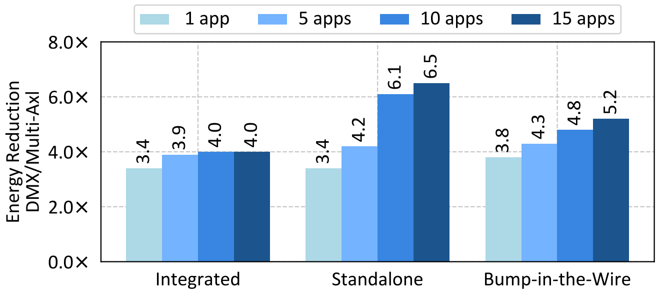
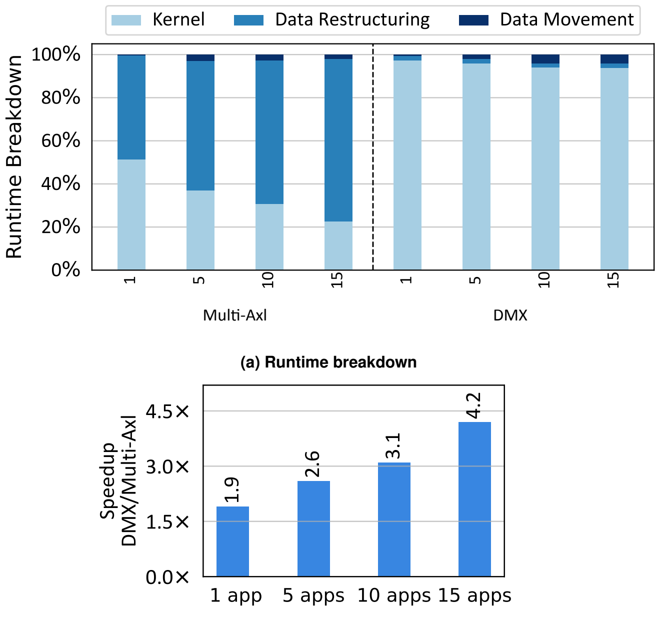


Fig. 15: System-wide energy reduction, including host CPU cores, accelerators, and DRXs. Bump-in-the-Wire DRX achieves less reduction than Standalone DRX due to its internal PCIe multiplexer shown in Fig. 4d. Integrated, Standalone, and Bump-in-the-Wire DRX draw up to 26%, 23% and 28% more power than the *Multi-Axl* baseline. PCIe-Integrated is not included because we are not able to estimate the power of a DRX-integrated PCIe switch.

The energy reduction does not scale with the number of concurrent applications because it only benefits from the energy efficiency of the DRX hardware acceleration for data restructuring operations. Standalone DRX and Bump-in-the-wire DRX provide energy reduction scaling with the increased number of concurrent applications. Bump-in-the-wire DRX placement delivers the best energy reduction of 3.8× and 4.3× for 1 and 5 concurrent applications. Standalone DRX delivers the best energy reduction of 6.1× and 6.5× for 10 and 15 concurrent applications because of the reduced bandwidth contention on PCIe links. This is because the extra glue logic and the dual-port PCIe multiplexer are replicated in each Bump-in-the-Wire DRX placement, while such overhead is amortized across the applications on a large Standalone DRX. PCIe-Integrated is not evaluated for energy reduction because of the difficulty of estimating the energy consumption of a PCIe switch integrated with DRX.

C. Sensitivity Studies

Speedup with more than two kernels. As real-world applications can consist of multiple kernels across domains, it is important for DMX to scale beyond two kernels. To evaluate DMX's scalability with multiple application kernels, we add a third application kernel to the Personal Info Redaction benchmark, along with its additional data restructuring kernel consisting of reshaping and typecasting. This third kernel is a Transformer model fine-tuned for Named Entity Recognition (NER). NER identifies personal and sensitive information that is hard to capture for regular expression kernel [125]. We use an open-source BERT implementation for the kernel [126]. Figure 16(a) shows the runtime breakdown of this three-kernel benchmark. Although the benchmark included the compute-intensive NER kernel, the runtime is still dominated by the data restructuring kernels for the *Multi-Axl* baseline. DMX alleviates the bottleneck of data motion and restores kernel to be the largest contributor that represents 97.2% to 93.7% of the end-to-end execution time for one to 15 concurrent applications. As such, DMX provides 1.9× to 4.2× speedup for one to 15 concurrent applications shown in Figure 16(b).



