2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00021

RaPiD: AI Accelerator for Ultra-low Precision Training and Inference

Swagath Venkataramani *, Vijayalakshmi Srinivasan*, Wei Wang*, Sanchari Sen*, Jintao Zhang*, Ankur Agrawal*, Monodeep Kar*, Shubham Jain*, Alberto Mannari†, Hoang Tran*, Yulong Li*, Eri Ogawa‡, Kazuaki Ishizaki‡, Hiroshi Inoue‡, Marcel Schaal*, Mauricio Serrano*, Jungwook Choi*, Xiao Sun*, Naigang Wang*, Chia-Yu Chen*, Allison Allain*, James Bonano¶, Nianzheng Cao*, Robert Casatuta‖, Matthew Cohen*, Bruce Fleischer*, Michael Guillorn*, Howard Haynie**, Jinwook Jung*, Mingu Kang*, Kyu-hyoun Kim*, Siyu Koswatta*, Saekyu Lee*, Martin Lutz*, Silvia Mueller§, Jinwook Oh*, Ashish Ranjan*, Zhibin Ren*, Scot Rider**, Kerstin Schelm§, Michael Scheuermann*, Joel Silberman*, Jie Yang*, Vidhi Zalani*, Xin Zhang*, Ching Zhou*, Matt Ziegler*, Vinay Shah††, Moriyoshi Ohara‡, Pong-Fei Lu*, Brian Curran**, Sunil Shukla*, Leland Chang*, and Kailash Gopalakrishnan*

*IBM Research, Yorktown Heights, NY, †IBM Research, Zurich, Switzerland, ‡IBM Research, Tokyo, Japan, §IBM, Boeblingen, Germany, ¶IBM, Austin, TX, ‖IBM, Hopewell Junction, NY, **IBM, Poughkeepsie, NY, ††IBM, Hursley, United Kingdom

***Abstract*—The growing prevalence and computational demands of Artificial Intelligence (AI) workloads has led to widespread use of hardware accelerators in their execution. Scaling the performance of AI accelerators across generations is pivotal to their success in commercial deployments. The intrinsic error-resilient nature of AI workloads present a unique opportunity for performance/energy improvement through *precision scaling*. Motivated by the recent algorithmic advances in precision scaling for inference and training, we designed RaPiD[1], a 4-core AI accelerator chip supporting a spectrum of precisions, namely, 16 and 8-bit floating-point and 4 and 2-bit fixed-point. The 36mm^2 RaPiD chip fabricated in 7nm EUV technology delivers a peak 3.5 TFLOPS/W in HFP8 mode and 16.5 TOPS/W in INT4 mode at nominal voltage. Using a performance model calibrated to within 1% of the measurement results, we evaluated DNN inference using 4-bit fixed-point representation for a 4-core 1 RaPiD chip system and DNN training using 8-bit floating point representation for a 768 TFLOPs AI system comprising 4 32-core RaPiD chips. Our results show INT4 inference for batch size of 1 achieves 3 - 13.5 (average 7) TOPS/W and FP8 training for a mini-batch of 512 achieves a sustained 102 - 588 (average 203) TFLOPS across a wide range of applications.**

***Keywords*-Hardware Acceleration, Deep Neural Networks, Reduced Precision**

I. Introduction

The past decade has witnessed a paradigm shift in the nature of workloads executed on computing platforms across the spectrum from mobile and IoT edge devices to the cloud and datacenters. Driven by the availability of massive amounts of data and advances in deep learning with Deep Neural Networks (DNNs), AI based applications and services have significantly grown in prevalence, even surpassing humans on several challenging AI tasks involving images, videos, text and natural language [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. However, the high accuracy of DNNs come at a high computational cost. For example, state-of-the-art image recognition DNNs (*e.g.* ResNet50) take ∼10 billion scalar operations to classify a single image. Furthermore, training DNN models require exa-FLOPs of compute and use massive training datasets, which are 100s of GB in size. Such high compute/storage/bandwidth requirements severely stress the capabilities of traditional computing platforms.

AI Accelerators. With the seeming slowdown of CMOS technology scaling and its associated benefits, meeting the computational demands of AI workloads and fueling future AI research on even more complex and robust models necessitates innovations across the hardware and software system stack. One approach that has been broadly adopted by the industry is *building specialized systems for AI with hardware accelerators*. AI workloads lend well to hardware acceleration as they are static dataflow graphs and computations in DNNs can be expressed using a small number (few tens) of primitives. This is evidenced by the many academic demonstrations of specialized accelerators for DNNs [21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32], and commercial AI cores (Google TPUs, NVIDIA Tensor Cores, Intel NNP, *etc.*) [33, 34, 35].

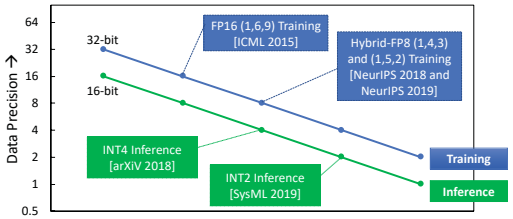


Figure 1: Precision scaling road map for training and inference

Precision Scaling. Scaling the performance of AI accelerators across generations is pivotal to their success in commercial deployments. Beyond traditional means of scaling performance at different levels of the compute stack *viz.* technology node, many-core/heterogeneous architectures, and others, AI workloads present a unique opportunity for performance/energy improvement through *precision scaling*. Advances in Approximate Computing in recent years have

[1]RaPiD expands as Reduced Precision Dataflow accelerator for AI

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00021

successfully demonstrated that, if done judiciously, the intrinsic error-resilient nature of AI workloads can be leveraged to reduce the bit width used for data representation during computation without loss in accuracy.

As shown in Figure 1, research efforts have consistently pushed down the precision requirements for both inference and training. Inference precision scaling driven by deployment in edge devices has gone to bit-widths as low as 2-4 bits for representing both weights and activations [36, 37, 38, 39, 40, 41, 42, 43]. Precision scaling for training is significantly more challenging due to the need to maintain fidelity of the gradients during the back-propagation step and a large dynamic range in the representation. Recently, 8-bit floating-point representations [44, 45] have been shown to be effective for DNN training.

Ultra-Low Precision Capable AI accelerators. Given these algorithmic advances, it is imperative that next generation of AI accelerators should be capable of ultra-low precision execution - beyond 16-bit floating point for training, and 8-bit integer for inference. In this work, we present RAPID, an accelerator architecture supporting a spectrum of precisions from 16-bit floating-point to 2-bit fixed-point. We have designed and fabricated a 4-core RAPID chip in 7nm EUV technology operating at 1.5 GHz.

At a high level, precision scaling has several advantages which makes it favorable to integrate within accelerator designs. First, it impacts all aspects of system design *i.e.,* improves compute efficiency, reduces memory footprint and data bandwidth requirement. Next, it preserves the regularity of compute which minimizes dataflow complexity and control overheads. However, it also introduces new challenges requiring careful architecture design. We highlight the key features of RAPID below.

1) **Mixed (Ultra-low) precision support.** The RAPID architecture supports 5 different precisions—16-bit floating point (FP16), two flavours of 8-bit floating point (FP8-fwd and FP8-bwd) with programmable bias together referred as Hybrid-FP8 (HFP8), 4-bit fixed-point (INT4) and 2-bit fixed-point (INT2). We select these precisions based on detailed algorithmic studies in the context of both training [44, 45] and inference [42, 46]. It is noteworthy that although we target ultra-low precision execution, it is critical to retain support for higher precisions. This is because while ultra-low precision can be applied to most computations, selected ones such as first and last layers, short-cut paths *etc.* require high precision to preserve accuracy [46]. Our 4-core 36mm^2 RAPID AI chip in 7nm EUV technology delivers 12/24/96 T(FL)OPS peak and achieves 1.8/3.5/16.5 T(FL)OPS/W in FP16/HFP8/INT4 precisions respectively.

2) **Scaling both TOPS and TOPS/W at low precision.** One of the key design tenets in RAPID is to improve both performance (TOPS) and energy efficiency (TOPS/W) at ultra-low precision catering to both real-time and battery-driven deployment scenarios. This implies scaling the number of compute engines commensurate with the power-saved at lower precisions while navigating the speeds *vs.* feeds trade-off at each precision.

3) **End-to-End application coverage.** While convolutions and GEMMs account for a significant fraction of DNN ops, they are embedded within a number of activation, pooling, normalization and data-shuffle operations. Executing these operations on the accelerator is vital as the data-transfer cost between the accelerator and the host could be costly. The Special Function Units (SFU) in RAPID includes both accurate and fast versions for a broad-range of such operations as well as permute engines for data-shuffling. It also uses 32-bit floating point (FP32) units for selected operations.

