Domain-Specific Computing Architectures and Paradigms

by

Ching-En Lee

A dissertation proposal submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical and Computer Engineering)
in the University of Michigan
2020

Doctoral Committee:

Associate Professor Zhengya Zhang, Chair
Associate Professor Reetuparna Das
Professor Michael P. Flynn
Professor Wei Lu

Ching-En Lee

lchingen@umich.edu

ORCID iD: 0000-0002-5130-8166

ACKNOWLEDGEMENTS

To all that have loved and supported me in my life.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Machine learning (ML) [1] and in particular deep learning (DL) [2] have proven to be the key approaches in solving complex cognition and learning problems. ML algorithms have demonstrated unprecedented success across a variety of applications including image recognition [3], object detection and tracking [4–6], natural language processing [7], robotic motion planning, perception and control [8]. Moreover, an emerging frontier of active research is real-time machine learning (RTML), which combines modern DL, ML and traditional statistical inference techniques to reinforce the optimal decision making in real time. Despite the advancement of powerful RTML algorithms, the required performance and stringent energy requirements create a gap for efficient domain-specific computing architectures and infrastructures, hindering the development of next-generation acceleration software and hardware.

This thesis explores three abstract levels within the spectrum of domain-specific computing acceleration, where specialized computing hardware and software architectures and frameworks are proposed to draw synergies with the increasingly complex ML and DL algorithms.

First, I will begin with the investigation of optimizations within neuro-inspired computing algorithms and hardware architectures. This work introduces a sparse spatio-temporal (ST) cognitive system-on-a-chip (SoC), designed to extract ST features from videos for action classification and motion tracking. The SoC core is a sparse ST convolutional auto-encoder that implements recurrence using a 3-layer network. High sparsity is enforced in each layer of processing, reducing the complexity

of ST convolution by two orders of magnitude and allowing all multiply-accumulates (MAC) to be replaced by select-adds (SA). The design is demonstrated in a 3.98mm^2 40nm CMOS SoC with an OpenRISC processor providing software-defined control and classification. ST kernel compression is applied to reduce memory by 43%. At 0.9V and 240MHz, the SoC achieves 1.63TOPS to meet the 60fps 1920×1080 HD video data rate, dissipating 127mW.

Second, I will elaborate on the impacts of dataflows and reduction mechanisms for deep neural network (DNN) acceleration. Specifically, I will quantify the inefficiencies of Cartesian Product-based dataflow and address its limitation for sparse DNN accelerations and propose Stitch-X [9], a novel DNN inference accelerator that efficiently *stitches* together both sparse weights and input activations. This design features a novel dynamic, look-ahead index matching unit in hardware [10] to efficiently extract reducible computation and feed them into a multi-level, spatial-temporal reduction dataflow, achieving high energy efficiency and low control complexity for a wide variety of DNN layers. Our evaluation demonstrates that Stitch-X delivers up to 4.3× speedup over an efficient, dense DNN accelerator, 1.6× speedup and 2.1× energy-delay-product improvement compared to a state-of-the-art sparse DNN accelerator.

Lastly, I will expand the domain-specific acceleration scope to cover RTML algorithms, and propose a new architecture modeling framework for joint software and hardware optimization. Specifically, I will introduce ERA, a new end-to-end development framework for developing RTML-specialized acceleration architectures and systems from software to hardware. ERA consists of two components: HANA, a set of high-performance RTML-specific architecture design templates, and PyHLM, an open-source Python-based high-level modeling and compiler tool chain for cross-stack architecture design and exploration. Using ERA, this work demonstrates a real-time visual object tracking (VOT). The optimized accelerator achieves an average speedup of 2.1× over state-of-the-art architecture design patterns across a wide-range of mod-

ern RTML algorithms.

The three pieces of work presented in this thesis constitute an ultimate vision of a high-level architecture modeling infrastructure that can enable agile hardware and software development and optimizations beyond the state-of-the-art compute-store patterns. To conclude, this thesis contributes to a concrete direction for liberating the development of domain-specific acceleration architectures and a foundation for enabling next-generation intelligence from the edge to the cloud.

CHAPTER I

Introduction

We live in an exciting era. Artificial intelligence (AI) has fundamentally changed the world and is also drastically shifting the dynamics of industries and businesses around the world. The powers granted by AI algorithms such as machine learning (ML) and deep learning (DL) has significantly altered the ways in which humans tackle problems in the emerging field of security and surveillance, autonomous driving, industrial automation and much more complex applications.

However, the path to achieving penetration of ML and DL applications is challenging. On one hand, the capability and development of ML and DL algorithms is still in flux. On the other hand, the hardware and software performance requirements (e.g. power, performance and cost) are also key factors that are hindering mass commercial deployment. Despite the obstacles ahead, the emerging capabilities of AI has also revealed rich opportunities for cross-stack architecture research. Specifically, the crossover from compute algorithm to software architectures, modeling frameworks to silicon architecture designs constantly drive new innovations within the research field of domain-specific computing acceleration.

Before diving into complete domain-specific architecture development frameworks, we will take a few steps back to revisit the history and fundamentals of AI algorithms. We find that one of the earliest attempts to achieve artificial cognition and learning is

by mimicking the method in which animals learn and perceive the world. Specifically, by modeling the ways neurons are structured, this approach is often referred to as biologically-inspired neural network modeling. In addition, the way that animals convey complex information and drive action has also inspired researchers to formulate computing models that surround the very basis of neuronal behaviors, and by using these models to perform computation, these methods are denoted as neuro-inspired computing.

In other words, neuro-inspired computing algorithms is: A class of computing algorithms for learning and inference that is referncing the ways in which animals gain cognition of the world. It is inspired by: 1) the way animals perform action, e.g., neuron firing and inhibition; 2) the way animals store data, e.g. over-complete and sparse data storage; and 3) the way animals perceive the world, e.g. spatio-temporal modalities. Notably, most neuro-inspired computing resemble some variant of neural networks (NNs). A neural network is essentially an explicit expression of neuron connections, representing features that could be learned in different dimensions. The term deep neural networks (DNNs) hence refers to multiple layers of neurons that are connected together, where the data can propagate from one layer to the next in a pipelined fashion.

Typically, there are two major functions that a NN needs to perform: 1) training; and 2) inference. Training refers to the process where a model neural network fits itself using a given set of input data and mathematically optimizes for a target metric (e.g. reconstruction loss) that describes how well the NN model is performing with respect to its current trained status. In addition, not all NN algorithms require labels (i.e. ground truths). When using a training algorithm that uses both input data and labels, it is referred to as supervised learning, and for those that do not require labels, they are referred to as unsupervised learning.

The training process can be static (offline) or dynamic (online). Where static

training refers to tuning the NN model representation before it is deployed into the field, and dynamic training refers to tuning the NN models on-the-fly while they are in use. The typical bottleneck for dynamic training lies in the required amount of memory, the introduced computation overhead and the training algorithms that can be used. For dynamic training, the method is likely to be unsupervised due to the lack of readily available labeled data.

The inference process takes a trained NN and passes the input data through the network to obtain the output result. Depending on the purpose of the NN, the output may be a class label prediction, bounding boxes, object boundaries etc. To elaborate, if the NN is tasked to discern different classes of objects, the application is referred to as classification. Moreover, the input data can take on may different forms and modalities, such as audio, images and videos.

Recent domain-specific computing works have focused on the acceleration of the inference process, mainly due to its deterministic simplicity and abundant data parallelism. Many works regarding DNN acceleration have emerged as the prime targets for domain-specific computing architecture research. The direct implications for these types of dedicated hardware is their capability to harness a sufficient amount of data parallelism, which requires a significant amount of compute and storage bandwidth. On top of these constraints, given the trend for the increasing depth of NNs together with the evolving network architecture, the complexities also scale exponentially for the underlying hardware, and in particular specialized computing hardware to service the NN inference demands.

On the two ends of the flexibility and efficiency spectrum, we find that specialized accelerators are situated at the efficiency extreme (Figure 1.1), in contrast to Von Neumann architectures such as CPUs. Specialized accelerator architectures benefit from compute-store paradigms such as dataflows, which enable large data and memory bandwidths while removing the burdens of conventional CPU control and

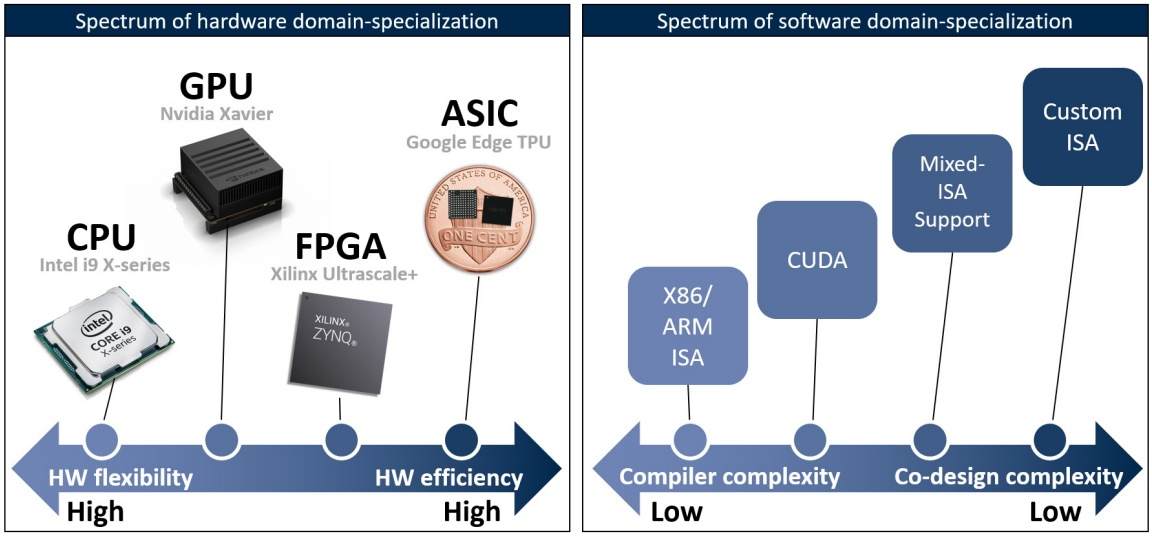


Figure 1.1: The spectrum of hardware and software domain specialization.

scheduling overhead. However, in terms of programmability and the capability to adapt to different algorithms and applications, they often fall short. Consequently, this introduces a significant gap in between the software infrastructure and the hardware optimizations that can be achieved.

Notably, domain-specific computing paradigms focus on three levels of software and hardware interactions: 1) the exploration of a specific algorithm (e.g. neuro-inspired computing algorithm) and its tailored optimizations and impacts on a specialized computing architecture; 2) the investigation of an algorithm regime (e.g. DNNs) and their generic implications to the hardware compute and storage architecture; and 3) the design of architecture modeling framework that can provide agile prototyping and deep SW-HW co-optimization capabilities for a wide range of algorithms and applications.

The following sections will briefly touch upon these three aspects: 1) neuromorphic computing acceleration; 2) sparse deep neural network acceleration; and 3) real-time machine learning processing architecture modeling framework.

1.1 Neuromorphic Computing Acceleration

Neuromorphic computing is an attempt at modeling the behaviors of how animals approach cognition and learning. It is a rich research field that has focuses on how to draw efficiency from tailored compute and storage patterns. Specifically, visual recognition is a major application that exhibits huge potentials in the field of security and surveillance and autonomous driving. One of the earliest works discussing the modeling of the mammalian primary visual cortex is presented by Olshausen and Field [11], where they showed a sparse coding algorithm that exhibits an overcomplete set of receptive fields (features) similar to the primary visual cortex whilst being able to exhibit sparse characteristics. This discovery is especially important for object detection, as traditional features such ad Harris [12], Hessian [13] are hand-crafted, and could not be well tailored to deliver the best detection performance with changing environments.

On top of the foundations of Olshausen and Field, Rozell [14] later presented the locally-competitive algorithm (LCA), a compressed sensing method where local neuron activation inhibits nearby neurons from firing; a means to enforcing sparse activation and representation of an input image. This feature is extremely important as the algorithm is unsupervised, leading to a potential for on-chip learning and efficient feature extraction for visual recognition and associated tasks.

Notably, action classification is an application within the field of visual recognition, where an image or video is used as an input, and the actions within need be classified. This is a compute and storage demanding problem, as modern images or videos demand high resolution, high frame rates, not to mention the requirement for real-time performance on resource-constrained platforms. Traditional methods for action classification uses 2-D images as inputs, however, only the spatial information could be extracted as features, whilst losing great potential for the temporal information from frame to frame. Formally put, action classification operates on sequences of image

frames to extract the activity or action from videos. Different from single-image recognition, classifying videos relies on extracting spatio-temporal (ST) features and using the ST features to build a classifier. Consequently, the computation is 3-D as opposed to 2-D for images.

Spatio-temporal receptive fields (STRFs) are understood as features or basis functions of videos [11]. STRFs can be extracted by unsupervised learning using an auto-encoder. Due to the high redundancy in video data, a compressed video encoding can be obtained using a sparse spatio-temporal (ST) auto-encoder. This neuro-inspired approach provides not only efficient video coding but also cognitive processing capabilities such as action classification and motion tracking [15].

1.2 Sparse Deep Neural Network Acceleration

Deep learning or more specifically, deep neural network (DNN), has emerged to be a key approach to solving complex cognition and learning problems [2,3]. State-of-the art DNNs [4,16–21] require billions of operations and hundreds of megabytes to store activations and weights. Given the trend towards even larger and deeper networks, the ensuing compute and storage requirements will prohibit real-time, low-power deployment on platforms that are resource and energy constrained. The compute and storage challenges motivated efforts in network pruning to zero out a large number of weights (W) of a DNN model with as little effect on the inference accuracy as possible [22–24]. In addition to sparsity in weights, the commonly-used rectifier linear unit (ReLU) clamps all negative activations to zeros, resulting in sparsity in output activations (OA), which become input activations (IA) of the next layer.

Data sparsity can be exploited to save power. Many DNN accelerators, e.g., Eyeriss [25], gates the computation, e.g., by turning off the clock, whenever a zero in the IA is detected in runtime. Most dense DNN accelerators can incorporate this technique to reduce power, but it does not shorten the latency or improve the throughput.

Cnvlutin [?] and Cambricon-X [26] are well-known early architectures that exploit sparsity in compressed IA for latency reduction and throughput improvement. However, they were designed to work with the sparsity in one of the two operands, W or IA, but not both. A dense processing architecture can be easily adapted to support one-operand sparsity by indirect data access.

To fully exploit sparsity in both operands, W and IA are stored in a compressed form where nonzero elements are represented by value-index pairs. Storage in a compressed form can reduce the memory size and bandwidth. However, unlike the common dense array and matrix storage, a compressed storage is not amenable to regular and efficient vector processing. One approach is to decompress the compressed form before processing, but decompression costs performance, memory, and power. Instead, state-of-the-art sparse DNN accelerators [27–31] process data directly in the compressed form, offering both low memory bandwidth and high degree of acceleration.

Data sparsity leads to better performance and efficiency, but major challenges remain:

- Front-end challenge: Multiplier under-utilization due to an insufficient number of W-IA pairs that can be extracted and dispatched to the multiplier array.
- Back-end challenge: Data traffic and access contention to support accumulation of psums whose destination addresses are seemingly random.
- Flexibility challenge: Limited support for different kernel sizes and layer types.

State-of-the-art sparse DNN accelerators including EIE [27], SCNN [28], Sticker [29, 30], and Eyeriss v2 [31] addressed some of the challenges in sparse DNN processing, but did not solve all of them. EIE exploits both W and IA sparsity but is restricted to fully-connected (FC) layers. SCNN is the first attempt at exploiting both W and IA sparsity for convolution (CONV) layers. It maximizes multiplier utilization at the cost

of massive psum writeback traffic and access contention, and it supports only CONV layers. Sticker follows SCNN's dataflow and uses 2-way set-associative processing elements (PEs) to alleviate the access contention but requires offline preprocessing to re-arrange IA data. Without the data re-arrangement, the access contention remains as significant as in SCNN. Eyeriss-v2 employs a two-step search frontend to find effectual W-IA pairs by first fetching nonzero IAs, and then using the channel index of the IA to look for nonzero Ws. Eyeriss-v2 adopts an Eyeriss-like row stationary dataflow [25] to avoid memory access contention.

1.3 Real-Time Machine Learning Processing Architecture Modeling Framework

From previous sections, we have formulated a preliminary consensus that machine learning (ML) [1] and in particular deep learning (DL) [2] have proven to be the key approaches in solving complex cognition and learning problems. ML algorithms unarguably demonstrated unprecedented success across a variety of applications including image recognition [3], object detection and tracking [4–6], natural language processing [7], robotic motion planning, perception and control [8]. One emerging frontier of active research is real-time machine learning (RTML) that combines modern DL, ML and traditional statistical inference techniques to reinforce the optimal decision making in real time.

Unlike conventional DL computation, RTML requires heterogeneous processing to support dynamically changing workloads. RTML applications often impose a short millisecond completion latency and a high processing throughput, and they are frequently deployed on edge platforms with stringent resource constraints [32]. To sum up, the heterogeneous processing, the dynamically changing workloads, the low-latency and high-throughput processing, and the resource-constrained platforms

present unique challenges for the systematic design of RTML processing architectures.

RTML can be considered an application domain. Recent work in domain-specific architecture (DSA) [33–35], and DSA development frameworks [36–39] have provided a number of elements that are needed in constructing a full-fledged RTML development framework. These elements include hardware primitives, software tools, and full-stack frameworks that capture all aspects of the optimization opportunities.

Hardware primitives refer to the building blocks for constructing hardware architectures. Buffets [37], for example, presents a hardware storage idiom featuring a modular memory hierarchy for accelerator design, reducing the design effort and streamlining DSA generation. However, Buffets may not reach the best performance potential due to the compute-store synchronization scheme that complicates module integration and programming. As another example, MatchLib [40] provides a suite of C/C++ modules designed to be synthesizable to RTL using high-level synthesis [41] frameworks to target application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs). However, MatchLib focuses on register-transfer level (RTL) generation, and lacks high-level architecture modeling and exploration that are necessary for capturing RTML's unique optimization opportunities.

Software tools focus on building compilers and generators to produce optimized implementations of specific suites of algorithms on given hardware platforms. TVM [42] is the first generic DL compiler that optimizes the DL computation graph for mapping to specialized accelerator architectures. Pixel Visual Core [43] leverages Halide [44] language for generating high-level virtual instruction set architectures (ISAs) that can be compiled to physical ISAs for on-device execution. Despite the benefits, current software-level tools are still limited to specific classes of well-studied algorithms, which are too restrictive to address RTML's diversity.

Full-stack frameworks combine both hardware primitives and software tools. A full-stack framework leverages compiler's flexibility together with parameterized

hardware modules to enable rapid design space exploration given a set of resource constraints. VTA [45] is a recent example of a full-stack development framework that combines a programmable DL-acceleration architecture template with a TVM compiler backend with its just-in-time (JIT) compiler for run-time instruction streaming. However, VTA is restricted to DL applications. VTA's hardware templates are based on general matrix multiply (GEMM), where compile-time scheduling is possible. RTML requires versatile processing patterns beyond GEMM, and it also requires run-time scheduling to support dynamically changing workloads. As a result, VTA captures only one aspect of RTML, but not the entirety.

Real-time machine learning (RTML) requires a full-stack design framework that encompasses both hardware primitives and software tools to target the unique features of RTML applications. Towards this goal, we target RTML's four key characteristics: 1) heterogeneous processing patterns; 2) dynamically changing workloads; 3) low-latency; and 4) high-performance processing, and resource-constrained platforms. By providing both hardware design patterns and custom-defined instructions, as well as a suite of tools from compiler to high-level architecture modeling and design space optimization.

CHAPTER II

Neuro-inspired Computing Accelerator

Spatio-temporal receptive fields (STRFs) are understood as features or basis functions of videos [11]. STRFs can be extracted by unsupervised learning using an auto-encoder. Due to the high redundancy in video data, a compressed video encoding can be obtained using a sparse spatio-temporal (ST) auto-encoder [46]. This neuro-inspired approach provides not only efficient video coding but also cognitive processing capabilities such as action classification and motion tracking [15, 47] (Figure 2.1).

2.1 Spatio-Temporal Cognitive SoC

The core of the SoC[1] chip is a sparse ST convolutional auto-encoder that consists of 192 neurons, with each supporting a kernel up to 6×6×8 (6×6 frame, spanning 8 time steps) (Figure 2.2). The auto-encoder is configurable with several settings: 64, 128 or 192 neurons, frame size from 1 to 36 and time steps from 1 to 8. Inputs are streamed in to the frame load queue, and ST kernels are reconstructed from their compressed storage prior to performing ST convolutions. The core is integrated with memory and an OpenRISC processor through a common control bus. The OpenRISC processor is programmed by an ISA together with a configuration and a classifier

[1] Thomas Chen contributed to the physical design of the STLCA chip, Jie-fang Zhang contributed to the machine learning model for off-line action classification and Chester Liu for algorithm and implementation consultation

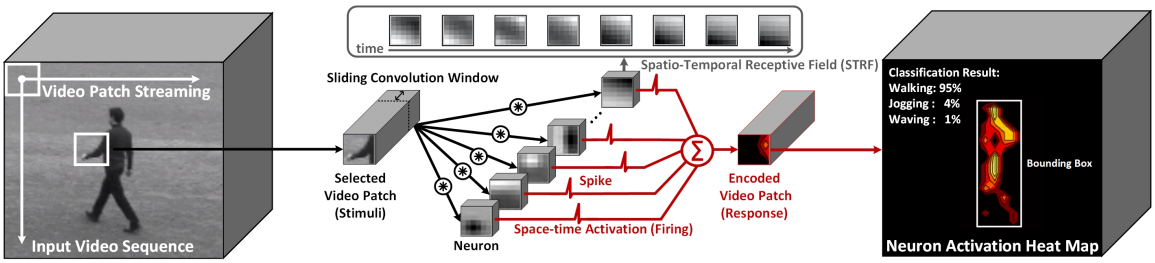


Figure 2.1: Cognitive processing of video input using spatio-temporal (ST) convolutional auto-encoder for human action classification and motion tracking.

profile. The configuration profile controls the operation of the core during runtime; and the classifier profile configures the on-chip classification algorithm. The outputs of the core are sent to a communication hub in the OpenRISC processor.

