Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration

Michael Pellauer* Yakun Sophia Shao* Jason Clemons* Neal Crago* Kartik Hegde†
Rangharajan Ventakesan* Stephen W. Keckler* Christopher W. Fletcher† Joel Emer*‡

*Nvidia †University of Illinois, Urbana-Champaign ‡MIT
{mpellauer, sshao, jclemons, ncrago, rangharajanv, skeckler, jemer}@nvidia.com
{kvhegde2, cwfletch}@illinois.edu

ABSTRACT

Accelerators spend significant area and effort on custom on-chip buffering. Unfortunately, these solutions are strongly tied to particular designs, hampering re-usability across other accelerators or domains. We present *buffets*, an efficient and composable storage idiom for the needs of accelerators that is independent of any particular design. Buffets have several distinguishing characteristics, including efficient decoupled fills and accesses with fine-grained synchronization, hierarchical composition, and efficient multi-casting. We implement buffets in RTL and show that they only add 2% control overhead over an 8KB RAM. When compared with DMA-managed double-buffered scratchpads and caches across a range of workloads, buffets improve energy-delay-product by 1.53× and 5.39×, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Other architectures** ;
• **Software and its engineering** → *Buffering* ;

KEYWORDS

Accelerators, Staging Buffers, Data Orchestration, Synchronization

1 INTRODUCTION

Architects are increasingly turning to domain-specific accelerators to satisfy the insatiable performance demands in areas like machine learning and image processing [6, 12, 18, 22, 27, 34, 35, 37]. Accelerators leverage dense, customized datapaths for computation, but off-chip memory accesses remain slow and costly in comparison. Thus accelerators spend significant area for on-chip memory hierarchies (Table 1).

A key difference from general-purpose processors is that accelerator architects leverage knowledge of the workload and domain to achieve high-performance, energy-efficient *data orchestration* – that is, the transfer of active regions in and out of the buffer hierarchy. We term this task *explicit* data orchestration, as the decisions are under the direct control of the architects. Indeed, several contemporary accelerators—such as DaDianNao [5], Eyeriss [6] and SCNN [35]—devote significant effort towards engineering custom buffering arrangements and present their orchestration as a major contribution inseparable from the architecture itself. Others mention custom buffering in passing but focus their presentation on other aspects such as novel datapaths [3, 19, 22, 28, 36]. Overall, a significant cost of any accelerator is the data orchestration buffer hierarchy, both in terms of area and effort.

Table 1: Percentage of area dedicated to on-chip memory for a selection of machine learning accelerators.

DaDianNao [5]:	48%	Eyeriss [6]:	40%-93%
EIE [18]:	93%	SCNN [35]:	57%
TPU [22]	35%	PuDianNao [27]	63%

A downside of this design-by-design approach is that the accelerator community lacks a generalized, reusable storage hierarchy that is not tied inextricably to any particular design or domain. Beyond wasteful duplication of engineering effort, the lack of a consistent abstraction for data movement, staging, and synchronization also means a dearth of toolflows exist to auto-orchestrate data-sets and generate control FSMs for data movement. In fast-moving application domains like machine learning, there is a particular need to decrease design effort, lest accelerators find themselves obsolete before reaching the market.

In the general-purpose computing community, existing reusable buffer idioms such as caches, scratchpads, and FIFOs have served as the foundation for several toolsets, but these approaches are ill-suited for the needs of accelerators. Caches direct too much area and power into dynamically making *implicit* data orchestration decisions, at odds with the desires of accelerator architects. FIFOs are too inflexible to serve the complex data reuse and update patterns of modern accelerator application domains. Scratchpads lack synchronization, making them difficult to hierarchically compose.

This paper introduces *buffets*, a novel storage idiom for explicit data orchestration. Buffets are efficient and composable, and are not tied to any particular accelerator design or domain. The interface of buffets raises the level of abstraction, encapsulating synchronization in the staging buffer and thus lowering the complexity of designing and verifying the accelerator itself. We make the following specific contributions:

ASPLOS'19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304025

(1) We develop a novel taxonomy of buffering idioms and discuss why an approach based on explicit *decoupled* data orchestration (EDDO) is the best match for accelerators.
(2) We present detailed operational behavior for buffet interfacing. We focus on efficient implementation of synchronization within the buffet itself without remote polling or barriers.
(3) We present a scheme for seamlessly composing buffets into hierarchies, similar to caches. We extend inter-buffet synchronization to support multicast of data from a single buffet access, an efficiency feature that is not available in traditional scratchpads or caches.
(4) We implement buffets in RTL and compare them to double-buffered scratchpads and caches across a range of workloads, and show that buffets improve energy-delay product by 1.53× and 5.39×, respectively. We also show that buffet-based accelerators achieve similar performance at 3.4× less area than caches.

To facilitate adoption, we provide an open-source reference implementation of a buffet, written in Verilog, here:

https://github.com/cwfletcher/buffets

2 CLASSIFYING DATA ORCHESTRATION

Accelerator architects leverage their design-time knowledge of workload characteristics and access patterns, allowing them to extract benefits such as:

- Preemptively transferring exactly the data that will be referenced in the future,
- Maximizing the number of accesses to data in the smallest, fastest and most energy-efficient buffer,
- Staging data at the least-upper bound buffer between sharers in the hierarchy,
- Overlapping the fill of the next data tile with the consumption of the current data tile,
- Simultaneously broadcasting (or multi-casting) the result of a buffer access to all consumers of the accessed data,
- Synchronizing data availability precisely and cheaply,
- And, removing data exactly when it is no longer needed.

Figure 1 shows a classification of traditional deployment scenarios for reusable buffering idioms along two axes. (We discuss specific contemporary related work in Section 7.) At a high level, the implicit/explicit distinction refers to the level of workload knowledge that can be leveraged to control staging buffer decisions, while the coupled/decoupled axis refers to whether memory responses and requests are round-trip or flow-forward. We now present a detailed discussion of this taxonomy, and establish why these buffering schemes are not able to sufficiently provide the accelerator features described above. Table 2 presents a comparative summary of the major points covered throughout the section.

2.1 Implicit versus Explicit Orchestration

In the general-purpose computing community, caches (Table 2A) have served admirably as a reusable, modular buffer abstraction based on load/store operations. Although the engineering and area costs for a given cache hierarchy may be quite high, the effort is often amortized across several design points with appropriate re-parameterization. Caches have several desirable properties, such as

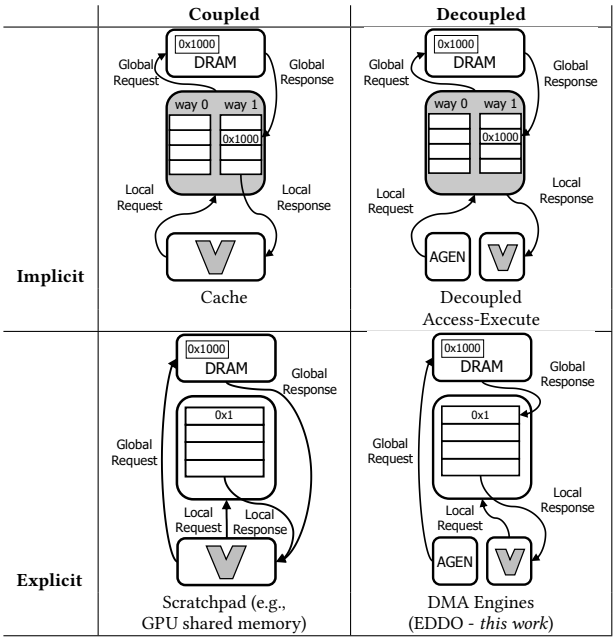

Figure 1: Taxonomy of data orchestration approaches, as used in typical deployment scenarios.

composing invisibly into hierarchies. Memory-level parallelism—both multiple outstanding fills, as well as concurrency between fills and accesses to current contents—can be achieved using well-studied additional hardware (often called *lockup-free* cache structures).

We say that caches perform *implicit* data orchestration as the load request initiator does not directly control the cache hierarchy's decisions about whether the response data is retained at any given level of the storage hierarchy, nor when it is removed. (In Figure 1 this is represented by the Global request/response being shielded from the datapath.) Heuristic replacement policies are advantageous in general-purpose scenarios because they are workload agnostic[1]. On the other hand, for domain-specific accelerators, the area and energy overheads for features like tag matches and associative sets are considered unacceptable. It is notable that no contemporary commercial machine learning ASICs incorporate caches.

