Efficient Algorithms, Hardware Architectures and Circuits for Deep Learning Accelerators

by

Miaorong Wang

B.S., Shanghai Jiao Tong University (2016)
S.M., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

Authored by:	Miaorong Wang Department of Electrical Engineering and Computer Science August 18, 2023
Certified by:	Anantha P. Chandrakasan Vannevar Bush Professor of Electrical Engineering and Computer Science Dean, MIT School of Engineering Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Efficient Algorithms, Hardware Architectures and Circuits for Deep Learning Accelerators

by

Miaorong Wang

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2023, in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Abstract

Deep learning has permeated many industries due to its state-of-the-art ability to process complex data and uncover intricate patterns. However, it is computationally expensive. Researchers have shown in theory and practice that the progress of deep learning in many applications is heavily reliant on increases in computing power, and thus leads to increasing energy demand. That may impede further advancement in the field. To tackle that challenge, this thesis presents several techniques to improve the energy efficiency of deep learning accelerators while adhering to the accuracy and throughput requirements of the desired application.

First, we develop hybrid dataflows and co-design the memory hierarchy. That enables designers to trade off the reuse between different data types across different storage elements provided by the technology for higher energy efficiency. Second, we propose a weight tuning algorithm and accelerator co-design, which optimizes the bit representation of weights for energy reduction. Last, we present VideoTime^3, an algorithm and accelerator co-design for efficient real-time video understanding with temporal redundancy reduction and temporal modeling. Our proposed techniques enrich accelerator designers' toolkits, pushing the boundaries of energy efficiency for sustainable advances in deep learning.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Vannevar Bush Professor of Electrical Engineering and Computer Science
Dean, MIT School of Engineering

Acknowledgments

This thesis would not have been possible without the help of a large group of people. First of all, I would like to express my sincere gratitude to my thesis supervisor, Professor Anantha P. Chandrakasan, for his guidance and support throughout my doctoral journey. I feel very fortunate to have had the opportunity to work under his mentorship. He is always very supportive and provides a great environment for me to learn and grow. Also, I am very fortunate to have a unique perspective into the professional life of an exceptional leader who has achieved remarkable success and global recognition in the field. Having a role model like him has an invaluable impact on my professional life.

I would like to acknowledge Professor Vivienne Sze and Professor Song Han for serving on my thesis committee and providing invaluable advice. I would like to thank Professor Vivienne Sze for providing me with invaluable feedback on paper writing and presentation, and Professor Song Han for the guidance in the VideoTime^3 project (Chapter 4).

I would also like to extend my appreciation to Professor Luca Daniel for serving on my RQE committee and Professor David J. Perreault for being my academic advisor and providing me with great advice every semester.

I would like to express my heartfelt gratitude to Professor Joel Emer. He led me into computer architecture and deep learning accelerator design with two of his courses (with Professor Daniel Sanchez and Professor Vivienne Sze). And his way of understanding and analyzing hardware architectures has greatly influenced my way of thinking.

I am very grateful to my mentor Joyce Wu. She is always there answering my questions and providing guidance about professional and personal life. Many thanks to her unlimited plant support that lights up my living environment.

My internship at the ASIC/VLSI research group of NVIDIA also contributed a lot to my professional development. I would like to thank my manager Dr. Brucek Khailany, my mentor Dr. Rangharajan Venkatesan, my co-mentor Prof. Yakun

Sophia Shao, and many other colleagues at NVIDIA for their helpful advice during my internship.

I am very grateful to all past and present members of Anantha Group for creating such a supportive working environment. I would like to thank Utsav Banerjee for providing great support for my first chip tape-out, Michael Price for answering my questions about speech recognition and chip measurement setups even though we did not overlap during MIT, Mehul Tikekar, Priyanka Raina and Phillip Nadeau for helpful advice about tape-out, Preetinder Garcha for being a good friend and cubicle neighbor, our visiting student Xiaopeng Zhong for answering my questions regarding chip measurement and other senior members for helpful discussions. Also, thanks to Alex Ji for being a wonderful peer and Ruicong (Ray) Chen, Rishabh Mittal, Skanda Koppula, Vipasha Mittal, Deniz Umut Yildirim, Zhiye (Zoey) Song, Adam Matthew Gierlach, Eunseok Lee, Kyungmi Lee, Maitreyi Ashok and Saurav Maji for many happy chats.

I would like to thank Margaret Flaherty, Yuvie Cjapi, Jessie-Leigh O Thomas, Katey Provost, and other former lab administrators for their help with appointment scheduling, reimbursement, purchase orders, and lab space maintenance. I would also like to thank MTL Compute Team for their quick response to emails for various IT issues (sometimes even during non-working hours).

My life at MIT would not be so colorful without my friends. I want to thank them for all the fun we had together, such as badminton games, meals, BBQs, movies, etc.

I would like to acknowledge Irwin Mark Jacobs and Joan Klein Jacobs Presidential Fellowship for funding the academic year 2016 – 2017, TSMC for funding the eMRAM evaluation, Foxconn for funding the weight tuning project, and Qualcomm for funding the VideoTime^3 project.

I would like to thank my boyfriend for his support. Thanks for trying his best to feeding me well when I was busy! Many thanks to all the wonderful moments we have together.

Finally, I would like to express my deepest gratitude to my family members. I am very grateful to my parents for being my biggest cheerleaders, and offering words of encouragement when I needed them the most. Their understanding gave me a lot of comfort in the difficult time during the pandemic. I would like to thank my grandmas for all they have done for me!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Deep learning (DL) is widely used in many applications for its state-of-the-art performance revolutionizing how we process and analyze complex data. Among its many uses, DL has demonstrated impressive results in image and video understanding, speech recognition, and natural language processing. In image and video understanding, DL models have been deployed on drones and autonomous vehicles to detect and track objects, such as pedestrians, roads, obstacles, etc. [13, 11]. In natural language processing, the state-of-the-art DL model, such as the generative pre-trained Transformer (GPT) series [12], enables a natural and interactive way of obtaining knowledge and assistance. With the growth in data and the increasing demand for more intelligent systems, the importance of DL is set to increase in the years to come.

However, DL is computationally expensive. Researchers have shown in theory and practice that the progress of DL in a wide variety of applications is strongly reliant on increases in computing power [52]. As shown in Fig. 1-1, traditional machine learning methods, e.g., linear regression, which use a small number of parameters, have their performance plateau at a low level as available computation (and amount of data) increases. In contrast, deep learning methods, which use an enormous number of parameters, achieve higher performance at the cost of more computation (and data). If we project this dependence into the future, it becomes evident that continuing on

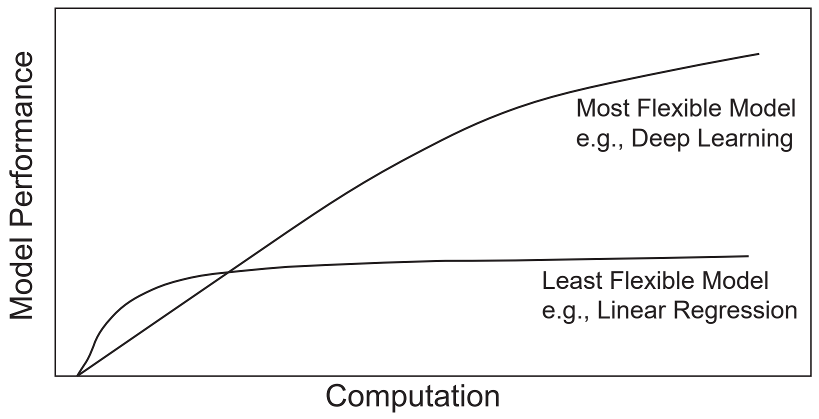


Figure 1-1: Implications of model performance versus computation of deep learning and linear regression [52].

the same path is unsustainable given the current energy cost of computation. This projection is supported by a probabilistic analysis of the emissions from computing onboard a global fleet of autonomous vehicles in the future. The model predicts that the computing power of 1 billion autonomous vehicles with 1-hour drive per day leads to the emission of all data centers on earth [50]. Even in the near term, the computing cost of DL is prohibitive. Quoted OpenAI, the popular DL tool ChatGPT [39] caused "eye-watering" computing cost when it hit 1 million users[1]. The increasing demand for computing power poses significant challenges to the growth of DL applications.

As a result, improving the energy efficiency of DL workloads while preserving their accuracy and meeting application throughput requirements becomes very important. The objectives of this thesis are to propose algorithm, architecture, and circuit co-design techniques to tackle this challenge, and experimentally evaluate them through simulation and/or chip tapeout and measurements.

1.2 Background

This section presents a general background and the terminology used in this thesis.

Convolution: Fig. 1-2 illustrates a general convolution layer, which convolutes input activation (IA) with weight, generates output activation (OA) and optionally

[1]https://analyticsindiamag.com/chatgpt-hits-one-million-users-burns-in-millions/

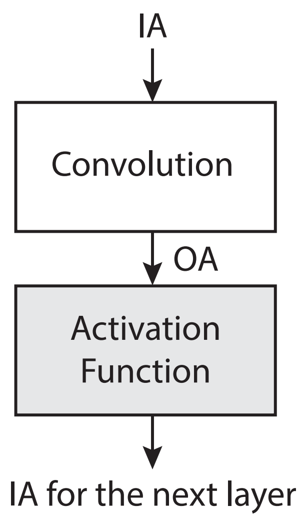


Figure 1-2: Illustration of a convolution layer. Some convolution layers do not have an activation function and thus OA equals IA for the next layer.

applies an activation function to OA to generate IA for the next layer. OA refers to the direct output of convolution instead of the output of the activation function in this thesis. Partial sum (PSum) refers to the partially computed OA. Feature map (FMap) refers to the activations in a NN, including both IAs and OAs. Illustration of ordinary convolution and depth-wise convolution [24] is shown in Fig. 1-3 along with the annotation of tensors and dimensions used in this thesis. The fully connected layer can be treated as a special case of ordinary convolution with FH, FW, IH, IW, OH and OW equal to 1.

Dataflow, mapping and loop nest: Sze *et al.* introduce *dataflow* to represent the ordering of calculations in convolution and which calculations run in parallel [51]. We split the dataflow into two parts, including *spatial mapping* which represents the parallel calculations, and temporal mapping which represents the sequential ordering of calculations. To precisely describe a dataflow, Sze *et al.* introduce *loop nests* [51]. In this thesis, we use the following syntax to represent a loop in temporal mapping,

```
For each dimension:
```

and the following syntax to represent a loop in spatial mapping.

```
Parallel_for dimension:
```

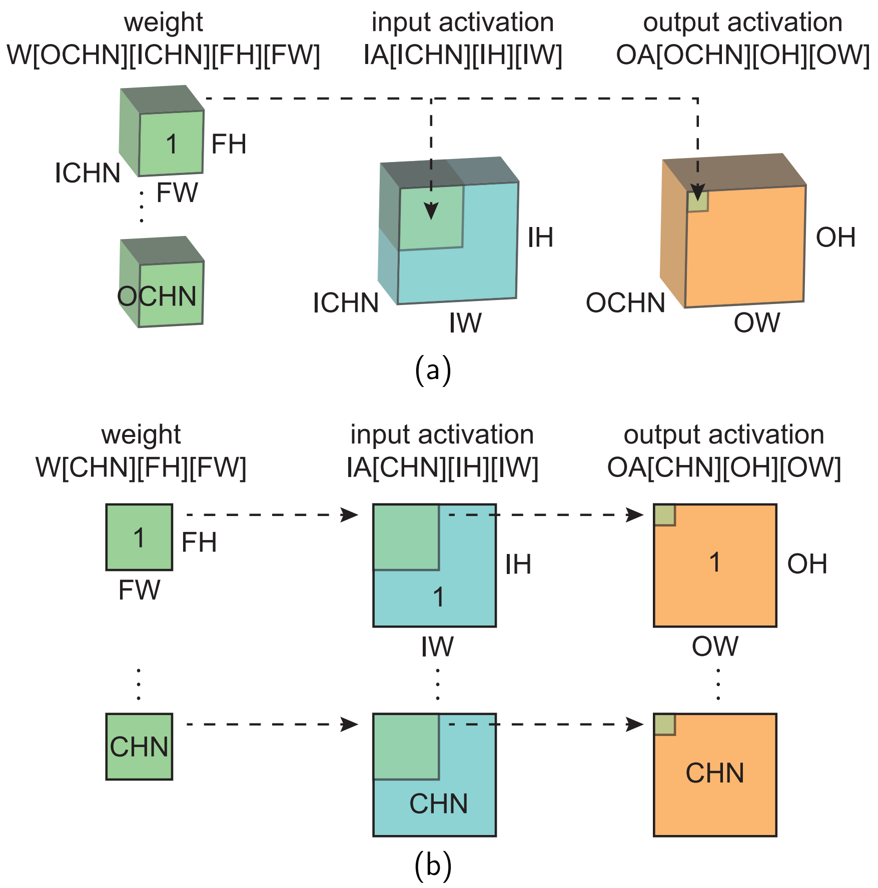


Figure 1-3: Illustration of (a) ordinary convolution and (b) depth-wise convolution and the annotations used in this thesis.

Although loop nests define how accelerators handle the computation, it does not show how data are buffered. To describe data buffering schemes of NN accelerators, we add read and write annotations of storage elements in the loop nests, such as the following.

```
Weight_L1_buffer.read()
Weight_L0_buffer.write()
```

Coordinate and position: *Coordinate* refers to the coordinate of each datum in the tensor. They are used to index the tensor and represent how convolution should be done mathematically. *Position* refers to the position of each datum in the memory relative to the base of this data block. They represent how to access the memory in hardware accelerators. If a data block is sequentially stored in the memory

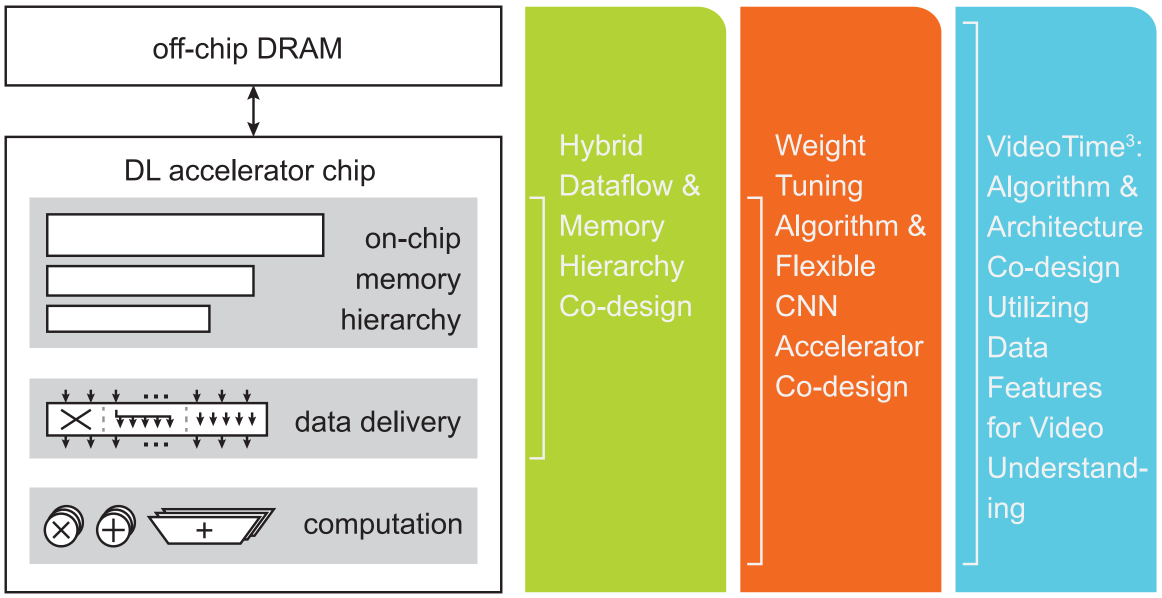

off-chip DRAM
DL accelerator chip
on-chip memory hierarchy
data delivery
computation
Hybrid Dataflow & Memory Hierarchy Co-design
Weight Tuning Algorithm & Flexible CNN Accelerator Co-design
VideoTime^3: Algorithm & Architecture Co-design Utilizing Data Features for Video Understanding

Figure 1-4: An overview of a general deep learning system and a summary of the thesis.

without compression, the position can be calculated from the coordinate based on the storing sequence and the dimensions of the data block. For 1-D arrays, the position is equal to the coordinate. If a data block is as a compressed form in the memory, the relationship between the position and the coordinate is not straightforward. It can be influenced by many factors, such as the sparsity pattern and the compression algorithm.

1.3 Thesis Overview

In this thesis, we focus on reducing energy consumption and improving energy efficiency across various system components while ensuring that we meet the required levels of accuracy and performance. The major components in a general DL accelerator are on-chip memory hierarchy, data delivery fabrics, and computing units. The DL accelerator may need off-chip storage if the processing data are too large to fit on chip. We take an algorithm, hardware architecture and circuit co-design approach to optimize different system components as summarized in Fig. 1-4.

Hybrid Dataflow and Memory Hierarchy Co-design: In Chapter 2, we focus on memory hierarchy design and dataflows to reduce the associated energy consumption. We propose hybrid dataflows that provide an option to tradeoff the reuse opportunities of weight and PSum in different memory hierarchies. We evaluate how the proposed hybrid dataflows affect the energy efficiency and other characteristics of two system designs with different technologies. In both systems, our proposed hybrid dataflows improve energy efficiency by around 1.5x – 2x and show significant improvements in performance per area and energy-delay product (EDP).

Weight Tuning Algorithm and Flexible CNN Accelerator Co-design: In Chapter 3, we propose a weight tuning algorithm to reduce the energy consumption of on-chip components—the memory access, data delivery, and computation—associated with weights. It focuses on reducing the weight-related switching activity through optimization of the bit representation of weights. In addition, we co-design a standalone deep learning accelerator with a specialized datapath and potentially applicable custom SRAM for the algorithm.

Algorithm and Architecture Co-design Utilizing Data Features for Video Understanding: In Chapter 4, we take the entire system into consideration and co-design the algorithm and architecture for video understanding applications utilizing their data features. Our design features the capability of temporal modeling for higher accuracy, the utilization of temporal redundancy to improve energy efficiency, and achieving single-frame latency for real-time applications, such as autonomous vehicles and AI drones. Our work also provides an extension to the existing sparsity handling taxonomy with the proposed map-guided sparse convolution and decoupled metadata generation.

Chapter 2

Hybrid Dataflow and Memory Hierarchy Co-design

As discussed in the previous chapter, it is important to keep improving the energy efficiency of NN accelerators. In this chapter, we focus on reducing the energy consumption of memory access and data delivery through the design of efficient dataflow and memory hierarchy.

The chapter is organized as follows. First, we analyze the energy consumption of memory access and data delivery in existing NN accelerators, point out the importance of energy reduction for memory access and data delivery, and summarize the related prior works. Then, we introduce our proposed hybrid dataflows and their corresponding memory hierarchy. Later, we present the evaluation results of our proposed techniques on several systems with different technologies followed by a summary of this chapter.

2.1 Introduction and Motivation

Optimizing memory access and data delivery is an important part of neural network (NN) accelerator design. Fig. 2-1 describes the contribution of memory access and data delivery to the overall power consumption of various NN accelerators. In terms of target workload, Eyeriss [6] and QUEST [53] are optimized for dense NNs, while

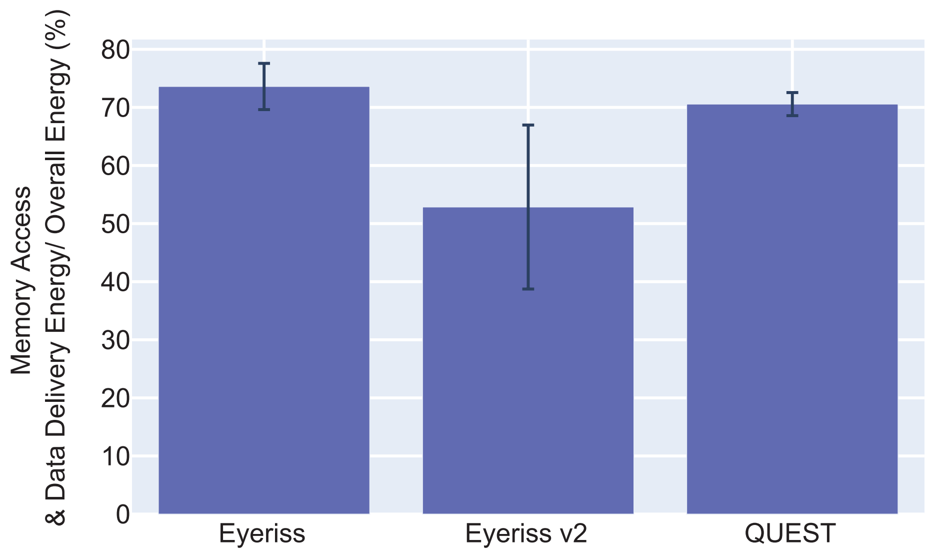


Figure 2-1: Memory access and data delivery energy is a significant part of the total NN accelerator energy. This figure only includes the reported on-chip energy consumption. Off-chip data access is more energy-consuming than on-chip memory access [23]. The energy ratio is obtained from the energy breakdown graphs/tables of prior NN accelerators running various workloads. We exclude the clock network energy from our analysis for Eyeriss [6] and Eyeriss v2 [7]. All the on-chip energy consumption is included except the "CLK and MISC." part in Fig. 18 of QUEST [53].

Eyeriss v2 [7] is designed for sparse NNs. In terms of the memory system, Eyeriss [6] and Eyeriss v2 [7] use regular DRAM and SRAM based systems, while QUEST [53] proposes a unique 3D-stacking SRAM with inductive coupling technology. As shown, the energy of memory access and data delivery takes up a significant portion of the total energy consumption of different NN accelerators.

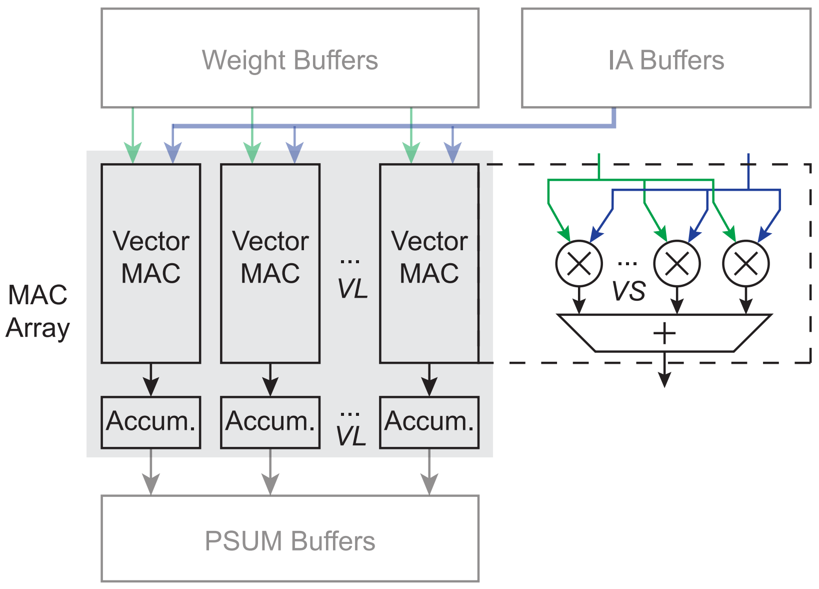
Extensive prior works have proposed many techniques to reduce the energy consumption of memory access and data delivery in NN accelerators, and some works investigated the factors influencing memory access and data delivery. Yu-Hsin *et al.* was pioneered in proposing a dataflow taxonomy and the convolution loop nest representation to systematically analyze the memory access and data delivery in NN accelerators [5, 51]. X. Yang *et al.* distinguished the impact of spatial mapping from that of temporal mapping in the dataflow. They showed that spatial mapping does not lead to much impact on the overall energy consumption as long as the temporal mapping and memory hierarchy are fully optimized [62]. That emphasizes the importance of the choice of memory allocation for each data type along with the loop permutation and temporal unrolling in the convolution loop nest. Prior work proposed weight stationary (WS), input stationary (IS), and output stationary (OS) dataflow corresponding to specific loop permutations that have weight/input/output-related dimensions in the outermost loops respectively. They result in a maximum reuse opportunity of weight/IA/OA respectively at the cost of losing reuse opportunities of other data types.