2.2 Sparse Recurrent Network Architecture

Sparsity is often enforced by an L-1 normalization term as part of the cost function in reference auto-encoder designs [11,46] (Figure 2.3). To achieve an even higher sparsity, we reformulate the auto-encoder as a 3-layer recurrent network (Figure 2.3(b)), and introduce L-1 normalization in two layers using rectification (Fig. 4): 1) in Layer 1 (L1), neurons compute ST convolutions to compute the recurrence and apply min/-max rectification (i.e., hard thresholding to binary levels) to enforce a sparse spike rate of S_1, reducing the downstream workload by a factor up to $1/S_1$; 2) in Layer 2 (L2), neurons compute ST convolutions to compute the potential update; and 3) in Layer 3 (L3), neuron potentials are thresholded to generate sparse spikes at a target rate of S_3. The spikes are fed back to L1, reducing L1's workload by a factor up to $1/S_3$. The three layers are fully parallelized using 192 neurons in each layer. Each L1 and L2 neuron performs a 6×6×8 ST convolution at a time, and each L3 neuron updates its potential and performs thresholding. The number of iterations through the 3 layers is adjustable between 2 to 32 to meet processing requirements. To achieve a high classification accuracy while maintaining a low-power operation, the sparsity

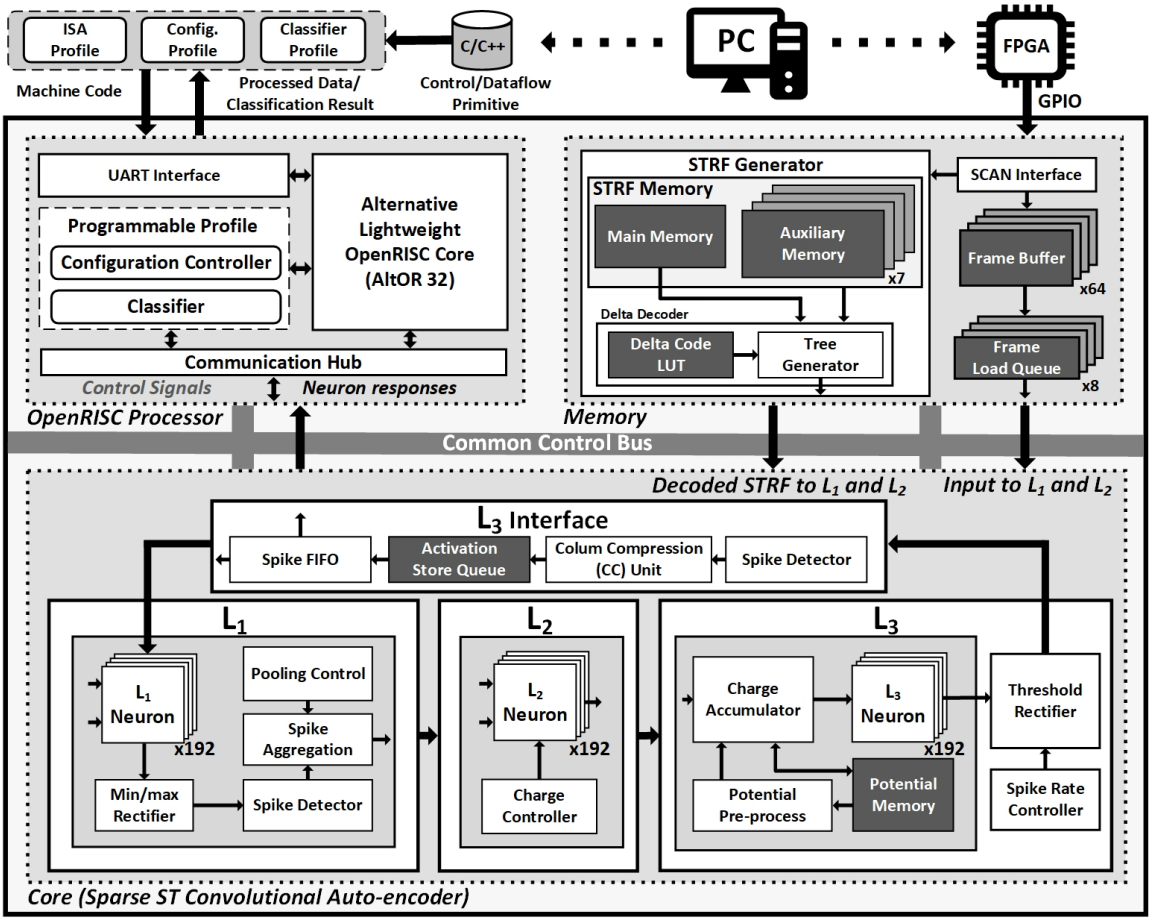


Figure 2.2: Sparse spatio-temporal (ST) cognitive SoC system architecture, including an OpenRISC processor, memory, and a sparse ST convolutional auto-encoder (core).

targets S_1 and $_1$ are set to 3% and 1% respectively (Figure 2.4), i.e., 97% and 99% of the L1 and L3 outputs are zero, enabling significant complexity and power reduction. In one iteration, the combined L1 and L2 workload is reduced to only 1 to 3% of the equivalent 3.54M OPs (an OP is defined as an equivalent 8b MAC) for a 6×6×64 (6×6 frame, 64 time steps) input video patch.

Spike-Based Inference and Sparsity-Enabled Compression Spike inputs to L1 and L2 simplify L1 and L2 neuron implementation from multiply-accumulates (MAC) to select-adds (SA) triggered by sparse spikes (Figure 2.5(a), (b)), thereby reducing neuron's power and area by 8.1× and 10.1× respectively. Spikes are detected by successively ANDing the spike train with its two's complement, which returns the

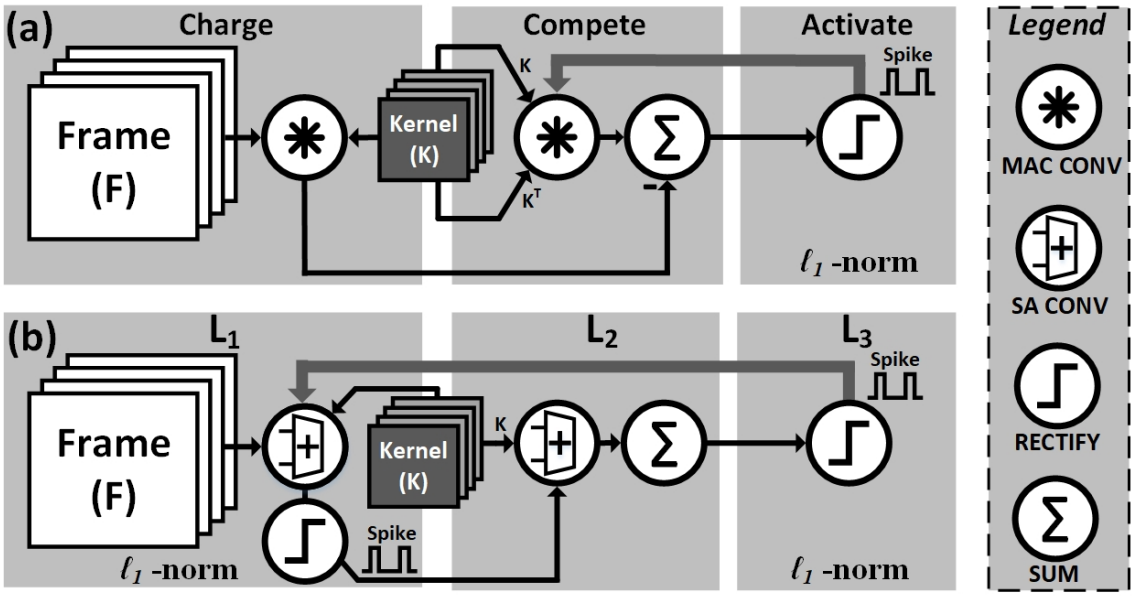


Figure 2.3: (a) Baseline 3-stage charge-compete-activate inference architecture, (b) Modified 3-layer recurrent inference architecture.

one-hot encoding of the spike locations, i.e., the addresses of the ST kernel memory to read. In the absence of spikes, an entire layer will be skipped, enabling an average 3.5× power reduction and 6.3× latency reduction. Dynamic clock gating is enabled by the OpenRISC processor based on the configuration profile to cut the dynamic power by 4.2× when switching from 192 to 64 neurons to adapt to problem requirements. The spike outputs of an L1 neuron are aggregated over 8 time steps to reduce dimensionality (Figure 2.5(d)), which is equivalent to a pooling operation in the time domain to compress data and reduce the latency of downstream processing. An L3 neuron's outputs are encoded using the compressed column storage format (Figure 2.5(c)). Due to sparsity, the compression results in 64 to 84% reduction in intermediate data storage.

ST kernels are quantized to 8 bits (Figure 2.6(a)), and their storage requires 108KB, occupying 2.5mm^2 area in 40nm CMOS. We observe that the pixel value difference for 95% of the time-adjacent ST kernels vary within 4 LSB (Figure 2.6(b)). Therefore, we apply non-uniform delta coding (Figure 2.6(c)) to compress ST kernels

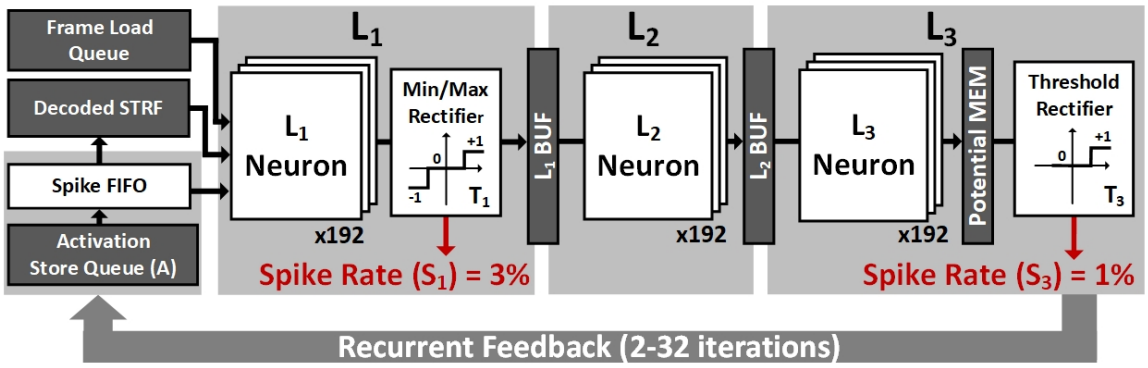


Figure 2.4: Spatio-temporal (ST) auto-encoder (core) implemented in a 3-layer recurrent network, consisting of configurable sparsity of outputs at different layers, and configurable feedback iterations.

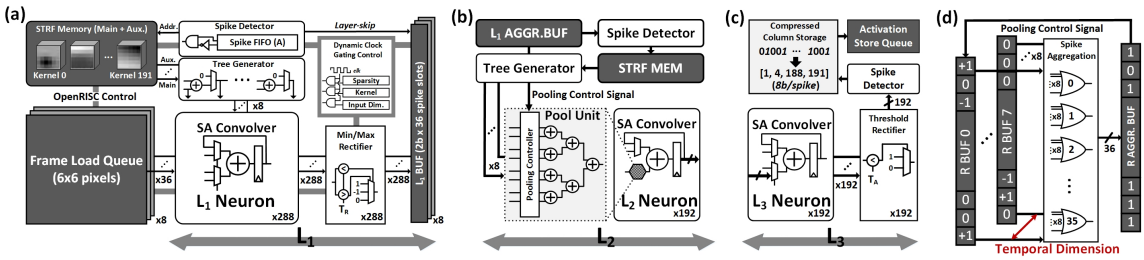


Figure 2.5: (a) Auto-encoder L1 implementation, (b) Auto-encoder L2 implementation, (c) Auto-encoder L3 implementation, (d) Temporal pooling of L1 output.

to 4 bits to reducing memory usage by 47.25KB and chip area by 43%. Prior to an ST convolution, ST kernels are reconstructed by a tree generator (Figure 2.6(a))

2.3 Chip Measurement and Classification Results

A 3.98mm$_2$ sparse ST cognitive SoC chip (Figure 2.7) is implemented in 40nm CMOS. The chip achieves an effective 1.63TOPS with 0.9V supply at 240MHz. The performance meets the 60fps 1920×1080 HD video data rate, while dissipating 127mW (Figure 2.8). The 6-class KTH human action dataset [48] is used for action classification testing (600 samples with train/test split ratio of 5:1). With the auto-encoder extracting the activation response of ST kernels, a softmax classifier implemented on

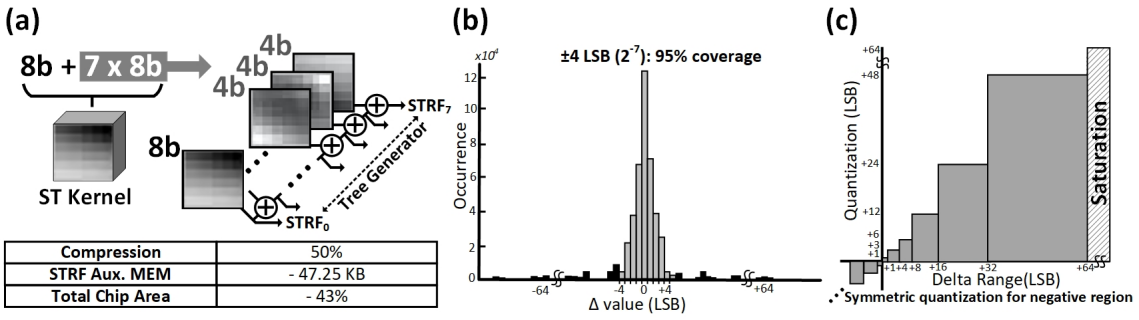


Figure 2.6: (a) Time-adjacent ST kernels are compressed to 4 bits, and reconstructed by a tree generator, (b) Histogram plot of all pixel value deltas for time-adjacent ST kernels, (c) Non-uniform delta coding quantization diagram.

the OpenRISC processor achieves a 76.7% classification accuracy. Using the same auto-encoder outputs, an off-chip SVM achieves an 82.8% accuracy (Figure 2.9). Motion tracking is also prototyped using a simple bounding box regression method based on the auto-encoder outputs. Compared to state of-the-art vision processors [49, 50], this design offers enhanced capabilities of action classification and motion tracking using a recurrent network. The design exploits sparse spikes to effectively reduce workload, demonstrating competitive performance and efficiency (Figure 2.10). The sparse spatio-temporal SoC is suitable for a range of cognitive processing tasks.

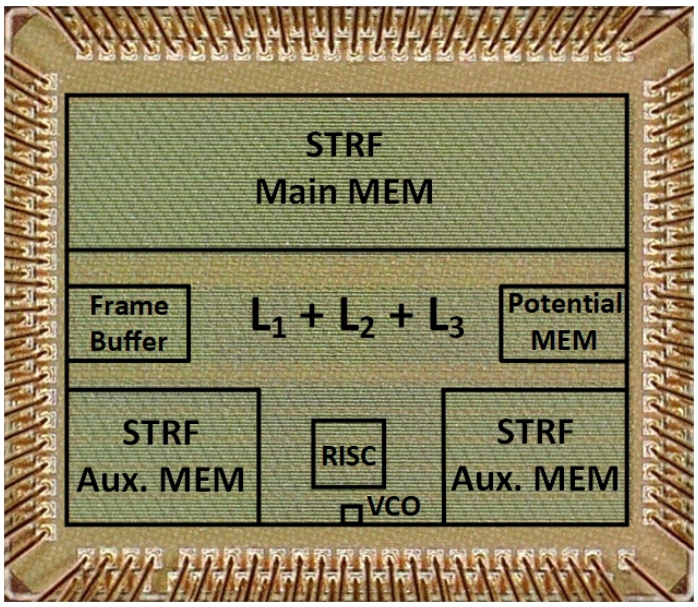


Figure 2.7: Packaged chip microphotograph.

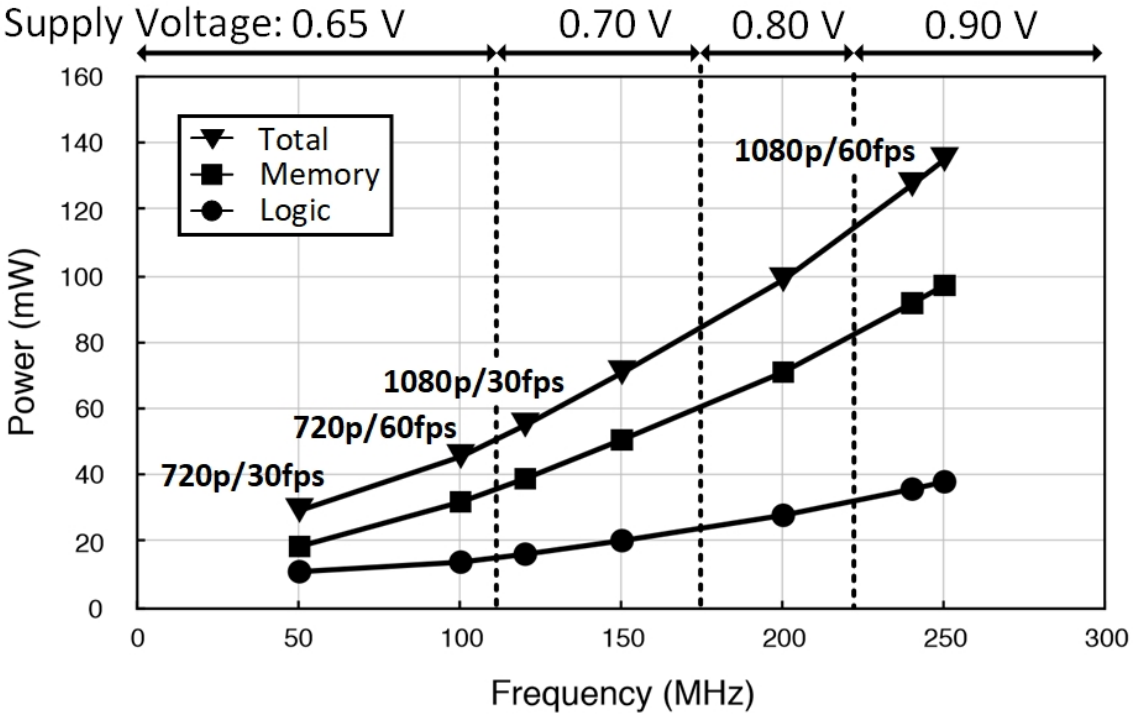


Figure 2.8: Measured power and frequency at room temperature.

TABLE I: COMPARISON OF CLASSIFICATION ACCURACY

Classes (individual classification accuracy)							
Algorithm	Box	Clap	Wave	Jog	Run	Walk	**Total**
On-Chip Softmax	70.0%	68.4%	85.0%	73.7%	94.4%	70.0%	**76.7%**
Off-Chip SVM	85.0%	78.9%	85.0%	73.7%	94.4%	80.0%	**82.8%**

Figure 2.9: Comparison Of classification accuracy.

Reference	VLSIC'16 Suleiman [6]	VLSIC'16 Knag [7]	This Work
Application	Multi-Object Detection	Object Recognition	Action Classification Motion Tracking
Topology	Deformable Parts Model	Deep Neural Network	Sparse Convolutional Auto-Encoder
Technology	65 nm	40 nm	40 nm
Area	16.0 mm^2	1.4 mm^2	3.98 mm^2
Voltage	0.77 – 1.11 V	0.65 – 0.90 V	0.65 – 0.90 V
Frequency	62.5 – 125 MHz	120 – 240 MHz	50 – 250 MHz
Power	58.6 – 216.5 mW	40.9 – 140.9 mW	29.2 – 134.93 mW[(a)]
Frame Rate[(b)]	30 – 60 fps	N/A – 30 fps	30 – 60 fps
TOPS[(b)]	0.068 – 0.137	0.449 – 0.898[(c)]	0.815 – 1.630[(d)]
TOPS/W[(b)]	1.169 – 0.624	10.98 – 6.37[(c)]	14.818 – 12.835[(d)]

Figure 2.10: Comparison with prior work; (a) Power is 127mW at 240MHz (60fps 1920x1080p HD video data rate), (b) Frame size is 1920x1080p HD video, (c) 1 OP is defined as an 8b multiply or a 16b add, (d) 1 OP is defined as an equivalent 8b multiply-accumulate (MAC).

CHAPTER III

Deep Neural Network Accelerator

Deep learning [2] has emerged to be a key approach to solving complex cognition and learning problems. Deep neural networks (DNNs) in particular have become pervasive due to their successes across a variety of applications, including image recognition [16–19, 51], object detection [4, 20], semantic segmentation [21, 52, 53], language translation [54], audio synthesis [55] and autonomous driving [56]. State-of-the-art DNNs [4, 16–21, 51–53] require up to billions of operations and hundreds of megabytes to store activations and weights. Given the trend towards even larger and deeper networks, the ensuing compute and storage requirements will prohibit any real-time, low-power deployment. This challenge has motivated efforts in building commercial DNN accelerators [57–59][1].

The core computation behind a DNN is the dot product of input activations and weights. Motivated by the potential performance and energy efficiency gains of specialized hardware [60–65], many prior works, from both research prototypes and industrial products, propose specialized hardware to accelerate dense DNN processing [57–59, 66–78]. The efficiency of a DNN accelerator is, to a large extent, determined by the amount and pattern of memory traffic. Eyeriss [69] proposed a dataflow taxonomy that categorizes different accelerators based on which type of

[1] Jie-fang Zhang contributed to the architecture, chip implementation and tapeout and Chester Liu for algorithm and implementation consultation

data, i.e., weight, input activation, or partial sum is reused over time to illuminate the traffic patterns and the efficiency of dense convolution architectures.