4) **Sparsity-aware Frequency Throttling.** To enhance energy-efficiency, the fused-multiply-and-accumulate (FMA) engines of RAPID are designed with zero-gating logic *i.e.,* a bypass path is triggered when one of the multiplicands is zero. It also includes a power management unit (controlled from software) that can rapidly throttle the effective clock frequency. In the context of inference, we leverage these features to boost the performance of sparse (or pruned) DNN models. Specifically, through offline analysis of the sparsity exhibited by each layer, we estimate the power saved through zero-gating and re-invest the power during the layer's execution by increasing the effective clock frequency to benefit performance.

5) **Multi-core Scaling.** RAPID contains a Memory Neighbor Interface (MNI) that enables core-to-core and core-to-memory communication and synchronization. Our communication protocol contains simple primitives that allow for concurrent data-transfers between overlapping multi-cast producer-consumer core groups, allowing software to effectively utilize the available bandwidth.

In summary, the RAPID architecture enables both training and inference at ultra-low precision. The design fosters scalability to multiple cores and provides the necessary functional converge to execute end-to-end AI workloads. In this paper, we evaluate the 4-core RAPID chip model for inference with a primary focus on 4-bit fixed point. We also evaluate a distributed 4-chip system using 32-core RAPID chip model for training using 8-bit floating point. To the best of our knowledge, RAPID is the first effort to support a mixed-precision architecture capable of 8-bit training and 4-bit inference.

The rest of the paper is organized as follows. Section II discusses the systolic dataflow architecture, akin to several recent prior work, which is used as a baseline core architecture and summarizes the advances in algorithmic approximations that serve as the foundation for enabling ultra-low precision support in RAPID. Section III presents the key architecture and microarchitecture innovations in the core to support for ultra-low precision and Section IV presents the 7nm

RAPID chip with 4 cores along with the programming model and discusses proposed systems for training and inference derived using scaled RAPID chip(s). Section V describes the effectiveness of RAPID for inference and training especially at ultra-low precision on a set of popular DNN benchmarks. Section VI discusses related efforts, and finally Section VII concludes the paper with future work.

II. Background

AI workloads are static dataflow graphs and the computations in DNNs can be expressed using a small number (few tens) of primitives. This has led to many academic demonstrations and commercial AI cores exploiting systolic dataflow architectures [23, 25, 26, 30, 47, 48, 49]. We use such a systolic dataflow architecture as a baseline and enhance the architecture to support ultra-low precision for training and inference.

A. Baseline: Systolic Dataflow Architecture

Figure 2 shows the building block of the baseline systolic dataflow architecture. The main computation unit comprises of a 8x8 2D-systolic array of Processing Elements (PEs) supporting 16-bit floating-point (FP16) computations to execute convolution and matrix multiplication operations in DNNs. and a 1D-array of Special Function Units (SFUs) supporting both 16 and 32-bit floating-point computations (FP16 and FP32) to perform activation functions, pooling, gradient reduction and normalization operations, which may require higher bit precisions.

Each PE contains a 8-way SIMD multiply-and-accumulate (MAC) unit whose operands are received from the PE's North/West neighbors or from its Local Register File (LRF). Similarly, the output of the MAC is sent to either the South neighbor PE or written back to the LRF. Since the typical dataflows used did not require diagonal flow of operands, the PEs of a given row execute the same instruction sequence in a systolic fashion. Each SFP also contains 8-way SIMD MACs in higher-precision which operate as a vector unit.

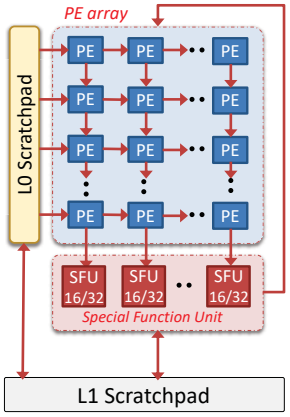


Figure 2: Baseline: Systolic Dataflow Architecture

A 2-tiered memory hierarchy of scratchpads feeds data to the PE array and SFUs. The L0 scratchpad is used to feed data along the rows (X direction). The $L1$ scratchpad memory is connected to the $L0$ memories and columns (Y direction) of the SFU/PE array on one side and interfaces with the external memory on the other.

To provide maximum flexibility, the baseline architecture is fully decentralized by *decoupling compute and dataflow through the different components into multiple separate threads of execution*. Similar to the access-execute paradigm, programmable units are located at the end points of each (or set of) link(s) in the architecture to have fine-grained control over the sequence of data through the link(s). For example, to orchestrate dataflow between L1 and L0, a programmable unit located near the L1 controls the address sequence read from the L1 and pushes the data on the link. Upon receiving the data, a programmable unit near L0 determines the location where it needs to be stored in L0. The SFUs also run their individual programs. They can read/write data operands from/to any of their incoming/outgoing links and their local register file.

Execution of a DNN operation is therefore orchestrated through multiple programs which can be broadly classified into: (i) *Data sequencing programs* that load/store data from the scratchpad memories and feed them in sequence to PE/SFUs, and (ii) *Data processing programs* that define the set of computations executed on PE/SFUs on the incoming data elements. To ensure correct functionality (*e.g.*, producer-consumer dependency), the architecture uses token-based hardware support for synchronization between selected programmable units. For example, consider when data is moved from L1 to L0 and then subsequently streamed to the PE array, the program writing data into L0 and the one reading it synchronize periodically to ensure writes precede reads.

B. Scaling Training beyond FP16

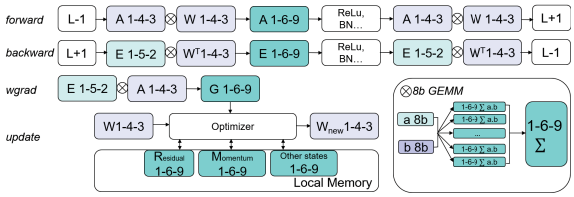


Figure 3: Support for mixed FP8 precisions for training [45]

Reduced precision DNN training is significantly more challenging due to the need to maintain fidelity of the gradients during the back-propagation step. Recently, algorithmic approximations [44, 45] resulted in a new Hybrid-FP8 (HFP8) data format, shown in Figure 3 for training DNNs. The HFP8 format involves using two different FP8 (sign, exponent, mantissa) representations - one representation with lower dynamic range (1,4,3) for activations and weights, and the other with higher dynamic range (1,5,2) for errors. Both operands in the forward pass uses FP8(1,4,3), whereas the backward pass and gradient computations have one operand in FP8(1,4,3) and the other in FP8(1,5,2) representation. In addition to using 2 different exponent-mantissa bit combinations in forward *vs.* backward passes, HFP8 requires the exponent bias to be configurable. This enables different DNN layers to take

different dynamic ranges, despite using the same number of exponent bits.

HFP8-based training [44, 45] has been shown to achieve model accuracy equivalent to FP32 representation training of deep learning models across a whole spectrum of applications including image classification, object detection, machine translation, and speech.

C. Scaling Inference beyond INT8

Precision scaling for inference has been demonstrated successfully for *4-bit* (INT4) and *2-bit* (INT2) fixed-point representations. Recently, two quantization techniques *viz.* PArameterized Clipping acTivation (PACT) [42] for activations and Statistics-aware Weight Binning (SaWB) for weights [46] have demonstrated *4-bit* (INT4) inference with negligible loss in accuracy and *2-bit* (INT2) inference with minimal accuracy loss (≈2%). PACT introduces a new activation function derived from ReLU that clips the output beyond a value thereby reducing its range. The key insight is that the clipping value is not statically fixed, but rather learnt during model training independently for each layer of the DNN. SaWB quantizes weights by using the first and second moment of the weight while retaining the shape of the weight distribution. Both PACT and SaWB have little/no impact on the model training time.

III. Core Architecture For Ultra-low Precision

One of the key features of precision scaling is that it preserves the regularity of the compute as all elements of a given tensor are scaled equally to the same precision. In addition, precision scaling saves energy both in the execution units and in the memory and interconnect subsystems as the capacity requirement of data-structures and the amount of data transferred between components are also reduced. Hence the baseline systolic dataflow architecture shown in Figure 2 has the organization well-suited for these precision-scaled AI workloads. But to meet the computational and bandwidth requirements for ultra-low precision training and inference the baseline architecture has to be enhanced to:

- Scale the overall peak TOPs to be commensurate with the scaled precision
- Balance the area and power of the PE array to effectively utilize the increased peak TOPs.
- Meet the bandwidth constraints for data flowing into the PE array by choosing data-flows to re-use the operands effectively across SIMD and rows/columns of the PE array.
- Continue to produce outputs in 16-bit format from the PE array so that the auxiliary operations can be performed in higher precision to maintain accuracy of classification.
- Balance the computational units to match the distribution of high-precision activations in SFU array and ultra-low precision convolutions and matrix operations in the PE array.