(a) Runtime breakdown

(b) Speedup

Fig. 16: DMX reduces data motion overhead to less than 5% for Personal Info Redaction benchmark extended with Named Entity Recognition kernel.

One-to-many and many-to-one data movement. Cross-domain multi-acceleration of end-to-end applications entails using multiple accelerators. The data movements in multi-acceleration, however, are not necessarily always one-to-one but likely include one-to-many and/or many-to-one data movement between accelerators. Therefore, we want to analyze whether DMX design can cope with the one-to-many and/or many-to-one data movements in multi-acceleration. To this end, we compare Bump-in-the-Wire DRX against the *Multi-Axl* baseline for one-to-many (i.e., broadcast) and many-to-one (all-reduce) data movement using 4 to 32 accelerators. For broadcast, the baseline first passes the output of the source accelerators to the main memory of the CPU using DMA. After data restructuring on the CPU, the driver then copies the restructured data and initiates *N* DMA transfers *sequentially* to the destination accelerators. All-reduce has two stages: scatter-reduce and all-gather. Both require similar DMA transfers between CPU and accelerators; however, scatter-reduce entails additional steps to first sum the inputs from sources and then scatter the outputs to all destinations. On the contrary, DMX's implementation of broadcast and all-reduce utilizes the Bump-in-the-Wire DRX for data restructuring and data movement.

Figure 17 shows that the DMX achieves 3.7× to 5.2× speedup on broadcast and 5.1× to 10.5× speedup for all-reduce on 4 to 32 accelerators. This is because DMX utilizes DRXs to (1) perform data restructuring and the DMA transfers *in parallel* and (2) eliminate the extra DMA transfers between the accelerators and the CPU. Furthermore, for all-reduce, DMX uses DRX to accelerate the summation operations. The speedup also scales with the number of accelerators because the amount of data restructuring and data movement scale accordingly to the number of accelerators. There is a dip

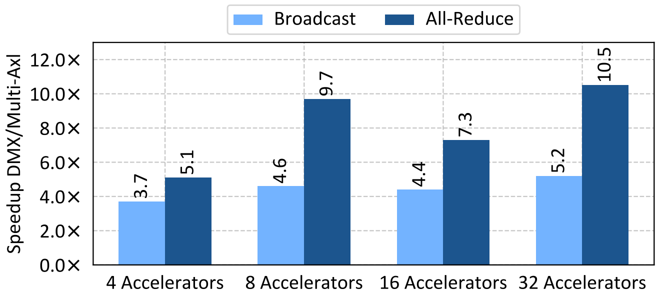


Fig. 17: DMX eliminates redundant DMA transfers and performs DMA in parallel for broadcast and all-reduce on multi-accelerator setup.

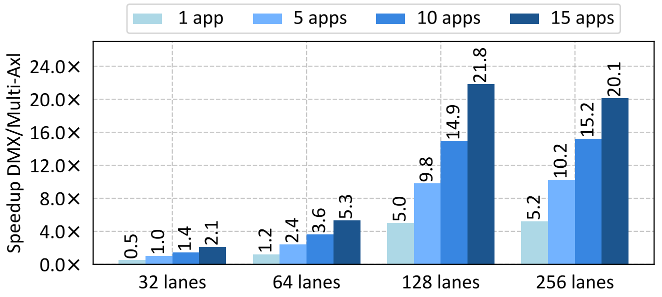


Fig. 18: Data restructuring latency speedup with different numbers of RE lanes on DRX. The increase of speedup is limited after 128 lanes. which is our default configuration.

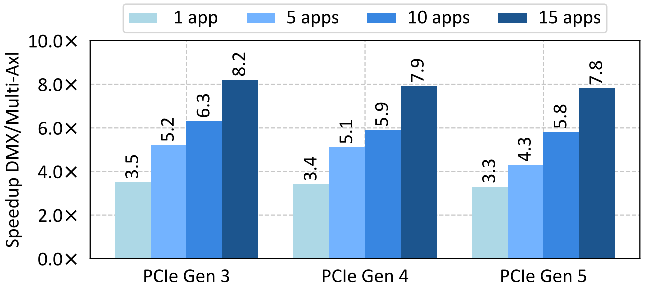


Fig. 19: DMX speedup across generations of PCIe. PCIe Gen4 and Gen5 result in a slight decrease of speedup because their corresponding *Multi-Axl* baselines improve more than their DMX counterparts.

when using 16 or more accelerators, but this is due to the additional latency on the PCIe switches that scales with the number of accelerators. DMX achieved higher speedup in all-reduce compared to broadcast because all-reduce involves more DMA transfers and data restructuring which provided more acceleration opportunity using DRX.