Recent research has shown that techniques such as quantization, pruning and re-training can zero out a large number of weights from a DNN without affecting classification accuracy [23, 79]. Input activations to each layer of DNN are also likely to be sparse due to the commonly-used rectified linear unit (ReLU) that clamps all negative values to zeros. Recent work has proposed sparse DNN accelerators architectures to make use of these types of sparsity to obtain higher performance and better efficiency: Cambricon-X [26] exploits zeros in weights, Cnvlutin [80] exploits zeros in input activations, and EIE [27] and SCNN [28] exploit zeros in both weights and input activations for fully-connected layers and convolution layers, respectively.

The key to exploiting sparsity is to align the non-zero weights and activations such that they can be multiplied together and then accumulated. The architectures that employ only weight or activation sparsity typically perform the alignment by using the indices of the sparse operand to index the dense operand, retaining the regular accumulator structure of the dense accelerators. However, such a simple indexing scheme does not work when both operands are sparse.

SCNN is the first attempt at exploiting both weight and input activation sparsity to improve the performance of sparse convolution computation. By adopting a Cartesian product-based dataflow, SCNN avoids the index matching complexity but suffers severely from the large partial sum traffic, leading to high energy cost in the multi-banked accumulation buffer design and low multiplier utilization due to bank conflicts. In addition, Cartesian Product-based dataflow requires the multiplier array to execute input activations and weights of the same input channel dimension, i.e., the C dimension, every cycle. With fully-connected and 1×1 convolution layers, the amount of unique data within a single input channel is quite limited, especially in the presence of sparse data. As a result, such an architecture cannot be used to ac-

celerate fully-connected layers and achieves less than 20% multiplier utilization when accelerating 1×1 convolution layers [28].

In this work, we propose the Stitch-X architecture that aims to exploit *both* weight and activation sparsity and accelerate *both* convolutional and fully-connected layers while mitigating limitations of prior sparse architectures. Specifically, Stitch-X makes the following contributions:

- We highlight that although Cartesian Product-based dataflow avoids the sparsity handling logic to align the non-zero weights and activations, it suffers severely from scattered partial sum accumulation in SRAM, leading to high energy cost and low multiplier utilization.

- We propose Stitch-X, a novel sparse DNN accelerator with a dynamic Parallelism Discovery Unit (PDU) and a multi-level, spatial-temporal reduction mechanism that efficiently accelerate a diverse range of Deep Neural Network layers with both sparse input activations and weights.

- We prototype Stitch-X architecture in RTL and evaluate it over a suite of modern DNNs [16–19]. Our evaluations demonstrate that Stitch-X achieves up to 4.3× speedup compared to an efficient dense accelerator and 1.6× performance with 2.1× energy efficiency improvement over a state-of-the-art, Cartesian Product-based sparse DNN accelerator.

3.1 Background and Motivation

This section presents an overview of the fundamental computations in modern DNNs and the state-of-the-art sparse DNN accelerators. In particular, we take a deep dive on the trade-off of Cartesian Product-based dataflow. We end this section with a characterization of different reduction mechanisms in DNN hardware design.

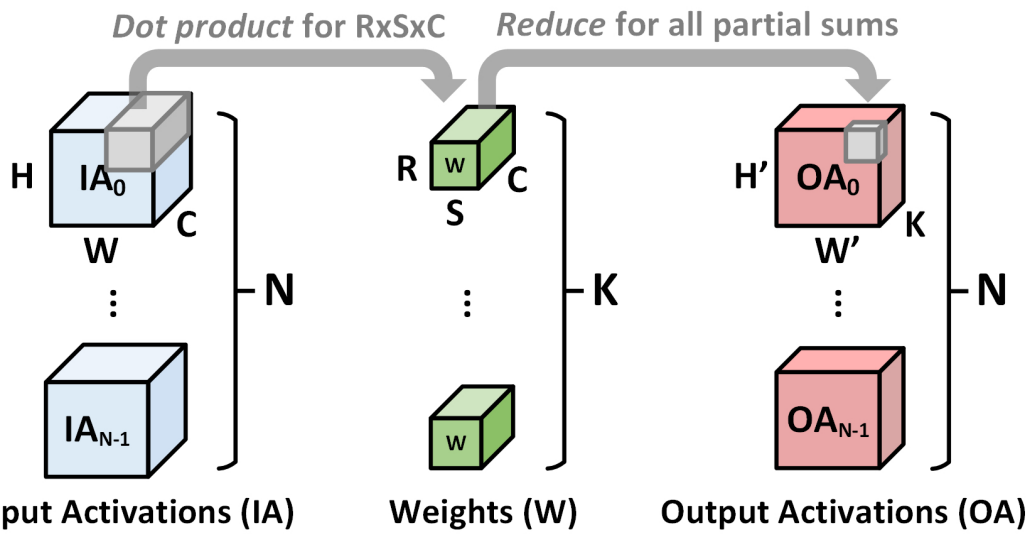


Figure 3.1: Input activation, weight, and output activation dimensions in DNN processing.

3.1.1 DNN Basics

Figure 3.1 shows a nested-loop representation of a convolution formulated as a nested loop over of an input activation (IA) array and a weight (W) array. The computation involves the generation and accumulation of partial sums, where the accumulation process is referred to as *reduction*. The same formulation also applies to fully-connected layers that are used in widely in multilayer perceptrons (MLPs) and recurrent neural networks (RNNs). This work focuses on accelerating *both* convolutional layers and fully-connected layers considering *both* input activation and weight sparsity.

Figure 3.2 illustrates the IA and W data densities, ranging between 41% and 67%, for four modern DNN networks after pruning with the techniques described in [79] and tested using Tensorflow [81] on ImageNet [3] without accuracy loss. The high sparsity in DNN has motivated recent work in DNN accelerator designs to efficiently skip computation and data movement with zero-valued data.

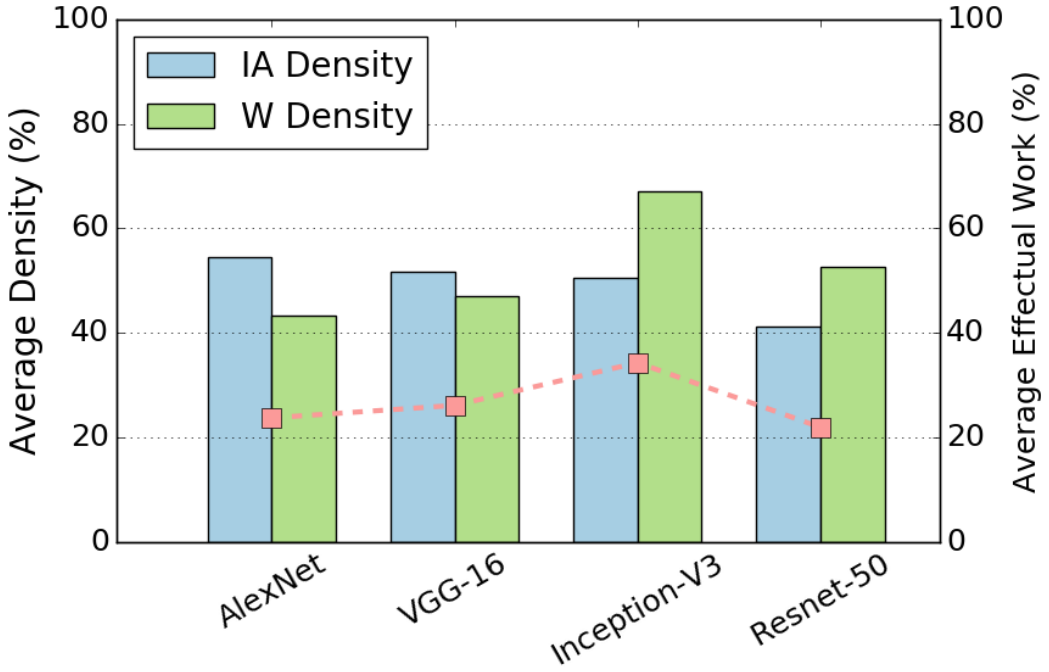


Figure 3.2: Average input activation, weight density and the effectual work after network pruning.

3.1.2 Index Matching in Sparse DNN

The fundamental challenge to exploit sparsity in both IA and W operands lies in the index matching problem, i.e., efficiently searching non-zero reducible pairs in parallel from sparse IA and W arrays. Reducible IA and W pairs produce partial sums that can be accumulated to the same final output. Most of the prior sparse DNN accelerators only skip computation that involves either zero-valued weights or input activations but not both, e.g., Cambricon-X [26] with sparse weight and Cnvlutin [80] with sparse input activation. By working with only one set of sparse operands, these approaches simplify the sparsity handling logic significantly but fail to fully benefit from skipping redundant computation and external memory movement for both operands. SCNN [28] is the first hardware accelerator that leverages sparsity in both input activations and weights in convolutional layers by adopting

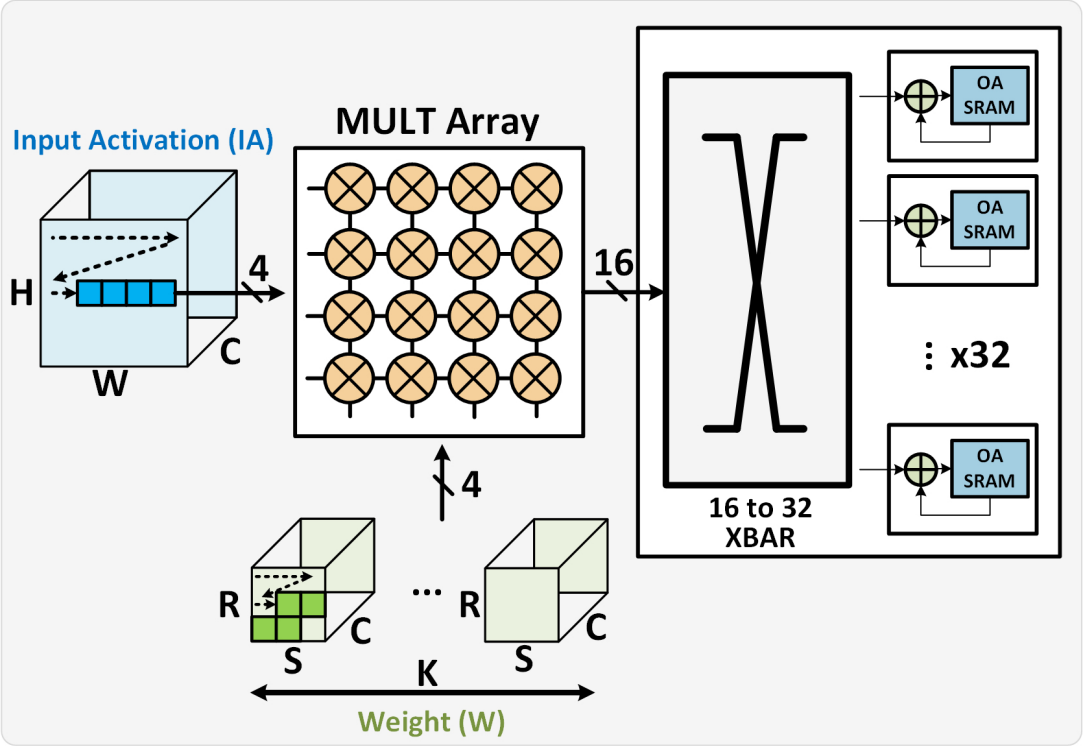


Figure 3.3: Cartesian Product-base dataflow in SCNN.

a Cartesian Product-based dataflow to avoid handling sparse index matching completely. We discuss the trade-offs in Cartesian product-based dataflow in the next section.

3.1.3 Cartesian Product-Based Dataflow

SCNN is the first DNN accelerator that adopts Cartesian Product-based dataflow to exploit data sparsity, as illustrated in Figure 3.3. It exploits a unique property of convolution that *all* the activations (IAs) need to be multiplied *once* with *all* the weights (W) of the same input channel to produce the final output, independent of the processing order. Leveraging this observation, Cartesian Product-based dataflow adopts an input-channel-last loop order, i.e., the input-channel dimension is the outer-most loop, and multiplies all the weights and inputs of the same input channel first. The advantage is that it requires no ordering between the loading of input IA and W

except that they need to have the same input channel since any pair of them can be multiplied together, making it appealing for sparse accelerator design. SCNN adopts this dataflow and multiplies all the combinations of 4 non-zero IAs and 4 non-zero Ws of the same input channel every cycle and produces 4×4 partial-sums.

However, the lack of alignment in non-zero IA and W inputs results in a large number of scattered partial sums produced every cycle, each of which has to be accumulated to a unique output address. This has led to two key limitations of Cartesian Product-based dataflow. First, it requires a high-bandwidth SRAM to accumulate all the scattered partial sums every cycle, causing high area and energy overheads. SCNN provisions a 32-bank accumulation buffer design with a 16×32 crossbar to accumulate 16 partial sums every cycle, taking more than 50% of the total accelerator area. Second, even with a multi-banked accumulation buffer, bank conflict still occurs frequently since partial sums are randomly scattered across the entire address space. Whenever a bank conflict happens, the entire processing pipeline stalls until the accumulation buffer absorbs all the partial sums, leading to low multiplier utilization. In addition, Cartesian Product-based dataflow cannot efficiently accelerate fully-connected layers as IA in fully-connected layer becomes a vector and there is only one data element per channel. As a result, SCNN only achieves a maximum of 25% multiplier utilization. Next section quantifies that there is at least $3\times$ energy efficiency loss in Cartesian Product-based dataflow.

3.2 The Stitch-X Dataflow

Reduction in convolution takes place in three dimensions, R, S and C. Different from the input-channel-*last* loop ordering in Cartesian Product-based dataflow, Stitch-X adopts an input-channel-*first* dataflow that *always* exploits cross-input-channel, i.e., cross-C, spatial and temporal reduction opportunity first. The reason is threefold. First, each layer typically has a deep C dimension, in the range of 64 to

2048, much deeper than R and S, which typically range from 1 to 7. Second, both convolutional and fully-connected layers can exploit cross-C reduction but cross-R/S reduction does not apply to fully-connected layers. Finally, cross-C reduction also simplifies the index matching problem, i.e., instead of doing two-dimension index matching for both R and S, it only requires a one-dimension index matching along C. Motivated by the above benefits, Stitch-X employs a multi-level, spatial-temporal, cross-C reduction dataflow that maximizes the use of SR and local TR while minimizing the use of expensive TR to SRAM.

3.2.1 Spatial and Temporal Reductions

As illustrated in Figure 3.1, each element in output activation (OA) requires $R \times S \times C$ accumulations, independent of how the nested loops are ordered. A reduction mechanism describes how these accumulations are done in hardware. Specifically, existing DNN architectures perform these accumulations either spatially or temporally:

Spatial Reduction (SR) performs partial-sum accumulation *spatially* without explicit storage during the reduction process. As illustrated in Figure 3.4(a), given T partial sums, SR is realized using an $T:1$ adder tree to produce an output in *one* time step (a single clock cycle). DianNao [66] and NVDLA [59] are examples of DNN architectures that adopt the SR approach. It requires all the T partial sums to be mapped to the *same* output but cuts the number of accesses to OA buffer by a factor of T, reducing OA buffer's bandwidth pressure and energy cost.

Temporal Reduction (TR) reduces over time by using a single adder to accumulate one partial sum per time step over T steps, shown in Figure 3.4(b). The advantage of TR is that it has no requirement on whether the T number of partial sums contribute to the same address, making it more flexible to handle irregular computation. As a result, accelerators with two sparse operands, e.g., EIE [27] and

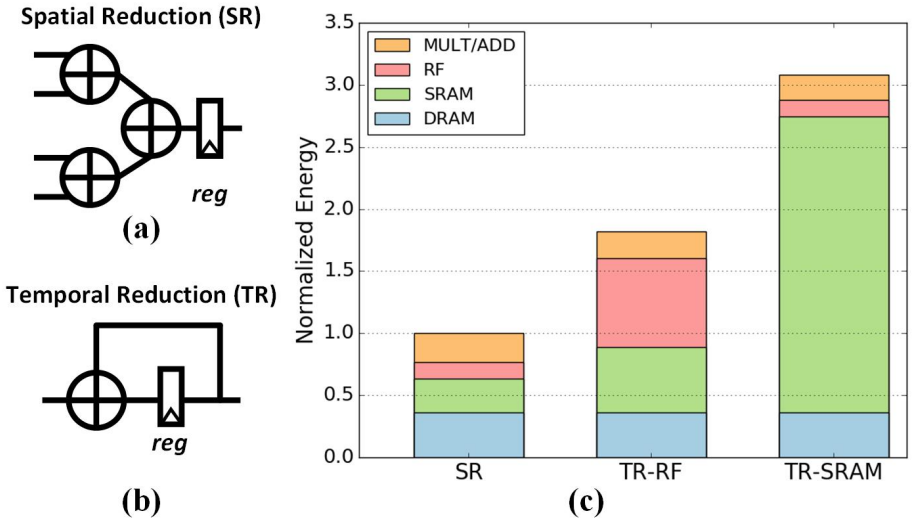


Figure 3.4: (a) Spatial reduction (SR). (b) Temporal reduction (TR). (c) Normalized Energy of DNN accelerators with Spatial Reduction (SR), Temporal Reduction to RF (TR-RF), and Temporal Reduction to SRAM (TR-SRAM).

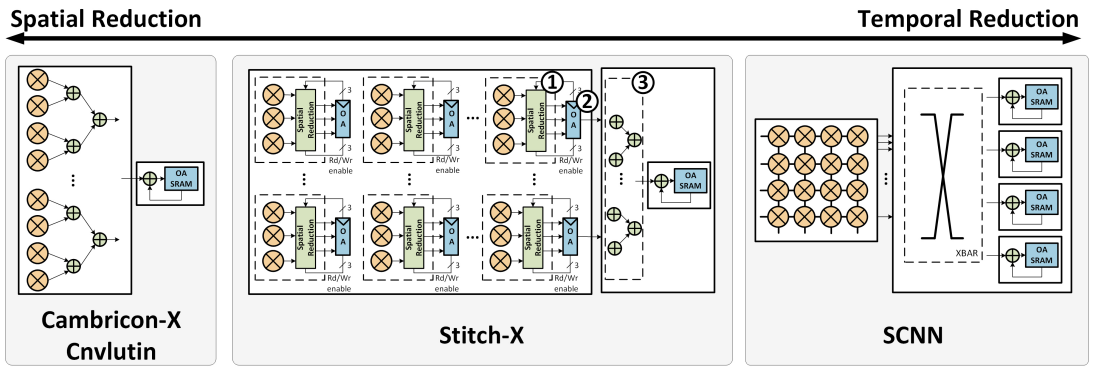


Figure 3.5: Reduction mechanisms in state-of-the-art, sparse DNN accelerators.

SCNN [28], tend to use TR. However, every TR requires two accesses, one read and one write, increasing the bandwidth requirement for the output OA buffer design.

We quantify the energy implications of different reduction mechanism in Figure 3.4(c). We compare three reduction mechanisms, 1) spatial reduction (SR), used in DianNao [66] and NVDLA [59], 2) temporal reduction to a register file (TR-RF), used in ShiDiaoNao [68] and EIE [27], and 3) temporal reduction to an SRAM (TR-SRAM) that Cartesian Product-based dataflow uses, e.g., SCNN [28]. To make a fair comparison, all the designs use the same data reuse pattern, i.e., input stationary,

the same size of W, IA, OA buffers, and the same number of multipliers. We also size the convolution dimensions based on the buffer size such that no data is refetched from DRAM. Hence, the only variable under test is the reduction mechanism. We see that SR is the most energy efficient due to its aggressive reduction of write-back accesses to OA buffers. TR-RF consumes 1.8× higher energy compared to SR, due to the additional RF read and write for partial sums. More importantly, we notice that TR-SRAM, the reduction mechanism used in SCNN [28], exhibits the highest energy cost at 3.1× SR. TR-SRAM has a significantly higher SRAM energy consumption since every partial-sum accumulation requires one read and one write access to an SRAM buffer, which has a much higher per access energy cost than a RF.

As a result, although Cartesian Product-based SCNN achieves a 2.3× energy reduction compared to its dense baseline — a TR-SRAM-based design [28], it is still not as efficient as a dense, spatially-reduced accelerator due to the high energy penalty associated with temporal reduction to SRAM. Such a high overhead is inevitable for Cartesian Product-based dataflow because the lack of alignment in IA and W makes a large number of scattered partial sums be reduced temporally in SRAM.

Understanding the intrinsic limitations of Cartesian Product-based dataflow motivates us to rethink the balance between costs of index matching and reduction. The key is to devise 1) a reducible dataflow that maximizes the use of SR and local TR before partial sums reach to OA SRAM, and 2) a low-cost index-matching mechanism that aligns reducible non-zero IA and W together before it reaches multipliers. We discuss how Stitch-X addresses each of the two challenges in Section 3.2 and Section 3.3

3.2.2 Multi-Level Reduction

Figure 3.5 illustrates the spectrum of reduction mechanisms adopted by today's sparse DNN accelerators. On one end of the spectrum, sparse DNN accelerators that

leverage only one type of input operand sparsity, i.e., Cnvlutin [80] and Cambricon-X [26], maximizes the use of SR through a wide adder tree to deliver better performance and energy efficiency. In such cases, it is feasible to use the indices of the sparse operand to directly index the dense one while keeping the wide adder tree fully utilized. However, attempting to leverage sparsity in both operands exposes the challenge of finding a sufficient number of reducible W-IA pairs every cycle. SCNN took the liberal approach of letting independent partial sums be reduced by TR to SRAM, leading to significant performance and energy penalties due to excessive SRAM accesses.