We now present the key architecture and microarchitecture innovations added to the different components of the fundamental building block of the baseline architecture to overcome the challenges and realize the above goals.

A. MPE Array: Mixed-Precision PE Array

As we continue to scale the precision of the computations, one of the key challenges is to scale the overall peak TOPs to be commensurate with the scaled precision. We now discuss the architecture and microarchitecture enhancements in the mixed-precision PE array that enabled scaling the peak TOPs with precision scaling while overcoming the challenges of bandwidth, area, and power.

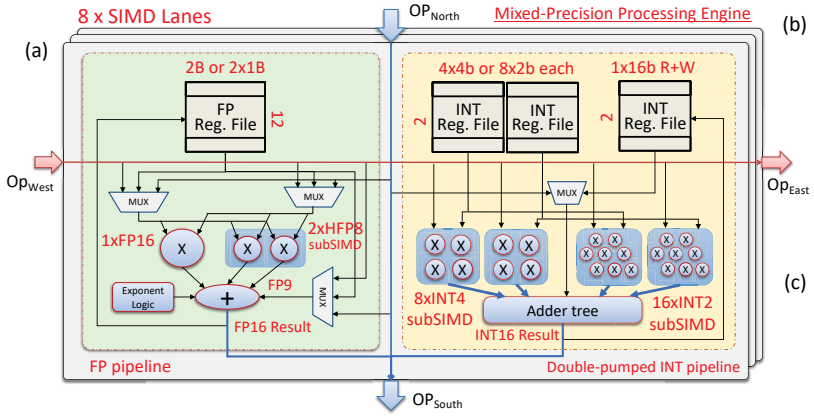
1) Supporting both INT and FP pipelines To support both training and inference in ultra-low precision, the MPE array has to have comprehensive support for mixed precision execution, which includes different number formats: *viz.* FP16, Hybrid-FP8, INT4 and INT2 as discussed in Section II. Figure 4(a) shows the block diagram of an MPE. Supporting both floating-point and fixed-point operations in the compute engines increases area and power overhead while also creating potential mismatches in pipeline depth and execution latency. Separation of the integer and floating point pipelines solves the architectural complexity of handling multiple precisions while providing circuit implementation opportunities to aggressively improve power efficiency. As in the baseline architecture, each MPE has 8 FPUs and 8 FXUS supporting FP16/HFP8 and INT4/INT2 formats, respectively.

2) HFP8 Training The MPE's FPU pipeline supports both FP16 and HFP8 using the same 128-bit datapath for the 8-way SIMD FPU. As shown in Figure 4(a), each MPE's FPU receives input operands North/West neighbors or from its Local Register File (LRF), and the input operands are also propagated to the East links, and the outputs are propagated to the South neighbor.

We enhanced the FPU with 2 key innovations for HFP8 support to enable supporting different representations for the inputs operands and achieving 2*X* the peak TOPs relative to FP16.

On-the-Fly Conversion to custom FP8 representation: As shown in Figure 3, FP8 training requires matrix multiplications and convolutions in the backward path of training to use tensors of different FP8 formats as inputs. However, this increases the hardware complexity for the floating-point units both in terms of area and power. We enhanced the FPU to use a custom (sign, exponent, mantissa) format of (1,5,3) with the input operands in both (1,5,2) and (1,4,3) formats converted on-the-fly to (1,5,3) format [50].

As a result, for HFP8 training, the weights and activations are stored in memory and scratchpads in either (1,4,3) (with bias), or (1,5,2) formats, and converted to 9-bits (1,5,3) formats on either the horizontal bus for data coming from the L0 scratch-pad, or the FIFO interface for data from the L1 scratch-pad via the north link.



(b)

	Format	Bias	Instruction
FP16	1/6/9	31	FMA: R = A.B + C (all FP16)
FP8-fwd	1/4/3	0-15	FMMA: $R = A_1.B_1 + A_2.B_2 + C$ A_1, A_2, B_1, B_2: FP8 R,C: FP16
FP8-bwd	1/5/2	15	
FP9	1/5/3	15	
INT4	2's complement		$R = \sum_8 A_i . Bi + C$ (A,B: 8x INT4; R,C: INT16)
INT2	Wei.: [-4, -1, 1, 4] Act.: [0,1,2,3]		$R = \sum_{16} A_i . Bi + C$ (A,B: 16x INT2; R,C: INT16)

(c)

FP & INT pipelines	TOP/W (normalized)			Area
	FP16	FP8	INT4	
Merged	1.0x	2.2x	5.6x	1.0x
Separate	1.1x	2.3x	10.6x	1.16x

Figure 4: Block diagram of Mixed-Precision Processing Element (MPE) and instruction formats

The ISA of the MPEs support input tensors for multiply-accumulate instructions (FMMA) to have different FP8 formats. The MPE program uses the desired format for the input operands based on the data-flow chosen. However, within a program, the precision of the operands remains fixed, and is set in registers to allow the hardware to determine data-gating width for the operands. Another key feature of HFP8 datapath is that its exponent bias is programmable. Based on the dynamic range of the computations under execution, the MPE is configured with the appropriate exponent bias.

sub-SIMD partition: One of the key enablers to scale the peak TOPs in HFP8 mode is the fine-grain partition the SIMD units (*sub-SIMD*) within the FPU to realize 2*X* performance at the same power as in FP16 while using the same interface bus width. As shown in Figure 4(a), in HFP8 mode, the multiply-accumulate instruction (FMMA) of the SIMD MPE realizes 2 multiplications and 2 additions. Even with the additional complexity, the logic depth of the HFP8 multiplicand path remains comparable to FP16 due to the 4-bit multipliers.

With the ISA extensions shown in Figure 4(b) and the microarchitecture enhancements described above, the MPE supports HFP8 and FP16 in the FPU pipeline.

To maintain end-to-end accuracy, the auxiliary operations require higher precision both in FP16 and HFP8 training. Since both FP16 and HFP8 mode produces FP16 results, the FPU compute paths of FP16 and HFP8 merge at the adder as shown in Figure 4(a).

Finally, HFP8 training also uses chunk-based accumulation [51] to accumulate partial sums in a hierarchical fashion in order to preserve the fidelity of the intermediate sums. The SFUs are used to realize chunk-based accumulation of the partial sums(FP16/INT16) produced by the MPEs.

3) INT4/INT2 inference As mentioned above, we augmented the MPE to have separate FPU and FXU pipelines. In addition, since the INT4/INT2 engines target only DNN inference, more circuit-level optimizations became feasible to reduce area and power.

Double pumping INT4 and INT2 pipelines Our area and power analysis of the de-coupled FPU and FXU units revealed opportunities to double the INT4 and INT2 engines within the FXU. As summarized in Figure 4(c) the addition of a separate INT pipeline incurs ∼16% area overhead, but the power of the INT4 pipeline was 0.3X that of the FP16 pipeline enabling *doubling* the INT4 and INT2 compute engines in the MPE. Therefore, each FXU has 8 INT4 (16 INT2) multiply-accumulate engines.

Operand Reuse: Sub-SIMD + Across Columns: As each of the 8-way SIMD unit completes 8 INT4 (and 16 INT2) multiply-accummulate operations in a cycle, we doubled the datapath width of the SIMD MAC unit to be 256 bits and enhanced the architecture to allow accessing 2 registers (256 bits) in the MPEs with a single MAC instruction. Doing so, also mitigates the energy cost of accumulation especially as the cost of multipliers reduce with precision scaling.

As shown in Figure 4(a) the 8-way SIMD FXU supports 4 and 2 bit integer MAC operations producing 16-bit integer results matching the 128 bit datapath between MPEs. Efficient data-flow mapping balances the L0 bandwidth constraints and the operand re-use within the FXU. For example, with a modified weight-stationery data-flow the 8 INT4 input operand B (32 bits) flowing from West link are re-used across each of the 8-way sub-SIMD, and is also propagated across the columns of the MPE to be reused overall in 64 multiply-add operations.

4) Convolution and GEMM Dataflows in the MPE array Figure 5 shows an example dataflow and simplified pseudocode that we use to map convolutions and matrix multiplications in the MPE array. Convolution layers can be expressed using 7 dimensions: input and output channels (Ci and Co), feature size ($H \times W$), kernel size ($Ki \times Kj$) and minibatch (N). Dataflow design involves: (i) defining dimensions that are spatially mapped along rows, columns,

SIMD lanes and local register file (LRF) of the MPE array, and subsequently (ii) determining which data-structures are streamed along rows, columns and held stationary in the LRF and the sequence in which the elements are accessed.

