SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads

SAM (LIKUN) XI*, YUAN YAO*, and KSHITIJ BHARDWAJ, Harvard University
PAUL WHATMOUGH, Harvard University and Arm ML Research
GU-YEON WEI and DAVID BROOKS, Harvard University

In recent years, there has been tremendous advances in hardware acceleration of deep neural networks. However, most of the research has focused on optimizing accelerator microarchitecture for higher performance and energy efficiency on a per-layer basis. We find that for overall single-batch inference latency, the accelerator may only make up 25–40%, with the rest spent on data movement and in the deep learning software framework. Thus far, it has been very difficult to study end-to-end DNN performance during early stage design (before RTL is available), because there are no existing DNN frameworks that support end-to-end simulation with easy custom hardware accelerator integration. To address this gap in research infrastructure, we present SMAUG, the first DNN framework that is purpose-built for simulation of end-to-end deep learning applications. SMAUG offers researchers a wide range of capabilities for evaluating DNN workloads, from diverse network topologies to easy accelerator modeling and SoC integration. To demonstrate the power and value of SMAUG, we present case studies that show how we can optimize overall performance and energy efficiency for up to 1.8×–5× speedup over a baseline system, without changing any part of the accelerator microarchitecture, as well as show how SMAUG can tune an SoC for a camera-powered deep learning pipeline.

CCS Concepts: • **Computing methodologies** → **Machine learning**; **Modeling and simulation**; • **Computer systems organization** → **Architectures**;

Additional Key Words and Phrases: Deep neural networks, hardware accelerators, architectural simulation

ACM Reference format:
Sam (Likun) Xi, Yuan Yao, Kshitij Bhardwaj, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2020. SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads. *ACM Trans. Archit. Code Optim.* 17, 4, Article 39 (November 2020), 26 pages.
https://doi.org/10.1145/3424669

*Both authors contributed equally to this research.

This work was supported in part by the U.S. Government, under the DARPA DSSoC program. This work was also supported in part by the Semiconductor Research Corporation, NSF grant # CNS-1718160, NSF grant #1533737 and Intel.

Authors' addresses: S. (Likun) Xi, Y. Yao, K. Bhardwaj, G.-Y. Wei, and D. Brooks, Harvard University, 29 Oxford Street, Maxwell Dworkin 308, Cambridge, MA 02138; emails: slxi1202@gmail.com, {yuanyao, kbhardwaj}@seas.harvard.edu, {gywei, dbrooks}@eecs.harvard.edu; P. Whatmough, Harvard University, 29 Oxford Street, Maxwell Dworkin 308, Cambridge, MA 02138 and Arm ML Research; email: paul.whatmough@arm.com.

1544-3566/2020/11-ART39
https://doi.org/10.1145/3424669

1 INTRODUCTION

The tremendous popularity of deep learning (DL) in recent years has been fueled by the increased capability of DL hardware and software systems. In particular, for both performance and energy efficiency, dedicated hardware accelerators for deep neural networks (DNNs) have received a phenomenal amount of interest [4, 10, 11, 15, 28, 29, 37]. Much of the focus on DNN accelerator design has been on optimizing core datapaths and dataflows to improve local reuse of data and reduce expensive data movement between the processing elements, local storage, and DRAM on a per-layer basis. However, at the end of the day, end-to-end performance is what truly matters, and additional overheads must be considered, such as data layout transformations that shuffle and reshape the data, the choice of accelerator interfacing with the SoC, which affects data movement efficiency, management of multiple independently programmed accelerators, and contention for shared system resources like memory bandwidth between different agents on the SoC.

As a motivating example of why overall performance is important, we profile end-to-end single-batch inference latency on a range of image classification DNNs. In this article, we define "end-to-end" as starting from the CPU receiving the inference request to when the result is computed. We break down the overall time spent on accelerator compute, data transfer to/from scratchpads, and CPU time spent in the software stack, on a system with one DNN accelerator connected over DMA. In this article, we define the term "accelerator" to refer to any independently programmable hardware block specialized for a few particular operations. Figure 1 shows that out of the entire execution time, only ~25% on average is spent waiting on accelerator compute, with the rest of the time taken up by data transfer (34%) and CPU processing (42%), performing tasks like data layout transformations, tiling, and more. This is particularly the case on a network like ResNet50 because of the many expensive tiling operations between each of the 50 layers. In some respects, this breakdown is not surprising, because the performance speedups offered by DNN accelerators can easily make software the primary bottleneck of overall latency. Nonetheless, this analysis shows several opportunities for optimization that would not have been revealed by a layer-by-layer analysis, which this article will explore in more depth. The impact of software stack time on overall performance has also been observed on industry-grade deep learning models written in TensorFlow [68] and Caffe2 [47].

Consequently, to holistically design DNN-centric SoCs, we must be able to study end-to-end behavior in simulation, as simulation is the usual methodology to evaluate early-stage/pre-RTL hardware designs. However, as shown in Table 1, there is no DNN framework available that supports fast, early-stage design exploration in simulation. Productivity-oriented frameworks like TensorFlow or PyTorch do not support simulation at all, and the ones that do support end-to-end simulation all require RTL/HLS for custom hardware accelerator integration, which is slow to write/generate and slow to simulate as well.

To address this gap in research infrastructure, we describe SMAUG: Simulating Machine Learning Applications Using gem5-Aladdin. It is the first architecture-simulation friendly deep learning framework that can run inside a user-level simulator. We chose compatibility with gem5-Aladdin [60], because it is built on the familiar gem5 simulator, supports flexible SoC, accelerator, and memory topologies, and also does not require RTL for design space exploration of accelerators, all of which greatly simplify the research and development process. SMAUG is designed to enable DNN researchers to rapidly evaluate different accelerator and SoC designs and perform hardware-software co-design, not to replace existing frameworks. SMAUG currently is targeted at DNN inference, but we plan to incorporate support for training as well.

SMAUG's headline features include a Python API for easy network configuration, support for a wide range of commonly used operators and network topologies, various hardware accelerator implementations of core kernels with easy plug-and-play of new custom operators, and a

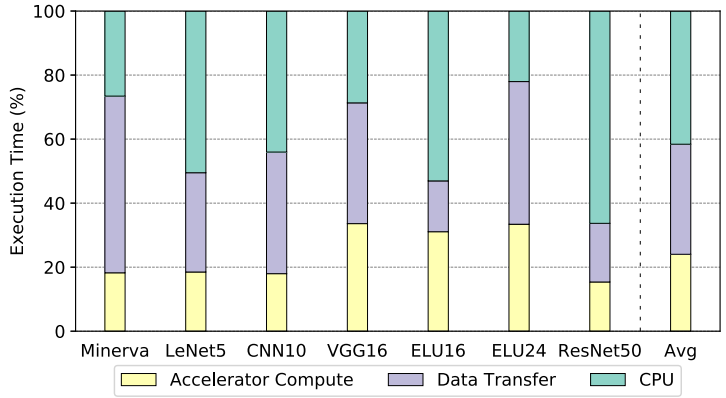


Fig. 1. Overall inference latency on a range of DNNs is bottlenecked by data transfer and CPU processing, because accelerators are already so well optimized. Evaluated on SMAUG (see Section 3 for baseline SoC.)

Table 1. Deep Learning Research Infrastructure

	End-to-End Evaluation	Simulation Support	Easy Custom Backend Integration	Simulation Speed
TensorFlow [1], PyTorch [31], Caffe/Caffe2 [27], MXNet [8]	✔	✗	✗	N/A
DNNWeaver [61], DNNBuilder [77], MAGNet [65]	✗	✔ (accelerator TLM/RTL simulation only)	✗ (requires RTL/HLS or detailed timing models)	Varies
TVM/VTA [9]	✔	✔ (behavioral simulation for template accelerator)	✗ (requires RTL/HLS)	Fast
SCALE-Sim [56], HSIM-DNN [62]	✗	✔ (accelerator analytical model)	✗ (backend specific)	Fast
Timeloop [45], MAESTRO [26]	✗	✗ (analytical models for evaluating dataflows/tiling)	✔ (high-level templates)	N/A
SMAUG	✔	✔✔ (flexible accelerator and SoC modeling)	✔ (no RTL required)	Fast

complete software stack that manages operator tiling, multi-accelerator and multi-thread scheduling, synchronization, and more. In addition, SMAUG solves several key problems of building architecture-simulation friendly deep learning frameworks:

(1) New accelerators are easy to implement in SMAUG's modular architecture. They can be implemented in just a few lines of code with Aladdin, or as a native gem5 object for more control over cycle-level timing.
(2) Running a complete forward pass through a DNN may require billions of operations, but for many core kernels, most of that work looks the same. SMAUG supports sampling of accelerator simulation with error as low as 1% for even the most aggressive sampling factors.
(3) Workarounds are provided for various simulator limitations to make up for the lack of complete OS feature support, such as a thread scheduler.

To illustrate the capabilities of SMAUG and the kinds of insights that only end-to-end DNN studies can provide, we present several case studies demonstrating how to improve overall performance of various DNNs by 45–79% (1.85×–5× speedup) over a baseline system without any changes to the accelerator microarchitecture itself:

(1) Using different SoC-accelerator interfaces to achieve tighter coupling between the CPU and accelerators for 17–55% overall speedup and up to 56% energy wins.
(2) Using multiple independent accelerators to exploit tile-level parallelism in DNNs for 24–62% overall speedup with eight accelerators over a single accelerator system.
(3) Using multithreading in the software stack to optimize data preparation time for up to 37% overall speedup.

Finally, we demonstrate how SMAUG can be integrated with a state-of-the-art camera pipeline, implemented in Halide, to model even more complex applications and identify opportunities for more efficient system design.