One alternative is to use *scratchpads* (Table 2B), which expose an address range of a particular staging buffer for loads/stores, thereby enabling *explicit* and precise control over the orchestration. (In Figure 1 this is represented by the datapath managing both local and global request/response.) A GPU's *shared memory* scratchpad [32] is the most widespread contemporary example of this idiom for explicit data orchestration. The size and address range of the scratchpad is exposed architecturally, and the transfer of data into and out of the scratchpad is managed via explicit instructions. While scratchpads avoid the hardware overheads of caches, extracting memory parallelism–both across fills and overlapping fills

[1] As many programmers care more about optimization than portability, they often reverse engineer the details of the cache hierarchy and replacement policy to try to explicitly manipulate them. This is an indication that architects could provide more officially-supported explicit data orchestration features in general-purpose processors.

Table 2: Summary of properties of traditional data orchestration approaches in typical deployment scenarios. Shaded cells indicate undesirable properties for domain-specific accelerators (though may be acceptable for general-purpose architectures).

	(A) Datapath + Cache (Implicit, Coupled)	**(B)** Datapath + Scratchpad (*Explicit*, Coupled)	**(C)** D.A.E. Dpaths + Cache (Implicit, *Decoupled*)	**(D)** DMA + FIFO + Dpath (*Explicit, Decoupled*)	**(E)** DMA + *Buffet* + Dpath (*Explicit, Decoupled*)
Buffer Non-RAM Area	High	Low	High	Low	Low
Buffer Access Energy	High	Low	High	Low	Low
Placement Policy	Workload-agnostic	Workload-controlled	Workload-agnostic	Workload-controlled	Workload-controlled
Achieving Multiple Fills in Flight	Complex (lockup-free structs.)	Complex (unrolling, multi-thread.)	Complex (lockup-free structs.)	Straightforward (credit scheme)	Straightforward (credit scheme)
Achieving Overlapped Fill and Access	Complex (static req. pipelining)	Complex (static req. pipelining)	Straightforward (dynamic rate matching)	Straightforward (dynamic rate matching)	Straightforward (dynamic rate matching)
Hierarchically Composable	Yes	No	Yes	Yes	Yes
Landing Zone Hold Time	Round-trip	Round-trip	Hop-to-hop	Hop-to-hop	Hop-to-hop
Access Multicast	Dynamic coalescing	Dynamic coalescing	Workload-controlled	Workload-controlled	Workload-controlled
Data Availability Synchronization	Encapsulated (load-to-use)	Encapsulated (load-to-use)	Out-of-band (supplemental queue)	Encapsulated (peek stalling)	Encapsulated (read stalling)
Access Order	Arbitrary	Arbitrary	Arbitrary	Strict FIFO	Arbitrary
In-Place Updates	Yes	Yes	Yes	No	Yes
Removal Policy	Workload-agnostic	Workload-controlled	Workload-agnostic	Strict FIFO (or clear all)	Workload-controlled

and accesses–is tedious and error-prone[2], and as a result they are difficult to compose into hierarchies.

2.2 Coupled versus Decoupled Orchestration

Caches and scratchpads both use a load/store paradigm where the initiator of the request also receives the response. We call this a *coupled* staging of data, reflected in the left column of Figure 1. With this setup, synchronization between data demand and data availability is efficient and intuitive—the requester is notified when corresponding response returns (load-to-use). The disadvantage to this approach is that it complicates overlapping the fill and access of data tiles (e.g., double-buffering) as the single requester/consumer must alternate between requesting and consuming responses. Additionally, a "landing zone" for the incoming data tile must be held reserved for the entire round-trip load latency, which increases pressure on RAM resources that could otherwise be used for larger tile sizes.

The alternative is to *decouple* the load request initiator from the response receiver. (In Figure 1 this is represented by the request/response arrows going to different modules). In this setup, a separate hardware module (e.g., a DMA engine, or *address generator* (AGEN)) is responsible for *pushing* data into one or more functional units' staging buffers.[3] To tolerate latency, these are often double-buffered and hence sometimes referred to as ping-pong buffers [9, 10]. The main advantage to this approach is that the requester can run at its own rate, and can multicast data to multiple simultaneous consumers. Additionally, the feed-forward nature of the pipeline means that the tile landing zone only needs to be reserved proportional to the latency between adjacent levels of the hierarchy, rather than the entire hierarchy traversal round-trip, allowing for increased utilization of equivalent RAM. Finally, this approach often can transmit large blocks of data, i.e., bulk transfers, which are more efficient than small requests, which must dynamically re-coalesce accesses to the same memory line.

This separate producer/consumer approach is similar to Smith's [42] *decoupled access-execute* (DAE) style of general-purpose computing architecture (Table 2C). In a DAE organization two processors are connected by a hardware queue. The access processor is responsible for performing all address calculations and generating loads—analogous to the DMA engine. Load responses are passed to the execute processor—analogous to an accelerator's functional units and their local staging buffers. DAE improves parallelism and reduces the critical paths of instructions while allowing both processors to compute at their natural rate. However, classical DAE does not explicitly control data orchestration buffers—decisions about staging data are still managed by the cache hierarchy, thus Figure 1 categorizes DAE as implicit decoupled.

We advocate that the *explicit decoupled* data orchestration (EDDO) approach best matches the needs of domain-specific accelerators, as it allows for the best opportunities to leverage static workload knowledge combined with efficient, high-performance hardware. Hardware FIFOs [21, 46] (Table 2D) are one traditional reusable EDDO staging buffer organization. The advantages are that FIFOs cleanly encapsulate synchronization via head and tail pointers, and are easily hierarchically composable. However, in practice FIFOs are not flexible enough to meet the needs of modern accelerators, which often require random access within an active window or tile of data [4, 6, 13, 45]. Additionally, for data types such as the partial sums in a convolutional neural net, staged data must be modified several times in place before being drained [6, 29]. This is not possible in single write-port FIFOs without costly re-circulation. Thus our goal is to combine the efficient hardware of FIFOs with the flexibility of scratchpads (Table 2E).

2.3 Synchronization Concerns

In a decoupled system, the timing of loading new tiles is a critical correctness component, as initiating a transfer too early could overwrite live data, and a transfer too late results in efficiency loss. Some accelerators use *systolic* approaches to bound tile processing

[2]GPU shared memory is paired with high multi-threading and loop unrolling to offset these problems, but this complexity is considered unacceptable for fixed-function accelerators.

[3]Cache pre-fetching can be considered an example of decoupling. Consideration of this large body of work is beyond the scope of this paper.

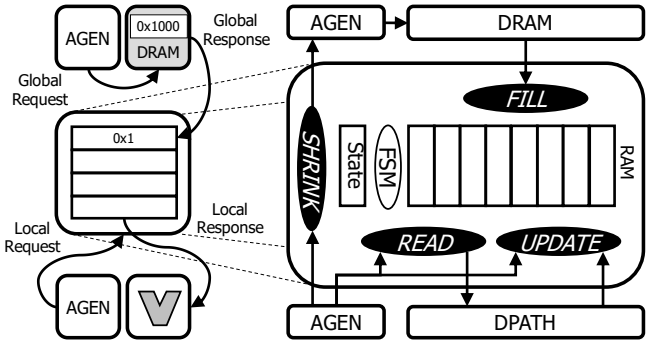

Figure 2: Buffet taxonomy characterization and operations.

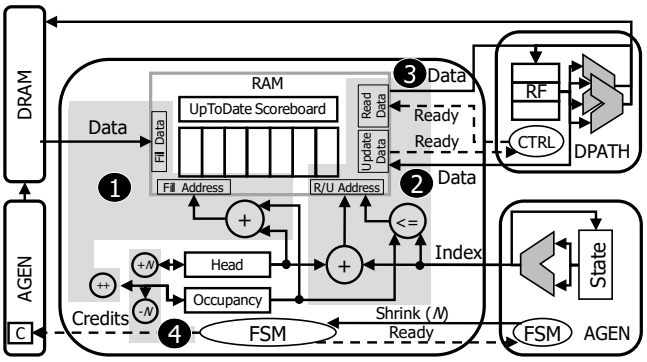

Figure 3: Buffet implementation details.

time—essentially removing the need for synchronization hardware beyond simple counters. However, these time bounds often become overly conservative in realistic systems that have non-deterministic latencies involving off-chip accesses, arbitrated networks-on-chip, or functional units whose processing time involves conditional execution. Therefore a reusable storage idiom cannot rely on them.

This paper assumes that accelerators are built using a standard ready-valid (Rdy/Vld) micro-protocol for pipeline flow control as in SystemC compilation [2, 31]. For example, a FIFO could assert Pop.Rdy when it is non-empty, and dequeue the oldest element when Pop.Vld is asserted. Without loss of generality, our techniques can be applied using other micro-protocols such as Try/Ack. For convenience, this paper uses an operation-centric terminology [20] and assumes a straightforward translation into micro-protocol. For example, with FIFOs the following pseudo-code:

```
qC.Push(qA.Pop() + qB.Pop())
```

is shorthand for "when `qC` is asserting Push.Rdy (not full) and `qA` and `qB` are asserting Pop.Rdy (not empty), perform an addition and assert all Vlds." We refer to the micro-protocol details as appropriate for clarity in cases where this shorthand is not sufficient.