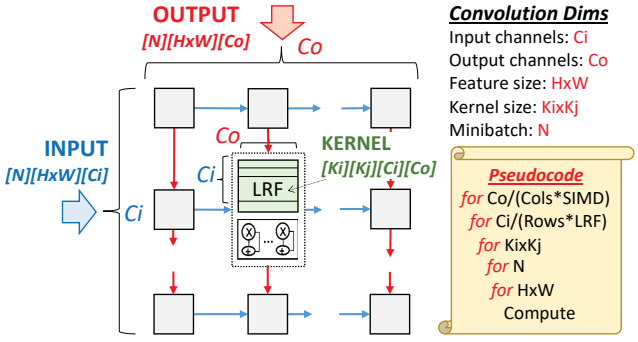


Figure 5: Convolution dataflow

We have utilized a novel weight-stationary dataflow, which was derived based on the following constraints to narrow the dataflow choices: (i) achieve high utilization all the way down to batch size of 1, which eliminated batch size as a spatial dimension, (ii) avoid cross-row or cross-column communication, which implies $H \times W$ cannot be chosen to map spatially, and (iii) minimize residue effects due to strip-mining when workload dimensions are not a multiple of hardware dimensions, which meant $Ki \times Kj$ is not a good choice as they are typically small prime numbers. Therefore, we selected input (Ci) and output (Co) channels as the spatial dimensions. Further, to avoid cross-SIMD reduction in hardware, we map Co along columns and SIMD, and Ci along rows and the LRF. In a systolic dataflow, the data operand fed along a row needs to be reused by all columns and vice versa. This implies, the dimension spatially mapped along columns should be unrelated to the data-structure streamed along rows. Therefore, we stream input data-structure along the rows and output data-structure along the columns. The weights are held stationary in the LRF, which holds the elements corresponding to the input and output channels processed by the MPE (based on its location in the array) for a given $Ki \times Kj$.

Figure 5 contains a simplified pseudocode that shows the loop sequence in which we ordered the elements. Since weights need to be block-loaded into the LRF before the computation begins, the interval between block-loads need to be maximized for high utilization. Therefore, we use $H \times W$ and N as our innermost loops, both of which reuse the block-loaded weights. We choose $Ki \times Kj$ as our next loop because the same input/output volume in the scratchpad can be reused in all iterations of the loop. Finally, we place the Ci and Co loops, whose loop counts are smaller as they mapped spatially in hardware. The chosen dataflow yields high utilization for almost all convolution layers other than the first layer whose Ci is small. It can also be used for fully-connected layers where $H \times W$ and $Ki \times Kj$ are 1, but requires frequent block-loads for small batch sizes.

B. SFU arrays: Full Spectrum of Activation Functions

Special Function Units (SFU) include both accurate and fast version for a spectrum of FP16 non-linear activation functions as well as higher precision FP32 operations. Activation functions including ReLU, Leaky-ReLU, Sigmoid, and Tanh for both forward and backward phases of DNN operations, normalization functions, and pooling operations use the SFUs. In addition, functions such as *sqrt*, *exponent*, *ln*, *Tanh*, *Sigmoid*, and *reciprocal* are realized using approximations. The SFU arrays are also used to realize operations such as shuffle, permute, and transpose which are used in the update phases of training.

As we scale the precision and the peak TOPs of the MPE array, the percentage of total cycles spent in auxiliary operations in the SFUs start to grow. This necessitated doubling the SFU arrays to maintain the balance between the compute time spent in ultra-low precision convolutions and matrix operations and higher-precision auxiliary operations.

C. Sparsity-aware Zero-gating and Frequency Throttling

To achieve a high TOPs/W when supporting ultra-low precision, the AI core architecture includes mechanisms to clock-gate FPU pipeline to save energy, and also includes sparsity-aware frequency throttling to maximize TOPs within the power limits.

1) Energy Savings: Zero-gating Support. Significant fraction of zeroes in the input operands opened up an opportunity to save power by not computing on zero operands. The MPEs include support to skip the entire FPU pipeline when multiplicands are zero and simply passes the addend to the result.

2) Sparsity-aware Frequency Throttling. Across the different layers of a given DNN, we observed significant differences in the sparsity of the weights. Since the distribution of sparsity in the weights are static for inference, RAPID exploits a hardware/software co-design to throttle power to maximize TOPs within the power limit. Unlike a DVFS based power modulation which involves costly voltage regulation and PLL loops, clock throttling guided by software is used to modulate the power within a single clock cycle time period.

As part of silicon characterization, we measured power (both dynamic and static) as a function of voltage, and also determined the frequency in the admissible voltage range. We extended this characterization to derive the stall rate for clock throttling at each operating point (voltage, frequency) by varying the degree of sparsity in the models. By using the power limits when operating at the nominal voltage and frequency, we determined the effective frequency for a given sparsity and used it to derive the stall rate at that operating point.

As shown in Figure 6 the graph compiler analyzes the sparsity of the weights for all the layers of DNN, and uses the

Si characterization data as input, to determine the throttling levels for individual layers, which pushes the power closest to the power envelope. As this can be done during compilation, the overhead is not in the critical path of the inference, and is also amortized over multiple inferences for a given DNN.

RAPID includes an on-chip power control module via clock-edge skipping which uses the throttling rate recommended by the compiler to reduce the overall execution time while operating within power and thermal limits.

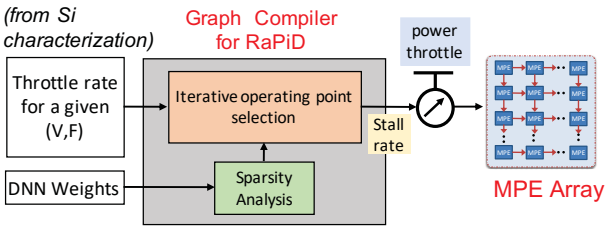


Figure 6: Workload-Aware Power Throttling

D. Ultra-low Precision Core with 2 corelets

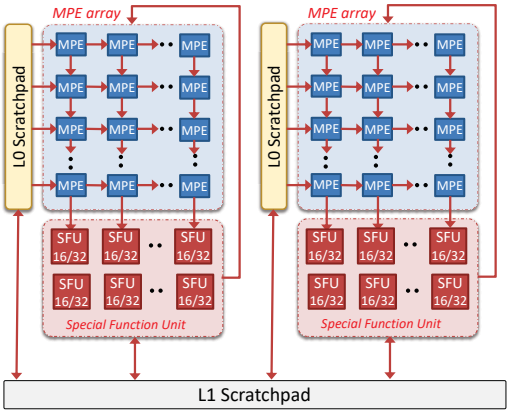


Figure 7: AI core: 2 corelets and shared L1 scratchpad

To maximize re-use of data from the L1 scratchpad and to exploit the reduction in capacity requirements due to ultra-low precision, the AI core combines 2 sets of MPE arrays, SFU arrays and L0 scratchpad, each referred to as a corelet, with a shared 2MB L1 scratchpad. As shown in Figure 7, the shared L1 scratchpad communicates with independent programmable load/store units with each of the corelet with a bandwidth of 128 bytes/cycle. The large 2MB capacity of the L1 scratchpad allows the intermediate outputs to be held on-chip especially in ultra-low precision inference use-cases. Similarly, the bandwidth from L1 to the 2 corelets is balanced to meet the speeds vs. feeds for the dataflows across different precisions. For example, with a modified weight stationary dataflow, the INT4 computations of the MPE still consume only $5/8^{th}$ of the available L1 bandwidth of 128 bytes/cycle. As the precision is scaled to INT2, each cycle a partial sum is written to the L1 scratch-pad reaching the limits of the available L1 bandwidth.

As shown in Figure 7, each AI core equipped with MPE arrays and SFU arrays along with the 2-tiered scratchpad hierarchy has all the necessary features to be used as a single-core edge accelerator, and at the same time can be used as a fundamental building block of a larger system comprising multiple cores.

E. Data Communication Among Cores and Memory

To communicate with the external memory and to be able to scale to a chip with multiple cores, we adopted a bi-directional ring interconnection with a bandwidth of 128 bytes/cycle in each direction to communicate data between cores and memory. Each core has a programmable Memory-Neighbor Interface (MNI) unit to facilitate data communication with memory and neighbors via a ring-interface unit (RIU). We have enabled asynchronous clock domain crossing support to allow the ring and the core to operate at different frequencies.

