A Tensor Processing Framework for CPU-Manycore Heterogeneous Systems

Lin Cheng*, Peitian Pan*, Zhongyuan Zhao*, Krithik Ranjan*, Jack Weber*, Bandhav Veluri†, Seyed Borna Ehsani†, Max Ruttenberg†, Dai Cheol Jung‡, Preslav Ivanov*, Dustin Richmond†, Michael B. Taylor†, Zhiru Zhang*, Christopher Batten*
*School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
†Paul Allen School of Computer Science and Engineering, University of Washington, Seattle, WA
‡Department of Electrical and Computer Engineering, University of Washington, Seattle, WA

***Abstract*—Future CPU-manycore heterogeneous systems can provide high peak throughput by integrating thousands of simple, independent, energy-efficient cores in a single die. However, there are two key challenges to translating this high peak throughput into improved end-to-end workload performance: (1) manycore co-processors rely on simple hardware putting significant demands on the software programmer; and (2) manycore co-processors use in-order cores that struggle to tolerate long memory latencies. To address the manycore programmability challenge, this paper presents a dense and sparse tensor processing framework based on PyTorch that enables domain experts to easily accelerate off-the-shelf workloads on CPU-manycore heterogeneous systems. To address the manycore memory latency challenge, we use our extended PyTorch framework to explore the potential for decoupled access/execute (DAE) software and hardware mechanisms. More specifically, we propose two software-only techniques, naïve-software DAE and systolic-software DAE, along with a lightweight hardware access accelerator to further improve area-normalized throughput. We evaluate our techniques using a combination of PyTorch operator microbenchmarking and real-world PyTorch workloads running on a detailed register-transfer-level model of a 128-core manycore architecture. Our evaluation on three real-world dense and sparse tensor workloads suggest these workloads can achieve approximately 2–6× performance improvement when scaled to a future 2,000-core CPU-manycore heterogeneous system compared to an 18-core out-of-order CPU baseline, while potentially achieving higher area-normalized throughput and improved energy-efficiency compared to general-purpose graphics processing units.**

I. Introduction

Manycore architectures integrate a large number of simple cores within a single die using a tiled physical design methodology, and these cores are usually interconnected through a packet-based on-chip network. Compared to general-purpose multicores, the manycore approach can improve energy efficiency and throughput per unit area on highly parallel workloads. Compared to application-specific accelerators, the manycore approach can be tailored to accelerate a wider range of applications. Early manycore research prototypes included 16–110 cores [1], [2], [3], [4], [5], [6] and manycore processors in industry now include 64–128 cores [7], [8], [9], [10], [11], [12]. Recent research prototypes have scaled core counts by an order-of-magnitude including the 496-core Celerity [13], 1000-core KiloCore [14], 1024-core Epiphany-V [15], and 4096-core Manticore [16]. General-purpose graphics processing units (GPGPUs) also seek to integrate a massive number of execution pipelines on a single die [17], [18], but GPGPUs take a fundamentally different microarchitectural approach from manycore architectures. GPGPUs group 16–32 execution pipelines and shared local memory into tens of SIMT/SIMD processors to amortize overheads with lock-step execution, while manycore architectures turn each execution pipeline into its own simple core with its own small local memory to enable completely independent execution. Like GPGPUs, manycore architectures are unlikely to completely replace traditional multicore CPUs as standalone computing platforms. Manycore architectures will likely remain as co-processors in CPU-manycore heterogeneous systems. We identify two key challenges to translating high peak throughput into improved end-to-end workload performance on such systems.

Manycore Programmability Challenge – The flexibility offered by manycore co-processors means programmers must navigate a broad software design and optimization space. This is compounded by the fact that manycore co-processors rely on simple hardware that requires programmers to manage many concerns explicitly in software. For example, some manycore co-processors leverage scratchpad memories to create a partitioned global address space (PGAS) instead of using hardware-based cache coherence, and this requires programmers to control data movement explicitly in software. In addition, programmers must carefully consider work distribution, load balancing, and on-chip network congestion. Compared to other architectures that have been studied extensively, the software stack of CPU-manycore heterogeneous systems remains less explored.

A promising approach to addressing the manycore programmability challenge is through high-level libraries that provide ready-to-use hand-optimized operators embedded within a high-level language. GPGPUs now provide many such libraries including CuPy [19], PyTorch [20], TensorFlow [21], and cuGraph [22]. In this work, we demonstrate the potential for a high-level library approach to address the manycore programmability challenge by extending the PyTorch framework for both dense and sparse tensor processing on a representative CPU-manycore heterogeneous system with a RISC-V manycore co-processor. Our extended PyTorch framework currently provides over 100 operators that leverage both a traditional optimized data-parallel approach (as in GPGPUs), and novel programming models and optimizations enabled by the unique features of manycore co-processors. For example, we propose a new *cyclic bank sparse row* sparse matrix format and padding technique that optimizes the data layout for manycore co-processors with global caches and memory controllers at the edge.

Manycore Memory Latency Challenge – Memory latency hiding is now at the center of modern microarchitecture design as the performance gap between compute and memory continues to increase. Multicore CPUs rely on complex out-of-order execution to hide memory latency, while GPGPUs rely on extreme temporal multithreading with fine-grain context switching to also hide memory latency. Both of these techniques require extensive hardware resources and are not applicable to the simple cores used in manycore arechitectures. *Stall-on-use*, which allows independent instructions to be issued while a long-latency memory instruction is still pending [23], [24], is a lightweight mechanism to enable memory latency hiding in simple in-order cores. However, our results show this technique alone cannot fully resolve the memory latency issue, and it still dominates the execution time of manycore co-processors for many critical PyTorch operators (e.g., matrix multiplication, 2D convolution, sparse matrix-vector multiplication, and matrix-vector multiplication). Moreover, as manycore architectures generally adopt a mesh-like on-chip network topology, both network bisection bandwidth and the bandwidth to higher levels of the memory hierarchy become scarcer when scaled to future manycore architectures with thousands of cores, leading to increased network congestion and memory access latencies.

Decoupled access/execute (DAE) architectures have been proposed in the literature to aid memory latency hiding by splitting one program into two instruction streams, an access stream and an execute stream [25]. The *access stream* contains all instructions related to accessing memory, and the *execute stream* contains the remaining instructions for computation. If the access stream can run sufficiently far ahead, the execute stream will no longer stall due to load-use dependencies. In this work, we use our extended PyTorch framework to explore DAE in the context of the target manycore co-processor. In Section IV, we propose two software-only techniques, naïve-software DAE and systolic-software DAE: *naïve-software DAE* pairs an *access core* with an *execute core* interconnected through software queues allocated in each core's scratchpad memory, while *systolic-software DAE* exploits data reuse to share one access core across multiple execute cores. In Section V, we propose combining lightweight *access accelerators* with our software techniques to further improve area normalized throughput. Our evaluation on several important PyTorch operators shows software/hardware co-design to enable DAE programming can achieve up to $1.32\times$ throughput improvement compared to an aggressive data-parallel baseline.

In Section VI, we evaluate three real-world workloads using the extended PyTorch tensor processing framework including: a dense residual neural network for computer vision, a dense deep-learning autoencoder-based recommender system for movie recommendations, and a sparse local graph clustering system based on an iterative shrinkage-thresholding algorithm for personalized page ranking. We execute the PyTorch CPU software natively and co-simulate the PyTorch manycore software on a detailed register-transfer-level model of a 128-core manycore co-processor with 32-bit RISC-V cores and a high-bandwidth main-memory system. Our results suggest these workloads can achieve approximately 2–6$\times$ performance improvement when scaled to a future 2,000-core CPU-manycore heterogeneous system compared to an 18-core out-of-order CPU baseline. At the same time, we argue that the manycore approach can enable higher area-normalized throughput and improved energy-efficiency compared to GPGPUs.

The primary contributions of this work are: (1) we extend PyTorch to enable optimized dense and sparse tensor processing on CPU-manycore heterogeneous systems with minimal modifications to existing workloads (Section III); (2) we propose two software-only techniques, naïve-software DAE and systolic-software DAE, to enable access/execute decoupling in the context of a manycore co-processor (Section IV); (3) we propose to combine lightweight hardware access accelerators with both software schemes to further improve area-normalized throughput on the target CPU-manycore heterogeneous system (Section V); (4) we conduct an end-to-end evaluation on three real-world tensor workloads to demonstrate the promise of the proposed framework (Section VI). While we conduct our studies on a specific manycore architecture, our techniques can be broadly applied to any manycore architecture that allows direct core-to-core communication.

II. Target CPU-Manycore Heterogeneous System

Although the manycore software and hardware design space is broad, there are several common features including relatively simple cores, mesh-based on-chip networks, software-managed memory systems, and low-level software APIs. In this section, we describe an early version of the HammerBlade (HB) architecture [26] that captures these common features.

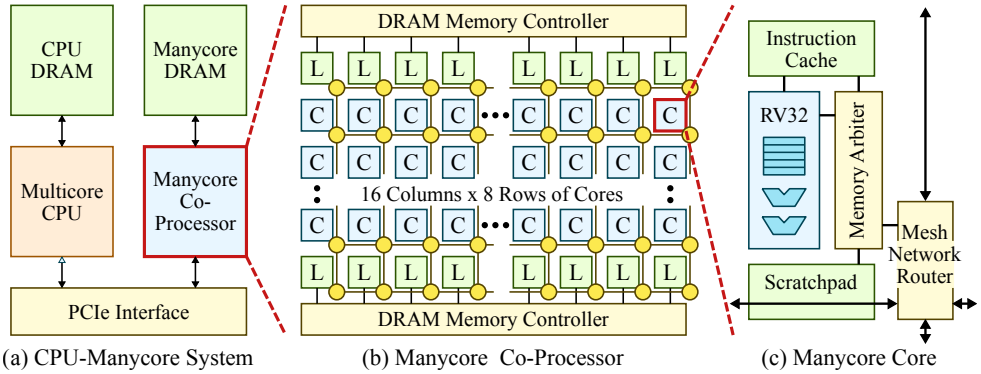
A. Target System Hardware

The HB manycore architecture includes hundreds of independent cores with simple scalar pipelines, low-latency software-managed scratchpad memories, and support for integer, floating-point, and atomic memory instructions. Cores communicate over the memory-mapped 2-D mesh on-chip-network (OCN), and adopt stall-on-use for exploiting pipeline parallelism and memory latency hiding. In addition to the scalar cores, there is a stand-alone host CPU that manages execution. Fig. 1 presents an architectural diagram of a small-scale HB CPU-manycore heterogeneous system.

The HB manycore memory hierarchy has four levels: DRAM; a banked, last-level cache (LLC); inter-core scratchpad(s); and a core-local scratchpad. The core-local scratchpad, remote scratchpads, caches, and other network locations are mapped to non-intersecting regions of a core's address space. Consequently, the HB manycore architecture exposes a PGAS-like memory model with software control over data placement.

