Symphony: Orchestrating Sparse and Dense Tensors with Hierarchical Heterogeneous Processing

MICHAEL PELLAUER, JASON CLEMONS, VIGNESH BALAJI, NEAL CRAGO, AAMER JALEEL, DONGHYUK LEE, MIKE O'CONNOR, ANGHSUMAN PARASHAR, SEAN TREICHLER, PO-AN TSAI, STEPHEN W. KECKLER, and JOEL S. EMER, NVIDIA, USA

Sparse tensor algorithms are becoming widespread, particularly in the domains of deep learning, graph and data analytics, and scientific computing. Current high-performance broad-domain architectures, such as GPUs, often suffer memory system inefficiencies by moving too much data or moving it too far through the memory hierarchy. To increase performance and efficiency, proposed domain-specific accelerators tailor their architectures to the data needs of a narrow application domain, but as a result cannot be applied to a wide range of algorithms or applications that contain a mix of sparse and dense algorithms.

This article proposes Symphony, a hybrid programmable/specialized architecture that focuses on the orchestration of data throughout the memory hierarchy to simultaneously reduce the movement of unnecessary data and data movement distances. Key elements of the Symphony architecture include (1) specialized reconfigurable units aimed not only at roofline floating-point computations but also at supporting data orchestration features, such as address generation, data filtering, and sparse metadata processing; and (2) distribution of computation resources (both programmable and specialized) throughout the on-chip memory hierarchy. We demonstrate that Symphony can match non-programmable ASIC performance on sparse tensor algebra and provide 31× improved runtime and 44× improved energy over a comparably provisioned GPU for these applications.

CCS Concepts: • **Computer systems organization → Heterogeneous (hybrid) systems;**

Additional Key Words and Phrases: Sparse tensor algebra, data orchestration, accelerators

ACM Reference format:
Michael Pellauer, Jason Clemons, Vignesh Balaji, Neal Crago, Aamer Jaleel, Donghyuk Lee, Mike O'Connor, Anghsuman Parashar, Sean Treichler, Po-An Tsai, Stephen W. Keckler, and Joel S. Emer. 2023. Symphony: Orchestrating Sparse and Dense Tensors with Hierarchical Heterogeneous Processing. *ACM Trans. Comput. Syst.* 41, 1-4, Article 4 (December 2023), 30 pages.
https://doi.org/10.1145/3630007

1 INTRODUCTION

Application domains such as deep learning, graph analysis, data analytics, and scientific computing are processing increasingly sparse data—including explicitly pruning dense data to reduce requirements on capacity or computation. Sparse data sets are highly diverse with densities ranging over

Authors' address: M. Pellauer, J. Clemons, V. Balaji, N. Crago, A. Jaleel, D. Lee, M. O'Connor, A. Parashar, S. Treichler, P.-A. Tsai, S. W. Keckler, and J. S. Emer, NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA, 95050, USA; e-mails: {mpellauer, jclemons, vbalaji, ncrago, ajaleel, donghyukl, moconnor, aparashar, sean, poant, skeckler, jemer}@nvidia.com.

0734-2071/2023/12-ART4 $15.00
https://doi.org/10.1145/3630007

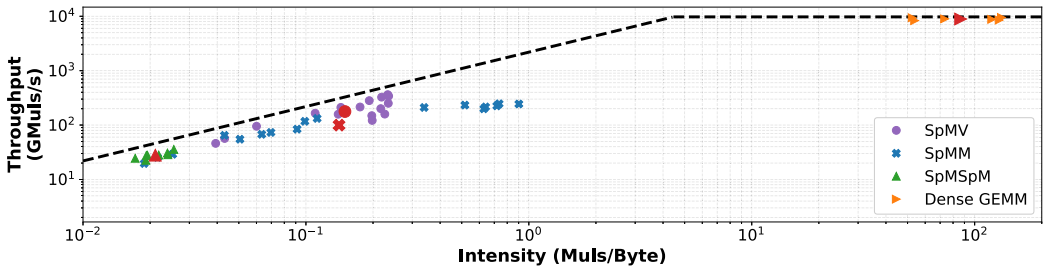


Fig. 1. A100 roofline [53] SIMD FP32 performance on a variety of sparse algebra kernels. Different points for a kernel correspond to different matrices from SparseTAMU [11], with average in red. Featured matrices have density values ranging from 3.869×10^{-7}% to 0.039%, with a mean density of 1.941×10^{-4}%, and 8.4 non-zeros per row on average.

many orders of magnitude, combined with complex non-uniform distributions. Recent research such as the TACO tensor algebra compiler [21] and the fibertree abstraction [49] have categorized the underlying principles beneath individual compression formats [7], increasing understanding and applicability of sparse tensor algebra. The common characteristic of sparse tensor workloads is low arithmetic intensity: the reuse of operand tensor A drops as operand B gets more sparse and vice versa, leading to a quadratic decay in computations when occupancies drop linearly.

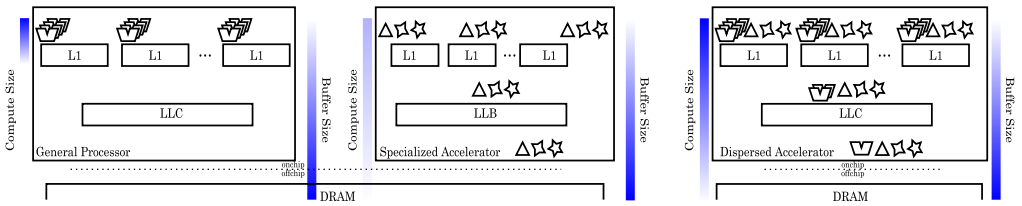
Platforms such as **Graphical Processing Units (GPUs)** and **Tensor Processing Units (TPUs)** target dense tensor algebra, which has high arithmetic intensity and regular tile sizes. The GPU's **Single Instruction, Multiple Threads (SIMT)** programming paradigm is appealing, since it achieves a high compute:buffering ratio while maintaining broad programmability, but this ratio over-provisions the compute roofline for sparse data. Beyond this, sparse workloads have several characteristics that make it challenging for GPUs to achieve peak memory roofline utilization: high divergence (both control and memory), difficulty aligning operands to vector math lanes, a large number of latency-sensitive memory indirections, and the need to process supplemental metadata to discover the effectual math operations [16]. Figure 1 shows the performance roofline [53] for the NVIDIA A100 GPU on **sparse-matrix by vector multiply (SpMV)**, **sparse-matrix by dense-matrix multiply (SpMM)**, and **sparse-matrix by sparse-matrix multiply (SpMSpM)** using the cuSPARSE library [34]. Looking at the X-axis, we see that performance is correlated to arithmetic intensity, however no sparse data set possesses sufficient reuse to saturate the A100's highly provisioned computation. Regarding the Y-axis, these applications achieve only 54%, 31%, and 60% of potential maximum memory bandwidth utilization, respectively. This demonstrates that the problem is deeper than simple re-provisioning of a general-purpose SIMT engine.

Custom hardware accelerators for sparse tensor algebra are an interesting alternative [16, 27, 38, 42, 47, 48, 58]. Because of the issues established above, these accelerators' main contributions are not in the datapaths themselves, but rather in their employment of data access **Finite-state Machines (FSMs)** and custom buffering hierarchies to approach their memory bandwidth roofline. Typically, this custom hardware performs lightweight metadata computation, filtering, and format translation to access the tensors in off-chip memory and store them efficiently in each level of a custom buffer hierarchy. The lower fundamental intensity means that these accelerators can provision smaller compute:buffering ratios than GPUs, to avoid wasting area and energy.

However, how should the area that is no longer needed by floating-point datapaths be filled up? Increasing on-chip buffer sizes is a natural idea, but we observe that this leads to little-to-no performance benefits due to the low-intensity phenomenon discussed above. In fact, larger buffers can counter-intuitively increase energy by harmfully raising energy-per-access for reuse. Thus custom sparse accelerators are in a strange situation where large chips are required simply

Table 1. Comparison of Hardware Execution Approaches for Various Accelerators

Architecture	Turing-Complete SIMT Datapaths	Dense Multiplier Array	**Sparse Data Orchestration** (i.e., custom buffers, intersect units, vector packing, etc.)
GPUs, pre-Tensor Core	✓		
Dense Tensor Accelerators (e.g., Eyeriss [6], TPU v1 [18])		✓	
GPUs, with Tensor Cores	✓	✓	
Sparse Tensor Accelerators (e.g., ExTensor [16], GAMMA [57])			✓
Symphony (this work)	✓	✓	✓



(A) Discrete General Processsor and Specialized Accelerator Blocks (B) Heterogeneous Hierarchical Processing

Fig. 2. Dispersing general-purpose and specialized units throughout the memory hierarchy. Traditional general-purpose processors only have computation capability at the leaves of the memory hierarchy. HHP places computation at every level, scaling throughput inversely to buffer size.

to provision high offchip I/O throughput, but filling the silicon with floating-point datapaths or large on-chip buffers is not strictly required.

In this article, we propose to leverage this extra available area to create a hybrid accelerator that can effectively accelerate both dense and sparse tensor algebra, while retaining the Turing-Completeness of GPUs, as shown in Table 1. Note that we do not propose *disjoint* co-processors as shown in Figure 2(A). Instead, we propose a novel arrangement that *disperses* key specialized hardware *elements* of the sparse accelerator throughout a dense accelerator's buffer hierarchy, as shown in Figure 2(B). Thus, instead of performing all computation at the leaves of the memory hierarchy as in a GPU or CPU, processing can occur local to each cache level. We call this approach ***hierarchical heterogeneous processing*** **(HHP)**, as we provision each level of the memory hierarchy with co-located computation bandwidth inversely proportional to the buffer size (but never lower than its bandwidth). Additionally, to emulate the custom buffer hierarchy of sparse accelerators, we allow the cache at each level to be reconfigured into hardware-managed scratchpads, queues, or buffets [40], similar to the ability of some GPU's L1 to serve as either cache or "shared memory" scratchpad. Compared to a traditional disjoint **System-on-Chip (SoC)**, the HHP approach leads to the following benefits:

(1) No private buffers between disjoint accelerator blocks, which traditionally can end up buffering redundant copies of data as it is moved between blocks.
(2) A reduction in "dim" silicon where large portions of the SoC chip are idle when a specific function is not running.
(3) A decrease in the granularity of offload and interaction between SIMT datapaths and specialized elements, which increases efficiency and applicability of the specialized blocks.
(4) An increase in flexibility and future-proofing, as the existing network-on-chip allows the reconfiguration of custom pipelining between specialized elements not found in a hardened co-processor.

We name our HHP system *Symphony* as an analogy to a musical orchestra where multiple sections of instruments play different sheet music while still contributing to the same harmonious orchestration. The Symphony approach allows construction of an accelerator that maintains dense tensor algebra performance, while allowing sparse tensor algebra to achieve roofline utilization. Beyond performance, we find additional compound energy-efficiency benefits for low-reuse data, as it moves through fewer buffers (i.e., hierarchy), is computed on by more efficient units (i.e., heterogeneity), requires smaller landing zones for data requests (i.e., decoupled access-execute), and can be sequentially processed in heterogeneous computations (i.e., pipelined). The specific contributions of this article are:

— A novel set of specialized, hardware units that support data orchestration and buffering idioms found in sparse tensor algebra algorithms. While these blocks are inspired by elements in custom sparse accelerators, we generalize them appropriately for tight integration with general-purpose computation.
— A method for integrating the reconfigurable hardware units into the memory hierarchy in a decoupled access-execute pipeline using the existing network-on-chip, without memory-based polling or synchronization.
— A specific HHP design point called Amadeus provisioned with similar area, compute and memory bandwidths as an NVIDIA Ampere GPU.
— A demonstration that these ideas can accelerate sparse tensor algebra algorithms while preserving and enhancing the performance of dense algorithms. We show that Amadeus achieves roofline performance on dense tensor algebra, and has mean improvements of 31× and 44× in performance and energy for a set of representative sparse tensor workloads over unmodified NVIDIA A100.