Figure 8 shows the overview of the MNI and the flow of requests and data returns between cores and memory using separate programmable load and store units (MNI-LU and MNI-SU). As the data access patterns in DNNs are both static and regular, data fetch latency can be effectively hidden by double-buffering data in the L1 scratchpad overlapped in time with computations in the core. The compiler blocks and tiles the program loops to guide the granularity of data fetches/stores by balancing the scratch-pad capacity and available bandwidth.

Each load/store request is assigned a unique identification tag which is generated as part of the execution of the send/receive primitives supported by the MNI-SU and MNI-LU, respectively. Using load and store queues, MNI supports multiple outstanding requests to neighbors and memory, and MNI-LU allows out-of-order data returns as the local scratch-pad address to be written is tracked in the load queue. MNI-LU and SU programs stall once the limit on the allowed outstanding requests is reached. To exploit the bi-directional ring bandwidth, the MNI-LU is designed to receive up to 2 data returns in any cycle, exploiting the banked architecture of the L1 scratchpad and the reservation management policy in the RIU to avoid bank conflicts in the L1 scratchpad.

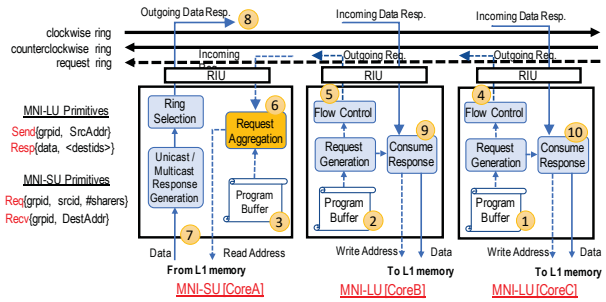


Figure 8: Multi-cast Support in Memory-Neighbor Interface

DNN workloads exhibit high-degree of parallelism allowing spatial partitioning of the work with data sharing across

multiple cores [52]. This data sharing behavior is exploited by supporting multi-cast communications both in the ISA and the hardware of MNI's load and store units. To support multi-cast data transfers, the send and receive primitives of MNI-SU and MNI-LU, respectively, are generated by the compiler by assigning common identification tags for the participating cores. As shown in Figure 8 a multi-cast data transfer from core A to cores B and C requires each consumer to make individual *Recv* request to core A (steps 1 and 2) using the common identification tag and specifying the number of participating consumers. Independently, core A's program includes a matching *Send* multi-cast data with the same identification tag, and number (list) of consumers (step 3) enabling scaling to large number of cores.

As highlighted in Figure 8 (steps 4 through 6), MNI-SU of core A includes hardware support for "request aggregation". After receiving requests from all the participating consumers, core A's MNI-SU *dynamically* constructs the list of consumers, reads data read from the scratch-pad (step 7) and posts to the ring with the common identification tag, and the list of consumers. Similar "request aggregration" support in the external memory interface enables MNI-LU of multiple cores to request shared data from the memory to be sent as a multi-cast transfer.

IV. RaPiD Chip For Training and Inference Systems

Figure 9 shows the architecture of the 4-core RaPiD chip [50] supporting 5 different data formats FP16 (1,6,9), FP8(1,4,3) (with programmable bias), FP8(1,5,2), INT4 and INT2 to take advantages of the break-through in the algorithmic approximation for training and quantized inference.

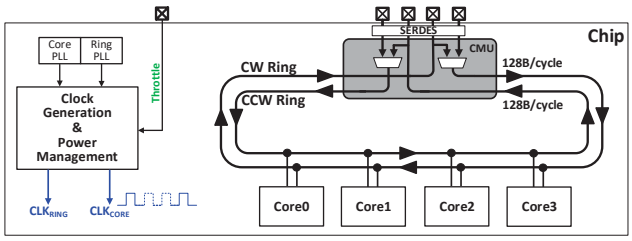
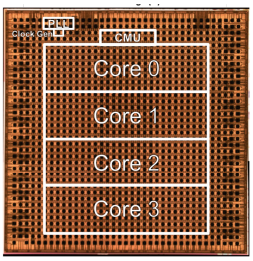


Figure 9: 4-core RaPiD architecture

The RaPiD chip consists of 4 cores connected to the bi-directional clockwise (CW) and counter-clockwise (CCW) ring. The ring and the cores operate in asynchronous clock domains with separate core and ring PLLs so as to optimally balance power/performance for compute and data movement. Cores communicate with each other and with memory using the memory/neighbor interface and the ring interface unit across the asynchronous boundary. To enable scaling to a larger system, the rings are connected through a chip management unit (CMU) that can either close the rings within a single RaPiD chip or connect multiple chips to form many-core systems. Finally, the RaPiD chip also includes power control module for workload-aware throttling via clock-edge skipping to fully utilize the chip's power budget for maximum application performance.



Technology	7nm
Chip Size	6mm x 6mm
Frequency	1.0 GHz - 1.6 GHz
Throughput	8 – 12.8 TFLOPS (fp16) 16 – 25.6 TFLOPS (hfp8) 64 – 102.4 TOPS (int4)
Power Management	Workload-Sparsity-Aware throttling via clock-edge skipping
Energy Efficiency	0.98 – 1.8 TFLOPS/W (fp16) 1.9 – 3.5 TFLOPS/W (hfp8) 8.9 – 16.5 TOPS/W (int4)

Figure 10: 4-core RaPiD Chip: 36mm^2 in 7nm EUV technology

Figure 10 shows the 36mm^2 RaPiD chip fabricated in 7nm EUV technology which at nominal voltage achieves 3.5 TFLOPS/W in HFP8 and 16.5 TOPS/W in INT4.

A. Inference and Training System

As the RaPiD chip architecture is designed to scale to a large number of cores, it is possible to construct multi-core/multi-chip inference and training systems.

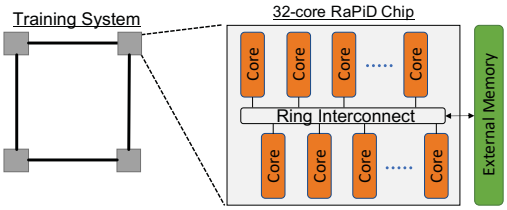


Figure 11: AI Training System: 4 chips with 32 cores

To estimate inference and training performance across different architecture configurations, we developed a detailed performance model of the RaPiD chip, and calibrated it to within 1% of the measurement results [50].

In this work, we study the performance of INT4 inference using a 4-core RaPiD chip model with 96 TOPs at 1.5 GHz. Figure 11 shows the HFP8 training system model with 760 TOPs using 4 RaPiD chips with 32-cores each at 1.5 GHz, connected using a high bandwidth interconnection to communicate gradients and weights during the update phase of training.

B. Software Architecture

Building and deploying an end-to-end AI system goes beyond designing only the accelerator core. AI systems must balance a diverse set of critical requirements: (i) deliver high *sustained* performance and processing efficiencies across workloads, (ii) have the flexibility to cater to future workloads which are rapidly evolving, (iii) integrate seamlessly within the existing AI software ecosystem while preserving end-user productivity.

Figure 12 shows the high-level overview of the in-house end-to-end software stack for the AI chip. We use a set of

compile-time and execution-time extensions [53] that are *pluggable* into existing frameworks This leverages existing capabilities of the DL frameworks yet enables aggressive, accelerator-specific performance optimization.

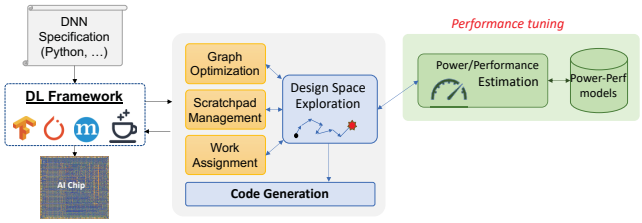


Figure 12: End-to-End Software Stack for Inference and Training

The key components of the software stack includes 1) a graph compiler which automatically identifies how best to execute a given DNN graph on the AI chip and constructs the program binaries; and 2) an execution runtime which triggers and manages the execution of compute and data-transfer operations on the AI chip.

As part of the compilation, a systematic design space exploration is performed focusing on graph optimization, scratchpad management, and work assignment to the cores of the AI chip. This design space exploration is guided by a bandwidth-centric analytical power-performance model of the AI chip which helps prune the search space to identify profitable mapping choices of operations to the AI chip. The performance model is validated against the hardware measurements and therefore serve to study scaled AI systems with multiple cores and chips in the context of both training and inference.

V. Results

In this section, we present the experimental methodology and summarize the performance and compute efficiency achieved for inference and training systems based on the RaPiD chip architecture.