While circuit-level micro-protocols form the necessary groundwork for adjacent stages of a properly synchronized accelerator, they are not a complete solution for data orchestration, which is a higher-level concern spanning remote producers and consumers. In the remainder of this paper, we present an EDDO storage idiom that encapsulates fine-grained synchronization within the staging buffer operations themselves. Encapsulation increases composability and re-usability, allowing accelerator architects to cheaply leverage the benefits of EDDO for multiple application domains.

3 THE BUFFET STORAGE IDIOM

Figure 2 depicts the data orchestration model of a *buffet*. In reference to the taxonomy found in Table 1, buffets fall into the EDDO quadrant, as data movement is explicit and they use decoupled fill engines. The buffet approach expands the decoupling farther than traditional DMA setups by also using decoupled address generators to iterate over staged data. In Section 4 we discuss the composability benefits of this organization.

Figure 2 also depicts the buffet's operational interface. Within the buffet, a finite-state machine controls the four fundamental storage operations: Fill(data), Read(index), Update(index, data), and Shrink(num). Upper-level modules in the hierarchy (e.g., DRAM) initiate Fill operations and move new data into the buffet, while lower-level modules (e.g., datapath) work with data in the buffet using Read and Update operations. Finally, Shrink operations remove data from the window.[4] Thus the lifetime of a piece of data in the staging buffer can be described with the following regular expression:

Eq. 1: $Fill \rightarrow (Read \rightarrow Update?)^* \rightarrow Read \rightarrow Shrink$

Unlike scratchpad loads and stores, these operations encapsulate synchronization. We now present a detailed description of buffets' internal logic to arbitrate stalls.

3.1 Buffet Operational Behavior

Figure 3 presents a detailed architectural diagram of a buffet, and Algorithm 1 shows specific behavior. Newly transferred data is installed into the RAM via the Fill logic, labeled ❶. This path closely matches a conventional FIFO—no remote address accompanies the data, and placement is based on local address generation in the order it is received. (This is not a fundamental restriction, but the complexities of un-ordered Fill are beyond the scope of this paper.) Unlike a FIFO, the Read ❷ request includes an externally-provided index, allowing data to be read in a different order than it is received. The index is relative to the window, so 0 represents the oldest installed datum in the staging buffer. It is not legal for the index to exceed the physical size of the RAM.

Logic ensures that reading a position outside the active window will stall until the data arrives, similar reading an empty FIFO. However, the presence of this index means that buffet read stalling must be handled differently than traditional FIFOs, such as one generated by commercial tools [21, 46]. In these circuits Pop.Vld is asserted whenever the FIFO is non-empty. In buffets, the presence of the requested data is a function of the index. Therefore we use a separate *read response* path, represented by the "read_rsp_out.Send()" operation. ReadRsp.Vld is only asserted when the requested data has been filled, as in caches. For simplicity we present read responses returning in the same order as requests—sufficient for the needs

[4] We choose the name buffet due to similarities with actual restaurants, where waiters bring out new dishes (fill) which are repeatedly iterated over by diners (read) until a course change (shrink). Of course, in restaurants diners are not allowed to return modified dishes to the buffet (update), due to food safety concerns!

Algorithm 1 Buffet Operational Details

```
// Initialize a buffet
function INIT(sz)
    head = 0;
    occupancy = 0;
    size = sz;
    credit_out.Send(size);
end function
// Emplace new data for staging.
function FILL(data)
    slot = (head + occupancy) % size;
    buffer[slot] = data;
    SetUpToDate(slot, true);
    occupancy++;
end function
// Iterate over staged data.
function READ(index, will_update)
    slot = head + index % size;
    wait_until(index < occupancy && IsUpToDate(slot));
    read_rsp_out.Send(buffer[slot]);
    SetUpToDate(slot, !will_update);
end function
// Update previously read locations.
function UPDATE(index, data)
    slot = (head + index) % size;
    buffer[slot] = data;
    SetUpToDate(slot, true);
end function
// Unstage data and free room for more Fills.
function SHRINK(num)
    wait_until(num < occupancy);
    head = (head + num) % size;
    occupancy = occupancy - num;
    credit_out.Send(num);
end function
```

of most accelerators—but this is not fundamental (as in caches). In scenarios where blocking is harmful, a supplemental non-blocking CHECK(INDEX) operation can be added to test whether a certain index is in range.

Beyond indexed reads, a significant distinction from a FIFO is that data elements within the active window can be modified in-place, which we call the UPDATE ❸ path. Internal logic stalls the modification of the RAM until both an index and a data element are asserting VLD. Thus the index generation FSM and datapath can produce at different rates, and the system can tolerate dynamic timing variation. Semantically distinguishing RAM writes into FILL and UPDATE operations raises the level of abstraction, allowing buffets to use customized synchronization logic for each case, as discussed below.

Finally, the SHRINK path ❹ depicts the logic for removing staged data from the buffet. This operation takes a size parameter and removes that many elements from the active window. This operation simply updates internal scoreboarding—no data movement occurs. A credit is released to the FILL address generator indicating room for another bulk transfer. Accelerators that modify in-place data should drain it via the standard READ path before invoking SHRINK. We advocate that the same index generation FSM that iterates staged data should generate calls to SHRINK, removing the need for explicit stall logic between index-based read requests and re-basing of the oldest element in the active window. This arrangement is not fundamental, but for simplicity of synchronization we use this presentation for the remainder of this paper.

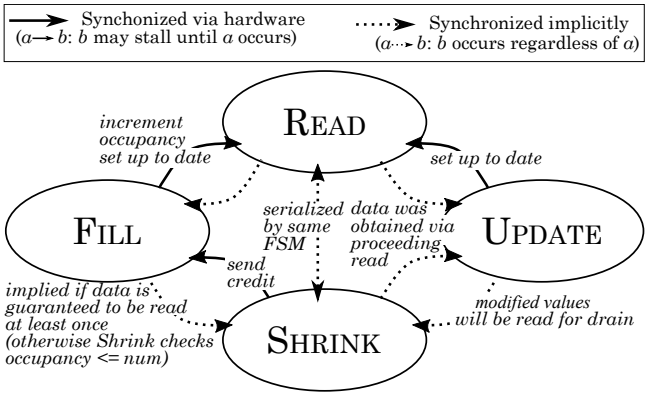

Figure 4: Operation synchronization relationships.

3.2 Buffet Synchronization Details

We leverage the semantics of buffet operations to provide fine-grained synchronization with minimal logic and stalls. The cases where explicit hardware synchronization is required are represented by the "wait_until" calls in Algorithm 1. Many other cases of synchronization can be handled without the need for explicit hardware support because of the order of operations in Equation 1, as shown in Figure 4. For example, because modified values are always READ before SHRINK, we do not need synchronization logic between UPDATE and SHRINK—they are transitively synchronized through READ.

Because modification of data by the datapath can take notable latency, buffets add RAW-hazard checks, which we present abstractly as an "UpToDate" scoreboard with perfect knowledge. In practice, any physically efficient tracker can be used, including imprecise hashing schemes. For simplicity, we present `will_update` as a parameter to READ that indicates that the datapath will modify the currently staged value—many alternative interfacing paradigms can be conceived of, including LOCK() methods. If a subsequent READ requests an index that is undergoing modification, the response is stalled from returning—indistinguishable from reading an index that has not yet been filled. FILL operations do not need to perform this check. Thus we can improve energy efficiency and performance using higher-level knowledge.

Encapsulating hazard detection inside the buffering interface removes the need for engineering custom stall logic on a per-deployment basis. To meet the efficiency demands of domain-specific accelerators, buffets provide options for design-time customization. If the architects can prove that no RAW hazard is possible, then the RAW hazard detection logic can be statically removed via a parameter. Similarly, if FILLs are proven to be mutually exclusive with UPDATEs then they can share a write port rather than using a more expensive RAM with multiple write ports. Finally, for buffets that hold read-only data the entire UPDATE path can be removed. The goal is to allow designers to quickly construct a functionally

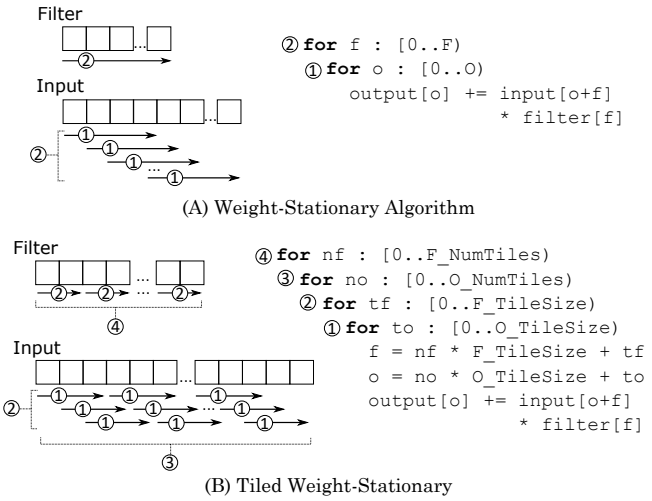

Figure 5: 1D Convolution example orchestration approach.

correct orchestration hierarchy that can serve as a starting point for optimization refinements.