While we present Symphony integrated in a setting that uses GPU-like SIMT datapaths for dense compute and broad programmability, we anticipate that its key concepts can be "packaged" and deployed into several scenarios, including multicore CPUs.

Sparsity focus. Starting with the NVIDIA A100, some modern GPUs contain native support for *structured sparse* GEMMs with a single compressed matrix operand with a fixed ratio of 2:4—meaning exactly two non-zero-values every four elements. This represents a valuable efficiency boost to workloads where this ratio applies (particularly deep learning inference with pruned weights), but is outside the scope of this work. Symphony focuses on unstructured sparsity and graph applications that feature notably lower densities. For example, the matrices in Figure 1 have a maximum density of 0.039%. Additionally, these workloads feature multiple sparse operand tensors, and require programmability beyond fixed-function GEMMs. Therefore, when this article refers to A100 capabilities it means the programmable SIMT or dense tensor cores, as appropriate.

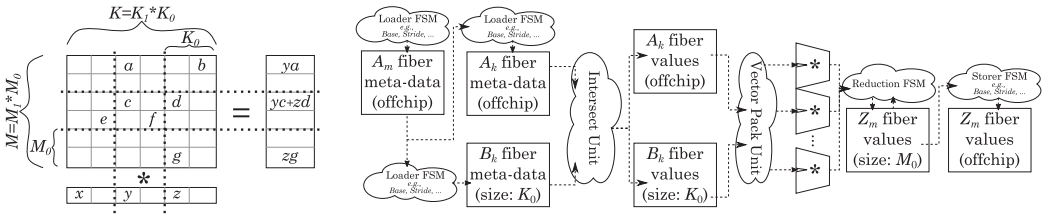
2 MOTIVATION AND OPPORTUNITY

This section details the key challenges to performance and efficiency that sparse tensor algorithms present to conventional broad-domain computing architectures. We describe the theory of general tensor algebra and use an example to demonstrate where moving from ASIC accelerators to general-purpose implementations results in notable efficiency loss.

2.1 Motivational Example: Sparse Matrix-Sparse Vector Multiply

In **Einstein summation (*Einsum*)** notation, a matrix-vector multiplication is expressed as:

$$Z_m = A_{m,k} * B_k.$$



(A) Example partitioning

(B) Custom ASIC with vectorized multipliers (one partition of size $M_0 * K_0$)

Fig. 3. Example sparse-matrix sparse-vector multiplictation (spMspV). ASIC is ExTensor-like [16] in style, using an $M \rightarrow K$ traversal order (i.e., B-stationary dataflow).

Mathematically this expression is agnostic to whether the underlying data structures are dense or sparse–it simply expresses a contraction of a particular set of *fibers* (i.e., rows or columns in this case) of the operand tensors [49]. The TACO tensor algebra compiler [21] takes this specification as input, along with a per-fiber format descriptor [7] and produces optimized format-specific code for a variety of sparse compression formats (e.g., CSR, CSC, COO, etc.). This results in memory-layouts that comprise *non-zero values*, and *metadata* that allows those values to be projected into the original dense tensor space. Exploiting the property that $\forall x . x * 0 = 0$, we use the metadata of A to avoid loading or operating on the ineffectual values of B, and vice-versa.

In both sparse- and dense-tensor algebra, the operands must be partitioned (i.e., tiled) to achieve parallelism and locality, which can be thought of as a transformation to the Einsum as follows:

$$Z_{m_2, m_1, m_0} = A_{m_2, k_2, m_1, k_1, m_0, k_0} * B_{k_2, k_1, k_0}.$$

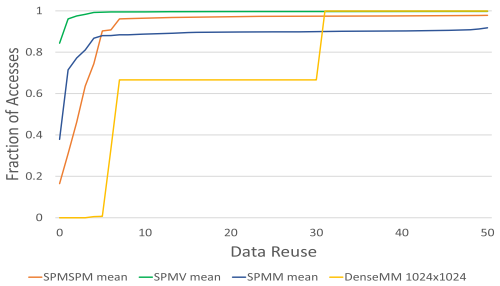
For example, in $m = m_0 * m_1 * m_0$, m_0 may represent the size of a tile, m_1 the number of parallel datapaths, and m_2 the number of temporal passes to cover the entire operand. Figure 3(A) shows an example concrete sparse matrix and vector along with an example concrete partitioning (m_2 and k_2 not shown). Partitioning dense tensors is straightforward, whereas partitioning sparse tensors is complex, with ramifications on occupancy, reuse, and load balance. This topic is outside the scope of this work but is well-covered by References [16, 27, 49, 58].

After partitioning, any Einsum equation can be paired with a traversal order as in References [23, 39], such as $M \rightarrow K$ or $K \rightarrow M$. This is sufficient information to derive a custom ASIC that performs exactly that equation, shown in Figure 3(B). This ASIC is capable of only computing this Einsum, but does so with maximal area- and energy-efficiency. Most notably, it uses custom scratchpads or buffets [40] to stage the B and Z operand tiles for reuse, whereas the A tile is unbuffered as it is algorithmically known to have no reuse. Additionally, it uses custom *intersection* datapaths to process metadata and find effectual computations as in ExTensor [16], and it uses custom *reduction* datapaths to align read-modify-writes of partial sums, as in Phi [27].

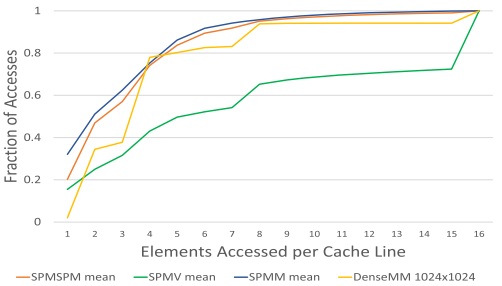
Of course, anything done by an ASIC can be done by a Turing-complete CPU or GPU, at reduced efficiency. In the remainder of this section, we discuss in detail the key sources of efficiency loss that these platforms suffer on these types of algorithms.

2.2 Challenge: Moving Data Too Far

General-purpose cache hierarchies aim to provide good quality of service without any knowledge of the workload. Unfortunately, sparse data structures exhibit low reuse of both metadata and data values. Figure 4(a) shows the cumulative reuse for dense matrix multiplication and the three sparse workloads from Figure 1, measured using the binary instrumentation tool NVBit [51]. The dense workload shows the effects of static data tiling via the discrete jumps in reuse. In all, 66% of data in the dense matrix multiply is reused eight or more times, while only 8% of data in the



(a) Cumulative distribution of temporal data reuse.



(b) Cumulative distribution of spatial locality.

Fig. 4. Characterization of workloads from Figure 1.

sparse workloads is used eight or more times. In a conventional cache hierarchy, low reuse data is needlessly deposited into multiple levels of the memory hierarchy and consumes energy as it traverses the interconnect to the processing units.

In addition to data, conventional architectures also move *addresses* farther than custom accelerators. All of their addresses are generated in the leaves of the memory hierarchy, and must be transmitted through all levels. Beyond the energy for moving addresses, the long latency of accesses through a deep memory hierarchy places structural pressure on buffering resources. Entries in the register file, scoreboard, and miss address files must all be allocated for each outstanding memory operation, which remain idle for the latency of the round-trip access. The cost of these idle resources becomes expensive with the large number of outstanding memory references needed to cover long memory latencies. Likewise, many contexts—e.g., threads or warps—can be required to tolerate long latencies, requiring additional register file capacity to manage the contexts' state. The custom ASIC in Figure 3(B) can physically locate low-locality data near the offchip interface to eliminate data movement costs, as seen in *processing near-memory* approaches.

2.3 Challenge: Moving Unnecessary Data

In addition to low temporal data reuse, sparse tensor algorithms often exhibit low spatial data locality. In our spMspV example, the access of A and B values is effectively a gather, whereas the off-chip store of Z is a scatter. Figure 4(b) shows that this matters in practice: all three sparse GPU algorithms shown previously access a substantive number of cache lines that only have one useful element: 15% for SpMV, 32% for SpMM, and 20% for SpMSpM. In contrast, dense MM has a single access to only 2% of lines and again demonstrates the regular spikes from tiling. SpMV has a notable spike for fully used lines for its streaming metadata.

Low spatial locality is doubly deleterious for GPUs compared to CPUs, as SIMT datapath widths typically match L1 cache line size. Thus, poor memory-level spatial locality often directly translates into idle datapaths via memory divergence. In Figure 3(B), the flow-forward nature of the pipeline means that a small *vector pack* buffer can line up the post-gather operands to match the vector datapaths, as in SpArch [58].

2.4 Challenge: Overly Coupled Access and Execute

Contemporary sparse tensor accelerators have consistently found that their workloads are best expressed as producer-consumer pipelines [4, 29] to expose parallelism across pipeline stages instead of relying only on data parallelism. Decoupling producers from consumers so that they can naturally run-ahead and catch up leads to better latency tolerance, similar to decoupled access-execute processors [46]. Custom accelerators can connect producer and consumer stages via simple hardware queues, and use dedicated credit-flow hardware to synchronize as appropriate, without active

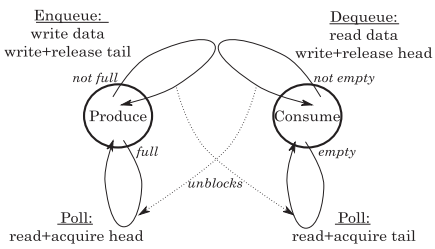
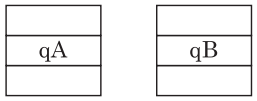


Fig. 5. Thread decoupling via memory polling on a single-producer single-consumer queue as in Reference [36].

```
while (kA < pA1_end) {
  qA.Push(A1_crd[kA], kA);
  kA++
}
qA.Finish();
```

(A) Address gen. and coordinate fetch




```
while (qA.IsDone() && qB.IsDone()) {
  [cA, kA] = qA.Peek();
  [cB, kB] = qB.Peek();
  if (cA == cB)) {
    Z_vals[kZ] = A_vals[kA] * B_vals[kB];
    kZ++; qA0.Pop(); qB0.Pop();
  }
  else if (cA < cB) qA.Pop();
  else qB.Pop();
}
```

(B) Intersection and multiply/add

Fig. 6. Tensor contraction from the TACO compiler [21] hand-decoupled to tolerate latency.

polling or memory transfers. This approach frees up their on-chip RAM for exploiting data locality in operand tensors.

Most general-purpose processors have difficulty expressing and efficiently implementing decoupled producer-consumer relationships. Loop unrolling and software pipelining within the thread couples the access and execute operations to a ratio and schedule that are fixed statically and cannot adjust to the dynamic operand occupancy. Decoupling access and execute into different threads is inefficient as the cross-thread synchronization is typically implemented through memory. Figure 5 shows an example of a simple polling synchronization operation through memory, which requires five memory operations per data item transferred. While some of this overhead can be amortized by aggregating multiple transferred items into a single synchronization, this approach increases the coupling and still results in extra stores to communicate data and expensive synchronization (e.g., fences, polling, or barriers) to signal control. An ideal system combines data and synchronization into a single fine-grained transaction.

2.5 Challenge: Inefficiency of Broad-Domain ISA

Specialized hardware units represent an opportunity and a liability for accelerator architectures. They can provide substantial performance and efficiency improvements for the operations they execute. For example, Figure 6 shows the code required to execute two common operations found in sparse tensor algorithms: (1) address generation/coordinate fetch and (2) intersection and multiply-add. When compiled by nvcc, the address generation loop takes 4 GPU instructions to generate one address (ignoring overheads for queue polling). Similarly, it takes 16 GPU instructions to perform the intersection loop, including three branches. In contrast, specialized intersect units such as shown in Figure 3(B) can output multiple operations per cycle [19, 44, 58], using lower

energy and area. Adding custom instructions to a general purpose processor can help it approach the efficiency of specialized units, but only if the data can be efficiently delivered via the register file, which limits applicability and ties the hardware closely to the leaves of the memory hierarchy.