Based on prior work, we propose hybrid dataflows that offer balanced reuse opportunities for different data types at different levels of memory hierarchies. To demonstrate the impact of our proposed techniques, we evaluate them on several systems with different technologies, including a system with off-chip DRAM using 16nm FinFET technology and a fully-on-chip NN accelerator with embedded magnetoresistive RAM (eMRAM) in 22-nm/28-nm CMOS technology. It is shown that our proposed output stationary with local weight stationary (OS-LWS) dataflow delivers around 1.5x – 2x improvements in the energy efficiency of all systems evaluated and significant improvements in performance per area and EDP. Part of the work was done during the author's internship at the ASIC/VLSI research group at NVIDIA.

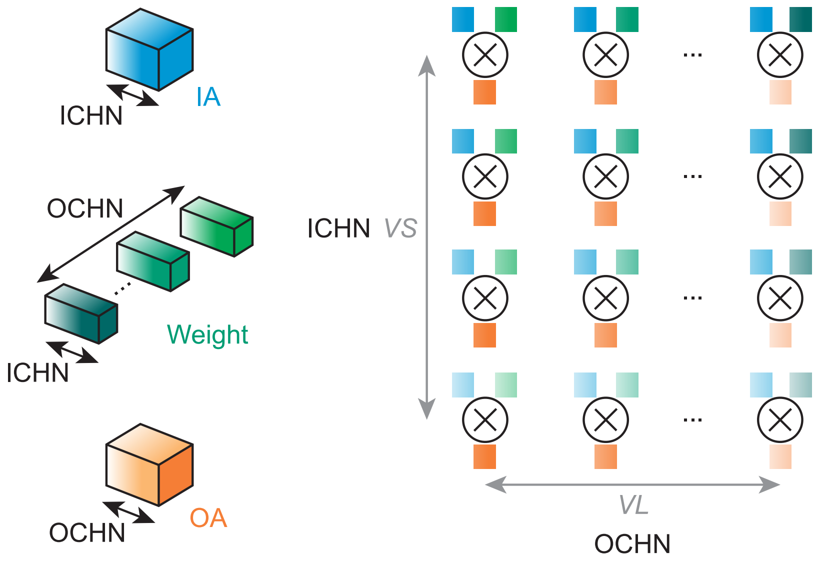
2.2 Hybrid Dataflow and Memory Hierarchy

This section presents our proposed hybrid dataflows, WS with local OS (WS-LOS) and OS with local WS (OS-LWS), which 1) provide designers the capability to balance the reuse of weight and PSum/OA based on the characteristics of the storage elements in the chosen technology, 2) enrich the design space for the architecture exploration framework to achieve potentially better energy efficiency and performance/area.

The baseline spatial datapath architecture and spatial mapping of the proposed dataflows are shown in Fig. 2-2. The spatial datapath architecture is composed of sev-



(a)



(b)

Figure 2-2: The baseline (a) spatial datapath architecture and (b) spatial mapping (to better illustrate the spatial mapping, we only show the multipliers from the MAC array in this figure). *VS*: the number of multipliers in a vector MAC; *VL*: the number of vector MACs and accumulators in the MAC array; accum.: accumulator. The colored squares indicate the mapping of IA, weight, and OA across the channel dimensions.

eral vector MACs and accumulators. The vector MAC consists of several multipliers to generate the products of weights and IAs, and an adder tree to spatially accumulate the products (PSums). The accumulator is used for the temporal accumulation

Table 2.1: Summary of reuse per read/write of L1 buffers and reuse per write of L0 registers in the WS, OS, and IS dataflows.

Data Type	Storage Unit	WS	OS	IS
Weight	L1 buffer	OH×OW	0	0
	L0 register	OH×OW	0	0
PSum/OA	L1 buffer	0	FH×FW×ICHN	0
	L0 register	0	0	0
IA	L1 buffer	0	0	FH×FW×OCHN
	L0 register	0	0	FH×FW×OCHN

of PSums locally before storing them in PSum/OA buffers. We spatially map the input and output channels to the MAC array for parallel computing. With high-level architecture modeling, prior work demonstrated that spatial mapping only has a limited impact on overall energy consumption, and channel-wise spatial mapping is good for the utilization of computing units as NNs usually have a large channel size [62]. That justifies our choice of spatial mapping. Moreover, the spatial accumulation of part of the PSums enabled by our vector MACs is shown to be more energy efficient than temporal accumulation as there is no need to access the accumulation register.

With this spatial architecture and spatial mapping, conventional dataflows—WS, IS, and OS—can be implemented as shown in Fig. 2-3 with a memory hierarchy that has one level of buffer for each data type, weight and IA registers at the inputs of MAC array, and PSum accumulation registers in the accumulators. The reuse opportunity for each data access to each buffer is summarized in Table 2.1. For WS, all the weight-related dimensions are in the outer loops and thus each weight buffer read and weight register write gets reused over the OH and OW loops while other data keep being accessed in every loop. Similar analysis can be applied to other conventional dataflows. For OS dataflow, the PSum accumulation register needs to be accessed each cycle for temporal accumulation, although all the PSum-related dimensions are in the outermost loops.

To balance the reuse opportunities across multiple data types, we propose hybrid

```
For each OCHN/ochn_parallel:
  For each FH:
    For each FW:
      For each ICHN/ichn_parallel:
        weight_l1_buf.read()
        weight_l0_reg.wr_rd()
        For each OH:
          For each OW:
            ia_l1_buf.read()
            ia_l0_reg.wr_rd()
            psum_l1_buf.read()
            Parallel_for ichn_parallel:
            Parallel_for ochn_parallel:
              MAC & psum_l0_reg.accum()
            psum_l1_buf.write()
```

(a)

```
For each OCHN/ochn_parallel:
  For each OH:
    For each OW:
      For each FH:
        For each FW:
          For each ICHN/ichn_parallel:
            ia_l1_buf.read()
            ia_l0_reg.wr_rd()
            weight_l1_buf.read()
            weight_l0_reg.wr_rd()
            Parallel_for ichn_parallel:
            Parallel_for ochn_parallel:
              MAC & psum_l0_reg.accum()
      psum_l1_buf.write()
```

(b)

```
For each IH:
  For each IW:
    For each ICHN/ichn_parallel:
      ia_l1_buf.read()
      ia_l0_reg.wr_rd()
      For each OCHN/ochn_parallel:
        For each FH:
          For each FW:
            psum_l1_buf.read()
            weight_l1_buf.read()
            weight_l0_reg.wr_rd()
            Parallel_for ichn_parallel
            Parallel_for ochn_parallel
              MAC & psum_l0_reg.accum()
            psum_l1_buf.write()
```

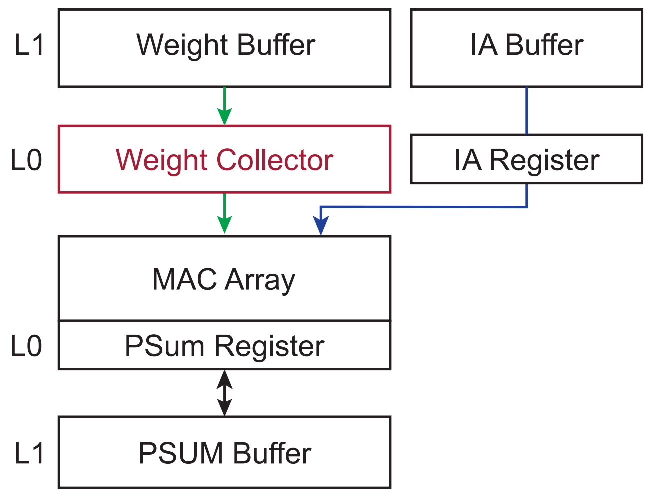
(c)

Figure 2-3: Examples of (a) WS, (b) OS, and (c) IS dataflow with L1 buffers and L0 registers. wr_rd: write and bypass read or read. accum: accumulation.

dataflows—WS-LOS and OS-LWS—and introduce new storage elements.

```
For each OCHN/ochn_parallel:
  For each FH:
    For each FW:
      For each ICHN/ichn0:
        weight_l1_buf.read()
        weight_l0_collector.write()
        For each OH:
          For each OW:
            psum_l1_buf.read()
            For each ichn0/ichn_parallel:
              weight_l0_collector.read()
              ia_l1_buf.read()
              ia_l0_reg.wr_rd()
              Parallel_for ichn_parallel:
              Parallel_for ochn_parallel:
                MAC & psum_l0_reg.accum()
            psum_l1_buf.write()
```

(a)



(b)

Figure 2-4: Illustration of WS-LOS (a) dataflow and (b) memory hierarchy. The differences between WS dataflow are highlighted in red.

Weight Stationary with Local Output Stationary

The weight stationary with local output stationary (WS-LOS) dataflow and its memory hierarchy implementation are shown in Fig. 2-4. Based on WS dataflow, we move part of the input channels to the innermost temporal loop and those input channels are temporally accumulated in the PSum registers without accessing the PSum buffer. Instead of having single-entry L0 weight registers at the input of the MAC array, we expand it to a $ichn0/ichn\_parallel$-entry weight collector to temporally hold $ichn0 \times ochn\_parallel$ weights during the OH, OW and $ichn0$ loops. In this way, reuse per weight buffer read stays the same as the WS dataflow while reuse per PSum buffer read goes up to $ichn0$ at the cost of no reuse opportunity for the weight collector. A summary of the reuse opportunity can be found in Table 2.2.

Output Stationary with Local Weight Stationary

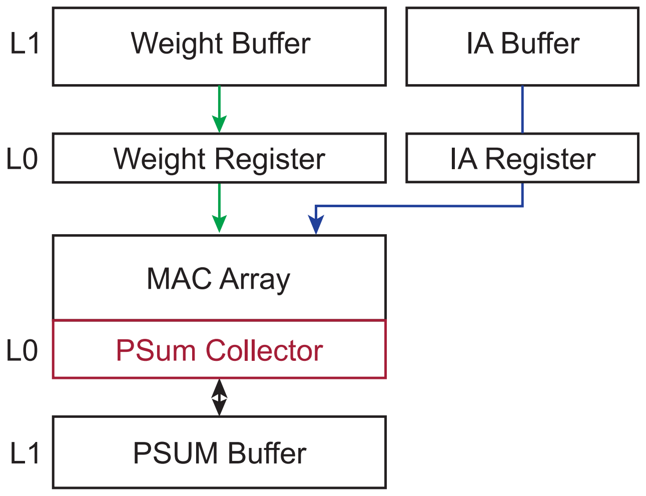
The output stationary with local weight stationary (OS-LWS) dataflow and its memory hierarchy implementation are shown in Fig. 2-5. Based on OS dataflow, we move part of the input width to the innermost temporal loop, and the same weights are reused across those inputs. Instead of having single-entry L0 PSum registers for temporal accumulation in the MAC array, we expand it to be a $ow0$-entry PSum accumulation collector to temporally hold $ow0 \times ochn\_parallel$ PSums during FW, FW, $ICHN$ and $ow0$ loops. In this way, reuse per PSum buffer write stays the same as the OS dataflow while reuse per weight buffer read and weight register write goes up to $ow0$. The overhead is that the multi-entry PSum collector needs muxes and thus the energy consumption of read and write is higher compared to that of the single-entry PSum accumulation register. A summary of the reuse opportunity can be found in Table 2.2.

2.3 System 1 and its Evaluation

The previous section introduces the proposed hybrid dataflows on a MAC-array-based spatial datapath with channel-wise spatial mapping. That expands the design space with options to balance the reuse opportunities between different data types and storage elements. In this section, we briefly present an evaluation of the impact of our proposed techniques on the energy efficiency and performance per area of a DL

```
For each OCHN/ochn_parallel:
  For each OH:
    For each OW/ow0:
      For each FH:
        For each FW:
          For each ICHN/ichn_parallel:
            weight_l1_buf.read()
            weight_l0_reg.wr_rd()
            For each ow0:
              ia_l1_buf.read()
              ia_l0_reg.wr_rd()
              Parallel_for ichn_parallel:
              Parallel_for ochn_parallel:
                MAC & psum_l0_collector.accum()
      psum_l1_buf.write()
```

(a)



(b)

Figure 2-5: Illustration of OS-LWS (a) dataflow and (b) memory hierarchy. The differences between the WS dataflow are highlighted in red.

accelerator done in collaboration with Rangharajan Venkatesan, Yakun Sophia Shao, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Yanqing Zhang, Brian Zimmer, William J. Dally, Joel Emer, Stephen W. Keckler, and Brucek Khailany during the author's internship at

Table 2.2: Summary of reuse per read/write of L1 buffers/L0 collectors and reuse per write of L0 registers in WS-LOS and OS-LWS dataflows.

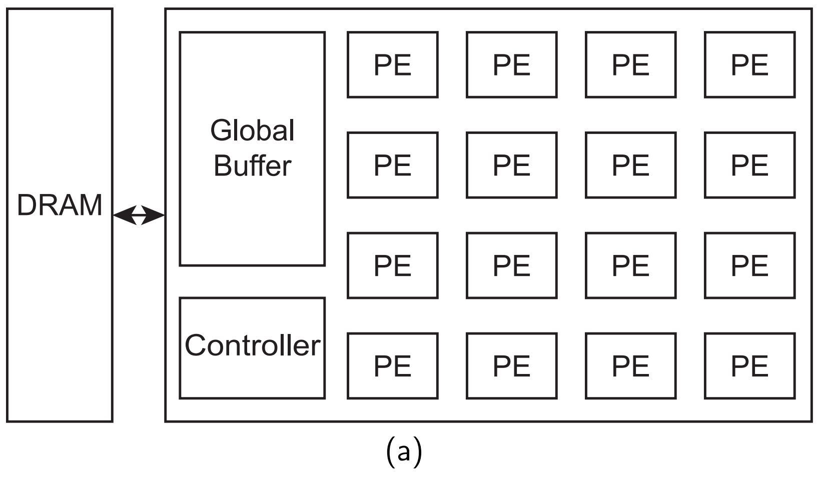
Data Type	Storage Unit	WS-LOS	OS-LWS
Weight	L1 buffer	OH×OW	ow0
	L0 collector/ L0 register	0*	ow0
PSum/OA	L1 buffer	ichn0	FH×FW×ICHN
	L0 collector/ L0 register	0	0*
IA	L1 buffer	0	0
	L0 register	0	0

The differences between the baseline WS/OS dataflow are highlighted in red.
*Collectors has multiple entries and thus muxes are needed per read/write. That increases the energy consumption per read/write compared to registers.

NVIDIA.

The proposed techniques were evaluated with a DL system shown in Fig. 2-6a. It comprises a processing element (PE) array as the main workhorse, a global buffer for data transfer between DRAM and chip, and a controller. The MAC-array-based spatial datapath is distributed in the PEs with local buffers and local controllers. The PE architecture is shown in Fig. 2-6b. SRAMs are used for L1 buffers and standard cells are used for the L0 registers/collectors.

MAGNet [54] shown in Fig. 2-7 is developed to explore the design space of PEs and search for the best design-time and runtime parameters for given workloads. The design space of PEs is shown in Table 2.3. The MAGNet designer takes that as the input and generates a set of design-time parameters. An RTL generator, which consists a systemC-based parameterized architecture template, uses those design-time parameters to generate RTL [31]. The MAGNet mapper takes in the design-time parameters and workloads, and searches for the best runtime parameters, such as the tile size, for the accelerator. Given the runtime parameters, a trace generator produces the configuration bits and data traces for simulation. Traditional RTL synthesis, place-and-route, and simulation tools are used to evaluate the power, performance,



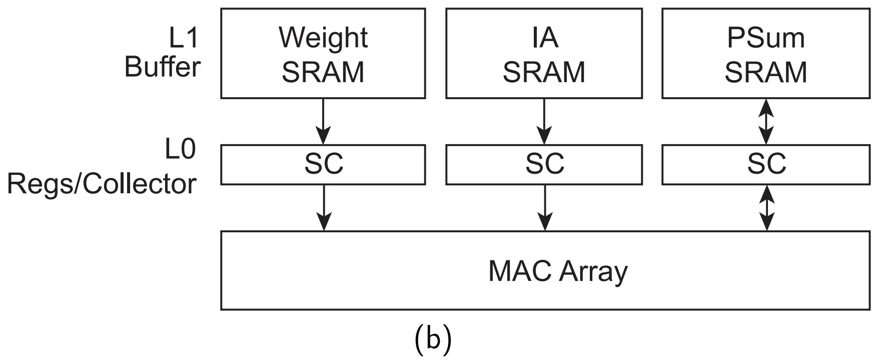


Figure 2-6: (a) The overall architecture of system 1. PE: processing element. (b) The PE architecture. The MAC array is shown in Fig. 2-2a. SC: standard cell.

and area of the generated accelerator. The designer sets the design goal for the MAGNet tuner and uses it to tune the designer and mapper to efficiently explore the design space guided by the power, performance, and area of generated designs.

The design is implemented with TSMC 16-nm FinFET technology. It is optimized across a workload of three NNs—AlexNet [33], ResNet [21], and DriveNet [3]—weighted by their number of operations. Fig. 2-8 shows the evaluation results of a baseline accelerator optimized with the conventional dataflows, and the most energy-efficient accelerator design with the proposed hybrid dataflows. As shown, our proposed hybrid dataflows improve energy efficiency by 1.75x and performance per area by 2.1x running ResNet. Fig. 2-9 shows the energy breakdown of the most energy-efficient PE with VL and VS of 16 under different dataflows. As shown, our proposed hybrid dataflows greatly reduce the energy of memory access compared to the baseline WS and OS dataflows.

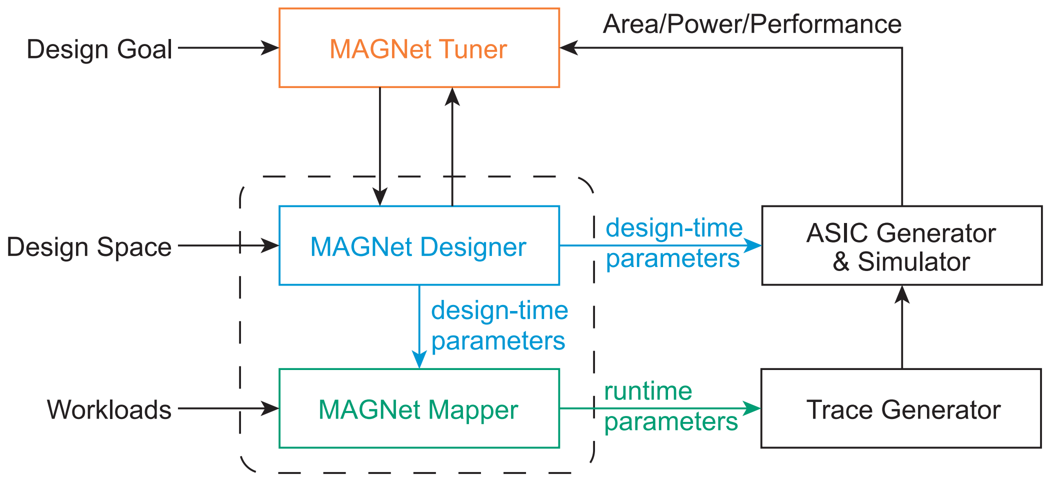


Figure 2-7: Overview of MAGNet framework [54]. Its capability to co-design the DL application is not used in our evaluation and thus is not included in this figure.

Table 2.3: The design space of system 1 [54]. A conventional design space contains WS and OS dataflow. The proposed hybrid dataflow expands the design space with WS-LOS and OS-LWS dataflows.

Bit-width	8-b weight/activation, 24-b PSum
VL/VS	4, 8, 16
Weight Collector Size	8 B – 2 KB
PSum Collector Size	8 B – 384 B
IA Buffer Size	2 KB, 8 KB, 16 KB
Weight Buffer Size	4 KB – 128 KB
PSum Buffer Size	1 KB – 6 KB
Global Buffer Size	64 KB
Target Frequencies	500 MHz, 1 GHz
Supply Voltage	0.6 V
Dataflow	WS, OS, WS-LOS, OS-LWS

2.4 System 2 and its Evaluation

The last section presents the benefits of hybrid dataflows on energy efficiency and performance per area on a 16-nm accelerator with a memory system composed of off-chip DRAM and on-chip SRAM. As discussed in Section 2.2, the technology and its characteristics of storage units affect the choice of optimal dataflow and temporal mapping. In this section, we expand our analysis to a fully on-chip NN accelerator with emerging eMRAM in 22-nm technology. In the evaluation, we compare

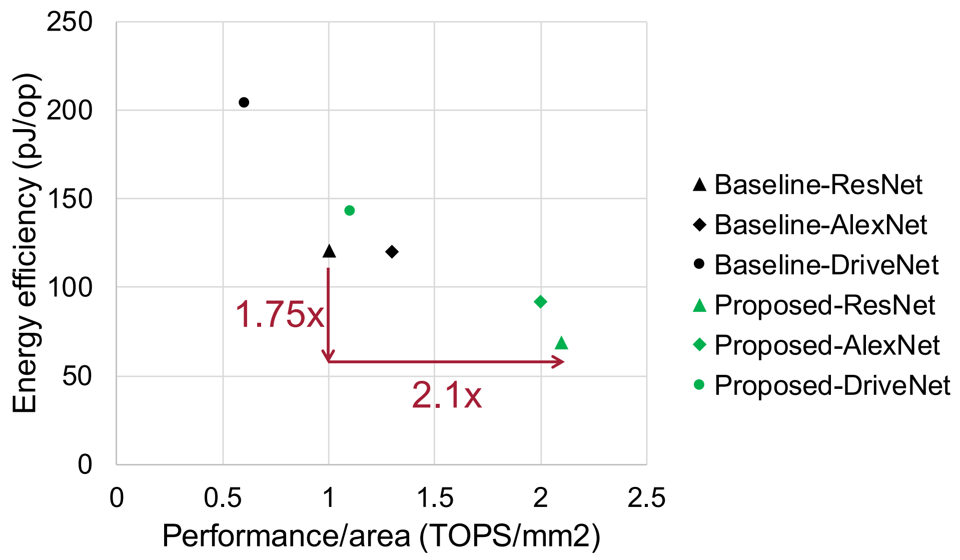


Figure 2-8: Comparison of energy efficiency and performance per area of the optimal accelerators with conventional dataflows and the proposed dataflows. The proposed hybrid dataflows greatly improve energy efficiency and performance per area.