DRX hardware configurations. To understand the sensitivity of DMX to the amount of compute resources in DRX, we sweep the number of RE lanes for DRX and compare its performance to the *Multi-Axl* baseline that performs data restructuring on CPU. Figure 18 shows the speedup achieved for the different number of lanes for DRX: from 32 to 256. The speedup improves with the number of lanes increasing up to 128 lanes by taking advantage of available data parallelism in data restructuring operations. However, the increase of speedup of DRX is limited after 128 RE lanes, and increasing the lanes to 256 does not provide noticeable benefits. Therefore, we use 128 RE lanes as the default configuration for DRX throughout the experiments.

Different PCIe generations. Newer PCIe generation provides significantly more bandwidth and the increased bandwidth can potentially negate the performance benefit of DMX. To understand the impact of different generations of PCIe, we compare the Bump-in-the-Wire DRX latency speedup on PCIe Gen 3 with PCIe Gen 4 and Gen 5. Figure 19 shows that using PCIe Gen 4 and Gen 5 resulted in a slight decrease of speedup because their corresponding *Multi-Axl* baselines improve more than their DMX counterparts. Across PCIe generations, the baselines and DMX show different levels of improvement only in data movement latency. Such differences come from the following two reasons. First, the baselines face more bandwidth contention than the DMX and thus benefit more from the increased PCIe bandwidth per lane. Second, the baselines are able to use more PCIe lanes to reduce bandwidth contention from accelerators to CPUs with PCIe Gen 4 and Gen 5 compared to CPUs with PCIe Gen 3 [127–129]. The results shown in Figure 19 suggests that the bottleneck of *Multi-Axl* configuration is not just the PCIe interconnect, but also the data restructuring computation.

VIII. Related Work

Real-world applications span multiple domains, posing a challenge for end-to-end acceleration. While the research community has explored accelerators across diverse domains [56–63, 80–84], the adoption of these heterogeneous accelerator to accelerate a single end-to-end application is challenging. The challenge arises due to the diverse data formats generated and consumed by each accelerator. This necessitates restructuring inputs and outputs across accelerators. While some prior works have focused on performed data restructuring using CPUs, this paper introduces the concept of Data Motion Acceleration (DMX) for efficient cross-domain multi-acceleration with heterogeneous DSAs. We review the most relevant related work in three areas: data movement, data restructuring, and interconnect fabrics integration below.

Data movement. Prior works studied point-to-point data movement between GPUs [130], between GPU and storage [131–133], between NIC and accelerator [134–136], and between on-chip accelerators [137]. Prior works have used various techniques such as scheduling [135, 136, 138–140] to co-locate multiple domains on the same system. While these works only optimize the data movement, non-trivial operations of the data restructuring still consumes a significant fraction of the data motion. Intel Data Stream Accelerator [141] and DCS [142, 143] share a similar insight, both lack programmability and hence have limited capacity to optimize data restructuring. This work in contrast leverages DRXs as a compute-enabled glue that links different heterogeneous accelerators together and makes them appear as a monolithic but composable accelerator for the application.

Data restructuring. For message serialization, Optimus Prime [144] and Protobuf accelerator [145] design an accelerator for RPC message serialization. HGum [146] and Fletcher [147] implement serialization on FPGAs for acceleration. For machine learning pipelines, tf.data [148], DSI [149], DALI [150] optimize data restructuring on GPU with programmable operations. In contrast to these prior works that only optimize data restructuring for a single accelerator, this paper investigates data restructuring and movement for multi-acceleration with heterogeneous devices.

Interconnect fabrics. Previous works have used PCIe's Non-Transparent Bridge (NTB) to enable PCIe to support multiple hosts with more than one root complex, which performs address translation for operations in a specific memory range [151, 152]. Point-to-point DMA over PCIe fabric is enabled by a shared address space across all devices [153]. CXL 3.0 or later allows accelerators on different servers to be connected seamlessly by using fabric switching to link racks of devices and accelerators [88]. DUA [154] creates an overlay fabric on top of the existing physical communication stacks, such as PCIe, Ethernet, DDR, etc. These works can connect accelerators without addressing data restructuring for multi-acceleration. This work, however, tackles data motion challenges to maximize the performance of multi-acceleration.