2 SMAUG FRAMEWORK

Figure 2 illustrates the overall architecture and execution flow of SMAUG. It is divided into three major components: a Python frontend for network configuration, a C++ runtime to manage the execution flow, and a backend consisting of a set of hardware-accelerated kernels. The accelerated kernels can be modeled either using Aladdin or as a native gem5 simulation object, depending on the user's desired level of flexibility and control. Ultimately, it compiles to a single binary that is run inside the simulator.

Users begin by building a network using a declarative Python API, complete with all input and weights data (which can optionally be taken from an existing trained model). They also specify in the configuration which set of accelerated kernels they want to use, the level of data quantization, and other metadata. The complete network specification is then converted into a dataflow graph and serialized (along with network parameters). This is a one-time operation for each network, so it is done as a separate step. The serialized model is loaded into the C++ runtime, and SMAUG begins a set of offline preprocessing steps. For example, certain read-only tensors (weights data) are pre-tiled (split into smaller contiguous tensors) during this preprocessing step to reduce time on the critical path. This preprocessing can also be fast forwarded in simulation to save time. Next, SMAUG invokes a tiling optimizer to compute the best available tiling shapes for each operation, so that the tiles utilize as much of the accelerator compute and memory resources as possible. Finally, SMAUG dispatches each tile of work to the appropriate processing elements, waits for them to finish, gathers all the results, and prepares for the next operation. If multiple independent PEs are available, then SMAUG can schedule all of them at once. Since the internal representation of the network is a graph, arbitrarily complex networks can be defined and scheduled; the architecture is not limited to linearly stacked layers. Multithreading support is available to parallelize CPU operations when possible and better utilize available shared resources like memory bandwidth.

As its name implies, SMAUG is compatible with the LLVM-based toolchains required by the Aladdin accelerator simulator and the gem5 APIs it exposes [60]. We have made extensive improvements to these toolchains to support compiling C++ binaries, tracing multi-threaded workloads, supporting sampling, and more. In the following sections, we will describe the frontend Python API, core runtime, hardware backend modeling and simulation workarounds in more detail.

2.1 Python API

Many deep learning frameworks use Python APIs to build models, and we wanted to follow in this same tradition to lessen the learning curve, rather than forcing users to manually write

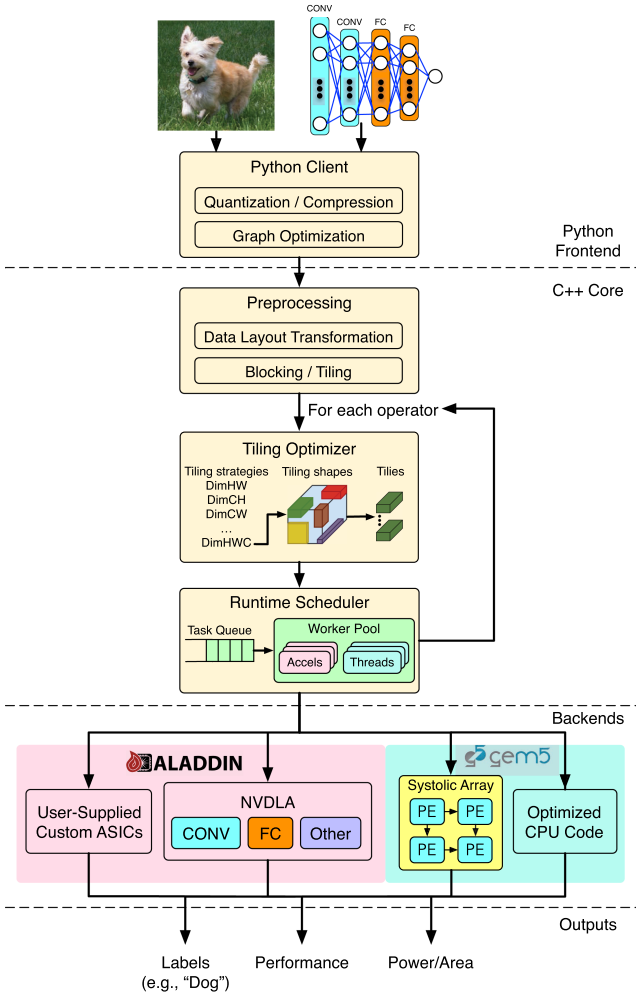


Fig. 2. Overview of SMAUG's execution flow.

configuration files or learn a new DSL. We wrote a new simple Python API instead of piggybacking on existing Python frontends of Tensorflow/PyTorch, because they have entirely different design objectives and we did not want to mislead users into thinking that they could simulate the full range of TF/PyTorch functionality. Figure 3 shows how a residual unit might be built.

This small example demonstrates the simplicity and familiarity of building networks in SMAUG. They are specified in a deferred execution style, and by using with-statement context managers, we can greatly reduce boilerplate without adding global state. All input tensors must be constructed inside the context before being used in an operator. Finally, the user can either supply random data or existing trained parameters as well as the data type (e.g., `float16` or `float32`). Certain optimizations like operator fusion (e.g., convolution + element-wise operators) are applied automatically by the framework. Finally, the user serializes the model specification and parameters; parameters are stored separately so that they can be easily swapped.

```
def create_residual_unit():
  with Graph(name="residual", backend="MyBackend") as g:
    # Tensor initialization.
    inputs = Tensor(np.random.rand(1, 8, 32, 32)))
    filter0 = Tensor(np.random.rand(64, 3, 3, 3)))
    filter1 = Tensor(np.random.rand(8, 64, 3, 3)))
    # If quantitization is desired:
    # filter0 = filter0.astype(np.float16)
    # Network topology:
    act = input_data("input", input_tensor)
    x = convolution("conv0", act, filter0,
          stride=[1, 1], padding="same", activation="relu")
    x = convolution("conv1", x, filter1,
          stride=[1, 1], padding="same")
    out = add("add", x, act, activation="relu") # residual
    return g

graph = create_residual_unit()
graph.write_graph() # Graph serialization.
```

Fig. 3. Constructing an operation in SMAUG uses a familiar Python style.

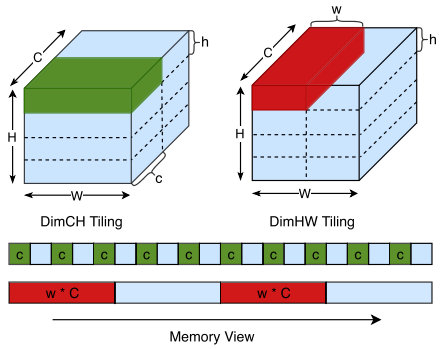
```
buffer input[IN_R][IN_C][IN_H];
buffer weights[NUM_PES][WGT_R][WGT_C][IN_H];
buffer output[NUM_PES][OUT_R][OUT_C];
parallel for (pe = 0 to NUM_PES)
  for (kr = 0 to WGT_R - 1)  // kernel row
    for (kc = 0 to WGT_C - 1)  // kernel col
      for (cb = 0 to IN_H/32 - 1) {  // channel block
        // Each PE has its own weight reg
        buffer wgt_reg[0:31] = weights[pe][kr][kc][cb:cb+31];.
        // Now iterate over the input rows and cols.
        for (r = 0 to OUT_R - 1)
          for (c = 0 to OUT_C - 1)
            parallel for (h = 0 to 31) {
              // 32-way spatial reduction in channel dimension.
              output[r][c][pe] += input[r+kr][c+kc][cb*32+h] * wgt_reg[h];
            }
      }
```

Fig. 4. Dataflow implemented by NVDLA. Apart from syntax, this is nearly the actual C code in SMAUG.

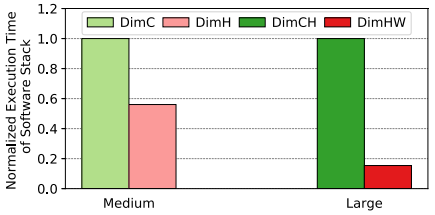
2.2 Tiling Optimizer

Due to the limited amount of local storage on an accelerator, individual layers of DNNs often have too many weights and/or inputs to run at once, thus requiring the operation to be tiled. Whenever tiling is required, redundant data movement is likely necessary, so identifying efficient tiling schedules (also called "loop nests") that maximize data reuse and minimize data movement between levels of the memory hierarchy is critical to achieving high performance. This has been studied extensively in the field; however, the general problem of finding the optimal solution is combinatorial in dimensionality and tiling factors, and beyond the scope of this work. In SMAUG, rather than trying to solve this general problem, we implement specialized tiling optimizers for every operator (convolution, matrix multiply, pooling, etc.). This section's discussion focuses on convolutions, but the takeaways apply to all supported operators.

As an example, consider the dataflow of the Nvidia Deep Learning Accelerator (NVDLA)'s convolution engine, which we use in our experiments (see Section 3 for details). As described in Figure 4, this convolution dataflow, reduces partial products in the channel dimension, so it benefits from a tiling schedule that maximizes the channel dimension of the input and weight tiles. But such a schedule is not suitable for an accelerator that computes one-dimensional (1D) or 2D convolutions, like the row-stationary dataflow [10]. By specializing the optimizer for a specific dataflow, we restrict the search space to a narrower set of possibilities that can be exhaustively



(a) A tensor can be tiled using different tiling strategies, which may produce very different layouts in memory.



(b) Different tiling strategies for a medium (1x16x16x128) and large NHWC tensor (1x64x64x512) exhibit very different transformation costs.

Fig. 5. Performance of different tiling strategies.

explored. The final schedule is one that maximizes both the utilization of the local scratchpads and the functional units.

The first major step of the tiling optimizer is to identify a *tiling strategy*, i.e., the best dimensions along which to tile the input and output tensors. This depends on three parameters:

(1) The input tensor shape (e.g., $N * H * W * C$).
(2) The minimum tile shape. This is the smallest tile shape that can do useful work and is determined by sensible heuristics. For example, if a convolution uses 3×3 kernels, then it does not make sense to have tiles smaller than 3×3. Similarly, if the data are stored in chunks of 8, then breaking these chunks up makes no sense either.
(3) The maximum tile size, determined by the capacity of the accelerator's local memory.