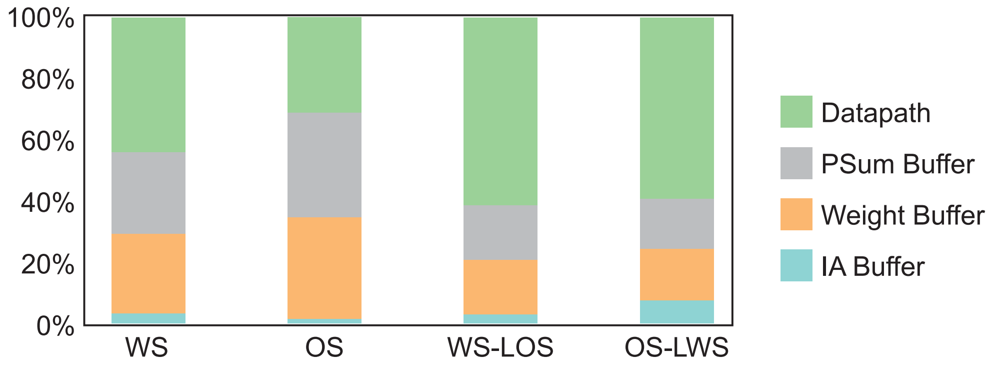
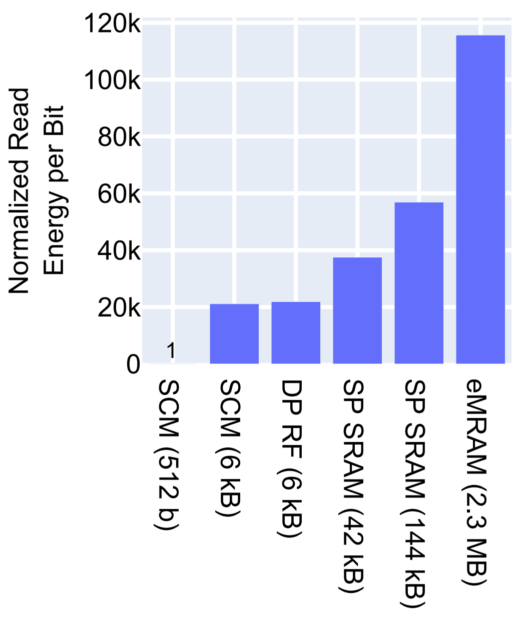
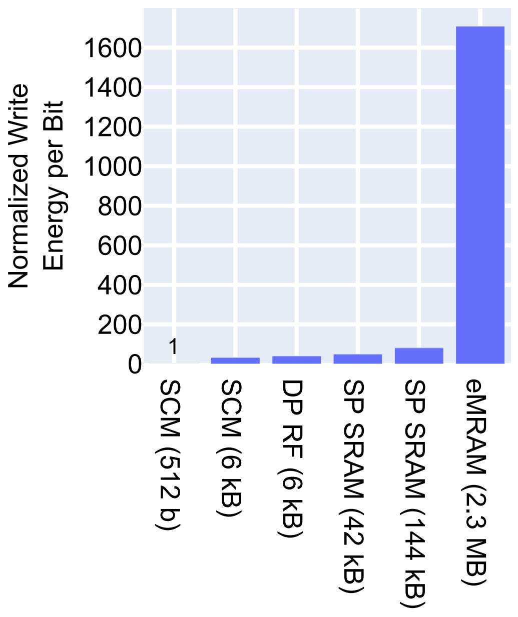


Figure 2-9: The energy breakdown of a PE with VL and VS of 16.

the optimal design achieved by our proposed techniques with the optimal design of conventional dataflow to show the benefits of our hybrid dataflows. The evaluation also serves as an early-stage exploration for their application with emerging embedded non-volatile memory for edge computing applications, such as automotive and Industry 4.0.



(a)



(b)

Figure 2-10: Normalized (a) read energy per bit, and (b) write energy per bit of 22-nm eMRAM and 28-nm standard cell memory (SCM) and SRAM relative to 512-b SCM. DP RF: the dual-port register file from 28-nm memory compilers (this thesis refers to it as SRAM). SP SRAM: single-port SRAM.

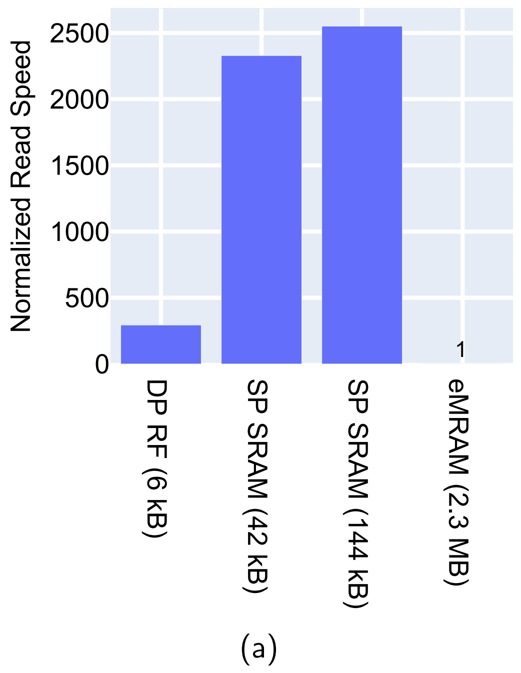
2.4.1 Technology

This system targets TSMC 22-nm technology with eMRAM. But as TSMC 22-nm memory compilers are not available for university access, we use 28-nm technology as an estimate in our evaluation.

eMRAM

STT-MRAM is a promising solution for next-generation embedded non-volatile memory. Technology developments have brought us logic-compatible and high-retention eMRAM [14, 44]. It helps keep all data on-chip which eliminates the need for energy-consuming off-chip data access. Moreover, it enables duty cycling to reduce the power consumption of applications that do not need to be always on and maintains NN parameters and boot codes locally avoiding the need for cloud access [25].

Compared to eFlash, which is difficult to scale any further, eMRAM reduces mask



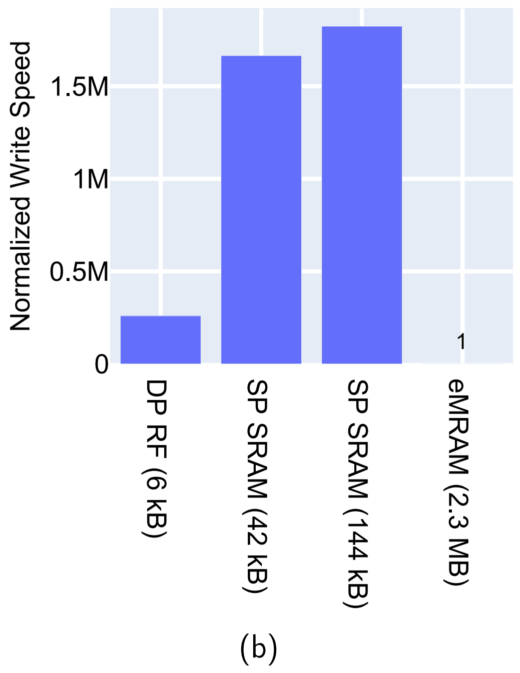
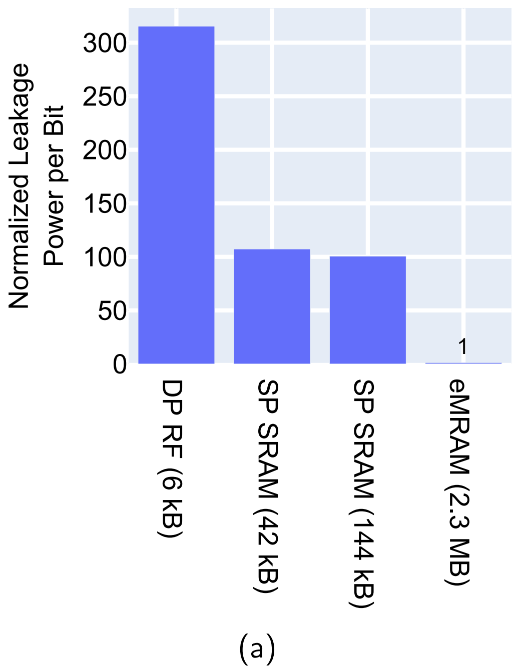


Figure 2-11: Normalized (a) read speed, and (b) write speed of 22-nm eMRAM and 28-nm SRAM relative to eMRAM. DP RF: the dual-port register file from 28-nm memory compilers (this thesis refers to it as SRAM). SP SRAM: single-port SRAM.

adders by over 2x, achieves similar read/write speed and read power, and delivers over 100x higher retention while leading to around 2x higher write power [14]. Compared to off-chip DRAM, MRAM achieves similar bandwidth with 44x less access energy [44]. Fig. 2-10, 2-11, and 2-12 compare eMRAM over 28-nm standard cell memory (SCM) and SRAM. As shown, eMRAM delivers over 2x higher area density and over 100x less leakage power compared to SRAM while having much higher access energy and lower access speed (the inverse of cycle time), especially for writes. Moreover, the minimum macro capacity of eMRAM is over 1000x compared to that of SRAM. The characteristics of eMRAM affect how it can be used in the DL accelerator, which will be discussed in the next section.

2.4.2 Overall Architecture

Fig. 2-13 shows the overall architecture of the accelerator chip. A monolithic MAC array is used. More complicated architecture with multiple MAC arrays distributed



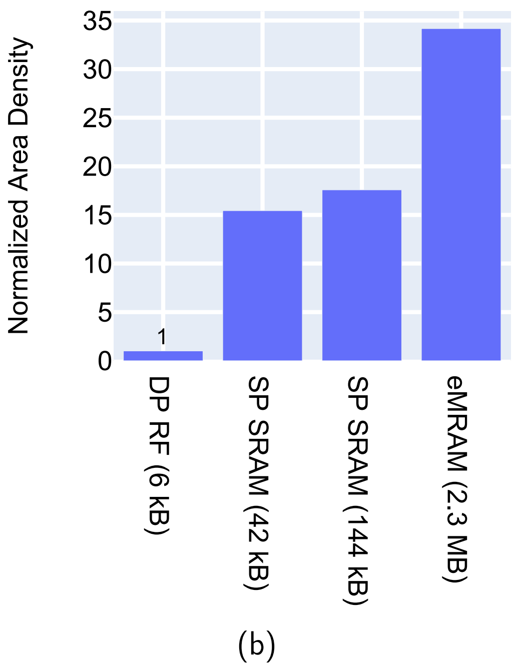


Figure 2-12: Normalized (a) leakage power per bit and (b) area density of 22-nm eMRAM and 28-nm SRAM. DP RF: the dual-port register file from 28-nm memory compilers (this thesis refers to it as SRAM). SP SRAM: single-port SRAM.

in a processing element array, like Simba [48], is left as future work. The lowest-level storage elements are either IA/weight/accumulation registers or a weight/PSum collector for hybrid dataflows. Above that, two levels of buffers are implemented for each data type. eMRAM is used as the last level buffer for weights for the following reasons—1) weights need to be kept in non-volatile memory during power down, 2) the high area density and low leakage of eMRAM make large memory less costly compared to SRAM, 3) the reuse opportunity per access can be high in the last level storage reducing the read bandwidth requirements. The last level buffer for IA and OA is made of SRAM instead of eMRAM, although the above 2) and 3) also hold in this case. The reason is that it is challenging to supply enough write bandwidth with eMRAM for medium/large-scale NN processing as indicated in Fig. 2-11b. Tiny NNs with several thousand parameters or even fewer are not included in the target workload as they cannot fully utilize the capacity of eMRAM available, which is in the MB range. Furthermore, the write energy of eMRAM is much higher than

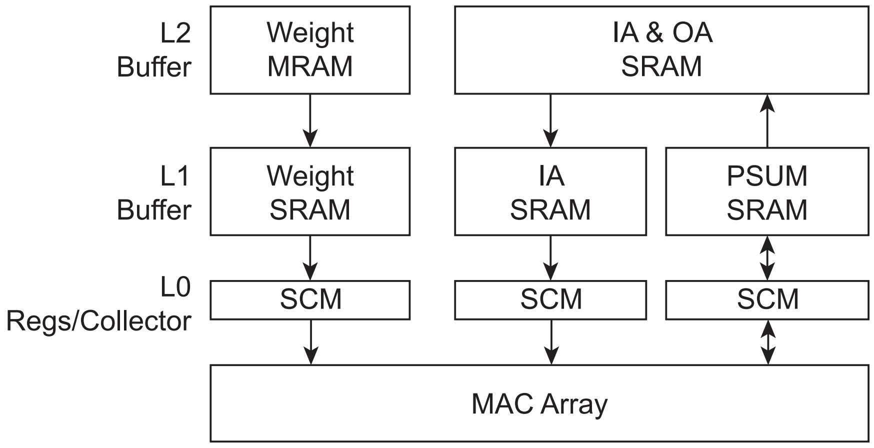


Figure 2-13: Overall system architecture. The system is fully on-chip with eMRAM.

Table 2.4: The design space of the system.

Bit-width	8-b weight/activation, 24-b PSum
VL/VS	16
Weight Collector Depth	2, 4, 8
PSum Collector Depth	4, 8, 32
L1 IA Buffer Depth	128, 256, 512, 1024
L1 Weight Buffer Depth	128, 256, 512, 1024
L1 PSum Buffer Depth	128, 256, 512, 1024
L2 Weight Buffer Size	2.3 MB
L2 IA/OA Buffer Size	0.3 MB
Target Frequencies	100 MHz
Supply Voltage	0.8 V/0.9 V
Outer Dataflow	OS
Inner Dataflow	WS-a, WS-b, OS-a, OS-b, WS-LOS-a, WS-LOS-b, OS-LWS-a, OS-LWS-b

that of SRAM shown in Fig. 2-10b, which may lead to a nonnegligible increase in system energy consumption. We adopt Buffets [42] for weight and IA L1 buffers and ping-pong buffer for the PSum L1 buffer.

The design time parameters are summarized in Table 2.4. We set the size of the L2 buffers to store the medium-size NNs, such as AlexNet [33], fully on-chip. We have two levels of buffers and temporal unrolling is applied to the convolution loop nest breaking it into two levels. OS dataflow is used at the outer level so that no read operation for PSum is needed from the L2 buffer. That keeps the buffer simple and area efficient—a single-port SRAM for IA read and a single-port SRAM for OA write are sufficient for the operations needed. The inner level has a large variety of choices. We expand the WS, OS, WS-LOS, and OS-LWS dataflows with two flavors using different buffering schemes for weights. We use OS-LWS dataflow as an example to illustrate the differences between the two weight buffering schemes. The details of other dataflows can be found in Appendix C.

Fig. 2-14 and Fig. 2-15 show the details of the OS-LWS-a and OS-LWS-b dataflow. Tiling is applied to each data type so that part of the data can be kept in L1 buffers for temporal reuse. The level 1 loops are similar to the one-level OS-LWS dataflow shown in Fig. 2-5a except that it includes the reads of L2 buffers and the writes of L1 buffers. The difference between OS-LWS-a and OS-LWS-b dataflow is the data buffered in the L1 weight buffer. OS-LWS-a dataflow buffers $FH \times FW \times ochn\_parallel \times ichn\_tile$ of weights, and each read of L2 buffer is reused for $oh\_tile \times ow\_tile$ times in the L1 buffer. OS-LWS-b dataflow buffers $FH \times FW \times ochn\_tile \times ICHN$ of weights, and each read of L2 buffer is reused for $OH \times OW$ times in the L1 buffer. OS-LWS-a and OS-LWS-b provide a tradeoff between the size of the L1 weight buffer, which impacts its area and energy per access, and the number of access to the L2 weight buffer, which impacts its energy consumption.

We evaluate the impact of the choice of inner-level dataflows on the overall performance, energy efficiency and EDP. In the rest of this chapter, the dataflows mentioned refer to the inner-level dataflow.

2.4.3 Evaluation Setup

The design space exploration framework is shown in Fig. 2-16. Instead of writing RTL codes or a cycle-accurate model of the entire system with all possible design-time parameters including a large variety of dataflows, we build an analytical model, like Timeloop [40], to calculate the memory accesses of each buffer and the computing cycles of the MAC array based on design-time and runtime parameters of the under-

```
Level 2
1.  For each OCHN/ochn_tile:
2.    For each OH/oh_tile:
3.      For each OW/ow_tile:
4.        For each ICHN/ichn_tile:
Level 1
5.          ia_l2_buf.read()
6.          ia_l1_buf.write()
7.          For each ochn_tile/ochn_parallel:
8.            weight_l2_buf.read()
9.            weight_l1_buf.write()
10.           For each oh_tile:
11.             For each ow_tile/ow0:
12.               psum_l1_ping_buf.read()
13.               For each FH:
14.                 For each FW:
15.                   For each ichn_tile/ichn_parallel:
16.                     weight_l1_buf.read()
17.                     weight_l0_reg.wr_rd()
18.                     For each ow0:
19.                       ia_l1_buf.read()
20.                       ia_l0_reg.wr_rd()
21.                       Parallel_for ichn_parallel:
22.                       Parallel_for ochn_parallel:
23.                         MAC & psum_l0_collector.accum()
24.               psum_l1_ping_buf.write()
Level 2
25.       psum_l1_pong_buf.read()
26.       psum_l2_buf.write()
```

Figure 2-14: OS-LWS-a dataflow (inner-level).

lying architecture for fast early-stage design exploration. We characterize the energy and throughput of major design components, including SRAMs, eMRAM, SCM, and the MAC array, using TSMC memory compiler or RTL synthesis and simulation. Using the operational statistics (e.g., memory accesses and operation cycles) of major components in the design and their energy and throughput characteristics, we estimate the energy and runtime of the entire system. Given the estimated energy

```
Level 2   1.  For each OCHN/ochn_tile:
          2.    weight_l2_buf.read()
          3.    weight_l1_buf.write()
          4.    For each OH/oh_tile:
          5.      For each OW/ow_tile:
          6.        For each ICHN/ichn_tile:
Level 1   7.          ia_l2_buf.read()
          8.          ia_l1_buf.write()
          9.          For each ochn_tile/ochn_parallel:
          10.           For each oh_tile:
          11.             For each ow_tile/ow0:
          12.               psum_l1_ping_buf.read()
          13.               For each FH:
          14.                 For each FW:
          15.                   For each ichn_tile/ichn_parallel:
          16.                     weight_l1_buf.read()
          17.                     weight_l0_reg.wr_rd()
          18.                     For each ow0:
          19.                       ia_l1_buf.read()
          20.                       ia_l0_reg.wr_rd()
          21.                       Parallel_for ichn_parallel:
          22.                       Parallel_for ochn_parallel:
          23.                         MAC & psum_l0_collector.accum()
          24.               psum_l1_ping_buf.write()
Level 2   25.       psum_l1_pong_buf.read()
          26.       psum_l2_buf.write()
```

Figure 2-15: OS-LWS-b dataflow (inner-level).

and runtime, a design space explorer and a mapper search for optimal design-time parameters and the corresponding runtime parameters given a design target, e.g., best energy efficiency.

We use Alexnet [33] as our target workload. The overall energy efficiency (i.e., TOPS/W) and throughput (i.e., GOPS) of the entire workload are weighted sums of energy efficiency and throughput of each layer based on its number of operations.

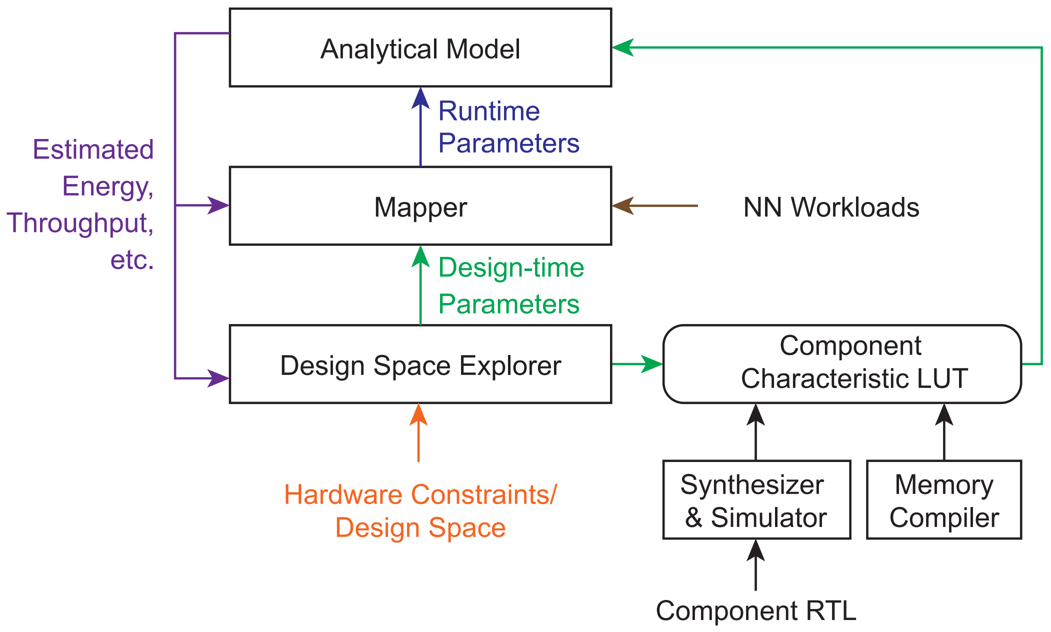


Figure 2-16: The design space exploration framework for system 2.

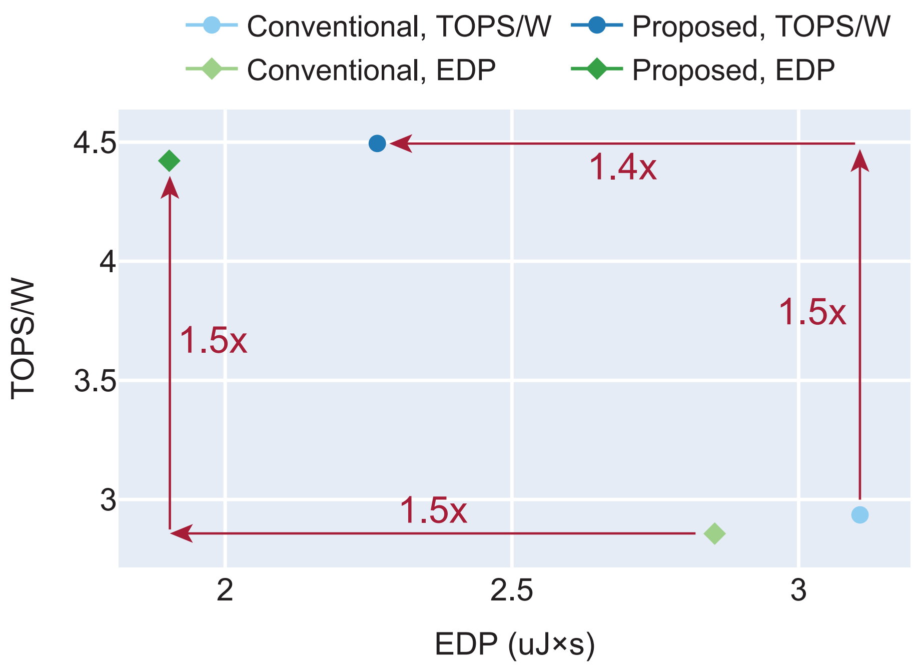


Figure 2-17: Comparison of the best energy efficiency and EDP achieved by conventional dataflows and proposed dataflows under different optimization targets—maximizing energy efficiency and minimizing EDP. The legend of (a): <design space>, <optimization target>.