IX. Conclusion

In this work, we quantified the data motion cost of chaining heterogeneous Domain-Specific Architectures (DSAs) for cross-domain multi-acceleration. The results showed that the data motion overhead curtails the end-to-end speedup of accelerating each domain on a set of heterogeneous DSAs. The paper introduced DMX that seamlessly weaves together multiple accelerators that deliver the performance of a large, monolithic cross-domain accelerator. On average, Data Motion Acceleration (DMX) provides between 3.4× to 8.2× speedup, 3.0× to 13.6× higher throughput, and 3.8× to 5.2× energy reduction.

Even with current single-domain DSAs, overheads of moving data on- and off-chip is presently a dominant factor that limits the performance and energy efficiency of gains [89, 155]. The impact of the data motion–highlighted in this paper–will worsen when cross-domain accelerators are chained in future datacenters to cater to the requirements of emerging end-to-end applications. This even includes the multimodal generative AI applications that use multiple models and require acceleration beyond neural networks (e.g., vector database lookups, search, etc.). Heterogeneous/3D integration coupled with emerging high-bandwidth chiplet-to-chiplet interconnects such as UCIe can improve data movement, but not data restructuring that requires computation. As such, embedding our DMX concept and architecture within these interconnects can synergistically unlock the the potential of cross-domain multi-acceleration for next-generation datacenters.

ACKNOWLEDGMENT

This work was in part supported by generous gifts from Google, Microsoft, Samsung, Qualcomm, AMD Xilinx as well as the National Science Foundation (NSF) awards CCF#2107598, CNS#1822273, National Institute of Health (NIH) award #R01EB028350, Defense Advanced Research Project Agency (DARPA) under agreement number #HR0011-18-C-0020, and Semiconductor Research Corporation (SRC) award #2021-AH-3039. This work was also supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of Google, Qualcomm, Microsoft, Xilinx, Samsung, IBM, NSF, SRC, NIH, DARPA or the U.S. Government.

References

[1] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589082. URL https://doi.org/10.1145/3579371.3589082.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *JSSC*, 1974.

[3] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[4] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–Aug. 2011.

[5] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[7] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.

[8] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai

Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ASPLOS*, 2015.

[9] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *ISCA*, 2015.

[10] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *ISCA*, 2016.

[11] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *MICRO*, 2016.

[12] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization. In *DAC*, 2016.

[13] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kim, Chenkai Shao, Asit Misra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *MICRO*, 2016.

[14] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *MICRO*, 2016.

[15] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, 2016.

[16] Yongming Shen, Michael Ferdman, and Peter Milder. Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer. In *FCCM*, 2017.

[17] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, 2017.

[18] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *ISCA*, 2017.

[19] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.

[20] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing cnn accelerator efficiency through resource partitioning. In *ISCA*, 2017.

[21] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ASPLOS*, 2018.

[22] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In *ISSCC*, 2018.

[23] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *JETCAS*, 2019.

[24] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *MICRO*, 2019.

[25] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *ASPLOS*, 2019.

[26] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *MICRO*, 2020.

[27] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A gcn accelerator with hybrid architecture. In *HPCA*, 2020.

[28] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *MICRO*, 2020.

[29] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. *HPCA*, 2020.

[30] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, Huawei LI, Dawen Xu, and Xiaowei Li. Engn: A high-

throughput and energy-efficient accelerator for large graph neural networks. *TC*, 2021.

[31] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *HPCA*, 2021.

[32] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO*, 2016.

[33] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H. Peter Hofstee, Gi-Joon Nam, Mark R. Nutter, and Damir Jamsek. Extrav: Boosting graph processing near storage with a coherent accelerator. *PVLDB*, 2017.

[34] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. An efficient graph accelerator with parallel data conflict management. In *PACT*, 2018.

[35] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *MICRO*, 2018.

[36] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *HPCA*, 2018.

[37] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *HPCA*, 2018.

[38] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *ASPLOS*, 2018.

[39] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. Conda: Efficient cache coherence support for near-data accelerators. In *ISCA*, 2019.

[40] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Phi: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *MICRO*, 2019.

[41] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *MICRO*, 2020.

[42] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. Depgraph: A dependency-driven accelerator for efficient iterative graph processing. In *HPCA*, 2021.

[43] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *MICRO*, 2021.

[44] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, and Haikun Liu. Lccg: A locality-centric hardware accelerator for high throughput of concurrent graph processing. In *SC*, 2021.

[45] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. In *ASPLOS*, 2018.