Given these parameters, the tiling strategy is determined by a preference order on tiling dimensions. As an example, the NVDLA preference order looks like this:

(1) If the entire tensor fits (less than the max tile size), then no tiling is needed.
(2) If the minimum batch size fits (using the min tile size's batch size), then tile by batches only.
(3) If we can partition the tensor both batchwise and channelwise to make a tile fit, then tile by batches *and* channels.
(4) If we can partition the tensor batch-size and row-wise to make a tile fit, then tile by batches and rows (but not channels).
(5) This procedure continues until we've considered all the possible tiling strategies.

The order of preferences is determined by two factors: the accelerator dataflow and the data layout of the tensor. The first factor is well understood; the second factor concerns how much work is required to shuffle and reshape the original input tensor into the required tile shapes and only arises when evaluating end-to-end performance.

As an example, Figure 5(a) shows how one NHWC tensor, when tiled in two different ways, produces two very different memcpy patterns. We describe a tiling strategy with the notation `DimXYZ`, where X, Y, and Z are the tiled dimensions. For this tensor, channels are the innermost dimension, so it is the most expensive to tile. To quantify this difference, we show in Figure 5(b)

how long it takes to tile two different tensors two different ways for a max tile size of 16,384 elements. The medium-sized tensor (1×16 × 16×128) can be tiled channelwise (1×16 × 16×64) or row-wise (1×8 × 16×128) but row-wise is 1.78× faster in software, because it only requires two large memcpys of 8×16×128 = 16 K contiguous elements, whereas for channelwise tiling requires 512 memcpys of 64 elements. This effect is even more pronounced on the larger layer, where we can use either DimCH (1×32 × 64×8) or DimHW (1×1 × 32×512). DimHW tiling is 6.5× faster to complete, because it only requires 128 memcpys of 16K elements to completely tile the input, compared to 262K memcpys of 8 elements. The effect of a different tiling strategy on the overall operation is harder to predict (but can be estimated with analytical models like Timeloop [45] or MAESTRO [26]). For element-wise operations, tiling strategy has next to no effect; for operations whose performance depends on exploiting data reuse, changing tiling shape may impact overall runtime. This is one of the new tradeoffs SMAUG enables researchers to explore.

The second major step of the tiling optimizer is to compute the best tile shapes, given the tiling strategy and max tile size. In SMAUG, this is done in two substeps. First, we generate a comprehensive set of compatible tile shapes for the tiling strategy and pick the one that maximizes local memory utilization. Note that not all combinations make sense. For example, if a candidate input tile has 16 channels, then the corresponding weight tile must also have at least 16 channels. We also want to avoid unnecessary data transfers: If the accelerator operates on vectors of 8 elements, then only produce tiles in multiples of 8. The chosen tile shape is the *basic tile shape*. The second substep is to iteratively partition each input tensor into tiles. Due to corner cases like zero-padding, halo regions, overlapping regions around each tile, non-unit strides, and more, not all input tiles will actually use the exact basic tile shape—they often vary by a few rows or columns. Non-uniform input tile shapes produce non-uniform output tiles, further complicating this procedure.

For a new accelerator, a new tiling strategy can be generated simply by changing the tiling strategy parameters and preference order (and also possibly by eliminating certain strategies that no longer make sense). For example, a pointwise convolution will prefer to tile by rows or columns *before* channels, since tiling by channels will require storing partial sums for further reduction later (more data movement and more arithmetic). The new accelerator can reuse the code to search for the best tiling shape, since we provide a library of tiling functions to facilitate writing new tiling optimizers. Aladdin supports modeling accelerators containing a variety of complex hardware constructs, since not all accelerators will be as simple as writing a new loop nest. But from SMAUG's perspective, all accelerators export a common interface: a dataflow, minimum tile shape, and max tile size, which are sufficient to determine how to tile a layer for it.

2.3 Runtime Scheduler

Once the tiling shapes have been generated for each operator's tensors, the scheduler prepares the tensor for computation by splitting it into the specified tile shapes. This has to wait until runtime, when the actual input data from the previous operator is ready. Then, the scheduler dispatches each tile to the appropriate compute element. If there are multiple compute elements and tile-level parallelism exists, then SMAUG can dispatch independent work to multiple compute elements at once. To help distribute work across multiple accelerators, SMAUG implements an accelerator worker pool and a command queue per accelerator. Tasks are pushed onto the command queue for the next available accelerator in the pool. Any operators that are not supported in the backend hardware accelerators are executed on the CPU instead. As the work completes, the scheduler gathers the tiled output data back into one contiguous tensor. Since the gem5-Aladdin API exposes an accelerator like a thread, managing multiple accelerators concurrently running is as straightforward as starting and joining on threads. SMAUG also supports dividing CPU work across multiple threads,

but since gem5's syscall-emulation mode does not have a thread scheduler, SMAUG implements a thread pool with round-robin scheduling of tasks from a work queue.

2.4 Backends

The backends run the convolutions, inner products, and so on, required by the model. In SMAUG, we provide a complete set of hardware accelerated kernels for all the included operators. These models can be written using Aladdin or as a native gem5 object, depending on the user's desired level of flexibility and control. We provide examples of both, most notably a convolution engine inspired by NVDLA, written with Aladdin, and a configurable systolic array, written as a native cycle-level gem5 object. For all other supported operators, we implement models in Aladdin. All accelerator models were validated against RTL implementations to be within 10% for total cycles.

The NVDLA-inspired convolution engine consists of eight PEs, each with a 32-way multiply-accumulate (MACC) array that operates on a different output feature map. The dataflow is described in Figure 4. Inputs and weights are 16-bit fixed point, while outputs are accumulated in 32-bit fixed point and reduced to 16-bit before being written to the scratchpad. In the emerging vernacular used to describe DNN dataflows, this dataflow is L0 weight-stationary (weights are reused every cycle at the register level within a MACC array), and L1 input/output stationary (for every weight, inputs are re-read and outputs are accumulated in-place in the SRAMs). It is backed by three SRAMs, one each for inputs, weights, and outputs. We only model the core datapath and dataflow of NVDLA, not other features like its convolution buffer.

The systolic array's dataflow is output stationary: Inputs stream through from the left, while weights stream from the top. There are three scratchpads, accessed from fetch and commit units, to supply the PEs with data. The dataflow was inspired by SCALE-Sim [56], but SCALE-Sim is primarily an analytical model that can generate SRAM and memory traces to feed to other tools like DRAMSim, whereas our model is entirely execution-driven and produces live memory traffic that affects (and is affected by) the rest of the system.

One of the key design features of SMAUG is how easy it is to implement a new HW accelerator model and integrate into the framework. The user simply needs to override a base `Operator` class to specify the operator parameters, implement an accelerator model, and invoke it from the `Operator::run` method. The user can reuse an existing tiling optimizer or build a new one if they desire, using a provided library of tiling functions we provide. Finally, a small additional amount of registration code link together the C++ and the Python sides. Implementing a new accelerator model is particularly easy if the user chooses to use Aladdin for modeling. For example, apart from syntax and variable declarations, Figure 4 is very similar to the code that models the convolution engine. In fact, merely 5% of the code is used to model all the hardware blocks with Aladdin. If users do choose to write cycle-level timing models using native gem5 APIs, then more code is needed (the systolic array model accounts for ~10% of the SMAUG codebase). The remaining 85% is devoted to computing tiling schedules, memory management, data movement, cache coherency, and task scheduling. Therefore, SMAUG eases the development of not only new hardware models but also studies of end-to-end system interactions, enabling researchers to spend their time on the topics that interest them the most.

2.5 Working with Simulator Limitations

We chose to use user-level simulators (like gem5 syscall-emulation mode) rather than full system simulators, because the latter requires new device drivers for their accelerators, which would likely deter many users. But to simulate end-to-end networks in user-level simulators, there are three constraints that must be addressed: reducing simulation time with sampling, handling incomplete

```
int reduction(int* a, int size, int sample) {
    // dmaLoad is placed outside the sampled loop, so that we don't
    // change the memory footprint of the application.
    dmaLoad(a, size * sizeof(int));
    int result = 0;
    setSamplingFactor("loop", (float)size / sample);
    loop:
    // Run only `sample` iterations of this loop; the result will be wrong,
    // but that's expected for sampling.
    for (int i = 0; i < sample; i++)
        result += a[i];
    return result;
}
```

Fig. 6. An example of specifying sampling factors on loops in Aladdin.

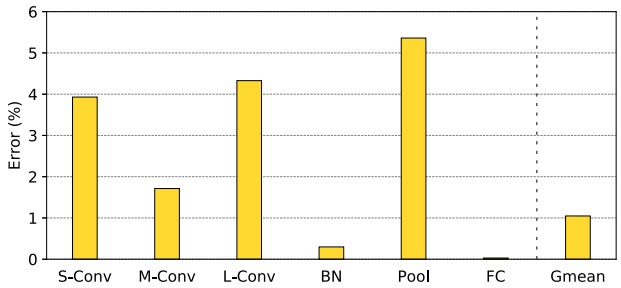


Fig. 7. Sampling performance validation. S-Conv uses 16 1×1×8 kernels; M-Conv uses 64 2×2×16 kernels; L-Conv uses 256 3×3×64 kernels.

system call emulation, and supporting multi-threading without a thread scheduler (typically implemented in the kernel).