A. Experimental Methodology

Performance Estimation. To estimate performance across different architecture configurations, we developed a detailed performance model of the RaPiD chip, which was calibrated to within 1% of the measurement results of [50]. The power consumed by the different DNN primitives (*e.g.* Convolution, ReLU, *etc.*) was measured in silicon and combined with the projected utilization of the different components (MPE array, SFU, scratchpad and others) to estimate compute efficiency (TOPS/W).

System Configuration. For inference, we study a RaPiD chip with 4 cores described in Section IV attached to an external DDR memory with 200 GBps bandwidth. For training, we consider a distributed system with 4 scaled-up RaPiD chips (Section IV-A), each containing 32 cores, 64MB distributed L1 scratchpad and attached to a High Bandwidth Memory (HBM) supplying 400 GBps bandwidth. The chip-to-chip interconnect bandwidth is 128 GBps. We also present the sensitivity of the performance as we scale both the inference and training systems.

Benchmarks. We use 11 state-of-the-art DNN benchmarks from a multitude of application domains: (i) Image classification - VGG16, Resnet50, InceptionV3, InceptionV4, MobileNetV1 trained on ImageNet dataset, (ii) Object detection - SSD300, YoloV3, YoloV3-Tiny trained on COCO dataset, (iii) Natural language - BERT (sequence length = 384) trained on WMT14 En-De dataset, 2-layer LSTM trained on PennTreeBank (PTB) dataset, and (iv) Speech - 4-layer bidirectional LSTM trained on SWB300 dataset.

Experimental Setup and Baseline. In our experiments, we consider a *batch size of 1* for inference, and a *minibatch size of 512* for training. For a fair comparison, we use the FP16 implementation on RaPiD (with identical system configuration) as the baseline to report relative improvements achieved at lower precisions. Based on performance numbers reported in MLPerf [54], our FP16 baseline is quite competitive compared to other accelerator designs, when normalized for technology and power/area. Further, we also provide the absolute inference latency and training throughput (inputs per second) achieved at lower precisions. For inference, we limit our study to FP8-fwd and INT4 precisions and reserve INT2 implementation for future work, as the models still result in $\approx$2% accuracy loss as mentioned in Section II. In addition, we study the benefits of sparsity-aware frequency throttling only in the context of DNNs pruned at FP16 precision, as combining pruning with ultra-low precision is still an evolving area of research.

B. Inference Performance and Efficiency

Figure 13 shows the inference latency (shaded contour) achieved by the 4-core RaPiD chip across the different benchmarks at FP8 (1-4-3 format) and INT4 precisions. Compared to the FP16 baseline on RaPiD, the FP8 and INT4 implementations achieve 1.2×-1.9× (average 1.55×) and 1.4×-4.2× (average 2.8×) improvement in end-to-end performance (bars) respectively. The speedup at lower precisions are primarily limited by the fraction of operations that are executed in FP16 *viz.* first and last layers, activation functions, normalization and pooling operations, among others. The image classification and object detection benchmarks with compute-heavy convolution layers achieve the best improvement, while mobile networks with lean convolutions and a significant fraction of auxiliary operations benefit the least.

Figure 14 shows the sustained compute efficiency (TOPS/W) achieved at FP8 and INT4 precisions (shaded contour). Thanks to the micro-architectural/circuit design of RaPiD, the TOPS/W scaling is quite strong across precisions - the FP8 implementations achieve 1.4-4.68 (average 3.16) TOPS/W, while INT4 achieves 3-13.5 (average

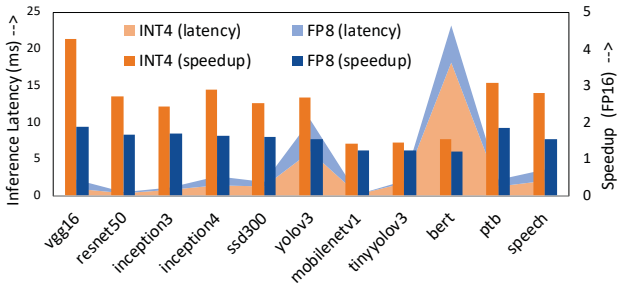


Figure 13: Classifications per second using 4-core RAPID chip

7) TOPS/W across benchmarks. This amounts to 1.6× and 3.6× improvement compared to the FP16 baseline (bars).

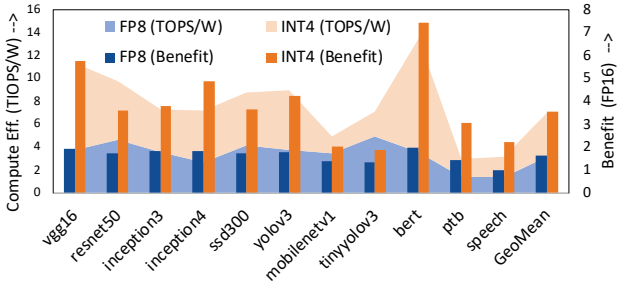


Figure 14: Sustained TOPS/W on 4-core RAPID chip

The results show that even for a batch size of 1, the RAPID chip achieves both high sustained TOPS and TOPS/W for inference at FP8 and INT4 precisions.

C. Training Throughput

Figure 15 shows the training throughput (*i.e.,* inputs trained per second) across the benchmarks in both FP16 and Hybrid-FP8 precisions for a training system with 4 RAPID chips (Section IV). Comparing FP16 *vs.* HFP8, the speedup ranges between 1.1×-2× (average 1.4×). Unlike inference, the availability of large mini-batch helps in achieving high core utilization at reduced precision during convolution and GEMM operations. However, overall speedups in training is slightly smaller compared to inference due to 2 key factors: (i) training incurs additional off-chip communication for gradient reduction and weight broadcast, and (ii) training is memory intensive as activations produced during the forward pass needs to be retained for computing the weight gradients during back-propagation.

D. Benefits of Sparsity-aware Throttling

We now present the improvement in performance achieved by the sparsity-aware zero-grating and frequency throttling scheme described in Section III-C. Given a power budget, Figure 16(a) shows the rate of frequency throttling applied at varying levels of sparsity for the 4-core RAPID chip derived from silicon measurements. We apply this scheme in the context of inference using publicly available sparse

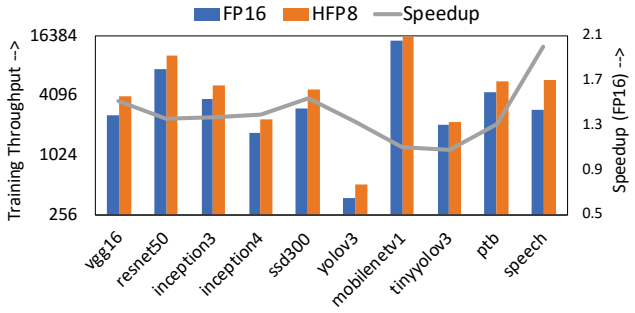


Figure 15: Throughput with 4-chip RAPID training system

(or pruned) models, as the degree of throttling can be ascertained at compile time. To this end, we consider pruned versions of a number of our benchmarks [55, 56, 57, 58]. The pruned models used FP16 precision; combining pruning with low precision is still an evolving area of research and we hope to explore this trade-off as part of future work. Figure 16(b) shows the average sparsity and the speedup achieved compared to a baseline with no sparsity-aware throttling across different benchmarks. The average sparsity varies (across layers and networks) between 50%-80% with negligible loss in accuracy. Correspondingly, we achieve 1.1×-1.7× (average 1.3×) improvement in performance by selecting correct operating frequency.

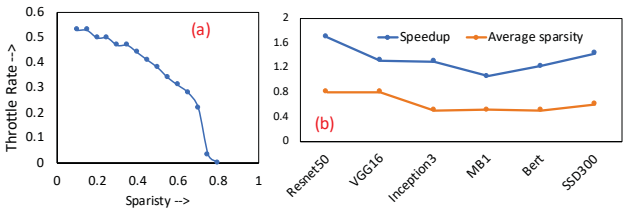


Figure 16: Performance benefits with sparsity-aware throttling

E. Performance Breakdown Analysis

We now present the key factors impacting the end-to-end performance on RAPID. For INT4 inference, Figure 17 shows the breakdown of the compute cycles into 4 key categories *viz.* Conv/GEMM, Conv/GEMM overheads, quantization and auxiliary operations. The first category constitutes Conv/GEMM operations that execute on the MPE array and leverage the full compute capabilities of the RAPID chip. Note that, while most layers are executed in INT4 precision, a small fraction of the layers may still need to executed in FP16 to preserve accuracy. The second category captures the overheads that occur during Conv/GEMM execution. These overheads come from several factors including dataflow inefficiencies due to spatio-temporal underuse of the MPE array for given workload dimensions and imbalance in work assigned to each core/corelet, among others. The third category includes additional quantization and scaling operations that needs to be performed to convert data between FP16 ⇔ INT4. Given the high throughput of the MPE array at low-precision,

this overhead becomes non-trivial, especially when the size of the activation is large. The final category includes the cost of other auxiliary operations (activation function, batch normalization *etc.*) which are executed in the SFU in FP16.