B. Target System Software

The HB manycore architecture provides a kernel-centric programming abstraction, similar to CUDA. Kernel code is written from the perspective of a single thread executing on a core. Kernel execution and scheduling is managed through runtime software on the host processor. This provides a SPMD-like execution model. Unlike CUDA, the target system software supports remote store programming [27], which allows a core to perform remote stores into any other core's scratchpad.



(a) CPU-Manycore System (b) Manycore Co-Processor (c) Manycore Core

Fig. 1: Target CPU-Manycore Heterogeneous System Hardware – (a) target system includes a CPU with its own attached DRAM and a manycore co-processor also with its own attached DRAM; (b) manycore co-processor includes 16×8 simple cores (C) and 32 last-level cache (L) banks interconnected via mesh-based on-chip network; (c) each core is a RISC-V RV32IMAF processor (RV32) with instruction cache and 4KB scratchpad memory.

C. Manycore Challenges

We identify two key challenges to realizing the promised peak throughput of CPU-manycore heterogeneous systems.

Manycore Programmability Challenge – Similar to other manycore architectures, the target manycore architecture exposes low-level hardware details to the software stack. This requires programmers to manage many concerns explicitly. In addition, programmers must carefully consider work distribution, load balancing, network congestion, and even instruction cache pressure. Facing vast options and a broad software design space, programmers can struggle to quickly develop optimal implementations.

Manycore Memory Latency Challenge – Memory latency hiding is critical to modern microarchitectures as the performance gap between compute and memory continues to increase. This memory wall has a more significant impact on manycore architectures for two reasons: (1) with a strong emphasis on area efficiency, the cores in a manycore architecture cannot leverage traditional complex hardware mechanisms for memory latency hiding (e.g., out-of-order execution, fine-grain multithreading), and have to rely on lightweight approaches such as stall-on-use; and (2) manycore architectures almost always adopt a mesh-like topology for their OCNs. As we scale to large-scale manycore architectures with thousands of cores, both mesh bisection bandwidth and mesh perimeter bandwidth to higher levels of the memory hierarchy scale slower (i.e., linearly) than the number of cores (i.e., quadratically). Scarce bandwidth can easily lead to severe congestion increasing overall memory access latencies.

III. A Tensor Processing Framework for CPU-Manycore Heterogeneous Systems

PyTorch [20] is a widely adopted open-source tensor processing framework that provides an easy to use Python frontend for highly optimized tensor operators implemented in a low-level C++ ATen library [28]. In this section, we first present our tensor processing framework for CPU-manycore heterogeneous systems developed from PyTorch. We then evaluate and analyze a set of representative operators with micro-benchmarks on the target system to identify performance bottlenecks.

A. PyTorch on CPU-Manycore Heterogeneous Systems

We extend PyTorch and build an open-source tensor processing framework for CPU-manycore heterogeneous systems to address the manycore programmability challenge. PyTorch's Python-level operators are platform agnostic; a dynamic dispatcher in ATen chooses the appropriate implementation for execution at runtime. The actual ATen operators can be either platform agnostic or platform specific. Platform specific implementations are grouped into *backends* (e.g., a CPU backend or a GPGPU backend). Platform agnostic operators are part of the CPU backend as well. New platforms can be easily supported by plugging new backends into ATen's dynamic dispatcher. We extend PyTorch with a new ATen backend to support both dense and sparse tensor processing on the target manycore co-processor. With our framework, tensor workloads can run exclusively on the CPU of the target heterogeneous system without any changes to the code. In this scenario, the CPU backend supports the framework's Python APIs and data is stored in CPU host memory (see Fig. 2(a)). One can also choose to accelerate tensor workloads on the manycore co-processor with minimal changes to the existing code (see Fig. 3(a)). Only changing three lines is necessary: one for migrating the neural network model to the manycore co-processor and two for migrating the input data and expected labels. PyTorch operators that are platform specific will be dispatched to the manycore backend, and data will be automatically migrated as needed (see Fig. 2(d)).

An example workload using the proposed framework is shown in Fig. 3. When PyTorch operator `nn.ReLu()` is used in Python code, its ATen counterpart `relu()` is called. In this case, `relu()` is platform agnostic (i.e., runs on the CPU), and is implemented by reusing a platform-specific ATen operator (i.e., `threshold()`). Since `model` in line 26 of Fig. 3(a) is on the manycore co-processor, the call to `threshold()` in line 4 of Fig. 3(b) is dispatched to the manycore implementation (Fig. 3(c)), and compute is then offloaded to the manycore co-processor (Fig. 3(d)).

We have ported over 100 tensor operators including matrix multiplication, 2D convolution, most element-wise operators (e.g., add, subtract), reductions (e.g., sum, mean), and sparse operators (e.g., sparse matrix-vector multiplication). All operators are hand tuned and aggressively optimized: scratchpad memory is utilized to enable data reuse and increase arithmetic intensity; stall-on-use is leveraged to exploit pipeline parallelism and hide memory latency; unrolling is used to balance instruction cache performance and loop overhead.

For sparse operators, prior work has shown that the layout of sparse tensors can significantly impact performance [29], [30], [31]. In our framework, we implement a novel *cyclic bank sparse row* (CBSR) tensor layout. CBSR is designed to reduce LLC bank conflicts and network congestion by ensuring cores

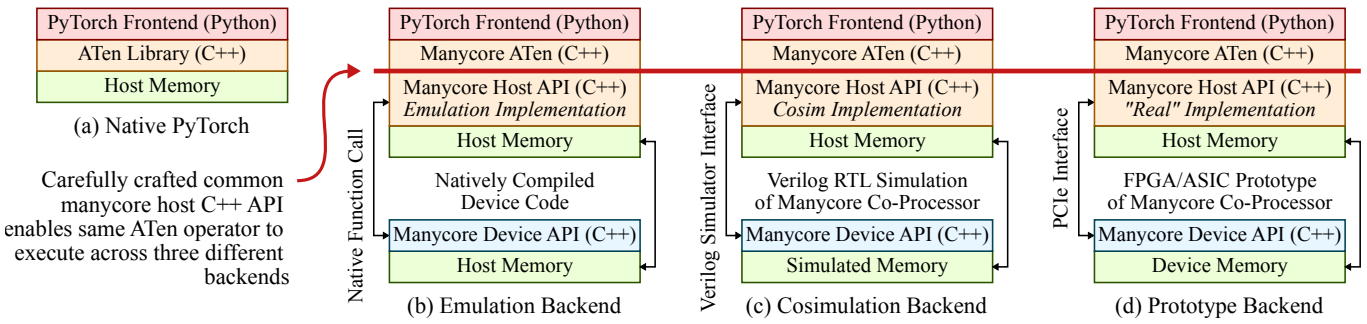


Fig. 2: Different Backends for Extended PyTorch Framework – (a) native execution on CPU without new backend; (b) *emulation backend:* host code executes natively on CPU, device code also executes natively on CPU for functional testing; (c) *cosimulation backend:* host code executes natively on CPU, device code executes on Verilog RTL simulator for cycle-accurate performance evaluation; (d) *prototype backend:* host code executes natively on CPU, device code executes on a real FPGA/ASIC prototype.

```python
class Autoencoder(nn.Module):
 def __init__(self):
  ...
  self.encoder = nn.Sequential(
    nn.ReLu(),
    nn.BatchNorm1d(800),
    nn.Dropout(0.5)
  )

  self.bneck = nn.Linear(800, 400)

  self.decoder = nn.Sequential(
    nn.ReLu(),
    nn.BatchNorm1d(400),
    nn.Dropout(0.5)
  )
  ...

 def forward(self, x):
  x = self.emb(x).sum(dim=1)
  x = self.encoder(x)
  x = self.bneck(x)
  x = self.decoder(x)
  x = self.output(x)

model = Autoencoder().manycore()
...
for x, y in dataloader_train:
 x = x.manycore()
 y = y.manycore()

 out = model(x)
 loss = F.MSELoss(out, y)

 opt.zero_grad()
 loss.backward()
 opt.step()
```

(a) Python Frontend

```
Tensor relu(const Tensor self) {
  Tensor res = opt_result.value_or(Tensor());
  auto iter = TensorIterator::binary_op(res, self, self);
  return at::threshold(iter,0,0);
}
```

(b) Platform Agnostic ATen Operator

```
void threshold_kernel_mc(TensorIterator& iter, Scalar t, Scalar v) {
  AT_DISPATCH_FLOAT_TYPE_ONLY(iter.dtype(), "threshold_mc",
    [&]() {
      offload_op_binary(iter, t.to<scalar_t>(),
                        v.to<sclar_t>(),
                        "tensorlib_threshold");
    });
}
```

(c) Manycore Backend CPU Host Function

```
int tensorlib_threshold(mc_tensor_t* res_p, mc_tensor_t* self_p,
                        float* threshold_p, float* value_p) {
  MCTensor<float> res(res_p);
  MCTensor<float> self(self_p);
  float threshold = *threshold_p;
  float value     = *value_p;

  mc_tiled_foreach(res, self, [&] (float self_v) {
    return (self_v <= threshold) ? value : self_v;
  });

  mc_barrier ();
  return 0;
}
```

(d) Manycore Backend Device Function

Fig. 3: Extended PyTorch Framework for CPU-Manycore Heterogeneous Systems – Blue lines 26, 29–30 in (a) are the only changes required to port an existing workload (e.g., training a deep neural network) written with PyTorch to run on the target CPU-manycore heterogeneous system. Red lines show the (simplified) dispatch chain for the PyTorch `ReLu` operator: Python frontend (a) dispatches to platform agnostic ATen operator (b), which dispatches to manycore backend CPU host function (c), which finally launches the manycore device function (d).

only access LLC banks located in the same column. Fig. 4 shows an example using traditional compressed sparse row (CSR), CBSR and CBSR+Padding formats for a 4×4 sparse matrix. In this simplified example, our architecture has one DRAM channel with four LLC banks. Each core only accesses one row of the sparse matrix. The data block size within each bank is two data elements and follows the cyclic memory partitioning scheme of [32]. In CSR, the indices of non-zero values of different rows may fall into the same bank, which leads to memory bank conflicts when different cores access either column indices or values (i.e., C0 accesses v2 and C1 accesses v3). Using CBSR can eliminate the memory bank conflict between cores when accessing either indices or values, but memory conflicts still remain when one core is accessing the indices and the other core is accessing the values (i.e., C0 is accessing v0 and C1 is accessing column indices of v3). CBSR+Padding makes indices and values aligned to the same LLC bank, and memory bank conflicts can be completely eliminated.