2.5.1 Sampling Support. Modern DNNs are very deep and compute intensive, often requiring billions of operations, and because Aladdin is a trace-based simulator, it may be infeasible to store and simulate a complete forward pass. However, since DNN computation is so regular, a sampling approach works well. We built sampling support into Aladdin with a new API called `setSamplingFactor`, in which the user specifies how many iterations of a particular loop to trace and simulate. Figure 6 demonstrates the API at work. During Aladdin's graph optimization process, we build a loop tree that captures the hierarchy of loop iterations. When the simulation is over, Aladdin examines each node in the loop tree, unsamples the latency of simulated iterations, and propagates the sampled execution time up the tree. After every loop is unsampled, Aladdin produces a final overall cycles estimate. This API supports pipelined loops as well, although at least two loop iterations are required to determine the pipeline latency. As a result, we can simulate a forward pass of ResNet50 in just 5 hours. We validated our sampling technique for a range of operators and input shapes, all at the highest sampling factors (so that the sampled loops only run one or two iterations). Figure 7 shows that sampled execution has less than 6% error across different kernel types, with an average of just 1%. Finally, note that sampled simulation will obviously produce incorrect functional results, since not all the code is being executed, so this is only suitable for loops whose control flow is not data dependent.

2.5.2 System Call Emulation. User-space simulators often do not implement the full range of OS features that we come to take for granted. For example, the `mmap` syscall can map the contents of a file into memory (among other use cases), so it can be manipulated directly via loads and stores rather than through the IO subsystem, but in gem5 syscall-emulation mode, stores to mmapped memory are not synchronized to the backing file. SMAUG was written to work within

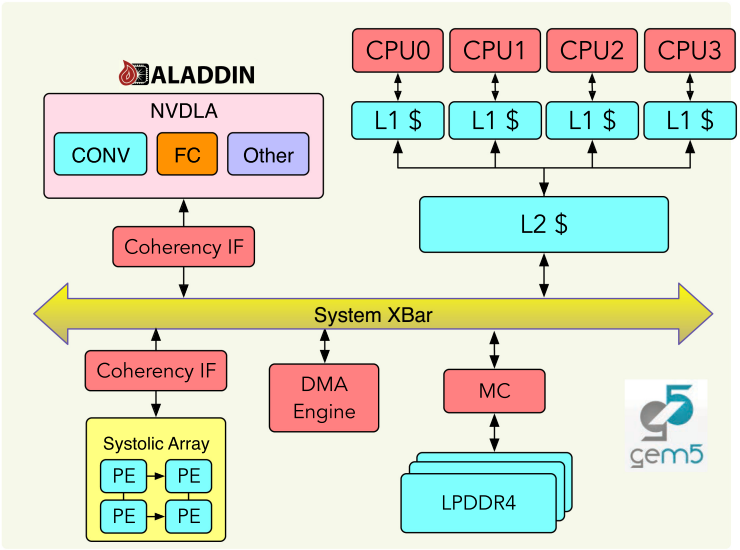


Fig. 8. The SoC platform used in this article's experiments.

all of these limitations; it compiles into a single C++ binary, never forks other processes, and interacts minimally with the OS, essentially only requiring the ability to access a filesystem, call `printf`, and start new threads that never exit. The drawback of syscall emulation is that we cannot fully capture effects like device offload and context switch latency, but compared to its benefits (faster simulation and user-friendliness of not having to write device drivers), we believe it is a worthwhile tradeoff.

2.5.3 Multithreading Support. It is common for user-level simulators to not implement preemptive thread schedulers, as thread scheduling is a kernel task. gem5 syscall emulation mode has limited support for multi-threading, with limitations on how thread contexts can be reused after a thread exits. To enable multi-threading in SMAUG, we implemented a custom thread pool and expose an API to dispatch work to it. Each task is executed to completion before yielding the CPU. Furthermore, to prevent idle threads from spinning endlessly in simulation and generating useless work that slows down the simulation, we use gem5 hooks to quiesce CPUs while they are waiting for work and wake them only when we assign them tasks.

3 METHODOLOGY

Now that we have described SMAUG, we will demonstrate how it can be used to provide insights into accelerated DNN performance that per-layer studies would not be able to show. First, we discuss our evaluation methodology.

3.1 Baseline System

Figure 8 shows the baseline SoC used in this article, with microarchitectural parameters listed in Table 2. In gem5-Aladdin, we use syscall-emulation mode with Ruby to model a MESI coherency protocol. The CPU communicates with the accelerator either via the `ioctl` system call or via shared memory. The baseline SoC transfers data over DMA and runs a single-threaded software stack. In Section 4 (also Figure 1), we run the convolutions and inner products on the NVDLA-inspired accelerator; in Section 5, we use the systolic array instead for diversity.

Table 2. SoC Microarchitectural Parameters

Component	Parameters
CPU Core	8 Out-of-order X86 cores @2.5GHz 8-μop issue width, 192-entry ROB
L1 Cache	64-KB i-cache & d-cache, four-way associative, 32-B cacheline, LRU, 2-cycle access latency
L2 Cache	2MB, 16-way, LRU, MESI coherence, 20-cycle access latency
DRAM	LP-DDR4, @16,00MHz, 4 GB, four channels, 25.6 GB/s
Accels	NVDLA conv engine and others, systolic array (8×8 PEs), 1GHz All scratchpads are 32 KB each

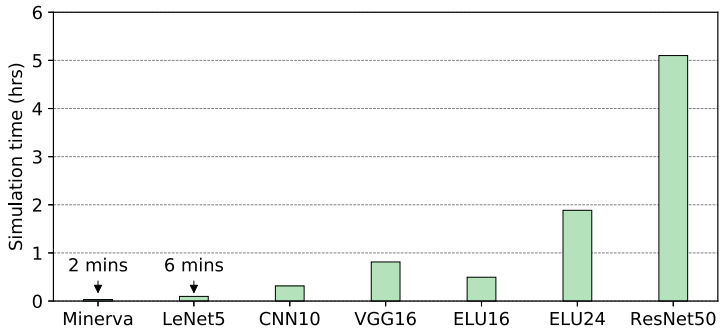


Fig. 9. With sampling, even large networks can be simulated in hours.

3.2 Workloads

With the flexible Python client and the complete SW/HW stack in SMAUG, we are able to evaluate a variety of DNN workloads. Here we investigate four image classification tasks: MNIST, CIFAR10, CIFAR100 and ImageNet. For the first three datasets, we select two different networks each. For ImageNet, we use ResNet50 [24] (included in the emerging MLPerf Inference Benchmark [54]). Table 3 summarizes the networks used. The goal was to cover a diverse set of network topologies that still map well to the accelerator's dataflow, which is optimized for convolution shapes deep in input/output feature maps. SMAUG implements a wide range of operators beyond these basic kernels (e.g., RNNs, LSTMs, attention layers, and more); we focus on image classification workloads in this article for brevity.

3.3 Simulation Time

Figure 9 shows the simulation time for running the workloads using the NVDLA backend in SMAUG on an Intel Xeon E5-2697 host (@2.6 GHz). For most of the networks, SMAUG simulations finish within 2 hours. The smaller MNIST workloads take less than 10 minutes, and with sampling, even the large ResNet50 network finishes in ~5 hours.

3.4 Power and Area Modeling

To obtain power and area estimates, we take a multi-pronged approach. We characterize various 16-bit functional units for power and area in a commercial 16nm FinFET process and plug them directly into Aladdin. To model accelerator local scratchpads, we characterize a variety of SRAM blocks using a commercial memory compiler in the same technology node. LLC power estimates are obtained from CACTI 7 [42], and DRAM power is modeled by DRAMPower [30], with timing

Table 3. Datasets and Networks Used in This Paper

Name	Dataset	Network Topology	Parameters	Accuracy	LOC
Minerva [53]	MNIST (28×28×1)	4 FC [784, 256, 256, 10].	665 KB	97%	20
LeNet5 [75]	MNIST (28×28×1)	5 layer CNN (3×3) 2 CONV [32, 32], POOL, FC [128, 10].	1.2 MB	98%	24
CNN10	CIFAR-10 (32×32×3)	10 layer CNN (3×3) 4 CONV [32, 32, 64, 64], 2 BN, 2 POOL, 2 FC [512, 10].	4.2 MB	85%	38
VGG16 [57]	CIFAR-10 (32×32×3)	16 layer CNN (3×3). 2 CONV [64, 128], POOL, 2 CONV [128, 128], POOL, 3 CONV [256, 256, 256], POOL, 3 CONV [512, 512, 512], POOL, 2 FC: [512, 10].	17.4 MB	90%	56
ELU16 [16]	CIFAR-100 (32×32×3)	16 layer CNN. 1 CONV [192], POOL, 2 CONV [192, 240], POOL, 2 CONV [240, 260], POOL, 2 CONV [260, 280], POOL, 2 CONV [280, 300], POOL, 2 CONV [300, 100]. Mostly 1×1 & 2×2 CONV.	3.3 MB	71.25%	66
ELU24 [16]	CIFAR-100 (32×32×3)	24 layer CNN. CONV [384], POOL, 4 CONV [384, 384, 640, 640], POOL, 4 CONV [640, 768, 768, 768], POOL, 3 CONV [768, 896, 896], POOL, 3 CONV [896, 1024, 1024], POOL, 4 CONV [1024, 1152, 1152, 100]. Mostly 1×1 & 2×2 CONV.	75 MB	77.72%	114
ResNet50 [24]	ImageNet (224×224×3)	50 layer CNN. 1 CONV [64], 3 stacks of 3 CONV [64, 64, 256], 4 stacks of 3 CONV [128, 128, 512], 6 stacks of 3 CONV [256, 256, 1024], 3 stacks of 3 CONV [512, 512, 2048], 1 FC [1000]. 1×1 & 3×3 CONV.	237 MB	76.46%	216

All parameters are stored as 16-bit fixed-point.

and power parameters taken from a commercial LP-DDR4 product datasheet [40]. We note that Accelergy [72] could also have been used for energy estimation, although it would not model area.

4 OPTIMIZING END-TO-END PERFORMANCE OF DNN WORKLOADS

SMAUG enables a wide range of architecture simulation tasks, from diverse DNN topologies to accelerator implementations, from the SoC integration of accelerators to evaluation of multi-accelerator systems. To illustrate the insights that SMAUG can bring to DL hardware architects, in this section we demonstrate several ways to improve end-to-end DNN performance on an SoC, all without changing the underlying accelerator themselves.