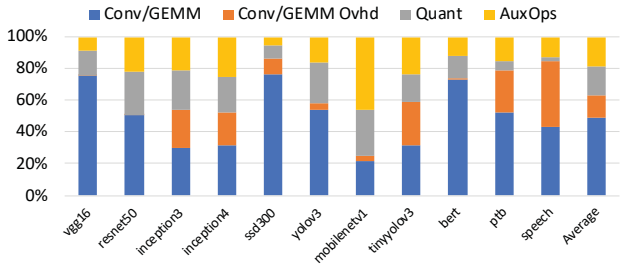


Figure 17: Breakdown of compute cycles for INT4 inference

We observe that the benchmarks are quite heterogeneous with respect to the compute cycles expended in each category. DNNs such as *inception3/4, Tiny-yolov3 and LSTMs*, incur overheads during CONV/GEMM operations as their workload dimensions do not exactly match the dimensions of the MPE array. Convolutional networks with large activation sizes incur quantization overheads, while mobile networks (*MobileNetv1, Tiny-yolov3*) contain the most auxiliary operations. On average, the Conv/GEMM occupy 50% of the compute cycles, while Conv/GEMM overheads, quantization and auxiliary operations amount to 14%, 17% and 19% respectively.

F. Inference/Training System Scaling

In this section we present the speedup achieved as we scale both the inference and the training systems. For inference systems, we increase the number of cores in the chip and show the speedup for INT4 precision, and for training systems, we increase the number of chips in the systems and show the speedup for HFP8 precision.

As shown in Figure 18(a) we see that even for a mini-batch of 1, performance scales as we scale the number of cores from 1 to 32. Compute-intensive benchmarks like *VGG16, Resnet50, Yolov3, SSD300* show performance improvement even as we scale to 32 cores. For benchmarks that are either auxiliary operations dominated (*MobileNetv1*), or memory stalls dominated due to the high TOPs of the INT4 engines, we see a saturation in the speedup as we increase the number of cores especially because we fixed the external bandwidth even as we scaled the number of cores in the system.

Similarly, Figure 18(b) shows the speedup with HFP8 training as we increase the total chips in the system from 1 to 32 with a chip-to-chip bandwidth of 128 GBps. These studies used data-parallelism and hence required communicating gradients, and the weights in the update phase of training. HFP8 reduces the communication overhead for weights since the forward pass uses only 8-bit weights, and each chip concurrently computes the updates for the weight portion it owns, and communicates only updated 8b weights to the neighbors.

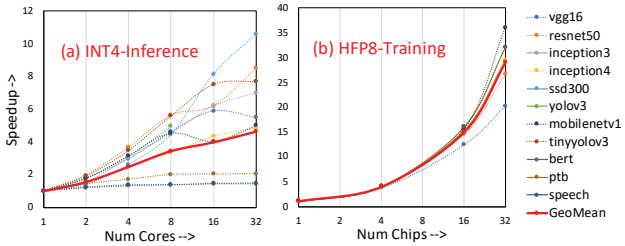


Figure 18: Performance scaling for inference and training

VI. Related Work

Improving efficiency of AI workloads on different hardware platforms is a vibrant topic of research. We describe related research efforts in accelerating AI workloads on CPUs, GPUs, accelerators and commercial AI chips.

CPU-based techniques. Accelerating AI workloads on CPUs includes the use of optimized linear algebra libraries [59], techniques for efficient parallelization on multi-cores [60, 61], as well as efficient data layouts and batching [62]. Some recent efforts have also proposed compiler, ISA and micro-architectural optimizations to exploit certain properties of AI workloads including sparsity [63, 64, 65].

GPU-based techniques. Research efforts on accelerating AI workloads on GPUs have focused on data/model/pipeline parallelization techniques [66], memory management [67] and locality-aware device placement [68, 69]. Similar to CPUs, some efforts have also explored exploiting sparsity in activations and weights [70].

Hardware accelerator techniques. Specialized hardware is key to satiate the computational needs of AI workloads. Recognizing this, a myriad of accelerators ranging from low-power ASIC / FGPA / CGRA cores [21, 22, 23, 24, 26, 32, 47, 71, 72, 73, 74, 75, 76, 77, 78] to large scale systems [29, 30, 31, 79, 80, 81] have been proposed. These architectures demonstrate impressive peak processing capabilities using dense arithmetic arrays, heterogeneous processing tiles, low-precision data representations and sometimes dynamic hardware reconfiguration. Recent efforts have explored exploiting sparsity in activations and weights [25, 27, 82], 3D memory technologies [28, 83], bit-serial architectures [84, 85] as well as in-memory computation [86, 87, 88, 89, 90, 91] to further boost efficiency.

Commercial AI chips. The immense success of specialized AI accelerators have further driven commercial efforts to design them. These include Google TPUs, NVIDIA Tensor Cores, Intel NNP, among others [33, 34, 35].

For all these AI accelerator solutions, scaling the performance across generations is pivotal to their success in commercial deployments. Since AI workloads present a unique opportunity for performance/energy improvement through *precision scaling*, we exploited that knob to design RaPiD to support a mixed-precision architecture capable of 8-bit training and 4-bit inference.

VII. Conclusion

We presented the design of a 4-core AI chip, called RAPID, supporting ultra-low precision training and inference [50]. RAPID supports mixed precision execution, which includes different number formats *viz.* FP16, Hybrid-FP8, INT4 and INT2. RAPID provides broad workload coverage (CNNs, LSTMs and transformers) and is scalable to multiple cores and chips. Silicon measurements from a 4-core RAPID chip in 7nm demonstrates 3.5 TFLOPS/W in FP8 mode and 16.5 TOPS/W in INT4 mode. Using a performance model calibrated to within 1% of the measurement results, we evaluated FP8 training for a 768 TOPs AI system comprising 4 RAPID chips, and INT4 inference for a 1 RAPID chip system. Our results show INT4 inference for batch size of 1 yields 3 - 13.5 (average 7) TOPS/W and FP8 training for a mini-batch of 512 achieves a sustained 102 - 588 (average 203) TFLOPS across a wide range of applications. As future work, we plan to study INT2 performance of RAPID and sparsity-aware power throttling for ultra-low precision.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[2] Pierre Sermanet, David Eigen, Xiang Zhang, Michael Mathieu, Rob Fergus, and Yann Lecun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *http://arxiv.org/abs/1312.6229*.

[3] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[5] C. Szegedy et. al. Going deeper with convolutions. In *Proc. CVPR*, 2015.

[6] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, June 2014.

[7] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016.

[8] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[9] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

[10] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

[11] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning, 2016.

[12] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[13] Y. Wu. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv*, 2016.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6000–6010, USA, 2017. Curran Associates Inc.

[15] G. Hinton et. al. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

[16] X. Zhang et. al. Text understanding from scratch. *arXiv*, 2015.

[17] S. Venugopalan et. al. Sequence to sequence - video to text. In *Proc. ICCV*, 2015.

[18] A. Hannun et. al. Deep speech: Scaling up end-to-end speech recognition. *arXiv*, 2014.

[19] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017.

[20] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, June 2018.

[21] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. *SIGARCH Comput. Archit. News*, 38(3):247–257, June 2010.

[22] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 WORKSHOPS*, pages 109–116, June 2011.

[23] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[24] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[25] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, June 2016.

[26] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 267–278, Piscataway, NJ, USA, 2016. IEEE Press.

[27] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 1–13, Piscataway, NJ, USA, 2016. IEEE Press.

[28] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392, June 2016.

[29] Abhinandan Majumdar, Srihari Cadambi, Michela Becchi, Srimat T. Chakradhar, and Hans Peter Graf. A massively parallel, energy efficient programmable accelerator for learning and classification. *ACM Trans. Archit. Code Optim.*, 9(1):6:1–6:30, March 2012.

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance

analysis of a tensor processing unit. In *Proceedings of the 44th International Symposium on Computer Architecture*, ISCA '17, 2017.

[31] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, volume 00, pages 13–26, June 2017.

[32] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 1–12, New York, NY, USA, 2013. ACM.

[33] Google supercharges machine learning tasks with tpu custom chip. *Google Research blog*, 2016.