However, specialized units are idle when their function is not being executed, leaving unused resources that could serve better purposes. In a traditional coarse-grained System-on-Chip, entire accelerators can idle when the workload does not match its exact supported configuration. If specialized hardware is not used judiciously, then it can end up as "dim" silicon.

2.6 Symphony Motivation

We see an opportunity to address all of these challenges simultaneously via *hierarchical heterogeneous processing*. We distribute the specialized data orchestration hardware of the custom ASICs throughout the general-purpose memory hierarchy. This architecture allows us exploit the efficiency boost of specialization and decrease energy and latency as in processing-near-memory ASICs. To combat the danger of dim silicon, we add key runtime reconfiguration options, and allow the assembly of different reconfigurable pipelines controlled by the application, thus enabling computation to pass between specialized and general-purpose datapaths as needed. In the remainder of this work, we propose the precise mechanisms to accomplish this vision and discuss practical methods for composing the mechanisms into a unified system.

3 SYMPHONY ARCHITECTURE ELEMENTS

To realize this vision, we must address a key challenge of specialization: namely, the tension between efficiency and programmability. Traditional fixed-function accelerators obtain maximal benefits from specialization, but reduce flexibility and generality. In contrast, fine-grained reconfigurable platforms such as FPGAs offer maximal reconfiguration possibilities, but lose out on the efficiency benefits of specialized hardware.[1] The overarching concept behind Symphony is to achieve the best of both worlds by providing a set of *medium-grained* specialized units, called ***logical elements*** **(LEs)**, that are responsible for orchestrating data buffers, performing compute operations, and generating addresses for data movement through the system. Programs are represented as dynamically configured pipelines of these logical elements, connected in producer/consumer relationships. A similar approach is explored in concurrent work called the Sparse Abstract Machine (SAM) [59]. See Section 7 for a detailed comparison. The main benefit of this approach is that it allows a strategic deployment of specialized hardware for the most common and impactful operations, while simultaneously keeping the set of configuration options tractable to the programmer (or compiler), without requiring EDA-style algorithms such as place-and-route.

The element is called "logical," because the physical datapath that implements it may (or may not) be time-multiplexed with other elements of the same type, or even elements of different types, as appropriate. In this way, the *logical topology* of elements need not match the physical layout of the chip, and producer-consumer relationships can be satisfied via virtual-channel-style routing in a **network-on-chip (NoC)** without requiring physical paths or expensive memory-mapped I/O. Thus, Symphony logical elements are analogous to threads in software, but for hardware finite-state machines instead of sequences of instructions. Physically, the computation datapaths that execute element functionality are distributed throughout the on-chip memory hierarchy, co-located with the RAMs they access for energy efficiency and latency minimization.

[1] In fact, the recent trend in FPGAs is to add an increasing number of heterogeneous specialized fixed-function elements, perhaps indicating a future convergence with other approaches.

Table 2. Data Orchestration Element Specifications

Element	Norm. Area	Norm. Energy	Notable Parameters		Elements Per Cycle		
			Design-Time	Config-Time	SW	HW	Mode
Pattern Gen.	12.9	0.4	Max. loop levels	Num. active loop levels, Base, Stride, Repeat	3.37	8 (2.4×)	affine
Distributor	30.7	1.0	Max. channels out	Num. active channels out, Modes: Demux/Round Robin/Last-Used-Last-Served	1.14	8 (7.0×)	demux
Collector	19.1	0.3	Max. channels in	Num active channels in, Modes: Mux/Reduce/Round Robin/Last-Used-Last-Served	1.14	8 (7.0×)	reduce
Merger	28.4	4.5	N/A	Modes: Intersection/Union/Filter	0.05	8 (160×)	i-sect
Adapter	21.8	1.6	N/A	Modes: Serialize/Deserialize/Vector Pack	max 0.8	max 8 (10×)	pack

Area and dynamic energy is normalized to a 32-bit floating point multiply-accumulate. Software comparison was done using the SIMT computation elements with vector width 8. Measurements obtained producing eight 32-bit data elements per logical element. Area and energy estimates are derived by composing instances of low-level logic (ALUs, shifters, crossbars, latch arrays, etc.) from post-layout data of a production design in 16 nm technology.
Common Design-Time Parameters: Data type, Vector width, Max. logical contexts, Max. parallel datapaths, Physical channels in/out.
Common Config-Time Parameters: Number of active logical contexts, Logical-to-physical-channel map, Channel routes in/out.

Logical elements and their hardware implementations have the following attributes:

— Operate autonomously without being step-by-step controlled by a central compute unit. Thus, they can be used for both decoupled-access-execute and processing-near-memory scenarios.
— Can be implemented using finite state machine driven datapaths for efficiency or as an instruction-based programmable engine for greater flexibility.
— Can be implemented in a time-multiplexed fashion to improve hardware utilization by having a single hardware unit provide the functionality of multiple *logical* contexts or using parallel hardware to improve throughput.
— Can directly communicate directly with each other using the underlying NoC, without using memory-mapped data structures, polling, or barriers.

This FSM-based approach does increase the risk of over-specialized silicon that is not useful across a wide range of workloads. To combat this pitfall, we carefully generalize the building blocks found in fixed-function tensor accelerators with a degree of configurability, increasing post-deployment applicability. Arguably, specialized hardware for data orchestration is broad-domain, and we expect these elements to find applicability outside of low-intensity sparse tensor algebra.

3.1 Logical Element Specifications

The specific logical elements we propose (Figure 7) fall into three categories: data orchestration, storage and compute.

Data Orchestration elements provide specialized functionality for common operations related to accessing, broadcasting, filtering, reducing, and outputting data. Table 2 details their exact parameters, and characterizes the increase in efficiency gained via specialization.

— **Pattern generators (Figure 7(A)):** configurable hardware to efficiently generate affine patterns of numbers at high throughputs. The affine pattern allows more addressing options than traditional direct memory access generators, such as deeper nestings of address loops and sliding window patterns, as well as being applicable for situations beyond addressing storage, such as generating repeated patterns of distribution across parallel channels.
— **Mergers (Figure 7(B)):** perform unions, intersections, and filtering inspired by ExTensor [16]. They are used to eliminate computations on compressed tensors by taking the union or intersection of streams of *coordinates* of the two operands of an **Arithmetic-logic Unit (ALU)** operation.

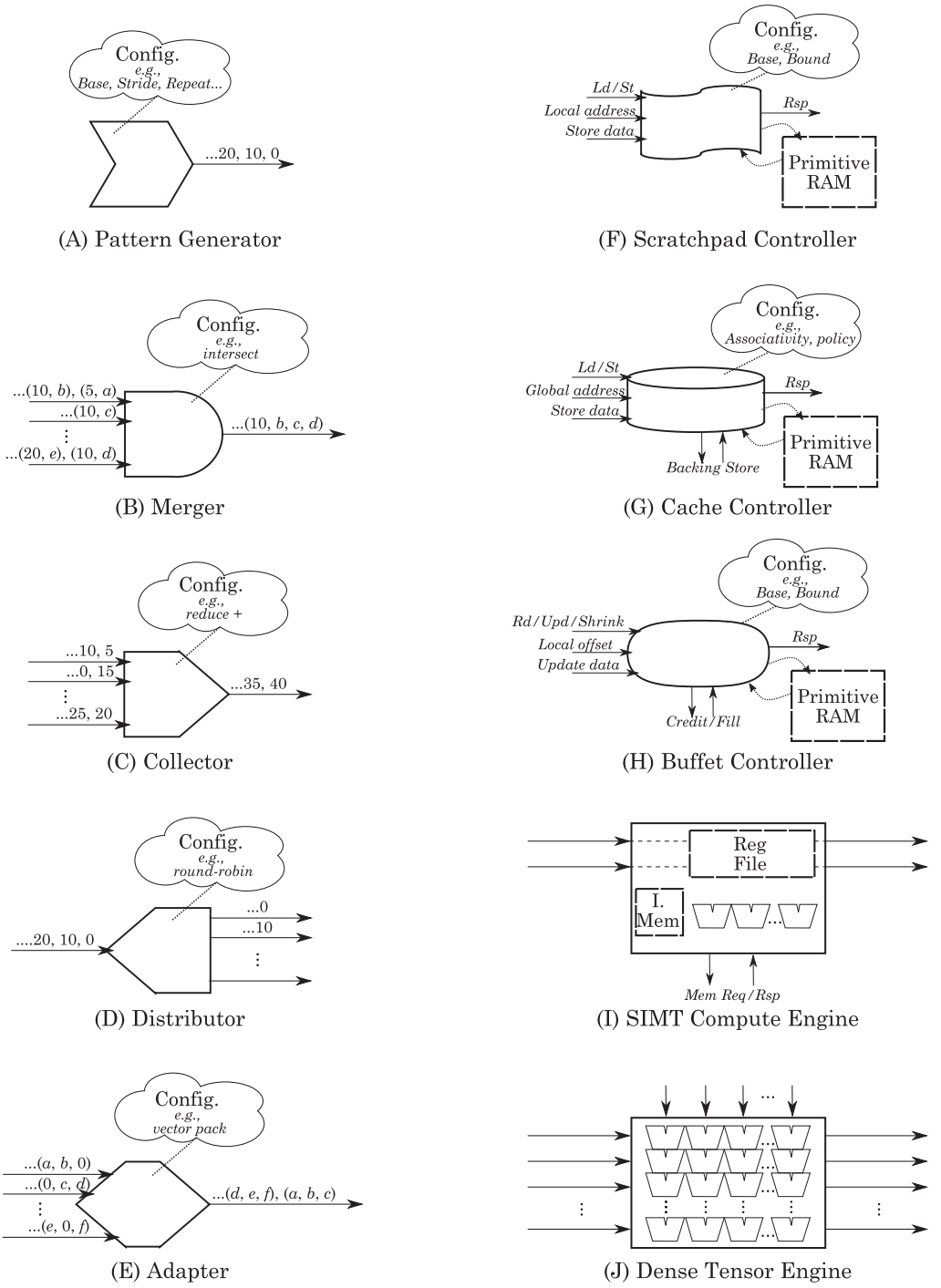


Fig. 7. Symphony Logical Elements with example dynamic configurations. Arrows represent logical channel end-points that are connected using the network-on-chip as described in Section 3.2.

— **Collectors (Figure 7(C)):** These units can perform reduction operations either as ordered merges of incoming data or update-style reductions of an indexed array of values inspired by Phi [27].
— **Distributors (Figure 7(D)):** These units unicast or multicast the values from an input stream down multiple output streams using a determined distribution scheme, or an external control channel.
— **Adapters (Figure 7(E)):** Efficiently perform a range of data transformations, such as packing and unpacking data streams. Also can realign sparse data for vector datapaths.

Storage elements configure on-chip RAM pools into higher-level storage idioms:

— **Scratchpad Controller (Figure 7(F)):** provides a configurable finite address space in a local storage array, similar to a GPU's *shared memory* scratchpad in its L1.
— **Cache Controller (Figure 7(G)):** leverages the local storage array for storing the tags and data while the element implements a set associative cache. Multiple cache elements may be connected together across different levels to provide a cache hierarchy.
— **Buffet Controller [40] (Figure 7(H)):** allows the system to provide *explicit decoupled data orchestration*, where the program locally controls data accesses at all levels of the storage hierarchy.

Computation elements can be placed throughout the system (e.g., near memory) to perform more general computation or additional data orchestration functions not supported by a specialized data orchestration element. Variants of these units can be parameterized at design time to support different throughputs of scalar, vector, or tensor computations.

— **SIMT compute engines (Figure 7(I)):** These instruction-based vector computational units can perform integer, floating-point, and/or binary operations. The instruction set is extended to operate on channels directly via the register file, in addition to traditional loads and stores.
— **Dense tensor engines (Figure 7(J)):** These specialized TensorCore-like datapaths exploit multiply-accumulate operations with high arithmetic intensity, operating on streams of dense input tiles and generating dense output tiles.