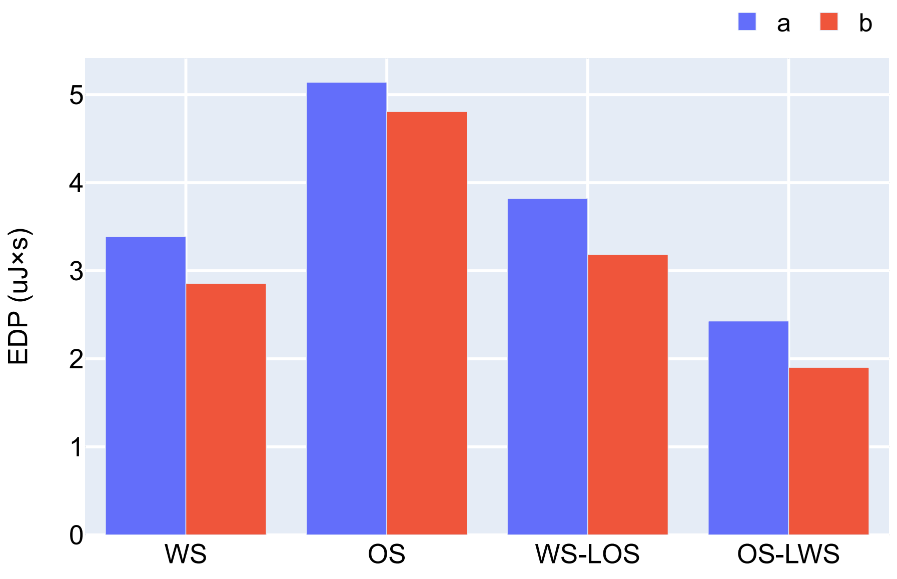
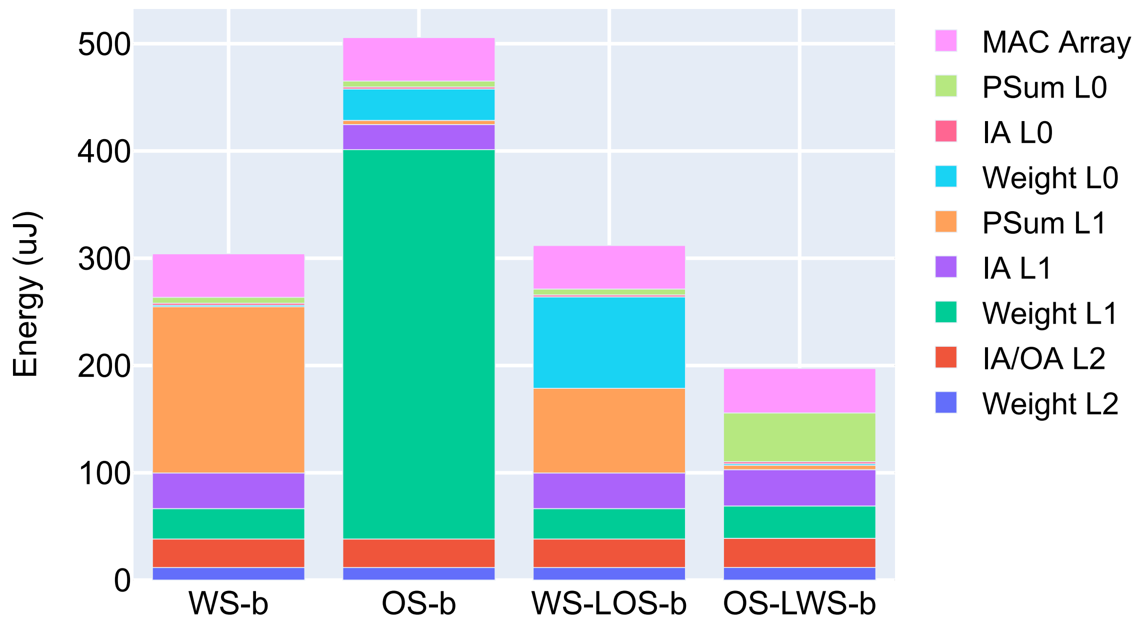


Figure 2-18: Comparison of EDP between different dataflows. The optimization target is minimizing EDP.

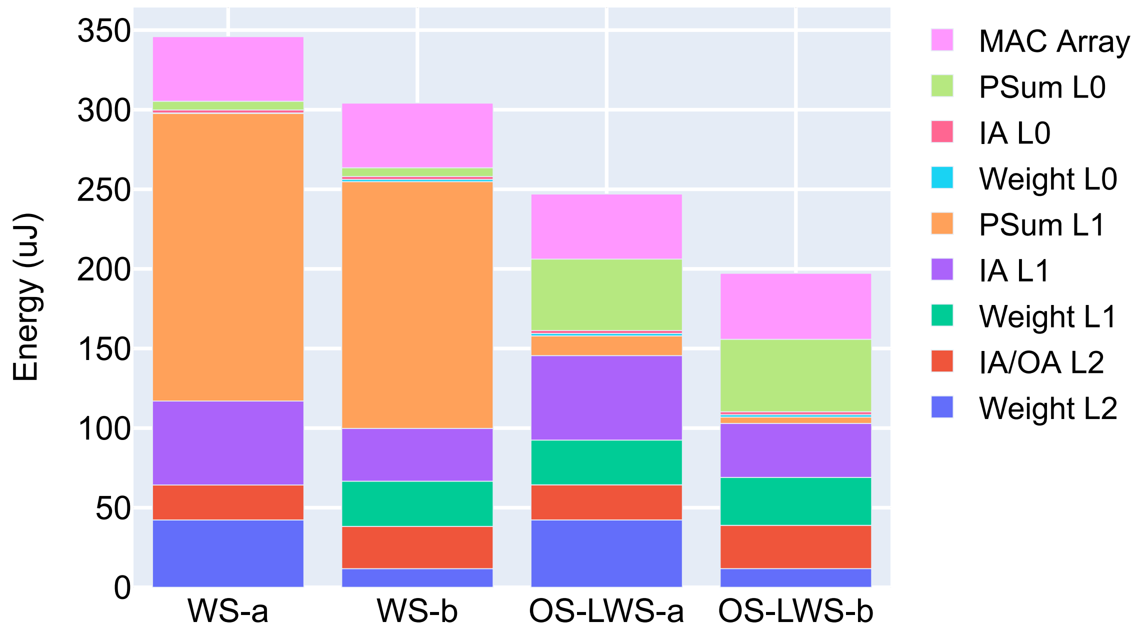
2.4.4 Evaluation Results

Fig. 2-17 compares the best energy efficiency and EDP achieved by the conventional dataflows and our proposed hybrid dataflows. When we optimize for energy efficiency, our proposed hybrid dataflows achieve around 1.5x improvement in the optimal TOPS/W compared to that of the conventional dataflows. When we optimize for EDP, our proposed hybrid dataflows improve the optimal EDP by around 1.5x compared to the conventional dataflows. The attained throughput of all cases is at least 99% of the highest throughput achieved when optimizing for maximum throughput. As shown, our proposed hybrid dataflows deliver significant energy efficiency improvements and energy savings for the entire NN accelerator system with a multi-level memory hierarchy while maintaining adequate throughput.

Fig. 2-18 shows a comparison of the optimal EDP achieved by different dataflows. WS dataflow achieves much better EDP compared to the OS dataflow given our architecture and technology, and our proposed OS-LWS dataflow outperforms all other dataflows. The *b* version of dataflows, which achieves more reuse of weight L2 buffer read at the cost of a bigger weight L1 buffer as presented in Section 2.4.2, leads to better EDP compared to the *a* version of dataflows.



(a)



(b)

Figure 2-19: The energy breakdown of different dataflows. The results are from an optimization target of minimizing EDP.

To analyze the differences between those dataflows, we evaluate the energy breakdown of the NN accelerator. As shown in Fig. 2-19a, WS-b dataflow delivers lower energy consumption compared to OS-b dataflow as zero reuse of the PSum L1 buffer read/write leads to much less energy than zero reuse of the weight L1 buffer read. The OS-LWS-b dataflow consumes less energy than the WS-b and OS-b dataflows as it greatly reduces the energy of the PSum and weight L1 buffers with a small PSum L0 collector as presented in Section 2.2. OS-LWS-b dataflow consumes less energy than WS-LOS-b dataflow as accessing the PSum L0 collector is less energy-consuming than accessing the weight L0 collector. The reason is that the weight L0 collector has an IO size of $16 \times 16 \times 8$ to match the bandwidth of the MAC array, while the PSum L0 collector only has an IO size of 16×24.

Fig. 2-19b compares the energy breakdown of *a* and *b* versions of dataflows. The *b* version of dataflows significantly reduces the energy consumption of the weight L2 buffer as the reuse of L2 buffer access increases. Moreover, the energy consumption of the L1 IA buffer is largely reduced in the *b* version of dataflows. The reason is that oh_tile and ow_tile do not affect the number of access of the weight L2 buffer in the *b* version of dataflows and thus can be more flexibly adjusted. That results in smaller oh_tile and ow_tile and thus smaller ih_tile and iw_tile, which can be fit in a smaller L1 IA buffer and reduces its energy consumption. This emphasizes that the design choice of one buffer may affect the design choice and energy consumption of another buffer through mapping optimization.

As shown in Fig. 2-19, the energy consumption of L2 buffers takes up a small portion of total energy consumption. Besides that each access of L2 buffers has large reuse opportunities thanks to temporal tiling, an important reason is that eMRAM enables on-chip storage of weights and eliminates the need for energy-consuming off-chip memory access. As discussed in Section 2.4.1, off-chip DRAM access is shown to increase energy consumption by 44x compoared to eMRAM access. If we replace eMRAM of the OS-LWS-b case with off-chip DRAM, off-chip weight access can take up 96.5% of total energy consumption assuming runtime stays the same.

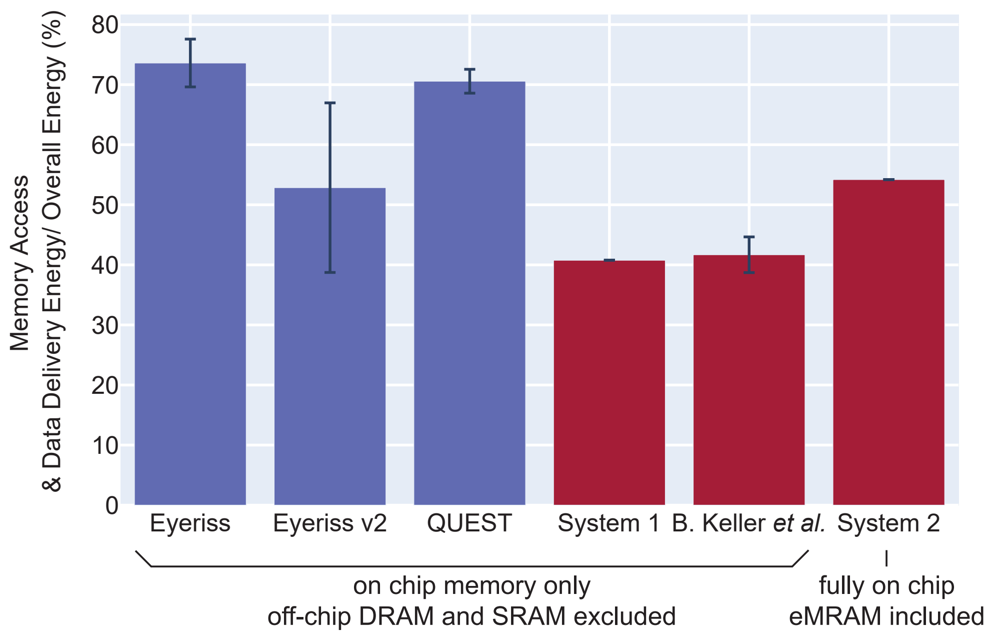


Figure 2-20: Comparison of overall proportion of on-chip energy used for memory access and data delivery of various NN accelerators, including Eyeriss [6], Eyeriss v2 [7], QUEST [53], B. Keller *et al.* [30] and the two systems presented in this chapter. System 2 is different from other works as it includes all memory hierarchy on chip. The energy consumption of off-chip memory hierarchies in other works is not included in the analysis.

2.5 Summary and Conclusions

This chapter shows the importance of reducing the energy consumption in memory access and data delivery for NN accelerators and presents our proposed techniques—hybrid dataflows and corresponding memory hierarchy—for this issue. We propose the OS-LWS dataflow with a PSum accumulation collector and the WS-LOS dataflow with a weight collector, which balances the reuse of weight and PSum/OA in different storage elements provided by the technology. Based on our evaluation of several systems with different technologies, our proposed hybrid dataflows significantly reduce the energy consumption of memory access and data delivery and the OS-LWS dataflow delivers around 1.5x–2x improvements in energy efficiency compared to conventional dataflows.

Besides the systems analyzed in this chapter, a more recent DL accelerator for transformers also adopts our proposed memory hierarchy with the PSum accumulation collector and a simplified OS-LWS dataflow for matrix multiplication [30]. Fig. 2-20 summarizes its overall proportion of on-chip energy used for memory access and data delivery along with other works. As shown, both the recent transformer accelerator (B. Keller *et al.*) and System 1 presented in this chapter, which applies our proposed OS-LWS dataflow, achieve a significant reduction in data access energy. Besides those systems with off-chip DRAM (not included in the energy analysis), we evaluate System 2 with all memory hierarchies on chip using eMRAM. It shows that our proposed techniques still delivers significant reduction in the overall proportion of energy used for memory access and data delivery compared to most prior works with their off-chip access energy excluded.

Chapter 3

Weight Tuning Algorithm and Datapath/SRAM Co-design for Flexible Fully-Integrated CNN Accelerator

The last chapter focuses on dataflow and memory hierarchy co-design. In this chapter, we expand our scope to an entire deep learning accelerator chip that runs small-footprint models fully on chip. We look into the data being processed and investigate how to manipulate them to save the energy consumption of the chip. This chapter presents a weight tuning algorithm that tweaks the bit representation of weights to lower the toggle count of weight sequences while preserving the accuracy. That reduces the switching activity of weight buses and a co-designed mixed representation datapath. With reduced switching activity, the power consumption of weight delivery and computation is lowered. Moreover, a low toggle count of weight read sequence leads to reduced read access energy of the weight buffer when SRAM with conditional pre-charge [10] is used. Besides the weight tuning algorithm, we highlight an architecture that is highly flexible for various NN structures.

This chapter is organized as follows. First, we provide an introduction to small-footprint NNs and their promising applications. Following that, we analyze the challenges of their accelerators, summarize the related prior work, and explain our unique contributions. Then, we present our weight tuning algorithm and the accelerator architecture. In the end, we explain our evaluation and measurement setup and show our evaluation results followed by a summary of this chapter. A more detailed explanation of the proposed algorithm can be found in M. Wang *et al.* [55]. This chapter highlights updated hardware design and analysis, and chip measurements.

3.1 Introduction and Motivation

Smart edge devices that support efficient NN processing have recently gained public attention. With algorithm development, previous work has proposed small-footprint NNs achieving high performance in various medium-complexity tasks, e.g. speech keyword spotting (KWS), human activity recognition, etc. Among them, convolution NNs (CNNs) achieve good accuracy [66]. This gives rise to the deployment of CNN models on edge devices that have limited storage due to their area and power constraints. Processing NNs on the edge has several benefits. For one, it reduces the amount of data transmission by handling part of the data locally instead of sending them all to the cloud computing unit. That can reduce the overall system power as data transmission can dominate the total power consumption of various systems, e.g., the sensor network [49]. For another, it can serve as a trigger for more complex and power-hungry downstream processing. In this way, the downstream system can be automatically switched off when not needed to save energy. Take KWS as an example. It is often achieved by small-footprint NNs running on edge devices. KWS filters out the noise and unrelated speech signals in the environment and only triggers the downstream speech recognition and/or natural language processing system, which uses complex models on the cloud when certain keywords are detected [16]. Thus, efficiently processing the small-footprint NNs on the edge devices is critical to the overall system performance. A hardware platform for edge devices should be (1) flexible to support various NN structures optimized for different applications; (2)

energy efficient to operate within the power budget; and (3) achieving high accuracy to minimize spurious triggering of the power-hungry downstream processing [56].

Both algorithms and accelerator designs for energy-efficient processing of CNNs have been proposed. On the algorithm side, quantization and model compression are the two main techniques. Quantization reduces the bit precision with the aim of lowering the data storage size and the complexity of the computation unit. However, some experiments show that quantizing NNs to extremely low bit-width, e.g. 1 bit, does not necessarily lead to model size reduction, because the model structure needs to be modified to retain the accuracy [18]. The use of 8-bit precision in weights generally achieves reliable performance without the need to modify the NN structure for classification tasks [28]. Model compression algorithms focus on reducing the model size with little loss in accuracy and thus reduce the needed memory size and the amount of computation. A widely investigated approach is to create sparsity in weights and/or activations using pruning. However, pruning-based algorithms usually need specialized hardware architecture to exploit the resulting sparse tensors for energy reduction. On the hardware side, previous work has demonstrated several CNN accelerators targeting edge computing. However, many of them support limited flexibility for the NN shapes, are designed only for a specific task, or sacrifice the accuracy [17, 63, 47].

To address the challenges in flexibility, energy efficiency, and accuracy in CNN accelerator design, this work takes an algorithm-and-hardware co-design approach. The key contributions of this chapter are highlighted as follows: (1) a weight tuning algorithm that reduces the energy consumption associated with weight delivery and computation by lowering the toggle count of weight sequence; (2) the co-design of a CNN accelerator that supports the proposed algorithm and is flexible for a wide range of NN model structures; and (3) the demonstration of speech KWS as an example on the FPGA [56] and a fully integrated ASIC with the proposed CNN accelerator and a feature extraction processor[1].

[1]This unit was first designed by M. Price [43] and then modified by S. Lauwereins from Prof. Marian Verhelst's group at MICAS – KU Leuven [15].

3.2 Weight Tuning Algorithm and Potential SRAM Co-design

The weight tuning algorithm reduces the energy consumption of the CNN accelerator with little loss in accuracy by tuning the bit representation of weights. Fundamentally different from quantization and model compression algorithms that aim to reduce NN size, the proposed algorithm focuses on the toggling of bit sequences in the circuits. As shown in Eq. 3.1 with the load capacitance C_L, the supply voltage V_{dd}, and the operating frequency f,

$$P_{dyn} = \alpha_{0\to 1} C_L V_{dd}^2 f, \tag{3.1}$$

the dynamic power of a CMOS gate is linearly proportional to its switching activity ($\alpha_{0\to 1}$), which is influenced by the toggle count of its input sequence. In a NN accelerator, weights are read from the memory, delivered through network-on-chip (NoC) and then multiplied with IAs following a sequence set by the designer. The toggle count of this weight sequence affects the switching activity of weight buses and the multipliers. Moreover, its impact can be extended to recently proposed data-dependent SRAMs, e.g. [10]. Given that the SRAM does conditional pre-charge based on previously read data, reducing the toggle count of weight read sequence reduces the pre-charge activity of bit-lines. As a result, minimizing the toggle count of weight sequence can reduce the power consumption of weight buses, multipliers and the weight buffer. To gain those benefits, we propose a weight tuning algorithm that contains three sequential steps: (1) tensor decomposition with retraining; (2) quantization and the sign-magnitude representation; and (3) weight scaling and bit perturbation with retraining.

3.2.1 Tensor Decomposition with Retraining

Tensor decomposition is used to compress the model size and reduce the number of calculations in the NN with little loss of accuracy after retraining [32]. As shown in Fig. 3-1, tensor decomposition breaks one convolutional layer into three succes-

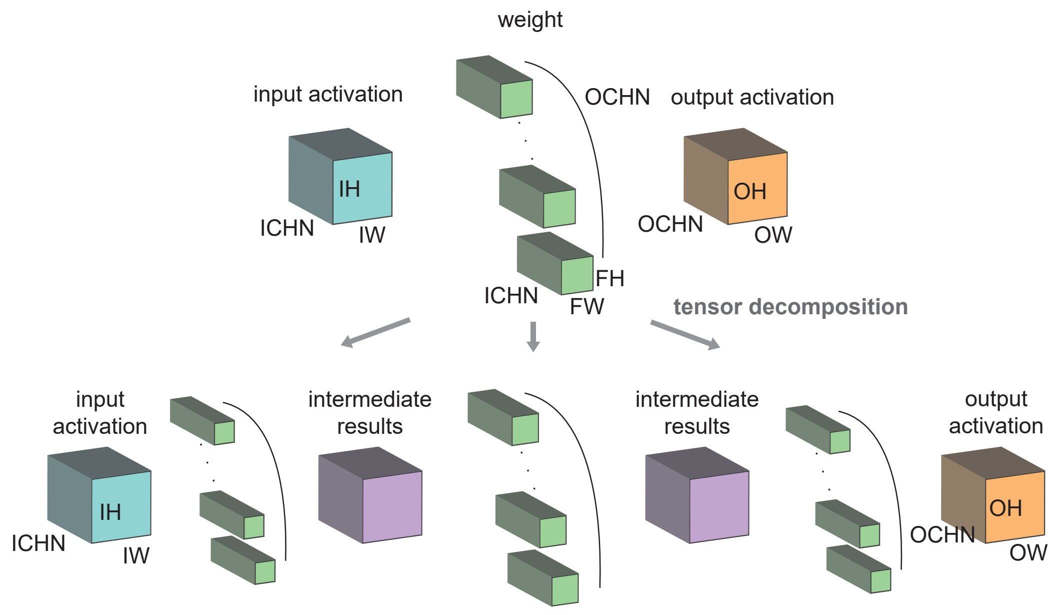


Figure 3-1: Illustration of tensor decomposition for CNNs [32]. The original weight tensor is first decomposed into three smaller tensors. The resulting NN is then retrained as proposed in [32].

sive layers without any activation function in between. The total parameters and computation of the resulting layers are less than those of the original layer. Thus, it is favorable for reducing the energy per inference. Since tensor decomposition introduced some error, directly using the decomposed tensors results in some loss in accuracy [32]. Therefore, retraining is needed after tensor decomposition.

3.2.2 Quantization and the Sign-Magnitude Representation

After decomposing and retraining the CNN, we use a linear quantizer to convert the model from floating point to fixed point numbers for deployment on the NN accelerator. It has been demonstrated by many works, e.g. [20, 66], that 8-bit precision in weights is enough for maintaining good accuracy. Instead of the 2's complement format, we use the sign-magnitude representation to reduce the toggle count of the weight sequence [4, 59]. Based on our experiments on various NNs, the sign-magnitude format reduces around 30% – 40% of the toggle count of the weight sequence compared to the 2's complement format. The overhead of using the sign-magnitude representation is the implementation of adders. Considering the fixed computation pattern of the

NN accelerator, we implement a mixed-representation PE to minimize the hardware cost. Section 3.3 discusses that in detail.

3.2.3 Weight Scaling and Bit Perturbation with Retraining

This step manipulates the bits of weights to further reduce the toggle count on top of the sign-magnitude representation. We first flatten the 4-D weight tensor of every layer to a 1-D vector following the sequence that weights are read, delivered, and calculated in the NN accelerator. Then we sequentially apply weight scaling and bit perturbation to further reduce the toggle count and incorporate them with retraining to maintain the accuracy.

Weight Scaling

To lower the toggle count of the weight sequence, we scale the weights and biases uniformly in every layer, which is inspired by the coefficient scaling for finite impulse response (FIR) filters [29]. The scaling factor K_l of layer l is determined by Eq. 3.2,

$$K_l = argmin_{k_l} ToggleCount(k_l \mathbf{W_l}), k_l \in (a, b), a \geq 0, \tag{3.2}$$

where $\mathbf{W_l}$ is the weight tensor at layer l, $ToggleCount$ is a function to calculate the toggle count, and a, b are the user-defined bounds of K_l. K_l is determined by an exhaustive search in the given range with a predefined step size s. It can be applied to layers using ReLU as the activation function without impact on the classification accuracy, given that $ReLU(K_l \mathbf{x}_l) = K_l ReLU(\mathbf{x}_l)$. For scale variant activation functions, users can skip this step.

The scaling in the range of weights may affect the quantization of both weights and activations. Although the user can choose arbitrary a and b as the range when searching for the scaling factor, we constrain it within $(0.5, 2)$ to restrict the search space and also prevent moving the decimal point in the fixed-point representation. Changing the average magnitude of weights affects the average magnitude of activations at each layer and may move the decimal point of activations. For example, if the weight scaling factor of two successive layers is 0.6, the resulting scaling factor on the output activations of the second layer is 0.36. It is important to prevent activations from overflow or underflow given the fixed-point representation when applying the proposed algorithm, otherwise, the accuracy may be greatly impaired. The user needs to keep track of the scaling factor and adjust the integer and fractional bit widths of activations in each layer accordingly after weight scaling is applied.

Bit Perturbation

Inspired by the coefficient perturbation for FIR filters [29], we perturb the bits of the weight sequence to reduce the bit toggling between successive weights. Bit perturbation changes the weight values, so it introduces some tuning errors. We use relative error averaged over all data to represent that. The relative error is defined as

$$e = \frac{|v - v_0|}{|v_0|} \tag{3.3}$$

where v_0 and v are the original value and the perturbed value respectively. The algorithm is illustrated in Fig. 3-2. Given the weight vector $\mathbf{D}$ and the maximum relative error e_{max} the system can tolerate, we split the weight vector into n sub-vectors and replace k LSBs of weights in each sub-vector with their average value. We loop through different combinations of n and k to find the minimum toggle count and the corresponding weight sequence.