[34] O. Wechsler, M. Behar, and B. Daga. Spring hill (nnp-i 1000) intel's data center inference chip. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–12, Aug 2019.

[35] Nvidia tensor cores: https://www.nvidia.com/en-us/data-center/tensorcore/. *NVIDIA blog*, 2017.

[36] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[37] Jan Achterhold, Jan Mathias Koehler, Anke Schmeink, and Tim Genewein. Variational network quantization. In *International Conference on Learning Representations*, 2018.

[38] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[39] Lu Hou and James T Kwok. Loss-aware weight quantization of deep networks. *arXiv preprint arXiv:1802.08635*, 2018.

[40] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. Weighted-entropy-based quantization for deep neural networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7197–7205, 2017.

[41] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5918–5926, 2017.

[42] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks, 2018.

[43] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[44] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 7686–7695, 2018.

[45] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In *Advances in Neural Information Processing Systems 32*, pages 4901–4910. Curran Associates, Inc., 2019.

[46] Jungwook Choiand, Swagath Venkataramani, Vijayalakshmi Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce I-Jen Chuang. Accurate and efficient 2-bit quantized neural network. *SysML*, 2019.

[47] Y. H. Chen, T. Krishna, J. Emer, and V. Sze. 14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, Jan 2016.

[48] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky, N. Cao, C.Y. Chen, P. Chuang, T. Fox, G. Gristede, M. Guillorn, H. Haynie, M. Klaiber, D. Lee, S.Lo, G. Maier, M. Scheuermann, S. Venkataramani, C. Vezyrtzis, N. Wang, F. Yee, C. Zhou, P. F. Lu, B. Curran, L. Chang, K. Gopalakrishnan. A scalable multi-teraops deep learning processor core for ai training and inference. In *Proc. VLSI Symposium*, June 2018.

[49] J. Oh, S. K. Lee, M. Kang, M. Ziegler, J. Silberman, A. Agrawal, S. Venkataramani, B. Fleischer, M. Guillorn, J. Choi, W. Wang, S. Mueller, S. Ben-Yehuda, J. Bonanno, N. Cao, R. Casatuta, C. Y. Chen, M. Cohen, O. Erez, T. Fox, G. Gristede, H. Haynie, V. Ivanov, S. Koswatta, S. H. Lo, M. Lutz, G. Maier, A. Mesh, Y. Nustov, S. Rider, M. Schaal, M. Scheuermann, X. Sun, N. Wang, F. Yee, C. Zhou, V. Shah, B. Curran, V. Srinivasan, P. F. Lu, S. Shukla, K. Gopalakrishnan, and L. Chang. A 3.0 tflops 0.62v scalable processor core for high compute utilization ai training and inference. In *2020 IEEE Symposium on VLSI Circuits*, pages 1–2, 2020.

[50] Ankur Agrawal, Sae Kyu Lee, Joel Silberman, Matthew Ziegler, Mingu Kang, Swagath Venkataramani, Nianzheng Cao, Bruce Fleischer, Michael Guillorn, Matthew Cohen, Silvia Mueller, Jinwook Oh, Martin Lutz, Jinwook Jung, Siyu Koswatta, Ching Zhou, Vidhi Zalani, James Bonanno, Robert Casatuta, Chia-Yu Chen, Jungwook Choi, Howard Haynie, Alyssa Herbert, Radhika Jain, Monodeep Kar, Kyu-Hyoun Kim, Yulong Li, Zhibin Ren, Scot Rider, Marcel Schaal, Kerstin Schelm, Michael Scheuermann, Xiao Sun, Hung Tran, Naigang Wang, Wei Wang, Xin Zhang, Vinay Shah, Brian Curran, Vijayalakshmi Srinivasan, Pong-Fei Lu, Sunil Shukla, Leland Chang, and Kailash Gopalakrishnan. 9.1 a 7nm 4-core ai chip with 25.6tflops hybrid fp8 training, 102.4tops int4 inference and workload-aware throttling. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 64, pages 144–146, 2021.

[51] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh R. Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[52] Swagath Venkataramani, Vijayalakshmi Srinivasan, Jungwook Choi, Philip Heidelberger, Leland Chang Chang, and Kailash Gopalakrishnan. Memory and interconnect optimizations for peta-scale deep learning systems. In *Proceedings of the 26TH IEEE International Conference On High Performance Computing, Data, and Analytics*, 2019.

[53] S. Venkataramani, J. Choi, V. Srinivasan, W. Wang, J. Zhang, M. Schaal, M. J. Serrano, K. Ishizaki, H. Inoue, E. Ogawa, M. Ohara, L. Chang, and K. Gopalakrishnan. Deeptools: Compiler and execution runtime extensions for rapid ai accelerator. *IEEE Micro*, Sep. 2019.

[54] Mlperf benchmark suite: https://mlperf.org.

[55] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

[56] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[57] Pravendra Singh, R Manikandan, Neeraj Matiyali, and Vinay Namboodiri. Multi-layer pruning framework for compressing single shot multibox detector. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1318–1327. IEEE, 2019.

[58] Mitchell A. Gordon, Kevin Duh, and Nicholas Andrews. Compressing bert: Studying the effects of weight pruning on transfer learning. In *RepL4NLP@ACL*, 2020.

[59] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proc. OSDI*, pages 265–283, 2016.

[60] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In P. Bartlett, F.c.n. Pereira, C.j.c. Burges, L. Bottou, and K.q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1232–1240. 2012.

[61] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidyanathan, Srinivas Sridharan, Dhiraj D. Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *CoRR*, abs/1602.06709, 2016.

[62] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[63] Sanchari Sen, Shubham Jain, Swagath Venkataramani, and Anand Raghunathan. Sparce: Sparsity aware general-purpose core extensions to accelerate deep neural networks. *IEEE Trans. Comput.*, 68(6):912–925, June 2019.

[64] Berkin Akin, Zeshan A Chishti, and Alaa R Alameldeen. Zcomp: Reducing dnn cross-layer memory footprint using vector extensions. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 126–138, 2019.

[65] Zhangxiaowen Gong, Houxiang Ji, Christopher W. Fletcher, C. Hughes, Sara S. Baghsorkhi, and J. Torrellas. Save: Sparsity-aware vector engine for accelerating dnn training and inference on cpus. *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 796–810, 2020.

[66] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR*, abs/1511.00175, 2015.

[67] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. Virtualizing deep neural networks for memory-efficient neural network design. *CoRR*, abs/1602.08124, 2016.

[68] Azalia Mirhoseini, Hieu Pham, Quoc Le, Mohammad Norouzi, Samy Bengio, Benoit Steiner, Yuefeng Zhou, Naveen Kumar, Rasmus Larsen, and Jeff Dean. Device placement optimization with reinforcement learning. 2017.

[69] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1662–1670, Stockholmsmassan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[70] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

[71] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 696–701, June 2014.

[72] S. Eldridge, A. Waterland, M. Seltzer, J. Appavoo, and A. Joshi. Towards general-purpose neural network computing. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 99–112, Oct 2015.

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

[74] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. A cgra-based approach for accelerating convolutional neural networks. In *Proceedings of the 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, MCSOC '15, pages 73–80, Washington, DC, USA, 2015. IEEE Computer Society.

[75] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 393–405, June 2016.

[76] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 16–25, New York, NY, USA, 2016. Association for Computing Machinery.

[77] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564, Feb 2017.

[78] Mateja Putic, Swagath Venkataramani, Schuyler Eldridge, Alper Buyuktosunoglu, Pradip Bose, and Mircea Stan. Dyhard-dnn: Even more dnn acceleration with dynamic hardware reconfiguration. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[79] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society.

[80] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018.

[81] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, and et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[82] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. SCNN: an accelerator for compressed-sparse convolutional neural networks. *CoRR*, abs/1708.04485, 2017.

[83] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 751–764, New York, NY, USA, 2017. Association for Computing Machinery.

[84] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[85] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, June 2018.

[86] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, June 2016.

[87] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *SIGARCH Comput. Archit. News*, 44(3):27–39, June 2016.

[88] L. Song, X. Qian, H. Li, and Y. Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 541–552, Feb 2017.

[89] J. Zhang, Z. Wang, and N. Verma. In-memory computation of a machine-learning classifier in a standard 6t sram array. *IEEE Journal of Solid-State Circuits*, 52(4):915–924, April 2017.

[90] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, and et al. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 715–731, New York, NY, USA, 2019. Association for Computing Machinery.

[91] Shubham Jain, Sumeet Kumar Gupta, and Anand Raghunathan. Tim-dnn: Ternary in-memory accelerator for deep neural networks, 2019.