[46] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. Genax: A genome sequencing accelerator. In *ISCA*, 2018.

[47] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. Smem++: A pipelined and time-multiplexed smem seeding accelerator for genome sequencing. In *FPL*, 2018.

[48] Subho Sankar Banerjee, Mohamed El-Hadedy, Jong Bin Lim, Zbigniew T. Kalbarczyk, Deming Chen, Steven S. Lumetta, and Ravishankar K. Iyer. Asap: Accelerated short-read alignment on programmable hardware. *IEEE Transactions on Computers*, 2019.

[49] Anirban Nag, C. N. Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. Gencache: Leveraging in-cache operators for efficient sequence alignment. In *MICRO*, 2019.

[50] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm. In *MICRO*, 2019.

[51] Damla Senol Cali, Gurpreet S. Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S. Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, Anant Norion, Allison Scibisz, Sreenivas Subramoneyon, Can Alkan, Saugata Ghose, and Onur Mutlu. Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *MICRO*, 2020.

[52] Yeseong Kim, Mohsen Imani, Niema Moshiri, and Tajana Rosing. Geniehd: Efficient dna pattern matching accelerator using hyperdimensional computing. In *DATE*, 2020.

[53] Wenqin Huangfu, Krishna T. Malladi, Shuangchen Li, Peng Gu, and Yuan Xie. Nest: Dimm based near-data-processing accelerator for k-mer counting. In *ICCAD*, 2020.

[54] Ann Franchesca Laguna, Hasindu Gamaarachchi, Xunzhao Yin, Michael Niemier, Sri Parameswaran, and X. Sharon Hu. Seed-and-vote based in-memory accelerator for dna read mapping. In *ICCAD*, 2020.

[55] Daichi Fujiki, Shunhao Wu, Nathan Ozog, Kush Goliya, David Blaauw, Satish Narayanasamy, and Reetuparna Das. Seedex: A genome sequencing accelerator for optimal alignments in subminimal space. In *MICRO*, 2020.

[56] Abbas Haghi, Santiago Marco-Sola, Lluc Alvarez, Dionysios Diamantopoulos, Christoph Hagleitner, and Miquel Moreto. An fpga accelerator of the wavefront

algorithm for genomics pairwise alignment. In *FPL*, 2021.

[57] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. Genstore: A high-performance in-storage processing system for genome sequence analysis. In *ASPLOS*, 2022.

[58] Damla Senol Cali, Konstantinos Kanellopoulos, Joël Lindegger, Zülal Bingöl, Gurpreet S. Kalsi, Ziyi Zuo, Can Firtina, Meryem Banu Cavlak, Jeremie Kim, Nika Mansouri Ghiasi, Gagandeep Singh, Juan Gómez-Luna, Nour Almadhoun Alserr, Mohammed Alser, Sreenivas Subramoney, Can Alkan, Saugata Ghose, and Onur Mutlu. Segram: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping. In *ISCA*, 2022.

[59] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for inmemory databases. In *MICRO*, 2013.

[60] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J. Sorin. The microarchitecture of a real-time robot motion planning accelerator. In *MICRO*, 2016.

[61] Jacob Sacks, Divya Mahajan, Richard C Lawson, and Hadi Esmaeilzadeh. Robox: an end-to-end solution to accelerate autonomous control in robotics. In *ISCA*, 2018.

[62] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. Pointacc: Efficient point cloud accelerator. In *MICRO*, 2021.

[63] Sabrina M. Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. Robomorphic computing: A design methodology for domain-specific accelerators parameterized by robot morphology. In *ASPLOS*, 2021.

[64] Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Wang, Sanchari Sen, Jintao Zhang, Ankur Agrawal, Monodeep Kar, Shubham Jain, Alberto Mannari, Hoang Tran, Yulong Li, Eri Ogawa, Kazuaki Ishizaki, Hiroshi Inoue, Marcel Schaal, Mauricio Serrano, Jungwook Choi, Xiao Sun, Naigang Wang, Chia-Yu Chen, Allison Allain, James Bonano, Nianzheng Cao, Robert Casatuta, Matthew Cohen, Bruce Fleischer, Michael Guillorn, Howard Haynie, Jinwook Jung, Mingu Kang, Kyuhyoun Kim, Siyu Koswatta, Saekyu Lee, Martin Lutz, Silvia Mueller, Jinwook Oh, Ashish Ranjan, Zhibin Ren, Scot Rider, Kerstin Schelm, Michael Scheuermann, Joel Silberman, Jie Yang, Vidhi Zalani, Xin Zhang, Ching Zhou, Matt Ziegler, Vinay Shah, Moriyoshi Ohara, Pong-Fei Lu, Brian Curran, Sunil Shukla, Leland Chang, and Kailash Gopalakrishnan. RaPiD: AI accelerator for ultra-low precision training and inference. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 153–166. IEEE Press, 2021. ISBN 9781450390866. doi: 10.1109/ISCA52012.2021.00021.