Our tensor processing framework and the emulation infrastructure are open-source[1]. We use state-of-the-art test-driven design based on `pytest`[2], Hypothesis [33][3], and continuous integration[4]. Operator development proceeds through three

[1]https://github.com/cornell-brg/hb-pytorch
[2]https://pytest.org
[3]https://github.com/HypothesisWorks/hypothesis
[4]https://travis-ci.com/github/cornell-brg/hb-pytorch

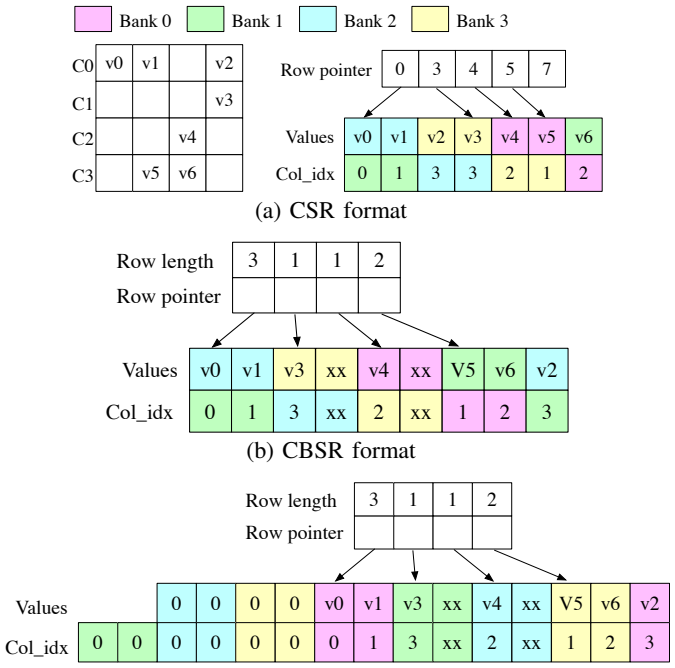


Fig. 4: CSR and CBSR Sparse Tensor Formats

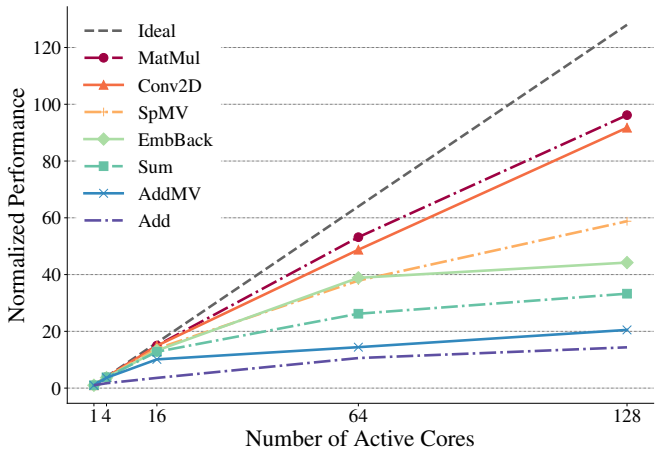


Fig. 5: ATen Operator Micro-Benchmarking – Scalability of a representative set of ATen operators. See Table I for operator description and input sizes. Normalized to single core performance.

levels of emulation, simulation, and finally hardware execution:

a) Emulation Backend: We first develop both the CPU and manycore functions of PyTorch operators using the emulation backend (Fig. 2(b)). Emulation provides the same APIs as the actual manycore co-processor runtime. It enables functional verification, fast turnaround time, and standard debugging tools (e.g, gdb) on manycore device functions. When building with the emulation backend, offloading uses native function calls, data migration uses regular memory copy, and device functions will be executed natively on the host.

b) Cosimulation Backend: After functional verification, we move to cycle-accurate RTL simulation (Fig. 2(c)). In this environment, we again verify correctness, and iterate to optimize performance with architectural counters. The cosimulation backend leverages an RTL simulator (e.g., Verilator[5]) to model a small-scale version of the HammerBlade system running at 1GHz with 16 columns and 8 rows. To model DRAM timing we use the open-source DRAMSim3 library [34], a timing accurate simulator. Architectural performance counters are inserted using non-synthesizable SystemVerilog `bind` statements for no-cost performance analysis of kernels. The RTL for this design has been validated in silicon. Host code executes natively on an Intel Xeon E7-8867v4 CPU.

c) Prototype Backend: Eventually, we plan to support moving to a real FPGA/ASIC prototype (Fig. 2(d)). Preliminary work has demonstrated the feasibility of using an FPGA prototype to study larger workloads than possible in simulation.

B. Micro-Benchmarking

We conduct a scalability study on a set of representative PyTorch operators shown in Table I. These operators vary in arithmetic intensity and enable understanding the performance of our framework on the target CPU-manycore heterogeneous system. Fig. 5 shows that arithmetic-intensive operators, such as MatMul and Conv2D, scale well and achieve a sustained throughput of 78.5 GFLOP/s and 68.0 GFLOP/s, respectively. Memory-intensive dense operators, such as AddMV, Sum, and Add, show only moderate scalability, as they can easily saturate the manycore co-processor's memory bandwidth. EmbBack is implemented with fine-grained locking, in which each embedding entry is associated with a spin-lock to resolve update conflicts and scales well up to 64 active cores. However, increased memory latency, instead of lock contention, is the primary reason EmbBack scales poorly to 128 active cores. SpMV scales better than other memory-intensive operators because of the CBSR tensor layout, which is specifically designed to avoid LLC bank conflicts on the target manycore co-processor.

We study four operators that are critical to many real-world tensor workloads in more detail: MatMul, Conv2D, AddMV, and SpMV. Fig. 6 shows that the cycles per instruction (CPI) increases with the number of active cores. For arithmetic-intensive operators such as MatMul and Conv2D, the number of stall-on-network cycles (i.e., load/store requests to LLC cannot be sent due to network congestion) reduces overall performance after reaching 64 active cores (see Fig. 6 (a–b)). Even with only one active core, MatMul and Conv2D cannot hide enough memory latency to avoid stall-on-use (i.e., true data dependency). Both MatMul and Conv2D can use tiling. Larger tiling blocks increase data reuse resulting in higher arithmetic intensity and thus better performance. However, the necessity of moving large data blocks to the scratchpads with in-order scalar cores introduces phased behavior into these arithmetic-intensive operators. A data-loading phase moves a large block of data into the scratchpad, followed by an execute phase to consume the data block. To move data to the scratchpads, we use a pair of regular load and store instructions.

[5]https://github.com/verilator/verilator

TABLE I: Operator Micro-Benchmarking

ATen Operator	Description	PyTorch Operator	AI	Input
MatMul	Matrix-Matrix Multiplication	`torch.mm`	High	$256 \times 256 \times 256$
Conv2D	2D Convolution	`torch.convolution`	Medium	32×32 input w/ 16 channels, 16 3×3 Filters, 32 Images Batch
AddMV	Matrix-Vector Multiplication	`torch.addmv`	Low	1024×128
SpMV	Sparse Matrix-Vector Multiplication	`torch.mv`	Low	FB-Johns55, 5157×5157 sparse matrix, density 1.4%
Sum	Reduction	`torch.sum`	Low	One Tensor w/ $192,000$ Elements
EmbBack	Backpropagation of Embedding	`torch.nn.Embedding`	Low	600×100 Embedding Table, 256 Records Batch, 50 Entries per Record
Add	Element-Wise Add	`torch.add`	Low	Two Tensors w/ $131,072$ Elements Each

Inputs used in the operator micro-benchmarking. See Figure 5. AI = arithmetic intensity.

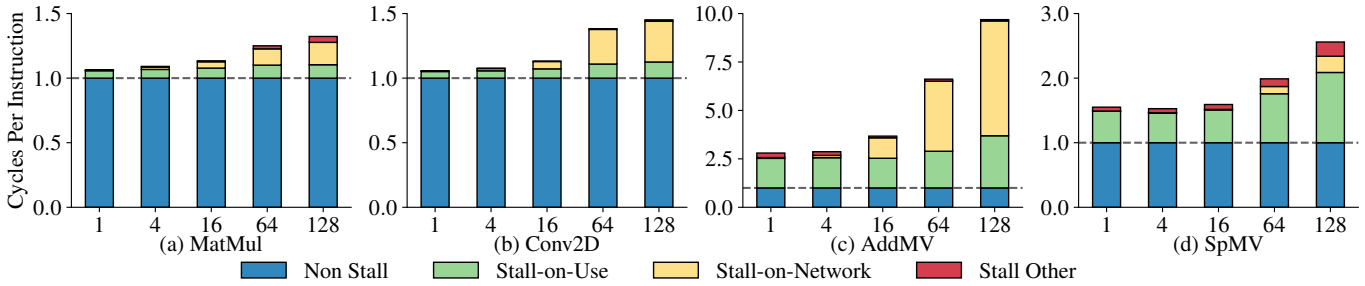


Fig. 6: Per Core Cycles Per Instruction – Cycles per instruction continues to increase with the number of active cores. Memory latency dominates execution time in all four operators when using 128 cores. Stall-on-Network = load request cannot be sent due to OCN contention; Stall-on-Use = load request has been sent but response haven't received; memory latency = Stall-on-Network + Stall-on-Use.

A core first loads a word into one of its registers and then explicitly stores the data into its core-local scratchpad. We can hide memory latency by unrolling the loop so that the instruction stream has a long sequence of loads followed by a long sequence of stores. With stall-on-use, we are able to have many memory requests in-flight which amortizes the memory latency. However, even after applying these optimizations, memory latency still contributes significantly to the overall execution time.

For memory-intensive operators, such as AddMV and SpMV, the number of stall cycles increases quickly beyond 16 active cores (see Fig. 6 (c–d)). This is likely due to a limited number of LLC banks. With more active cores than available LLC banks, even if memory accesses from cores can be evenly distributed, LLC contention remains. Fig. 6 shows that unlike AddMV, SpMV execution time is dominated by stall-on-use cycles instead of stall-on-network cycles. This indicates the CSBR tensor layout is able to significantly reduce network congestion.

IV. Software-Enabled DAE

Section III confirmed that memory latency is a major factor in the performance of both dense and sparse tensor operators on the target architecture. We expect memory latency to become an even more significant issue in future CPU-manycore heterogeneous systems with thousands of cores and 2D mesh on-chip networks, as bisection bandwidth and bandwidth going off the mesh to higher levels of the memory hierarchy scale linearly while the number of cores scales quadratically. We can either tolerate the ever growing memory latency, or we can reduce the amount of data transferred. GPGPUs explored both directions through extreme temporal multithreading with fine-grain context switching (latency hiding) and memory coalescing (reducing data movement). As demonstrated for conventional processors in prior work [25], [35], [36], decoupled access/execute can reduce or eliminate memory latency and improve performance. In this section, we leverage software-based decoupled access/execute to realize both latency hiding and data movement reduction in the context of a manycore architecture. We propose naïve-software DAE and systolic-software DAE, and we then evaluate their performance against optimized data-parallel baseline implementations.