Another case for design-time optimization is the synchronization between Shrink and Fill. Algorithm 1 conservatively uses explicit synchronization for Shrink: e.g., "wait_until(num < occupancy)". This logic can be removed at design time if the architects can prove each staged data element will be read at least once. Figure 4 depicts this as implicitly synchronized, as we expect this to be the more common case. Synchronization with the external fill request generator is handled via backwards credit flow—similar to a network-on-chip protocol, discussed in detail in Section 4.

3.3 Example Orchestration with Buffets

Figure 5A shows an example 1-dimensional convolution that demonstrates the use of buffets for EDDO. We purposely choose a weight-stationary dataflow [6] which involves re-loading partial sums several times until the final sum is produced to demonstrate all buffet features. These same principles generalize to other dataflows of full convolutional neural networks and other kernels.

The baseline formulation maximizes reuse of filter weights with only a single on-chip register, but it must resort to expensive off-chip accesses for inputs and outputs (if 2× the size of O is larger than on-chip buffering). Figure 5B shows a more realistic tiled formulation. This introduces some re-reads of weights, but significantly reduces reloads of inputs and outputs as the tiles can be held resident on-chip.

Figure 6 shows a straightforward accelerator implementation of this algorithm using buffets. This accelerator uses separate buffers per datatype—similar to an instance of Eyeriss [6] with a single processing element and a different dataflow. Following the EDDO principles, separate address generation FSMs (labeled ❶) generate requests to the DRAM that install data into the staging buffers. The weight-stationary dataflow means that the weights are staged once while input and output tiles are re-staged per weight tile. The "wait_until" in the transfer FSMs ❷ represents blocking until sufficient credit is available. The backwards path that increments the credit count is not shown.

Looking in-depth at the Read FSMs ❸, we highlight several points that distinguish buffets from FIFOs. First, both the input and output buffet are performing window-based access relative to the oldest element. Second, the size-based shrink gives more control over data liveness—the input is reading a tile volume larger than its shrink, and so represents a sliding window. Finally, the Output buffet is being used to perform in-place updates of staged sums. Each sum is modified `F_TileSize` times per tile. The buffet's internal scoreboarding ensures that subsequent Reads will block on previous Updates—a scenario which could occur if the MACC datapath was implemented as pipeline with internal latency. As stated above, unnecessary scoreboard logic can be removed using design-time parameterization. In this algorithm, every partial-sum output that is read is also modified, so the accelerator can use the ReadAndUpdateOutput FSM to generate Read and Update indices. Other scenarios can require separate index generators for these two classes.

Furthermore, the Read FSMs reveal some key ways that buffets differ from traditional scratchpads. In a scratchpad, read requests return the current RAM value, so it is the responsibility of the iterator FSM to not issue a read request until it knows the desired data has been staged. Buffets' encapsulation of fine-grained synchronization means that no explicit checks are present in the Read FSM. The index generator issues Read operations at its natural rate—if the requested data is not available, then the read_rsp.Vld signals will stall the Datapath FSM ❹, as described in 3.1. Furthermore, with scratchpads all accesses are done via absolute addresses into the underlying RAM, which places the burden of dividing the RAM into active and inactive regions onto the index generator. With buffets, the FSMs contain no explicit base address manipulation, nor wrapping-around of addresses relative to the size of the RAM. This hardware has not disappeared, but has been encapsulated inside the buffet, simplifying the creation of the index generation FSM to only repetition counts, bounds, and offsets.

This encapsulation of the modular arithmetic is important for another reason: offset-based indexing separates the size of the active tile from the size of the underlying storage which is pre-buffering future tiles. One way this manifests is that Figure 6 does not specify concrete sizes for the underlying RAM. If it is equal to the tile size, no pre-buffering will occur, as each buffet can only hold the active window. If it is twice the tile size, then the arrangement is equivalent to traditional double-buffering. It can also be set to any arbitrary constant. This flexibility has an important implication: the functionality of the FSMs in Figure 6 is unchanged across all these options, and do not need any alteration or re-verification if the underlying RAM size of the buffets is increased as part of design-space exploration. This will not affect correctness, only performance as more room is available to pre-fill future tiles.

Designers often talk about double-, triple-, or even quad-buffering, but extra buffering should not be limited to multiples of tile size. By changing the Fill FSMs to transfer after receiving smaller credit totals such as 1/4, or 1/8 tile (or even arbitrary absolute values unrelated to tile size) architects can determine the optimal buffering needed to tolerate the forward latency through the memory system, which is unlikely to be an exact multiple of tile size. Additionally, if

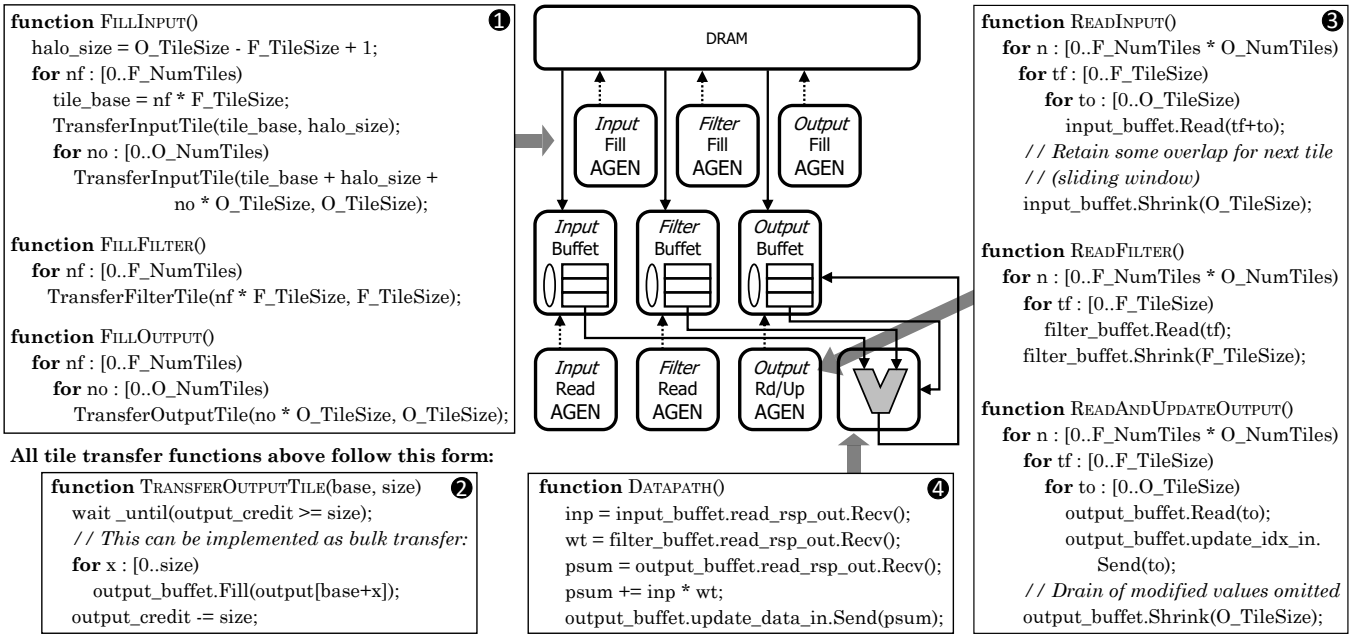

Figure 6: Basic accelerator of the example from Figure 5 using buffets, with pseudo-code describing the EDDO data transfers.

some datapaths are farther from the memory buffets' encapsulation makes it straightforward to give them more buffering for extra latency tolerance without altering their logic. In effect, this approach makes the staging buffer RAM sizes a low-level micro-architectural feature that can be determined late in the design process based on underlying physical properties, rather than a first-order design consideration.

3.4 Automatically Deriving Configuration

The EDDO approach requires the implementation of separate index generation engines per buffet. These iteration loops all relate to the original tiling and inter-tile schedule chosen by the architects using workload knowledge. As a specific example, the loop for ReadFilter in Figure 6 removes the `O_TileSize` loop level because in the original algorithm (Figure 5) the variable `o` is not used to index the `filter` array. This implies that the same filter weight value is being held local to the datapath while the input and output changes with `o`—hence the name "weight stationary". Similarly, the sliding window on the input buffet is created by addition of the `f` offset into the `input` vector.