In these case studies, the baseline system uses one NVDLA accelerator with the DMA interface, running on a single-threaded software stack. Figure 1 has shown that not only the accelerator compute, but also data movement and CPU processing spent on "between-the-layer" work are crucial to end-to-end DNN performance. Therefore, in the rest of this section, we present three case studies that attack all these components of performance. First, we optimize data transfers

by using a one-way coherent interface between the accelerator and CPU LLC instead of DMA. Second, we explore multi-accelerator systems to exploit tile-level parallelism for greater compute and data-transfer throughput. Third, we optimize tiling transformations in software to reduce CPU processing time. As a whole, these optimizations speed up overall inference latency by 1.8×–5×.

4.1 Improving Data Transfer: Coherent Accelerator-SoC Interfaces

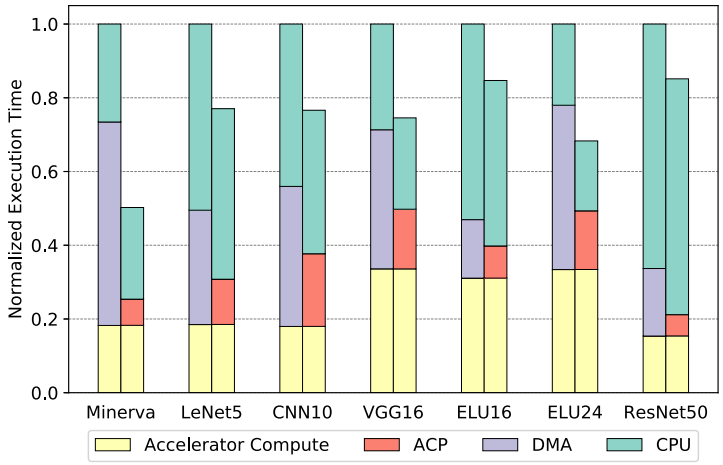
Fixed-function accelerators typically use private scratchpads for local storage and communicate with the memory system of the SoC through a DMA interface. DMA is the simplest approach to sharing data from a hardware point-of-view, but its implementation on many system architectures (e.g., ARM) requires software to be responsible for explicitly flushing and invalidating cache lines that the accelerator is going to read and/or write, resulting in both costly performance overheads and a challenging programming model where developers must manage complex coherency operations. This has driven researchers to investigate alternative interfaces, such as hardware-managed caches [2, 3, 33, 43, 44]. While cache coherency for programmable accelerators like GPUs have been extensively studied [5, 23, 49–51, 64], only in recent years have academia and industry started investigating use of caches for fixed-function accelerators and FPGAs. Full cache coherency represents the ideal programming model, but the hardware is more expensive and generally requires the accelerator to maintain a cache, which may not actually suit the accelerated kernel.

In this case study, we explore a recent interface design that occupies a middle ground between SW-managed and fully hardware-managed coherency. Here, the interface provides one-way coherent access from the accelerator to the host memory system. This interface takes the form of a special port, referred to as an accelerator coherency port (ACP), that issues coherent memory requests directly to the CPU's last level cache (LLC). The LLC handles all coherency traffic on the accelerator's behalf, enabling the accelerator to access coherent memory without adding more area and complexity [70]. To model such an interface, we augment a standard MESI cache coherence protocol using the Ruby modeling framework with a custom controller. This controller is connected to an accelerator's memory interface and generates requests to the LLC on behalf of the accelerator. Unlike a standard cache controller, it does not implement a cache and leaves ownership of the relevant cache lines with the LLC rather than the accelerator itself. Using Verilog simulation of an ARM Cortex A53 CPU, we measure ACP hit latency to be 20 cycles, which we set as the LLC latency.

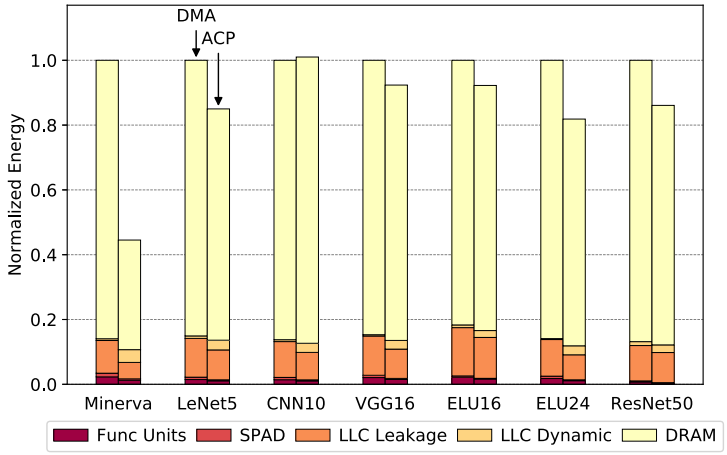
Figure 10 shows the performance and energy of the ACP interface, relative to the baseline DMA. By attaching the DNN accelerator over ACP, DNN performance improves by 17–55% and energy consumption drops by up to 56%. This is attributed to two effects. First, as a coherent interface, ACP eliminates the software coherency management overhead associated with using DMA for data transfers, which prior work [60] has shown to be a significant fraction of overall data transfer time. This accounts for the majority of the speedup on data transfers. Second, when using this coherent interface, many expensive DRAM accesses are converted into cheaper LLC hits, which reduces overall energy consumption by around 20% on average, as shown in Figure 10(b). While the actual improvements vary based on the size of the network and the tiling configurations, all these performance and energy wins were achieved just by changing the interface, not the accelerator. As the number of specialized blocks on SoCs increases over time, optimizing interfacing choices will become increasingly important and challenging. SMAUG enables researchers to study these challenging system-level architecture choices using full-stack deep learning workloads.

4.2 Improving Accelerator Compute: Multi-Accelerator Systems

DNN workloads have many different levels of parallelism, whether it is in the parallel arithmetic operations within a tile, across tiles within a single operation, or across independent operations



(a) Execution time.



(b) Energy.

Fig. 10. Performance and energy of the ACP interface, compared to DMA.

entirely (like residual branches in ResNet50). In this section, we explore scaling multi-accelerator systems to better exploit tile-level parallelism. Compared to a single monolithic block, a multi-accelerator system (e.g., spatial arrays or multi-chip modules) with independently programmable components can potentially scale to larger designs more easily and be more flexible for different workloads. We choose to exploit tile-level parallelism for two reasons: First, exploiting parallelism across arithmetic operations lies at the intersection of finding better tiling shapes for wider, more efficient accelerator datapaths, and, second, it is a more universal feature of DNNs compared to inter-operator parallelism (like residual branches).

When multiple accelerators are available, SMAUG places them into an pool of workers. Each accelerator is controlled directly by the runtime scheduler. When tiling for a layer is finished, the scheduler pushes tiles of work to the command queue of the assigned accelerator and tracks the progress of all tiles in flight. New tiles are pushed to the queue once their data dependencies

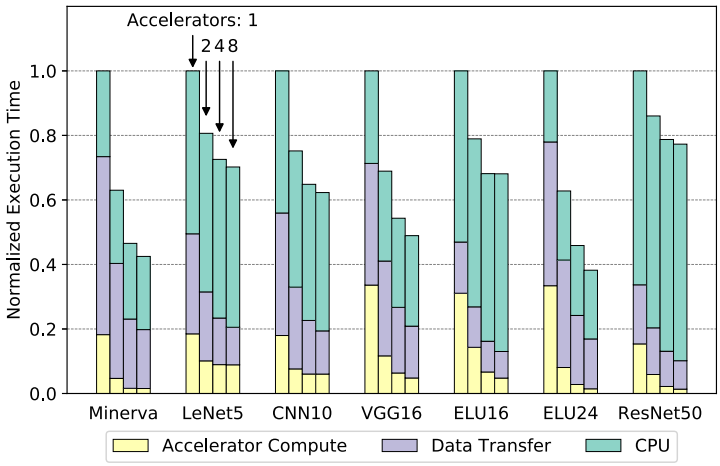
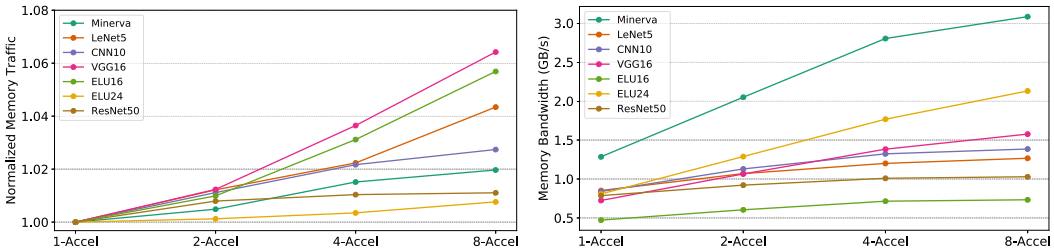


Fig. 11. Execution time of multiple-accelerator systems.



(a) Overall memory traffic for the entire workload. (b) Average end-to-end memory bandwidth utilization.

Fig. 12. Memory traffic and bandwidth usage of multiple-accelerator systems, averaged over the course of the entire workload.

are resolved; for example, some tiling configurations need all partial products along the channel dimension to be reduced before moving on to the next block of rows or columns. However, dividing work across multiple accelerators is not free, nor does it always improve performance. For example, if the dataflow is input-stationary and requires each of the N accelerators to share a weight tile, the weight data must now be broadcast to N destinations instead of just 1.