A. Naïve-Software DAE

We first explore decoupled access/execute using pairs of cores: one as the access core and one as the execute core. In a typical DAE architecture, access and execute are connected by hardware queues for communication. In the context of a PGAS manycore, we leverage remote store programming and create software queues in the execute core's scratchpad for the same purpose. We refer to this software decoupled access/execute scheme as *naïve-software DAE*.

In naïve-software DAE, the access core sends requests to higher levels of the memory hierarchy to load data into its registers. Unlike the data-movement scheme described in Section III, the access core stores the loaded value into its peer's scratchpad (i.e., the software queue). When data becomes available, the execute core reads the data block, performs computation, yields the queue space, and writes back the results (if necessary). In many DAE architectures, writing back the results is also done by the access core. However, our early analysis suggested writing results from an execute core to an access core, and then to higher levels of memory hierarchy provided no benefit. Thus, in naïve-software DAE, execute cores write results directly back to DRAM. Since the block currently being processed stays in the software queue (i.e., the execute core pops the entry only after finishing computation), at least two entries in each software queue are necessary to enable access/execute decoupling. This puts increased demand

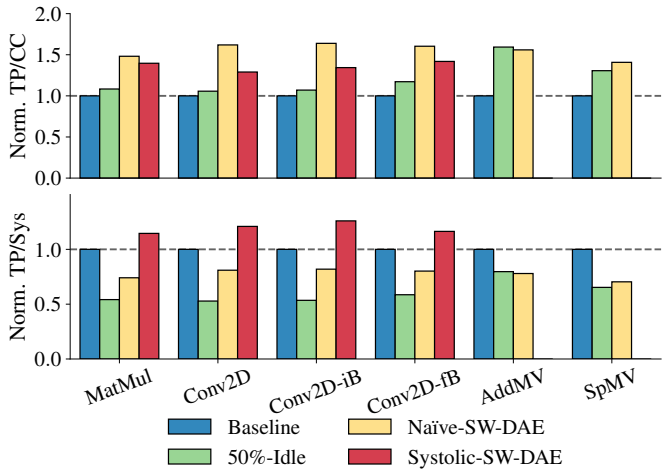


Fig. 7: Naïve and Systolic Software DAE – TP/CC = throughput per compute core; TP/Sys = overall throughput per system; MatMul showing 768 × 768 × 768; Conv2D, Conv2D-iB, and Conv2D-fB showing 32 images batch; AddMV showing 768 × 768; SpMV showing FB-Johns55. See Table II for detailed input specification.

on the scratchpad resulting in smaller tile sizes compared to a data-parallel baseline.

We implement six operators with naïve-software DAE: MatMul, Conv2D, Conv2D-iB (i.e., Conv2D backward w.r.t. input images), Conv2D-fB (i.e., Conv2D backward w.r.t. filters), AddMV, and SpMV. The baselines are hand-tuned data-parallel implementations. We add a second baseline for each operator, in which we only activate 50% of the cores in the manycore co-processor using the data parallel implementation. We refer to this second baseline as 50%-idle. We include this baseline to understand if the benefit of naïve-software DAE comes from fewer cores making memory requests. Since the target manycore is built with scalar cores, each core can inject at most one memory request every cycle. With only 50% cores active, the maximum possible new requests per cycle is halved. This may relieve network congestion and improve operator performance.

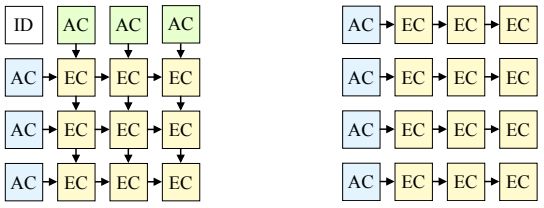
Results are summarized in Fig. 7 and Table II. Compared to the baseline, 50%-idle generally achieves much lower overall throughput, as expected with half of the cores active. However, we also observe an increase in per-core throughput, especially in the cases of AddMV and SpMV. This improvement matches our observation in Section III, that increasing the number of active cores can *reduce* performance due to network congestion. We also observe that for these two operators, naïve-software DAE only provides marginal improvement, or hurts performance because low arithmetic intensity means there is not enough time for the access core to load a block before the execute core needs to consume this block. However, for arithmetic-intensive operators (i.e., MatMul, Conv2D, Conv2D-iB. and Conv2D-fB), naïve-software DAE significantly improves the per-compute-core throughput. Compared to the baseline, naïve-software DAE is able to improve per-compute-core throughput by 1.5–1.9×. Compared to 50%-idle, naïve-software DAE is able to improve per-compute-core throughput by 1.3–1.5×, despite using smaller tiling block sizes than both the baseline and 50%-idle. While this improvement over 50%-idle partially comes from having 2× the resources and offloading load and address generation instructions to access cores, the main source of performance benefit comes from memory-latency hiding. In Conv2D, 13% of the dynamic instructions are related to load and address generation, and these instructions are offloaded to access cores. However, we observe 53% performance improvement over 50%-idle.

B. Systolic-Software DAE

While naïve-software DAE implementations show significant per-compute-core improvement, the overall performance decreases because the per-compute-core improvement does not outweigh the reduced number of compute cores performing useful work. To translate the high per-compute-core throughput to an overall performance improvement, we must change the ratio of access to execute cores. However, having one access core serve two or more execute cores can also degrade performance when the execute cores finish faster than the access core can supply data. For example, in MatMul an access core cannot finish loading data for two execute cores before its execute counterparts finish consuming their current blocks, and thus the execute cores will need to stall. Alternatively, multiple access cores could fetch data for a single execute core. Unfortunately, an asymmetric ratio of access and execute cores results in access cores writing data to execute cores located multiple hops away, which can increase network congestion and further slow down data transfers. Instead of having an access core load independent data blocks for each execute core it serves, we can exploit the fact that the same data is needed by multiple execute cores by intelligently placing the compute and having execute cores pass data blocks in a systolic fashion (i.e., in-compute array reuse). We call this scheme *systolic-software DAE*. Since systolic-software DAE is only feasible for operators with significant data reuse, we focus on the arithmetic-intensive operators (i.e., MatMul, Conv2D, Conv2D-iB, and Conv2D-fB) in the following sections.

The systolic-software DAE implementation of MatMul uses a similar approach as output-stationary systolic hardware accelerators for MatMul, although the systolic-software DAE implementation operates at block granularity instead of scalar value granularity. In systolic-software DAE, blocks of input data are loaded by access cores on the West and North edges of the manycore array, and these blocks are passed along either horizontally or vertically (see Fig. 8(a)). The systolic-software DAE implementation of Conv2D is implemented in a 1D systolic manner with replication. An input block is passed along a chain of execute cores, in which each execute core applies a different filter to the block (see Fig. 8(b)). MatMul and Conv2D implemented with systolic-software DAE on a 128-core device has 64% or 88% more respectively execute cores compared to naïve-software DAE.

We implement the four arithmetic-intensive operators (i.e., MatMul, Conv2D, Conv2D-iB, and Conv2D-fB) with systolic-software DAE. Results are summarized in Fig. 7 and the systolic-software DAE columns of Table II. Conv2D-iB and Conv2D-fB can be implemented in ways that are similar to Conv2D and MatMul, respectively. Across all four operators, systolic-software DAE has a per-compute-core throughput that is lower than naïve-software DAE, but still up to 1.5× higher than the data-parallel baseline. This is because execute cores



(a) SSD MatMul (b) SSD Conv2D

Fig. 8: Systolic Mapping – SSD = systolic-software DAE; ID = idle core; AC = access core; EC = execute core. In (a) data is loaded by access cores, and is passed along by execute cores to the South and to the East, while in (b) data is passed in one direction only.

in systolic-software DAE need to pass data blocks to their neighboring execute cores in addition to performing the actual computation. Additional instructions for data movement lead to lower throughput. However, systolic-software DAE benefits from the additional execute cores, and achieves up to 1.25× increased system throughput. Note that systolic-software DAE also has fewer compute cores than the baseline. There are three cases (i.e., Conv2D with a batch size of 2 and Conv2D-fB with a batch size of 2 and 4) where systolic-software DAE performs worse than the baseline. This is because in systolic-software DAE data blocks need to be passed from execute core to execute core. Thus, there is a much longer warmup phase for systolic-software DAE, and this results in worse performance when the batch size is small.

V. Hardware-Accelerated DAE

Naïve-software DAE and systolic-software DAE leverage existing hardware mechanisms in the CPU-manycore heterogeneous system and demonstrate both per-compute-core and per-system throughput improvements. However, software-only approaches have two disadvantages. First, general-purpose cores are area-inefficient for data access tasks. Most access tasks only require basic integer arithmetic and simple control flow for 1D and 2D array accesses, but cores in the manycore co-processor are equipped with instruction caches, data scratchpads, and floating point units. Second, dedicating general-purpose cores to data access tasks reduces the peak throughput of the manycore co-processor. While systolic-software DAE can help mitigate this issue by reducing the number of access cores, most operators still require the first column and/or the first row of cores in the manycore co-processor to load data.

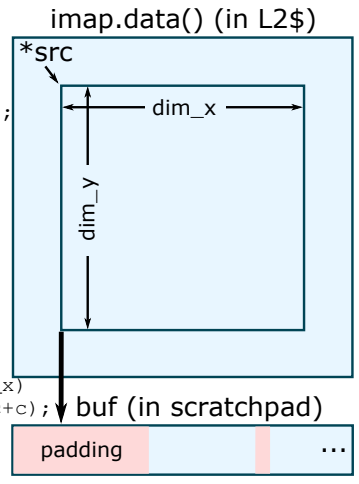
We adopt a software/hardware co-design approach to address these challenges. We design and implement an *access accelerator (AX)*, a configurable hardware unit that streams data from the LLC to the scratchpad of a target execute core. Compared to general-purpose cores, an access accelerator is significantly more area efficient, yet still provides the benefits of decoupled access/execute. This light-weight access accelerator also achieves the same peak computation throughput as the baseline manycore with very low area overhead. While having hardware engines that are dedicated for moving data (e.g., DMA engines) is not a new idea, the proposed access accelerator is unique in its ability to act as a first-class citizen in both the mesh-based on-chip network and the remote store programming model.

```
float* src = imap.data()
  + img_id  * strides[0]
  + chnl_id * strides[1]
  + blk_y   * strides[2]
  + blk_x   * strides[3];

if ( pad_first_col )
  pad_col(buf, 0);
if ( pad_last_col )
  pad_col(buf, dim_x-1);
if ( pad_first_row )
  pad_row(buf, 0);
if ( pad_last_row )
  pad_row(buf, dim_y-1);

for r in range(0, dim_y)
  for c in range(0, dim_x)
   buf[offset+c] = *(src+c);
  src += strides[2];
  offset += dim_x;
}
```

(a) Data Access Pseudo Code



(b) Data Access Illustration

Fig. 9: Conv2D Forward Data Access – In the Conv2D forward kernel, the access cores run program in (a) and load input feature map blocks into the target data scratchpad as shown in (b). Note the access cores calculate `src` and pad zeros (in red) to the imap buffer.

