Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin

Yakun Sophia Shao[§]   Sam (Likun) Xi   Vijayalakshmi Srinivasan[†]   Gu-Yeon Wei   David Brooks

NVIDIA Research[§]   Harvard University   IBM Research[†]

sshao@nvidia.com   {samxi,guyeon,dbrooks}@eecs.harvard.edu   viji@us.ibm.com

Abstract—**Increasing demand for power-efficient, high-performance computing has spurred a growing number and diversity of hardware accelerators in mobile and server Systems on Chip (SoCs). This paper makes the case that the co-design of the accelerator microarchitecture with the system in which it belongs is critical to balanced, efficient accelerator microarchitectures. We find that data movement and coherence management for accelerators are significant yet often unaccounted components of total accelerator runtime, resulting in misleading performance predictions and inefficient accelerator designs. To explore the design space of accelerator-system co-design, we develop gem5-Aladdin, an SoC simulator that captures dynamic interactions between accelerators and the SoC platform, and validate it to within 6% against real hardware. Our co-design studies show that the optimal energy-delay-product (EDP) of an accelerator microarchitecture can improve by up to 7.4× when system-level effects are considered compared to optimizing accelerators in isolation.**

I. Introduction

In the era of diminishing returns from technology scaling, hardware acceleration is widely used to gain performance, power, and energy improvements [1]. Accelerators are now an integral component in modern SoCs, powering a variety of applications like video decoding, image processing, cryptography, machine learning, and more [2], [3], [4], [5], [6].

Accelerators are often designed as standalone IP blocks that communicate with the rest of the system using a Direct Memory Access (DMA) interface. This modularity simplifies IP design and integration with the rest of the system, leaving tasks like data movement and coherency management to software device drivers. As a result, the costs of these overheads are hard to predict and accommodate for at accelerator design time. Our detailed characterization of accelerator behavior shows that the combination of just these two effects can occupy over 40% of the total runtime. Hence, when it comes to accelerator design, architects must take a holistic view of how they interact in the overall system, rather than designing them in isolation.

Fundamentally, all systems should be designed in a way that balances the bandwidth of the memory interface with the amount of compute throughput. An overly aggressive

design will have more computational units than the memory interface can supply, leading to wasted hardware and additional leakage power. We identify three major system-level considerations that strongly affect accelerator design: local memory interface, cache coherency management, and behavior under shared resource contention.

The typical local memory interface is DMA, a push-based system that requires software to setup bulk transfers and manage coherency. An alternative is to embed a hardware-managed cache with the accelerator design, leading to a fine-grained, pull-based memory system that loads data on-demand and transparently handles coherency state. Despite these conveniences, caches are rarely used in accelerators due to hardware overheads leading to power and area penalties. However, there has been growing interest from industry in providing coherent accelerator cache interfaces [7], [8], [9] for the ease of programmability. We investigate the system-level considerations for both approaches to understand when each is preferable.

Such studies require detailed simulation infrastructure for heterogeneous accelerator-rich platforms like SoCs. There is a wide selection of CPU simulators [10], [11], [12] and standalone accelerator simulators like Aladdin [13]. However, existing SoC simulators are unable to model dynamic interactions between accelerators and the memory system [14]. In this paper, we introduce gem5-aladdin, which integrates the gem5 system simulator with the Aladdin accelerator simulator to enable simulation of SoCs with complex accelerator-system interactions. We validate gem5-aladdin against the Xilinx Zynq platform and achieve less than 6% error.

We demonstrate that co-designing accelerators with system-level considerations has two major ramifications for accelerator microarchitectures that are not yet fully understood in the literature. First, datapaths should be less aggressively parallel, which results in more balanced designs and improved energy efficiency compared to accelerators designed in isolation. Second, the choice of local memory interfaces is highly dependent on the dynamic memory characteristics of the accelerated workload, the system architecture, and the desired power/performance targets. We show that accelerator-system co-design can improve energy-delay-product by up to 7.4× and on average 2.2×.
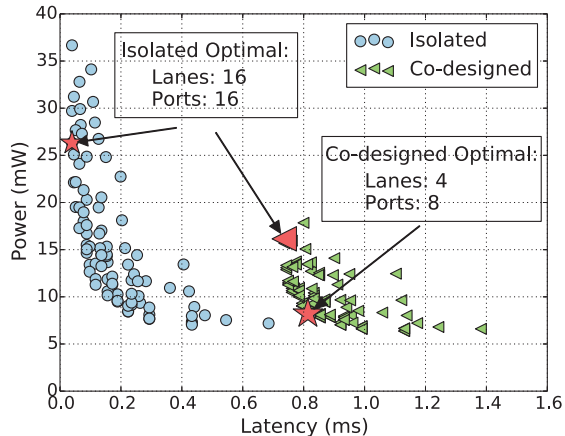

Fig. 1: Design space exploration for `stencil3d` for both isolated and co-designed cases.

II. MOTIVATION AND BACKGROUND

In this paper, we use the term "accelerator" to refer to an application-specific hardware block. These accelerators are comprised of multiple customized datapath lanes, and customized local memories. Each lane is a chain of functional units controlled by finite state machines. When the local memory is comprised of scratchpads, each scratchpad can be partitioned into smaller arrays to increase memory bandwidth to the lanes. Such accelerators are representative of recent academic proposals [3], [15], [16], [17], [18], [19], [20] and commercial designs [21], [22], [23].

A. Co-design: A Motivating Example

To demonstrate the differences between isolated vs. co-designed accelerators, we perform a design sweep exploration for both scenarios on a 3D stencil kernel. We sweep compute parallelism and scratchpad partitioning. Compute parallelism is described by the number of datapath lanes. Figure 1 shows these two design spaces.

We consider an accelerator designed in isolation to be one that focuses design optimization on the computation phase. This design space (blue circles) leans towards more parallel, power-hungry designs, as exemplified by the isolated energy-delay-product (EDP) optimal design point. But if we account for effects like initial data movement, the design space (green triangles) shifts dramatically towards the lower right, preferring less parallel designs at lower power. If we take the isolated EDP optimal design and then apply these system effects, we find that it is quite different from the co-designed EDP optimal point. Unaccounted data movement becomes a significant part of total runtime, making aggressively parallel datapaths unnecessary.

B. Typical CPU-Accelerator Communication

The existence of the difference between the two design spaces is due to how CPUs and accelerators traditionally
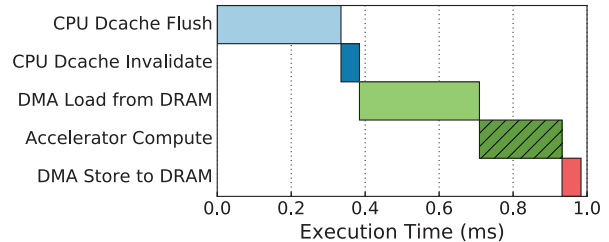
(a) `md-knn` execution time on the Zynq platform
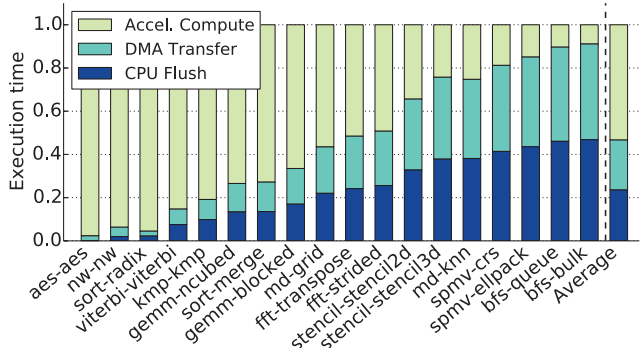
(b) Breakdown of flush, DMA, and compute time in MachSuite for 16-way parallel designs.