Figure 11 shows how performance of multi-accelerator systems scales with accelerator count. As expected, accelerator compute time speedup is consistent with the increase in available processing units. It continues until we saturate the available tile-level parallelism, which naturally occurs earlier for smaller networks than larger ones. Increasing accelerator count also means increasing total DRAM bytes transferred, because some data will need to be broadcast to all PEs, but as Figure 12(a) shows, this effect is small in the context of the entire workload, with at most a 6% increase in overall traffic. However, Figure 12(b) shows that multiple accelerators are also able to make better use of the available memory bandwidth. Overall, data transfer time drops by around 60% on average. Together, with eight accelerators in the system, end-to-end latency improves by between 20 and 60% over a single-accelerator system. This case study demonstrates how SMAUG can clearly illuminate the overall performance bottlenecks in DNN performance: By the time we reach eight accelerators, compute time is negligible compared to data transfer time and software stack time, and therefore those are the next components to optimize.

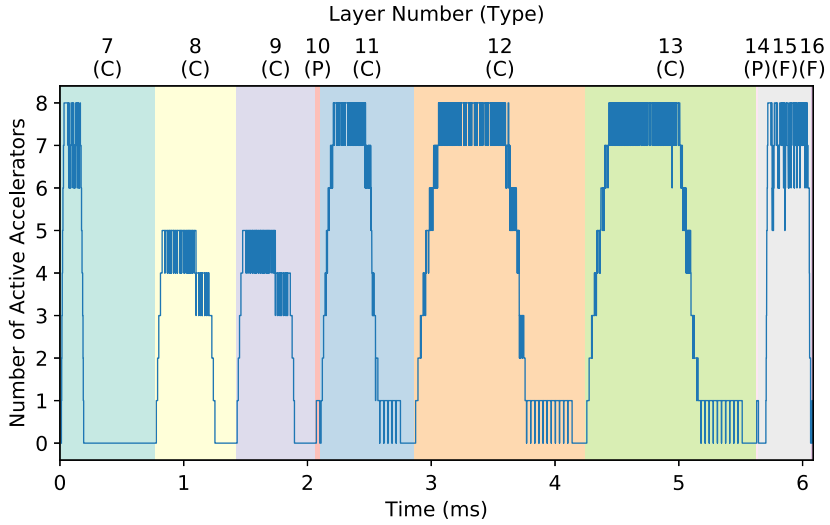


Fig. 13. Accelerator utilization of VGG16 with eight accelerators. C, P, and F stand for layer types of convolution, pooling, and fully connected, respectively.

When debugging bottlenecks in DNN inference, it is useful to inspect per-operation performance or performance between two particular operations. With SMAUG, we can generate an execution timeline of important events for users to visualize. For example, Figure 13 shows the accelerator utilization for the last 10 layers of VGG16, when the system has eight accelerators in total. These layers contain the largest six convolutional layers by number of parameters, two pooling layers (2×2), and two fully connected layers (512 and 10 neurons each). The timeline illustrates several opportunities for further optimization, which we summarize below.

Work balancing for higher accelerator utilization. The timeline shows that layers 8 and 9 are not fully utilizing all the accelerators in the system, because for this accelerator, the runtime scheduler only supports in-place reduction of partial products along the channel dimension, so all the tiles whose partial products must be reduced are put onto the same accelerator's command queue. Then for this layer shape, there are only five output tiles (i.e., independent streams of work), so only five accelerators are used. It is possible to evenly distribute work across all workers, which would require the runtime scheduler to support inter-accelerator reduction. Overall, the runtime scheduler in SMAUG does a good job in exploiting tile-level parallelism in the DNN, but as is the case with all software, there is always room to improve.

Accelerating inter-layer tiling operations. The timeline shows that on layer 7, the accelerator finishes computation very quickly, followed by a long period of CPU activity. This is the CPU performing "data finalization": gathering all the output tiles from the accelerators and rearranging them into a single tensor ("untiling" the tensor), because the next layer will likely need different input tile shapes. Ways to optimize this includes adjusting tiling shapes to maximize regions for contiguous `memcpy`s (see Figure 5(b)) and distributing the work across multiple CPUs to increase task-level parallelism and memory bandwidth utilization, which is the subject of the next section.

4.3 Improving Software Stack: Multithreaded Data Management

After all the effort spent optimizing core kernels like matrix-multiply and convolution, the performance bottleneck shifts to the cost of preparing data for use, and since this preparation is typically part of the software framework, the overhead is exaggerated in comparison to the accelerated

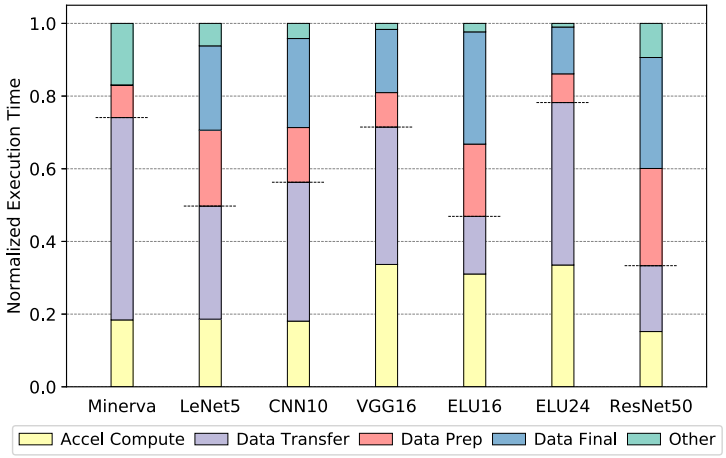


Fig. 14. Execution time breakdown of the baseline system. The dotted line separates hardware and software stack time.

kernels. This is not specific to SMAUG; on industry-grade recommendation models, data preparation, other framework native operations, and synchronization can take up anywhere from 10 to over 70% of inference latency [47, 68]. In this case study, we look at ways to reduce this overhead.

We break down the execution time of the software stack into three parts: data preparation, data finalization, and other software activities. Data preparation includes layout transformations, in which the dimensions of a tensor are either rearranged (e.g., NCHW to NHWC) or flattened, and tensor tiling, which copies non-contiguous logical regions of one tensor into contiguous smaller tensors that can then be directly transferred to the accelerator for computation. As a result, when accelerators finish their work, their output tensors are also tiled, which must now be "untiled" to obtain the final output tensor. We refer to this untiling operation as data finalization. Finally, other software activities include tasks like control flow management, memory management, various glue logic, and thread synchronization.

Figure 14 shows that on the baseline system, data preparation and finalization account for 85% of the software stack time, so there is ample room for improvement. As with the previous section, we attack this problem through tile-level parallelism. We use SMAUG's thread pool (see Section 2.5.3) to distribute data preparation and finalization tasks across multiple threads. Each thread is responsible for copying data to/from a set of tiles. The baseline system has already accounted for the cost of tiling transformations when determining the best available tiling strategy.

With multithreaded tiling, we can achieve up to 3×–4× speedup on data preparation/finalization with eight threads, as shown in Figure 15, resulting in an end-to-end latency reduction of up to 37%. This speedup is primarily due to an increase in memory bandwidth utilization when multiple threads are active. Figure 16 shows the memory bandwidth usage during the data preparation and gathering phases of the multithreaded software stack. On large networks like ResNet50, which have a lot of tiles, multiple threads increases bandwidth utilization by 2.7× and leads to a 2.8× speedup on data preparation and finalization tasks, while smaller networks like Minerva do not have enough tile-level parallelism for multi-threading to exploit.

4.4 Overall Combined Speedup

Figure 17 summarizes the combined effect of the three case studies on a single forward pass through all the networks. The SoC uses the ACP interface with eight accelerators and eight

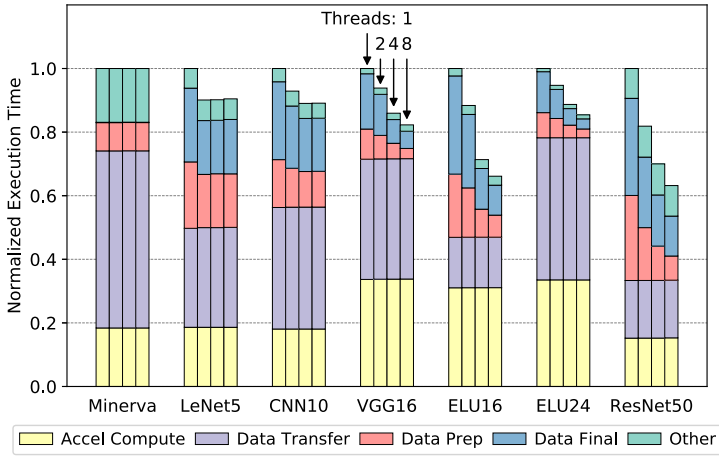


Fig. 15. Execution time of the system with a multithreaded software stack.

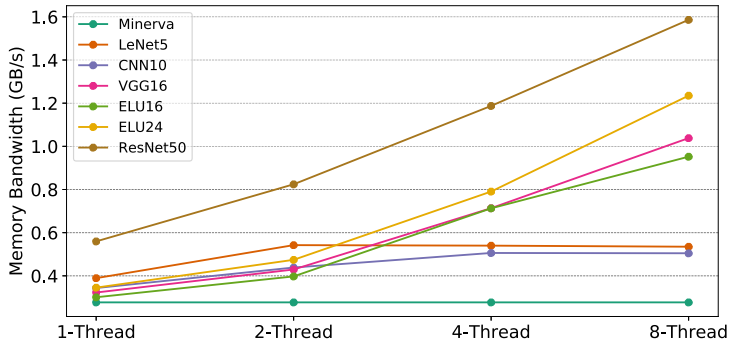


Fig. 16. Bandwidth utilization during the data preparation and gathering.