Stitch-X applies a multi-level reduction approach that maximizes the use of SR and TR-RF. As illustrated in Figure 3.5, three levels of SR and TR are employed by Stitch-X:

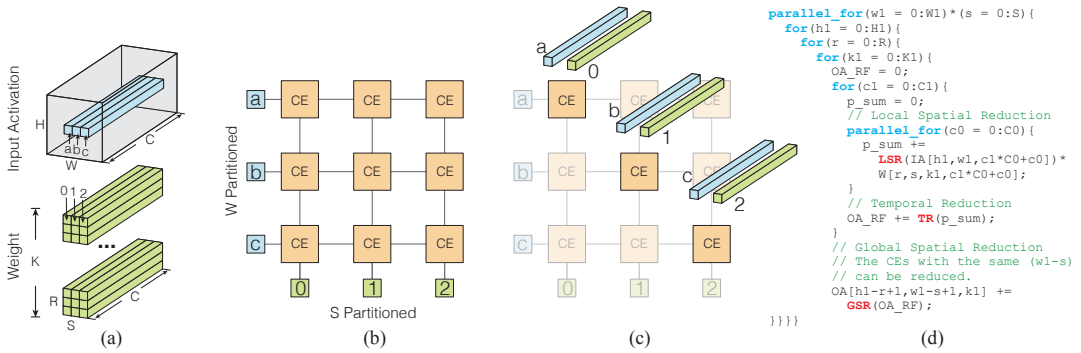


Figure 3.6: $R \times S \times C$ convolution. (a) Input Activation (IA) of size $H_1 \times W_1 \times C$ and Weight of size $K_1 \times R_1 \times S_1 \times C$. (b) IA is partitioned in its W dimension and multi-casted in each row of CEs, while Weight is partitioned in its S dimension and multi-casted in each column of CEs. (c) Each CE receives a slice of IA and Weight, both of size C, and reduces them locally. The partial sums from the highlighted CEs in diagonal direction can be further reduced globally since they contribute to the same output. (d) Stitch-X's Dataflow, with $C = C_0 \times C_1$.

1. **Local SR**. 1 in Figure 3.5 shows the first-level SR employed by Stitch-X to perform cross-C reductions. We choose a $3:1$ adder tree to balance multiplier utilization and reduction benefit.

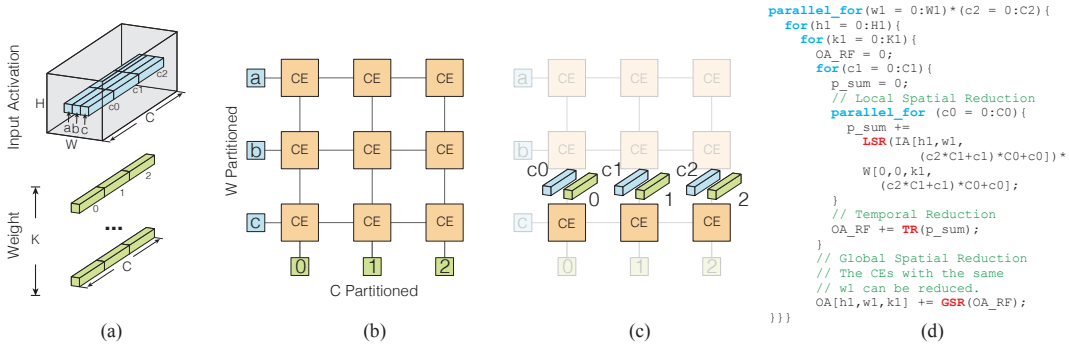


Figure 3.7: $1 \times 1 \times C$ convolution. (a) IA is still of size $H_1 \times W_1 \times C$, but Weight size is $K_1 \times C$ as R and S are one. (b) IA is partitioned in its W and C dimensions while Weight is partitioned in its C dimension. Each slice of Weight is multi-casted in column while each slice of IA is uni-casted to its corresponding CE. (c) Each CE receives a slice of matching IA and Weight of size $C/3$. The highlighted CEs can be reduced globally. (d) Stitch-X's Dataflow, with $C = C_0 \times C_1 \times C_2$.

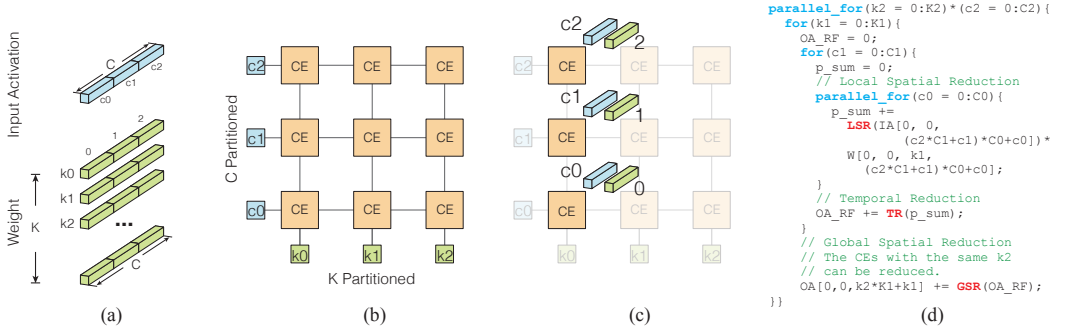


Figure 3.8: Fully-Connected Layers. (a) IA becomes a vector of size C while Weight is a matrix of size $K \times C$. (b) IA is partitioned in its C dimension while Weight is partitioned in its K dimension and C dimensions. (c) Each CE receives a slice of matching IA and Weight of size $C/3$. The partial sums from highlighted CEs can be further accumulated globally. (d) Stitch-X's Dataflow, with $C = C_0 \times C_1 \times C_2$, $K = K_1 \times K_2$.

2. **Intermediate TR**. 2 shows the second-level TR-RF employed by Stitch-X to accommodate the irregular sparsity in the operands. Although TR-RF not as efficient SR, it relaxes the throughput requirement of the index matching logic.

3. **Global SR**. 3 shows the third-level SR employed by Stitch-X to perform spatial reduction globally. It further reduces the number of accesses to OA SRAM by grouping the RF write-back traffic streams that go to the same SRAM address.

3.2.3 Mapping DNN layers to Stitch-X

The multi-level reduction dataflow of Stitch-X exploits cross-C reduction that is common to all types of layers, i.e., $R \times S \times C$ convolutions, $1 \times 1 \times C$ convolutions, and fully-connected layers. Specifically, Stitch-X adopts a *PlannarTiled-InputChannel(C)-KernelWidth(S)-Spatial* dataflow for $R \times S \times C$ convolutions and a *PlannarTiled-InputChannel(C)-Spatial* dataflow for $1 \times 1 \times C$ convolutions and fully-connected layers.

R × S × C convolution is one of the most commonly used convolution layers in modern DNNs. Figure 3.6 shows an example of how data is streamed to Stitch-X's Compute Elements (CE) for $R \times S \times C$ convolution. Input activation (IA) is of size $H_1 \times W_1 \times C$, and weight (W) is of size $K_1 \times R_1 \times S_1 \times C$, shown in Figure 3.6(a). We choose PlannarTiled-C-S-Spatial dataflow here because it maximizes the spatial reduction opportunities across both the input-channel (C) and kernel-width (S) dimensions.

Specifically, Figure 3.6(b) illustrates Stitch-X partitions the image width (W) dimension of IA across rows of CEs and the S dimension of W across columns and streams a slice of IA and W to each CE. Taking the CE on the upper left corner as an example, Figure 3.6(c) shows that it gets the slice a of size C from IA and the slice *0*, also of size C, from W. The data from the IA and W slices can be multiplied and reduced together, both spatially and temporally in this CE, to a single piece of output

data. In addition, the final OA outputs from the highlighted CEs in Figure 3.6(c) can be further spatially reduced diagonally so that only one output needs to be written back to the OA SRAM. Figure 3.6(d) illustrates this dataflow in a nested-loop form. CEs in Stitch-X process slices of IA and W across W and S dimensions in *parallel*. Inside each CE, it first multiplies and *spatially* reduces chunks of data across C_0 in *parallel* followed by a TR-RF across C_1. Partial sums for the same output can be further reduced *spatially* across the S dimension cross-CE.

$1 \times 1 \times$ C convolution is also widely used in newer networks like ResNet-50 [18] and Inception layers [19]. Figure 3.7(a) shows the dimensions for $1 \times 1 \times C$ convolution. The IA dimension is the same as $R \times S \times C$ convolution, but W becomes a two-dimensional matrix of size $K_1 \times C$ since both R_1 and S_1 are one. We apply PlannarTiled-C-Spatial dataflow for $1 \times 1 \times C$ convolution as C is the only available reducible dimension in the algorithm.

Figure 3.7(b) shows how Stitch-X partitions IA and W across CEs. IA is partitioned in its W dimension across rows of CEs, and both IA and W are partitioned in C dimension across columns of CEs. Figure 3.7(c) demonstrates which pieces of IA and W are delivered to the bottom row of CEs. Each CE gets a slice of matching IA and W of size $C/3$ which can be locally reduced to a single output. In addition, the outputs from the highlighted CEs can be reduced globally before writing the final output to OA SRAM. Figure 3.7(d) illustrates the dataflow in more details.

PlannarTiled-C-Spatial dataflow splits the slice along C dimension into multiple sub-slices that are processed in CEs in *parallel*. In this case, CEs with the same w_1 can be globally reduced across the C_2 dimension.

Fully-connected layers are commonly used in multi-layer perceptrons (MLPs), recurrent neural networks, and the end of convolutional neural networks for classification and prediction.

Different from SCNN [28] that cannot support fully-connected layers, Stitch-X

continues exploiting cross-C reductions in matrix-vector multiplication. Figure 3.8(a) illustrates the IA and W dimensions for fully-connected layers. In this case, IA is a one-dimensional vector of size C, and W is a two-dimensional matrix of size $K_1 \times C$. Similar to $1 \times 1 \times C$ convolution, we also apply PlannarTiled-C-Spatial dataflow to fully-connected layer execution as C is also the only reducible dimension. The only difference is that CEs processes parallel slices of IA and Weight in C and K dimension instead of C and W because IA becomes a vector.

Figure 3.8(b) shows how Stitch-X partitions the IA vector and W matrix across rows and columns. Taking the CE on the bottom left corner as an example, Figure 3.8(c) shows that it gets the *c0* slice from IA and *0* slice from W. Both slices are of size $C/3$ and can be reduced cross-C into a single output. Similar to the other two cases, the outputs of the highlighted CEs can also be reduced globally into one final output. Cross-C reduction still starts with local spatial reduction within a CE and ends with a cross-CE, global spatial reduction.

3.3 The Stitch-X Accelerator

The second key innovation in Stitch-X is a low-cost, look-ahead Parallelism Discovery Unit (PDU) that dynamically aligns non-zero IA and W pairs. This section introduces the full Stitch-X architecture and details how PDU works to efficiently find reducible IA and W pairs.

3.3.1 Architecture Overview

Figure 3.9 shows the top-level diagram of the Stitch-X architecture consisting of compute, control, and memory modules.

The **compute module** contains an array of Compute Elements (CEs), Parallelism Discovery Units (PDUs), and a Global Spatial Reduction Unit. Specifically, Figure 3.10a shows the structure of the compute module and the microarchitecture

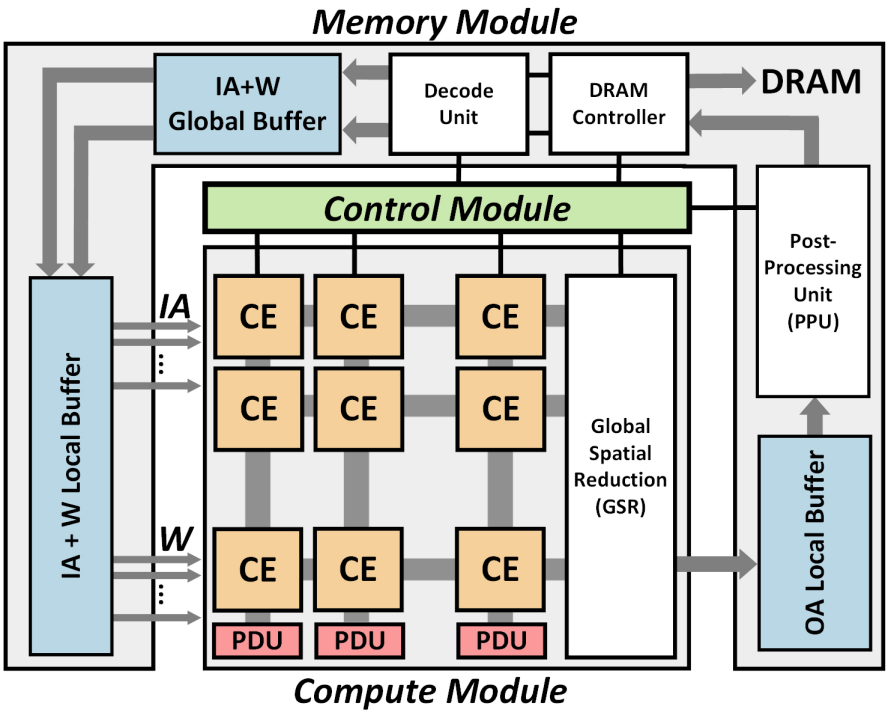
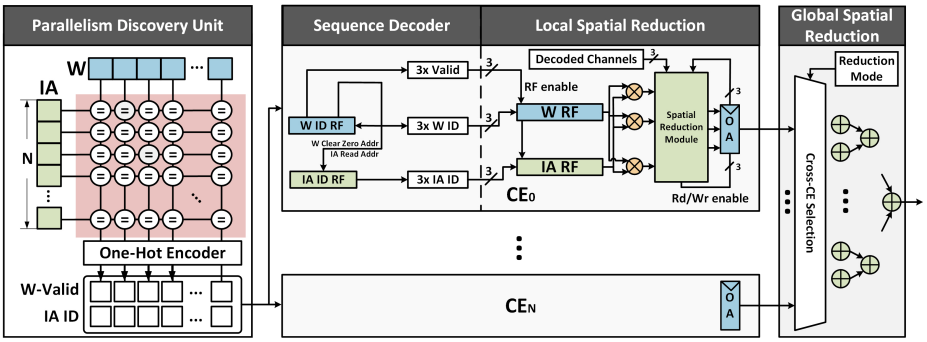


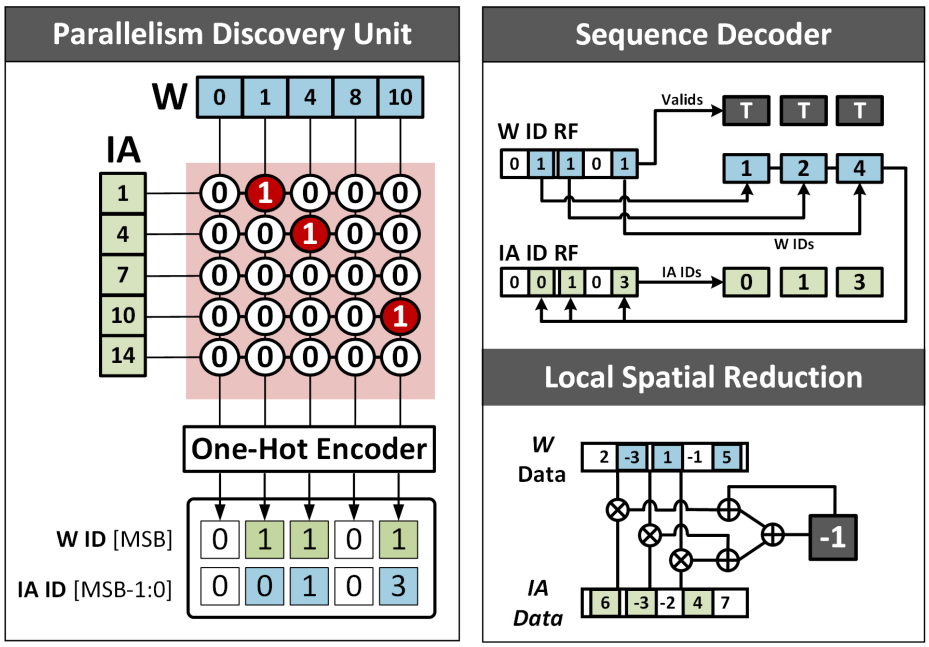
Figure 3.9: Stitch-X microarchitecture overview.

of a CE. Each CE includes three multipliers, a $3:1$ SR tree, and TR register file to support local SR and intermediate TR. Each column of CEs share a PDU to find IA and W pairs of the matching input channel C. The CEs are connected via multiple parallel horizontal and vertical data buses that distribute IA and W operands.

The **memory module** includes SRAM to store IA, W, and OA on-chip and a DRAM controller to access off-chip DRAM. Global buffers store blocks of the IA and W arrays, each of which are transferred successively to a multi-banked IA and W local buffer for processing. Each bank of the IA and W local buffer supports data distribution to a row or a column of the CEs. The banked local buffers accommodate different data layouts needed to facilitate processing different DNN layers. Stitch-X adopts a simple run-length-encoding (RLE) mechanism to store compressed sparse IA and W. A Post-Processing Unit (PPU) applies activation functions, *e.g.* ReLU, and compress OA with the same RLE before transferring it back to DRAM.



(a) Stitch-X's Compute Module. Each column of Compute Module includes a Parallelism Discovery Unit (PDU) and an array of Compute Elements (CEs). The partial sums of CEs are further spatially reduced via the Global Spatial Reduction unit.



(b) Example of PDU encoding and the CE's sequence decoding and local spatial reduction process.

Figure 3.10: Finding reducible IA and W pairs in Stitch-X.

The **control module** contains three controllers for execution, CE buffer, and writeback. The execution controller orchestrates the input operand streaming from the memory module to the compute module and the OA local buffer writeback to DRAM. The CE buffer controller fetches and delivers operands that are required for computation. The writeback controller determines the writeback address of partial sums once the reduction is complete. The writeback address is determined by

performing simple arithmetics on the IA and W indices.

3.3.2 Finding Reducible Pairs

With the Stitch-X dataflow, the index matching problem amounts to finding all reducible pairs of non-zero IA and W elements of the same input-channel, i.e, C, dimension. Predetermining the interactions between the two arrays is impossible as the IA density is statically unknown. Thus, we propose a low-cost, dynamic PDU that efficiently performs parallel search on IA and W arrays to find matching IA and W pairs of the same C index. PDU enables Stitch-X to fully exploit the cross-C reduction opportunity and achieve high multiplier utilization with low area and energy cost.

PDU finds matching IA and W pairs in two steps: encoding and decoding. PDU first takes the input channel indices of IA and W stored in the input-channel-first, i.e., C-first order and finds the matched IA and W indices, sending them to the target CE. To decode, a Sequence Decoder in CE decodes the sequences from PDU to extract reducible IA and W pairs for computation. Since the throughput of the PDU, i.e., N per cycle, is much higher than the CE's throughput, i.e., 3 per cycle, a PDU can be shared across multiple CEs. We discuss the encoding and decoding processes in more details in the following sections.

3.3.2.1 PDU Encoding

The left part of Figure 3.10a illustrates the microarchitecture of a PDU design. A PDU operates on a block of N pairs operands at the same time, where it receives the input-channel index vectors of IA and W, both of size N, as inputs. Internally, the PDU uses an $N \times N$ comparator array to search for matching IA and W channel indices in parallel. The comparator array produces a binary output at each junction, 1 for match and 0 for mismatch. For each column, *i.e.*, a unique weight W, at most

one matching IA can be found, since each weight is multiplied with a unique IA for the target OA.

An one-hot encoder is connected to each column of the comparator array to find the row address of the matched IA. Each one-hot encoder produces a valid bit to indicate whether a match is found and an IA index (IA-ID) to index the matched IA. The encoded W-Valid and IA-ID sequences are sent to a Sequence Decoder in each CE to index the reducible IA and W pairs.

3.3.2.2 Sequence Decoding

As Figure 3.10a shows, a Sequence Decoder in a CE decodes the encoded sequence generated by the PDU and obtains the matching pair of IA and W for multiplication. Depending on the local SR reduction factor r, the Sequence Decoder scans the W-valid sequence to find r valid signals that are used directly as the read enable signals to access the W and IA data registers. At the same time, the positions of these valid signals are also 1) the positions of the matched W since if the weight is valid, there is a matched IA found for that W; and 2) the positions in the IA-ID sequence to locate the matched IA for the corresponding W.

3.3.2.3 An Example

Figure 3.10b shows an example of PDU encoding and decoding, where the PDU encoder width, $N = 5$ and the decoding way, $r = 3$. Starting from the left side of the figure, the PDU first loads N input channel indices of W and IA since we always look for cross-C reduction opportunity first. The array of comparators in the PDU compares W's and IA's input-channel indices, producing a 1 (match) for each matching pair. The one-hot encoder in each column checks the outputs of comparators in the same column and produces two values: the first is a one-bit value indicating whether a match is found; the second is the one-hot encoded IA ID indicating which

IA matches the weight in that column. Taking the first column as an example, since no matching IA is found for the weight of input channel 0, the W-valid bit of the first column is 0. For the second column with the weight from input channel 1, the PDU finds the first IA is also from input channel 1. In this case, W-valid is set to 1 and the matched IA-ID is 0, indicating row zero is the match. In this particular example, out of the five Ws and IAs that the PDU checks, it identifies three matching (W-ID, IA-ID) pairs: (1,0), (2,1), and (4,3) and encodes the results into encoded W-Valid, IA-ID sequences.

The encoded sequences are sent to a Sequence Decoder. The decoder first checks the W-valid sequence to find three valid signals — in this case, it finds the 1st, 2nd, and 4th (counting starts from 0) positions contain valid entries. These are the positions from which we should fetch weight data. In addition, these are also where we should check the IA-ID sequence to find the corresponding IA indices for the matched weight. In the example in Figure 3.10b, using the valid positions, the Sequence Decoder successfully decodes the matched W and IA pairs. This simple example only shows a unique IA-W pair per IA, i.e., there is a unique weight for every IA. However, when we run large networks, it is possible that there are multiple weights of the same input-channel (C) index but different output channel (K) indices mapped to the same IA. A unique one-hot encoding of IA-ID per weight is required to differentiate different mapped pairs.