Retraining

Since weight scaling and bit perturbation modify the weight values, retraining is applied to restore the potential accuracy loss. The flowchart in Fig. 3-3 summarizes the steps of quantization and sign-magnitude representation, and weight scaling and bit perturbation with retraining. The proposed algorithm is applied to the pre-trained floating point NN as a wrapper function of weights in the forward pass. During back-propagation, the straight-through estimator [67] is adopted, which passes the gradients through the wrapper function as-is. The activations are also quantized

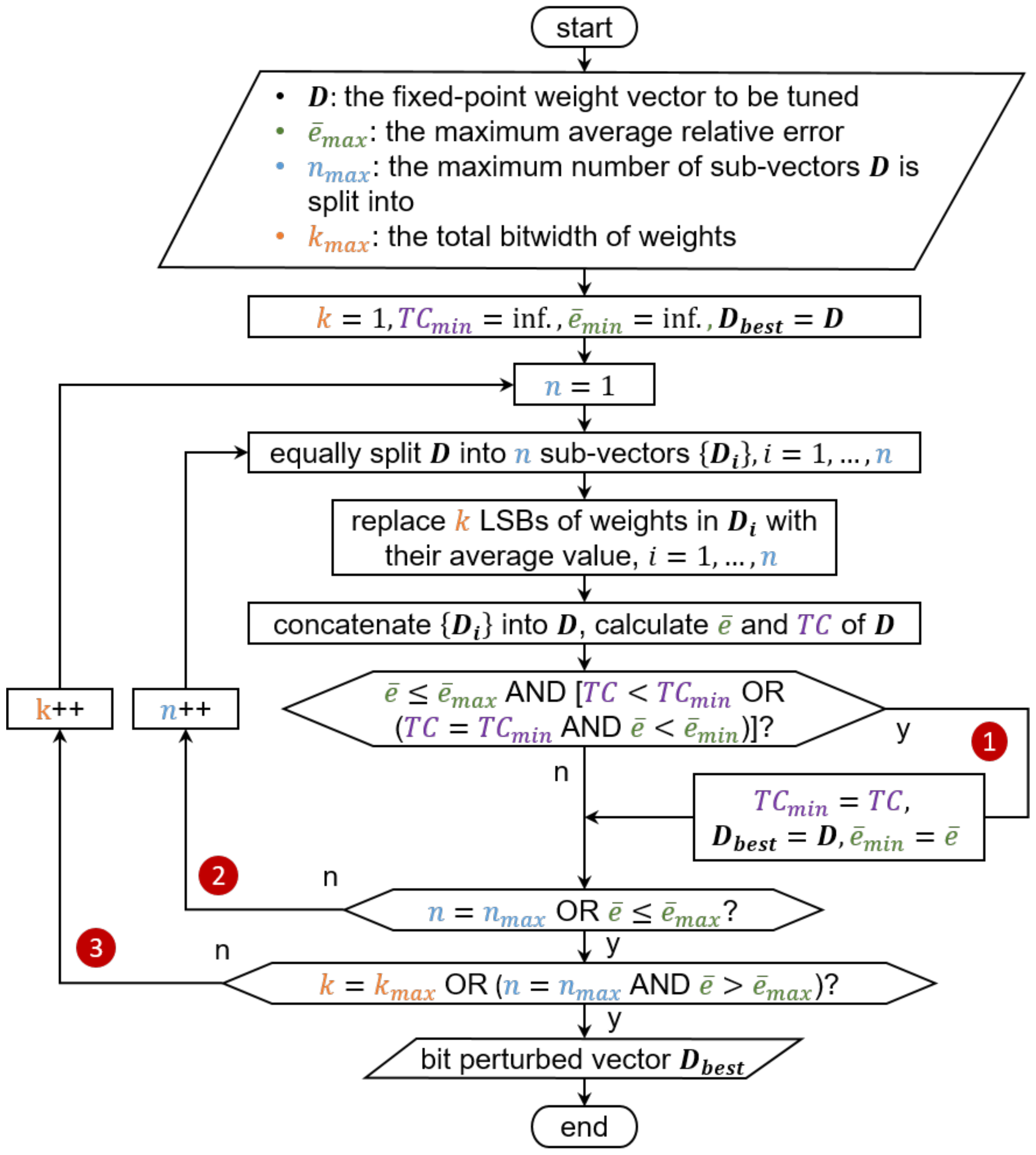

start
D: the fixed-point weight vector to be tuned
$\bar{e}_{max}$: the maximum average relative error
n_{max}: the maximum number of sub-vectors D is split into
k_{max}: the total bitwidth of weights
$k = 1, TC_{min} = \text{inf.}, \bar{e}_{min} = \text{inf.}, D_{best} = D$
$n = 1$
equally split D into n sub-vectors $\{D_i\}, i = 1, \dots, n$
replace k LSBs of weights in D_i with their average value, $i = 1, \dots, n$
concatenate $\{D_i\}$ into D, calculate $\bar{e}$ and TC of D
$\bar{e} \le \bar{e}_{max}$ AND [$TC < TC_{min}$ OR ($TC = TC_{min}$ AND $\bar{e} < \bar{e}_{min}$)]?
y
1
$TC_{min} = TC$, $D_{best} = D, \bar{e}_{min} = \bar{e}$
n
k++
n++
2
n
$n = n_{max}$ OR $\bar{e} \le \bar{e}_{max}$?
y
3
n
$k = k_{max}$ OR ($n = n_{max}$ AND $\bar{e} > \bar{e}_{max}$)?
y
bit perturbed vector D_{best}
end

Figure 3-2: The flowchart of the bit perturbation algorithm. The first branch is to keep the lowest possible $\bar{e}$ and the corresponding toggle count and the weight vector. The second branch is to find out how many sub-vectors $\mathbf{D}$ needs to be split into so that $\bar{e}$ can be lower than $\bar{e}_{max}$ given that k LSBs are tuned. The third branch is to loop through all possible k.

in the linear quantization step, given that activation-quantization-aware retraining provides higher accuracy.

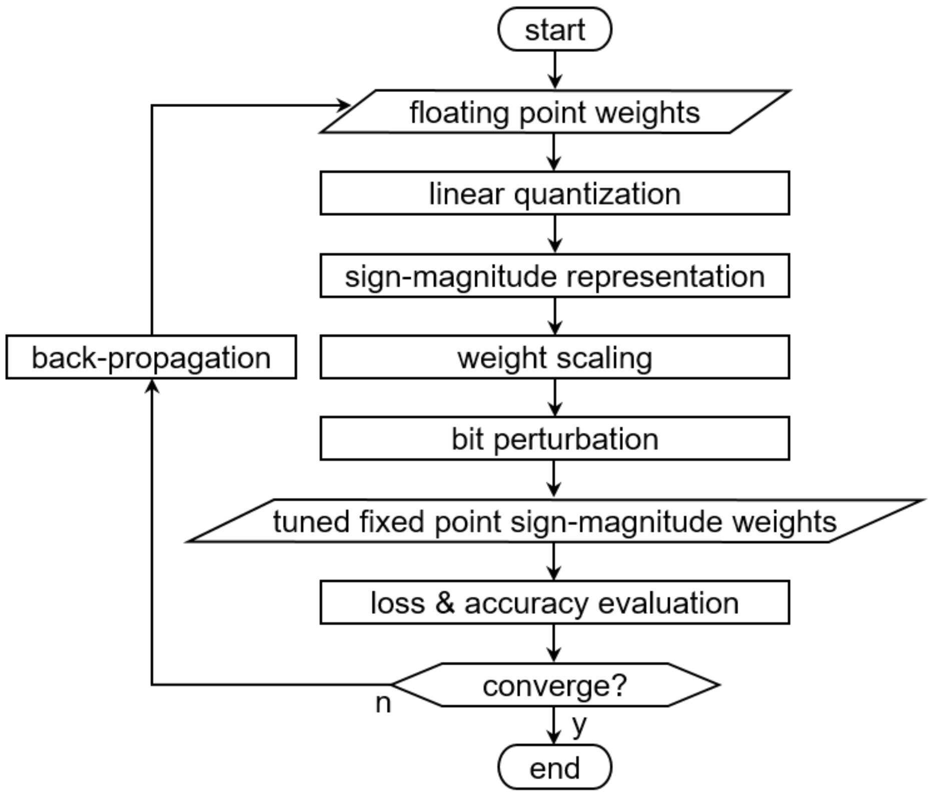


Figure 3-3: The flowchart of quantization and sign-magnitude representation and weight scaling and bit perturbation with retraining.

3.3 Datapath Co-design and Flexible CNN Accelerator Architecture

This section presents the proposed mixed-representation datapath to support the sign-magnitude representation of weights resulting from the weight tuning algorithm, and shows the standalone CNN accelerator architecture flexible for a wide-range of CNN structures.

3.3.1 Overall Architecture

The overall architecture of our proposed standalone system for CNN processing is shown in Fig. 3-4. It has an 80 kB weight buffer and a 1 kB configuration buffer for storage and configuration of the entire NN with up to 12 layers during the setup phase. All the data buffering is done on-chip using a 2 kB circular input buffer and a 48 kB

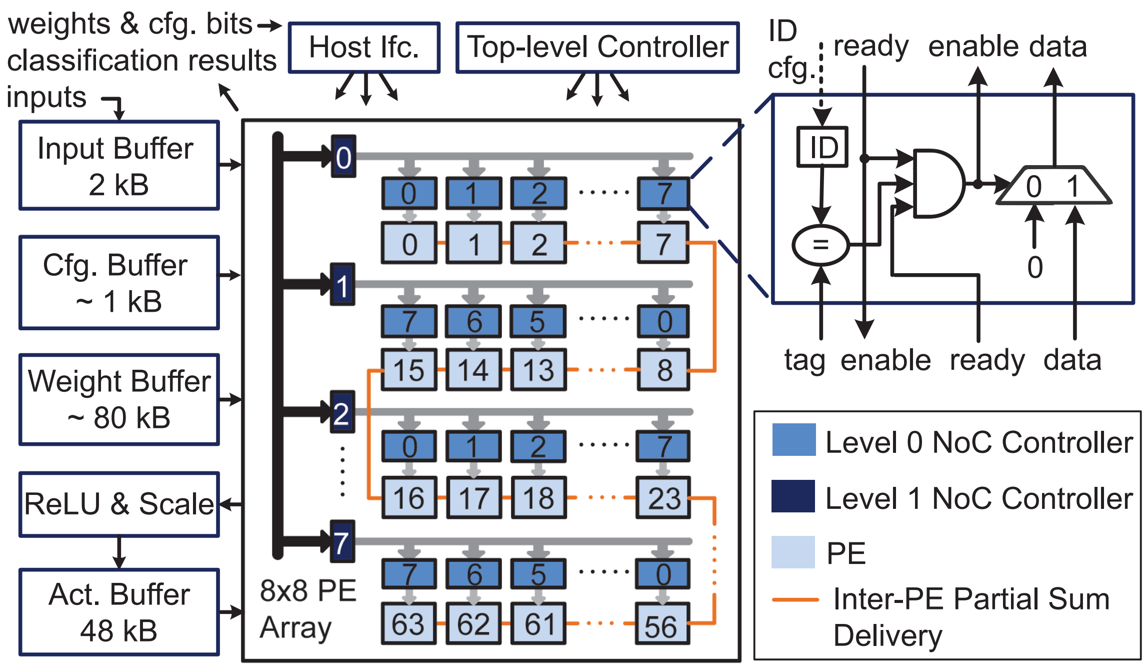


Figure 3-4: System architecture of the NN accelerator and the micro-architecture of the NoC controller.

activation buffer without the need for off-chip DRAM. NoC delivers data following programmable dataflow settings. Convolution/matrix multiplication is handled by the 8x8 PE array, and the activation scaling and the ReLU function are done by a separate unit before storing the activations in the buffer.

3.3.2 Flexbile Dataflow and NoC

The PE array level dataflow for convolutional layers is shown in line 4 – 10 of Fig. 3-5. The 8×8 PE array can be logically treated as having ICHN1 columns and OCHN1s rows handling the channel dimensions as shown in line 9 – 10.

After tensor decomposition, the channel sizes of different layers have large variances as shown in Appendix B. To achieve high utilization of the PE array when running the decomposed NNs, our architecture supports fully flexible logical rows and columns as long as ICHN1 $\times$ OCHN1s ≤ 64. That is different from Eyeriss [6], which limits the logical rows and columns to be less than the physical rows and columns. Such flexibility is achieved by a reconfigurable NoC [6] for data deliver-

```
1)  # IA[ICHN][IF][IW], ICHN = ICHN1*ICHN0 <= 256
2)  # W[OCHN][ICHN][FH][FW], OCHN = OCHN1t*OCHN1s*OCHN0 <= 256
3)  # OA[OCHN][OH][OW]
# PE array level – temporal dataflow
4)  For each FH:
5)   For each FW:
6)     For each OCHN1t:  # FH*FW*OCHN1t <= 2^16
7)       For each OH:
8)         For each OW:
 # PE array level – spatial dataflow
9)           Parallel_for ICHN1:  # ICHN1 * OCHN1s <= 64
10)            Parallel_for OCHN1s:
# PE level – temporal dataflow
11)              For each OCHN0:  # OCHN0 = {0, 1, 2, 3}
12)                For each ICHN0:  # ICHN0 = {3, 4}
13)                  # MAC
```

Figure 3-5: The dataflow illustrated with loop nests. The tensor dimension parameters are defined in Fig. 3-1. Bias is ignored for simplicity. The batch size is 1 for real-time application. For the first layer, where ICHN = 1, we replace ICHN with FW and remove the original FW loop to increase the PE array utilization. The read, delivery, and computation sequence of weight is as shown. PEs are filled up with weights in sequence. The limits on natively supported NN shapes are annotated.

ies between buffers and PEs, and fully connected and flexible inter-PE partial sum delivery. Following this logical mapping, weights are unicast to PEs, and thus a tree-structured NoC with a depth of two, as shown in Fig. 3-4, is used for weight delivery between the weight buffer and PEs. The ID of every controller at each level is unique and fixed. IAs are multicast across multiple logical rows. To support that, only level 0 NoC controllers are used. Their IDs, determined by OCHN1s and ICHN1, are configured at runtime before the execution of every layer. The deliveries of biases, partial sums, and OAs to/from the buffers are similar except that not every PE needs data. Thus, the unused controllers and FIFOs can be gated. For the inter-PE partial sum delivery, all PEs are connected in a sequence as shown in Fig. 3-4. Partial sums can be spatially accumulated across an arbitrary number of PEs. Thus, logical rows and columns can be fully flexible and accommodate a large variety of channel sizes.

As shown in line 4 – 8 of Fig. 3-5, our proposed design follows WS dataflow in the PE array level, different from Eyeriss which implements row stationary dataflow. The reason is that 1×1 or $1 \times x$ convolutional layers resulting from tensor decomposition take up a large part of the CNNs. WS dataflow can have more data reuse than row stationary dataflow in those layers. For fully-connected layers, we treat it as a special case of convolutional layers where OW = 1, OH = 1, IW = FW, IH = FH.

The PE level dataflow is shown in line 11 – 12 of Fig. 3-5 and the PE structure is illustrated in Fig. 3-6. PE has local storage to hold OCHN0 $\times$ ICHN0 weights and ICHN0 IAs to achieve temporal reuse of OW $\times$ OH and OCHN0 times respectively as shown in line 7 – 8 and line 11 of Fig. 3-5. OCHN0 and ICHN0 are configurable in runtime to balance the workload between PEs given layer shapes. Each PE has its local controller and local states so that they can execute independently once data are available.

3.3.3 Mixed-Representation Datapath

As discussed in Section 3.2.2, weights are represented in the sign-magnitude format to reduce toggle count. Given the fixed calculation pattern in CNN, we implement a mixed-representation datapath as shown in Fig. 3-6. Weights and IAs are multiplied in the sign-magnitude format using an unsigned multiplier and an XOR gate that generates the sign bit. An adder-subtractor is then used to convert the sign-magnitude product to the 2's complement representation with the sign bit of the product as the carry bit, and at the same time do accumulation. The 2's complement outputs are delivered to other PEs for spatial sum or buffered in memory for temporal accumulation to generate the output activations. Only after all the computation of this layer is finished, do we need to convert them back to the sign-magnitude format for the next layer. Thus, the energy overhead of the conversion is mitigated.

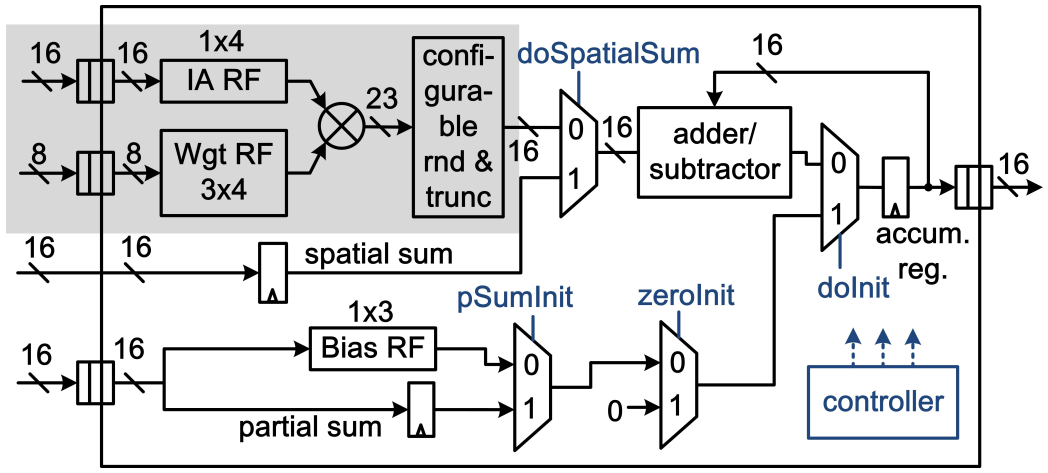


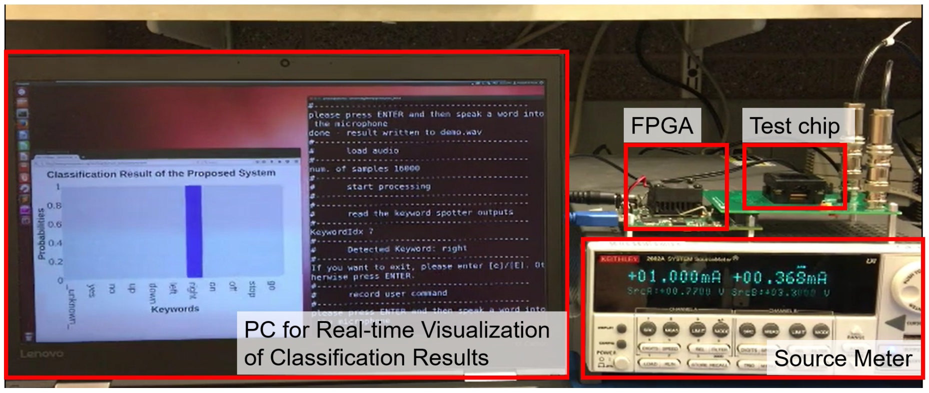
Figure 3-6: The PE structure. The shaded part is the sign-magnitude domain. The rest is in the 2's complement domain. Wgt: weight; rnd: round; trunc: truncate; RF: RegFile; accum. reg.: accumulation register.

3.4 Evaluation and Test Setup

3.4.1 Algorithm Evaluation Setup

We evaluate the accuracy and toggle count reduction of the weight tuning algorithm on several CNNs designed for KWS on the Google speech command dataset [58], including CNNs under 80 kB (referred to as CNN80) and 200 kB memory constraints in [66] and the fstride-4 model in [45]. Tensor decomposition with retraining is applied to most layers except the ones that largely impact the accuracy, e.g. the last layer. The resulting models in the 2's complement format serve as the baseline. Quantization and sign-magnitude representation, and weight scaling and bit perturbation with retraining are applied to the decomposed layers with a step size s of 0.05, the scaling factor bounds $a = 0.8, b = 1.8$, $e_{max} = 0.15$ and n_{max} equal to half of the vector length.

To evaluate how much energy savings our weight tuning algorithm provides, we implemented a baseline CNN accelerator that uses 2's complement MACs following the same procedures for synthesis and place-and-route (PnR) and using the same technology as the proposed accelerator. After PnR, both accelerators are simulated to run the entire tensor decomposed CNN80 on the actual inputs from the Google



```
#----------------------------------------
please press ENTER and then speak a word into
 the microphone
done   result written to demo.wav
#----------------------------------------
#       load audio
#----------------------------------------
num. of samples 16000
#----------------------------------------
#       start processing
#----------------------------------------
#----------------------------------------
#       read the keyword spotter outputs
#----------------------------------------
KeywordIdx 7
#----------------------------------------
#       Detected Keyword: right
#----------------------------------------
If you want to exit, please enter [c]/[E]. Ot
herwise press ENTER.
#----------------------------------------
#       record user command
#----------------------------------------
please press ENTER and then speak a word into
 the microphone
```

Figure 3-7: The chip measurement setup.

speech command dataset [58] to obtain the switching activity. Switching activity, parasitics, and timing information obtained after PnR are annotated during power analysis.

3.4.2 Chip Measurement Setup

Fig. 3-7 shows the chip measurement setup. A Keithley source meter provides power supplies. An Opal Kelly FPGA board generates clocks and transmits data between the chip and the PC. After the chip is powered up, clocked, and reset, it enters the setup phase. The PC sends the configuration bits, including the layer shapes of the entire NN up to 12 layers and rounding and shifting settings. Then the pre-trained weights of the NN for certain tasks can be sent to the chip. In our measurements, we use CNN80 [66] on the Google speech command dataset [58] as an example. After the setup phase, the chip starts taking in streaming inputs, computing the entire NN fully on chip, and streaming out the output classification results. The PC receives the outputs and generates real-time visualization.

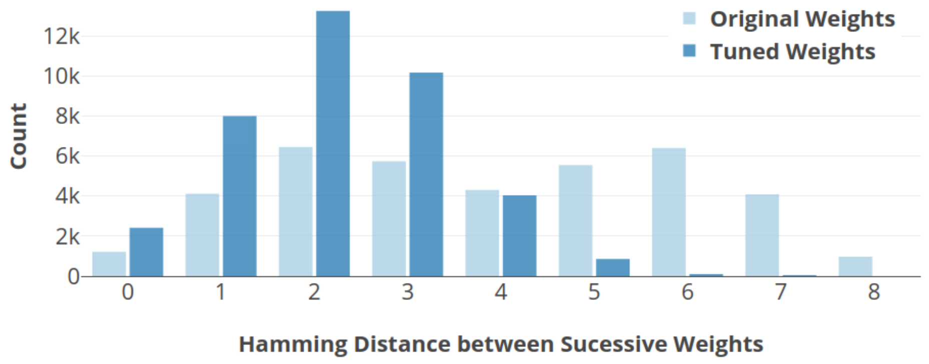


Figure 3-8: The weight tuning algorithm reshapes the histogram of Hamming distance between successive weights. CNN80 is shown as an example.

3.5 Evaluation and Measurements

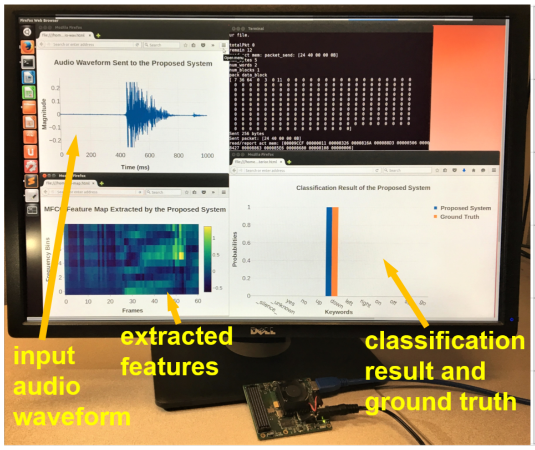
3.5.1 Weight Tuning Algorithm Evaluation Results

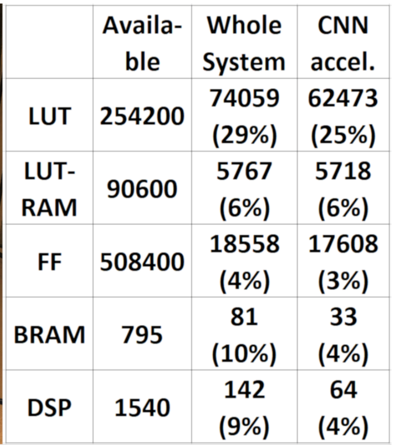
Based on our evaluation shown in Section 3.4.1, the weight tuning algorithm reduces the toggle count of weight sequences by 1.79x – 2.56x with less than 0.75% accuracy loss on the testing set for those cases. As shown in Fig. 3-8, the Hamming distance between successive weights is greatly reduced.