3.2 Physical Implementation and Networking

The logical element functionality described above are provided by hardware and are designed adapt to the conflicting demands of maximizing throughput, minimizing latency, and avoiding idle hardware. To balance these demands, Symphony hardware employs a variety of tactics. For efficiency, a single hardware unit may support more than one type of logical element. In addition, a hardware unit may hold the state or *context* of one or more logical elements that are *configured* to it simultaneously by a program. When more than one logical element is configured to a single physical unit, that unit may execute their operations in parallel up to the provisioning of its datapaths. If the number of contexts exceeds the parallel execution capability of a physical unit, then the unit may time-multiplex its contexts, which can keep the utilization of the hardware high [30].

Figure 8(a) shows the general hardware template used for creating physical units that support logical elements. Each physical unit has a configuration controller that manages the logical element contexts based on the configuration sent from a host. The controller communicates with the Physical Manager, which is responsible for tracking resource usage such as available logical contexts and datapaths. The configuration controller also communicates configuration data to active logical element contexts and datapaths. The input logical channel map is used to route incoming data to the correct hardware in the physical unit based on incoming channel stream id. The output logical

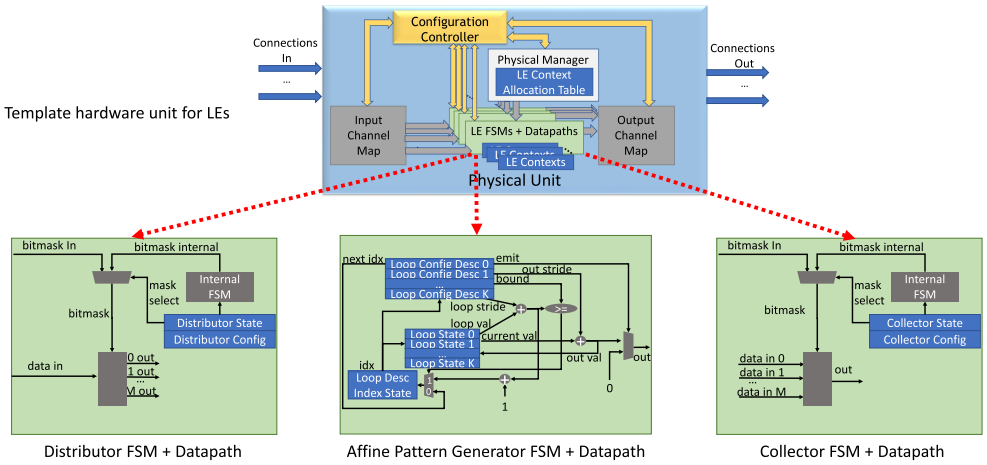


Fig. 8. Example hardware unit design for logical element execution, with alternate specializations of the green area.

Table 3. Example Storage Requirements

LE Context	Pattern Generator State per LE (bytes)	Distributor State per LE (bytes)
LE Config	30	12
LE Active State	79	5
Input Logical Channel Map	1	1
Output Logical Channel Map	1	48

channel map routes the data to the correct physical channel for outbound **latency-insensitive (LI)** channel streams.

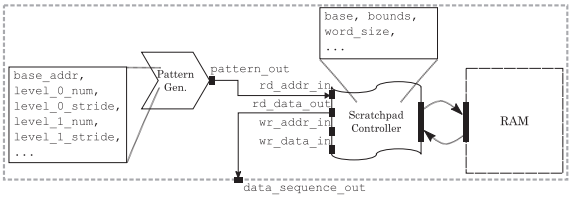
Figure 8(b) shows the specific datapath of the Distributor, which has hardware to broadcast data based on a mask enable bitmask. The mask enable can come from either an input channel or from an internal custom finite state machine. The internal FSM supports multiple patterns of bitmask generation based on the mode of operation such as round robin selection of the output channel or least recently used. In contrast, Figure 8(c) shows the specific FSM and datapath for an affine pattern generator. Nested loops are described with loop descriptors. Each loop descriptor contains base, repetition count, and stride for the loop along with an offset to the next descriptor. This loop descriptor is used in conjunction with an output stride for each loop to generate a pattern of outputs from the pattern generator. Table 3 shows the amount of memory associated with each LE for the given functionality. In general, we find 128 bytes per LE context seems to be sufficient.

Semantically, logical elements are connected via *logical channels*, which are network channels that provide LI channel semantics [5, 12]. To support this, Symphony hardware incorporates a NoC that supports flexible circuit reconfiguration allowing construction and destruction of circuit connections, and provides routing of both unicast and multi-cast communications.

4 PROGRAMMING AND CONFIGURATION

4.1 Algorithm Representation

A Symphony program consists of the instantiation and configuration of the logical elements described above. Each element is configured for a particular task in the program, e.g., a pattern



(a) The SequentialMemReader is a simple transfer object for reading a contiguous tensor in order.

```
SequentialMemReader::SequentialMemReader(Name& name, Tensor& t,
                                         cached = false) {
  pgen_cfg = new PatternGenConfig();
  pgen_cfg.base[0] = t.base_address();
  pgen_cfg.stride[0] = t.elem_size();
  pgen_cfg.num[0] = t.total_size();
  pgen = new PatternGenerator(name + "_pgen", pgen_cfg);

  scratch_cfg = new ScratchpadConfig();
  scratch_cfg.word_size = t.elem_size();
  scratch_cfg.line_size = t.elem_size() * 8;
  scratch_cfg.read_only = true;
  scratchpad = new LogicalScratchpad(name + "_scratch", scratch_cfg);
...
```

```
...
  storage = cached ? new LogicalCache(name + "_storage", t) :
                     new LogicalRAM(t, name + "_storage", t);

  connect(pgen.pattern_out, scratchpad.rd_addr_in);
  connect(scratchpad.prim_ifc, storage.prim_ifc);

  data_sequence_out = scratchpad.read_data_out;
}
```

(b) Code sample of Symphonic CUDA for creating the SequentialMemReader.

Fig. 9. Example demonstrating the encapsulation of multiple logical elements into a compound *transfer object* used in a library for data orchestration.

generator may be configured to generate a sequence of affine addresses by connecting its output channel to the address input channel of a scratchpad or cache controller. The connection of logical circuits in the program expresses a *pipeline*, i.e., a directed graph of producer/consumer relationships connected via the NoC. Performance and energy-efficiency of an algorithm is maximized when all possible code that can be offloaded from general-purpose SIMT units is covered by logical elements. However, these Turing-complete elements are always available as a fallback to run arbitrary compiled code.

4.2 Programming Logical Elements

After the high-level structure of a program is determined, the next task is to program the logical elements to implement each worker's functionality.

4.2.1 Low-level Interface. Expert programmers can directly program the data-orchestration and storage elements by setting configuration-register values. For example, an affine pattern generator is programmed by setting the nest depth and the descriptor configuration for each loop in the nest representing the generated pattern (Figure 8(c)). We expose these configuration registers via a simple memory struct-based interface.

Turing-complete computation engines are programmed using assembly language and a GPU-like SIMT programming model. This code can describe the main computations in the inner loops of an algorithm or express uncommon data-orchestration patterns that cannot be effectively mapped onto a data-orchestration element. To control input and output data streams, the programmer interacts with register-mapped channels using instruction operands.

4.2.2 Symphonic CUDA. The low-level programming method described above offers precise control over the hardware but is tedious to use. To remedy this limitation, we created a higher-level language for encapsulating multiple logical elements into compound *transfer objects*. These components have internal configuration state and connections, but hide those details from the user in a black-box fashion, as shown in Figure 9(A). We use this technique to code a set of common data-orchestration motifs (such as tiled data movement between storage levels and strided

distribution/collection of data) into a library of reusable higher-level abstractions. Transfer objects can also encapsulate inline assembly code to be mapped onto computation elements. These transfer objects are then exposed through an extension of **Compute Unified Architecture (CUDA)** that we call ***Symphonic CUDA* (SCUDA)**, a representative sample of which is shown in Figure 9(B). Our library currently contains 14 data-orchestration transfer objects, with each of our evaluation workloads using almost half of them on average.

We have developed a compiler pass that translates a SCUDA program into the hardware configuration and bindings for all logical elements needed to execute the program, including expanding transfer objects into underlying primitive logical elements. Additionally, for dense tensor algebra workloads, we use an automated optimizer [39, 54] to find optimal mappings, followed by manual implementation of those mappings in SCUDA. We expect that this manual translation step can be automated, and that the set of applicable workloads for automated mapping can be broadened. For example, the concurrent work on the Sparse Abstract Machine [59] translates TACO [21] sparse tensor expressions into dataflows and configuration of LE-like hardware.

As more complete example, Figure 10(A) revisits the sparse-matrix-sparse-vector example from Figure 3, showing the same dataflow and execution order implemented as a logical element pipeline instead of a fixed-function ASIC. Transfer objects such as `SequentialMemReader` instantiate pattern generators to load the metadata, which is used to perform intersection, allowing the appropriate nonzero values to be gathered, packed into dense vectors, and computed on. The result is then reduced to a final sum and scattered back to the Z tensor memory. Notably, even though the program contains complex memory interaction patterns, it can be expressed as a flow-forward pipeline without round-trip memory accesses, improving its ability to saturate DRAM bandwidth. The compound transfer object abstraction is overhead-free and is removed entirely during compilation.

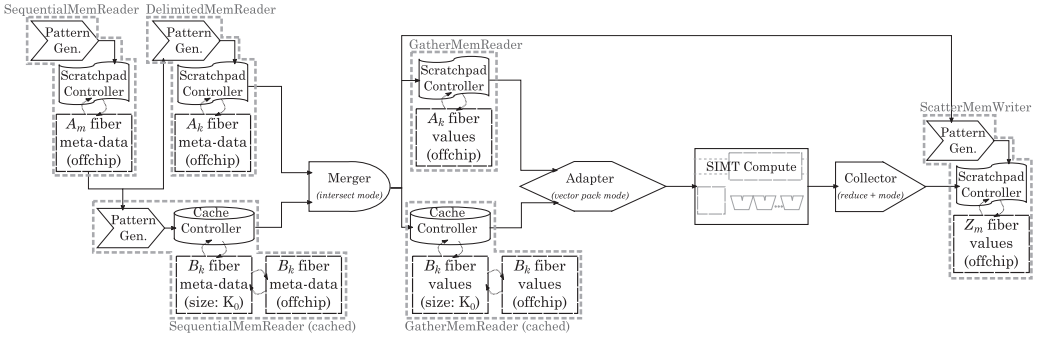
As shown in Figure 10(B), the program also includes a *binding* that specifies which physical units in the system run which logical element tasks. To be legal, the binding must obey the restrictions of the hardware. For example: the intersection task cannot be bound to hardware that only executes pattern generation.[2] In this example the number of physical pattern generators and buffer controllers is less than the logical elements used by the program so multiple logical contexts are bound to the same physical unit. Channels between producers and consumers are routed via the shared network. The SIMT compute program interacts directly with these channels to keep throughput high and avoid becoming the rate limiting step. Of course, the overall throughput and latency depend on the provisioning of the physical system. In general, well-studied techniques such as overlaying a small number of logical contexts onto the same units are sufficient to achieve balanced utilization.