Fig. 2: Data movement overheads on MachSuite.

communicate data. In this typical flow, DMA is the transfer mechanism, but typical DMA implementations can only access main memory or LLC, so the CPU first flushes all input data from private caches and invalidates the region used to store return data [24]. Then it programs a DMA transaction into the DMA engine and initiates the transfer. The accelerator begins execution after receiving all the data and streams its output data via DMA back to main memory when it is done. The CPU, having invalidated that memory region from its caches, can now access the return data correctly.

For many benchmarks, this flow works quite well. DMA is quite efficient at copying large blocks of data, and accelerators whose compute-to-memory ratios are large are well served by DMA. However, for other workloads with more irregular memory access patterns, this flow can impose severe overheads, because the accelerator must wait to receive all the data before it can begin computation. As an example, Figure 2a shows the execution timeline for a 16-lane implementation of an `md-knn` accelerator (a k-nearest-neighbor molecular dynamics), running on a Xilinx Zynq platform. As shown, the accelerator's computation only occupies about 25% of the total cycles, with the rest of the time spent on preparing and moving data. We expanded this study in simulation for all the MachSuite benchmarks [25] and find that about half of them are compute-bound and the other half data-movement-bound, as shown in Figure 2b.

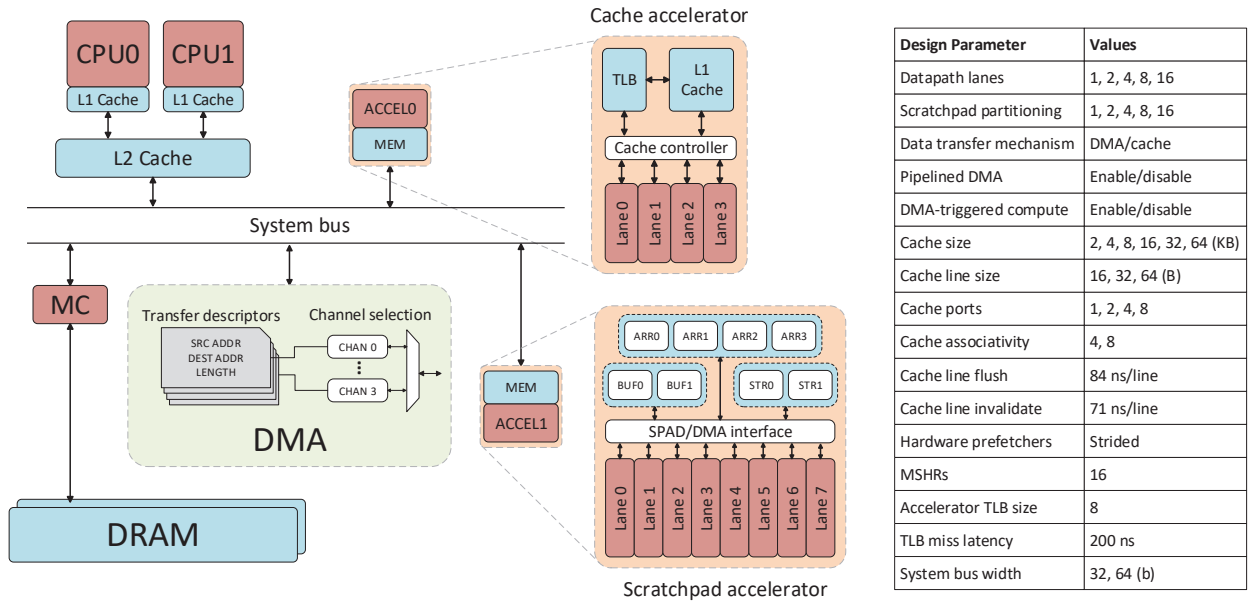| Design Parameter	Values
Datapath lanes	1, 2, 4, 8, 16
Scratchpad partitioning	1, 2, 4, 8, 16
Data transfer mechanism	DMA/cache
Pipelined DMA	Enable/disable
DMA-triggered compute	Enable/disable
Cache size	2, 4, 8, 16, 32, 64 (KB)
Cache line size	16, 32, 64 (B)
Cache ports	1, 2, 4, 8
Cache associativity	4, 8
Cache line flush	84 ns/line
Cache line invalidate	71 ns/line
Hardware prefetchers	Strided
MSHRs	16
Accelerator TLB size	8
TLB miss latency	200 ns
System bus width	32, 64 (b)

Fig. 3: An example SoC that can be modeled using gem5-Aladdin. The table on the right shows the set of design parameters that we swept in this work and their values; this is just a small subset of what can be configured.

Clearly, DMA is not an optimal solution for some workloads. One alternative, as mentioned earlier, is to replace push-based DMA with pull-based hardware-managed caches. In recent years, the scope of workloads that we desire to accelerate has widened from dense computational kernels to more irregular applications which could benefit from a less rigid memory system. Although caches have seldom been used for accelerators, the increased workload diversity motivates a more comprehensive study of new CPU-accelerator communication strategies.

III. MODELING INFRASTRUCTURE

Figure 3 shows an example of an SoC, including general-purpose cores, memory controllers, a DMA engine, and different types of fixed-function accelerators, all of which are connected through the system bus. In order to understand how system-level effects impact the behavior of accelerators, we need simulation infrastructures that can model these heterogeneous systems. In this work, we integrate Aladdin with the gem5 system simulator [10], a widely-used system simulator with configurable CPUs and memory systems.

gem5-aladdin models interactions between accelerators and CPUs, DMA, hardware-managed caches, and virtual memory. All of these features have implications on how the accelerator behaves and in the following sections, we describe how each is modeled.

A. Overview

For the experiments in this paper, we run gem5-aladdin in syscall emulation mode because it is sufficient to capture the effects of our system-level considerations on performance and power. Full-system simulation would enable us to model operating system effects, but most are beyond the scope of this study. Some interactions with the operating system, such as device driver to hardware interactions, are characterized through real hardware measurements and analytically included in our models. Finally, syscall emulation is much faster than full system simulation, easing rapid design space exploration.

B. Accelerator Modeling

The Aladdin accelerator simulator [13] takes a first step towards modeling the power, performance, and cycle-level activity of standalone, fixed-function accelerators without needing to generate RTL. Aladdin is a trace-based accelerator simulator that profiles the dynamic execution of a program and constructs a dynamic data dependence graph (DDDG) as a dataflow representation of an accelerator. The vertices in the DDDG are LLVM IR instructions, and the edges represent true dependences between operations. Aladdin then applies common accelerator design optimizations and schedules the graph for execution through a breadth-first traversal, while accounting for user-defined hardware constraints. Aladdin was validated to be within 7% accuracy compared to standalone, RTL accelerator designs.

However, Aladdin only focuses on the standalone datapath and local memories. It assumes that all data has been pre-loaded into the local scratchpads. This skips the modeling of any interactions between accelerators and the rest of the system in which they belong.

C. DMA Engine

DMA is a software managed mechanism for transferring bulk data without CPU intervention. To set up a transaction, the programmer constructs a DMA transfer descriptor that contains the source and destination memory addresses along with the size of the transfer. Multiple descriptors can be constructed and connected through a linked list. When all descriptors are ready, the programmer initiates the transfer by writing the address of the head of the descriptor linked list into a hardware DMA engine's control register. The DMA engine then fetches and services these descriptors one by one. Meanwhile, the CPU is free to perform other work.