A. Access Accelerator Design

Data Access Tasks – Fig. 9 shows the data access pseudocode of the Conv2D kernel and illustrates how the access cores load data from the LLC and pad zeros to the input feature map block. While we explored several operators with software-only DAE schemes, their data access patterns are all similar. In general, data access tasks involve two nested `for` loops that load a matrix of size `dim_x` by `dim_y` into the scratchpad of the target execute core and an optional padding process that pads zeros around the matrix. This generic data access pattern can be efficiently implemented as an access accelerator that correctly performs common data access tasks given the metadata about the accesses (i.e., the source address, dimensions, strides, padding information, and the destination address).

Accelerator Design – Fig. 10(a) shows the architecture of the access accelerator and how it is connected to a mesh network router. At the core of the access accelerator is a configurable address generator and a padding engine. These two modules generate a stream of memory requests. Since the mesh network in the target manycore system is only point-to-point ordered, the access accelerator also includes a reorder queue to reorder the memory responses from different LLC banks. The request arbiter arbitrates between memory read requests to the LLC and remote store requests to the target scratchpad because there is only one master interface exposed by the mesh network router. Finally, an address translator is required because the execute cores configure access accelerators using virtual addresses.

Accelerator Integration – Fig. 10(b) illustrates how access accelerators are integrated into the target manycore array. In the baseline manycore, each mesh network router is connected to a RISC-V core. To integrate the access accelerators, we extend the mesh network and instantiate access accelerators at the top row and the left-most column. This composition works particularly well with systolic-software DAE implementations where most on-chip network traffic is between neighboring cores or accelerators. This composition also ensures a fair comparison with the baseline manycore system for two rea-

TABLE II: Operator Throughput

		Baseline		50%-Idle		NSD		SSD		NAD		SAD	
Operator	**Input**	**TP/C**	**TP/S**	**TP/C**	**TP/S**	**TP/C**	**TP/S**	**TP/C**	**TP/S**	**TP/C**	**TP/S**	**TP/C**	**TP/S**
MatMul	$768 \times 48 \times 768$	0.53	67.8	0.60	38.3	0.89	57.2	0.67	70.4	0.86	81.5	0.64	79.6
	$768 \times 96 \times 768$	0.56	71.6	0.63	40.1	0.92	58.9	0.74	77.9	0.92	87.9	0.71	88.1
	$768 \times 192 \times 768$	0.60	77.3	0.66	42.5	0.95	60.9	0.78	81.7	0.95	90.6	0.75	93.3
	$768 \times 384 \times 768$	0.60	76.5	0.66	42.5	0.96	61.4	0.80	83.7	0.97	92.1	0.77	95.9
	$768 \times 768 \times 768$	0.57	73.6	0.62	39.9	0.85	54.5	0.80	84.3	0.97	92.4	0.78	96.4
Conv2D	Batch Size 2	0.46	58.7	0.52	33.3	0.79	50.4	0.48	57.5	0.74	70.2	0.46	57.5
	Batch Size 4	0.50	63.5	0.55	35.3	0.82	52.6	0.58	69.4	0.78	74.0	0.57	71.0
	Batch Size 8	0.52	66.2	0.56	35.6	0.84	54.1	0.64	76.2	0.80	75.9	0.63	78.9
	Batch Size 16	0.52	67.2	0.56	35.8	0.86	54.7	0.67	80.2	0.81	76.9	0.66	82.0
	Batch Size 32	0.53	68.0	0.56	35.9	0.86	55.0	0.68	82.0	0.81	77.4	0.67	83.6
	Batch Size 64	0.53	68.2	0.56	35.9	0.86	55.2	0.69	82.5	0.82	77.8	0.68	84.3
Conv2D-iB	Batch Size 2	0.46	59.2	0.54	34.4	0.78	50.0	0.49	59.2	0.73	69.7	0.46	56.9
	Batch Size 4	0.50	63.7	0.55	35.3	0.82	52.5	0.59	70.8	0.77	73.7	0.57	70.7
	Batch Size 8	0.52	66.1	0.56	35.6	0.84	53.6	0.65	77.6	0.80	75.9	0.64	79.7
	Batch Size 16	0.52	67.0	0.56	35.7	0.85	54.4	0.68	82.1	0.81	77.2	0.66	81.9
	Batch Size 32	0.52	66.9	0.56	35.8	0.86	54.8	0.70	84.0	0.82	77.7	0.67	83.2
	Batch Size 64	0.53	68.2	0.56	35.9	0.86	55.0	0.71	85.6	0.82	78.0	0.67	83.6
Conv2D-fB	Batch Size 2	0.32	41.3	0.49	31.2	0.64	41.2	0.34	35.4	0.64	60.9	0.28	34.5
	Batch Size 4	0.39	49.5	0.53	33.9	0.76	48.5	0.46	48.0	0.72	68.6	0.40	49.4
	Batch Size 8	0.44	55.9	0.55	35.0	0.76	48.5	0.56	59.2	0.77	73.2	0.51	64.0
	Batch Size 16	0.46	58.3	0.56	35.5	0.75	48.0	0.64	66.7	0.79	75.2	0.58	72.0
	Batch Size 32	0.47	60.6	0.56	35.5	0.76	48.6	0.67	70.6	0.80	76.4	0.61	76.2
	Batch Size 64	0.47	60.0	0.56	35.9	0.76	48.6	0.69	72.9	0.80	75.9	0.63	78.9
AddMV	256×256	0.02	3.0	0.04	2.5	0.04	2.4	–	–	–	–	–	–
	512×512	0.02	3.1	0.04	2.5	0.04	2.7	–	–	–	–	–	–
	768×768	0.03	4.4	0.05	3.5	0.05	3.4	–	–	–	–	–	–
	1024×1024	0.03	3.7	0.04	2.9	0.05	3.1	–	–	–	–	–	–
SpMV	FB-Johns55	0.04	4.9	0.05	3.2	0.05	3.5	–	–	–	–	–	–
	Facebook	0.02	2.9	0.03	2.2	0.04	2.5	–	–	–	–	–	–
	Cora	0.01	1.0	0.01	0.8	0.02	1.0	–	–	–	–	–	–
	CiteSeer	0.01	0.9	0.01	0.7	0.01	0.9	–	–	–	–	–	–

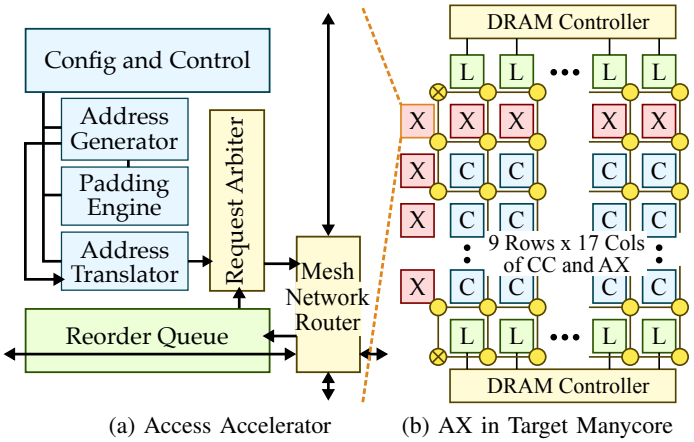
MatMul = matrix multiplication; Conv2D = 2D convolution; Conv2D-iB = 2D convolution backward w.r.t. input image; Conv2D-fB = 2D convolution backward w.r.t. filters; AddMV = general matrix-vector multiplication; SpMV = sparse matrix-vector multiplication; TP/C = throughput per compute core; TP/S = overall throughput per system; NSD = naïve-software DAE; SSD = systolic-software DAE; NAD = naïve-accelerated DAE; SAD = systolic-accelerated DAE. The target system has 128 cores. Conv2D, Conv2D-iB, Conv2D-fB are run with 16-channel 32×32 images with 32 3×3 filters. FB-Johns55 has sparsity of 1.4×10^{-2}; Facebook has sparsity of 5.4×10^{-3}; Cora has sparsity of 1.4×10^{-3}; CiteSeer has sparsity of 8.3×10^{-4}. Per system throughput in naïve-accelerated DAE and systolic-accelerated DAE are area-normalized to baseline manycore. All numbers are in GFLOP/s.

sons. First, the access accelerator manycore (AX manycore) has the same number of LLC banks and the same DRAM bandwidth as the baseline manycore. Second, the AX manycore has the same effective mesh network bandwidth as the baseline. The AX manycore mesh network does have larger bisection bandwidth than in the baseline manycore. However, this additional bandwidth does not translate into improved throughput because the extra network links and routers are mostly used to provide access to LLC banks to the access accelerators. The AX is a first-class citizen in the remote store programming model: execute cores control a neighbor AX by performing remote stores into the AX's memory-mapped control registers, and the AX performs remote stores into its neighboring execute core's scratchpad upon receiving data from the LLC.

B. Access Accelerator Evaluation

Area – Fig. 11 compares the post-place-and-route area of an access accelerator in a CMOS 14/16 nm technology and a general-purpose core from prior work in a similar process [37]. We can see from the figure that the access accelerator is highly area-efficient. The network router and endpoint consumes about 40% and the accelerator data path consumes about 30% of the access accelerator area. The transmit adapter (TX) includes a 32-element FIFO to buffer responses from the LLC, and consumes around 30% of the accelerator area. Overall, the access accelerator is 5× smaller than the general-purpose core, making it an area-efficient choice for data access tasks. The AX manycore (with an extra AX row and AX column as shown in Fig. 10(b)) only increases the overall area by 2.9% compared to the baseline manycore.

Naïve-Accelerated DAE – Similar to the naïve-software DAE evaluation (NSD, see Section IV-A), we evaluate the area efficiency of the access accelerators using a naïve-accelerated DAE (NAD) composition. In NAD, each execute core is paired with an access accelerator that replaces the access core. Fig. 12(a) and the NAD column of Table II shows the per-compute-core throughput and the area-normalized per-system throughput of different operators under NAD. We can see that compared to NSD, NAD has similar per-compute-core throughput since both access cores and access accelerators are able to decouple data access from the computation on execute cores. However, NAD has significantly higher area-normalized per-system throughput (46% on average) than



(a) Access Accelerator (b) AX in Target Manycore

Fig. 10: Access Accelerator Architecture and Integration – (a) architecture of the access accelerator and how it connects to a mesh network router; (b) access accelerators integrated in the first row and first column of the target manycore. X = access accelerator (AX), L = LLC bank, C = compute core (CC).