[65] AWS inferentia, . URL https://aws.amazon.com/machine-learning/inferentia/.

[66] AWS trainium, . URL https://aws.amazon.com/machine-learning/trainium/.

[67] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, 2014.

[68] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *MICRO*, 2016.

[69] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, 2018.

[70] Rohan Mahapatra, Soroush Ghodrati, Byung Hoon Ahn, Sean Kinzer, Shu ting Wang, Hanyang Xu, Lavanya Karthikeyan, Hardik Sharma, Amir Yazdanbakhsh, Mohammad Alian, and Hadi Esmaeilzadeh. Domain-specific computational storage for serverless computing. *arXiv*, 2023.

[71] Joon Kyung Kim, Byung Hoon Ahn, Sean Kinzer, Soroush Ghodrati, Rohan Mahapatra, Brahmendra Yatham, Shu-Ting Wang, Dohee Kim, Parisa Sarikhani, Babak Mahmoudi, Divya Mahajan, Jongse Park, and Hadi Esmaeilzadeh. Yin-yang: Programming abstractions for cross-domain multi-acceleration. *IEEE Micro*, 42(5):89–98, 2022. doi: 10.1109/MM.2022.3189416.

[72] Azure zipline, . URL https://azure.microsoft.com/en-us/blog/improved-cloud-service-performance-through-asic-acceleration/.

[73] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *ISCA*, 2021.

[74] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clin-

ton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. Warehouse-scale video acceleration: Co-design and deployment in the wild. In *ASPLOS*, 2021.

[75] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, Adam Hutchin, Utku Diril, Krishnakumar Nair, Ehsan K. Aredestani, Martin Schatz, Yuchen Hao, Rakesh Komuravelli, Kunming Ho, Sameer Abu Asal, Joe Shajrawi, Kevin Quinn, Nagesh Sreedhara, Pankaj Kansal, Willie Wei, Dheepak Jayaraman, Linda Cheng, Pritam Chopda, Eric Wang, Ajay Bikumandla, Arun Karthik Sengottuvel, Krishna Thottempudi, Ashwin Narasimha, Brian Dodds, Cao Gao, Jiyuan Zhang, Mohammed Al-Sanabani, Ana Zehtabioskuie, Jordan Fix, Hangchen Yu, Richard Li, Kaustubh Gondkar, Jack Montgomery, Mike Tsai, Saritha Dwarakapuram, Sanjay Desai, Nili Avidan, Poorvaja Ramani, Karthik Narayanan, Ajit Mathews, Sethu Gopal, Maxim Naumov, Vijay Rao, Krishna Noru, Harikrishna Reddy, Prahlad Venkatapuram, and Alexis Bjorlin. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi: 10.1145/3579371.3589348. URL https://doi.org/10.1145/3579371.3589348.

[76] Rohit Badlaney. How IBM is helping clients deploy foundation models and AI workloads with new GPU offering on IBM cloud, 2023. URL https://newsroom.ibm.com/How-IBM-is-Helping-Clients-Deploy-Foundation-Models-and-AI-Workloads-with-New-GPU-Offering-on-IBM-Cloud.

[77] T. Gershon, B. Karacali-Akyamac, S. Seelam, D. Thorstensen, and R. Badlaney. Supercharging IBM's cloud-native AI supercomputer, 2023. URL https://research.ibm.com/blog/vela-ai-supercomputer-updates.

[78] Jeffrey Burns and Leland Chang. IBM artificial intelligence unit, 2022. URL https://research.ibm.com/blog/ibm-artificial-intelligence-unit-aiu.

[79] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 9781450378956. doi: 10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

[80] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 255–268, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323055. doi: 10.1145/2541940.2541961. URL https://doi.org/10.1145/2541940.2541961.

[81] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 468–479, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450326384. doi: 10.1145/2540708.2540748. URL https://doi.org/10.1145/2540708.2540748.

[82] David Sidler, Muhsen Owaida, Zsolt István, Kaan Kara, and Gustavo Alonso. doppiodb: A hardware accelerated database. In *FPL*, 2017.