In gem5-Aladdin, accelerators can invoke the DMA engine already present in gem5. To do so, a programmer inserts calls to special `dmaLoad` and `dmaStore` inside the accelerated function with the appropriate source, destination, and size arguments. When the function is traced by Aladdin, Aladdin will identify these calls as DMA operations and issue the request to the gem5 DMA engine. As part of the DMA engine, we include an analytical model to account for cache flush and invalidation latency, using the measured numbers mentioned in Section IV-B1.

D. Caches and Virtual Memory

For the accelerator caches, we use gem5's classic cache model along with a basic MOESI cache coherence protocol. When Aladdin sees a memory access that is mapped to a cache, it sends a request through a cache port to its local cache. Aladdin will receive a callback from the cache hierarchy when the request is completed. To support virtual memory, we implement a special Aladdin TLB model. We do not use gem5's existing TLB models for two reasons. First, the existing TLB models are tied to particular ISAs, which do not pertain to accelerators [26]. Second, as a trace-driven simulator, the trace address that Aladdin originally uses does not directly map to the simulated address space that CPU is accessing. To maintain correct memory access behavior, our custom TLB model translates the *trace* address to a simulated *virtual* memory address and then to a simulated *physical* address. TLB misses and page table walks are modeled with a pre-characterized miss penalty.

E. CPU-Accelerator Interface

On the CPU, a simulated user program can invoke an attached accelerator through the `ioctl` system call, a system call widely used in practice for arbitrary communication with devices. In the `ioctl` emulation code, we assign a special file descriptor value for Aladdin and use command numbers to refer to individual accelerators. When the accelerator finishes, it writes to a shared pointer between the CPU and the accelerator. The CPU will see the update due to cache coherence. After invoking the accelerator, the CPU can either spin wait for the status to update or continue to do other
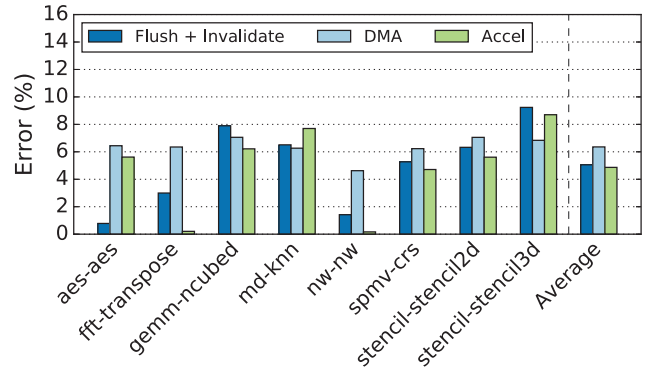

Fig. 4: Error between Zedboard and gem5-Aladdin cycles.

work, periodically checking the status variable to see if the accelerator is completed.

Sharing virtual memory between CPUs and accelerators means that any mismatches in memory consistency models must be resolved. In our experiments, we handle this by strictly limiting pages of memory that the accelerator may access and enforcing mutual exclusion on these pages. For our simple synchronization primitive, the accelerator issues an `mfence` before signaling to the CPU that it is finished through the shared pointer.

F. Performance Validation

We have validated gem5-Aladdin's performance models using the Zynq Zedboard for a subset of the MachSuite benchmark suite. For each benchmark, we implement the AXI4-Stream interface to transfer data via Xilinx's DMA IP blocks. Accelerator RTL is generated using Vivado HLS 2015.1. To maintain a consistent view of the model, we use HLS without specifying any additional design optimizations, so Vivado HLS generates a default design whose parameters we then match in Aladdin.

The complete system (including the DMA engine, accelerators, crossbars, etc.) is implemented in in Vivado Design Suite 2015.1. Software running on the CPU first initializes all devices in the system and generates the accelerator input data. Then it performs the necessary cache flushes and invalidates and starts the DMA transfer. The accelerator automatically begins computation when the DMA transfer is complete.

To measure performance, we instrument this code using cycle counters on the A9 CPUs. Because we cannot directly measure the DMA transfer time, we include logic analyzers in the synthesized system to capture waveforms using Xilinx tools during execution. Most benchmarks were implemented on a 10ns clock; a few used slower clocks for timing reasons.

The results of our validation are shown in Figure 4. Our DMA performance model achieves 6.4% average error across this suite of benchmarks, while Aladdin achieves 5% average error, and the flush and invalidation analytical model

achieves 5% average error. These results demonstrate the ability of gem5-Aladdin to model a wide range of accelerator workloads accurately for both the accelerated kernels and important system-level considerations.

1) Validation omissions: Our validation focuses on the features required by our DMA techniques: cache flushes and invalidates, DMA transfer time, and accelerator runtime. In general, we validated as much of the new additions as we could. Below are the components this work does not validate and our reasons for omitting them.

- CPU performance models: Existing work by Gutierrez et al. has already produced an accurate gem5 CPU model for the ARM A9 core[27], and gem5-Aladdin uses that validated model.
- Power model: All power results represent only the accelerator power. We do not account for CPU power in any of our results. We use the same validated Aladdin's power models with TSMC 40nm technology.
- Cache: To the best of our knowledge, there is no existing IP block available on Zynq such that we could implement a cache controller on the programmable fabric. Furthermore, we never modified gem5's cache models.

IV. MEMORY SYSTEM OPPORTUNITIES

In this section, we will discuss the primary design considerations when deciding whether to use a DMA- or cache-based memory system for an accelerator. Because baseline DMA leaves much room for improvement, we will also apply two optimizations to DMA. We will then describe design considerations specific to cache-based accelerators. Finally, we will evaluate the performance of both memory systems for a set of representative benchmarks.

A. Primary design considerations

First, we compare and contrast DMA and caches across the three system-level considerations mentioned earlier: push vs. pull, data movement granularity, management of coherency, and behavior under shared resource contention.

Push vs. Pull: DMA is designed for efficient bulk data transfer where the data requirements of the program are well known a priori. This works well for streaming applications and applications with high compute-to-memory ratios. However, applications with more irregular memory access patterns, such as indirect memory accesses, can suffer without an on-demand memory system like a cache. In addition, because caches have the feature of automatic cache line replacement, a cache can often afford to be smaller than a scratchpad that must hold all the data.

Data Movement Granularity: Because DMA is software controlled, the overheads of setting up a transaction are usually amortized over a large bulk transfer. In contrast, caches pull in data at cache line granularity, enabling fine-grained overlap between compute and data movement. Although fine-grained DMA is possible, each new transactions adds additional overheads. On the other hand, caches must perform tag

comparisons, replacements, and address translations, which make them inefficient for bulk data movement.

Cache Coherence Management: DMA engines typically can only access main memory or last level cache. Therefore, the programmer must conservatively flush any data the accelerator may read out of private caches. Figure 2b shows that on average, accelerators employing traditional DMA spend 20% of their total cycles on cache flushes. The flush is typically performed by software because DMA engines rarely participate in coherency (although there have been exceptions, like IBM Cell [28]). In contrast, hardware-managed caches handle all of this complexity transparently at the cost of additional hardware.

Shared Resource Contention: In a real scenario where resources like the main system interconnect and main memory are shared across multiple agents, invariably a DMA operation or cache fill will stall to allow another process to make progress. A coarse-grained mechanism like DMA will be affected much more by shared resource contention because the accelerator usually waits for the entire transfer to complete. In comparison, fine-grained memory accesses like cache fills are less likely to contend due to their smaller size, and hit-under-miss allows other independent operations to proceed even while an earlier cache load or store missed.

B. DMA Optimizations

In this section, we improve the baseline DMA method by overlapping various stages of the process. We will examine two DMA latency optimizations: pipelined DMA and DMA-triggered computation, which are depicted in Figure 5.