Table 3.1 summarizes the accuracy, the toggle count, and the total energy consumption of different components during the execution. $E_{multipliers}$ and E_{adders} are the total energy consumption of all the multipliers and adders in the PEs respectively. E_{MAC} is the sum of them. $E_{weightBus}$ is the energy of weight buses between the weight memory and PEs. It is obtained by summing up the internal and switching energy of buffers inserted in between. As shown, the weight tuning algorithm with the mixed-representation MAC reduces the computation energy by 1.20x compared to the 2's complement baseline. The energy of weight buses is reduced by 1.70x. Although the energy consumption of memory (the data dependant custom-SRAM is not used) and activation delivery is not affected by the algorithm, a 1.16x reduction in the total switching energy of the entire system $E_{totalSwitch}$ is observed.

Table 3.1: The Effect of the Weight Tuning Algorithm on Accuracy and Energy Consumption Based on Post-P&R Simulation

	Baseline	Proposed	Loss/Reduction
Accuracy	89.3%	88.8%	0.5%
Toggle Count	154k	86k	1.79x
$E_{multipliers}$	1.58uJ	1.11uJ	1.42x
E_{adders}	0.55uJ	0.66uJ	0.83x
E_{MAC}	2.13uJ	1.77uJ	1.20x
$E_{weightBus}$	96.91nJ	56.89nJ	1.70x
$E_{totalSwitch}$	8.94uJ	7.68uJ	1.16x


Audio Waveform Sent to the Proposed System
MFCC Feature Map Extracted by the Proposed System
Classification Result of the Proposed System
Proposed System
Ground Truth
input audio waveform
extracted features
classification result and ground truth



	Availa-ble	Whole System	CNN accel.
LUT	254200	74059 (29%)	62473 (25%)
LUT-RAM	90600	5767 (6%)	5718 (6%)
FF	508400	18558 (4%)	17608 (3%)
BRAM	795	81 (10%)	33 (4%)
DSP	1540	142 (9%)	64 (4%)

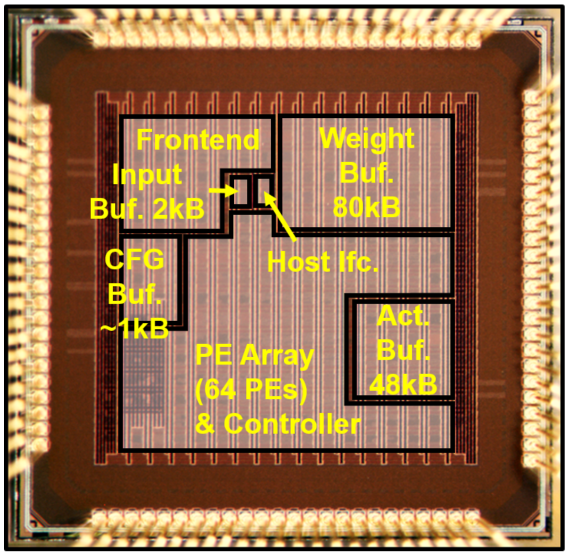
Figure 3-9: The FPGA demo of the CNN accelerator with a feature extraction processorv [15] on KWS, and a summary of FPGA post-PnR resource utilization.

3.5.2 FPGA Demonstration Results

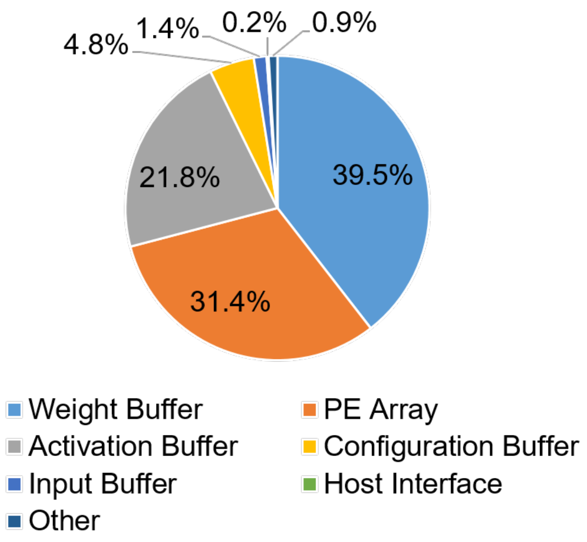
We demonstrate our design on FPGA with a speech feature extraction processor [15]. Implemented on Xilinx XC7K410T, the proposed CNN accelerator operates at 50MHz and consumes 68mW based on Vivado power estimation. A photo of the FPGA demo and the post PnR resource utilization are shown in Fig. 3-9.

3.5.3 Chip Implementation Results

The proposed system is fabricated using TSMC 40-nm LP process, including the speech feature extraction front end [15] and the proposed CNN accelerator. It has two operation modes – the general CNN acceleration mode with the front end clock-gated or the standalone KWS mode with both blocks activated. The die shot is shown



a)



b)

Figure 3-10: a) Die micrograph. b) Area breakdown of the CNN accelerator.

in Fig. 3-10-a. The total core area is 2.16 mm^2 and the area breakdown of the CNN accelerator is shown in Fig. 3-10-b.

The chip specifications are listed in Table 3.2. The CNN accelerator operates from 0.76 V to 1.1 V with a clock frequency from 20 MHz to 31.25 MHz. It supports 8-bit weights and 16-bit activations, which provides the arithmetic precision for most of the classification tasks to achieve good accuracy. The chip supports fully standalone processing for NNs with less than 80 kB of weights and less than 12 layers. Few limits are imposed on the layer shapes as long as they can fit in the on-chip buffers. Such flexibility makes the chip capable of running various NN structures specifically designed for different tasks. The power breakdown of the CNN accelerator is shown in Fig. 3-11. The weight buffer does not take a dominant part of the total power consumption, since weight stationary dataflow is used to minimize the number of reads of weights. The maximum energy efficiency of 14.87 pJ/MAC achieves at 0.76 V at the clock frequency of 20 MHz and the power consumption of 1.94 mW, for the CNN accelerator.

Compared with other digital ASICs for KWS, e.g. [47, 17, 63], our design is flexible, accurate, and achieves comparable energy efficiency. The proposed architecture

Table 3.2: Chip Specifications

Chip Specifications	
Technology	TSMC 40 nm LP 1P10M
Die Size	2.2 mm × 2.2 mm = 4.84 mm^2
Core Area	1.47 mm × 1.47 mm = 2.16 mm^2
Package	80-pin QFN
CNN Accelerator	
Logic Gates	454.7K (NAND2 equiv.)
On-Chip SRAM	133.12K bytes
Arithmetic Precision	8-bit weight, 16-bit activation
Supported Layer Types	conv. layer, linear layer, ReLU
Natively Supported Conv. Layer Shapes	activation height/width: flexible filter height/width: flexible channels: 1, 3 ~ 256 stride: flexible
Supply Voltage	core: 0.76 V – 1.1 V I/O: 3.3 V
Clock Frequency	20 MHz – 31.25 MHz
Avg. Power	1.94 mW – 4.46 mW
Latency	6.4 ms – 10 ms
Energy/MAC	14.87 pJ/MAC – 21.68 pJ/MAC

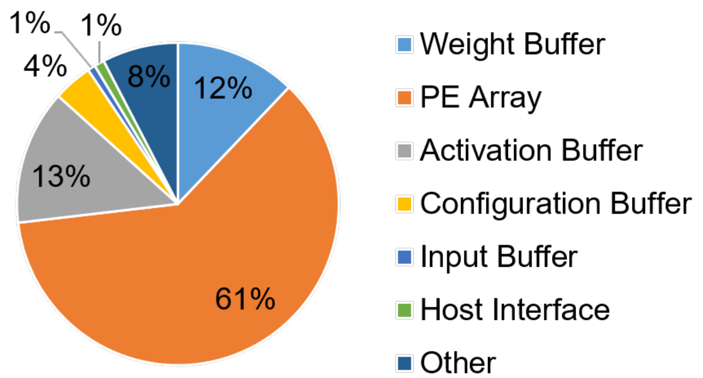


Figure 3-11: Power breakdown of the CNN accelerator based on post-P&R simulation.

supports flexible shapes and strides of inputs and weights for up to 12 CNN layers. However, [47] is restricted to a fixed structure, [17] supports up to 2 layers with up to 64 nodes/layer for LSTMs, and [63] is designed for a fixed input stride and 3x3 1-bit convolution. Our supported CNNs can achieve 91.6% accuracy with 12 output classes on the public available dataset [66], while [47, 17, 63] only report accuracy on the custom-designed dataset and [17] only shows binary classification.

3.6 Summary and Conclusions

We co-designed a weight tuning algorithm and the datapath of a CNN accelerator to improve energy efficiency with little loss in accuracy. Potentially, a data-dependent SRAM [10] can be used with the weight tuning algorithm to reduce weight read energy by around 2x. Furthermore, the accelerator features high flexibility and runtime reconfigurability to support various applications. The proposed algorithm reduces the energy consumption of weight delivery and computation by 1.70x and 1.20x respectively. The CNN accelerator consumes 14.87 pJ/MAC with a latency of 10 ms for real-time KWS applications. We made demonstrations with an integrated feature extraction processor for KWS on both FPGA and ASIC.

Chapter 4

Algorithm and Architecture Co-design Utilizing Data Features for Video Understanding

The last chapter focuses on the energy reduction of the fully integrated DL accelerator chip. In this chapter, we expand our scope to the entire system including DRAM. We co-design algorithms and architecture specifically for real-time video understanding systems and tapeout the accelerator chip, VideoTime^3 [57]. We utilize the temporal similarities in video data to reduce DRAM traffic and improve the energy efficiency and throughput of the system during real-time processing. It also captures temporal information between frames to achieve higher video understanding accuracy. Moreover, we expand the existing analysis and taxonomy of sparsity handling architecture for CNNs with our proposed techniques for the unique sparsity in PSums and OAs resulting from our algorithm co-design.

This work is done in collaboration with Prof. Song Han and his students as listed below. The training framework of conventional CNNs for object tracking is done in collaboration with Ji Lin. The design of the metadata generator is joint work with Yujun Lin and Zhekai Zhang.

The chapter is organized as follows. First, we provide an overview of deep-learning-based video understanding and analyze the challenges in existing accelerators. Then, we present our proposed algorithm and its architecture co-design. Following that, we explain our evaluation and measurement setup and then present our evaluation results. In the end, we discuss our analysis and taxonomy of sparsity handling architecture followed by a summary and conclusion of this chapter.

4.1 Overview of Deep-learning-based Video Understanding and Related Work

Video understanding is a classic area in computer vision. Common tasks in video understanding include video classification (what action the agent is performing), object detection and tracking (where the objects are located and where they are moving to), dense captioning (producing natural language annotations that describe what is happening at different times throughout the video), etc. [8]. They are widely used in many applications, including autonomous vehicles, augmented reality/virtual reality (AR/VR), artificial intelligence (AI) drones, health monitoring, etc. A lot of them require real-time video understanding on the edge. For example, autonomous vehicles need edge processing of enormous data due to latency, network connectivity, and energy limits. Also, a large amount of the workload requires real-time processing for safety considerations. With a video frame rate of 24 fps, a multi-frame latency of 4 frames can lead to a processing delay of 0.17 seconds, during which the vehicle operating at the speed of 65 miles/hour moves more than 16 feet. Thus, batch processing can lead to severe safety issues. With the rise of those edge applications, there is an increasing need for accurate, energy-efficient, and real-time video understanding on the edge.

Deep learning, especially deep CNN, has been extensively applied to video understanding and achieved considerable advances compared to traditional methods [35, 27, 26, 60]. Numerous CNN accelerators have been proposed over the past years for edge applications. We classify them into three categories in Table 4.1 and analyze existing challenges for video understanding.

Table 4.1: The Summary of Existing Challenges in Prior Work

Accelerator category	1: Conventional single-frame inference accelerators	2: A single-frame inference accelerator with conventional DiffFrame method	3: 3D CNN accelerators	This work
Capture temporal information?	X	X	✓	✓
Utilize temporal redundancy?	X	✓	✓	✓
Single-frame latency?	✓	X	X	✓

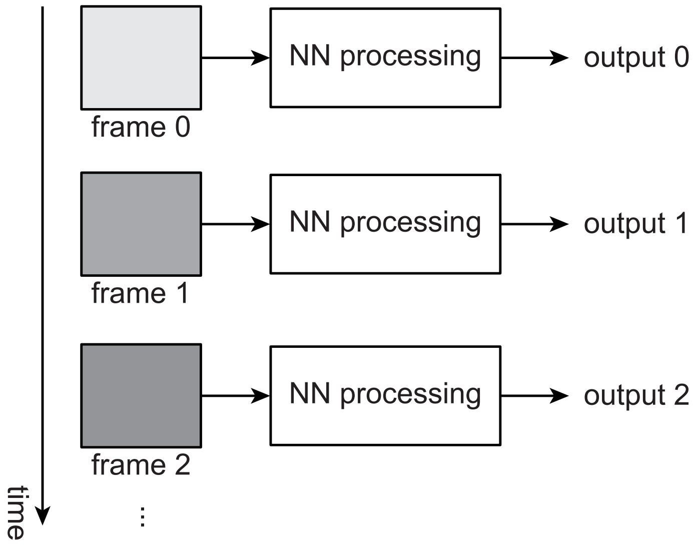


Figure 4-1: The NN accelerator that only supports single-frame/image processing fails to capture temporal information across video frames.

The first category is the accelerators, e.g. [34, 37], which only support image processing for a single frame as illustrated in Fig. 4-1. Thus, it fails to capture temporal information between successive frames and leverage their similarities.

To utilize those similarities, the second category proposed a DiffFrame method [64] based on the linearity of convolution, i.e., $\text{conv}(f_t^l) = \text{conv}(f_t^l - f_{t-1}^l) + \text{conv}(f_{t-1}^l)$, where $f_t^l - f_{t-1}^l$ and $\text{conv}(f_{t-1}^l)$ are DiffFrame and RefFrame at time t of layer l respectively. As shown in Fig. 4-2, instead of directly convolving every frame, it subtracts successive frames to generate a difference feature map $f_t^l - f_{t-1}^l$, which is referred to as DiffFrame, for convolution. As the difference between frames is typically sparse, it can reduce computation during convolution to generate $\text{conv}(f_t^l - f_{t-1}^l)$. However, since only the DiffFrame is convolved, the reference frame (RefFrame)

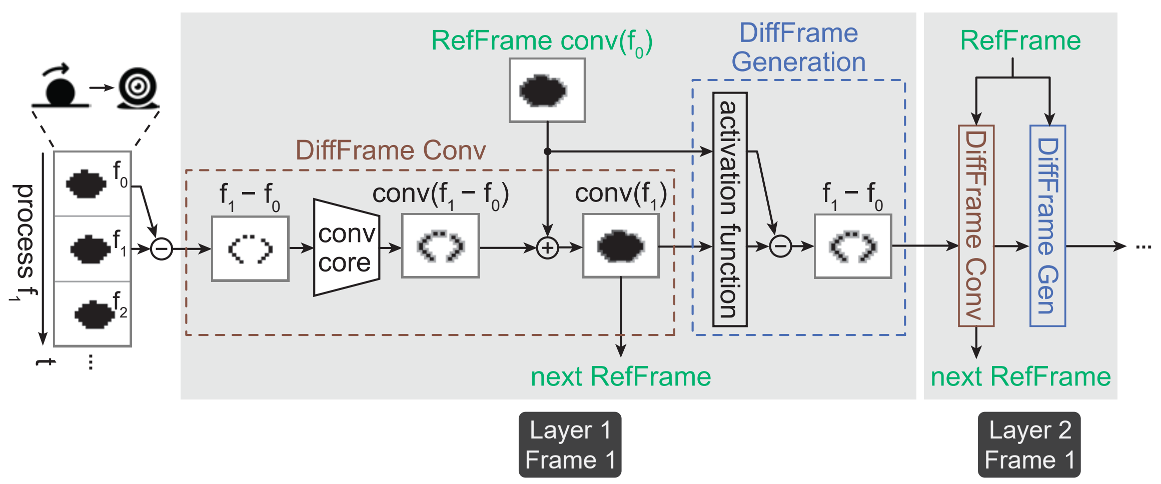


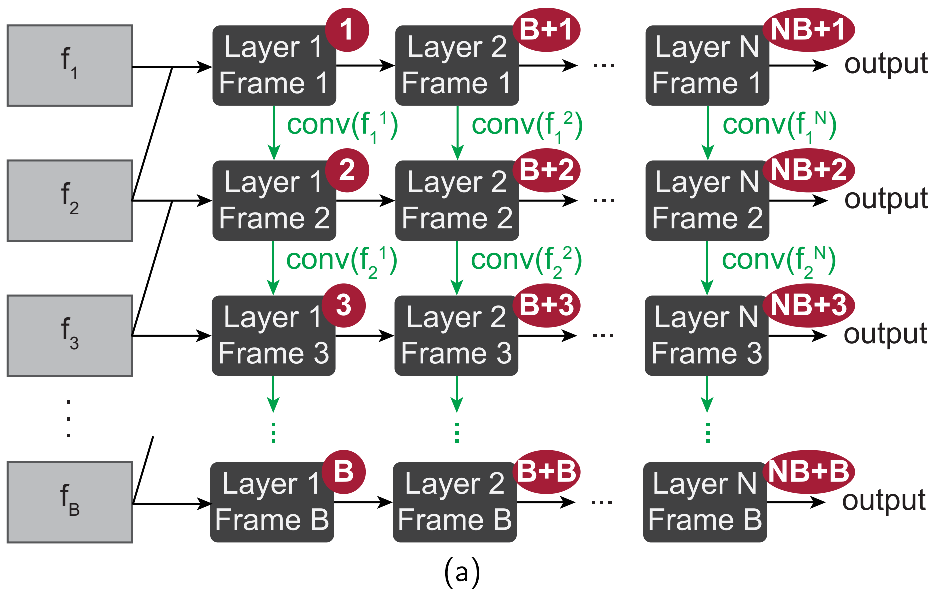
Figure 4-2: Illustration of the conventional DiffFrame method, which is based on the linearity of convolution, i.e., conv($f_t - f_{t-1}$) + conv(f_{t-1}) = conv(f_t). $f_t - f_{t-1}$ and conv(f_{t-1}) are DiffFrame and RefFrame at time t respectively.

conv(f^l_{t-1}) needs to be added to the convolution output to generate the convolution of the original frame conv(f^l_t). Also, this result of the current frame, conv(f^l_t), needs to be kept as the RefFrame for the computation of the next frame f^l_{t+1}.

Although convolving the sparse DiffFrame may lead to some computation and energy savings, it is challenging to handle the orchestration of RefFrames. To efficiently buffer RefFrames, prior work followed serial batch processing as shown in Fig. 4-3a. A batch of frames is buffered and processed serially for every layer. Once the RefFrame is generated by a frame at a layer, it will be immediately consumed by the next frame at the same layer. If the on-chip RefFrame buffer is big enough to hold the entire RefFrame of a layer, then the RefFrame can be reused on-chip across frames eliminating access to DRAM as shown in Fig. 4-3c. However, serial batch processing leads to multi-frame latency since a batch of frames needs to be buffered.

The third category is designed for 3D CNN which includes the temporal dimension in convolution to capture temporal information and reduce temporal redundancy across layers by downsampling in the temporal dimension [22]. However, it has multi-frame latency as multiple frames must be buffered for temporal convolution as shown in Fig. 4-4.

Despite prior work, it remains challenging to capture temporal information for



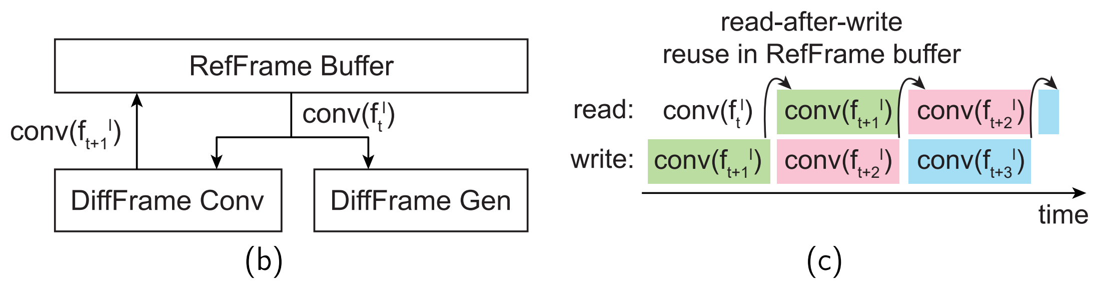


Figure 4-3: (a) An illustration of serial batch processing. The processing sequence is noted in red. N is the total number of layers and B is the batch size. (b) The high-level block diagram of the DiffFrame convolution accelerator and an illustration of RefFrame buffer access during the process of layer l at frame t. (c) The reuse of RefFrame over time in the on-chip buffer.

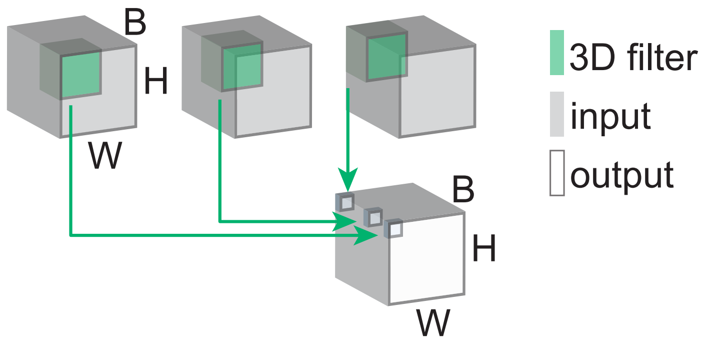


Figure 4-4: Illustration of 3D CNN convolving the temporal dimension, which is composed of a batch of frames. For simplicity, the channel size is set to one in the illustration. W: width. H: height. B: batch.

Table 4.2: Comparison of Conventional DiffFrame Convolution and Our Proposed Real-time DiffFrame Convolution

	Conventional DiffFrame Convolution		Proposed Real-Time DiffFrame Convolution
	Serial Batch Processing	Frame-by-frame Processing	
Sparsity in Convolution	Yes	Yes	Yes
Latency	High	Low	Low
RefFrame DRAM Traffic	Low	High	Low
Input DRAM Storage	Large	Small	Small

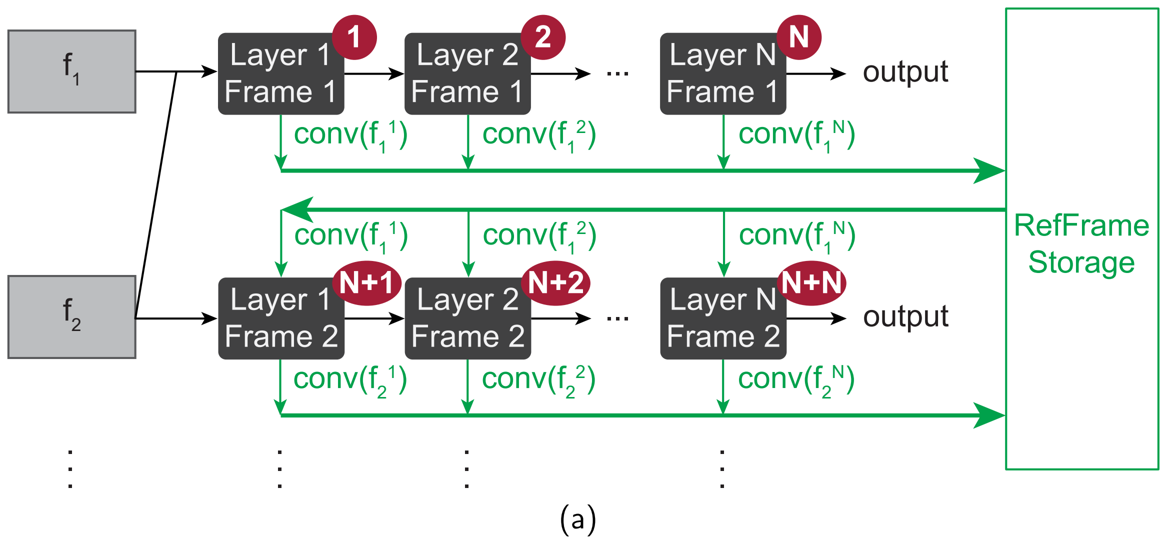
high accuracy and utilize temporal redundancy in videos for energy savings while achieving single-frame latency for real-time applications.