4.3 Configuration Procedure

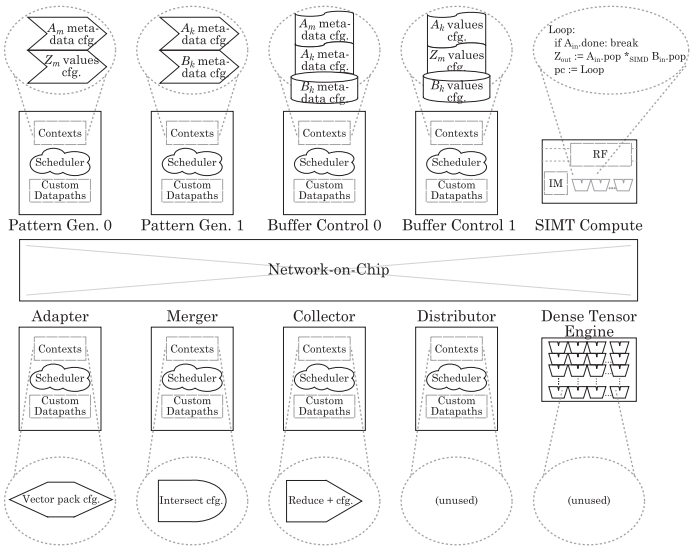
Configuration begins with the configuration controller of each active hardware unit receiving a set of configurations to be executed. Each set of configurations is subdivided into epochs, where configurations in an epoch can be executed in parallel. The LE context is initialized, and the controlling finite state machine is moved to an active state. The FSM for each LE will execute until it completes and returns to an inactive state. For example, a distributor may be expecting only a fixed number of items, or may move to inactive upon receipt of a particular control marker from its bitmask channel.

Configuration registers are memory mapped with a read/write configuration interface, shown in Figure 8(a) as gold arrows. The active context responds to the configuration controller using dedicated response channels. The configuration controller uses the configuration data to set up

[2] Analogous to not giving bassoon sheet music to the drum section.



(a) Example of one partition of SPMSPV from Figure 3 expressed as a pipeline of Symphony transfer objects and logical elements.



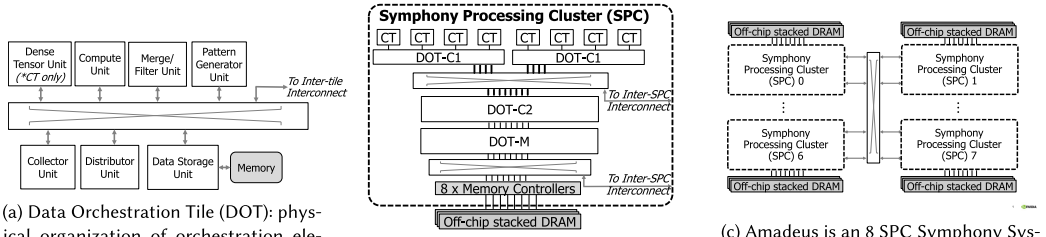
(b) Binding the SPMSPV partition to a pedagogical physical cluster. Remaining partitions could be bound to other clusters, or to the same cluster based on bandwidth requirements.

Fig. 10. A Symphony program is a pipeline of storage, compute and data orchestration logical elements, plus a binding to the physical implementations of those logical elements and configured network circuits between their input/output channels (not shown).

the channel map for the logical channels of each active LE. It then stores the LE configuration into a statically assigned LE context, setting up required dynamic state in the process.

Execution is begun by using the configuration data interface to write to a specific location in the memory map for the FSM/datapath associated with the individual context. Once the LE has reached an inactive state, it sends a completion message to the configuration controller. Each LE context can also be paused for partial dynamic reconfiguration during execution by writing a a specific location using the memory interface. To save time and energy, reconfiguration only requires writing data to the LE context that is different between the previous and current configuration.

In summary, Symphony's goal is to simultaneously configure a heterogeneous set of hardware modules without requiring each module to invent a custom configuration procedure. By arranging



(a) Data Orchestration Tile (DOT): physical organization of orchestration elements designed to be hierarchically composable.

(b) Amadeus Symphony Process Cluster (SPC).

(c) Amadeus is an 8 SPC Symphony System.

Fig. 11. Amadeus design point in the same area, frequency, energy, and memory bandwidth budget as NVIDIA Ampere A100.

these modules with Turing-complete datapaths and a particular buffer hierarchy, the same compilation techniques can be used to target a wide variety of concrete designs with varying subsets of logical elements.

5 HIERARCHICAL COMPOSITION INTO CHIPS

This section provides a specific instance and provisioning of a Symphony-based system we call Amadeus, shown in Figure 11. Amadeus's total system capabilities were chosen to match the hardware capabilities of the NVIDIA Ampere-A100 GPU [33] in terms of compute and storage resources, DRAM interfaces, and clock frequency. The Amadeus configuration is not the result of an extensive design-space exploration, but rather a straightforward application of the architecture concepts described earlier with a goal of evaluating these concepts relative to a similarly provisioned GPU architecture.

In our design, the physical units are grouped into tiles to provide specific capabilities. While **Compute Tiles (CTs)** focus on high-intensity computations, **Data Orchestration Tiles (DOTs)** provide a memory hierarchy with data orchestration support and near-memory processing. These tiles are grouped together into processing clusters called a **Symphony Processing Cluster (SPC)**. As shown in Figure 11(c), the baseline Amadeus has eight SPCs.

Symphony Processing Cluster (SPC). Figure 11(b) illustrates the details of an SPC, including CTs and DOTs. Both CTs and DOTs have a similar internal structure consisting of interconnected computation, data orchestration, and storage units. The primary difference between them is that the CTs have more compute capability, while the DOTs have more storage capacity. As an example, Figure 11(a) shows the architecture used by the CT, DOT-C1, and DOT-C2 from Figure 11(b). The data storage unit within each tile implements collections of configurable storage units, such as buffets, scratchpads, or caches. In the DOT-C2, the storage managed by the storage units is implemented in SRAM. The pattern generator unit holds a configurable collection of logical pattern generators that help orchestrate data movement between tiles. The set of compute units implements configurable collections of the logical compute elements of the architecture.

The CTs and DOTs of the Symphony physical architecture form a set of layers in the SPC. Each layer comprises a collection of storage and compute/control units with different granularities and sizes. The four layers of an SPC are:

— CT: The main Compute Tile units receive densely packed, marshaled operands (usually from a DOT-C1) and performs data-parallel computations. The compute operations are performed by either the **Dense Tensor Unit (DTU)** or more general SIMT compute engines. Each CT also includes a storage collection.

Table 4. Symphony Amadeus SPC Components and Capabilities for Each Tile

Unit	Compute	Storage	Storage BW (B/clk)	Per SPC
CT	128 ALUs 512 DTU ALUs	128 KB 4 KB	512 256	8
DOT-C1	64 ALUs	256 KB	1,536	2
DOT-C2	40 ALUs	4 MB	1,366	1
DOT-M		off-chip DRAM	342	1

— DOT-C1 and DOT-C2: Data Orchestration Tiles for Compute (Level-1 and Level-2) are responsible for marshaling operands to and from the CT. DOTs are dominated by SRAM storage managed by the DOT memory manager but also have a modest number of SIMT compute engines for basic streaming data manipulation and near-memory processing.
— DOT-M: The **Data Orchestration Tiles for Memory (DOT-Ms)** interact directly with the memory controllers for DRAM access that allow the DRAM to be used as the memory manager's backing store. They do not have programmable compute capabilities. The storage element for a DOT-M is off-chip in DRAM.

Amadeus capabilities. Figure 11 shows that Amadeus includes four Symphony SPC modules interconnected by a NoC. Table 4 summarizes the compute and storage characteristics of each tile within an SPC. The computational capabilities are greatest in the DTU of the CT and diminish across the layers towards the DOT-M attached to the DRAM. For example, the DTU module is equipped with the largest compute capability, supporting dense tensor kernels with 512 FP32 **multiply-accumulate (MAC)** units, which can also be configured as 1,024 FP16 MAC units. The compute capabilities in the next tile types (DOT-C1, DOT-C2, and DOT-M) include progressively fewer computational resources at each level. Conversely, the storage is largest at the DRAM end of the architecture and smallest in the CT. Table 4 shows the aggregate storage bandwidths for the SPC layers, which are computed by multiplying the number of tile instances by the storage bandwidths. The DTU in the CT requires only a total of 256 B/clk due to the amount of reuse it can exploit in dense tensor operations.

Table 5 shows the hardware provisioning for each layer of the Amadeus hierarchy. For the CT, the compute engine is configured to have four physical datapaths that each support 32-thread warps. For the other levels in the system with compute engines, the warps are 8-wide. Each unit has a maximum number of logical contexts, which is selected to meet the needs of our applications in Section 6. Based on the maximum number of logical contexts, we find that Amadeus has 512 KB of LE context data, not including state for the compute engines. In the worst case, loading the LE context data from DRAM takes 100 ns based on our DRAM bandwidth. As most applications will only need to initialize a subset of this state, the will take less time to initialize the full system. Furthermore, partial reconfiguration only requires modifying a subset of active LE configuration.

A configurable NoC provides virtual circuits that connect the units within and between tiles of the SPC. The DOT-C NoC provides connectivity between DOT-C1s and a DOT-C2 within and between SPCs, while the DOT-M NoC interconnects the DOT-M Manager to Memory controllers within and between SPCs. These virtual circuits are used to implement the LI channels between logical elements. The Symphony NoC supports both shortest path communication, as well as programmatic unicast and multicast, eliminating unnecessary on-chip traffic amplifications, and resulting in energy-efficient communication. The latency for data transfers between the tiles of the system are similar to those within a high-performance GPU.

Table 5. Amadeus Physical Unit Capabilities

	LE Contexts Per Tile / *(Peak LE Throughput)*			
	CT	DOT-C1	DOT-C2	DOT-M
Compute Unit	128/*(4)*	512/*(8)*	256/*(5)*	
Pattern Generator Unit	128/*(16)*	512/*(32)*	256/*(64)*	512/*(64)*
Memory Manager Unit	128/*(8)*	512/*(32)*	1,024/*(64)*	2,048/*(64)*
Distributor Unit	32/*(8)*	128/*(8)*	256/*(8)*	
Collector Unit	32/*(8)*	256/*(8)*	512/*(8)*	
Merge/Filter Unit	32/*(4)*	128/*(8)*	256/*(5)*	32/*(8)*

Table 6. Amadeus Interconnect Feeds and Speeds

Interconnect Interconnect	Count Count	Ports Ports	BW per Port	Total BW
CT ↔ DOC-C1	16	4×4	192 GB/s	12 TB/s
DOT-C1 ↔ DOT-C2	8	8×8	96 GB/s	6 TB/s
DOT-C2 ↔ DOT-M	8	8×8	32 GB/s	2 TB/s
SPC ↔ SPC	1	8×8	256 GB/s	2 TB/s

Interconnect. The interconnect between the CT and DOT-C1s, the DOT-C1 and DOT-C2, and between the SPCs are all full-bandwidth full-duplex crossbars. The details of these crossbars are shown in Table 6. This architecture effectively guarantees that the interconnect itself is not a bottleneck, while at the same time is not prohibitively expensive.

6 EVALUATION

6.1 Methodology

To evaluate the Symphony architecture, we use a combination of hardware platforms and simulation systems. Table 7 lists the specifications of the GPU platform we use as a baseline to compare to Symphony. For Symphony systems, we developed the SymSim simulator by extending the NVArchSim simulator [50] to support Symphony features. The evaluation combines SymSim with Accelergy to provide energy projections [55]. The Amadeus parameters are summarized in Table 7. The sections below describe the specific methodology for measuring execution time and energy consumption for each of these platforms.

6.1.1 GPU Methodology. We primarily compare our proposed architecture against an NVIDIA Ampere A100 GPU. Specifically, we use an NVIDIA DGX system with an AMD EPYC 7742, 512 GB RAM and Ampere A100-SXM-80GB GPUs to provide a performance and energy comparison to a state of the art GPU platform. The GPU implementations of evaluated workloads leverage various frameworks and libraries, including PyTorch, Gunrock [52], and NVIDIA libraries such as cuSPARSE [34], as well as hand-coded CUDA implementations for performance and efficiency.

We use `nvprof` to collect the performance for the key regions of the target kernel and `nvidia-smi` to collect the power statistics during the key regions. We isolate the dynamic power (and energy) by measuring the full static and dynamic power of the GPU during application execution and then subtracting the power measured when the GPU is idle.