We believe that automatic derivation of control FSMs from a single tiling specification is feasible future work. This has been accomplished before for specific high-value domains [16, 40] but not in a generalized tool. The domain-specific nature of accelerators limits the data orchestration patterns that need to be supported by automatic generation solutions. Furthermore, architects of accelerators can always fall back on handcrafting the data flows if the tools fail. The design of such tools is beyond the scope of this paper.

In summary, buffets' encapsulation of synchronization into the operational interface allows design to more productively occur at a higher level of abstraction. We view buffets as combining the best properties of FIFOs, N-buffered scratchpads, and custom sliding window buffers. In the next section we will demonstrate that these same features also enable composition of buffet hierarchies.

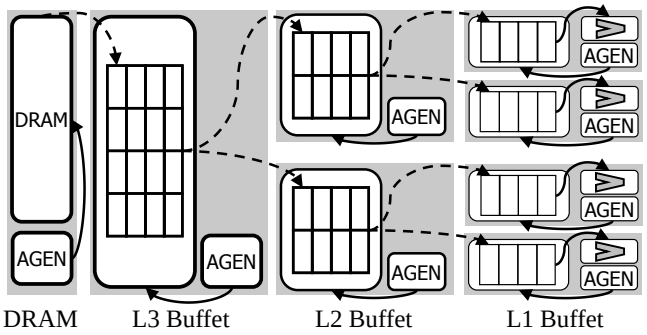

Figure 7: An example buffet hierarchy.

4 COMPOSITION OF BUFFETS

No staging buffer can be considered entirely in isolation. Domain-specific accelerators require the ability to hierarchically stage and distribute data, which facilitates maximizing data reuse from small buffers physically co-located with functional units. This section describes the interaction, modularity, and composability of buffets.

4.1 Buffet Hierarchies

Multiple levels of buffets can be seamlessly arranged into a hierarchy, as shown in Figure 7. An upstream buffet's address generation FSM's Read operations take on a role similar to a traditional DMA engine, driving data to Fill the next buffet using the crediting mechanism presented previously. The downstream buffet's Read FSM

blocks locally, without any external polling. Thus serially composing N buffets requires $N+1$ address generators, not $2N$. Larger tiles are staged, then broken down and distributed to lower levels for further staging and processing. We show data as flowing through all levels, but this is not fundamental—fills can bypass intermediate levels as appropriate. The data also need not be inclusive as upstream buffets can shrink away data before it is fully consumed downstream.

Importantly, a hierarchy of buffets allows applications to fully exploit data reuse at each level in the hierarchy. Unlike FIFOs, buffets' arbitrary iteration ordering means that data iteration order can be changed at each address generation level. This both allows interleaving of tile delivery across lower levels and altering the data layout in the memory dynamically so that each level may use a customized layout.

Backwards paths to return modified data to larger, upper-level buffets are accomplished using the Update functionality. Thus the slot for the value is held in a non-up-to-date state by the upper-level buffet until the modified value is produced. As a result Update traffic does need any kind of credit-check mechanism. Additionally, there is no requirement that modified values pass through all levels of staging buffer during writeback. Accelerators can provision connectivity to transmit results from the datapath directly to higher levels, or to off-chip memory.

As both Read and Update requests are generated locally, any given index FSM only needs to concern itself with generating addresses sized to its local RAM. This means that low-bitwidth address calculation datapaths can be used rather than 32 or 64-bit address arithmetic. (Domain-specific accelerators generally do not use address translation for on-chip accesses, but it is not uncommon to use large-bitwidth addressing to ease system integration.) As a secondary benefit, addresses are never transmitted remotely between buffets, which can reduce transfer bandwidth and energy compared to traditional memories.

Buffets make no assumptions about being deployed with direct connections between levels, or being connected via a network-on-chip. In the latter case, buffets' credit flow for forward Fills and use of Update for backward writeback of modified values act as a guarantee of deadlock freedom, as any injected data will be unconditionally drained—a so-called "consumption assumption". This guarantee can simplify network flow-control design and verification, and result in simpler hardware. For example, buffet-based accelerators may require fewer or no virtual channels for correctness—though they still may still be employed for routing algorithm or quality-of-service reasons. Furthermore, a buffet's Fill path may be directly attached to the output of a router port, essentially supplanting the need for separate egress buffer hardware.

4.2 Sharing Fills via Multicast

If the transport substrate supports broadcast or multicast, either via dedicated wires or a network, then buffets can leverage it. Multicast is a significant source of energy efficiency, as it means the output of one access to a large physical RAM can be delivered efficiently to all consumers in the EDDO specification. Leveraging static workload knowledge is more area- and energy- efficient than dynamically detecting and coalescing multiple requests to the same address

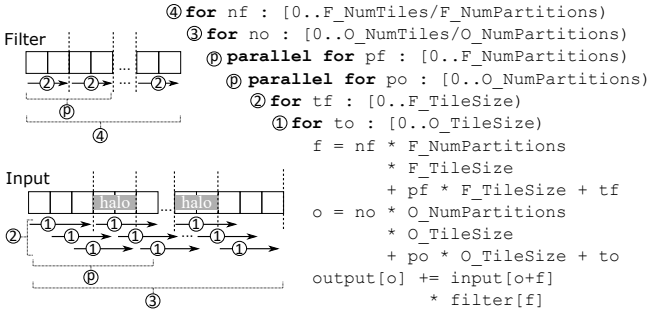

Figure 8: Parallel partitioned version of Figure 5.

in a cache or scratchpad hierarchy, and does not rely on multiple requests arriving within a limited time window.

To achieve multicast we use a straightforward extension of the credit scheme presented in Section 3. The Fill FSM's credit register is extended to a vector tracking credits of multiple target buffets. All downstream targets must have sufficient room before a transfer can be initiated, and credits are decremented from all targeted counters upon transmission. Finally, the backwards credit path is supplemented with an ID field indicating from which buffet the credit originated, which is used to increment the appropriate counter in the credit vector. An extension of this scheme uses runtime configurable routes and IDs allows the design of accelerators with dynamically reconfigurable buffet hierarchies.

4.3 Sharing Physical RAMs Efficiently

Scenarios exist where it is advantageous for multiple buffets at the same level of the hierarchy to share a single physical RAM. Sharing can avoid internal space fragmentation due to RAM size constraints and decrease per RAM overheads. Additionally, architects may wish to exploit dynamically reconfiguring buffet sizes so that more space can be allocated to the most critical data structure.

To implement this feature, we supplement the logical buffet bookkeeping with `base` and `bound` registers. All increments to the head pointer are carried out modulo base and bound. Various multi-port and/or banking schemes can be used to maximize RAM efficiency by matching the required bandwidth across data types. For example, the accelerator previously presented in Figure 6 uses a weight stationary dataflow which means that new filter weights are only required every `O_TileSize` cycles, whereas new inputs and outputs are required every cycle. Therefore it may be profitable for the filters to share the same RAM as either the inputs or outputs, but multiplex the inputs and outputs themselves would reduce throughput more significantly.

Overall, RAM-sharing buffets represent a combination of design-time customization and runtime flexibility. The aim is that these design-time parameters allow buffet deployments to approach the efficiency of ad-hoc accelerator-specific buffer schemes with minimal engineering effort and without sacrificing re-usability.

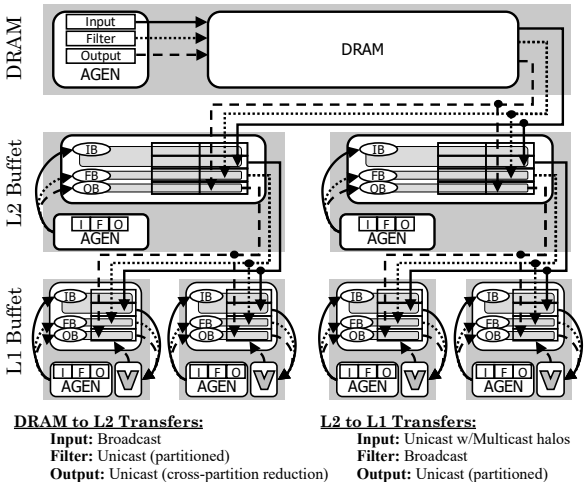

Figure 9: Example accelerator derived from the EDDO in Figure 8 with multiple levels of buffets sharing RAMs. For simplicity, paths for cross-partition reduction of partial sums are not shown.