4.2 Algorithm and Accelerator Architecture

To tackle the remaining challenges in prior work, we propose a real-time DiffFrame convolution with temporal modeling algorithm and co-design novel sparsity-handling architecture and efficient data orchestration. In this section, we first present our real-time DiffFrame convolution algorithm—how it efficiently utilizes temporal redundancy at single-frame latency. Following that, the sparsity handling architecture we co-designed with the algorithm is explained. Then we discuss how temporal modeling is achieved with the real-time DiffFrame convolution. Succeeding that, we present how we handle the unique data orchestration requirements of both real-time DiffFrame convolution and temporal modeling. In the end, we illustrate the DRAM data layout, other components and overall architecture of the accelerator.

4.2.1 Real-Time DiffFrame Convolution

We propose real-time DiffFrame convolution that 1) utilizes temporal redundancy, which generates sparsity in convolution, as the conventional DiffFrame method, 2) delivers single-frame latency for real-time applications, and 3) achieves efficient DRAM utilization. A comparison between our proposed real-time DiffFrame convolution and



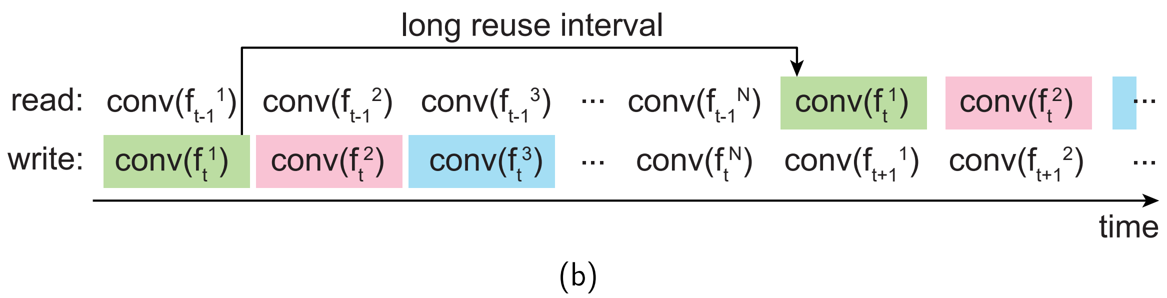


Figure 4-5: (a) An illustration of frame-by-frame processing. The processing sequence is noted in red. N is the number of layers. (b) A long interval between the reuse of RefFrame in frame-by-frame processing.

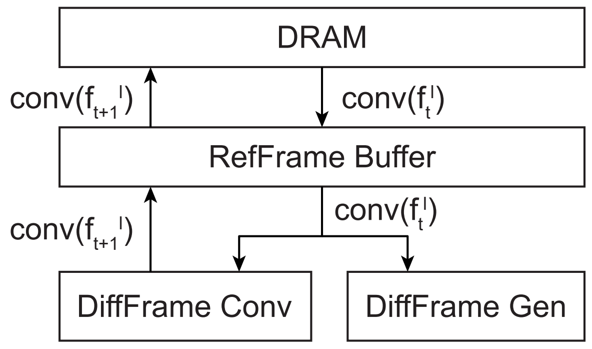


Figure 4-6: The block diagram of naive DiffFrame generation and buffering.

conventional DiffFrame convolution is summarized in Table 4.2. How we achieve those advantages against conventional techniques is presented in this subsection.

To achieve single-frame latency, we use frame-by-frame processing, as shown in Fig. 4-5a, instead of serial batch processing. That eliminates the need to buffer a

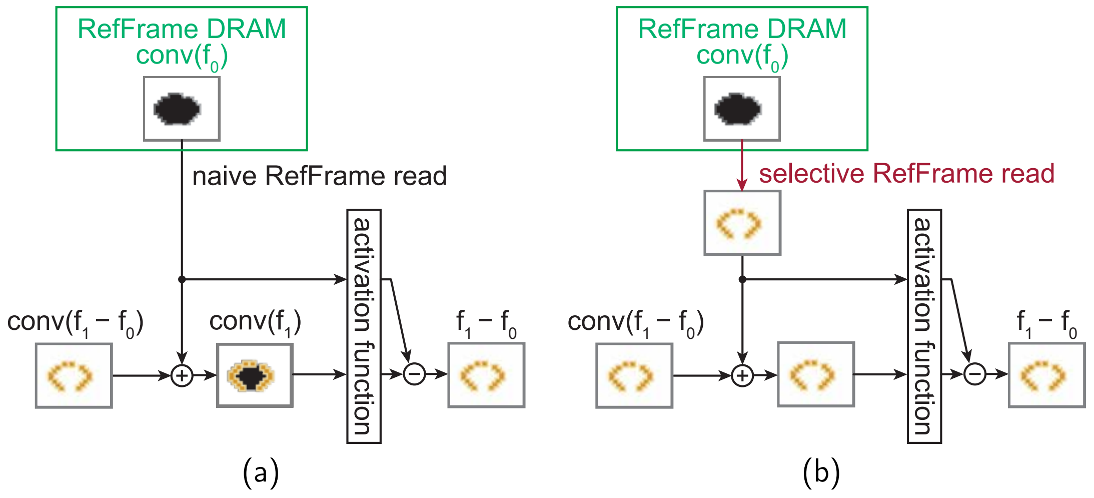


Figure 4-7: (a) An illustration of naive RefFrame read. (b) An illustration of proposed selective RefFrame read and update.

batch of input frames and thus reduces the DRAM storage for inputs. However, frame-by-frame processing leads to a long interval between the reuse of RefFrame as illustrated in Fig. 4-5b and thus all the RefFrames generated in between, which are all the activations in a NN, have to be kept. Since the size of activation in a NN can be really big, e.g., bigger than 10 MB for MobileNet-v2, off-chip DRAM is usually required to store those data. The system block diagram for naive RefFrame generation and buffering is shown in Fig. 4-6. Reading and updating the entire RefFrame for each layer and frame leads to high DRAM traffic and system energy.

To solve this problem, we propose a selective RefFrame read and update scheme, which reduces RefFrame DRAM and buffer traffic by 70% at 30% DiffFrame density (nonzeros) with zero loss in accuracy. The selective RefFrame read is illustrated in Fig. 4-7. To compute the DiffFrame for the next layer, only the pixels at the positions of nonzero pixels of conv($f_1 - f_0$) are needed in conv(f_1) and conv(f_0) (highlighted in yellow in Fig. 4-7a). Because conv(f_1) and conv(f_0) are equal elsewhere and yield zero difference after the activation function. Thus, only RefFrames at those positions need to be read and updated as shown in Fig. 4-7b. Moreover, the selective read and update not only reduce DRAM traffic as shown in Fig. 4-7, but also reduce buffer traffic accordingly.

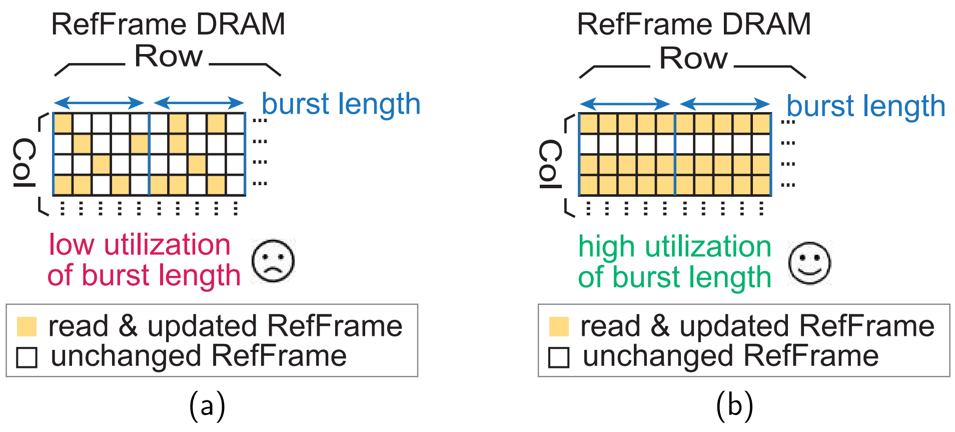


Figure 4-8: (a) Fine-grained irregular sparsity in DiffFrame leads to DRAM burst length under-utilization for selective RefFrame load and store. (b) Channel-wise coarse-grained sparsity in DiffFrame leads to high utilization of DRAM burst length for selective RefFrame load and store.

With our selective RefFrame read and update scheme, the read and update pattern of the RefFrame is determined by the sparsity pattern in the convolution output of DiffFrame $\text{conv}(f_t - f_{t-1})$. Thus, the sparsity pattern in $\text{conv}(f_t - f_{t-1})$ greatly affects the load and store efficiency of DRAM for RefFrame. If we allow fine-grained irregular sparsity, the DRAM burst length cannot be fully utilized for selective RefFrame load and store as shown in Fig. 4-8b. To address this issue, we enforce channel-wise sparsity in $\text{conv}(f_t - f_{t-1})$ and maintain a channel-first DRAM storage sequence as shown in Fig. 4-8b. It achieves 2.5x higher utilization of DRAM burst size compared to the fine-grained load/store pattern at 40% density.

To efficiently enforce channel-wise sparsity in $\text{conv}(f_t - f_{t-1})$, we propose DiffFrame SparseConv, which applies pruning and sub-manifold sparse convolution [19] on DiffFrames.

We prune the DiffFrame based on the L1 norm of its channel values to obtain a density similar to fine-grained sparsity in DiffFrame. To the best of our knowledge, it is the first work that explores the application of pruning on temporal similarities to generate sparsity in activations. Accuracy analysis is presented in Section 4.4.1.

Sub-manifold sparse convolution [19] is different from conventional convolution in that it enforces specific rules between input and output coordinates as shown in

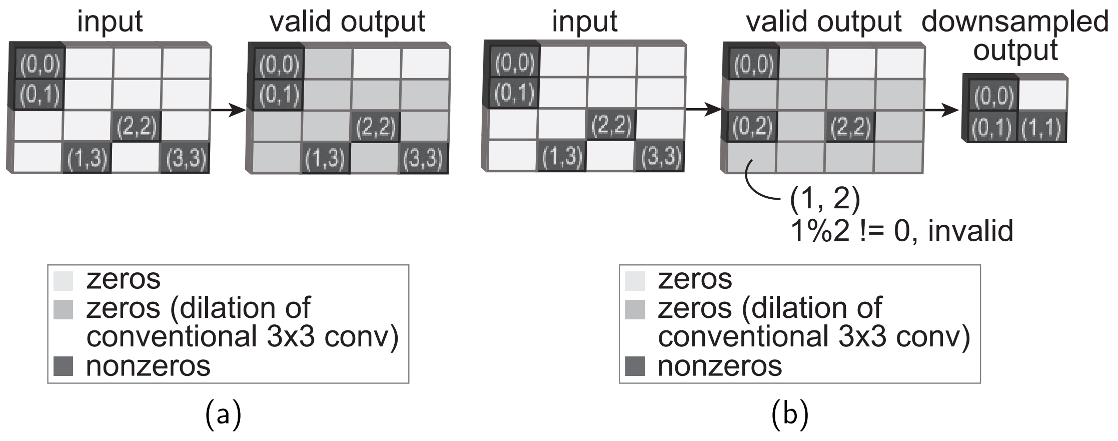


Figure 4-9: The rules of input and output coordinates in sub-manifold sparse convolution. (a) For stride = 1 layers, the input and output coordinates are exactly the same. (b) For stride = 2 layers, the valid outputs are those with $x\%2 = 0$ and $y\%2 = 0$, where (x, y) is the output coordinate of conventional convolution.

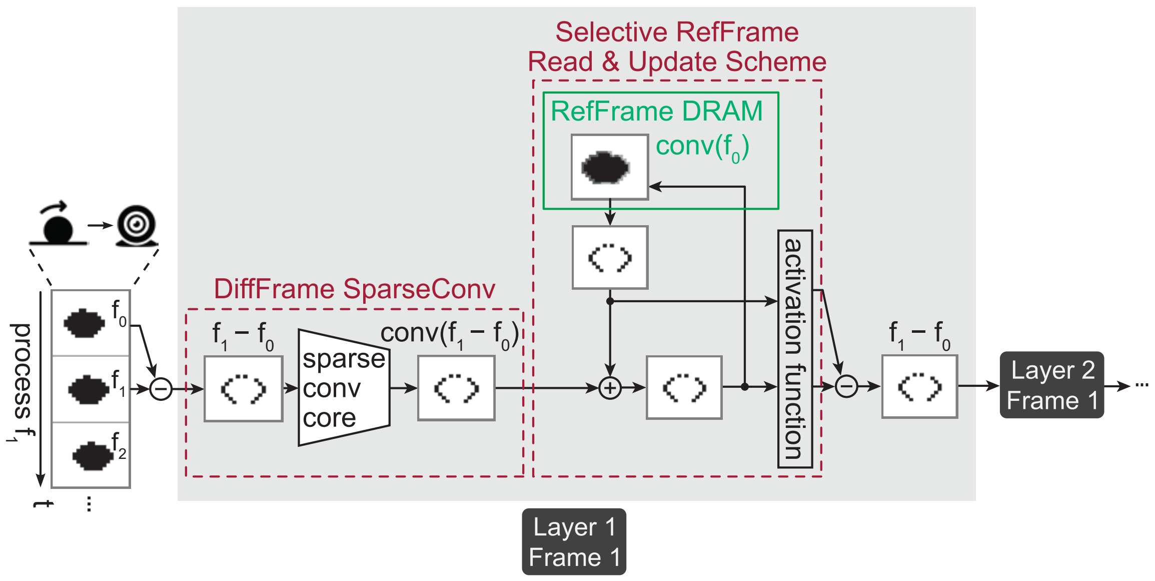


Figure 4-10: The proposed real-time DiffFrame method illustrated with stride = 1 layer.

Fig. 4-9. That removes the dilation in conventional convolution, which is marked in gray in Fig. 4-9, and prevents the activations from getting denser through convolution. Thus, it reduces the RefFrame traffic. Also, metadata involved in activation sparsity handling, e.g., the coordinates of nonzeros, can be reused across stride=1 layers, which significantly reduces metadata computation overhead. Detailed discussion about sparsity handling is presented in Section 4.2.2.

The overall algorithm diagram of the proposed real-time DiffFrame method is shown in Fig. 4-10. The output of DiffFrame SparseConv is the same as its input for stride = 1 layers without dilation and the selective read and update of RefFrame follow the same pattern reducing DRAM traffic. DiffFrame SparseConv and selective RefFrame read and update scheme work together to achieve computation and memory access reduction at single-frame latency.

4.2.2 Sparsity Handling

To efficiently handle the channel-wise sparsity in our real-time DiffFrame convolution, we propose a decoupled metadata generation and map-guided convolution scheme. It brings more flexibility to metadata generation, achieves metadata reuse, and delivers the first 2D CNN accelerator that stores both IA and PSum/OA directly in the compressed format during convolution. It enables the accelerator to skip all computation and memory traffic of zero IA and PSum/OA pixels with low overhead. This section focuses on explaining how our proposed scheme works and the design of our sorter-free metadata generator. A comparison of different sparsity handling schemes and a proposed representation for analyzing sparsity handling architecture are presented in Section 4.5.

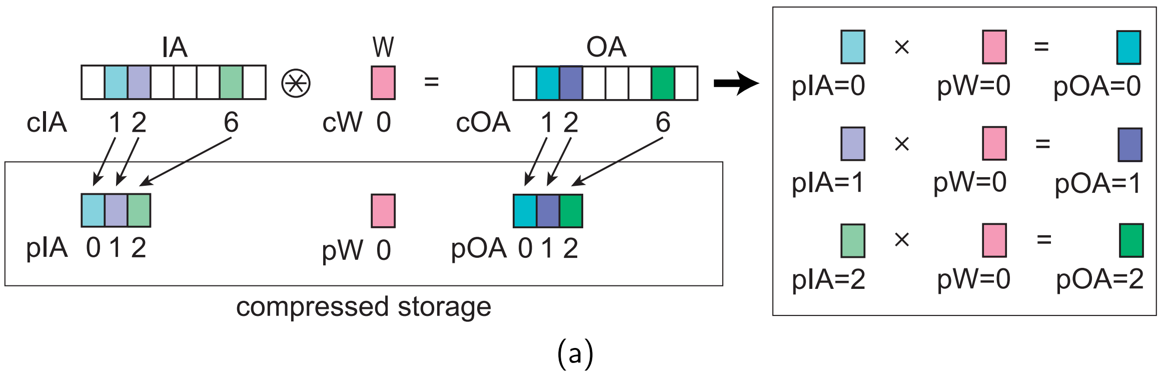
Overview of Sparse Point-wise Convolution and Sparse Depth-wise Convolution

As plenty of prior work has presented various techniques to handle sparsity in weights, our work focuses on sparsity handling for activations, which can be more challenging as discussed in Section 4.5. The proposed real-time DiffFrame convolution generates sparse DiffFrame as the input to the convolution core and outputs sparse PSum/OA following the rules specified in Fig. 4-9. Both IA and PSum/OA can be stored in a compressed format. Compared to conventional sparse convolution where PSum/OA is usually dense, our method provides around 3x reduction in PSum/OA storage at 30% density. Moreover, compressed PSum can lead to a large impact on overall data access and memory energy consumption of DNN accelerators. This is because: 1) PSum is usually set to be more than 20 bits to preserve the accuracy during accumulation, while weight and IA are set to be 8 bits or even less in most DNN inference accelerators; 2) PSum needs to be read and written for every temporal accumulation, while weight and IA only need to be read.

The storage and computation of sparse point-wise (PW) and depth-wise (DW) convolution are illustrated in Fig. 4-11. PW convolution uses a 1×1 filter and thus the sparse convolution on compressed IA and OA storage is straightforward. IAs and OAs share a one-to-one relationship based on their positions. For example, an IA pixel at the position of 1 generates an OA pixel at the position of 1. Therefore, we only need to go over each pixel in the compressed IA storage, multiply that with the weight, and store them at the same position in the OA storage. DW convolution is more challenging as the filter size can be bigger than 1×1. The filter pixels need to be multiplied with IA pixels at certain coordinates. However, it is not straightforward to know the position of an IA pixel given its coordinate in compressed IA storage. To tackle this challenge, we propose map-guided convolution and decoupled metadata generation for sparse DW convolution.

Map-Guided Convolution and Decoupled Metadata Generation

Fig. 4-12 illustrates our proposed map-guided convolution and decoupled metadata generation. To get the positions of data needed for each multiplication, we use a ConvMap, which is a list of the set (pIA, pW, pOA), to guide sparse convolution. For example, a ConvMap entry (1, 0, 1) shows that an IA pixel at the position of 1 and a filter pixel at the position of 0 need to be loaded and multiplied to generate a PSum of an OA pixel at the position of 1. The ConvMap is generated by a metadata generator decoupled from the convolution core, i.e., the loading sequence of coordinates and the operations on coordinates can be fully separate from data loading and processing. Given a fixed filter size (e.g., 3×3), the output coordinates and ConvMap are only determined by the coordinates of nonzero IAs and the stride of this layer. As long as the IA coordinates (the IA sparsity pattern) and stride stay the same, metadata



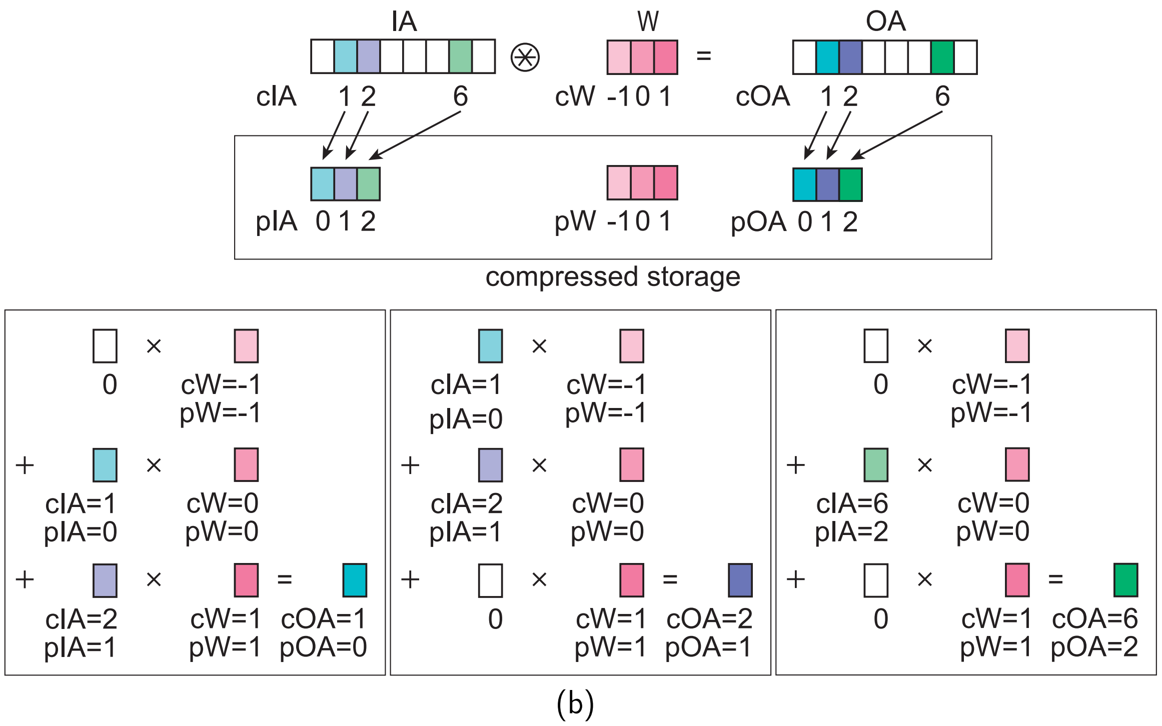
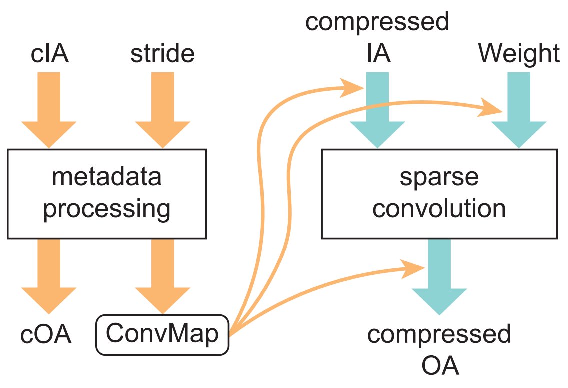


Figure 4-11: An illustration of sparse (a) PW convolution and (b) DW convolution. Channels are set to 1 for simplicity and clarity of the figure. Colored and white boxes present nonzeros and zeros respectively.

(including the nonzero OA coordinates and the ConvMap) keep unchanged. Thus, it can be reused across those layers. Metadata only needs to be generated 5 times for our target workload—MobileNet-v2-34 and MobileNet-v2-47 as shown in Appendix B.