1) Pipelined DMA: Pipelined DMA reduces latency by dividing the flush and DMA operations into page sized blocks and overlapping the DMA of block b with the flush of block $b + 1$. We choose page size granularity to optimize for DRAM row buffer hits. In the best case, we can hide all but 4KB of the flush latency. Note that the correctness of this optimization is ensured by never starting a DMA block before its flush has completed.

Cache line flush latency varies across ISAs and implementations. For example, we characterized the flush throughput on the Zedboard's Cortex A9 CPU to be one cache line per 56 cycles at 667MHz. To achieve optimal pipelining and avoid bubbles, we want to match the flush and DMA latencies of a 4KB transaction. On the Zedboard, this is achieved with an accelerator clock frequency of 100MHz, which is why we use this frequency for the rest of our experiments.

Breaking up a large flush and DMA operation introduces additional overheads. The DMA engine must fetch new metadata from main memory for every block, and the CPU must synchronize flushes with dependent DMA operations. For this, we add a fixed 40 cycle delay to every DMA transaction, also based on characterization. At 100MHz, this accounts for metadata reads (4 cycles), the one-way latency
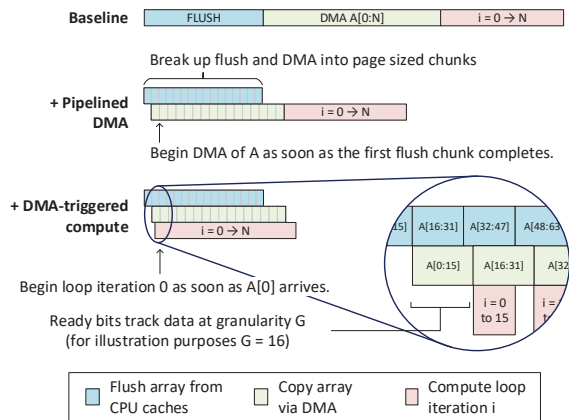
Fig. 5: A demonstration of the DMA latency reduction techniques.

of initiating DMA from the CPU (17 cycles), and additional CPU cycles spent on housekeeping actions.

2) DMA-Triggered Computation: Even with pipelined DMA, the accelerator still must wait for the entire DMA transaction to finish before it can start. To overcome this, we augment our accelerators with full/empty-bits, which are often used in producer-consumer situations to indicate that data is ready [29]. In our designs, we track data at cache line granularity to be consistent with the preceding flush operations (which operate on cache lines). Full/empty bits are stored in a separate SRAM structure and indexed by a slice of the load address. With full/empty bits, the accelerator immediately begins computation without waiting for DMA to complete until it reaches a load. A load accesses both the full/empty bit arrays and the data arrays in parallel and returns the data if the full/empty bit is 1. If not, the control logic stalls the datapath until the DMA engine eventually fills that data and sets the full/empty bit. Note that double-buffering could be implemented in this scheme by tracking the granularity of data transfer at half the array size instead of cache line size, without any manual intervention. If an accelerator has multiple datapath lanes, other lanes are free to proceed even while some are blocked.

C. DMA Evaluation

To quantify the performance improvements from each of the techniques described, we start from the baseline design and cumulatively apply our DMA optimizations. From execution traces, we break down the runtime into four parts based on how cycles are spent: flush-only time, DMA/flush time, compute/DMA time, and compute-only time. Flush-only and compute-only are self-explanatory; compute/DMA time includes all cycles when compute and DMA are overlapped, while DMA/flush includes all cycles when DMA and flush *but not compute* are running.

Increasing the parallelism of accelerator datapaths through additional datapath lanes and memory partitioning is a widely

used and effective way to achieve higher performance at the cost of greater area and power. However, the presence of memory movement imposes an upper bound on achieveable speedup, and our DMA optimizations will affect realized speedup as well. To understand how parallel an accelerator must be in order to approach this upper bound, we take all the optimizations, sweep the parallelism of the accelerator datapath, and analyze the speedups realized.

1) Performance gains from DMA optimizations: The performance improvements from each optimization are shown in 6a. For brevity, we only present a subset of benchmarks whose DMA times spans the range shown in Figure 2b. We fix the parallelism of all accelerators to four datapath lanes.

We immediately observe that in the baseline design, flush-only time is a significant fraction of the total execution time. Pipelined DMA is thus shown to be very effective, almost completely eliminating flush-only time for all the benchmarks shown. This is because the benefits of pipelined DMA are only dependent on the amount of data transferred and not on the memory characteristics of the application.

DMA-triggered computation is able to improve performance even more, but its effectiveness clearly varies across workloads. It is most effective when a benchmark exhibits some level of streaming behavior. For example, stencil2d uses a 3x3 kernel and thus only requires the first three rows of the input matrix to arrive before it can start computation, so ready bits recover a significant amount of performance. A similar logic applies to md-knn – in fact, ready bits are so effective here that with just four datapath lanes, we achieve 99% compute/DMA overlap. This is in contrast to fft-transpose, where each unit of work requires eight loads strided across the entire input arrays. This is not a streaming memory access pattern and so DMA-triggered compute is ineffective.

2) Impact of parallelism on DMA optimizations: The results of sweeping accelerator parallelism, while applying all the DMA optimizations, is shown in Figure 6b. This figure demonstrates two points.

First, on several workloads, if there is enough parallelism, the entire computation can be overlapped with DMA. This means that without reducing flush or DMA time, no more speedup is achievable. Benchmarks without this property either have very little data to transfer to begin with (aes) or are so serial that they don't benefit from data parallelism in the first place (nw).

Second, increased parallelism has no effect on the amount of compute-DMA overlap. This is due to the *serial data arrival* effect: no matter how parallel a datapath is, DMA will always copy data sequentially starting with the first byte, and until that critical first byte of data arrives, no compute can start. As our DMA engine already fully utilizes the available bus bandwidth, this data cannot arrive any faster, and therefore compute also cannot be overlapped any more.

In conclusion, these sweeps show that memory movement,
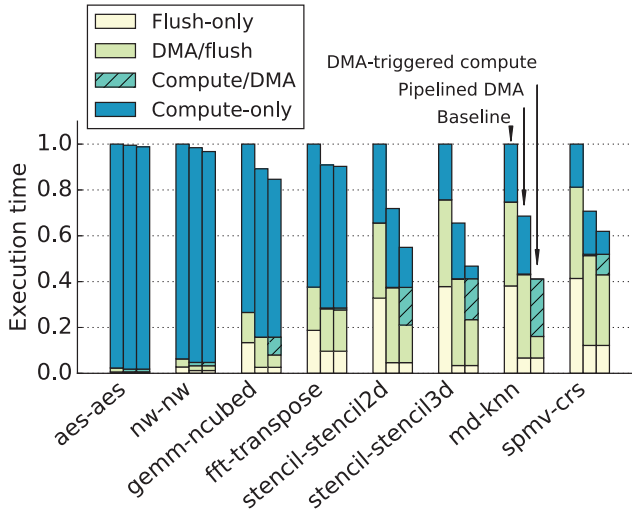
(a) Performance improvements from each technique.
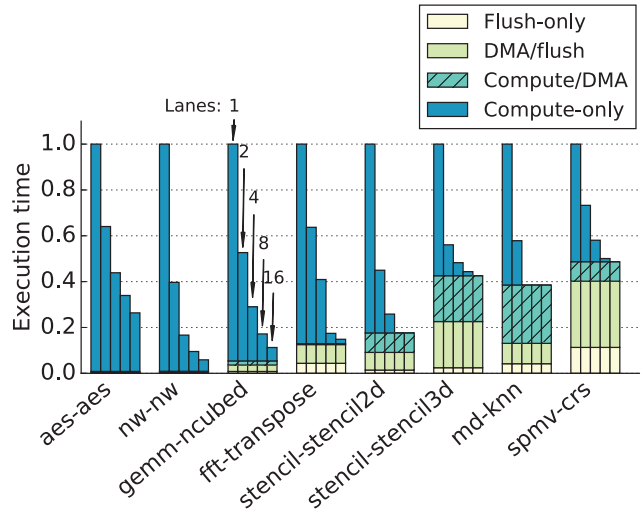
(b) Effect of parallelism on performance gains.