3.3.2.4 PDU Scalability

Both PDU encoding the decoding are easily scalable, *i.e.*, given N IA and W input-channel indices, the PDU can produce encoded sequences of length N every cycle. To scale the decoding, we make the Sequence Decoder search for r valid signals in the W-Valid sequence, so that a maximum of r reducible pairs can be obtained and reduced using a $r : 1$ local SR tree in one cycle. In addition, by ensuring that

Table 3.1: Stitch-X area breakdown.

	Area (mm^2)	Percentage
PDU (N = 64)	0.118	4.3%
Sequence Decoder	0.090	3.3%
MULT (8x3x3=72)	0.032	1.2%
Local SR (3:1)	0.051	1.8%
Global SR (8:1)	0.160	5.8%
OA Local Buffer (4 KB)	0.099	3.6%
IA+W Local Buffer (16 KB)	0.174	6.4%
Global Buffer (128 KB)	1.100	40.0%
Register File (28 KB)	0.843	30.7%
Control	0.081	2.9%
Total	2.748	100%

$N \gg r$, we can make PDU a centralized module shared between multiple CEs, so that each CE requests to the PDU in a time-multiplexed fashion. The actual N and r parameters depend on the buffer bandwidth, throughput requirements and physical constraints of the design.

3.4 Experimental Methodology

We implemented the Stitch-X accelerator architecture in SystemVerilog and synthesized it using Synopsys Design Compiler at a 1.0 GHz clock frequency in a commercial 40 nm CMOS technology. We also constructed a cycle-accurate architectural performance and energy model to explore the design space and evaluate the performance of Stitch-X running real-world DNNs.

Architecture model. We constructed a cycle-accurate performance and energy model based on detailed hardware characterization. We use the performance model for three purposes: 1) to evaluate the performance of the Stitch-X prototype running representative DNNs with IA and W extracted from Tensorflow to faithfully capture the sparse data patterns; 2) to explore the design space of Stitch-X by sweeping

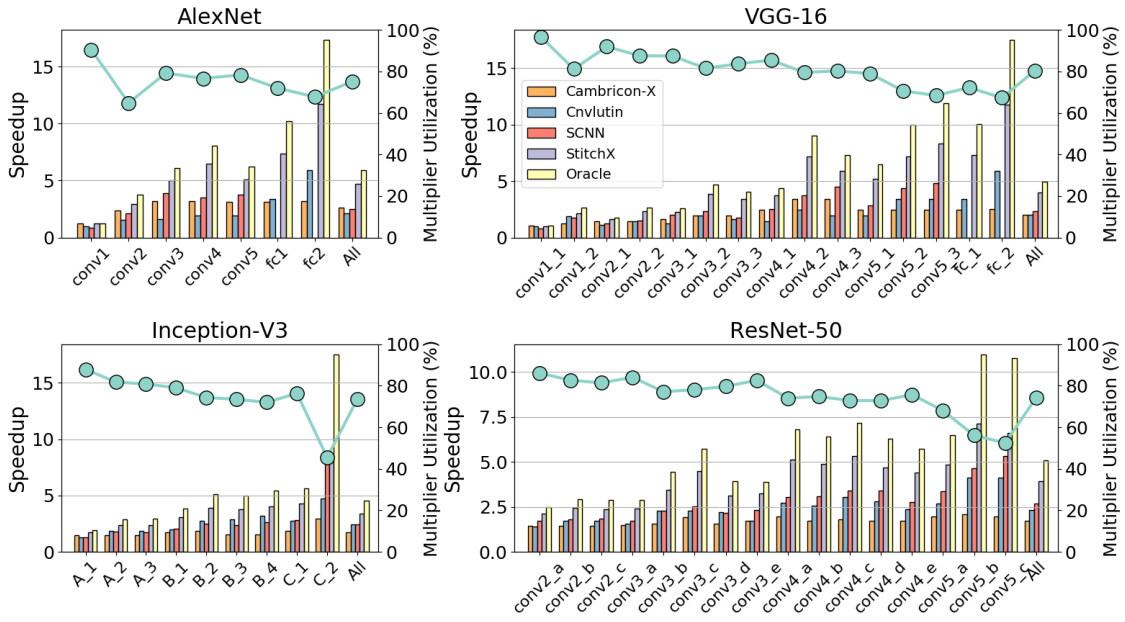


Figure 3.11: Overall performance improvement of Stitch-X compared with Cambricon-X, Cnvlutin, and SCNN, running a range of modern DNNs: (a) AlexNet, (b) VGG-16, (c) Inception-v3, and (d) ResNet-50. Performance is normalized to a dense SR accelerator baseline. The multiplier utilization of Stitch-X is plotted on the right Y-axis.

hardware parameters, e.g., on-chip SRAM capacity, bandwidth, PDU encoding width, and register file sizes; and 3) to generate runtime activity counts of operations and accesses to memory modules. These counts are inputs to our loop-based energy model to obtain the energy consumption of executing a DNN layer. We build the energy model following the same methodology described in [69, 82] with a detailed energy characterization of hardware components of Stitch-X, including SRAMs, RFs, multipliers, adder trees, local and global SR, and PDU, based on RTL synthesis results.

Design space exploration. We use the performance-energy model to explore different design choices for Stitch-X. Specifically, we explore three aspects of Stitch-X design: 1) compute throughput, including reduction factors, both locally and globally, and the dimensions of the CE array, 2) memory sizes, including on-chip SRAM sizes

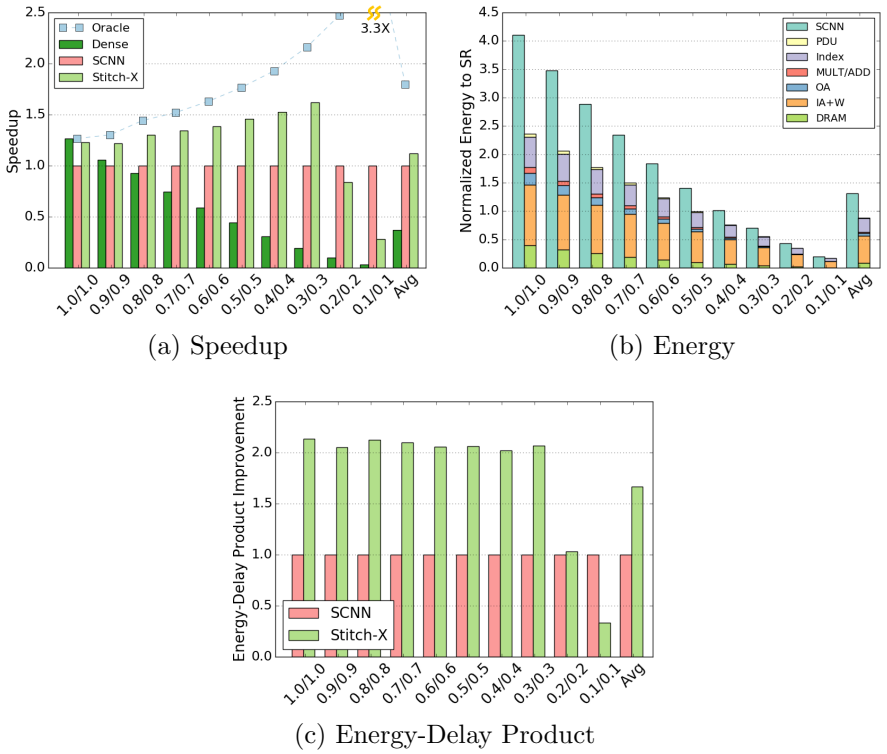
and their bandwidth, and 3) PDU throughput, including the width of PDU and how a PDU is shared across CEs. We explore different design parameters with our performance and energy model and evaluate a wide range of real-world networks (Section 3.5.1). Table 4.1 lists the specific parameters we choose in our design that delivers high energy efficiency under the area envelop.

In addition, to quantitatively evaluate Stitch-X performance compared with other architectures, we extend our modeling framework to capture the designs of other sparse accelerators, e.g., SCNN, Cambricon-X, and Cnvlutin, validated our models against the published results.

Hardware implementation. Table 4.1 shows the area breakdown of the key components in the final Stitch-X implementation. Stitch-X's compute module consists of an 8×3 grid of CEs, a PDU per column of CEs, and a global spatial reduction (GSU) unit. Each CE has three 16-bit multipliers and a $3:1$, 48-bit adder tree for local SR. A 1.15 KB register file is allocated per CE for storing IA, W, their coordinates, and the PDU encoded sequence. The global SR is done by the GSU that includes a 4 KB register file to store reduced partial sums. The parallel search overhead of Stitch-X, i.e., PDU and sequence decoder, together occupy only 7.6% of the total area, much lower than the decoding overheads reported in previous work [26].

3.5 Evaluation

We first demonstrate Stitch-X's performance by evaluating over real-world DNN workloads, compared with state-of-the-art sparse accelerators. We then analyze the performance and energy efficiency of Stitch-X at different IA and W densities. Finally, we discuss two important design parameters and their impacts on Stitch-X architecture: OA buffer bandwidth and PDU encoder width.



(a) Speedup

(b) Energy

(c) Energy-Delay Product

Figure 3.12: (a) Performance comparison of Stitch-X, SCNN, and Oracle. (b) Energy comparisons of Stitch-X and SCNN. (c) Energy-Delay Product Improvement of Stitch-X over SCNN. Energy is normalized to the same efficient dense baseline with SR. X-axis indicates W and IA densities. We choose a layer that fits entirely in on-chip SRAM to focus the comparison on microarchitecture differences.

3.5.1 Overall Performance

Figure 3.11 shows the overall performance of Stitch-X running a range of modern DNNs, i.e., AlexNet [16], VGG-16 [17], Inception-v3 [19], and ResNet-50 [18]. For each network, we evaluate the speedup performance of Stitch-X and state-of-the-art sparse accelerators, e.g. Cnvlutin, Cambricon-X, and SCNN, over an efficient dense SR baseline. The Oracle performance shows the maximum achievable speedup by exploiting both input activation and weight sparsity in these networks, i.e., divide the total number of non-zero multiplications by the number of multipliers in the design. We also plot Stitch-X's multiplier utilization for each layer on the right Y-axis.

Stitch-X achieves a 3.8× average speedup over the dense baseline while main-

taining on average 74% multiplier utilization, across all the networks evaluated. In addition, Stitch-X significantly improves the performance of fully-connected layers of AlexNet and VGG-16 shown in Figures 3.11(a) and (b) and 1×1 convolution layers of ResNet-50 and Inception-v3 shown in Figures 3.11(c) and (d), both of which are typically neglected in existing sparse DNN accelerators. Compared with previous sparse accelerators, Stitch-X achieves 2.0×, 1.8× and 1.6× average speedup improvement over Cambricon-X, Cnvlutin and SCNN, respectively. To highlight the effectiveness of Stitch-X in exploiting sparsity in DNNs, we use the metric Proximity to Oracle Speedup (PTOS) to measure the percentage of achieved speedup over the oracle speedup. Stitch-X achieves a PTOS of 77.4%, while SCNN only achieves a 48.4% for the benchmarked networks.

3.5.2 Sensitivity to Network Sparsity

Performance. Figure 3.12a specifically compares the performance of dense SR, SCNN, Stitch-X, and Oracle, sweeping W and IA densities. We also use the PTOS metric to compare the effectiveness of Stitch-X over SCNN in executing sparse DNN computation. Stitch-X achieves a PTOS of 97% at 1.0/1.0 density and sustains at least 75% PTOS until the density drops to 0.3/0.3, while the highest PTOS achieved by SCNN across all densities is only 79%. At lower densities, Stitch-X shows a diminishing speedup since we provision 4 parallel accesses for the Stitch-X's OA buffer while SCNN has 32. Conceptually, when the input data is extremely sparse, the writeback to the OA buffer becomes more frequent even with multi-level reduction. However, less than 20% density for both weights and input activation is highly uncommon in modern networks, whose the typical density range (also the target of Stitch-X's design) is 40% to 60%, as shown in Figure 3.2. In the typical case of 0.5/0.5, Stitch-X delivers 1.5× better performance than SCNN while the maximum achievable speedup, *i.e.*, oracle speedup over SCNN, is only 1.75×.

Energy. Figure 3.12b shows the energy breakdown of Stitch-X and SCNN normalized to the dense SR design for different IA and W densities. For a typical network density of 0.5/0.5, Stitch-X delivers more than 3.0× better performance than the dense SR baseline while consuming the same energy, leading to an overall 3.0× improvement in energy-delay product. When the data is completely dense, i.e., 1.0/1.0, Stitch-X dissipates 2.4× higher energy than the dense SR baseline. The reason is twofolds: 1) as discussed earlier, Stitch-X uses an intermediate TR level to ease the throughput requirement of index matching, but when it comes to executing fully dense data, TR is less efficient compared to SR; 2) Stitch-X needs the PDU and index storage for sparse data handling, introducing extra energy cost. As data becomes sparser, Stitch-X makes more efficient use of memory and compute resource, leading to better energy scaling. On average, Stitch-X achieves 1.1× better energy efficiency compared to our dense SR baseline. Compared to SCNN, in the fully dense case, Stitch-X is 1.7× more energy- efficient due to the inefficiency of TR-SRAM-based dataflow, as discussed in Section 3.2.1. As a result, Stitch-X achieves an average of 1.4× energy efficiency improvement over SCNN across all data densities.

EDP. To better compare the energy and performance improvement together, we use the energy-delay-product (EDP) as the metric to evaluate the overall efficiency. Figure 3.12c shows the EDP improvement of Stitch-X compared to SCNN. Stitch-X sustains more than 2× EDP improvement over SCNN from fully-dense case to 0.3/0.3 W/IA density. Stitch-X fundamentally addresses *both* the performance and energy limitation in SCNN design and proposes an efficient sparsity handling mechanism to significantly improve the overall execution of sparse DNN accelerators.

3.5.3 Sensitivity to OA Buffer Bandwidth

We quantify Stitch-X'sensitivity to OA buffer bandwidth by sweeping the the bandwidth and measuring the achieved performance at different data densities. Fig-

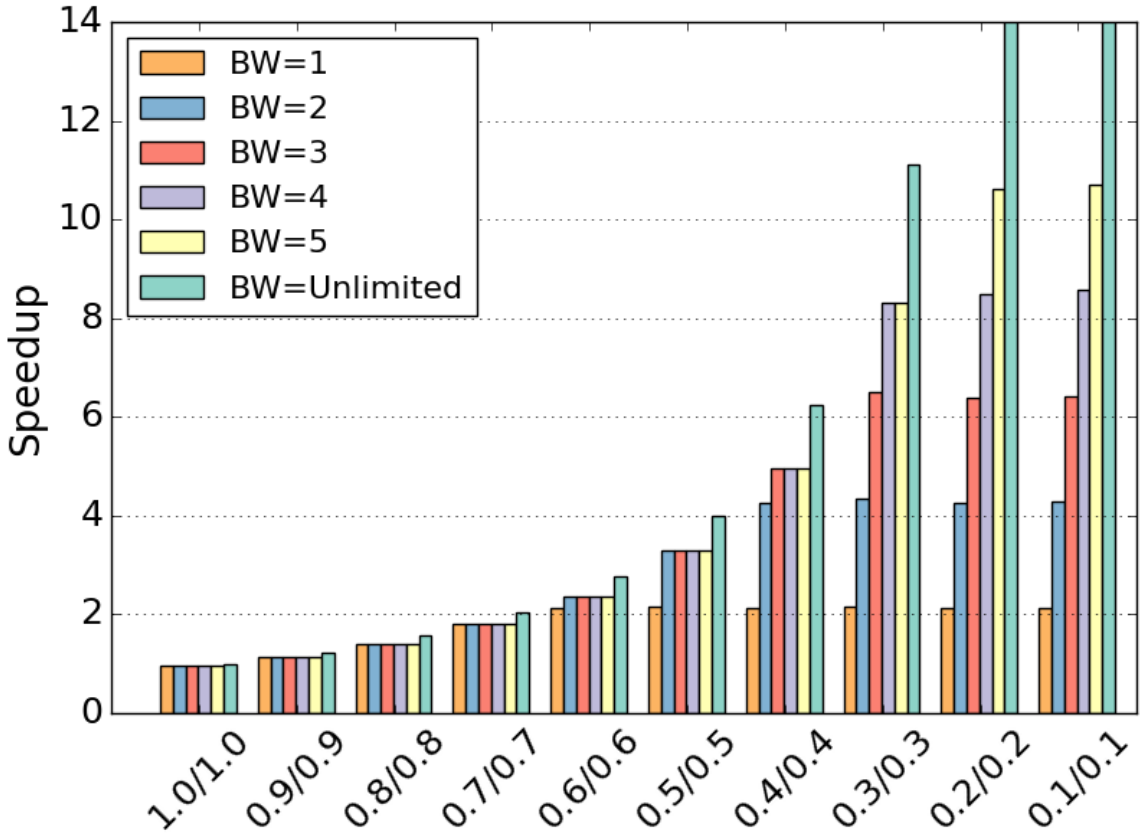


Figure 3.13: Sensitivity to the output activation buffer bandwidth.

ure 3.13 presents the effect of the OA buffer's bandwidth on Stitch-X's performance. The Y-axis plots relative speedup over Stitch-X performance with OA buffer of $BW = 1$ on fully dense data. Stitch-X's multi-level reduction dataflow fully exploits reduction opportunities to significantly reduce the number of parallel accesses to OA buffer. When the data is relatively dense (¿ 0.7/0.7), even a single-port OA buffer is sufficient to achieve the ideal speedup. When data density drops below 0.6/0.6, a single-port OA buffer causes stalls due to bank conflict. Although a 2-way banked OA buffer is sufficient to meet the network's bandwidth requirement at typical workload of 0.5/0.5 density, we implement a 4-way banked OA SRAM in our design to achieve higher speedup for both high and low density workloads.

3.5.4 PDU Time-Multiplexing

Similar to the size of instruction window in the out-of-order processor, the width of PDU's comparator array plays an important role in the number of reducible IA-W pairs that can be found every cycle. For a fixed data density, a wider PDU typically produces a larger number of IA-W pairs, leading to higher multiplier utilization. At the same time, building a wide PDU for each CE can be expensive as PDU'area grows quadratically with its width. One nice property of PDU is that it can be easily shared with multiple CEs and time-multiplexed during its execution. Specifically, we explore four different PDU sharing strategies with the same overall PDU throughput: 1) no sharing, every CE gets its own PDU of width 8, 2) two CEs share a PDU of width 16, 3) four CEs share a PDU of width 32, and 4) eight CEs share a PDU of width 64.

Figure 3.14 captures the trade-off in area and speedup of the above four designs executing ResNet-50 [18]. We only sweep the encoder width from 8 to 64 as wider PDU cannot be synthesized with the same timing constraint. We see that the design PDU of width 64 shared between eight CEs improves the overall performance by 70% with a 55% area increase. The PDU time-multiplexing mechanism scales the total PDU area linearly instead of quadratically, making it possible to design wide PDU, i.e. Stitch-X uses PDU of width 64, for higher hardware utilization.

3.6 Conclusion

This work presents a comprehensive characterization of Cartesian Product-based dataflow, highlighting its intrinsic limitation in achieving high performance and energy efficiency. To address its limitation, we propose Stitch-X, a sparse DNN accelerator architecture that efficiently exploits unstructured sparsity in both input activations and weights. Stitch-X builds Parallelism Discovery Unit, a dynamic, low-cost index matching mechanism, to align non-zero pairs of IA and W so that they

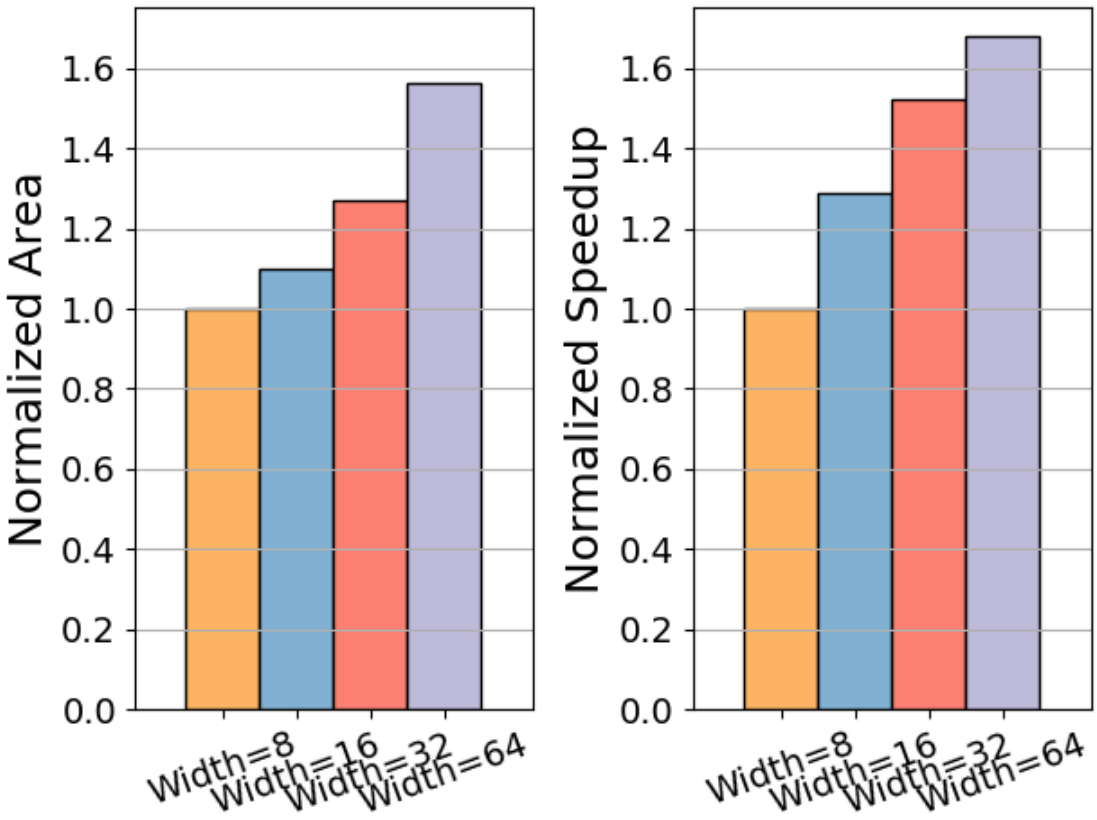


Figure 3.14: Design space exploration of different PDU time-multiplexing strategies.

can be reduced before reaching the output SRAM. Stitch-X also makes use of a multi-level reduction mechanism consisting of local spatial reduction, intermediate temporal reduction, and global spatial reduction to significantly reduce the writeback traffic to SRAM, fundamentally improving the energy consumption and performance over state-of-the-art sparse DNN accelerators. Stitch-X's flexible spatial reduction supports a variety of convolutional and fully-connected layers, making it more versatile than prior designs. Evaluated with modern DNN workloads, Stitch-X achieves more than 1.6× higher performance and 2.1× energy efficiency than a state-of-the-art sparse accelerator.