6.1.2 Symphony Methodology. We implemented SCUDA code for each application and optimized it for execution on the Amadeus architecture. We configured the SymSim simulator to model one Amadeus SPC. While a full Amadeus system consists of 8 SPCs, we simulated a single SPC and scaled the performance linearly with SPC count. The parallelization of the workloads on Symphony make them scale linearly as compute bandwidth, memory bandwidth, and on-chip memory capacity are scaled from one to eight SPCs. In some cases, we measured performance of individual workload kernels and used an analytical model to compute the execution time and energy of the entire workload. We also ran some of the workloads for smaller numbers of iterations to limit simulation wall-clock time; we then scaled the results up to reflect the full iteration count.

For energy estimation, we assume Amadeus is implemented in a the same TSMC 7 nm process technology node as the NVIDIA A100 GPU. We created a Symphony-specific Accelergy plug-in based on a 7 nm technology. Accelergy uses the energy-relevant action counts from SymSim to generate estimates of dynamic energy consumption. We focus on dynamic energy as it depends on application behavior; we do not model standby power or static energy consumption.

Table 7. GPU and Amadeus Parameters

Reference GPU		Symphony System	
Model	A100	Instance	Amadeus
Process Technology	TSMC 7nm	Process Technology	TSMC 7nm
FP32 Throughput	20.9 TFLOPS	FP32 Throughput	24.6 TFLOPS
FP16 Tensor T.put	334 TFLOPS	FP16 Tensor T.put	197 TFLOPS
Peak Warp Count	6,912	Peak Warp Count	4,096
Max Clock Freq.	1.5 GHz	Max Clock Freq.	1.5 GHz
Register File	27.6 MB	Register File	8 MB
Cache L1	20.25 MB	DOT-C1 Capacity	4 MB
Cache L2	40 MB	DOT-C2 Capacity	32 MB
Memory Channels	40	Memory Channels	40
HBM DRAM	80 GB	HBM DRAM	80 GB
HBM Bandwidth	2048 GB/s	HBM Bandwidth	2048 GB/s

Table 8. Amadeus Area Breakdown

Unit	Total Instances	Total Area
CT (w DTU)	64	255 mm^2
DOT-C1	16	57 mm^2
DOT-C2	8	47 mm^2
DOT-M	8	19 mm^2
NOC/Interconnect	1	157 mm^2
HBM PHY	6	66 mm^2
Total Amadeus Area		599 mm^2
NVIDIA Ampere A100 Area		826 mm^2

We use a bottom up area estimation methodology to assess the area of the Amadeus system. For each physical unit, we estimate the area to support the logical element. We then account for the number of physical instances and number of contexts to estimate the size of a tile. These estimates are combined analytically to estimate the total area of the Amadeus system. Table 8 shows the area for each component of Amadeus. Some of the difference between Ampere chip area ($826mm^2$) and Symphony chip area (599 mm^2) is due to differences in provisioning of math throughput and register file/cache capacities while some is due to redundancy in Ampere to boost yield.

6.2 Micro-benchmark Characterization

We use roofline analysis [53] to understand the peak capabilities with respect to memory and computation bandwidth provisioning in Amadeus and Ampere, respectively. Traditionally, roofline analysis has been applied to dense tensor algebra using the concept of arithmetic intensity:

$$intensity = Num_{multiplications}/Transfers_{bytes}. \tag{1}$$

In a sparse context, this only includes *effective* multiplications (i.e., where both operands are non-zero). Additionally, $Transfers_{bytes}$ includes both the metadata (e.g., coordinates and positions) and data (e.g., actual non-zero values). Because non-zero values are only loaded after a match, the more overlap between non-zero operands, the more memory bandwidth the system must provide to maximize performance. For this study, we fix the rate of overlap between operands to 100%, placing the highest possible pressure on memory bandwidth.

Microbenchmark kernel. We select the following *parallel sparse tensor contraction* kernel:

$$Z_s = \sum t(A_{s,t} * B_{s,t})^i. \tag{2}$$

The s rank is distributed in space, while the t rank is processed temporally. The A tensor is stored in a {compressed, compressed} format (i.e., CSF, also called DSCR [3]), and the B tensor is stored {dense, compressed} (i.e., CSR). Thus, both compressed versus dense and compressed versus compressed contractions are included. Finally, i is an artificial static intensity parameter that increases reuse, which enables our roofline sweep.

GPU/Symphony Comparison. To characterize Ampere, we took the code output from TACO [21] and hand-ported it to CUDA by partitioning the tensors' s ranks across multiple thread blocks, warps, and threads. Figure 12 shows that A100 achieves approximately 1/4 of the ideal roofline performance for low-intensity sparse algebra. The GPU requires a total intensity of 64 effective multiplications per successful intersection to become compute-limited, much greater than the intensities observed in practice in Figure 1.

As Amadeus has computation capabilities at the DOT-C2, the DOT-C1, and the CT, we swept intensities of three different bindings of the kernel separately. Figure 12 shows that Amadeus makes notably better use of available memory bandwidth to maintain roofline performance across all intensities. When multiple Amadeus variants achieve the same performance, the binding closer

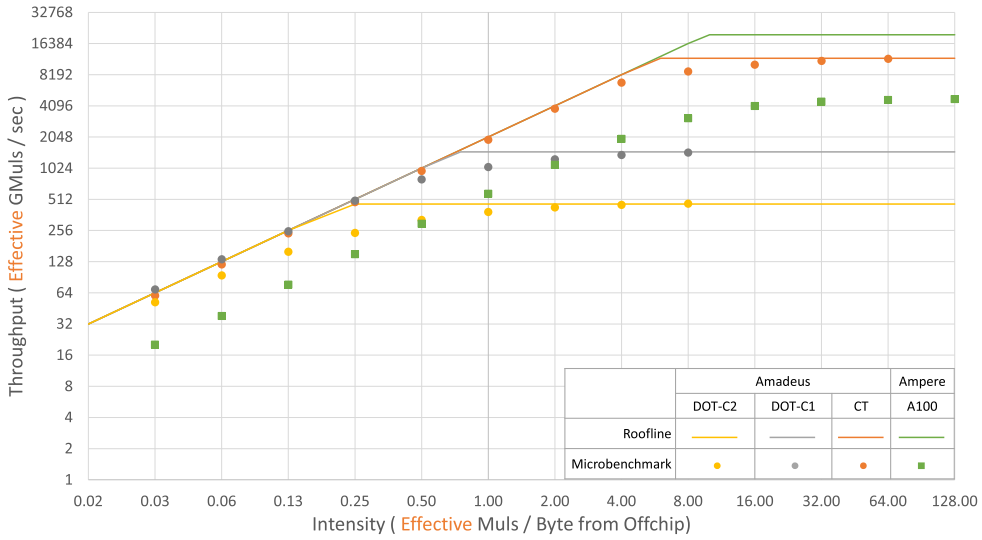


Fig. 12. Roofline characterization of Ampere and Amadeus on the micro-benchmark kernel (32-bit floating point SIMT).

Table 9. Symphony Benchmarks

Workload	Description	Dataset
GraphSAGE [14]	Deep learning on graph-structured data	1M node graph; 32-elmt feature vectors; batch-sz 256; 2143 batches; 3 epochs
IPNSW [25, 26]	Maximum inner-product search (navigable small world algorithm)	1M node graph; 400 bytes per node
PR-Nibble [45]	Local graph clustering (pagerank nibble algorithm)	5,157 vertices; 373,144 edges
SinkhornWMD [8]	Word-movers distance using Sinkhorn-Knopp algorithm	558,536 doc database; 100,000 word dict.ionary
ConvNet [15]	Convolutional neural network for image classification	Training on 10K 32×32 images

to DRAM will generally use less energy, and potentially allow SIMT units to be power-gated. This strategy of mapping of the computation to the "least-upper bound" computation datapath in Amadeus is leveraged by our workloads, as described in the following sections.

6.3 Benchmarks

We perform our evaluations on a set of workloads provided by DARPA for the Software-Defined Hardware project. The benchmarks cover a variety of domains such as graph processing, graph neural networks, natural language processing, and convolutional neural networks. Table 9 presents the set of benchmarks and their datasets. Table 10 details which key Symphony features each of the workloads utilize.

Graphsage. GraphSAGE is a graph-based deep learning algorithm whose objective is to produce *embeddings* for a given set of graph nodes by sampling random neighbors via a trained model, then propagating iteratively [14]. On Amadeus, we map the DRAM-dominated neighbor sampling operations to the DOT-C2 and DOT-C1, which fetch feature data into dense blocks mapped to buffers in the DOT-C2. All subsequent operations are formulated as a series of compute-intensive

Table 10. Symphony Features Used by Benchmarks

	μbench-mark	graph-sage	ip-nsw	PR-nibble	sink-horn	conv net
Storage idioms						
Buffets/FIFOS	✓	✓	✓	✓	✓	✓
Scratchpad			✓			
Cache				✓		
Configurable communication patterns						
Producer-consumer	✓	✓	✓	✓	✓	✓
Distributors	✓	✓	✓	✓	✓	✓
Collectors/Reducers		✓	✓			
Bypass buffers	✓	✓	✓	✓	✓	✓
Specialized reconfigurable datapaths						
Pattern Generators	✓	✓	✓	✓	✓	✓
Merge	✓					
Adapters: Vector re-packing	✓	✓			✓	
DTU		✓	✓			✓
Processing near-memory						
DOT-C2	✓	✓	✓	✓		
DOT-C1	✓	✓	✓			✓
CT	✓	✓	✓	✓	✓	✓

dense linear algebra steps. We configure the Amadeus DTUs to compute these as GEMM tiles and reductions using features, model weights, and biases read from the DOT-C2.

IPNSW. Inner Product Navigable Small World (IPNSW) accelerates finding K closest nodes to a given query node by approximating the closest node on sampled graphs [25, 26]. The application first traverses unvisited neighbors of a candidate node in a graph (mapped to DOT-C2) and calculates the distance between the neighbors and the query node (mapped to DTUs). Then, two sorted lists of 128 closest nodes are updated for future candidates (mapped to DOT-C1) and potential results (mapped to CT). The application iterates this process until no more candidates exist. IPNSW consists of 512 independent query processes that are launched simultaneously.

PRnibble. PRnibble is a algorithm whose runtime is dominated by Sparse-Matrix by Sparse-Vector multiplication. This memory bandwidth bound phase is mapped onto Symphony by performing the memory indirections in the DOT-C2 while mapping computations across the different CTs. To avoid on-chip interference, streaming data is mapped onto FIFOs (in the DOT-C2) while data with unpredictable locality is mapped to a cache hierarchy in the CT and DOT-C2 storage, which is configured to support commutative scatter updates [27].

Sinkhorn. SinkhornWMD measures the similarity between a query document to a database of documents using the Sinkhorn-Knopp iterative algorithm [8]. Runtime is dominated by **Sampled GEMM (SDDMM)** and **Sparse-Dense GEMM (SPMM)** sparse linear algebra kernels. Both memory bandwidth-bound kernels are mapped to Symphony by keeping a dense input stationary in CT buffets and streaming the other input from DRAM, using the nonzeroes of the sparse document database to direct which data values to access. The core computation executes in the CTs, while a pipeline of pattern generators at the DOT-C2 manage data orchestration.

Convnet. ConvNet is a workload that trains a *dense* **convolutional neural network (CNN)** to perform image classification on the CIFAR-10 dataset [22]. We use ConvNet to show that Symphony does not sacrifice performance on dense tensor algebra to support unstructured sparsity.