Fig. 6: Cumulatively applying each technique reduces the additional cycles spent on DMA, with some benchmarks seeing more benefit than others. After applying all techniques, increasing parallelism through loop unrolling reduces compute cycles until near-complete overlap is achieved, causing performance to saturate.

not compute, has become a significant bottleneck, and only accelerating computation will quickly bring diminishing returns. In fact, Figure 6b shows that for many benchmarks, we can achieve the upper bound performance with relatively fewer datapath lanes. As a result, to continue to get better performance, we must somehow further overlap computation with data by overcoming the serial data arrival effect, motivating the study of fine-grained, *on-demand* memory systems.

D. Cache-Based Accelerators

In a cache-based accelerator, one of the most important questions is how to handle variable latency memory accesses in a statically scheduled datapath. The simplest way is to stall the entire datapath until the miss resolves, but this significantly hurts performance. Techniques like multithreaded accelerators have been proposed in the CAD community to hide cache miss latency [30], [31], but these require additional resources to store thread contexts.

We choose a simpler cache miss handling scheme. Accelerators are typically designed with multiple parallel lanes. When a cache miss happens in one of the lanes, only that lane is stalled until the miss resolves. Other lanes are free to continue. We include MSHRs to enable hit-under-miss and multiple outstanding misses. Any lane with a dependence on a result from a blocked lane is also blocked via control logic mechanisms. This scheme lets independent computation proceed while waiting for the missed data to be returned without requiring storage for thread contexts. When lanes

are finished executing, they must wait and synchronize with all other lanes before the next iteration can begin.

Another important design choice is what data is cached. In our experiments, only data that must be eventually shared with the rest of the system is sent through the cache, and local scratchpads are used for private intermediate data. For example, nw uses an internal score matrix to align DNA sequences. This matrix is kept in local scratchpads.

E. Cache Evaluation

In this section, we will analyze the impact of datapath parallelism on cache-based accelerator performance. We decompose total execution time into processing time, latency time, and memory bandwidth time, using a similar technique as Burger et al. [32]. Each component is the additional execution time after applying a realistic constraint to a memory system parameter. To briefly summarize:

1) Processing time: assume memory accesses are single-cycle and always hit.
2) Latency time: allow memory accesses to miss in the cache, but the system bus has unlimited bandwidth to service cache fills.
3) Bandwidth time: constrain the system bus width to 32 bits, thus limiting the rate at which cache fill requests can be serviced.

1) Impact of Datapath Parallelism: Figure 7 shows how the performance of cache-based accelerators scales with datapath parallelism. In this set of experiments, we first sweep cache sizes to find the smallest cache at which performance saturates for each benchmark. This is labeled at the top
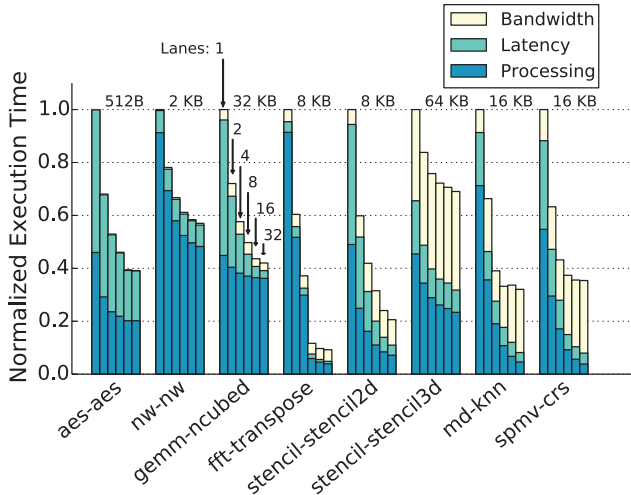
Fig. 7: Effect of datapath parallelism on cache-based accelerator performance.

of each group of bars. The datapath parallelism sweep is performed with this cache size per benchmark.

Naturally, we observe that processing time decreases with increased parallelism, as expected. However, parallelism also improves *latency* time, which is in contrast to the DMA experiments where parallelism did not affect flush or DMA time. This is because caches are a fine-grained pull-based memory system, and increased datapath parallelism also increases memory-level parallelism (more memory accesses per cycle). Furthermore, the fine granularity more effectively masks cache miss latency with computation, thereby decreasing latency time.

On the other hand, more parallelism does not improve bandwidth time due to increased memory bandwidth pressure. In fact, bandwidth time becomes a larger fraction of total execution time as we go to increasingly parallel designs. For example, the performance of spmv-crs and md-knn is eventually bottlenecked by bandwidth, even though the increased memory level parallelism improves both processing and latency time. Accelerators that are designed without consideration of the available memory bandwidth in the SoC are likely to be over-designed, provisioning more functional units than can be fed with data by the system.

V. ACCELERATOR DESIGN CHOICES

Thus far, we have discussed in detail how the performance of accelerated workloads changes when connected to two different memory systems, scratchpad with DMA and hardware-managed caches. However, it has been unclear when to select one over the other. Performance is not the only goal as well; accelerator designers especially must balance performance targets against power and energy constraints. It is also unclear how differently one must think about designing accelerators

when system-level effects like data movement and its mechanisms are considered. In this section, we will shed light on the DMA vs. cache question as well as illustrate that without consideration for data movement, accelerator designers are highly to overprovision and underutilize their accelerators.

A. DMA vs. Caches

One of the earliest decisions a designer needs to make is decide whether private scratchpads with DMA or hardware-managed caches is a better fit for the application at hand. In this experiment, we performed a comprehensive design space sweep for all the parameters listed in Figure 3 for all of the MachSuite benchmarks. We show the resulting Pareto optimal design curves, distinguished by memory system type, in Figure 8. For brevity, we only show eight benchmarks that span the range of design space characteristics observed. The energy-delay-product (EDP) optimal design point for each memory system is labeled with a star of the corresponding color. All DMA design points apply all the optimizations discussed in Section IV-B.