F_NumPartitions	10
O_NumParititions	10
Total Functional Units	100
F_TileSize	256
O_TileSize	1024
Indiv. L2 Buffet Size	45.5 KB
Total L2 Buffet Size	455.0 KB
Indiv. L1 Buffet Size	5.0 KB
Total L1 Buffet Size	500.0 KB
Input size	1.64 MB
Filter size	74 KB
Input:Filter ratio	21.78

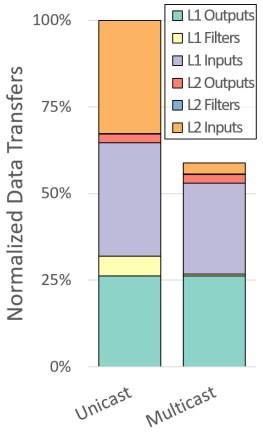

Figure 10: Analysis of the impact of enabling multicast on a Figure 9-style accelerator.

4.4 Example of Hierarchical Orchestration

We now revisit the 1-dimensional convolution example presented in Section 3.3 in the context of a more realistic accelerator with parallel functional units, a multi-level buffet hierarchy, and physically shared RAM. Figure 8 shows an extension of the weight-stationary dataflow by spatially partitioning individual tiles across the separate functional units—represented by parallel-for loops. We use the term "partition" to refer to tiles that are executed by parallel hardware instances. As the number of partitions is a design-time parameter, any remaining mismatch between data size must be handled as passes in the outer loop. We omit edge cases from the algorithm.

Figure 9 shows the accelerator that results from this EDDO architecture when `F_NumPartitions=2` and `O_NumPartitions=2`. Buffets within a level share the same multi-banked RAM, which could potentially allow `F_TileSize` and `O_TileSize` to be configured at runtime, with the constraint that the total tile size across all data types size fits in the underlying RAM. This dataflow employs separate partitions updating the same partial sum, and so requires an additional datapath for reduction before updating DRAM. This path is not shown in the figure as it has no effect on the underlying orchestration patterns.

Despite the pedagogical nature of this accelerator, it has several interesting characteristics. First, each L2 buffet holds a disjoint filter tile, so the L2 filter buffets are filled via unicast from DRAM. The delivery of tiles can either be interleaved or serial, as determined by the iteration order of the Read index generation FSM. Partitioning filters means that inputs are broadcast from DRAM to L2, as the same input must be convolved against all weights (excepting the edges).

The L2 buffets' Read FSMs distribute the second level of spatial partitioning. Each L1 processes a different output tile against the same weights, therefore output tiles are now unicast. One the other hand, the filter tiles are now broadcast from the L2 to all connected L1s. For inputs, the tile is unicast, but the halo region depicted in Figure 8 can be multicast simultaneously to two L1 buffets. In the steady state, this savings is doubled because of overlap with both previous and next partitions.

Multicast can represent a significant savings. Figure 10 shows an analysis of two Figure 9-style accelerators that differ only in support of multicast. No standard 1D convolutional benchmark exists, so we approximated data set sizes by examining the work to produce one output channel of VGG-Net16 [41], convolutional layer 13 and projecting the data ratios down into one dimension, giving an input size 21.78× larger than the filter. The multicast version reduces traffic between staging buffers to 58.9% of the unicast-only arrangement, and therefore performs only 58.9% of RAM accesses as well.

5 EVALUATION

In this section, we present an evaluation using buffets. We first discuss our evaluation methodology and experimental setup (Section 5.1). We then evaluate buffet-integrated accelerators compared with state-of-the-art accelerator memory systems (Section 5.2 and Section 5.3). In particular, we show that using buffets can result in significant better performance and energy efficiency than using DMA- and cache-based memory system.

5.1 Methodology

To better understand the costs of buffets, we implemented a buffet in RTL. The design supports all the buffet operations and is fully parameterizable so that it can be attached to RAM of different sizes. We synthesize the design with Synopsys Design Compiler with a commercial 16*nm* technology with a 1*GHz* frequency. We use Synopsys PrimeTime PX for accurate power analysis.

Table 3 shows the area and energy characterization of a buffet designed for an 8KB RAM, a representative RAM size in commonly-used accelerator kernels. RAM estimates are based on CACTI [26] with the same technology. We demonstrate that buffets are a lightweight mechanism that can be efficiently integrated as an accelerator's on-chip RAM.

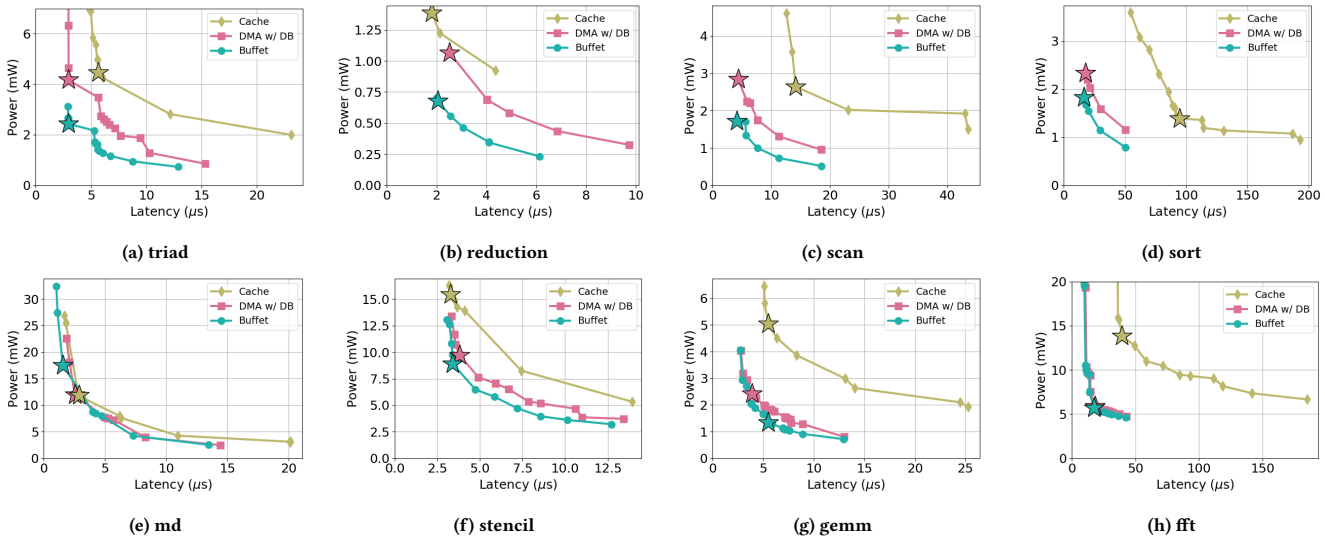

Figure 11: Power-Performance Pareto frontiers for buffet-, DMA-, and cache-based accelerator designs for SHOC benchmarks. Energy-optimal points are shown as stars.

To capture cycle-level behavior of buffets on a range of different accelerator benchmarks, we use a modified version of gem5-Aladdin [39]. Gem5-Aladdin is a cycle-level, power-performance-area simulator that captures the behaviors of both CPUs and specialized accelerators. Specifically, gem5-Aladdin models two types of memory systems for accelerators: DMA-managed scratchpad and private cache. We augment gem5-Aladdin's DMA-managed scratchpad model to support the buffet operations and proposed hardware components. The area and energy costs of a buffet are based on our RTL implementation. Our evaluations run on eight workloads from the SHOC benchmark suite [11], the same benchmark suite that was validated in Aladdin [38].

5.2 Buffet versus DMA versus Cache

To quantify the performance and energy efficiency of accelerators that integrate buffets, we perform a comprehensive design space exploration sweeping a range of accelerator design parameters, listed in Table 4. In this analysis, we fix algorithmic design choices, e.g., tile sizes, for all the configurations. Figure 11 shows the performance-power Pareto curves for each SHOC benchmark, distinguished by memory system types: cache, DMA with double-buffering (DB), and a buffet. The energy-optimal design point for each memory system is labeled with a star of the corresponding color. All the DMA design points apply double-buffering for better throughput.

Table 3: Buffet Area and Energy Characterization.

Component	Area (um^2)	Area %	Energy ($pJ/Access$)	Energy %
Buffet Control	446	2%	0.47	14%
RAM	17,571	98%	2.98	86%
Total	18,016	100%	3.45	100%

We see that a buffet-based memory system delivers better performance and power efficiency than DMA- and cache-based systems across all the benchmarks. Cache-based designs tend to have higher power cost and lower performance due to cache's expensive structures, e.g., tag array and MSHR, to support implicit data orchestration. Buffet's advantage over DMA is more pronounced for benchmarks in the top row, because these benchmarks are streaming applications without much local reuse. In this case, the execution of these kernels is heavily data movement constrained. Buffet's fine-grained synchronization between Fill and Read operations efficiently overlap the data movement and execution. In addition, compared to DMA-managed double-buffering, a buffet-based design does not require an over-provisioned on-chip RAM, as the fine-grained overlapping opportunity is captured within a tile, leading to better power efficiency. In contrast, benchmarks in the second row have more data reuse and spend most of their execution time performing computation. As a result, we see the performance difference between buffet- and DMA-based accelerator designs is smaller. However, a buffet-based accelerator design still achieves lower power, and smaller area cost, as RAM over-provisioning is not needed.