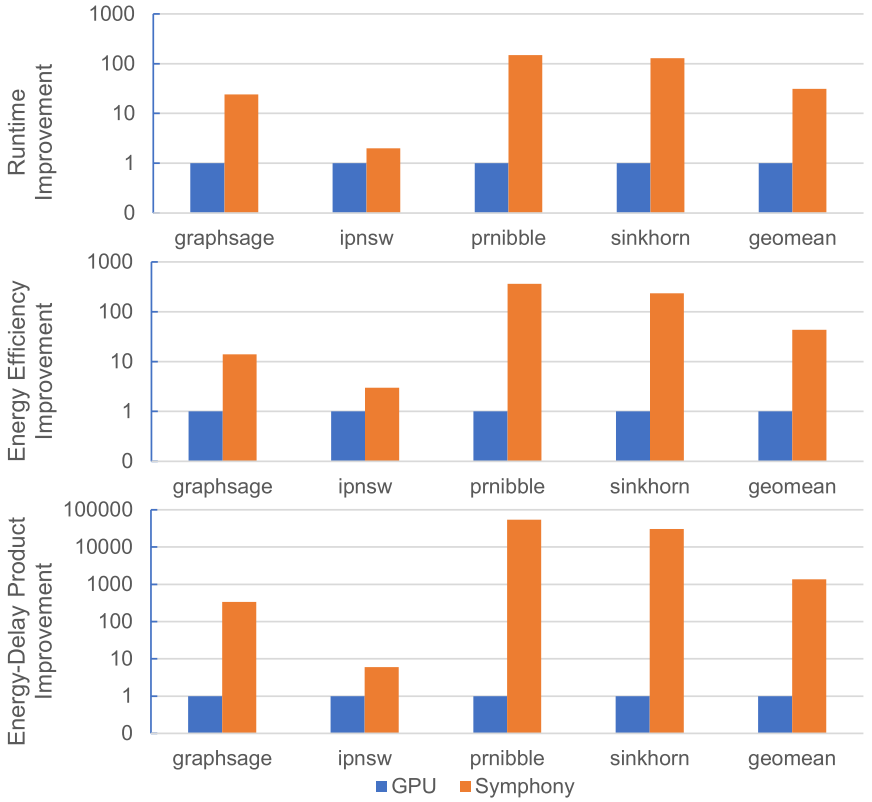


Fig. 13. Amadeus runtime and efficiency compared to Ampere.

The CNN is a simplified **residual network (ResNet)** [15] with layers including CONV, BNorm, ReLU, Add, and MaxPool. Over 98% of the computation in ConvNet happens in the CONV layers of the CNN.

6.4 Sparse Workloads

Figure 13 compares Amadeus performance, energy efficiency, and energy-delay product results to the Ampere GPU hardware platform baseline. The results show that Symphony Amadeus is 1.6–60× faster and consumes 6.3–330× less energy than a comparably provisioned GPU.

GraphSAGE. Amadeus achieves 17.5× speedup and 12.7× energy improvement versus the reference GPU. Our analysis projects that these numbers will remain largely unchanged with larger workload sizes. We attribute these improvements to Symphony's ability to expose a large amount of memory parallelism, to leverage processing near-memory (minimizing initiation latency and data-movement energy), and to pipeline heterogeneous computation using efficient built-in synchronization mechanisms. Additionally, the GPU is under-performing in this specific workload for two main reasons. First, the Ampere GPU cannot use its tensor cores in FP32 mode. Second, communication between separate kernels goes through GPU global memory, leading to excess DRAM traffic as opposed to producer-consumer data movement in Symphony via on-chip buffers/channels.

IPNSW. The application leverages several Symphony features, which allows Amadeus to achieve 1.6× speedup and 6.3× energy savings over the reference GPU. First, the buffet stores the query data and delivers it to DTU repeatedly, maximizing data reuse. Second, two temporal

lists of candidates and results are stored in DOT-C2 and DOT-C1 scratchpads, lowering the movement cost of the frequently accessed data. Third, computations (for graph traversal, visited filter, and candidate sort) are processed in local compute engines (in DOT-C2, DOT-C1, and DOT-C1, respectively) that are located near data. Finally, the compute stages can be executed in parallel with minimal synchronization over the producer-consumer interface. We expect the relative speedup and energy savings to be constant with increasing graph size, since the cost of determining a vertex's neighbors is independent of graph size.

PR-Nibble. PR-Nibble on Amadeus achieves 38× speedup and a 330× energy consumption reduction over the GPU implementation. The benefits stem from near memory processing (in the DOT-C2), which avoids the latency introduced with chained memory accesses that are common in graph applications. Further, we efficiently utilize the on-chip storage space between the different operands and ensure that the majority of on-chip storage space is dedicated to data with locality. Additionally, the commutative scatter updates in the cache hierarchy also avoids unnecessary data-movement.

SinkhornWMD. SinkhornWMD on Amadeus achieves a 60× speedup and a 230× energy consumption reduction over the GPU implementation. First, Amadeus pipelines indirect memory accesses to the sparse matrix, enabling better memory-level parallelism and utilization of the memory subsystem. Amadeus avoids energy waste by generating DRAM requests at the DOT-C2 and streaming data directly to the CT, bypassing intermediate storage. Finally, SPMM leverages multicasting and better request coalescing for a 45% DRAM traffic reduction in Amadeus. Much of this improvement is caused by a suboptimal cuSPARSE implementation of SDDMM, which performs a GEMM before sampling using the sparse matrix, instead of sampling which dot-products to perform. We estimate Symphony would be 10× more efficient than the GPU if both employed the same algorithm. We expect the relative benefit of Amadeus over Ampere to remain constant with increasing dataset size given the memory-bound nature of the application and extremely low sparsity of the sparse matrix.

6.5 Dense workload on Amadeus: ConvNet

We map CONV layers to use the DTU and further adopt an optimized dataflow that exploits all the available data reuse in the kernel, such as using the *multicast* distributors at DOT-C1. For the bandwidth-limited layers, we leverage the hierarchical computation in Amadeus and map BNorm/ReLU/Add to DOT-C1 and MaxPool to CT. We also fuse layers in ConvNet to reduce data movement with hardware-managed queues in Amadeus. For Ampere, we ran ConvNet with mixed-precision training and enabled FP16 tensor cores.

Our evaluation shows that Amadeus is 12.5× faster and 2.3× more energy efficient than GPUs with tensor cores, for two reasons. First, without explicit data orchestration and optimized dataflow, the Ampere GPU requires large channel counts (>128 input/output channels) to create sufficient compute intensity for high math unit utilization. However, the CNN model in convnet has much smaller channel counts (between 16 and 64), resulting in low throughput on GPUs for these CONV kernels. Second, fusing multiple layers in GPU kernels requires communicating via data structures in memory and barriers, while in Amadeus we leverage the hardware-managed queue to support layer fusion. While a larger ConvNet may show a smaller gap between Ampere and Amadeus, we still expect Amadeus to show an advantage due to its ability to map various optimized dataflows.

6.6 Comparison to Fixed-function Sparse Accelerators

To quantify the overhead of supporting programmability, we compare the Symphony HHP approach to ExTensor [16], a dedicated accelerator for sparse tensor algebra. To make as fair a

Table 11. Comparing ExTensor [16] (Scaled into 7 nm) and Wolfgang, a Similarly Provisioned Symphony System

Fixed-Function Sparse Accelerator		Symphony System	
Work	ExTensor [16]	Instance	Wolfgang
Process Technology	TSMC 7 nm	Process Technology	TSMC 7 nm
Max Clock Freq.	1.5 GHz	Max Clock Freq.	1.5 GHz
Scalar MACC PEs	128	SIMT-8 Compute Tiles	128
Peak MACC Throughput	192 GFLOPS	Peak FP Throughput	1.5 TFLOPS (8×)
Last-Level Buffer	30 MB	Dot-C Capacity	30 MB (1×)
DRAM Bandwidth	68.2 GB/s	DRAM Bandwidth	68.2 GB/s (1×)
Total PE area	28.2 mm^2	Total CT area	39.7 mm^2 (1.41×)
LLB area	13.1 mm^2	DOT-C area	21.1 mm^2 (1.61×)
Memory Ctrl/PHY area	20.7 mm^2	Memory Ctrl/PHY area	20.7 mm^2 (1×)
Total area	62.0 mm^2	Total area	81.5 mm^2 (1.32×)

comparison as possible, we examine the ExTensor architecture in Amadeus's TSMC 7 nm process using the same area model as above. Then, we construct a down-scaled variant of Amadeus that we name *Wolfgang* with the same memory bandwidth, number of buffer levels, and entries per buffer as ExTensor. However, Wolfgang keeps the HHP arrangement of using Symphony logical elements for Intersection, Distribution, Collection, and so on, and uses the same configurable buffer controllers, arranged as in Amadeus's DOT-C2. For compute units, Wolfgang discards ExTensor's fixed-function MACC PEs and replaces them with SIMT Compute Tiles from Amadeus, excepting the Dense Tensor Unit. This results in the 32% area overhead shown in Table 11, which demonstrates that the bulk of the cost comes from the Compute Tile, with the benefits of introducing programmability and increasing peak throughput by 8×.

As ExTensor is not general enough to execute arbitrary workloads (such as the ones evaluated above), we use Wolfgang to run **sparse-matrix sparse-matrix multiplication (SPMSPM)**, a key workload of ExTensor. Via private communication with the authors, we obtained the ExTensor simulator used in Reference [35], which includes action counts for performing SPMSPM across various data sets. We then use the energy costs-per-action as in Section 6.1, to fairly compare Wolfgang and ExTensor, both in 7 nm. To obtain the energy consumption of Wolfgang, we adjusted Amadeus's energy model, following the programmability overheads quantified in Reference [17] where necessary.

The results of the comparison are shown in Figure 14. As Wolfgang is general enough to run the same dataflow and work schedule as ExTensor it obtains the same performance. The latency differences in our generalized on-chip network are not significant enough to detract from the steady-state performance over the course of the workload. This result is intuitively confirmed by the roofline analysis of ExTensor in Reference [16] and our micro-benchmark analysis in Section 6.2, which both independently demonstrate that these platforms are able to reach peak memory-bandwidth utilization on this class of applications. Even though Wolfgang includes SIMT datapaths, we only use them in scalar mode to facilitate a direct comparison to ExTensor, which lacks SIMT units. As a result, Wolfgang pays for but does not use the Vector Pack Adaptor from Section 3.1. Thus the performance result is conservative, as Wolfgang has a significantly higher dense computation roofline, but this difference does not manifest on a memory-bound workload like SPMSPM.

The energy cost for both accelerators is dominated by offchip memory. As 70%–97% of the overall energy stack comes from DRAM, overall EDP only degrades by 4% on average. While the

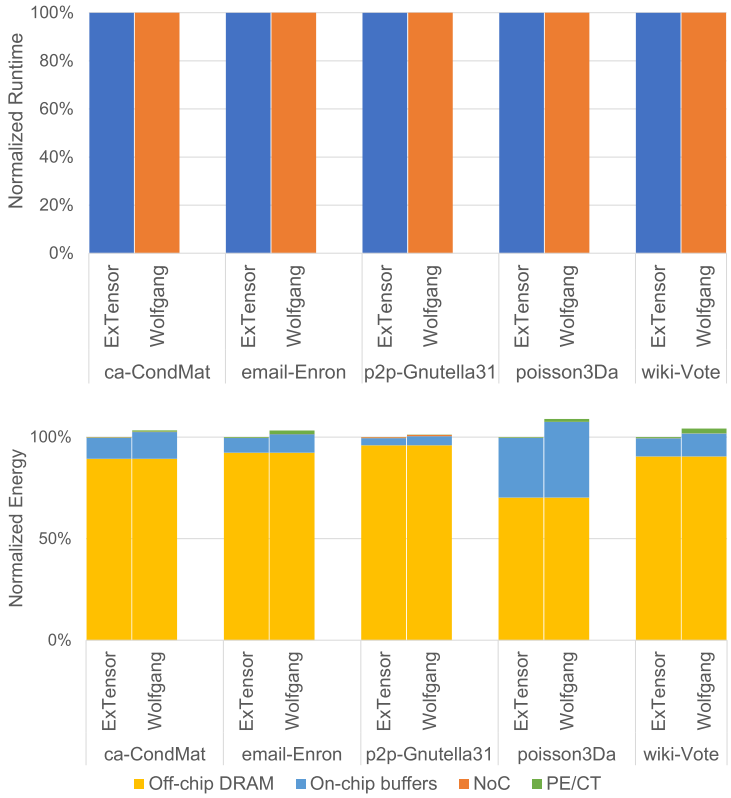


Fig. 14. Quantification of overheads compared to fixed-function accelerator running sparse-matrix sparse-matrix multiplication (SPMSPM). GeoMean reduction in EDP is 4% to support Symphony-style programmability.

architectures differ in their distribution of energy consumption across the programmable PE, flexible NoC, and on-chip buffers, these components are limited to less than 14% of the total energy usage–with the exception of the high-locality `poisson3Da` data-set, where on-chip buffering becomes more dominant (29% of energy for ExTensor versus 37% for Wolfgang).