[83] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. Hardware acceleration of compression and encryption in sap hana. *Proc. VLDB Endow.*, 15(12):3277–3291, 2022.

[84] Afzal Ahmad, Muhammad Adeel Pasha, and Ghulam Jilani Raza. Accelerating tiny yolov3 using fpga-based hardware/software co-design. In *IEEE ISCAS*, 2020.

[85] Justin Salamon, Christopher Jacoby, and Juan Pablo Bello. A dataset and taxonomy for urban sound research. In *ACM Multimedia*, 2014.

[86] Dmitrii Krylov, Remi des Combes, Romain Laroche, Michael Rosenblum, and Dmitry V Dylov. Reinforcement learning framework for deep brain stimulation study. In *IJCAI*, 2020.

[87] Presidio: Data protection and anonymization sdk, . URL https://microsoft.github.io/presidio/.

[88] Cxl 3.0 specification. URL https://www.computeexpresslink.org/download-the-specification.

[89] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014. doi: 10.1109/ISSCC.2014.6757323.

[90] Arijit Biswas. Sapphire rapids. In *Hot Chips*, 2021.

[91] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke,

Haren Myneni, and Charlie Wang. Data compression accelerator on IBM POWER9 and z15 processors : Industrial product. In *ISCA*, 2020.

[92] Cedric Lichtenau, Alper Buyuktosunoglu, Ramon Bertran, Peter Figuli, Christian Jacobi, Nikolaos Papandreou, Haris Pozidis, Anthony Saporito, Andrew Sica, and Elpida Tzortzatos. AI accelerator on IBM Telum processor: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1012–1028, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450386104. doi: 10.1145/3470496.3533042. URL https://doi.org/10.1145/3470496.3533042.

[93] Intel built-in accelerators. URL https://www.supermicro.com/en/accelerators/intel/built-in-on-demand.

[94] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021.

[95] ODSA-BoW specifications. URL https://opencomputeproject.github.io/ODSA-BoW/bow_specification.html.

[96] UCIe 1.1 specifications. URL https://www.uciexpress.org/specifications.

[97] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2013.

[98] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. Drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[99] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017.

[100] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[101] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[102] Intel vtune profiler, . URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[103] CloudSuite | A Benchmark Suite for Cloud Services. URL https://www.cloudsuite.ch/.

[104] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[105] Sean Kinzer, Soroush Ghodrati, Rohan Mahapatra, Byung Hoon Ahn, Edwin Mascarenhas, Xiaolong Li, Janarbek Matai, Liang Zhang, and Hadi Esmaeilzadeh. Restoring the broken covenant between compilers and deep learning accelerators. *arXiv*, 2023.

[106] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, pages 3389–3400, 2018.

[107] Linux kernel drm-gem drivers, . URL https://www.kernel.org/doc/html/latest/gpu/drm-mm.html.

[108] Lwn.net article on gem, . URL https://lwn.net/Articles/283798/.

[109] dma-buf. URL https://docs.kernel.org/driver-api/dma-buf.html.

[110] Napi. URL https://www.kernel.org/doc/html/next/networking/napi.html.

[111] Xilinx u30 vcu, . URL https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds970-u30.pdf.

[112] Xilinx vitis dsp library, . URL https://xilinx.github.io/Vitis_Libraries/dsp/2022.1/index.html.

[113] Xilinx vitis data analytics library, . URL https://xilinx.github.io/Vitis_Libraries/data_analytics/2022.1/index.html.

[114] Xilinx vitis security library, . URL https://xilinx.github.io/Vitis_Libraries/security/2022.1/index.html.

[115] Xilinx vitis data compression library, . URL https://xilinx.github.io/Vitis_Libraries/data_compression/2022.1/index.html.

[116] Xilinx vitis database library, . URL https://xilinx.github.io/Vitis_Libraries/database/2022.1/index.html.

[117] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[118] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. Freepdk: An open-source variation-aware design kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, pages 173–174, 2007. doi: 10.1109/MSE.2007.44.

[119] Amazon EC2 F1 Instances. URL https://aws.amazon.com/ec2/instance-types/f1/.

[120] Aws vt1 instance, . URL https://xilinx.github.io/video-sdk/v1.5/getting_started_on_vt1.html.

[121] Xilinx vitis libraries, . URL https://xilinx.github.io/Vitis_Libraries/.