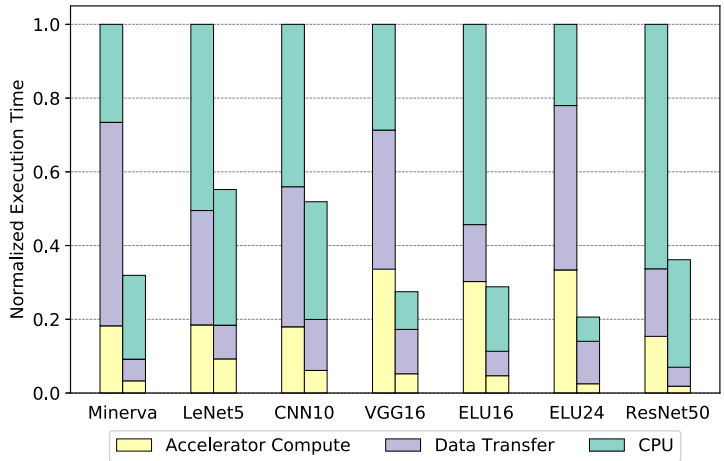


Fig. 17. By combining all the optimizations to each component of performance from the case studies, we can reduce overall latency by 42%–80%.

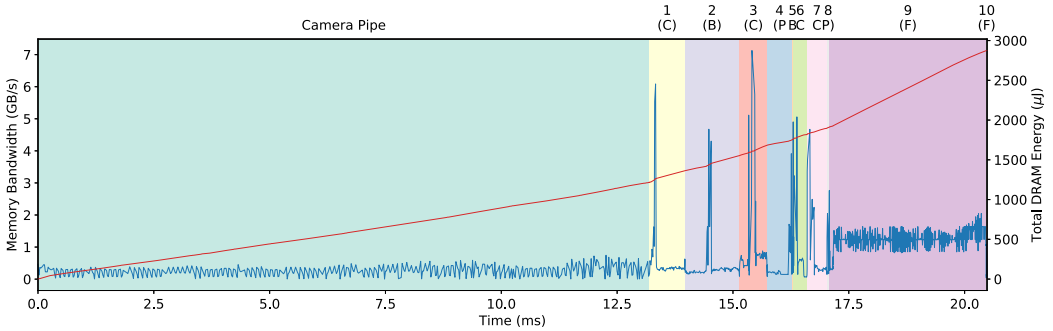


Fig. 18. Execution trace of the camera vision pipeline running one frame. The vision part runs CNN10. C, B, P, and F stand for layer types of convolution, batch normalization, pooling, and fully connected, respectively.

software threads. Overall latency drops by between 45% on LeNet5 to as much as 80% on ELU24 (1.8×–5× speedup), all without changing any part of the accelerator microarchitecture. This is a demonstration of the power of SMAUG applied to system-level performance optimization of DNN workloads.

5 OPTIMIZING A CAMERA-POWERED DEEP LEARNING PIPELINE

In recent years, it has become increasingly common to attach deep learning models at the end of other applications. One notable such application uses the camera pipeline with a DNN [22] to perform real-time tasks such as object classification, detection, segmentation, and labeling. In this study, we demonstrate how SMAUG can also model this kind of application and enable hardware-software co-design for better performance and energy efficiency.

The camera pipeline is a long series of spatial linear and non-linear filters and transforms to convert the image sensor's raw output into a realistic RGB representation. The sensor itself sits behind a Bayer color filter, so each photodiode only captures light from one of the primary colors (RGB). As a result, the output of the sensor is an array of pixel values, each representing the intensity of a single color. The process that estimates the original color of each pixel from this raw image is called demosaicing. The subsequent image processing then proceeds through many more processing steps, like white balance correction, color space conversion, chroma subsampling, and more. Finally, the image is compressed in a lossy format (e.g., JPEG), which preserves low frequency details that human eyes are sensitive to while removing the imperceptible high frequencies [18, 22].

To construct such a camera vision pipeline, we integrate the complete camera pipeline implementation shipped with Halide [52] into SMAUG and simulate it as a single process running on the CPU. The camera pipeline transforms raw data recorded by camera sensors into usable 720p images, including several stages: hot pixel suppression, deinterleaving, demosaicing, white balancing, and sharpening. Modern image sensors use multi-megapixel resolutions, but that resolution is often not necessary for DNNs; in this study, we feed 720p images through the camera pipeline, then downsample it to the size required by the DNN. For real-time applications, frame-time is a more representative metric of responsiveness than throughput, so assuming the application targets 30 FPS throughput, each frame must complete within 33 ms. The baseline system configuration we use is the same as the earlier case studies, except that to show the accelerator variety in SMAUG, we use the systolic array model (a cycle-level timing model written as a native gem5 object), configured as an 8×8 PE array instead of the NVDLA-inspired model.

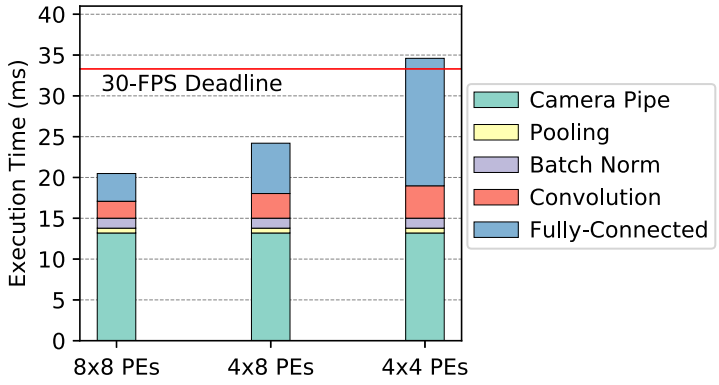


Fig. 19. Execution time of the camera vision pipeline with different systolic array PE configurations.

Figure 18 shows the execution timeline for the camera vision pipeline, using the CNN10 network. With SMAUG, we can produce a trace of memory bandwidth utilization and total memory energy consumed during the application. In this case, the overall pipeline takes 20.5 ms to finish (13.2 ms of camera pipeline and 7.3 ms of DNN), and memory energy consumption is well balanced between the CPU (43%) and accelerator (57%). The slack time (12.8 ms) before the frame deadline means that in energy or chip area constrained scenarios, we could afford to use an even smaller systolic array. As shown in Figure 19, reducing the PE array in half (4×8) increases the DNN latency to 11.0 ms, which still meets the frame-time limits. However, further decreasing the PEs to a 4×4 array results in an overall latency of 34.6 ms, violating the real-time constraint. Most of this extra latency comes from the final classifier layer.

6 RELATED WORK

Simulation frameworks. SoC-accelerator simulators usually require the user to implement the accelerators in RTL or using HLS tools [12, 14, 25, 36, 48, 63]. Centrifuge proposes a prototyping methodology that leverages HLS to generate accelerator SoCs and deploy them to FPGAs [25]. PARADE combines HLS with gem5 for full-system simulation [14]. GemDroid couples gem5 with the Android Emulator and integrates various hardware IP models to enable SoC-level simulation [12]. The heavy reliance on RTL implementation significantly increases the algorithm-to-solution time, even with HLS tools. In contrast, SMAUG builds on top of gem5-Aladdin, which uses a pre-RTL approach to accurately model the power, performance, and area of accelerator designs.

Table 1 lists recent deep learning research frameworks. Some are end-to-end systems, like TensorFlow [1] or TVM [9], but they either lack simulation support or require detailed pipeline models or RTL. Other tools focus on exploring dataflows and efficiently map DNN kernels to FPGAS or ASICs [45, 61, 65, 72, 76, 77]. These often implement a component library or templated designs for hardware optimization, but with a heavy focus on optimizing the accelerator, they cannot evaluate networks end-to-end, leaving a lot of design opportunities unexplored. While all of these tools have their place in the deep learning research infrastructure landscape, SMAUG is the only one that enables end-to-end *early-stage* design space exploration of the SoC as well as the accelerator.

Due to the regularity of DNNs, there are simulation tools that apply analytical models for DNN performance analysis [26, 45, 56, 63]. In general, analytical models have their place in the landscape of DNN tools since cycle-level simulation can be slow, but, because they do not actually simulate the workload, they would not be able to accurately model dynamic effects from multiple accelerators and multiple CPUs competing for shared resources (on-chip bandwidth, LLC capacity,

etc.), which is precisely the focus of our case studies. SMAUG also supports regular and irregular compute (sparsity, compression, etc.), since it uses cycle-level simulation to model performance and energy; this is often difficult to capture in analytical models.

DNN Accelerator Designs. There has been an incredible amount of interest in DNN hardware acceleration. Broadly speaking, the architecture community has focused on designing efficient dataflows to maximize local reuse of data and functional unit utilization [4, 10, 11, 15, 28, 34, 37, 39], explore the space of possible dataflows and mappings [26, 45, 74], exploit model sparsity and data quantization [17, 21, 29, 38, 46, 53, 71, 73, 78], map DNN accelerators to FPGAs [20, 66, 69], and explore alternative compute, memory, and packaging technologies [35, 58, 59, 67]. All of these works are highly relevant to this field. In particular, past work like Timeloop [45], which search the tiling space for efficient dataflows and mappings, could potentially be integrated into SMAUG's tiling optimizer by generating executable code from the high level mapping descriptions. However, these articles do not address end-to-end performance evaluation, CPU-accelerator interactions, or between-the-layer operations, like data layout transformations.

SoC-Accelerator Interfacing. Over the years, there have been a few publications investigating SoC-accelerator interfacing and interactions in a variety of contexts, such as CoRAMs [13], μLayer [32], and Google mobile system workloads [6]. A few recent works have considered interfacing between the SoC and accelerators [7, 19, 79]. A handful of other works have used the ARM accelerator coherency port for tighter coupling between CPU and accelerators, albeit not in the context of DNNs [41, 55].