Overall, this comparison demonstrates that accelerators for memory-bound applications can profitably consider adding programmability features without significantly affecting EDP. The main benefit is increasing peak throughput on a wide range of applications, for modest area investment. Symphony represents an architectural attempt at blending the styles of programmable GPUs, compute-bound dense tensor accelerators, and memory-bound sparse tensor accelerators in a complementary way. The Heterogeneous Hierarchical Processing approach shows a promising path forward for combining the best features of all three worlds.

7 RELATED WORK

Recent research has developed a wide range of different acceleration approaches to sparse tensor algorithms. Outerspace [38] is an SPMSPM accelerator based around an outer-product dataflow that uses custom reduction logic for partial sums. SpArch [58] demonstrates that this approach can be improved by custom update batching and reduction scheduling for higher spatial locality. Matraptor [47] uses Gustavsson's algorithm as an alternative dataflow that reduces read-modify-writes of partial sums in exchange for more operand reads, which are less expensive. Sigma [42]

uses a flexible dataflow approach that improves adaptability via customized network-on-chip routing and buffers. Tensaurus [48] demonstrates that co-designing the storage format and the custom hardware can lead to further improvements. These and other narrow-domain architectures focus on specific characteristics or dataflows of sparse tensor algorithms, while Symphony deploys finer-grained building blocks and aims to serve a wider range of range of algorithms.

In addition to tensor accelerators, the domain of graph algorithms has been popular for accelerator research. Most proposed graph accelerators use forms of decoupled access-execute and have focused on a subset of the algorithm space. Examples include Graphicionado [13], Ozdal's energy-efficient graph analytics accelerator architecture [37], Chronos [1], and GraphPulse [43]. Dadu's PolyGraph paper makes a strong case for flexibility for graph analytics accelerators [9], which has similar objectives to our Symphony architecture.

Coarse-grained reconfigurable architectures (CGRAs) aim to provide performance and efficiency approaching that of ASICs, coupled with the flexibility of FPGAs [24]. CGRA's tend to provide large building blocks such as the **pattern compute unit (PCU)** and pattern memory unit of Plasticine [41] or the processing elements in the Wave Computing architecture [31]. In contrast, Symphony employs medium-grained components that are more specialized (for efficiency) and can be composed to create more complex data pipelines.

Possibly the most closely related work is the concurrent Sparse Abstract Machine project [59] which extends a dense CGRA architecture [60] with a set of LE-like hardware units that fetch and store tensor fibers, and perform intersection, reduction, and computation. The SAM compiler translates tensor algebra expressions in TACO format [21] into configurations, which could be added as automation on top of SCUDA. SAM is not Turing Complete and only supports TACO tensor algebra, making it complementary to SCUDA's capabilities and Symphony's notions of a hybrid fully-programmable and specialized architecture using Heterogeneous Hierarchical Processing.

Separate from full domain specific accelerators, prior research has introduced a range of building blocks to accelerate sparse data processing that include: Extensor for fast metadata intersection [16], PHI for efficient data reductions [27], SpZip for streaming data compression and decompression [56], and SPU for efficient processing of sparse streams [10]. Near-data processing has also been explored in wide array of contexts [28]. For instance, prior work has developed near-data processing architectures to process chains of dependent memory references in graph workloads [2], execute small snippets of code in smaller processing elements in/near the DRAM stack [20], and incorporate vector math units near various caches in a CPU [32]. In addition, prior work has proposed Buffets as a composable storage idiom [40] and Pipette, which facilitates data pipelining through time-division multiplexing of processing elements [29]. Symphony leverages these prior works and incorporates them into a broader application and architecture domain, introducing them as reusable logical elements.

8 CONCLUSION

With the demise of Moore's Law and Dennard scaling, architecture innovations must play a greater role in providing computer system performance and energy efficiency improvements. For applications areas such as sparse tensor algebra and graph processing, efficiency opportunities arise in accelerating and streamlining compute and the movement of data. Our proposed Symphony architecture provides medium-grained specialized hardware datapaths aimed at data orchestration for both sparse and dense data structures. These datapaths are convenient building blocks that when combined with programmable processing elements can form pipelines that are efficient like dedicated accelerators but are flexible like programmable processors. Our results show that the Symphony mechanisms facilitate mean improvements of 31× and 44× in runtime and energy over

a comparably provisioned GPU. Such a platform opens opportunities for further innovation in extending the range of dynamic streaming operations, managing load balance and locality, and even more efficient sparse data representations.

The key insight behind Symphony is that specialization is not just a technique that is impactful on datapaths and processing elements, but equally applicable in the context of data orchestration. In the future, we hope to expand Symphony's toolbox of specialized modules and expand the number of applicable domains that can be improved via explicit data orchestration. Furthermore, we believe Symphony's approach lends itself to productive system construction, and so we hope to make it easier for others to create Amadeus-like design points out of the existing building blocks. All in all, we hope that Symphony demonstrates an interaction model for general-purpose and specialized hardware without memory-based polling that ultimately opens up new opportunities for architectural innovation in all aspects of the chip.

REFERENCES

[1] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient speculative parallelism for accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS'20)*.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*.

[3] Aydin Buluç and John R. Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS'08)*.

[4] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*.

[5] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Trans. Comput.-Aided Design Integr. Circ. Syst.* 20, 9 (2001), 1059–1076.

[6] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-state Circ.* 52, 1 (2016), 127–138.

[7] Stephen Chou. 2018. *Unified Sparse Formats for Tensor Algebra Compilers*. Master's thesis. Massachusetts Institute of Technology.

[8] Marco Cuturi. 2013. Sinkhorn distances: Lightspeed computation of optimal transport. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

[9] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the value of flexibility for graph processing accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA'21)*.

[10] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards general purpose acceleration by exploiting common data-dependence forms. In *the Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. 924–939.

[11] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1–25.

[12] Kermin Elliott Fleming, Michael Adler, Michael Pellauer, Angshuman Parashar, Arvind Mithal, and Joel Emer. 2012. Leveraging latency-insensitivity to ease multiple FPGA design. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'12)*.

[13] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the International Symposium on Microarchitecture (MICRO'16)*.

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS'17)*.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR'16)*.

[16] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An accelerator for sparse tensor algebra. In *Proceedings of the International Symposium on Microarchitecture (MICRO'19)*.

[17] Mark Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'14)*. 10–14.
[18] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.
[19] Brucek Khailany, Evgeni Krimer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam Likun Xi, Yanqing Zhang, and Brian Zimmer. 2018. A modular digital VLSI flow for high-productivity SoC design. In *Proceedings of the Design Automation Conference (DAC'18)*.
[20] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward standardized near-data processing with unrestricted data placement for GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'17)*.
[21] Fredrik Kjølstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. TACO: A tool to generate tensor algebra kernels. In *Proceedings of the International Conference on Automated Software Engineering (ASE'17)*.
[22] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Master's thesis. Department of Computer Science, University of Toronto.
[23] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 754–768.
[24] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *Comput. Surveys* 52, 6 (Oct. 2019).
[25] Yury A. Malkov and D. A. Yashunin. 2016. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. Retrieved from http://arxiv.org/abs/1603.09320
[26] Stanislav Morozov and Artem Babenko. 2018. Non-metric similarity graphs for maximum inner product search. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS'18)*.
[27] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *Proceedings of the International Symposium on Microarchitecture (MICRO'19)*.
[28] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Enabling practical processing in and near memory for data-intensive computing. In *Proceedings of the Design Automation Conference (DAC'19)*.
[29] Quan M. Nguyen and Daniel Sanchez. 2020. Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism. In *Proceedings of the International Symposium on Microarchitecture (MICRO'20)*.
[30] Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*.
[31] Chris Nicol. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing. Retrieved from https://wavecomp.ai/wp-content/uploads/2018/12/WP_CGRA.pdf
[32] Anant V. Nori, Rahul Bera, Shankar Balachandran, Joydeep Rakshit, Om J. Omer, Avishaii Abuhatzera, Belliappa Kuttanna, and Sreenivas Subramoney. 2021. REDUCT: Keep it close, keep it cool!: Efficient scaling of DNN inference on multi-core CPUs with near-cache compute. In *Proceedings of the International Symposium on Computer Architecture (ISCA'21)*.
[33] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. Retrieved from https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf
[34] NVIDIA. 2021. cuSPARSE. Retrieved from https://docs.nvidia.com/cuda/cusparse/index.html
[35] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. 2023. Accelerating sparse data orchestration via dynamic reflexive tiling. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS'23)*. ACM, New York, NY, 18–32.
[36] Peizhao Ou and Brian Demsky. 2017. Checking concurrent data structures under the C/C++11 memory model. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*.
[37] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*.
[38] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2019. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'19)*.

[39] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangarajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to DNN accelerator evaluation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*.

[40] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Karthik Hedge, Rangharajan Ventakesan, Stephen Keckler, Christopher W. Fletcher, and Joel Emer. 2018. Buffets: An efficient, flexible, composable storage idiom for accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS'18)*.

[41] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan H adjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *Proceedings of the International Symposium on Computer Architecture (ISCA'17)*.

[42] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'20)*.

[43] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. GraphPulse: An event-driven hardware accelerator for asynchronous graph processing. In *Proceedings of the International Symposium on Microarchitecture (MICRO'20)*.

[44] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel S. Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO'19)*.

[45] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W. Mahoney. 2016. Parallel local graph clustering. *Proc. VLDB Endow.* 9, 12 (August 2016).

[46] James E. Smith. 1982. Decoupled access/execute computer architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'82)*.

[47] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2002. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *Proceedings of the International Symposium on Microarchitecture (MICRO'02)*.

[48] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'20)*.

[49] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2020. Efficient processing of deep neural networks. *Synth. Lect. Comput. Arch.* 15, 2 (2020), 1–341.

[50] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. 2021. Need for speed: Experiences building a trustworthy system-level GPU simulator. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'21)*.

[51] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO'19)*.

[52] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*.

[53] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (Apr. 2009).

[54] Y. N. Wu, P. Tsai, A. Parashar, V. Sze, and J. S. Emer. 2021. Sparseloop: An analytical, energy-focused design space exploration methodology for sparse tensor accelerators. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'21)*.

[55] Yannan (Nellie) Wu, Joel Emer, and Vivienne Sze. 2019. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *Proceedings of the International Conference On Computer Aided Design (ICCAD'19)*.

[56] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2021. SpZip: Architectural support for effective data compression in irregular applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA'21)*.

[57] G. Zhang, N. Attaluri, J. Emer, and D. Sanchez. 2021. GAMMA: Exploiting gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS'21)*.

[58] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. 2020. SpArch: Efficient architecture for sparse matrix multiplication. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'20)*.

[59] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The sparse abstract machine. In *Proceedings of the 28th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) (ASPLOS'23). Association for Computing Machinery, New York, NY, USA, 710–726.

[60] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra. In *2022 IEEE Symposium on VLSI Technology and Circuits*. 70–71.

Received 21 December 2022; revised 11 September 2023; accepted 18 October 2023