[122] Intel rapl, . URL https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[123] Broadcom pex88000 managed pci express 4.0 switches. URL https://www.broadcom.com/products/pcie-switches-bridges/expressfabric.

[124] Noah Beck, Sean White, Milam Paraschou, and Samuel Naffziger. Zeppelin: An soc for multichip architectures. In *IEEE ISSCC*, 2018.

[125] Transformers based named entity recognition models. URL https://huggingface.co/Jean-Baptiste/roberta-large-ner-english.

[126] Hadi Esmaeilzadeh, Soroush Ghodrati, Jie Gu, Shiyu Guo, Andrew B. Kahng, Joon Kyung Kim, Sean Kinzer, Rohan Mahapatra, Susmita Dey Manasi, Edwin Mascarenhas, Sachin S. Sapatnekar, Ravi Varadarajan, Zhiang Wang, Hanyang Xu, Brahmendra Reddy Yatham, and Ziqing Zeng. Verigood-ml: An open-source flow for automated ml hardware synthesis. In *ICCAD*, 2021.

[127] Intel cascade lake, . URL https://ark.intel.com/content/www/us/en/ark/products/192447/intel-xeon-gold-6252-processor-35-75m-cache-2-10-ghz.html.

[128] Intel ice lake, . URL https://ark.intel.com/content/www/us/en/ark/products/212456/intel-xeon-gold-6348-processor-42m-cache-2-60-ghz.html.

[129] Intel sapphire rapids, . URL https://ark.intel.com/content/www/us/en/ark/products/231750/intel-xeon-platinum-8468h-processor-105m-cache-2-10-ghz.html.

[130] Gpudirect. URL https://developer.nvidia.com/gpudirect.

[131] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. In *ISCA*, 2016.

[132] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless operating system integration of Peer-to-Peer DMA between SSDs and GPUs. In *ATC*, 2017.

[133] Yu-Chia Liu and Hung-Wei Tseng. Nds: N-dimensional storage. In *MICRO*, 2021.

[134] Ryo Nakamura, Yohei Kuga, and Kunio Akashi. How beneficial is peer-to-peer dma? In *APSys*, 2020.

[135] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In *ASPLOS*, 2020.

[136] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. Flexdriver: A network driver for your accelerator. In *ASPLOS*, 2022.

[137] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Architecture support for accelerator-rich cmps. In *Proceedings of the 49th Annual Design Automation Conference*, 2012.

[138] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. In *SIGMETRICS*, 2022.

[139] Rohan Mahapatra, Byung Hoon Ahn, Shu-Ting Wang, Hanyang Xu, and Hadi Esmaeilzadeh. Exploring efficient ml-based scheduler for microservices in heterogenous clusters. In *Machine Learning for Computer Architecture and Systems 2022*, 2022.

[140] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[141] Intel data streaming accelerator, . URL https://www.intel.com/content/www/us/en/develop/articles/intel-data-streaming-accelerator-architecture-specification.html.

[142] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. Dcs: A fast and scalable device-centric server architecture. In *MICRO*, 2015.

[143] Dongup Kwon, Jaehyung Ahn, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim. Dcs-ctrl: A fast and flexible device-control mechanism for device-centric server architecture. In *ISCA*, 2018.

[144] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *ASPLOS*, 2020.

[145] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO*, 2021.

[146] Sizhuo Zhang, Hari Angepat, and Derek Chiou. Hgum: Messaging framework for hardware accelerators. In *ReConFig*, 2017.

[147] Johan Peltenburg, Jeroen Van Straten, Lars Wijtemans, Lars Van Leeuwen, Zaid Al-Ars, and Peter Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *FPL*.

[148] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12), 2021.

[149] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *ISCA*, 2022.

[150] Nvidia dali. URL https://developer.nvidia.com/dali.

[151] Rui Hou, Tao Jiang, Liuhang Zhang, Pengfei Qi, Jianbo Dong, Haibin Wang, Xiongli Gu, and Shujie Zhang. Cost effective data center servers. In *HPCA*, 2013.

[152] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stens-

land, and Carsten Griwodz. Smartio: Zero-overhead device sharing through pcie networking. *ACM Trans. Comput. Syst.*, 38(1–2), 2021.

[153] Gigaio fabrex. URL https://gigaio.com/wp-content/uploads/2022/01/GigaIO-FabreX-composability_v1-1.pdf.

[154] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct universal access: Making data center resources available to FPGA. In *NSDI*, 2019.

[155] Bill Dally. Accelerator clusters: the new supercomputer. In *HOTI*, 2023.