This experiment shows that some benchmarks umambiguously prefer scratchpads with DMA (on the left), some clearly are better with caches (on the right), and several work equally well with either (in the middle). We will briefly discuss each benchmark's behavior in turn.

aes-aes and nw-nw:

These two benchmarks always both perform better and use less power with DMA than with caches. They have have very regular access patterns, and importantly, they only require a small amount of data before computation can be triggered. In contrast, a cache-based memory system will first experience a TLB miss followed by cache misses, causing significant performance slowdown.

gemm-ncubed:

This benchmark, unlike the previous two, is actually able to match its DMA counterpart in performance. However, due to the various overheads of caches (tag lookups, TLB lookups, etc.), more power must be expended to reach this performance.

stencil-stencil2d:

Although DMA can always outperform the cache system on this benchmark, a cache-based design can actually achieve same performance with lower power. This is because the cache system can capture enough locality to use a *smaller* cache, whereas the scratchpad design must fit the entire data set into local memory.

stencil-stencil3d:

The 3D stencil kernel distinguishes itself from its 2D counterpart because the cache system can outperform the optimized DMA system at the cost of additional power. This
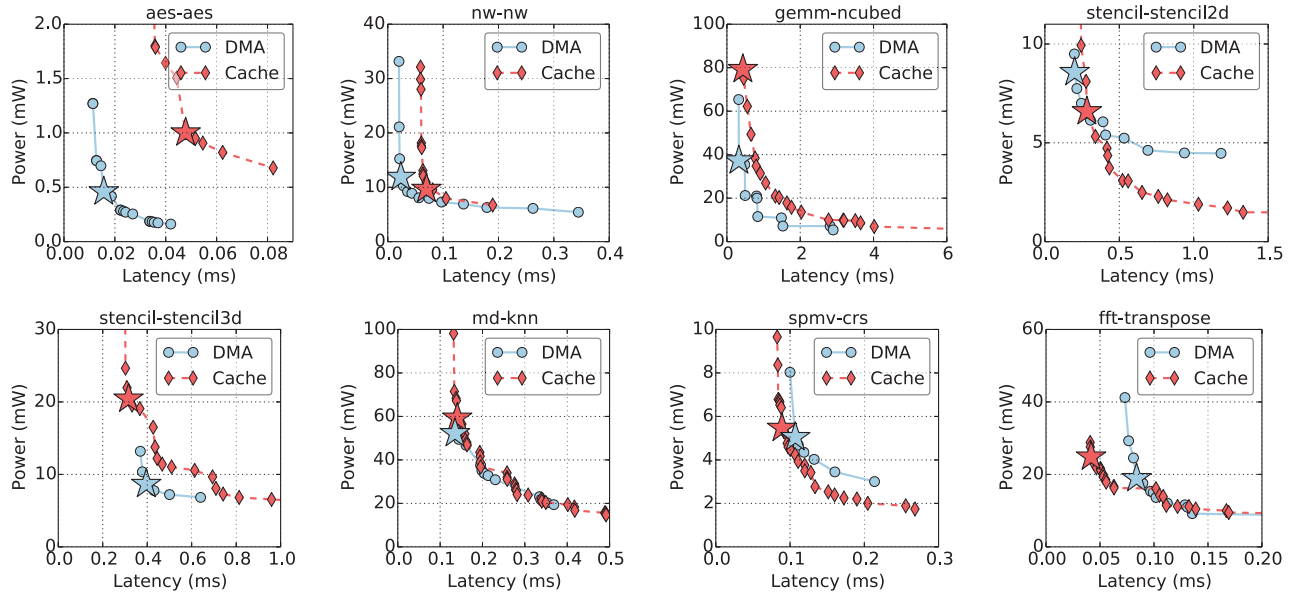
Fig. 8: Power-performance Pareto curves for DMA- and cache-based accelerators. EDP optimal design points are shown as stars. Benchmarks are ordered left-to-right, top-down by preference for a DMA-based vs. a cache-based memory system.

is because the kernel's three-dimensional memory access pattern creates nonuniform stride lengths, which are gracefully handled by the on-demand nature of a cache. In contrast, even the most optimized DMA design spends half of its execution time waiting for DMA and flush operations. The cost of this performance is $2\times$ to $3\times$ increased power.

`md-knn:`

`md-knn` is a very compute intensive application. In this benchmark, there are 12 FP multiplies per atom-to-atom interaction, so the power consumption of this benchmark is dominated by functional units rather than memory. Also, the optimized DMA system is able to fully overlap compute with data movement because full/empty bits are very effective in this benchmark. Figure 8 shows that the Pareto curves for cache and DMA designs largely overlap, demonstrating that either memory system can be an acceptable choice.

`spmv-crs:`

On this benchmark, a cache system is able to outperform a DMA system with lower power as well. This is due to the indirect memory accesses inherent to sparse matrix multiply algorithms, where the first set of loads provide the memory addresses for the next set that actually returns the data. Full/empty bits may not be effective on this benchmark if the data pointed to by a matrix index has not yet arrived, since DMA sends data sequentially, but a cache can fetch arbitrary memory locations. Caches thus eliminate most of the idling time, leading to better performance. Lower power on caches is achieved by being able to use a smaller cache than the scratchpads.

`fft-transpose:`

`fft-transpose` also performs better with caches than DMA but for slightly different reasons. There are no indirect memory accesses in this benchmark. Instead, the parallel implementation of this benchmark possesses a stride length of 512 bytes, meaning that each loop iteration (aka datapath lane) only reads eight bytes per 512 bytes of data. As a result, even with full/empty bits, a DMA system must supply nearly all of the data before the computation can begin, whereas this is not a problem for the cache system. Again, lower power is achieved by a smaller cache than scratchpads.

B. Design Decision Comparison

In addition to deciding the type of memory system to use, accelerator designers must also select local design parameters like the datapath parallelism and local memory size and bandwidth. In this section, we show that when system-level effects are considered, these parameters can change considerably compared to when an accelerator is designed in isolation.

To illustrate how optimal design parameters are affected by system-level effects, we consider the following design scenarios:

1) Baseline: design accelerators in isolation.
2) Co-designed DMA: use DMA to transport data over a 32-bit system bus.
3) Co-designed cache: use a hardware-managed cache for the accelerator's local memory.
4) Co-designed cache with 64-bit bus: Same as above, but we double the width of the system bus.

Parallelism (32 Lanes) · SRAM (512B) · Local BW (4 Ports) · Isolated Optimal · DMA Optimal w/ 64-bit Bus · Cache Optimal w/ 32-bit Bus · Cache Optimal w/ 64-bit Bus

(a) `aes-aes`

Parallelism (4 Lanes) · SRAM (34KB) · Local BW (16 Ports) · Isolated Optimal · DMA Optimal w/ 64-bit Bus · Cache Optimal w/ 32-bit Bus · Cache Optimal w/ 64-bit Bus

(b) `nw-nw`

Parallelism (32 Lanes) · SRAM (49KB) · Local BW (16 Ports) · Isolated Optimal · DMA Optimal w/ 64-bit Bus · Cache Optimal w/ 32-bit Bus · Cache Optimal w/ 64-bit Bus

(c) `gemm-ncubed`

Parallelism (32 Lanes) · SRAM (64KB) · Local BW (16 Ports) · Isolated Optimal · DMA Optimal w/ 64-bit Bus · Cache Optimal w/ 32-bit Bus · Cache Optimal w/ 64-bit Bus

(d) `stencil-stencil2d`

Parallelism (16 Lanes) · SRAM (128KB) · Local BW (16 Ports) · Isolated Optimal · DMA Optimal w/ 64-bit Bus · Cache Optimal w/ 32-bit Bus · Cache Optimal w/ 64-bit Bus

(e) `stencil-stencil3d`

Parallelism (32 Lanes) · SRAM (45KB) · Local BW (16 Ports) · Isolated Optimal · DMA Optimal w/ 64-bit Bus · Cache Optimal w/ 32-bit Bus · Cache Optimal w/ 64-bit Bus

(f) `md-knn`

Parallelism (32 Lanes) · SRAM (30KB) · Local BW (16 Ports) · Isolated Optimal · DMA Optimal w/ 64-bit Bus · Cache Optimal w/ 32-bit Bus · Cache Optimal w/ 64-bit Bus