Table 4: Design Space Parameters.

Parameters	Values [start:step:end]
Data Orchestration	Buffet, DMA, Cache
Datapath Lanes	[1:2:8]
Pipelining	Enable/Disable
Scratchpad Bandwidth	[1:2:8]

Parameters	Values [start:step:end]
Hardware Prefetchers	Strided
Cache Size	[1KB:2:32KB]
Frequency	1GHz
System Bus Width	32b

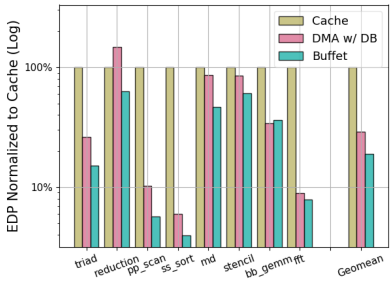

Figure 12: EDP comparison for buffet-, DMA-, and cache-based accelerator designs.

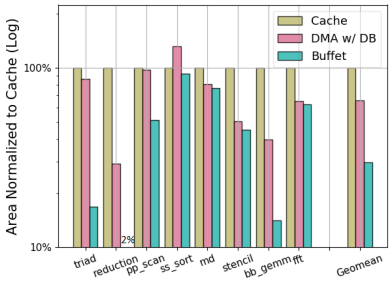

Figure 13: Area comparison for buffet-, DMA-, and cache-based accelerator designs.

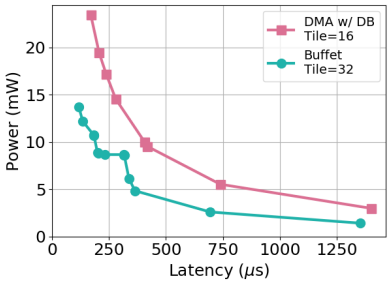

Figure 14: Design spaces of buffet- and DMA-based GEMM accelerators with the same on-chip RAM size.

Figure 12 and Figure 13 show the energy and area breakdowns of the energy-optimal design points (stars in Figure 11) for buffet-, DMA-, and cache-based accelerator designs. Buffet-based accelerator design overall achieves 2.3× energy reduction compared to DMA-based designs and 4.5× compared to cache-based designs. As mentioned previously, buffets generally require less storage space than DMA-based designs due to fine-grained data transfer and lack of double-buffering, leading to 2.1× area efficiency compared to double-buffered DMA.

5.3 Algorithm-Memory System Co-Design

Tile-size is an important algorithmic parameter that is highly dependent on available physical RAM and communication latency. Buffets' fine-grained synchronization support removes the strict need for over-provisioning physical RAM sizes in the design process. As a result, buffet-based accelerators can support much larger tiles with the same physical RAM budget, compared to double-buffered scratchpad for DMA-based memory system.

Figure 14 quantifies the benefits of buffet's efficient use of on-chip RAM. In this experiment, we focus on the design spaces of a tiled-GEMM accelerator with buffet- and DMA-based memory system under the same physical RAM budget. Buffet-based accelerators of tile size 32 delivers significantly better performance and power efficiency compared to DMA-based design of tile size 16. With buffet's fine-grained synchronization between Fill and Read, a buffet-based design can dedicate the entire on-chip RAM for a single tile, while double-buffered DMA must dedicate half of its physical RAM for next tile. In this case, a larger tile in a buffet enables more data reuse for each tiled computation and reduces the overall DRAM accesses, leading to better overall energy efficiency.

6 EXPERIENCES USING BUFFETS

We have taped out two domain-specific neural net accelerator test chips using buffets to store tiles of weights, inputs, and partial sums. We now share our non-quantitative learnings from this experience.

Test Chips: The first accelerator (details in [23]–although to avoid confusion the term "buffet" is not used in that work) is a proof-of-concept prototype that uses three levels of buffets: private L1s, shared L2, and an off-chip L3. Buffets for weights, inputs, and sums share the same RAM banks and individual buffets can be dynamically resized to use more or less memory. Design-time customizations are: (A) removing update paths from all buffets except sums, (B) multiplexing RAM ports for writes and fills, and (C) using bulk RAW-hazard tracking. The second chip is an efficient, specialized accelerator that was derived from the first with high design productivity. This uses private L1s and distributed L2s. Buffet RAM is hard-partitioned across data types, reducing area and energy. Details of this chip are not yet published.

Design Effort: One benefit was that buffets eliminate the need for designing customized data movement engine for inputs, weights, and sums. For example: weights, inputs, and partial sums have distinct data access patterns. Instead of spending engineering effort to design customized FSMs for each individual data type, we directly instantiate buffets with different design-time customizations to support these different access patterns. Second, buffets significantly simplify the data delivery and computation overlap in the system. For example, instead of manually designing double- or triple-buffering mechanism in hardware, buffets use the Shrink operation to signal the available space in the consumer and use the tail pointer to indicate data availability.

Using Buffets: One lesson is that buffets' flexible Shrink size results in a tradeoff between convenience and efficiency. Small shrinks are generally easier to generate addresses for—as they better leverage the encapsulation of modular arithmetic—but result in more credit-flow traffic and smaller bulk transfers. A second lesson is that the contiguous Shrink requires that data layout match unstaging order rather than access order. For example, simultaneously shrinking 1 column of each input channel requires a non-standard column-major/channel-minor memory order. This was not a restriction in practice as we used buffets' flexible access order in the L2 to transpose tile layout while filling the L1s.

7 RELATED WORK

A variety of approaches have been used for the data orchestration of accelerators. Many accelerators use a customized buffering solution based on their application [5, 6, 15, 35, 47]. While these proprietary buffering designs are efficient, they require significant engineering effort. The accelerator community needs a general reusable accelerator storage design to provide high performance with minimal effort. Buffets fill this need as an efficient and full featured solution for accelerator designs. Table 5 provides a comparison of buffets to

Table 5: Categorizing storage idioms for accelerators.

Jenga [44]	Implicit	Coupled
DeSC [17]	Explicit	Decoupled
PDAE Cache Prefetch [4]	Implicit	Coupled
PDAE DAE [4]	Implicit	Decoupled
Stash [24]	Implicit	Coupled
Accelerator Store [30]	Explicit	Coupled
Patch Memory [8]	Implicit	Coupled
LEAP Scratchpads [1]	Implicit	Coupled
CoRAM [7]	Explicit	Coupled
Stream dataflow [33]	Explicit	Decoupled

other works that provide general storage solutions for accelerators. Table 6 provides a detailed comparison of buffets to previous works that use the explicit decoupled idioms.

Jenga [44] treats the cache SRAM banks as a general pool that can be used to create distributed virtual cache hierarchies that are targeted to given application. This solution uses the implicit coupled idioms to provide a cache based solution. This approach provides a more efficient solution than a traditional cache but still has more overhead than an explicitly managed memory architecture.

Decoupled Supply-Compute (DeSC) [17] uses a traditional DAE approach to divide memory accesses instructions from compute instructions and place them on separate datapaths, connected by a queue. However, the queue size in DESC is not architecturally visible, and therefore cannot be used as an explicit target for determining data tiling size. Additionally, DeSC uses a traditional DAE FIFO queue which prevents random iteration without transferring the data to the L0 registers thus limiting access order and preventing in-place updates. It also has a high non-RAM overhead associated with having a full OoO processor as the access engine.

Prefetching with decouple access-execute (PDAE) [4] has two hierarchies for data orchestration. PDAE uses a prefetching cache for accelerators with regular access patterns and a traditional decoupled-access-execute approach for systems with irregular access patterns. The prefetching cache has traditional cache overheads while the DAE approach has similar iteration restrictions as DeSC. Buffets provide a general structure that can be used with minimal overhead.

Stash [24] is a data staging scheme for accelerators that makes scratchpads more like caches. It unifies scratchpads into the global address space like a cache, but uses a specific user-provided translation function to fill the scratchpad on a miss. Translation reduces the number of explicit operations required to fill the scratchpad though the creation of a load-and-store-scratchpad. Stash does not use a decoupled fill approach, and explicit double-buffering is still required for concurrency.