7 CONCLUSION

This article demonstrates the critical importance of evaluating full-stack performance of a hardware accelerated computing task like neural network inference. Recent years have brought great advances in accelerator design and efficient DNN dataflows, but several important components of overall performance, like data transformation and movement cost and software framework overheads, have received far less attention, partly because of a lack of suitable research infrastructure. We developed SMAUG, a DNN framework that can be simulated in a cycle-level SoC simulator, and demonstrate how it can be used to optimize end-to-end performance on a wide range of DNNs to achieve between 1.8× and 5× speedup by optimizing SoC-accelerator interfaces, exploiting multi-accelerator systems, and optimizing the software stack. Since SMAUG provides architects with a straightforward approach to simulate complex full-stack workloads, we hope it will spur renewed interest in broader optimization of end-to-end performance in DNN hardware studies. SMAUG is available at www.github.com/harvard-acc/smaug.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[2] Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. 2018. Spandex: A flexible interface for efficient heterogeneous coherence. In *Proceedings of the International Symposium on Computer Architecture (ISCA'18)*.

[3] Johnathan Alsop, Matthew D. Sinclair, Srikant Bharadwaj, Alexandru Dutu, Anthony Gutierrez, Onur Kayiran, Michael LeBeane, Sooraj Puthoor, Xianwei Zhang, Tsung Tai Yeh, et al. 2019. Optimizing GPU cache policies for MI workloads. *arXiv:1910.00134*. Retrieved from https://arxiv.org/abs/1910.00134.

[4] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.

[5] AMD. 2014. *Compute Cores*. Technical Report. Retrieved from www.amd.com/computecores.

[6] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google workloads for consumer

devices: Mitigating data movement bottlenecks. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*.

[7] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. Exploiting temporal redundancy in live computer vision. In *Proceedings of the International Symposium on Computer Architecture (ISCA'18)*.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274*. Retrieved from https://arxiv.org/abs/1512.01274.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*.

[10] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *Proceedings of the International Solid-State Circuits Conference (ISSCC'16)*.

[11] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*.

[12] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2014. GemDroid: A framework to evaluate mobile platforms. *ACM SIGMETRICS Perf. Eval. Rev.* 42, 1 (2014), 355–366.

[13] Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An in-fabric memory architecture for FPGA-based computing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'11)*.

[14] Jason Cong, Zhenman Fang, Michael Gill, and Glenn Reinman. 2015. PARADE: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'15)*.

[15] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. 2017. CIRCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*.

[16] Djork-Arné Clevert, Thomas Underthiner, and Sepp Hochreiter. 2016. Fast and accurate deep network learning by exponential linear units (ELUs). In *Proceedings of the International Conference on Learning Representations (ICLR'16)*.

[17] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2019. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'19)*.

[18] Kayvon Fatahalian. 2011. A Camera Image Processing Pipeline. Retrieved from http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/lectures/16_camerapipeline1.pdf.

[19] Yu Feng, Paul N. Whatmough, and Yuhao Zhu. 2019. ASV: Accelerated stereo vision system. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*.

[20] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*.

[21] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark Horowitz, and William Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*.

[22] Patrick Hansen, Alexey Vilkin, Yury Khrustalev, James Imber, David Hanwell, Matthew Mattina, and Paul N. Whatmough. 2020. ISP4ML: Understanding the role of image signal processing in efficient deep learning vision systems. In *Proceedings of the 25th International Conference on Pattern Recognition (ICPR'20)*.

[23] Mark Harris. 2013. Unified Memory in CUDA 6. Retrieved from https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[25] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and John Wawrzynek. 2019. Centrifuge: Evaluating full-system HLS-generated heterogeneous-accelerator SoCs using FPGA-acceleration. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'19)*.

[26] Hyoukjun Kwon and Prasanth Chatarasi and Michael Pellauer and Angshuman Parashar and Vivek Sarkar and Tushar Krishna. 2019. Understanding reuse, performance, and hardware cost of DNN dataflows: A data-centric approach using MAESTRO. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*.

[27] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia (MULTIMEDIA'14)*.

[28] 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 International Symposium on Computer Architecture (ISCA'17)*.

[29] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.

[30] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. [n.d.]. DRAMPower: Open-Source DRAM Power and Energy Estimation Tool. Retrieved from http://drampower.info.

[31] Nikhil Ketkar. 2017. Introduction to pytorch. In *Deep Learning with Python*. Springer.

[32] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. uLayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the European Conference on Computer Systems (EuroSys'19)*.

[33] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. 2015. Fusion: Design tradeoffs in coherent cache hierarchies for accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*.

[34] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*.

[35] H. Li, M. Bhargav, P. N. Whatmough, and H. Philip Wong. 2019. On-chip memory technology design space explorations for mobile deep neural network accelerators. In *Proceedings of the 2019 56th ACM/IEEE Design Automation Conference (DAC'19)*. 1–6.

[36] Tingyuan Liang, Liang Feng, Sharad Sinha, and Wei Zhang. 2017. Paas: A system level simulator for heterogeneous computing architectures. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL'17)*. IEEE, 1–8.

[37] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*.

[38] Z. Liu, P. N. Whatmough, and M. Mattina. 2020. Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference. *IEEE Comput. Arch. Lett.* 19, 1 (2020), 34–37.

[39] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'17)*.

[40] Micron Technology 2014. *Mobile LPDDR4 SDRAM*. Micron Technology.

[41] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. SNNAP: Approximate computing on programmable socs via neural acceleration. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'15)*.

[42] Norman P. Jouppi and Andrew B. Kahng and Naveen Muralimanohar and Vaishav Srinivas. 2015. CACTI-IO: CACTI with off-chip power-area-timing models. *IEEE Trans. VLSI Syst.* 23, 7 (2015), 1254–1267.

[43] Lena E. Olson, Mark D. Hill, and David A. Wood. 2017. Crossing guard: Mediating host-accelerator coherence interactions. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

[44] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. 2015. Border control: Sandboxing accelerators. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*.

[45] 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 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*.

[46] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA'17)*.

[47] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill

Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. arxiv:cs.LG/1811.09886. Retrieved from https://arxiv.org/abs/cs.LG/1811.09886.

[48] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2017. Broadening the exploration of the accelerator design space in embedded scalable platforms. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'17)*. IEEE, 1–7.

[49] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs designing memory management units for CPU/GPUs with unified address spaces. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[50] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*.

[51] Jonathan Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'14)*.

[52] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the Programming Language Design and Implementation (PLDI'13)*. ACM.

[53] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose Miguel Hernandez-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*.

[54] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2019. MLPerf Inference Benchmark. arxiv:cs.LG/1911.02549. Retreived from https://arxiv.org/abs/cs.LG/1911.02549.

[55] Mohammadsadegh Sadri, Christian Weis, Norbert Wehn, and Luca Benini. 2013. Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ. In *Proceedings of the FPGAworld Conference (FPGAWorld'13)*.

[56] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A systematic methodology for characterizing scalability of DNN accelerators using SCALE-sim. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'20)*.

[57] Sergey Zagoruyko. 2015. Torch Blog: 92.45 on CIFAR10 in Torch. Retrieved from https://torch.ch/blog/07/30/cifar.html.

[58] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the International Symposium on Computer Architecture (ISCA'16)*.

[59] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Ross 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 IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. 14–27.

[60] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-designing accelerators and Soc interfaces using gem5-Aladdin. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.

[61] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. 2016. Dnnweaver: From high-level deep network models to fpga acceleration. In *Proceedings of the Workshop on Cognitive Architectures*.

[62] Mengshu Sun, Pu Zhao, Yanzhi Wang, Naehyuck Chang, and Xue Lin. 2019. HSIM-DNN: Hardware simulator for computation-, storage-and power-efficient deep neural networks. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'19)*. ACM, 81–86.

[63] Synopsys Inc. [n.d.]. Synopsys Platform Architect. Retrieved from https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html/.

[64] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. 2018. G-TSC: Timestamp based coherence for GPUs. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 403–415.

[65] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. MAGNet: A modular accelerator generator for neural networks. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'19)*.

[66] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. 2017. DLAU: A scalable deep learning accelerator unit on FPGA. *IEEE Trans. CAD Integr. Circ. Syst.* 36, 3 (2017).

[67] Peiqi Wang, Yu Ji, Chi Hong, Yongqiang Lyu, Dongsheng Wang, and Yuan Xie. 2018. SNrram: An efficient sparse neural network computation architecture based on resistive random-access memory. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC'18)*.

[68] Yu Emma Wang, Carole-Jean Wu, Xiaodong Wang, Kim Hazelwood, and David Brooks. 2019. Exploiting Parallelism Opportunities with Deep Learning Frameworks. arxiv:cs.LG/1908.04705. Retrieved from https://arxiv.org/abs/cs.LG/1908.04705.

[69] Xuechao Wei, Yun Liang, and Jason Cong. 2019. Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC'19)*.

[70] P. N. Whatmough, S. K. Lee, M. Donato, H. Hsueh, S. Xi, U. Gupta, L. Pentecost, G. G. Ko, D. Brooks, and G. Wei. 2019. A 16nm 25mm2 SoC with a 54.5x flexibility-efficiency range from dual-core arm Cortex-A53 to eFPGA and cache-coherent accelerators. In *Proceedings of the 2019 Symposium on VLSI Circuits*. C34–C35.

[71] Paul N. Whatmough, Chuteng Zhou, Patrick Hansen, Shreyas Kolala Venkataramanaiah, Jae sun Seo, and Matthew Mattina. 2019. FixyNN: Efficient hardware for mobile computer vision via transfer learning. In *Proceedings of the Conference on Systems and Machine Learning (SysML'19)*.

[72] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'19)*.

[73] Xiaowei Wang, Jiecao Yu, Charles Augustine, Ravi Iyer, and Reetuparna Das. 2019. Bit prudent in-cache acceleration of deep convolutional neural networks. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA'19)*.

[74] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 2020. Interstellar: Using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[75] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (1998), 2278–2324.

[76] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. ACM, 161–170.

[77] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'18)*.

[78] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. 2019. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*.

[79] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018. Euphrates: Algorithm-SoC co-design for low-power mobile continuous vision. In *Proceedings of the International Symposium on Computer Architecture (ISCA'18)*. 547–560.

Received May 2020; revised July 2020; accepted September 2020