(g) `spmv-crs`

Parallelism (32 Lanes) · SRAM (21KB) · Local BW (16 Ports) · Isolated Optimal · DMA Optimal w/ 64-bit Bus · Cache Optimal w/ 32-bit Bus · Cache Optimal w/ 64-bit Bus
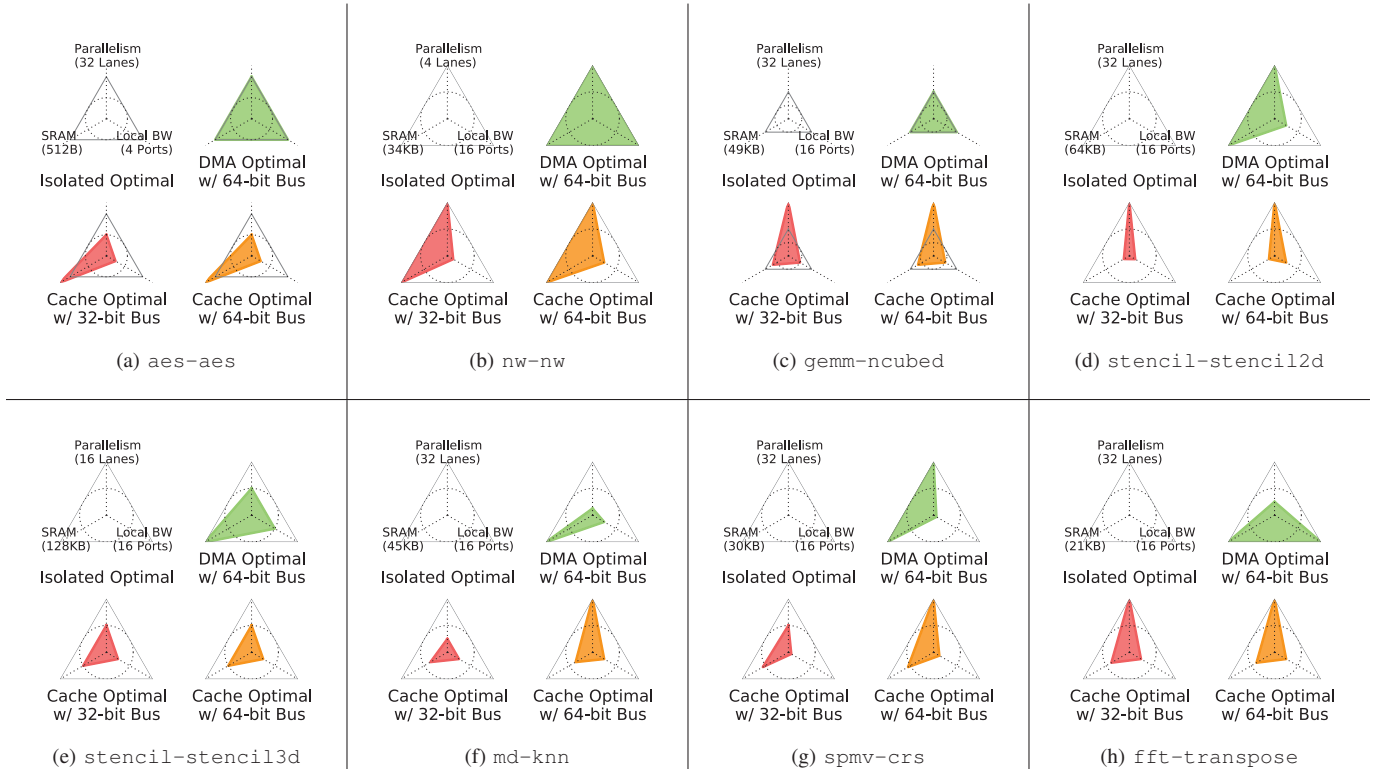
(h) `fft-transpose`

Fig. 9: Comparison of accelerator microarchitectural parameters across four design scenarios. The vertices of the Kiviat plots represent the number of datapath lanes, SRAM sizes, and local memory bandwidth, normalized to the isolated optimal design, shown on the upper-left corner, for each benchmark.

We focus our comparisons on three accelerator microarchitectural parameters: datapath lanes, local SRAM/cache size, and local memory bandwidth to datapath lanes. As before, we select the EDP optimal points from each design scenario for comparison.

Figure 9 shows the differences in these three dimensions for each benchmark under the four design scenarios. For each benchmark, the triangle on the upper-left corner shows microarchitecture parameters for isolated optimal designs. The colored triangles, in turn, represent optimal design choices for DMA with 32-bit bus, cache with 32-bit bus, and cache with 64-bit bus. To show differences between isolated optimal and co-designed optimal choices, we normalize all the designs to the design parameters of the isolated design.

1) Isolated vs Co-Designed Microarchitecture: It is immediately apparent that accelerators designed in isolation overprovision accelerator resources. In Figure 9, almost every colored triangle is smaller than the baseline triangle, showing that isolated designs tend to over-provision computational resources, and more balanced designs can be found by accounting for system-level effects.

This over-design is most pronounced in local memory bandwidth and SRAM size for cache-based designs. Isolated designs attempt to parallelize computation as much as pos-

sible, requiring very high internal memory bandwidth, but in a more realistic environment, the need to move data from system to accelerator imposes a upper bound on performance that makes internal memory-level parallelism less critical. For example, on `spmv-crs` and `md-knn`, both DMA- and cache-based designs require much lower local memory bandwidth than the isolated design. In addition, because caches have the feature of automatic data replacement, they can be sized smaller than scratchpads which must hold all the data, resulting in energy improvements.

In general, caches tend to prefer more parallel datapaths than DMA, as shown in `md-knn` and `fft-transpose`, since their fine-grained nature allows more parallel memory accesses. In fact, `gemm-ncubed` an example where a co-designed cache-based accelerator is more parallel than both the isolated design and a DMA-based one.

2) Impact of System Bus Bandwidth: As a proxy for resource contention in a loaded system, we vary the system bus width to modulate the bus bandwidth available to accelerators. If we compare accelerators designed with a 64-bit bus to those designed with a 32-bit bus (orange and red triangles in Figure 9, respectively), we see that accelerators designed with lower bus bandwidth tend to provision fewer datapath lanes (`md-knn`, `spmv-crs`) and local memory bandwidth
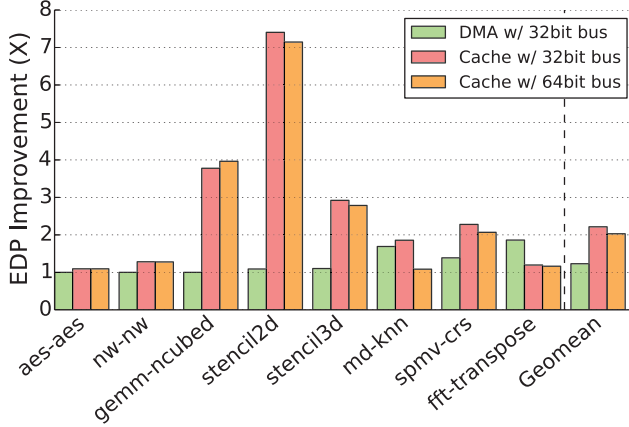
Fig. 10: EDP improvement of co-designed accelerators in different scenarios, normalized to EDP of isolated designs. The design parameters of each optimal design point are illustrated in Figure 9.

(nw, stencil2d, and spmv-crs). These effects happen for the same reasons co-designed accelerators are leaner than isolated accelerators.

3) EDP Improvement: Figure 10 shows the improvements in EDP when accelerators are co-designed, compared to how an accelerator designed in isolation would behave under a more realistic system. This is the same analysis as Figure 1, but applied to more benchmarks and three different design scenarios. Overall, average EDP improves by $1.2\times$, $2.2\times$, and $2.0\times$ for accelerators with DMA, caches with 32-bit system bus, and caches with a 64-bit bus, respectively.