CHAPTER IV

Domain-Specific Architecture Modeling Framework

4.1 Introduction

This section presents the ERA framework, a systematic design framework for RTML accelerators. The design is done in four steps as illustrated in Figure 4.1: ❶ an RTML accelerator architecture model is designed based on the HANA architectural template using parameterized module instances from PyHLM's component library; ❷ the RTML accelerator's instruction set is defined based on the referenced PyHLM components and the customizable compiler template; ❸ an RTML program is compiled and compressed as instruction binary mapping onto the RTML accelerator architecture model, and the entire model is simulated by PyHLM's back-end; and ❹ the RTML accelerator architecture's performance is evaluated and the system parameters are optimized using the PyHLM Design Explorer. The ERA design steps ❶ and ❷ are supported by HANA, and the ERA steps ❷, ❸ and ❹ are supported by PyHLM[1].

We envision our hardware platform will feature a RTML accelerator to perform application acceleration and a host processor to configure and stream instructions in

[1]Haolei Ye contributed to PyHLM and HANA and Junkang Zhu contributed to the ERA-VOT implementation and benchmarking

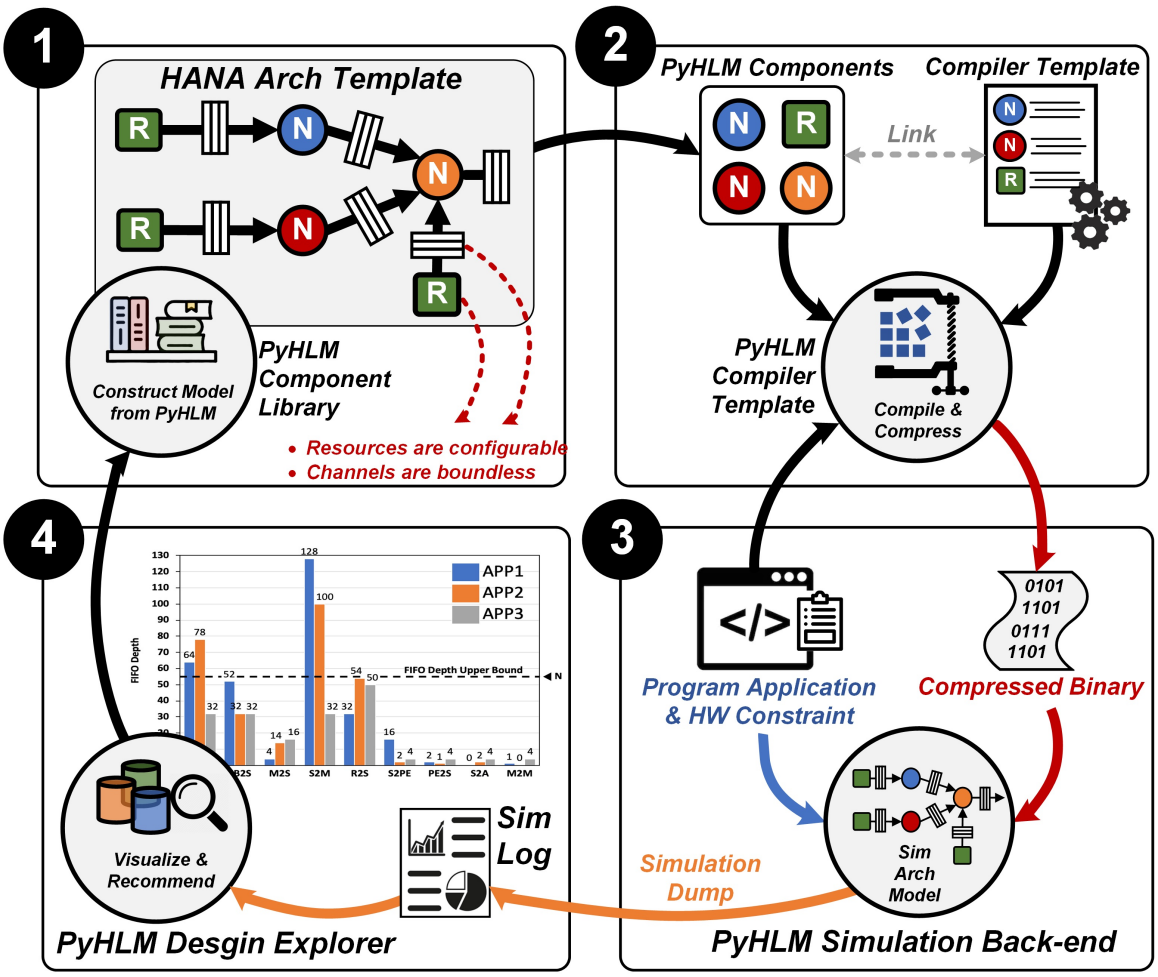


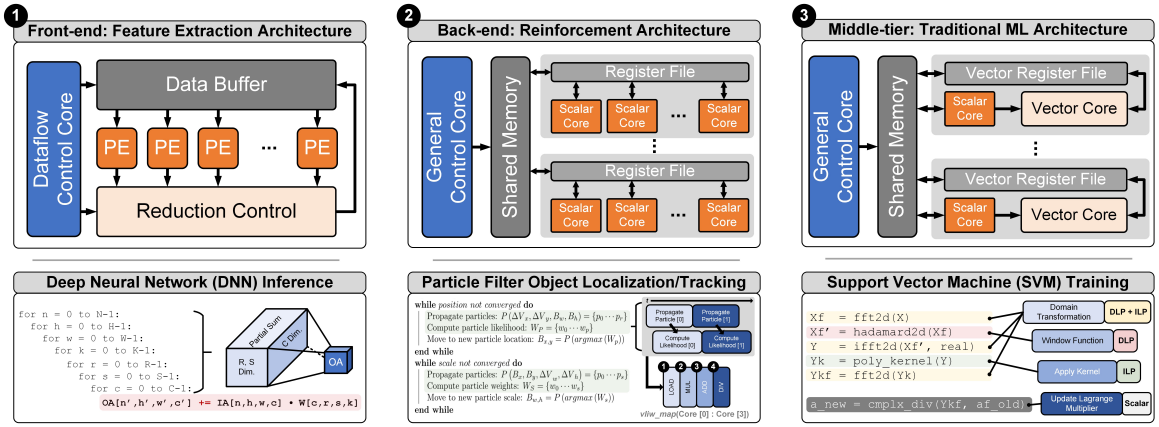
Figure 4.1: Four systematic steps of ERA's architecture modeling framework for RTML accelerator design.

run-time to the accelerator for executions. We showcase **ERA-VOT**: a 64-core visual object tracking (VOT) accelerator designed following the ERA design framework.

4.2 RTML Design Patterns and Templates

RTML is the cornerstone of real-time cognition and learning systems for robotic navigation [83], autonomous driving and augmented realty/virtual reality. A state-of-the-art RTML application typically contains three types of processing patterns:

- **Feature extraction (front-end):** shown in Figure 4.2(a), normally implemented by deep neural networks (DNNs). SIMD architectures are popular for feature



(a) (b) (c)

Figure 4.2: Three processing patterns seen in a RTML program: (a) SIMD pattern is the most advantageous for well-structured data-parallel processing, (b) scalar with VLIW extension pattern best captures data-dependent processing, (c) vector DSP pattern is suitable for programs with a mix of DLP and ILP processing.

extraction due to the inherent data level parallelism (DLP). Static (compile-time) scheduling is essential for achieving high performance and energy efficiency.

- **Reinforcement (back-end):** shown in Figure 4.2(b), incorporates statistical approaches such as Kalman filters [84], Particle Filters [85], or advanced reinforcement learning [86] techniques. Scalar processors with VLIW extensions are the most suitable for accelerating this class of processing by extracting the instruction-level parallelism (ILP).

- **Traditional ML processing (middle-end):** shown in Figure 4.2(c). Accelerating traditional ML learning and inference requires exploiting both DLP and ILP. Vector DSP architectures that can support both SIMD and VLIW are the ideal candidates.

The diverse processing patterns make it difficult to develop one fixed hardware architecture and one fixed set of instructions to bring about the versatility, performance, and energy efficiency required for RTML.

4.2.1 Design Patterns

Architecture design patterns, as described in [87], are the templates that are representative of a variety of common parallel and non-parallel compute patterns. They are the generalizations of computing structures to allow target programs to be adequately and optimally mapped to achieve high performance and energy efficiency.

Recent work on design patterns primarily targets DL models [3, 4, 88, 89]. DL computation typically follows deterministic, offline models. These DL design patterns are optimized for efficient data streaming and reuse by taking advantage of DL's well-established memory access. However, RTML consists of not only offline models, but also online learning algorithms [5, 6] to dynamically adapt for robust performance. Consequently, the prior DL design patterns are inadequate for RTML.

The ERA framework provides Hardware Aligned Nano-service Architecture (HANA): a set of hardware and software architecture design patterns and templates specifically targeting high-performance RTML. HANA contains three important components:

- A decoupled-operator dataflow pattern to provide high performance and flexible architectural construction to support ERA design step ❶.
- A peer-to-peer (P2P) compute-store pattern for efficient parallel process mapping and hardware scale-up to support ERA design step ❶.
- A flexible and custom-defined instruction template for the agile development of decoupled operators and its processing elements to support ERA design step ❷.

4.2.2 Decoupled-operator Dataflow

To meet the "real time" in RTML and the diverse processing modes, HANA adopts a decoupled-operator dataflow architecture to support both high performance and low latency, and yet offer the flexibility to adapt to different processing modes

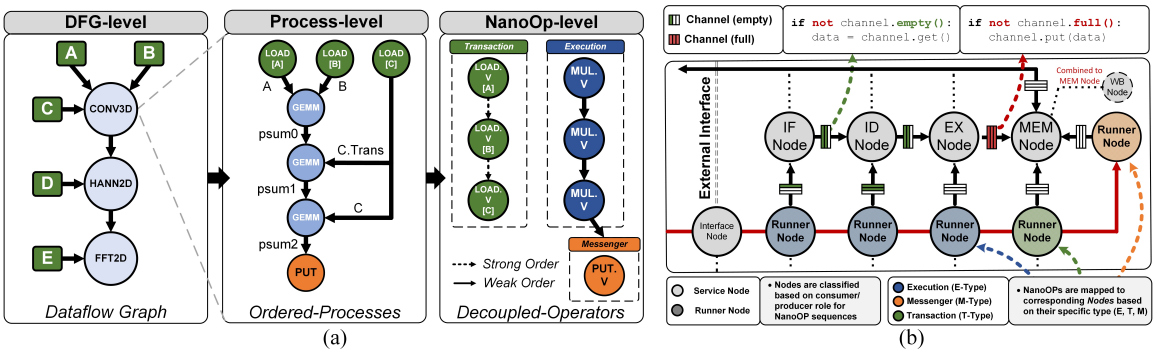


Figure 4.3: (a) Illustration of a dataflow graph (DFG), a process-level graph and a decoupled NanoOP-level graph representation of a RTML program. (b) A 5-stage CPU architecture model following the decoupled-operator dataflow.

from scalar to vector-SIMD and VLIW. Inspired by early works on dataflow architectures [90] and decoupled access execute [91], the decoupled-operator dataflow architecture embeds decoupled execution in a dataflow paradigm. Decoupled execution enables concurrent execution to hide latency, while dataflow provides high throughput, low latency, and a simple flow control. Figure 4.3(a) shows the decoupled-operator dataflow paradigm. A dataflow diagram (DFG) representing a RTML program can be divided as ordered-processes and these processes are further separated as decoupled operators. Specifically, the decoupled-operator dataflow pattern contains the following essential elements:

- **Nano-operator**: or NanoOP, is the generalized decoupled instruction. There are three types of NanoOPs depending on its function: execution (E-type), transaction (T-type) and messaging (M-type). A RTML program is decomposed to decoupled NanoOPs that can be executed concurrently. NanoOPs are coarse-granular execution units to enable a high mapping and processing efficiency. A NanoOP is represented by a single instruction or a set of instruction sequences. Note that in the following, we will use NanoOP and its representing instruction interchangeably.

- **Node**: responsible for processing NanoOPs, i.e., executing instructions. Specifically, there are two types of nodes: 1) runner node that dispatches sequences of

NanoOPs; and 2) service node that executes NanoOP sequences. A node corresponds to a hardware module, or a part of a hardware module.

- **Channel**: A channel refers to a set of FIFOs to allow the data exchange between nodes as well as between nodes and a host.

- **Flow control**: As in a typical dataflow, data transactions are implicitly dependent on the availability of data and the pre-set trigger conditions. This approach foregoes explicit control, and allows a simple flow following node connectivity.

In Figure 4.3(b), a simple example of a 5-stage pipelined in-order CPU architecture model is shown based on the decoupled-operator dataflow. Each execution stage of the CPU is represented by a pair of service node and runner node. A runner node parses NanoOPs to provide data addresses. At the IF, ID and EX stages, the runner nodes dispatch E-Type NanoOPs to the corresponding service nodes; and at the MEM/WB stage, a runner node dispatches T-Type NanoOPs to conduct transactions with memory and another runner node to dispatch M-Type NanoOPs to message external parties to the CPU. Channels connect all the nodes together. The decoupled-operator dataflow is an abstract architecture model and it captures the behavior of general DFG programs. The architecture model can be constructed and simulated by PyHLM discussed in Section 4.3.

The decoupled execution and dataflow provide low-latency and high-throughput execution. The decoupled-operator architecture is sufficiently flexible to allow nodes to operate in parallel or in series. NanoOPs operating on parallel nodes can be SIMD or MIMD to support diverse processing modes. E-Type and T-Type NanoOPs can be mapped to nodes by a compiler to support static processing models; and M-Type NanoOPs can be used to dynamically message connected nodes to provide run-time coordination.

4.2.3 P2P Compute-Store Hardware Pattern

The decoupled-operator dataflow is an abstract architecture model, but it can be readily translated to a set of hardware patterns to make it implementable. Akin to decentralized P2P networks and services rest, a P2P compute-store is a distributed cache-less memory design pattern that allows for flexible integration and scale-up. Nodes or group of nodes can be readily mapped to "peers", and channels can be implemented as physical FIFOs, queues, wire connections, or shared memory. In the following, we describe the hardware patterns that are used for physical implementation.

- **Peers**: As shown in Figure 4.4(a), a peer (in the context of P2P) implements one or a collection of service and runner nodes, including the required channels. A peer is a physical compute-store. A peer can contain service nodes (e.g., multiply-accumulates (MACs), arithmetic logic units (ALUs)), runner nodes (e.g. instruction decoders and generators), and channels (e.g. FIFOs and memory).

- **Memory and data mapping**: The store within a peer is made of distributed memory blocks that provide compute with local memory access determined at compile time. Each block can also be referenced for external memory transactions in run time. Data mapping in memory locations takes into account data locality. As shown in Figure 4.4(b), data required by local compute in a peer are prioritized to be allocated within the peer to avoid data movement across peers. For dynamic workloads, data are mapped in neighboring memory to minimize movement and the access to the external memory.

- **Scaling up**: The P2P compute-store pattern can be scaled up to fit a given problem size, a required degree of parallelism, and a particular depth of operator dependency in a RTML program. As shown in Figure 4.4(c), peers can be grouped into clusters

to exploit an increased level of locality. In addition, peer clusters can be joined by routers.

- **Synchronization**: The decoupled operators can be viewed as parallel threads making concurrent progressions. The NanoOP ordering and inter-node messaging are critical to the potential performance bottlenecks. Figure 4.4(d) shows the abstract method for synchronization support using special annotations in compilation and a special message queuing structure. The specifics will be discussed in Section 4.4.3.

4.2.4 Custom-Defined Instructions

RTML applications span a wide range, and its rapid progression require that the hardware adapt to not only a wide application space but also the evolving RTML algorithms. Therefore, the instructions need to be custom-defined, and even modifiable after the hardware is designed and fabricated. As the instructions are streamed from a host processor to the RTML accelerator during run-time, they need to be flexible and efficient. The following describes HANA's templates for designing, compressing and evolving instructions.

- **Flexible instructions**: A NanoOP is a basic execution unit under HANA's decoupled-operator dataflow architecture. A NanoOP is represented by a variable-field, variable length (VFVL) instruction that consists of multiple fields to represent the sequencing pattern for compute and data access needed for completing the NanoOP. The length of the instruction is variable depending on the operation complexity, and the fields are also variable for instruction streaming efficiency. As shown in Figure 4.5(a), a compiler template is provided for users to define the operator formats.

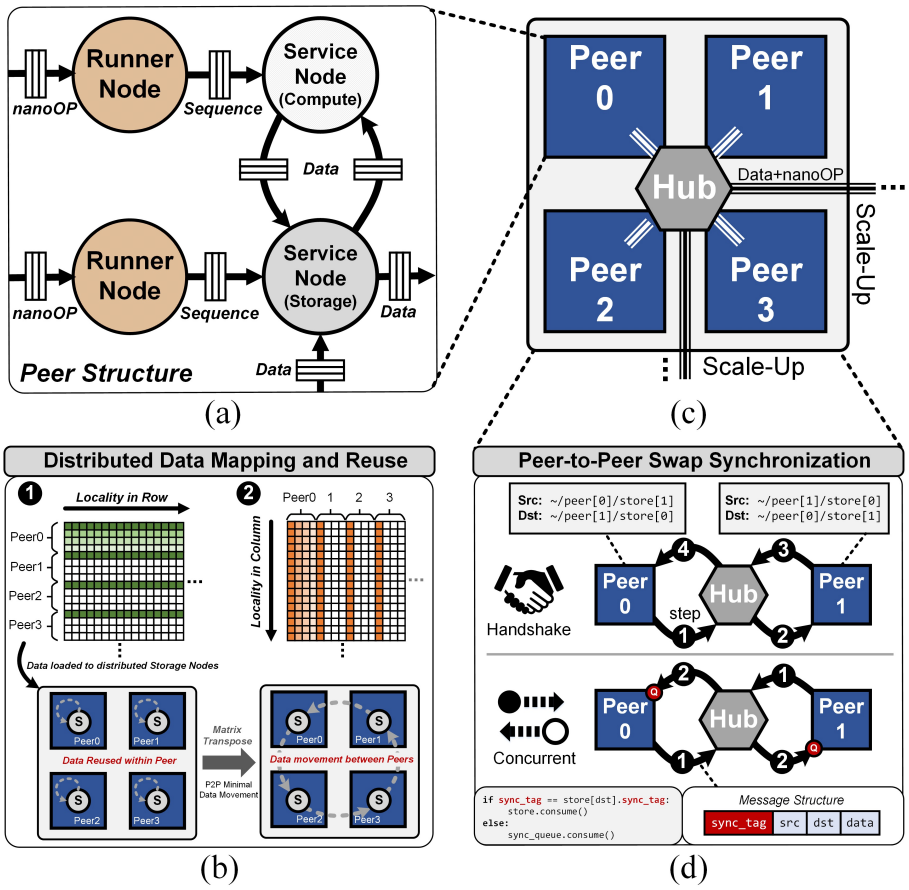
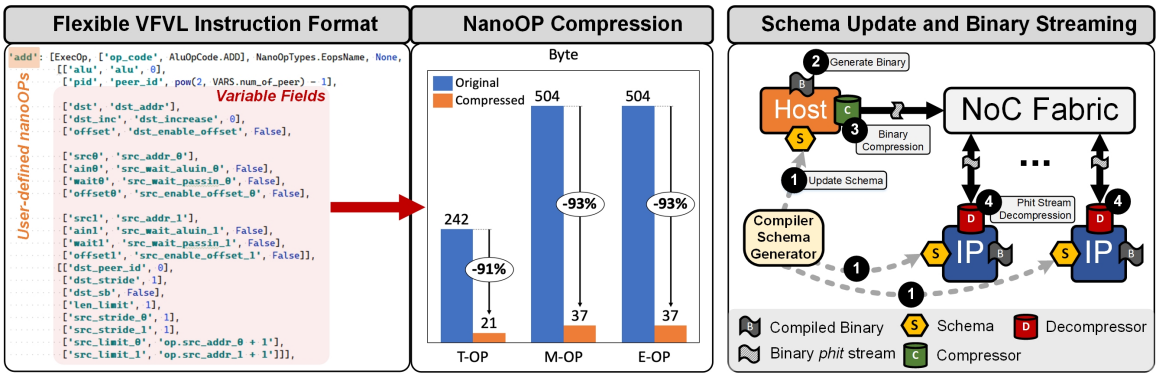


Figure 4.4: The P2P compute-store hardware pattern: (a) Peer structure, (b) locality of a 2D matrix from row to column; data allocation on Peers is prioritized to reduce data footprint, (c) example of clustering Peers and scaling using an interconnect topology, and (d) high-performance P2P synchronization with compiler annotation and synchronization tag.

- **Instruction compression**: To provide efficient instruction streaming, compression is applied to the compiled instruction binary at the host and the instructions are decoded at the RTML accelerator interface. Compression is done by de-serializing instructions into bytes and parsed in order using the variable integer encoding (*varint*) as described in Google Protobuf [92]. Figure 4.5(b) demonstrates more than 90% compression rate for the three types of NanoOPs in HANA.