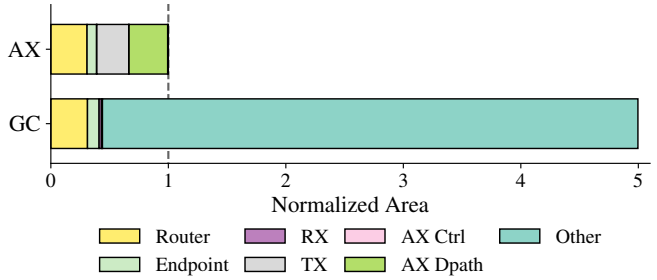
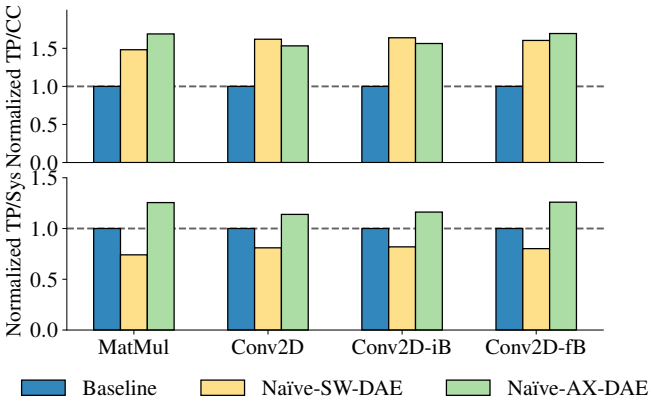
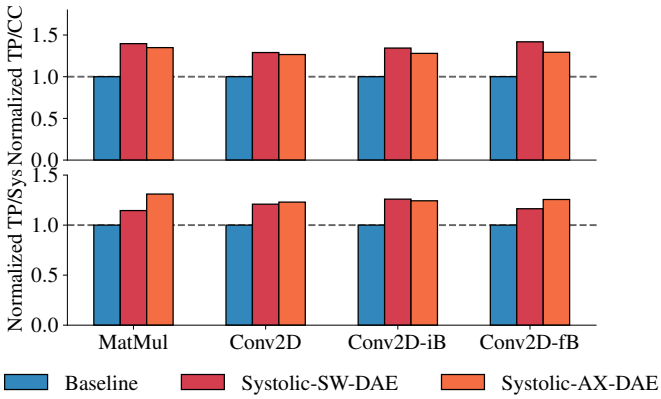


Fig. 11: Access Accelerator (AX) and General-Purpose Core (GC) Normalized Area – AX eliminates instruction cache, data scratchpad, FPU, etc. and is 5× smaller than a GC in a similar CMOS technology. RX/TX = RX/TX adapter, Ctrl = control logic, Dpath = data path.



(a) Naïve Accelerated DAE



(b) Systolic Accelerated DAE

Fig. 12: Naïve and Systolic Accelerated DAE – TP/CC = throughput per compute core; TP/Sys = overall throughput per system; MatMul showing 768 × 768 × 768; Conv2D, Conv2D-iB, and Conv2D-fB showing 32 images batch; AddMV showing 768 × 768; SpMV showing FB-Johns55. See Table II for detailed input specification.

NSD. This difference is the largest on the matrix multiplication (MatMul) operator, where NAD achieves 52% higher area-normalized per-system throughput. The superior area-normalized per-system throughput of NAD over NSD confirms that our access accelerator is significantly more area-efficient on data access tasks than general-purpose cores, and still provides the same throughput benefits of DAE. We did not implement and evaluate NAD versions of memory-intensive operators (i.e., AddMV and SpMV). NAD cannot address the fact that these operators are largely limited by memory bandwidth. Prior evaluation has shown that a data-parallel scheme is more effective (see Section IV-A).

Systolic-Accelerated DAE – As discussed earlier, systolic-software DAE dedicates multiple general-purpose cores to load data at the cost of manycore compute resources. Based on the systolic-software DAE (SSD, see Section IV-B), we create the systolic-accelerated DAE composition (SAD), which uses the access accelerator manycore described in Section V-A to run systolic-software DAE implementations. Fig. 12(b) and the SAD column of Table II shows the per-compute-core throughput and area-normalized per-system throughput of different operators under SAD. We can see that compared to SSD, SAD has similar per-compute-core throughput since both designs are able to achieve decoupled access/execute. In terms of overall area-normalized per-system throughput, SAD has an average of 4.8% better throughput than SSD. On MatMul, SAD is able to achieve 13.9% better average throughput than SSD. On the target 16×8 manycore array, the SSD approach uses eight (Conv2D and Conv2D-iB) or 23 (MatMul and Conv2D-fB) general-purpose cores for data accesses. Therefore, the maximum overall per system throughput improvement of SAD on the same manycore is 6% or 18% (depending on the kernel). In addition, the execute cores in SAD need to perform remote memory stores to configure the access accelerators for every input feature map block, which occupies computation cycles. Despite having more moderate throughput improvements over the highly optimized SSD design, SAD still achieves the highest area-normalized throughput on the four evaluated kernels among all six designs (baseline, 50%-idle, NSD, SSD, NAD, SAD). Compared to the baseline, the AX manycore introdues one extra cycle to the memory latency when accessing LLC banks in the north. However, this should have negligible performance impact on operators that cannot leverage SAD, as our prior results in Section III-B have shown that network congestion is the main source of stalls for operators implemented with a data-parallel scheme.

VI. First-Order Analysis of SW/HW Scalability

In this section, we conduct first-order end-to-end evaluation on three tensor workloads to evaluate our framework's ability to enable optimized dense and sparse tensor processing on CPU-manycore heterogeneous systems with minimal modifications to existing workloads. We first introduce the workloads and then describe our evaluation methodology. We finish by estimating the performance of the these workloads when scaled to a future 2,000-core CPU-manycore heterogeneous system against an aggressive multicore CPU.

A. Emerging Tensor Workloads

a) Residual Neural Network (ResNet): Residual neural networks are one form of convolutional neural networks (CNNs) for image classification, which won the 2015 ImageNet Large Scale Visual Recognition Challenge by allowing the network's accuracy to scale with its depth [38]. ResNet introduces *residual blocks*, which are shortcut connections between non-neighboring layers, to overcome a number of training difficulties (e.g., vanishing gradient problem) faced by conventional CNN models. In this work, we build and train a 9-layer ResNet model (i.e., ResNet-9) on the CIFAR-10 dataset.

b) Recommender System (RecSys): The input to a recommender system is a list of items a user has previously "liked", and the output is a list of items with scores predicting how much the user might like an unseen item. An autoencoder is a specific kind of unsupervised artificial neural network that learns to copy its input to its output through an intermediate "bottleneck" layer for dimensionality reduction. In this work, we build and train this recommender system on the MovieLens 10M dataset.

c) Local Graph Clustering (LGC-ISTA): Local graph clustering is an approximate variant of the personalized PageRank algorithm. Its goal is to find a cluster of nodes that are neighbors of a given seed node. We implement iterative shrinkage-thresholding, which minimizes the loss function of a graph signal vector such that all nodes in the neighborhood of the seed node are associated with high scores, while other nodes receive low scores. The algorithm uses the input adjacency matrix and degree matrix to generate a sparse matrix. It then iteratively updates the gradient, vector, and loss function using SpMV, element-wise multiply, add, and subtraction operations. We run 50 iterations for each seed node on the FB-Johns55 dataset.

B. Methodology

A common practice to evaluate full-size workloads on simulators is to extract each occurrence of the kernels, and evaluate them individually with either random data or reconstructed data outside of PyTorch. However, this approach leads to inaccuracies since random or reconstructed data may not represent the actual data layout during execution. To address this challenge, we have developed a *re-dispatching* approach that automates the evaluation process and preserves runtime data layout. We first determine which operators in a workload we would like to evaluate, flag them, and then start running the workload on the CPU. When a call-site is reached the execution is forked into a CPU instance (running natively), and a manycore instance (running on an RTL simulator). After both runs return, manycore results are validated against CPU results. With re-dispatching, workload evaluation can be easily parallelized by launching many copies of the workload; one copy for each kernel of interest.

Since it is not feasible to simulate a 2,000-core manycore architecture at reasonable simulation speed, we simulate a smaller 128-core heterogeneous system running 1/16 of the work using the co-simulation infrastructure described in Section III. We then scale the performance of the manycore co-processor to a full 2,000-core system via weak scaling. We compare the scaled performance against the performance of running the full workload on the host multicore CPU, which is an aggressive 18-core out-of-order superscalar running at 2.4GHz (Intel Xeon E7-8867v4).

C. Results

By leveraging 2D convolution operators with SAD implementations in ResNet, we estimate ResNet can achieve 2× better performance on the target manycore system than on the aggressive multicore CPU (see Table III). 2D convolution operators run much faster on the manycore system by exploiting massive parallelism, but batch normalization and its backward pass (i.e., BatchNorm and BatchNormBack) perform worse on the manycore system compared to the CPU. This is because frequent synchronization is needed in batch normalization operators, and synchronizing the manycore system currently involves higher overhead than synchronizing a multicore CPU. Compared to having 2D convolution operators implemented with a traditional data-parallel approach, we are able to train ResNet-9 13% faster with systolic-accelerated DAE. Specifically, we observed that Conv2D-fB with systolic-accelerated DAE achieves 2.1× better performance than its data-parallel counterpart, which is higher than we have observed in microbenchmarks (see Table II). Further inspection reveals that unlike the microbenchmarks we used in prior sections, inputs to convolution layers in ResNet do not fit in the LLC. Unstructured memory accesses in the data-parallel implementation lead to significantly more LLC misses.

We estimate RecSys can achieve 5.9× better performance on the target manycore system than on the multicore CPU. Compute intensive operators, such as AddMM and AddMMBack, generally have better performance on the target system because the manycore can better exploit the parallelism in these operators. We also observe that the largest performance improvement comes from embedding (Emb), EmbBack, and Sum. This improvement can be traced to two causes: (1) these operators are memory intensive, and compared to a multicore CPU, the manycore co-processor has a much higher total memory bandwidth (1TB/s); and (2) we apply optimization techniques that are not available by default in the CPU ATen backend such as kernel fusion and intermediate value removal. On the manycore co-processor, we are able to fuse Emb and Sum together to eliminate intermediate value reads and writes. We also explored leveraging systolic-accelerated DAE MatMul in RecSys. However, the dimensions of MatMul instances in RecSys generally lead to severe internal fragmentation [39], and thus worse than baseline performance due to wasted computation. TPUv1 faced a similar issue. Unlike specialized hardware accelerators, we have the flexibility of falling back to

TABLE III: ResNet Execution Breakdown

ATen Operator	Baseline Time (ms)	MC Total Time (ms)	MC Host Time (ms)	MC Device Time (ms)
Conv2DBack	169.9	45.2	0.9	44.3
Conv2D	77.1	21.9	1.3	20.6
BatchNormBack	18.8	38.2	0.5	37.7
BatchNorm	17.8	36.9	1.9	35.0
Relu	8.5	2.2	0.5	1.7
ThresholdBack	6.3	3.1	0.4	2.7
MaxPool2DBack	6.2	1.2	0.5	0.7
MaxPool2D	5.6	1.1	0.7	0.4
Sqrt	4.3	1.8	0.9	0.9
ZerosLike	3.8	3.0	1.6	1.4
Add	3.3	6.2	2.6	3.6
AddCDiv	3.1	2.2	0.9	1.3
Div	3.1	3.0	1.3	1.7
Other	58.4	32.7	27.6	5.1
Data Transfer	0.0	0.03	0.03	0.0
Total (1 Epoch)	611.2 (s)	310.5 (s)	65.0 (s)	245.5 (s)

One training epoch; 1563 batches per epoch; 32 images per batch. MC = target CPU-manycore system. MC total = MC host + MC device.