The Accelerator Store [30] is a shared memory scratchpad that allows multiple logical buffers to be mapped onto a single scratchpad, similar to buffets. The logical buffers can be configured in FIFO, random-access, or a hybrid mode. However RAMs are directly accessed by explicit datapath load/store (or get/put for FIFO mode) operations, without any dedicated index generators and synchronized using bulk interrupts.

Patch Memory [8] is a domain-specific scratchpad specifically designed for data tiling in image processing accelerators. Users define patch parameters and a dataflow order between patches, and

Table 6: Comparison of contemporary explicit decoupled data orchestration approaches with categories from Table 2.

	DeSC [17]	Stream dataflow [33]	*Buffet*
Non-RAM Area	High	Low	Low
Access Energy	Low	Low	Low
Placement Policy	Workload-controlled	Workload-controlled	Workload-controlled
Multiple Fills in Flight	Straight-forward	Straight-forward	Straight-forward
Overlapped Fill and Access	Straight-forward	No	Straight-forward
Hier. Composable	No	No	Yes
Access Multicast	Workload-controlled	Workload-controlled	Workload-controlled
Data Availability Synchronization	Encapsulated (peek stalling)	Explicit (barrier)	Encapsulated (read stalling)
Access Order	Limited OoO	Arbitrary	Arbitrary
In-Place Updates	No	No	Yes
Removal Policy	Workload-controlled	Workload-controlled	Workload-controlled

the data is filled in a DAE style. Iterations into the patch use explicit loads and stores from the datapath without DAE. Patch memory is not hierarchically composable.

LEAP Scratchpads [1] are a memory architecture for FPGAs that allow users to define logically separate scratchpads that are mapped onto a unified cache structure with tags and traditional fill-on-load behavior. Concurrent accesses are possible to other logical scratchpads while one scratchpad is filling. Based on the use of caches, they are hierarchically composable.

CoRAM [7] is a memory architecture for FPGAs used as accelerators. CoRAM defines a set of operations that is used to construct custom fill engines and scratchpads that are programmed into the FPGA's logic and block RAM. Fine-grained synchronization is possible directly through FPGA signals. However CoRAM does not use DAE for iteration accesses, but rather a traditional load/store interface. Additionally no indication is given of an ability to compose CoRAM hierarchically, though this may be possible given the generality of the CoRAM operations.

Stream-dataflow [33] is an architecture and programming model based on streams and CGRAs. The data is stored locally in a scratchpad and streamed into a dataflow CGRA for efficient acceleration. The architecture uses an address generators for accessing the memory system and scratchpad. However, the synchronization of the accesses requires the use of explicit barriers. This need of barrier prevents overlapping fill and access. The Stream-dataflow architecture does not support in-place updates due to the flow of the data through the system. Furthermore their memory system was not designed to be composable.

8 DISCUSSION AND FUTURE WORK

The domain-specific accelerator era of heterogeneous computing is poised to tackle key problems in compute through the integration of application specific engines. However, achieving this efficiency requires tremendous effort to design and verify a wide variety of accelerators. We have identified commonalities in the efficient transfer and staging of data that can be leveraged to generate a

design effort and area efficient generalized accelerator memory system.

In this paper we made several contributions towards achieving a practical, reusable, accelerator-agnostic buffering idiom. We discussed why reusable idioms that have worked well in the general-purpose computing space fall short for accelerators. Through our taxonomy of approaches, we identified explicit decoupled data orchestration (EDDO) as the tactic that best allows accelerator architects to leverage their static workload knowledge for efficiency. We described buffets' operational behavior, and why encapsulating synchronization within the buffer interface increases efficiency and lowers design effort. We demonstrated the compositionality of buffets, and discussed the benefits multicast can bring. When compared with DMA managed double-buffered scratchpads and caches across a range of workloads, buffets offer 1.53× and 5.39× energy-delay product advantage respectively. With respect to area, we implement buffets in RTL and show that buffet-based accelerators achieve similar performance at 3.4× less area than caches, and 2% control overhead over an 8KB RAM.

We see many future extensions for this work. A key component is the development of a domain-agnostic auto-orchestration toolflow that can generate accelerator FSMs and hardware configurations. We believe that run-time reconfiguration of orchestration patterns can be added into this toolflow as well. This feature allows data orchestration to be tuned to the particular sizes and ratios of the current data-set, rather than settling for an average-case approach, which has shown to be advantageous for individual accelerators [14, 25, 29, 43]. We hope that accelerator-independent EDDO abstractions such as buffets help bring together the necessary engineering effort to create such a toolflow, rather than each accelerator creating their own tools. To facilitate adoption, we provide an open-source reference implementation of a buffet, written in Verilog, here:

https://github.com/cwfletcher/buffets

ACKNOWLEDGMENTS

The authors would like to thank Adrian Sampson and the anonymous reviewers for feedback that greatly improved this paper. Michael Fetterman, Thomas Bourgeat, Arvind, Daniel Sanchez, Hyoukjun Kwon, Tushar Krishna, Aamer Jaleel, Angshuman Parashar, and Sean Treichler contributed valuable discussions and feedback. This research was, in part, funded by the U.S. Government under the DARPA Software Defined Hardware (SDH) program (HR0011-18-3-0007). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

[1] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 25–28, February 2011.

[2] Cadence. *Stratus High-Level Synthesis Reference Guide*, 2015.

[3] 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 *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 269–284, 2014.

[4] T. Chen and G. E. Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *The Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[5] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 609–622, December 2014.

[6] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[7] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An In-fabric Memory Architecture for FPGA-based Computing. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 97–106, February 2011.

[8] J. Clemons, C. C. Cheng, I. Frosio, D. Johnson, and S. W. Keckler. A Patch Memory System for Image Processing and Computer Vision. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 1–13, October 2016.

[9] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-rich architectures: Opportunities and progresses. In *Proceedings of the Design Automation Conference (DAC)*, 2014.

[10] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni. An analysis of accelerator coupling in heterogeneous architectures. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[11] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[12] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[13] 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 *Proceedings of the Design Automation Conference (DAC)*, 2011.

[14] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2011.

[15] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale DNN processor for real-time AI. In *The International Symposium on Computer Architecture (ISCA)*, 2018.

[16] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159, 2017.

[17] T. J. Ham, J. L. Aragón, and M. Martonosi. Desc: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 191–203, December 2015.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 243–254, June 2016.

[19] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 674–687, Piscataway, NJ, USA, 2018. IEEE Press.

[20] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '00, pages 511–519, Piscataway, NJ, USA, 2000. IEEE Press.

[21] Intel. *FIFO: Intel FPGA IP User Guide*, 2018.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean,

A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2017.

[23] B. Khailany, E. Khmer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao, S. Srinath, C. Torng, S. L. Xi, Y. Zhang, and B. Zimmer. A modular digital vlsi flow for high-productivity soc design. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 72:1–72:6, New York, NY, USA, 2018. ACM.

[24] 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 *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 707–719, June 2015.

[25] H. Kwon, A. Samajdar, and T. Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2018.

[26] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 694–701, 2011.

[27] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. PuDianNao: A Polyvalent Machine Learning Accelerator. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pages 369–381, March 2015.

[28] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. Pudiannao: A polyvalent machine learning accelerator. *SIGPLAN Not.*, 50(4):369–381, Mar. 2015.

[29] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *The International Symposium on High-Performance Computer Architecture (HPCA)*, 2017.

[30] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks. The Accelerator Store: A Shared Memory Framework for Accelerator-based Systems. *ACM Transactions on Architecture and Code Optimization*, 8(4):48:1–48:22, January 2012.

[31] Mentor Graphics. *Catapult Synthesis User and Reference Manual*, 2016.

[32] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, March 2010.

[33] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. Stream-dataflow acceleration. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[34] NVIDIA Deep Learning Accelerator (NVDLA). http://nvdla.org, 2017.

[35] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 27–40, June 2017.

[36] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. Plasticine: A Reconfigurable Architecture For Parallel Patterns. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 389–402, June 2017.

[37] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 24–35. ACM, 2013.

[38] 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 *The 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 97–108, 2014.

[39] Y. S. Shao, S. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks. Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[40] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[41] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[42] J. E. Smith. Decoupled Access/Execute Computer Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 112–119, April 1982.

[43] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li. C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization. In *Proceedings of the Design Automation Conference (DAC)*, 2016.

[44] P.-A. Tsai, N. Beckmann, and D. Sanchez. Jenga: Software-defined cache hierarchies. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[45] L. Wu, A. Lottarini, T. Paine, M. Kim, and K. Ross. Q100: The architecture and design of a database processing unit. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2014.

[46] Xilinx. *FIFO Generator v13.1: LogiCORE IP Product Guide, Vivado Design Suite*, 2017.

[47] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh. GANAX: a unified mimd-simd acceleration for generative adversarial networks. In *The International Symposium on Computer Architecture (ISCA)*, 2018.