- **Instruction evolution**: After a RTML accelerator is fabricated, the instructions can still be modified to adapt to new applications. In order to achieve cross-stack



(a) (b) (c)

Figure 4.5: The custom-defined instructions tempalte: (a) a template for flexible instruction design, (b) custom-defined instructions are compressed using *varint* encoding, (c) four steps in deploying an updated schema to module interfaces and streaming compressed data within the system.

compatibility, the Google Protobuf API is integrated within our compiler template to provide instruction descriptions. The API converts custom-defined instructions to a dictionary structure called schema. A schema is the decoding lookup and is portable to the RTML accelerator and other IPs within a system.

Figure 4.5(c) illustrates aspects of the programmable schema deployment, and streaming of instructions from a host processor to the RTML accelerator. HANA's flexible instruction design combined with the infrastructure for instruction compression and evolution represents a systematic approach towards ISA development for RTML accelerators.

4.3 Modeling and Simulation

Specialized computing architectures can be complex, involving both hardware and software.PyMTL [93] is a representative open-source Python-based hardware generation, simulation and verification tool. PyMTL encapsulates multi-level modeling, but it does not permit a close-loop application development, e.g., program-level compilation, transaction-level (TLM) modeling [94], and hardware-software co-design. In

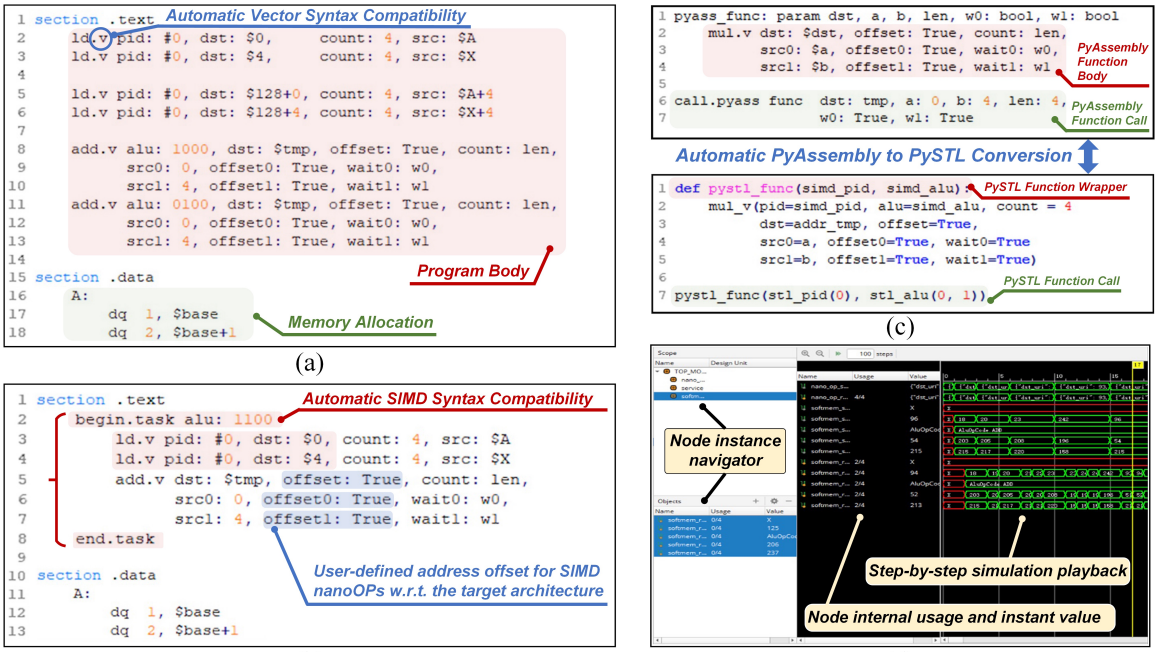


Figure 4.6: The PyHLM toolchain: (a) PyAssembly code structure with automatic NanoOP vector syntax compatibility, (b) automatic SIMD syntax compatibility, (c) automatic API conversion from PyAssembly to PySTL, (d) PyHLM IDE interface for design visualization and debugging.

essence, PyMTL is limited to modeling hardware only. Similarly, Aladdin [95] and DeepBurning [96] are well established architecture simulation, however, the former does not support software-level co-optimization levers and the latter only focuses on neural network accelerator generation. To the best of our knowledge, no current high-level modeling (HLM) tool provides both software and hardware modeling levers for developers to systematically design and jointly optimize an architecture for its target application's performance.

Observing the previous gaps in HLM tools, a new framework need not only retain the qualities of the past, but needs to be developed to emphasize the modeling and simulation of both hardware and software. Such a HLM tool provides the following essential features: 1) high-productivity language constructs to enable a simple architecture abstraction; and 2) cross-stack modeling of hardware-software interactions.

Towards these goals, the ERA framework provides PyHLM, a new Python-based

HLM toolchain for RTML architecture modeling, compilation and design space exploration. PyHLM consists of three key components:

- Model construction based on a parameterized component library for the decoupled-operator flow to support ERA design step ❶.
- A flexible compiler allowing custom-defined instructions to fit the underlying hardware while adapting to changes in the programs to support ERA design step ❷ and ❸.
- A Python-based simulation engine, an integrated design environment (IDE) for debugging, and a design-space explorer for selecting the optimal system parameters to support ERA design step ❹.

4.3.1 High-Level Model Construction

PyHLM provides a basic component library called BaseLib. BaseLib contains all the parameterized components of HANA's decoupled-operator dataflow. Three main classes of components make up the BaseLib: 1) nodes: correspond to the nodes in the decoupled-operator dataflow; 2) ports: I/O gateways that define the transactional behavior between nodes; and 3) channels: correspond the channels in the decoupled-operator dataflow.

In simulating the architectural model, the instruction executions are distributed to individual node instances. Each node also enforces inter-node synchronizations and performs data exchanges.By default, a channel operates like a FIFO, and data-driven exchange is simply operated by "put" and "get" to/from the FIFO.

4.3.2 Compilation Tool

PyHLM supports HANA's compiler template and provides a compiler to generate the NanoOP-level instructions for the target hardware platform.Software-hardware

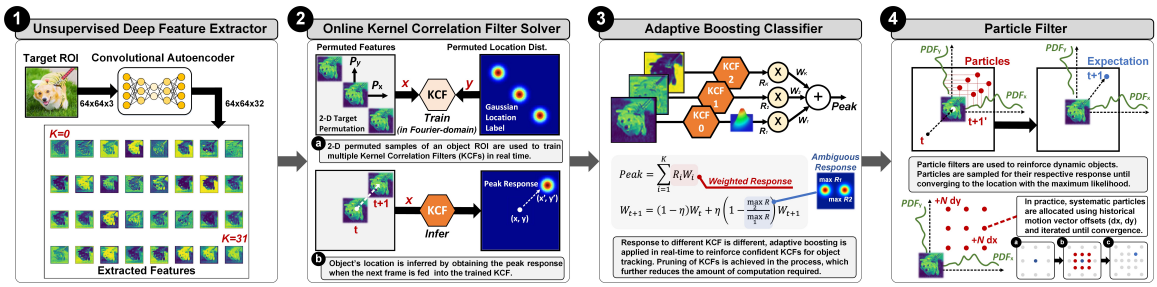


Figure 4.7: The four stages of the OPCF visual object tracking pipeline.

co-design has been historically difficult for custom accelerators due to the lack of flexible compiler support and interaction with the underlying architecture. To reduce the deployment effort of a RTML program, the PyHLM compiler provides two levels of language constructs:

- textbfPyAssembly is an assembly-like low-level programming language that is positioned to co-design the NanoOP instructions with the underlying hardware. A PyAssembly program snippet consists of two main sections, as shown in Figure 4.6(a): *data* for memory that requires allocation during compile time, and *text* for functions and program routines. The PyHLM compiler features automatic instruction field derivation (AIFD) to allow automatic vector and SIMD extensions as shown in Figure 4.6(b). Specifically, the same set of instructions could be easily deployed on multiple processing elements by simply wrapping the original code within a SIMD *task* bracket, significantly reducing the recurring effort in modifying the instruction extensions during prototyping phases.

- **PySTL** is an automatically generated Python API, allowing developers to infer loops, methods and any other Pythonic syntax to easily develop higher-level programs. The value proposition of this language construct is facilitate rapid program construction, especially in early stages of an RTML algorithm development. Figure 4.6(c) shows an example of using PySTL to construct the program as opposed to using PyAssembly. Obviously, PySTL is more efficient in tracking the changes at

the program-level, while PyAssembly can be fine tuned to reflect the changes of the underlying hardware and NanoOp instructions. PySTL improves instruction streaming, code readability and re-usability, reducing the line-of-code (LOC) by at least 3.6×, and more significantly, reducing the compilation time by three orders of magnitude compared to the same program constructed using PyAssembly.

4.3.3 Architecture Simulation

A complete architecture model can be simulated by PyHLM. PyHLM's simulator consists of three parts:

- **PyHLM simulation engine** is based on Python. It invokes a simulation body denoted as a session, wherein all module instances within the model are elaborated, linked to a single *TopModule* class and attached to the session's processing queue for asynchronous execution. Upon initiation, the session will gather all instructions stored in each node and place them in a task list. The simulator goes through the instructions in a round-robin fashion. A single iteration is defined as a *step*, which is the simulation granularity of PyHLM.

- **Design space explorer** utilizes the logged transactional states of each respective node to assist the iterative design parameter selection process. The design space exploration will be explained using an example in Section 4.5.4.

- **Integrated design environment (IDE)** allows all nodes and their states to be visualized and re-played at any *step*. Figure 4.6(d) shows PyHLM's IDE interface, displaying not only the intermediate values of each node instance, but also data traffic hotspots.

4.4 ERA-VOT Accelerator

We use a visual object tracking (VOT) accelerator as an example to demonstrate the result of a full-fledged RTML system designed in the ERA framework. We will refer to this design as the ERA-VOT accelerator and we focus on its microarchitecture and the system parameter optimizations.

4.4.1 Visual Object Tracking

Modern DL-based object detection algorithms [4,88,89] are often used in conjunction with VOT to handle frame-to-frame dependencies and real-world non-idealities, e.g. occlusion, scale-change, rotation, illumination change and motion blurring. The VOT-enabled object detection is faster and more efficient.

Based on state-of-the-art VOT algorithms [5,6], we present a new VOT processing pipeline named oriented-particle correlation filter (OPCF). OPCF handles real-world non-idealities, e.g. occlusion, motion blur etc., via online adaption and reinforcement. OPCF is a representative example of an RTML algorithm. Specifically, OPCF features a mix of unsupervised learning and Bayesian inference techniques, and an OPCF implementation requires learning and inference to be conducted in real time. Figure 4.7 provides a high-level overview of the OPCF algorithm pipeline, which consists of four main computation stages: 1) deep feature extraction; 2) kernel correlation filter (KCF) training and inference; 3) KCF adaptive boosting; and 4) particle filter (PF) reinforcement.

Provided with an initial region-of-interest (ROI) of the target object, features within the ROI are extracted by a convolutional autoencoder. The extracted features are used in run time to train multiple KCFs that serve as discriminators for the target. The KCF correlation responses are inferred across consecutive frames. By aggregating all responses from multiple KCFs, the peak response denotes the most probable object location. Taking into account the confidence level of individual KCFs,

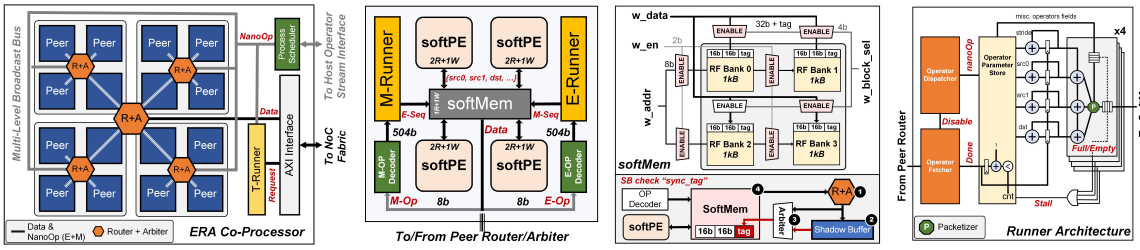


Figure 4.8: An example ERA-VOT accelerator architecture: (a) an example accelerator design using 16-Peer compute-store architecture organized using a 2-layer 4-ary fat tree NoC topology, (b) Peer microarchitecture, (c) SoftMem design and shadow buffer mechansim, (d) runner design.

the responses are adaptively boosted and bootstrapped with a particle filter (PF) to leverage historical motion vectors of the object to further reinforce the detection accuracy whenever dynamic motion is triggered.

OPCF's pipeline stages encapsulate all of the RTML processing patterns described in Section 4.2. In order to efficiently accelerate this complex algorithm pipeline, we present a RTML accelerator architecture design in the next section.

4.4.2 Top-Level Architecture

Figure 4.8(a) shows a possible implementation of the VOT accelerator following HANA design patterns. For example, an implementation of the VOT accelerator can consist of 16 identical peers. Each peer contains 4 processing cores as service nodes. Peers can be organized into groups of four and connected to a network-on-chip (NoC) via a router and arbiter interface. The interconnect is organized as a 4-ary fat tree, accommodating the aggregate data bandwidth requirement at the interface of the accelerator.

The accelerator communicates externally via a standard AXI interface for data transactions and a process scheduler interface for instruction packet distribution and parameter configuration. The process scheduler is made of a pre-decoder that checks the header packets sent from the host for the process identifiers in order to determine

the peers that will receive the instructions. The scheduler acts to reduce the communication traffic needed for instruction streaming. External data requests are made through a transaction runner node, which not only acts as a decoupled DMA for sequence generation, but also implements request coalescing and burst transactions to improve NoC fabric efficiency.

4.4.3 Parameterized Peer Design

Peers are decentralized compute-store units, and each consists of both service nodes and runner nodes. As an example illustrated in Figure 4.8(b), a peer can be internally organized in 4 service nodes, i.e., 4 SoftPEs that share a single SoftMem, and a set of runner nodes for execution-type (E-runner) and messaging-type (M-runner) NanoOPs. The peer microarchitecture implements either SIMD for parallel processing modes, or VLIW and scalar processing for all the other processing modes. Data and instructions are distributed to SoftMem and exchanged between peers via router and arbiter interfaces.

The components of a peer are described in detail below.

- **Runner** is a NanoOP sequence generator for read and write data addresses to SoftMem to trigger SoftPE executions. Figure 4.8(c) shows an implementation of a runner. Runners use variable-loop-depth counters with programmable base, bound and stride to support different memory access patterns such as bound wrap-around and incremental striding. A runner sequences NanoOPs in lock-step or independently depending on whether it is SIMD or VLIW processing.

- **SoftPE** is part of a service node. For VOT processing, an implementation of a SoftPE can contain a 16-bit pipelined floating-point ALU core for arithmetic and binary operators. The SoftPE can be run-time configured to either floating-point or integer modes. A SoftPE contains reuse buffers to efficiently handle recurrent

dependency patterns (e.g. MAC, reduction). The reuse buffers are enabled by the PyHLM compiler upon recognizing such patterns in the program or when specified by the programmer.

- **SoftMem** is shared by SoftPEs. In one implementation, a SoftMem can be a multi-block multi-ported register files (RFs), serving as a data store for the SoftPEs and the NoC for handling P2P data exchange. The SoftMem is designed to contain multiple memory banks, with each bank primarily serving a single SoftPE. SoftMem's internal address space is configurable, allowing virtual allocation of address spaces for all SoftPEs within a peer. A SoftPE can access data that are not physically allocated to its bank.

- **Shadow buffer** is a FIFO for resolving synchronization in P2P messaging and load ordering without incurring the high cost of memory store duplication. A shadow buffer interfaces with SoftMem. Figure 4.8(c) shows the process in determining whether the messages from the shadow buffer are allowed to be written to the SoftMem to guarantee synchronization. Specifically, if an incoming message packet contains a special synchronization tag (sync tag) annotated by the compiler, the message is held in the shadow buffer; otherwise the message is written to the SoftMem. The queued message at the head of the shadow buffer is constantly checked against the destination address' sync tag. The message is written to the SoftMem when the destination address' sync tag is cleared. The queued messages held in the shadow buffer are arbitrated along with the other unqueued messages to ensure the forward progression of the program.

- **NanoOP decoder** is a single *phit* (byte-granularity) *varint* decoder pipelined into two stages: 1) field decoding by a programmable schema look-up table and 2) *varint* decompression by a hardware FIFO that aggregates the decoded *phit* segments into an NanoOP.

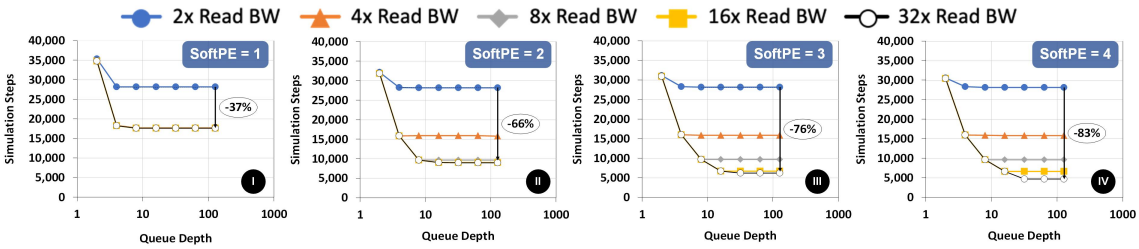


Figure 4.9: The impact of external I/O FIFO depth and width on the latency in processing the 64×64 CONV3D function in OPCF.

4.5 Optimizations and System Benchmarking

We optimize the system parameters for the ERA-VOT accelerator architecture using PyHLM. The latency performance is bencharmked and compared with a dedicated ASIC, a GPU and a CPU model using PyHLM. We also extend the scope of the evaluations beyond RTML applications to show the applicability of the work.

The ERA-VOT accelerator architecture is entirely parameterized. The PyHLM design space exploration is performed to obtain the optimal system parameters following three steps:

❶ **High-level constraints:** a set of high-level system constraints including the program size, the hardware resources available, and the performance target are used to set the bounds of parameter optimization.

❷ **Parameter optimization:** System simulations are done in PyHLM. The sensitivity of each design parameter's impact on performance is evaluated. The most sensitive parameters are tuned in order to obtain a preliminary parameter set.

❸ **Application profiling:** a set of application programs are profiled to validate the design.

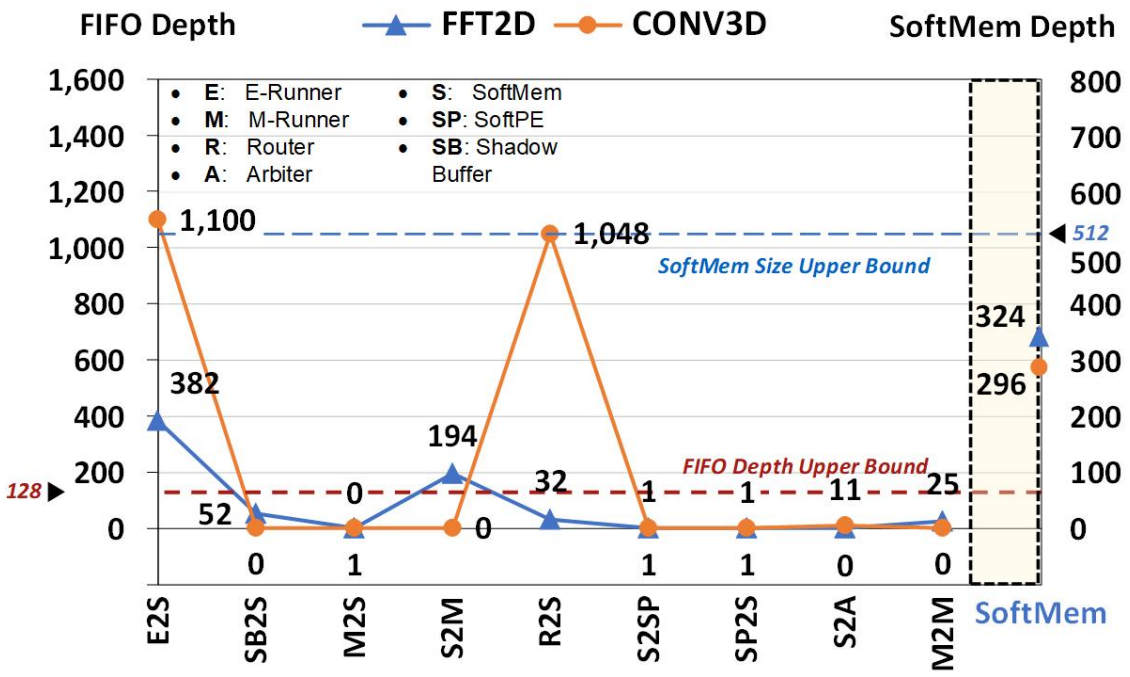


Figure 4.10: The maximum internal FIFO size and SoftMem needed in processing the 64×64 CONV3D and FFT2D functions in OPCF.

4.5.1 SoftMem and Internal Buffer Size

In step ❶, we apply high-level system constraints to bound the parameter selection for the SoftMem size and the internal buffer size. Figure 4.10 plots the maximum SoftMem occupancy and various internal buffers' occupancy for the two most stressing functions within OPCF, the 64×64 16-bit floating-point CONV3D function and the FFT2D function. In performing PyHLM simulations, the size of the SoftMem and each buffer are initially set to be unconstrained to allow the exploration.

Based on the maximum occupancy shown in Figure 4.10, the SoftMem size is set to 512 entries to accommodate the most stressing functions. Also according to Figure 4.10, we identify the potentially large sizes of the SB2S buffer (Shadow Buffer to SoftMem) and the R2S buffer (router to SoftMem) required to support the CONV3D function. However, a more in-depth analysis reveals the imbalance between the slow data consumption and the fast data production causing the buffer size to explode.