TABLE IV: Recsys Execution Breakdown

ATen Operator	Baseline Time (ms)	MC Total Time (ms)	MC Host Time (ms)	MC Device Time (ms)
EmbBack	427.8	8.2	1.2	6.0
Emb	94.8	1.4	0.5	0.9
Sum	35.7	0.0	0.0	0.0
AddmmBack	23.3	16.4	2.4	14.0
ZerosLike	15.1	4.9	3.9	1.0
CrossEntropyLoss	14.4	10.6	2.7	8.9
Addmm	11.1	7.7	0.5	7.2
BatchNorm	10.1	11.6	1.6	10.0
Addcdiv	8.3	5.4	2.2	3.2
Sqrt	8.3	8.5	1.9	6.6
Div	8.1	7.8	3.4	4.4
BatchNormBack	8.0	8.6	0.6	8.0
Add	7.9	8.9	5.1	3.8
Mul	7.4	11.6	6.6	5.0
Dropout	6.9	6.1	1.4	4.7
Other	17.9	12.4	5.4	7.0
Data Transfer	0.0	3.5	3.5	0.0
Total (1 Epoch)	185.5 (s)	31.5 (s)	11.2 (s)	20.3 (s)

One training epoch; 273 batches per epoch; 256 users per batch. MC = target CPU-manycore system. MC total = MC host + MC device.

TABLE V: Local Graph Clustering Execution Breakdown

ATen Operator	Baseline Time (ms)	MC Total Time (ms)	MC Host Time (ms)	MC Device Time (ms)
SpMV	23960.0	2267.4	1776.0	491.4
Sub	365.9	1120.0	1024.0	96.0
Add	368.8	544.0	496.0	48.0
Max	759.5	480.0	432.0	48.0
Mul	31.1	65.9	56.3	9.6
Clone	0.2	9.6	9.0	0.6
Data Transfer	0.0	2.3	2.3	0.0
Total	25.5(s)	4.5(s)	3.8(s)	0.7(s)

Personalized PageRank for 500 seed nodes; 50 iterations per seed node. MC = target CPU-manycore system. MC total = MC host + MC device.

a data-parallel implementation with a manycore architecture. We believe other workloads which have more systolic DAE friendly MatMul dimensions will see significant benefits.

We estimate LGC-ISTA can achieve 5.7× better performance on the target manycore system than on the multicore CPU. We observe that unlike RecSys, clustering spends more time on the CPU host than on the co-processor. This is because the input graph has high sparsity, and thus manycore device functions for those operations will not run for long enough time to cover the offloading overhead.

In summary, we estimate all three workloads will be able to achieve much higher (i.e., up to 5.9×) performance on the target CPU-manycore heterogeneous system compared to an aggressive multicore CPU baseline. Note that the weak scaling approach we adopt is optimistic and meant for demonstrating the potential of a future full manycore system, rather than as a rigorous comparison. While computing 1/16 of the output on a 128-core system demonstrates that we have enough software parallelism to fully utilize the 2,000-core system, various architectural challenges (e.g., LLC coherence, DRAM channel scaling, and cross channel data movement) must be solved with minimal performance penalty to realize the estimated performance. This work provides a software stack that lays the groundwork for researchers to explore solutions to these challenges in future work. To help estimate how a future 2,000-core system might compare to a GPGPU, we can consider a previously proposed manycore architecture with 496 RISC-V cores [40], [37]. This prior work has shown the ability to achieve 93.04 Giga RISC-V instructions/s per watt and 45.57 GRVIS/mm^2. Given these prior results, the target CPU-manycore heterogeneous system can potentially achieve significantly higher area-normalized throughput and energy efficiency compared to GPGPUs. Again, this work provides a software stack that can enable more detailed comparative analysis of manycore architectures versus GPGPUs and other programmable accelerators.

VII. Related Work

A wide variety of coarse-grain parallel architectures have been developed over the past decade to exploit pipeline parallelism. Architectures like Eyeriss [41] and DianNao [42] are domain-specific accelerators for convolutional neural networks. Later versions support operations on sparse tensors. These proposals demonstrate similar parallel dataflow patterns. The TPU [43] and VTA [44] architectures integrate systolic matrix-multiply and vector processing units to accelerate more general machine learning computations. More general purpose architectures also exist: RAW [45] uses an inter-processor scalar operand network to forward results between processors. Plasticine [46] contains a mesh of general-purpose compute units for processing workloads from machine learning, data, and graph analytics. These architectures exploit pipeline parallelism by composing coarse grain functional units, similar to our work.

Many architectural solutions have been proposed to decouple memory and compute operations [25]. Decoupled Supply Compute (DeSC) [35] is an automatic extension of DAE for general-purpose CMPs that uses a "Supplier Device" and a "Compute Device", similar to our naïve-software DAE approach. The Load Slice Core [24] is a form of restricted out-of-order machine. With an additional pipeline, load and address generation slices can be issued out-of-order and speculatively with respect to compute slices, while remaining in-order within a slice. Slice formation is handled by hardware. Tran et al. [47] proposed a SW/HW co-design method. Instructions are grouped into access and execute phases at compile time. Access phases can run and commit out-of-order with respect to

execute phases at runtime. Both techniques rely on hardware that is more complex than the target manycore architecture provides (e.g., superscalar cores). Manticore [16] introduces custom ISA extensions to leverage DAE and improve FPU utilization. Techniques proposed in this work aim to enable DAE in the context of a manycore with thousands of simple stall-on-use in-order scalar cores, and with existing programming model and core microarchitecture. The Cell processor [36] includes per-core DMA engines to overlap computation with data transfer. The Epiphany processor [15] also includes a DMA engine. This prior work explores pairs of memory and compute engines, while our approach extends this idea with AX's along the periphery of the target architecture. Our approach is more similar to CoRAM [48], where a control thread can manage multiple scratchpads on an FPGA device. Recent work has shown the potential of using a chiplet-based approach to scale the target manycore achitecture to thousands of cores [6], [16].

Several high-level languages have been created to express complex pipeline parallelism in programming. StreamIt [49] exposed pipeline parallelism for the RAW architecture. More recent work has enabled pipeline parallelism for general-purpose machines. Interstellar [50] is an extension to Halide's scheduling with pipeline parallelism expressions. Spatial [51] is a general-purpose DSL for expressing pipelines and can target Plasticine [46]. These languages are higher-level than our own development language and can be used in the future to ease programmer expression of pipeline parallelism on manycore architectures.

One approach to exploiting software pipelines is through parallel frameworks like PyTorch [20]. These frameworks use pre-built libraries with hand-optimized primitives that exploit software pipelines, and abstract designers from the complexity of expression. For example, TVM [52] supports CPUs, GPUs, and also the VTA [53] architecture. TensorFlow [21] has backends for CPUs, GPUs, as well as the Google TPU [43]. Our work adds another backend to these state-of-the-art software stacks.

VIII. Conclusion

Programmability and memory latency are the key challenges in CPU-manycore heterogeneous systems. In this paper, we address the programmability challenge with a tensor processing framework in a high-level library that abstracts hand-optimized operators for dense and sparse workloads. Through end-to-end evaluation of dense and sparse tensor workloads, we show that the proposed framework can potentially achieve up to 5.9× better performance on a 2,000-core CPU-manycore heterogeneous system compared to an aggressive multicore CPU. We address the manycore memory latency challenge by exploring both software and hardware-accelerated decoupled access/execute schemes on the manycore co-processor. Operators implemented with our techniques achieve up to 1.32× throughput improvement, compared to an aggressive data-parallel baseline.

Acknowledgments

This work was supported in part by NSF CRI Award #1512937, NSF SHF Award #1527065, NSF SHF #1909661, DARPA SDH Award #FA8650-18-2-7863, a research gift from Facebook and Xilinx, and equipment, tool, and/or physical IP donations from Intel, Synopsys, Cadence, and ARM. The authors acknowledge and thank Kexin Zheng, Janice Wei, Angela Zou, Yuwei Hu, and Adrian Sampson for using the proposed PyTorch framework and providing useful feedback. The authors also thank Shunning Jiang and Hanchen Jin for their advice in developing domain-specific accelerators for integrating into manycore co-processors, and Zichao Yue for his contributions to the proposed CBSR format. Finally, the authors thank the entire Bespoke Silicon Group at the University of Washington for manycore RTL development and the PyTorch and RISC-V communities for developing and supporting the software infrastructure that serves as the foundation for this work. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DARPA, or the U.S. Government.

References

[1] M. B. Taylor *et al.*, "A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network," *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2003.

[2] M. McKeown *et al.*, "Piton: A manycore processor for multitenant clouds," *IEEE Micro*, vol. 37, no. 2, pp. 70–80, Mar/Apr 2017.

[3] J. Howard *et al.*, "A 48-core ia-32 message-passing processor with DVFS in 45nm CMOS," *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2010.

[4] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz mesh interconnect for a teraflops processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, Sep/Oct 2007.

[5] M. Lis, K. S. Shim, M. H. Cho, I. Lebedev, and S. Devadas, "Hardware-level thread migration in a 110-core shared-memory multiprocessor," MIT CSAIL CSG, Tech. Rep. 512, Nov 2013.

[6] P. Vivet *et al.*, "A 220GOPS 96-core processor with 6 chiplets 3D-stacked on an active interposer offering 0.6ns/mm latency, 3Tb/s/mm2 inter-chiplet interconnects and 156mW/mm2@ 82%-peak-efficiency DC-DC converters," *ISSCC*, Feb 2020.

[7] S. Bell *et al.*, "Tile64 processor: A 64-core soc with mesh interconnect," *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2008.

[8] C. Ramey, "TILE-Gx100 manycore processor: Acceleration interfaces and architecture," *Symp. on High Performance Chips (Hot Chips)*, Aug 2011.

[9] D. Kanter, "Knights Landing reshapes HPC," Sep 2015.

[10] B. Wheeler, "Ampere maxes out at 128 cores," *Microprocessor Report, The Linley Group*, Jul 2020.

[11] T. R. Halfhill, "Thunderx3's cloudburst of threads: Marvell previews 96-core 384-thread arm server processor," *Microprocessor Report, The Linley Group*, Apr 2020.