The EDP improvements for co-designed cache-based accelerators is higher than that for DMA-based accelerators because an overly aggressive design for a cache-based accelerator results in a large, highly multi-ported cache, which are much more expensive to implement than partitioned scratchpads. Furthermore, we see that on average, improvements are greater for cache-based accelerators with a 32-bit system bus than a 64-bit bus. In other words, co-design is even more important for contended systems than uncontended systems.

VI. Related Work

Much of the existing literature on accelerators focuses on the design and optimization of the computational datapaths and/or internal memory while assuming that all the data needed already resides in on-chip SRAM. To understand how data movement affects accelerator design, we have discussed caches, coherency, and virtual memory for accelerators, DMA optimizations, and simulators and prototyping platform. Each of these has a considerable body of existing work, and we will describe how our work relates to and differs from them.

We observe that there are two classes of accelerated workloads for which optimization of data movement from global memory to local memory is absolutely critical: big data applications and near-data processing applications. Accelerators for memcached [18], database partitioning [17], and those built with near-data CGRAs [33] all contain specialized interfaces co-designed with the system bus interface and/or memory substrates for efficient bulk data movement and communication. Accelerators that do not fall into these workloads are often tightly coupled with the existing general purpose core and rely on it for data [3].

Caches, coherency, virtual memory, and memory consistency models are all devoted to accessing data on-demand in a safe, understandable, and familiar manner. They have been well studied in the GPU literature in industry and academia [34], [35], [36], [37], [38], but only recently has there been movement towards more fixed function, less programmable accelerators [39], [40], [41], [42], [43]. Examples include IBM's Coherent Accelerator Processor Interface [7], ARM's AXI Accelerator Coherency Port [9], The IBM Cell BE architecture featured a hardware coherent DMA engine, which addresses the software coherency management issues we have raised [44]. and the Intel Heterogeneous Architecture Research Platform [8]. Researchers have investigated specialized coherence protocols for accelerators [45] and hybrid memory models for heterogeneous platforms [46]. With access to global memory spaces, researchers have also devised methods to protect the SoC and accelerators from unsafe memory accesses [47].

Finally, others have integrated accelerator simulators with gem5, such as gem5-gpu [48] and PARADE [14]. PARADE is also an SoC simulation framework, but it only models traditional DMA-based accelerators where all data must be copied to local scratchpads before compute begins. In contrast, gem5-Aladdin is able to model a cache-based accelerator with variable latency memory accesses as well as various optimizations on DMA to reduce idle time.

VII. Conclusion

This paper considers an holistic approach to co-design accelerator microarchitecture and SoC platform parameters. We demonstrate that co-design is critical to achieving balanced, efficient accelerator designs. We highlight that data movement and coherence management for accelerators are significant yet often unaccounted components of total accelerator runtime, resulting in misleading performance predictions and inefficient accelerator designs. We develop gem5-Aladdin, an SoC simulator that captures dynamic interactions between accelerators and the SoC platform, and validate it to within 6% against real hardware. This allows us to explore the design space of accelerator-system co-design, and we show that the optimal energy-delay-product (EDP) of an accelerator microarchitecture can improve by up to $7.4\times$ when system-level effects are considered compared to optimizing accelerators in isolation.

VIII. Acknowledgments

References

[1] Y. S. Shao and D. Brooks, "Research Infrastructures for Hardware Accelerators," *Synthesis Lectures on Computer Architecture*, 2015.

[2] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *ISSCC*, 2016.

[3] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *ISCA*, 2013.

[4] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[5] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in *ISCA*, 2016.

[6] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "ASIC Clouds: Specializing the Datacenter," in *ISCA*, 2016.

[7] J. Stuecheli, "POWER8 Processor," in *HotChips*, 2013.

[8] D. Bryant, "Disrupting the Data Center to Create the Digital Services Economy," *Intel Announcement*, 2014.

[9] S. Neuendorffer and F. Martinez-Vallina, "Building Zynq accelerators with Vivado high level synthesis," in *FPGA*, 2013.

[10] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation," in *SC*, 2011.

[12] S. Kanev, G.-Y. Wei, and D. Brooks, "XIOSim: Power-Performance Modeling of Mobile x86 Cores," in *ISLPED*, 2012.

[13] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures," in *ISCA*, 2014.

[14] J. Cong, Z. Fang, M. Gill, and G. Reinman, "PARADE: A Cycle-Accurate Full-System Simulation Platform for Accelerator-Rich Architectural Design and Exploration," in *ICCAD*, 2015.

[15] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," *ASPLOS*, 2010.

[16] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?," in *MICRO*, 2010.

[17] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning," in *ISCA*, 2013.

[18] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *ISCA*, 2013.

[19] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding Sources of Inefficiency in General-Purpose Chips," in *ISCA*, 2010.

[20] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *MICRO*, 2012.

[21] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel, "Hardware Acceleration in the IBM PowerEN Processor: Architecture and Performance," in *PACT*, 2012.

[22] "TI OMAP Applications Processors."

[23] B. Blaner, B. Abali, B. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J. Reilly, and P. Sandon, "IBM POWER7+ Processor On-Chip Accelerators for Cryptography and Active Memory Expansion," *IBM Journal of Research and Development*, 2013.

[24] P. Yedlapalli, N. C. Nachiappan, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Short-Circuiting Memory Traffic in Handheld Platforms," in *MICRO*, 2014.

[25] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Mach-Suite: Benchmarks for Accelerator Design and Customized Architectures," in *IISWC*, 2014.

[26] Y. S. Shao and D. Brooks, "ISA-Independent Workload Characterization and its Implications for Specialized Architectures," in *ISPASS*, 2013.

[27] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of Error in Full-System Simulation," in *ISPASS*, 2014.

[28] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," in *IEEE Micro*, 2006.

[29] D. Lustig and M. Martonosi, "Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization," in *HPCA*, 2013.

[30] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded Pipeline Synthesis for Data-Parallel Kernels," in *ICCAD*, 2014.

[31] J. Huthmann, J. Oppermann, and A. Koch, "Automatic High-Level Synthesis of Multi-Threaded Hardware Accelerators," in *FPL*, 2014.

[32] D. Burger, J. R. Goodman, and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors," in *ISCA*, 1996.

[33] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *HPCA*, 2015.

[34] M. Harris, "Unified Memory in CUDA 6," 2013.

[35] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *MICRO*, 2013.

[36] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *HPCA*, 2014.

[37] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs," *ASPLOS*, 2014.

[38] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have Your Scratchpad and Cache It Too," in *ISCA*, 2015.

[39] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The Accelerator Store: A Shared Memory Framework for Accelerator-Based Systems," *TACO*, 2012.

[40] M. Lyons, G.-Y. Wei, and D. Brooks, "Multi-accelerator system development with the shrinkfit acceleration framework," in *ICCD*, 2014.

[41] C. F. Fajardo, Z. Fang, R. Iyer, G. F. Garcia, S. E. Lee, and L. Zhao, "Buffer-integrated-cache: a cost-effective sram architecture for handheld and embedded platforms," in *DAC*, 2011.

[42] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Toward Cache-Friendly Hardware Accelerators," in *Sensors and Cloud Architectures Workshop (HPCA)*, 2015.

[43] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures," in *MICRO*, 2015.

[44] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its First Implementation - A Performance View," in *IBM Journal of Research and Development*, 2007.

[45] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," in *ISCA*, 2015.

[46] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: a hybrid memory model for accelerators," in *ISCA*, 2010.

[47] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border Control: Sandboxing Accelerators," in *MICRO*, 2015.

[48] J. Power, J. Hestness, M. Orr, M. Hill, and D. Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *Computer Architecture Letters*, 2014.