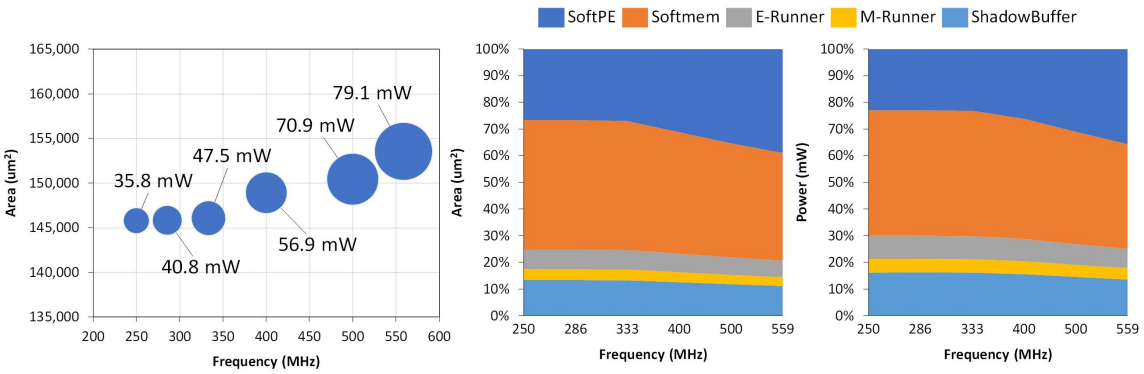


Figure 4.11: a) A Peer's power consumption and area synthesized at different clock frequencies, (b) the power consumption and area breakdown of a Peer synthesized at different clock frequencies.

Table 4.1: ERA-VOT accelerator final parameters

	Search Range	Finalized Value
Peers	1, 2, 4, 8, 16	16
SoftMem Size	128 - 4096B	2048B
SoftMem Banks	1 - 4	4
SoftPE	1 - 4	4
I/O FIFO Depth	1 - 256	16
Read Bandwidth	32b× 1 - 32	32b × 4
Internal FIFO Depth	1 - 1024	See Figure 10

Since the performance is only limited by the slow data consumption, we choose to bound the size of these two buffers to 128, which will have no impact on the system performance.

4.5.2 SoftPE Allocation and External I/O Sizing

In step ❷, we demonstrate the process for selecting the optimal number of SoftPEs per Peer and the external I/O FIFO width and depth again using the most stressing function, a 64×64 CONV3D in the OPCF algorithm. We select the number of SoftPEs per Peer and the external I/O FIFO depth and width to reduce the system latency until it reaches diminishing returns. As shown in Figure 4.9, allocating more SoftPEs

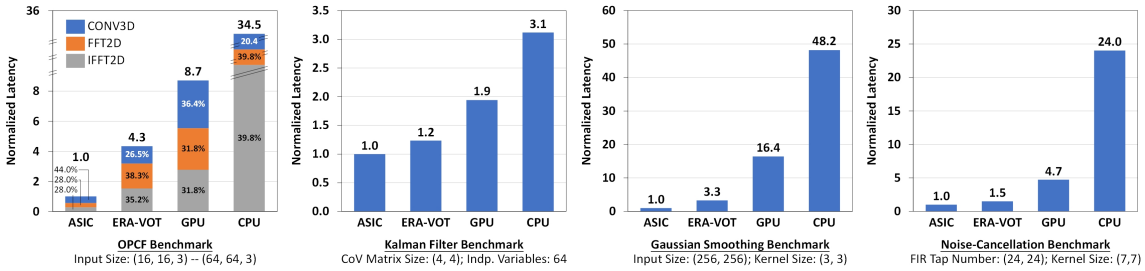


Figure 4.12: Performance comparison between the ERA-VOT accelerator and the ideal ASIC, CPU and GPU for OPCF, Kalman filter, Gaussian smoothing, and noise-cancellation algorithm benchmarks.

per Peer reduces the system latency until it reaches 3 SoftPEs per Peer. However, considering the relatively low marginal silicon area cost to add one additional SoftPE, we choose 4 SoftPEs per Peer for the best performance to cost.

Considering the Peer design with 4 SoftPEs, the design point of 4 to 8 for the external I/O FIFO width and 16 for the depth is optimally positioned as shown in Figure 4.9, beyond which we reach diminishing returns on latency reduction. Therefore, we use these parameters for the external I/O FIFO design.

4.5.3 Hardware Design Validation

Before proceeding to step ❸, we validate the power and area of the selected systems parameters listed in Table I. The ERA-VOT accelerator architecture model can be readily converted to System Verilog. The design is then synthesized using Synopsys Design Compiler in a 28nm technology library for evaluation.

Since the architecture is modular, we present the evaluation of a single Peer. Figure 4.11(a) shows that

a Peer operates a maximum clock frequency of 558MHz, consuming 79.1mW and occupies 0.155 um^2. The critical path is located within the SoftPE's ALU datapath.

Figure 4.11(b) shows the silicon area and power breakdown for a Peer's constituent blocks.

SoftMem and SoftPE take up more than 75% and 70% of the area and power,

respectively and the remaining portions relate to the decoupled execution and synchronization, which presents only minimal overhead for the improved performance.

4.5.4 Application Benchmarking

In step ❸, OPCF and three other applications that cover different processing patterns are profiled for the ERA-VOT accelerator and the common alternatives including ASIC, CPU and GPU. The ASIC model is constructed such that every sub-function within an application benchmark can be perfectly accelerated at full utilization. The CPU model is an in-order 5-stage pipelined single-core architecture (Figure 4.3(b)) and the GPU model is a 64-core SIMD architecture with a local register file and a top-level shared memory. Figure 4.12 shows their normalized latency. Across the four applications, the ERA-VOT accelerator achieves a range of speedups compared to CPU and GPU, while the gap between the ERA-VOT accelerator and the ideal but inflexible ASIC can be kept relatively low.

- **OPCF** represents a generic prototype of an RTML algorithm. It consists of a mix of feature extraction (DNN inference), transformation for online ML (2D FFT/IFFT and KCF kernel functions) and reinforcement (PF), which result in a diverse set of DLP and ILP operators. The ERA-VOT achieved a speedup of 7.9× and 2× compared to CPU and GPU, respectively. The speedup is mainly attributed to exploiting the data locality for FFT2D and IFFT2D operators by the SIMD processing internal to each Peer in the ERA-VOT accelerator without resorting to expensive data movement. While the ERA-VOT performance is 4.3× worse than the ideal ASIC, the inflexible ASIC is an impractical solution for RTML.

- **Kalman filter** is intrinsically data-parallel, but it is wasteful and difficult to draw the full benefits of data-parallel architectures due to the degree of parallelism being awkwardly positioned between SIMD and scalar compute granularity. Con-

sequently, Kalman Filters can be mapped to all hardware platforms with a relatively high utilization, but it requires delicate control of the processing elements to achieve a high performance and efficiency. The ERA-VOT accelerator leverages the PyHLM compiler to generate instructions that are tailored to the compute granularity at the Peer-level. The ERA-VOT accelerator achieved a speedup of 2.5× and 1.6× compared to CPU and GPU, respectively.

- **Gaussian smoothing** is both compute and memory intense. It represents a class of feature extraction algorithms that can be easily mapped to SIMD architectures. For this class of applications, the buffered data need to be maximally reused to obtain a high performance. The ERA-VOT accelerator leverages its low-latency P2P synchronization mechanism to achieve a high speedup of 16× and 5.3× compared to CPU and GPU, respectively. The gap between the ERA-VOT accelerator and the ASIC is due to the ideal ASIC's large on-chip buffers that remove the need for excessive external data access.

- **Noise-cancellation** is an algorithm for self-interference cancellation in full-duplex radios [97]. It also shares many computational similarities with a broader suite of statistical inference and online ML applications. The complex function composition is similar to what is seen in OPCF, but the data access is more unstructured than OPCF, which undermines the degree of ILP. The ERA-VOT accelerator allows tailored VLIW NanoOP at the program level, and it is convenient to perform in-Peer VLIW with locality-oriented data access. The ERA-VOT accelerator achieved a speedup of 16.0× and 4.7× compared to CPU and GPU, respectively. The performance of the ERA-VOT accelerator is only 33% worse than the ideal ASIC, which is mainly due to the P2P data exchange overhead.

4.6 Conclusion

This work presents the ERA framework; an end-to-end architecture development framework, including hardware design templates and a complete software toolchain tailored for designing and optimizing RTML processing architectures. We elaborate on the structured steps in approaching an RTML architecture design using our framework, and demonstrate a high-performance ERA-VOT accelerator model. Our prototype accelerator outperforms in all four benchmark applications by an average factor of 8.4× and 3.0× over CPU and GPU, respectively. The ERA framework not only provides programming flexibility and low design barrier, but also unique opportunities in exploiting both DLP and ILP at different compute granularities to obtain efficient acceleration. Broadly, our framework establishes a complete infrastructure stack for systematic architecture development, where we believe in the future, it has the potential to be extended beyond its current domain.

CHAPTER V

Conclusion

This dissertation presents the study ~~and advancement~~ of tightly-coupled algorithm and hardware co-design and domain-specific computing architectures and paradigms. The dissertation work is summarized from three perspectives:

- **Algorithms and specialized microarchitectures**. Intimate algorithm and hardware co-design is the bread and butter of modern domain-specific computing acceleration. We investigated a neuromorphic computing algorithm and proposed a specialized computing architecture targeting action classification and motion tracking. Specifically, we introduce a sparse spatio-temporal (ST) cognitive system-on-a-chip (SoC), designed to extract ST features from videos. The SoC core consists of a sparse ST convolutional auto-encoder. High sparsity is enforced at each layer of processing, reducing the complexity of ST convolution by two orders of magnitude and simplifying all multiply-accumulates (MAC) to select-adds (SA). The SoC compresses ST kernels, reducing memory by 43%, and at 0.9V and 240MHz, the SoC achieves 1.63TOPS to meet the 60fps 1920×1080 HD video data rate, dissipating 127mW.

- **Domain-specific acceleration architectures**. Machine learning and deep learning algorithms have drastically advanced the capabilities of artificial intelligence and rejuvenated the demand for domain-specific acceleration. We investigated the

inefficiencies of state-of-the-art Cartesian product-based dataflow for deep learning inference and addressed its limitation for sparse deep neural network (DNN) acceleration. Specifically, we proposed a novel DNN inference accelerator that efficiently leverages both sparse weights and input activations within neural networks. Our design features a novel runtime look-ahead index matching unit in hardware to efficiently extract reducible computation, achieving high energy efficiency and low control complexity for a variety of DNN layers. Our protoype accelerator delivers up to 4.3× speedup over an efficient, dense DNN accelerator, 1.6× speedup and 2.1× energy-delay-product improvement compared to a state-of-the-art sparse DNN accelerator.

- **Domain-specific modeling paradigms**. Real-time machine learning (RTML) is an emerging frontier of machine learning, shifting the demand for domain-specific acceleration to incorporate both software and hardware optimizations, especially during the architecture modeling phase. We propose a new architecture modeling framework to tackle this complex problem. Specifically, a new end-to-end development framework is proposed to develop RTML-specialized acceleration architectures and systems from software to hardware. Our framework consists of two key components: 1) a set of high-performance RTML-specific architecture design templates; and 2) an open-source Python-based high-level modeling and compiler tool chain for cross-stack architecture design and exploration. Using our framework we demonstrated a real-time visual object tracking (VOT) accelerator, achieving an average speedup of 2.1× over state-of-the-art architecture design patterns across a range of modern RTML algorithms.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] C. M. Bishop, Pattern recognition and machine learning. springer, 2006.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," Nature, vol. 521, 2015.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A Large-scale Hierarchical Image Database," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2009.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-time Object Detection," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[5] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-speed tracking with kernelized correlation filters," IEEE transactions on pattern analysis and machine intelligence, vol. 37, 2014.

[6] M. Danelljan, G. Bhat, F. Shahbaz Khan, and M. Felsberg, "Eco: Efficient convolution operators for tracking," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2017.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in Advances in neural information processing systems, 2017.

[8] H. M. Choset, S. Hutchinson, K. M. Lynch, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, Principles of robot motion: theory, algorithms, and implementation. MIT press, 2005.

[9] C.-E. Lee, Y. S. Shao, J.-F. Zhang, A. Parashar, J. Emer, S. W. Keckler, and Z. Zhang, "Stitch-X: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," in SysML Conference, 2018.

[10] C.-E. Lee, Y. Shao, A. Parashar, J. Emer, and S. W. Keckler, "Deep neural network accelerator with fine-grained parallelism discovery," December 5 2019, uS Patent App. 15/929,093.

[11] B. A. Olshausen, "Learning sparse, overcomplete representations of time-varying natural images," in Proceedings 2003 International Conference on Image Processing (Cat. No. 03CH37429), vol. 1. IEEE, 2003.

[12] I. Laptev, M. Marszalek, C. Schmid, and B. Rozenfeld, "Learning realistic human actions from movies," in 2008 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, 2008.

[13] G. Willems, T. Tuytelaars, and L. Van Gool, "An efficient dense and scale-invariant spatio-temporal interest point detector," in European conference on computer vision. Springer, 2008.

[14] C. J. Rozell, D. H. Johnson, R. G. Baraniuk, and B. A. Olshausen, "Sparse coding via thresholding and local competition in neural circuits," Neural computation, vol. 20, 2008.

[15] S. Savarese, A. DelPozo, J. C. Niebles, and L. Fei-Fei, "Spatial-temporal correlatons for unsupervised action classification," in 2008 IEEE Workshop on Motion and video Computing. IEEE, 2008.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in Proceedings of the Conference on Neural Information Processing Systems (NIPS), 2012.

[17] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-scale Image Recognition," arXiv preprint arXiv:1409.1556, 2014.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[19] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[20] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2014.

[21] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[22] S. Han, J. Pool, J. Tran, and W. Dally, "Learning Both Weights and Connections for Efficient Neural Network," in Proceedings of the Conference on Neural Information Processing Systems (NIPS), 2015.

[23] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," arXiv preprint arXiv:1510.00149, 2015.

[24] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in European Conference on Computer Vision, 2018.

[25] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," IEEE Journal of Solid-State Circuits (JSSC), vol. 52, 2016.

[26] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An Accelerator for Sparse Neural Networks," in Proceedings of the International Symposium on Microarchitecture (MICRO), 2016.

[27] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2016.

[28] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," ACM SIGARCH Computer Architecture News, vol. 45, 2017.

[29] Z. Yuan, J. Yue, H. Yang, Z. Wang, J. Li, Y. Yang, Q. Guo, X. Li, M.-F. Chang, H. Yang, and Y. Liu, "STICKER: A 0.41-62.1 TOPS/W 8Bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers," 2018.

[30] Z. Yuan, Y. Liu, J. Yue, Y. Yang, J. Wang, X. Feng, J. Zhao, X. Li, and H. Yang, "STICKER: An Energy-Efficient Multi-Sparsity Compatible Accelerator for Convolutional Neural Networks in 65-nm CMOS," IEEE Journal of Solid-State Circuits (JSSC), 2019.

[31] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 9, 2019.

[32] R. Nishihara, P. Moritz, S. Wang, A. Tumanov, W. Paul, J. Schleier-Smith, R. Liaw, M. Niknami, M. I. Jordan, and I. Stoica, "Real-time machine learning: The missing pieces," in Proceedings of the 16th Workshop on Hot Topics in Operating Systems, 2017.

[33] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.

[34] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.

[35] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in Proceedings of the 46th International Symposium on Computer Architecture, 2019.

[36] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable domain-specific computing," IEEE Design & Test of Computers, vol. 28, 2010.

[37] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019.

[38] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina et al., "Magnet: A modular accelerator generator for neural networks," in Proceedings of the International Conference on Computer-Aided Design (ICCAD), 2019.

[39] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," IEEE Transactions on computer-aided design of integrated circuits and systems, vol. 20, 2001.

[40] B. Khailany, E. Krimer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter, M. Pellauer, N. Pinckney, Y. S. Shao et al., "A modular digital VLSI flow for high-productivity SoC design," in 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). IEEE, 2018.

[41] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, High—Level Synthesis: Introduction to Chip and System Design. Springer Science & Business Media, 2012.

[42] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze et al., "{TVM}: An automated end-to-end optimizing compiler for deep learning," in 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 2018.

[43] J. Redgrave, A. Meixner, N. Goulding-Hotta, A. Vasilyev, and O. Shacham, "Pixel Visual Core: Google's Fully ProgrammableImage Vision and AI Processor For Mobile Devices," in Proc. IEEE Hot Chips Symp.(HCS), 2018.

[44] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," Acm Sigplan Notices, vol. 48, 2013.

[45] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "VTA: an open hardware-software stack for deep learning," arXiv preprint arXiv:1807.04188, 2018.

[46] M. Baccouche, F. Mamalet, C. Wolf, C. Garcia, and A. Baskurt, "Spatio-Temporal Convolutional Sparse Auto-Encoder for Sequence Classification."

[47] K. Zhang, L. Zhang, and M.-H. Yang, "Fast compressive tracking," IEEE transactions on pattern analysis and machine intelligence, vol. 36, 2014.

[48] C. Schuldt, I. Laptev, and B. Caputo, "Recognizing human actions: a local SVM approach," in Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004., vol. 3. IEEE, 2004.

[49] A. Suleiman, Z. Zhang, and V. Sze, "A 58.6 mW 30 Frames/s Real-Time Programmable Multiobject Detection Accelerator With Deformable Parts Models on Full HD 1920×1080 Videos," IEEE Journal of Solid-State Circuits, vol. 52, 2017.

[50] P. Knag, J. K. Kim, T. Chen, and Z. Zhang, "A sparse coding neural network ASIC with on-chip learning for feature extraction and encoding," IEEE Journal of Solid-State Circuits, vol. 50, 2015.

[51] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[52] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation," arXiv preprint arXiv:1511.00561, 2015.

[53] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs," arXiv preprint arXiv:1606.00915, 2016.

[54] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in Proceedings of the Conference on Neural Information Processing Systems (NIPS), 2014.

[55] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A Generative Model for Raw Audio," arXiv preprint arXiv:1609.03499, 2016.

[56] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang et al., "End to End Learning for Self-driving Cars," arXiv preprint arXiv:1604.07316, 2016.

[57] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch,

N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2017.

[58] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengil, M. Liu, D. Lo, S. Alkalay, M. Haselman, C. Boehn, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Keil, K. Holohan, T. Juhasz, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, S. Reinhardt, A. Sapek, R. Seera, B. Sridharan, L. Woods, P. Yi-Xiao, R. Zhao, and D. Burger, "Accelerating Persistent Neural Networks at Datacenter Scale," in HotChips, 2017.

[59] NVIDIA, "NVIDIA Deep Learning Accelerator (NVDLA)," https://github.com/nvdla/, 2017.

[60] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding Sources of Inefficiency in General-purpose Chips," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2010.

[61] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?" in Proceedings of the International Symposium on Microarchitecture (MICRO), 2010.

[62] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS), March 2010.

[63] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "ASIC Clouds: Specializing the Datacenter," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2016.

[64] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The Architecture and Design of a Database Processing Unit," in Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS), March 2014.

[65] S. Kumar, N. Vedula, A. Shriraman, and V. Srinivasan, "DASX: Hardware Accelerator for Software Data Structures," in Proceedings of the International Conference on Supercomputing (ICS), 2015.

[66] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning," in Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS), March 2014.

[67] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun et al., "DaDianNao: A Machine-learning Supercomputer," in Proceedings of the International Symposium on Microarchitecture (MICRO), 2014.

[68] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2015.

[69] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," ACM SIGARCH Computer Architecture News, vol. 44, 2016.

[70] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in Proceedings of the International Symposium on Microarchitecture (MICRO), 2012.

[71] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-level Deep Neural Models to FPGAs," in Proceedings of the International Symposium on Microarchitecture (MICRO), 2016.

[72] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A Runtime Reconfigurable Dataflow Processor for Vision," in Computer Vision and Pattern Recognition Workshops (CVPRW), 2011.

[73] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2016.

[74] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), 2015.

[75] Y. Shen, M. Ferdman, and P. Milder, "Escher: A CNN Accelerator with Flexible Buffering to Minimize Off-Chip Transfer," in Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017.

[76] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN Accelerators," in Proceedings of the International Symposium on Microarchitecture (MICRO), 2016.

[77] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2018.

[78] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Programmable Interconnects," in Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS), 2018.

[79] S. Han, J. Pool, J. Tran, and W. Dally, "Learning Both Weights and Connections for Efficient Neural Network," in Proceedings of the Conference on Neural Information Processing Systems (NIPS), 2015.

[80] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2016.

[81] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[82] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing Energy-efficient Convolutional Neural Networks Using Energy-aware Pruning," in Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR), 2017.

[83] A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, and V. Sze, "Navion: A 2-mW fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones," IEEE Journal of Solid-State Circuits, vol. 54, 2019.

[84] R. E. Kalman, "A new approach to linear filtering and prediction problems," 1960.

[85] J. M. Hammersley and K. W. Morton, "Poor man's monte carlo," Journal of the Royal Statistical Society: Series B (Methodological), vol. 16, 1954.

[86] Y. Xiang, A. Alahi, and S. Savarese, "Learning to track: Online multi-object tracking by decision making," in Proceedings of the IEEE international conference on computer vision, 2015.

[87] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in Proceedings of the International Symposium on Computer Architecture (ISCA).

[88] R. Girshick, "Fast r-cnn," in Proceedings of the IEEE international conference on computer vision, 2015.

[89] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in Proceedings of the IEEE international conference on computer vision, 2017.

[90] D. E. Culler, "Dataflow architectures," Annual review of computer science, vol. 1, 1986.

[91] J. E. Smith, "Decoupled access/execute computer architectures," ACM SIGARCH Computer Architecture News, vol. 10, 1982.

[92] K. Varda, "Protocol buffers: Google's data interchange format," Google Open Source Blog, Available at least as early as Jul, vol. 72, 2008.

[93] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2014.

[94] L. Cai and D. Gajski, "Transaction level modeling: an overview," in First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (IEEE Cat. No. 03TH8721). IEEE, 2003.

[95] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). IEEE, 2014.

[96] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family," in 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, 2016.

[97] Y. Kurzo, A. Burg, and A. Balatsoukas-Stimming, "Design and implementation of a neural network aided self-interference cancellation scheme for full-duplex radios," in 2018 52nd Asilomar Conference on Signals, Systems, and Computers. IEEE, 2018.