[12] D. Wentzlaff *et al.*, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, pp. 15–31, Sep/Oct 2007.

[13] S. Davidson *et al.*, "The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, Mar/Apr 2018.

[14] B. Bohnenstiehl *et al.*, "KiloCore: A 32-nm 1000-processor computational array," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 52, no. 4, pp. 891–902, Apr 2017.

[15] A. Olofsson, "Epiphany-V: A 1024-processor 64-bit RISC system-on-chip," *Computing Research Repository (CoRR)*, vol. arXiv:abs/1610.01832, Aug 2016.

[16] F. Zaruba, F. Schuiki, and L. Benini, "Manticore: A 4096-core RISC-V chiplet architecture for ultraefficient floating-point computing," *IEEE Micro*, Mar/Apr 2021.

[17] J. Burgess, "RTX on: The NVIDIA Turing architecture," *Symp. on High Performance Chips (Hot Chips)*, Aug 2019.

[18] M. Mantor, "7nm "Navi" GPU," *Symp. on High Performance Chips (Hot Chips)*, Aug 2019.

[19] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," *Conf. on Neural Information Processing Systems (NeurIPS)*, Dec 2017.

[20] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Conf. on Neural Information Processing Systems (NeurIPS)*, Dec 2019.

[21] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," *Symp. on Operating System Design and Implementation (OSDI)*, Nov 2016.

[22] "cuGraph - GPU graph analytics," Online Webpage, 2020 (accessed Nov 22, 2020), https://github.com/rapidsai/cugraph.

[23] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[24] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The Load Slice Core microarchitecture," *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2015.

[25] J. Smith, "Decoupled access/execute computer architectures," *ACM Trans. on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 289–308, Nov 1984.

[26] A. Brahmakshatriya *et al.*, "Taming the zoo: The unified graphit compiler framework for novel architectures," *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2021.

[27] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," *Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, Jan 2010.

[28] "ATen: A TENsor library for C++11," Online Webpage, 2020 (accessed Nov 22, 2020), https://github.com/zdevito/ATen.

[29] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," *Symp. on FPGAs for Custom Computing Machines (FCCM)*, May 2014.

[30] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2020.

[31] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2020.

[32] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," *Design Automation Conf. (DAC)*, Jun 2013.

[33] D. R. MacIver, Z. Hatfield-Dodds, and many other contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open-Source Software (JOSS)*, vol. 4, no. 43, Nov 2019.

[34] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A cycle-accurate, thermal-capable dram simulator," *Computer Architecture Letters (CAL)*, Jul 2020.

[35] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled supply-compute communication management for heterogeneous architectures," *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2015.

[36] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in Cell's multicore architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, Mar 2006.

[37] A. Rovinski *et al.*, "A 1.4 GHz 695 Giga RISC-V Inst/s 496-core manycore processor with mesh on-chip network and an all-digital synthesized PLL in 16nm CMOS," *Symp. on VLSI Technology and Circuits (VLSI)*, Jun 2019.

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Computing Research Repository (CoRR)*, vol. arXiv:abs/1512.03385, Dec 2015.

[39] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking TPU, GPU, and CPU platforms for deep learning," *Computing Research Repository (CoRR)*, vol. arXiv:abs/1907.10701, Jul 2019.

[40] A. Rovinski *et al.*, "Evaluating Celerity: A 16nm 695 Giga-RISC-V instructions/s manycore processor with synthesizable pll," *IEEE Solid-State Circuits Letters (SSCL)*, vol. 2, no. 12, pp. 289–292, Dec 2019.

[41] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2016.

[42] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2014.

[43] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[44] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "VTA: An open hardware-software stack for deep learning," *Computing Research Repository (CoRR)*, vol. arXiv:abs/1802.04799, Aug 2018.

[45] M. B. Taylor *et al.*, "Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams," *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2004.

[46] R. Prabhakar *et al.*, "Plasticine: A reconfigurable architecture for parallel paterns," *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[47] K.-A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras, "SWOOP: Software-hardware co-design for non-speculative, execute-ahead, in-order cores," *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2018.

[48] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," *Int'l Symp. on Field Programmable Gate Arrays (FPGA)*, Feb 2011.

[49] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2006.

[50] X. Yang *et al.*, "Interstellar: Using Halide's scheduling language to analyze DNN accelerators," *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2020.

[51] D. Koeplinger *et al.*, "Spatial: A language and compiler for application accelerators," *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2018.

[52] T. Chen *et al.*, "TVM: End-to-end optimization stack for deep learning," *Computing Research Repository (CoRR)*, vol. arXiv:abs/1802.04799, Aug 2018.

[53] T. Moreau *et al.*, "A hardware-software blueprint for flexible deep learning specialization," *IEEE Micro*, Sep 2019.

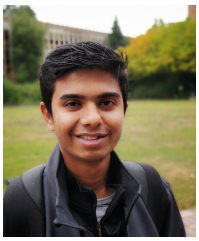
Lin Cheng received the B.S. degree and the M.S. degree in Computer Science in 2017 from University of Illinois at Urbana-Champaign. He is currently a Ph.D. student in Computer Science at Cornell University. His research interests include improving the performance of dynamic languages and supporting them on emerging compute platforms. Contact him at lc873@cornell.edu.

Peitian Pan received the B.S. degree in Computer Science in 2018 from Shanghai Jiao Tong University. He is currently a Ph.D. student in Electrical and Computer Engineering at Cornell University. His research interests include agile hardware development methodologies and computer architecture. He is a student member of the IEEE. Contact him at pp482@cornell.edu.

Zhongyuan Zhao received the B.S. degree from the School of Electronics and Information, Harbin Institute of Technology and the Ph.D. degree in the Department of Nano/Micro Electronics, Shanghai Jiao Tong University. He is currently a Postdoctoral Research Associate at Cornell University. His research interests include compiler and architecture optimization for coarse-grained reconfigurable computing platform and deep learning accelerators, programming language design and performance optimization for manycore architectures. Contact him at zz546@cornell.edu.

Krithik Ranjan is currently pursuing a B.S. degree in Electrical and Computer Engineering at Cornell University, and he is epxected to graduate in May 2022. He is an embedding software engineering intern at Qualcomm Technologies, San Diego, CA. His research interests include embedding systems, robotics, human-computer interaction, and assistive technology. Contact him at kr397@cornell.edu.

Jack Weber received the B.S. degree in Electrical and Computer Engineering at Cornell University in 2021. He is currently an advanced application engineering analyst at Accenture, New York, NY. Contact him at jlw422@cornell.edu.

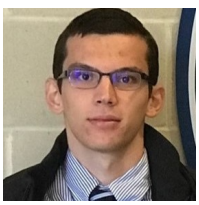


Bandhav Veluri received his B.Tech. degree from IIT Roorkee in 2016 and M.S. degree from University of Washington in 2020. He is currently pursuing a PhD degree with Bespoke Silicon Group and Networks & Mobile Systems Lab at University of Washington. His research interests include Systems, Low-power Sensing and Machine Learning. Contact him at bandhav@uw.edu.

Seyed Borna Ehsani is a Graphics Software Engineer at Apple Inc. He received his B.Sc. degree in Computer Engineering from Sharif University of Technology in 2018, and his M.Sc. degree in Computer Science and Engineering from the University of Washington in 2020. His research interests include computer architecture, GPUs and manycore systems design, 3D graphics, Application Programming Interface design, and parallel programming. Contact him at borna.ehsani@gmail.com.

Max Ruttenberg received his B.S. degree from Lehigh University in 2014. He is currently pursuing his PhD in the Bespoke Silicon Group at the University of Washington. His research interests include computer architecture, parallel programming, high performance computing, graph analytics, and emerging memory technologies. Contact him at mrutt@cs.washington.edu.

Dai Cheol Jung received his B.Sc degree from Brown University in 2015, and his MSc from the University of Washington in 2019. He is currently pursuing a PhD degree at the University of Washington. His research interests include parallel architecture, network-on-chip, and VLSI. Contact him at dcjung@uw.edu.



Preslav Ivanov received his B.S. degree in electrical and computer engineering from Old Dominion University, Norfolk, Virginia in 2020 and is currently pursuing a PhD in Electrical and Computer Engineering at Cornell University. His research focus is in Computer Architecture, particularly modeling application specific accelerators for combinations of performance, energy efficiency, and lowered cost while optimizing algorithms to leverage the new hardware. Contact him at pi57@cornell.edu.

Dustin Richmond is a Postdoctoral Research Associate in the Bespoke Silicon Group at the University of Washington. He received a Ph.D. in computer engineering from the University of California, San Diego in 2018, and B.Sc. degrees from the University of Washington in 2012. His research interests include programming languages, reconfigurable systems, and hardware security. Dr. Richmond was awarded a National Science Foundation Graduate Research Fellowship in 2012, and a Powell Fellowship in 2013. Contact him at dustinar@uw.edu.

Zhiru Zhang (S'02-M'08-SM'18) received the B.S. degree from Peking University, Beijing, China, in 2001, and the M.S. and Ph.D. degrees from the University of California at Los Angeles, Los Angeles, CA, USA, in 2003 and 2007, respectively, all in computer science.

He is an Associate Professor with the School of Electrical and Computer Engineering at Cornell University, Ithaca, NY, USA. Prior to joining Cornell University, he was a co-founder of AutoESL Design Technologies Inc., Cupertino, CA, USA, a high-level synthesis start-up company. He later served as a Software Development Manager with Xilinx Inc., San Jose, CA, USA, after Xilinx acquired AutoESL. His current research interests include new algorithms, architectures, design methodologies, and automation tools for heterogeneous computing.

Dr. Zhang's research has been recognized with the DAC Under-40 Innovators Award, the Rising Professional Achievement Award from the UCLA Henry Samueli School of Engineering and Applied Science, a DARPA Young Faculty Award, the IEEE CEDA Ernest S. Kuh Early Career Award, an NSF CAREER Award, the Ross Freeman Award for Technical Innovation from Xilinx, as well as multiple best paper awards. Contact him at zhiruz@cornell.edu.

Christopher Batten received the B.S. degree in EE from the University of Virginia in 1999, the M.Phil. degree in Engineering from the University of Cambridge in 2000, and the Ph.D. degree in EECS from the Massachusetts Institute of Technology in 2010. He is currently an Associate Professor in ECE at Cornell University. His research is at the intersection of computer architecture, electronic design automation, and digital VLSI. He is a member of the IEEE. Contact him at cbatten@cornell.edu.

Michael B. Taylor received the AB degree in computer science from Dartmouth College in 1996, and the SM and PHD from the Massachusetts Institute of Technology in 1999 and 2007, respectively. He has been an associate professor in the Paul Allen School of Computer Science and the Department of Electrical and Computer Engineering at the University of Washington since 2017. Previously he was a visiting research scientist at Google and YouTube, and an associate professor with tenure in the Computer Science and Engineering Department at the University of California, San Diego.