Sorter-free Coordinate and ConvMap Generator

To efficiently generate metadata, we design a sorter-free coordinate and ConvMap generator. It reads sorted IA coordinates sequentially and outputs sorted OA coor-



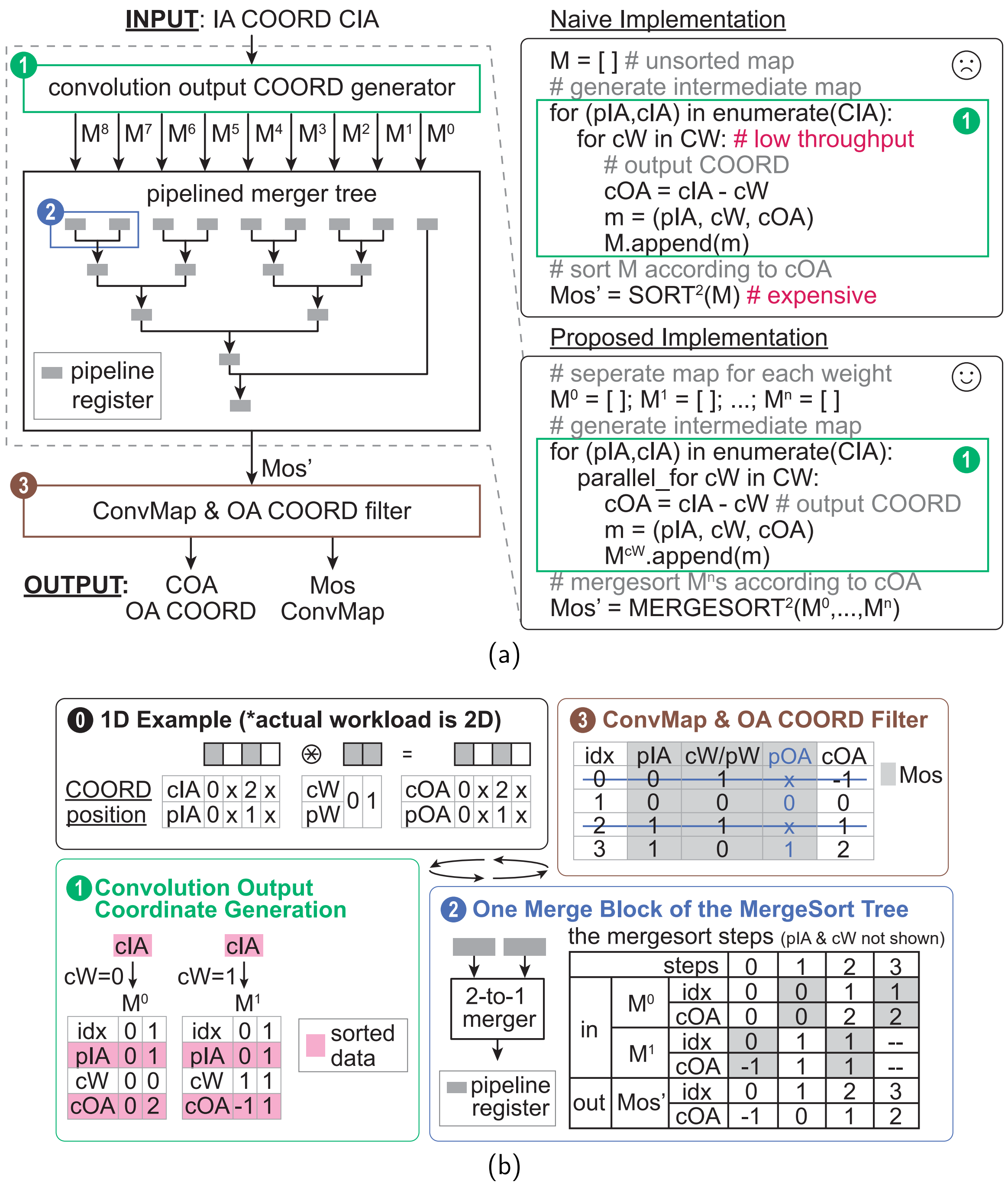
```
For each (pIA, pW, pOA) in ConvMap:
  OA[pOA] += IA[pIA] * W[pW]
```

Figure 4-12: An illustration of map-guided convolution and decoupled metadata generation.

dinates and a ConvMap sorted by OA positions guiding an OS dataflow for sparse convolution. The coordinates are kept in the COO format [2] in this work. Other formats, e.g., the compressed sparse row (CSR) format, may also be used to further reduce storage size, which is left for future work.

The generator consists of three parts. The first two parts are a convolution output coordinate generator and a sorting unit. The inputs of them are sorted cIA and the output is an intermediate map (M'_{OS}) sorted by OA coordinates, which is a list of the set (pIA, cW, cOA) (cW and cOA are coordinates of the weight and the OA respectively). The example pseudo codes for 1D convolution are shown in Fig. 4-13a. We enumerate the sorted cIA to get the pairs of IA positions and coordinates. The naive implementation loops through all the weight coordinates sequentially to calculate the coordinates of all possible convolution outputs given each IA coordinate. All the (pIA, cW, cOA) sets generated in the loops (M) are passed to a sorter with the key of cOA to generate the intermediate ConvMap M'_{OS}. The drawbacks of naive implementation are that the sorter is very expensive and the sequential loops lead to low throughput.

To solve those problems, the proposed work handles the calculation of each weight coordinate in parallel and the output (pIA, cW, cOA) sets are kept separately (M^n). An example can be found in Fig. 4-13b-1. Since input cIA is sorted and cOA is shifted from cIA by a fixed number, each M^n is sorted by cOA. Thus, simple mergesort, instead of expensive sort, can be used to combine the sorted M^ns into the sorted


INPUT: IA COORD CIA
1 convolution output COORD generator
M8 M7 M6 M5 M4 M3 M2 M1 M0
pipelined merger tree
2
pipeline register
Mos'
3 ConvMap & OA COORD filter
OUTPUT: COA OA COORD
Mos ConvMap
Naive Implementation
Proposed Implementation
(a)
0 1D Example (*actual workload is 2D)
3 ConvMap & OA COORD Filter
Mos
1 Convolution Output Coordinate Generation
cIA
cW=0
M^0
cW=1
M^1
sorted data
2 One Merge Block of the MergeSort Tree
the mergesort steps (pIA & cW not shown)
2-to-1 merger
pipeline register
(b)

Naive Implementation

```
M = [ ] # unsorted map
# generate intermediate map
for (pIA,cIA) in enumerate(CIA):
    for cW in CW: # low throughput
        # output COORD
        cOA = cIA - cW
        m = (pIA, cW, cOA)
        M.append(m)
# sort M according to cOA
Mos' = SORT²(M) # expensive
```

Proposed Implementation

```
# seperate map for each weight
M⁰ = [ ]; M¹ = [ ]; ...; Mⁿ = [ ]
# generate intermediate map
for (pIA,cIA) in enumerate(CIA):
    parallel_for cW in CW:
        cOA = cIA - cW # output COORD
        m = (pIA, cW, cOA)
        M^cW.append(m)
# mergesort Mⁿs according to cOA
Mos' = MERGESORT²(M⁰,...,Mⁿ)
```

COORD / position				
cIA	0	x	2	x
pIA	0	x	1	x

cW / pW	0	1

cOA	0	x	2	x
pOA	0	x	1	x

idx	pIA	cW/pW	pOA	cOA
~~0~~	~~0~~	~~1~~	~~x~~	~~-1~~
1	0	0	0	0
~~2~~	~~1~~	~~1~~	~~x~~	~~1~~
3	1	0	1	2

M^0

idx	0	1
pIA	0	1
cW	0	0
cOA	0	2

M^1

idx	0	1
pIA	0	1
cW	1	1
cOA	-1	1

		steps	0	1	2	3
in	M^0	idx	0	0	1	1
		cOA	0	0	2	2
	M^1	idx	0	1	1	--
		cOA	-1	1	1	--
out	Mos'	idx	0	1	2	3
		cOA	-1	0	1	2

Figure 4-13: (a) An overview of the proposed sorter-free ConvMap and coordinate generator. cIA, cOA and cW: coordinates of nonzero IA, OA and weight respectively; pIA, pOA and pW: positions of nonzero IA, OA and weight respectively; M'_{OS}; the intermediate unfiltered map containing all possible outputs of conventional convolution; M_{OS}: the final ConvMap satisfying the rules in Fig. 4-9. (b) An example of simple 1D ConvMap and coordinate generation.

M'_{OS}. We implement the mergesort algorithm with a pipelined merger tree shown in Fig. 4-13a for the target workload of 3×3 convolution. Fig. 4-13b-2 illustrates one of the merger blocks. In step 1, the 2-to-1 merger takes in $M^0[0]$ and $M^1[0]$, and compares their cOAs. Since cOA of $M^1[0]$ is no greater than that of $M^0[0]$, the merger consumes and outputs $M^1[0]$. Next, the merger compares $M^0[0]$ and $M^1[1]$. And so on. Using parallel processing and simple mergers, the proposed method generates the sorted intermediate map M'_{OS} with higher throughput and lower complexity compared to naive implementation.

The last part is the ConvMap and OA coordinate filter. It 1) filters out invalid OAs and related M'_{OS} entries based on the rules shown in Fig. 4-9, 2) generates positions of OAs (pOA), and 3) groups pIA and pW in M'_{OS} and pOA to form a ConvMap M_{OS}. An example is shown in Fig. 4-13b-3. Since stride = 1, the first valid cOA (0) is assigned a pOA of 0 and so on. Since M_{OS} is sorted by cOA, pOA can be generated by a simple counter. The proposed design eliminates a separate OA coordinate sorter needed in the existing point cloud accelerator with WS dataflow and a different OA filtering rule to generate sorted OA coordinates [36]. Moreover, OS dataflow results in best energy efficiency and latency compared to other dataflows for our workload.

4.2.3 Temporal Modeling

One of the major differences between video and image is that video embeds temporal information between successive frames. To capture temporal information, our work adopts online temporal shift module (TSM) [35] and handles it natively in hardware. TSM enjoys great accuracy-cost trade-off and online performance compared with other popular video understanding algorithms [35].

Fig. 4-14 illustrates how TSM works on a CNN backbone. It does not introduce any changes in the convolution kernel. It only shifts some data in the feature map—part of the channels in a feature map is replaced by the corresponding part from the previous frame. The resulting feature map mixes information from the previous frame and the current frame and serves as the input to the next layer. In this way, convolution captures the information between successive frames. In our work, TSM is

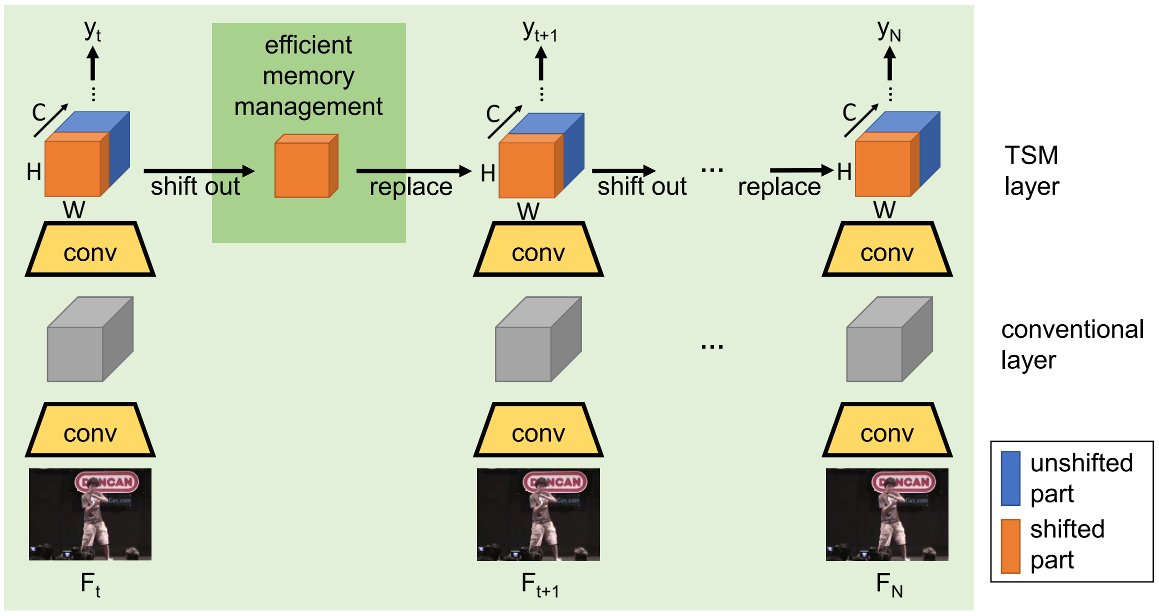


Figure 4-14: An illustration of temporal shift module [35]. C: channel.

applied to a few point-wise convolution layers of the CNNs following [35]. The shifted channel size is set to be a multiple of 8, which takes up the entire DRAM burst size, to achieve good DRAM burst utilization. The efficient data buffering scheme is designed for TSM, which is presented in Section 4.2.4.

4.2.4 Activation Buffering

Compared to conventional convolution, our proposed real-time DiffFrame convolution with temporal modeling has two unique operations on activation orchestration—1) the shifting operation for TSM on some PW layers; 2) the ConvMap-guided reads of IA for sparsity handling in DW layers. We designed a multi-mode IA buffer to handle them efficiently.

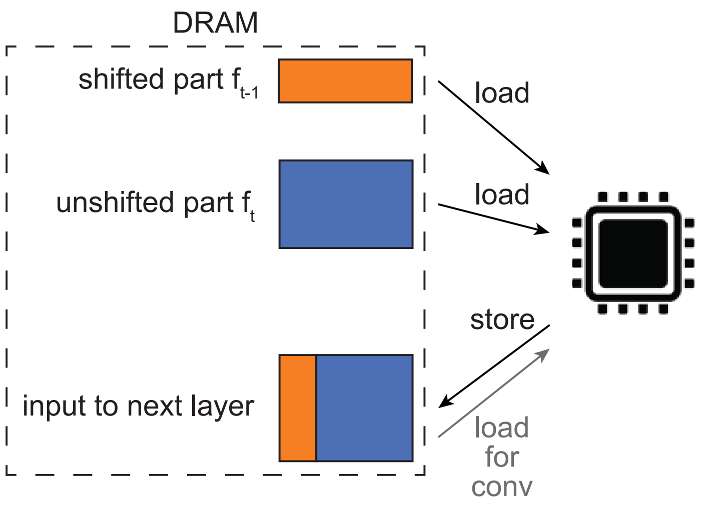
Dual-mode IA Buffering for Temporal Shift Module in Point-wise Diff-Frame SparseConv

The shifting operation of activations is illustrated in Fig. 4-14. A software approach shifts data in DRAM to form the mixed IA for the next layer via load and store operations on the entire feature map as shown in Fig. 4-15a. The DRAM traffic overhead of this approach equals twice the size of IA.

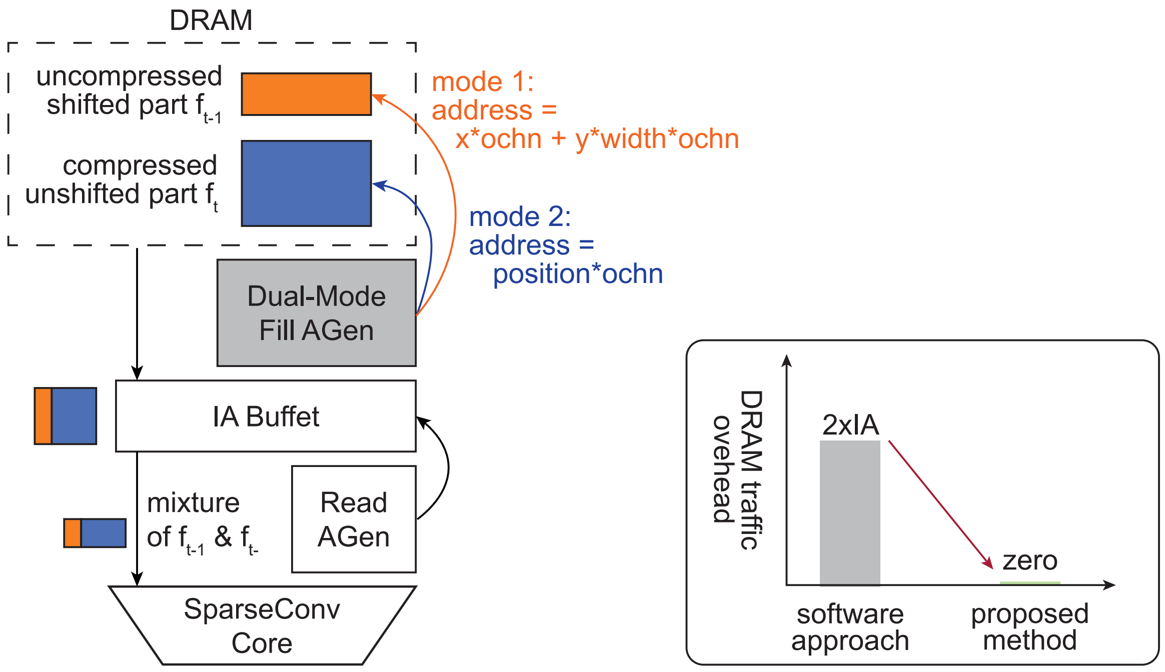
To reduce DRAM traffic overhead, we propose a hardware-based approach with a dual-mode Buffet [42] buffer. The activations of the shifted part from the previous frame and the unshifted part of the current frame are kept separate in DRAM throughout convolution. Different from conventional explicit decoupled data orchestration with Buffet, dual-mode read and fill address generators and IA Buffet [42] are used to handle activation buffering. The dual-mode fill address generator loads and mixes the shifted and unshifted tiles on chip for the next layer with two address generation modes. The shifted part is a small portion of the entire feature map [35] and is stored in the uncompressed format for efficient indexing based on the sparsity pattern of the current frame. The unshifted part is generated by the previous layer of the current frame, stored in the compressed format needed for the current frame, and thus indexed with the position. Without shifting data in DRAM, our proposed dual-mode IA buffering removes the DRAM traffic overhead of TSM.

Implicit Decoupled Data Orchestration for Depth-wise DiffFrame SparseConv

For DW DiffFrame SparseConv, load and store addresses are determined by ConvMap, which can be generated on-the-fly. Fig. 4-16 contains an example ConvMap. As shown, both weight and IA accessing sequences are random with the temporal locality. Since the weight size of DW layers is small, we can store weight fully on-chip and maintain efficient bandwidth for random access. However, IA can be big, e.g., hundreds of kB, and needs DRAM storage. To reduce DRAM traffic, we adopt implicit decoupled data orchestration (IDDO) [42] for IA. We turn off the Buffet [42] controller and the fill address generator used for other layers and activate a direct-mapped cache controller with a block size of 1 pixel with a programmable channel size. The average cache hit rate for stride = 1 layers is 79% and stride = 2 layers is 55%. It greatly reduces DRAM traffic and accessing latency compared to a circular buffer by utilizing the temporal locality. The underlying SRAMs are shared across IDDO for DW DiffFrame SparseConv layers and explicit decoupled data orchestration



(a)


DRAM
uncompressed shifted part f_{t-1}
compressed unshifted part f_t
mode 1: address = x*ochn + y*width*ochn
mode 2: address = position*ochn
Dual-Mode Fill AGen
IA Buffet
mixture of f_{t-1} & f_{t-}
Read AGen
SparseConv Core
DRAM traffic overhead
2xIA
zero
software approach
proposed method

(b)

Figure 4-15: (a) An illustration of software approach of TSM data handling. (b) An illustration of the proposed dual-mode IA Buffet for TSM data handling. AGen: address generator.

in other layers. Since SRAMs take up much more area compared to the controller logic, our method does not lead to a significant increase in the die area.

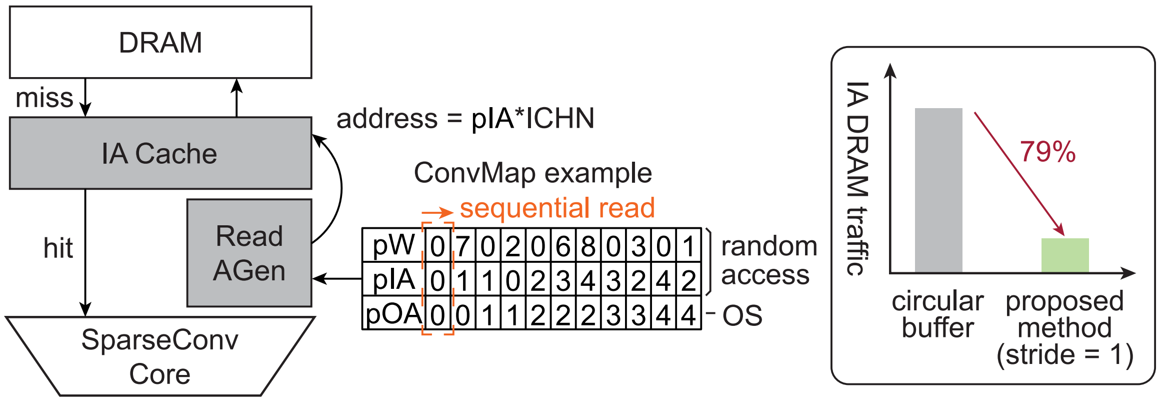


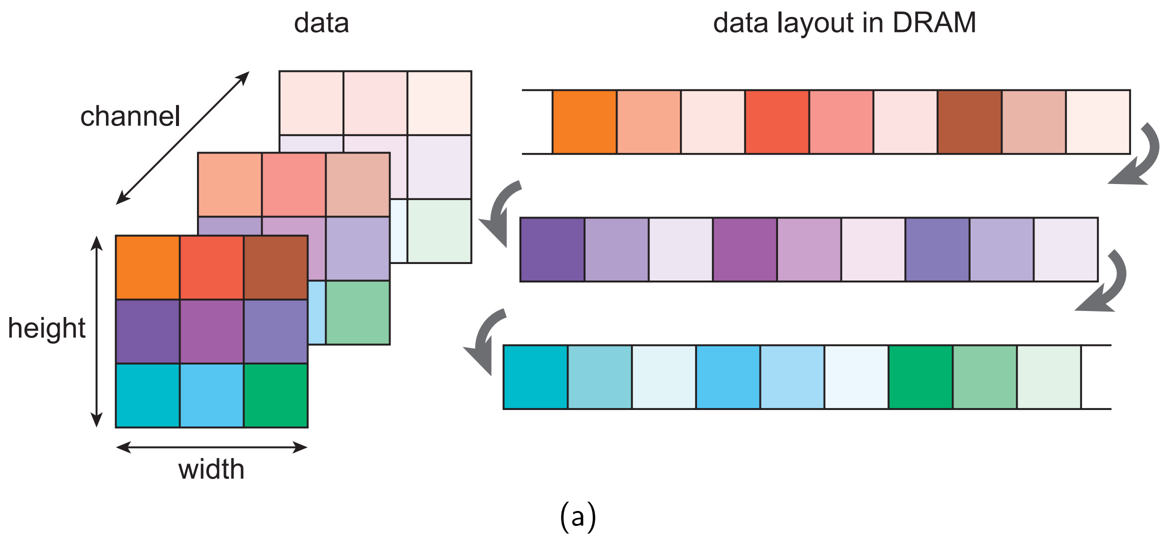
Figure 4-16: A cache-based implicit decoupled data orchestration for ConvMap-guided IA load of DW DiffFrame SparseConv.

Table 4.3: A Summary of Data Types and Their Layout in DRAM

	name	**compressed**	**memory order**
data	current frame	n	row major
	previous frame (& shifted part)	n	row major
	DiffFrame	y	row major
	weight	n	row major
	RefFrame	n	row major
	OA	y	row major
metadata	input coordinate	y	row major
	output coordinate	y	row major
	ConvMap	NA	NA (1D array)

4.2.5 Data Layout in DRAM

Table 4.3 summarizes different data and metadata involved in our proposed real-time DiffFrame convolution and temporal modeling algorithm. Our algorithm works on channel-wise sparsity and thus it is natural to have data in a channel stored together and compute in parallel. So all data are stored in a row-major order with a channel-first sequence, which is illustrated in Fig. 4-17. Our work focuses on the utilization of activation sparsity, thus DiffFrame and OA are compressedly stored. Other data remain uncompressed. Input and output coordinates are also stored in row-major



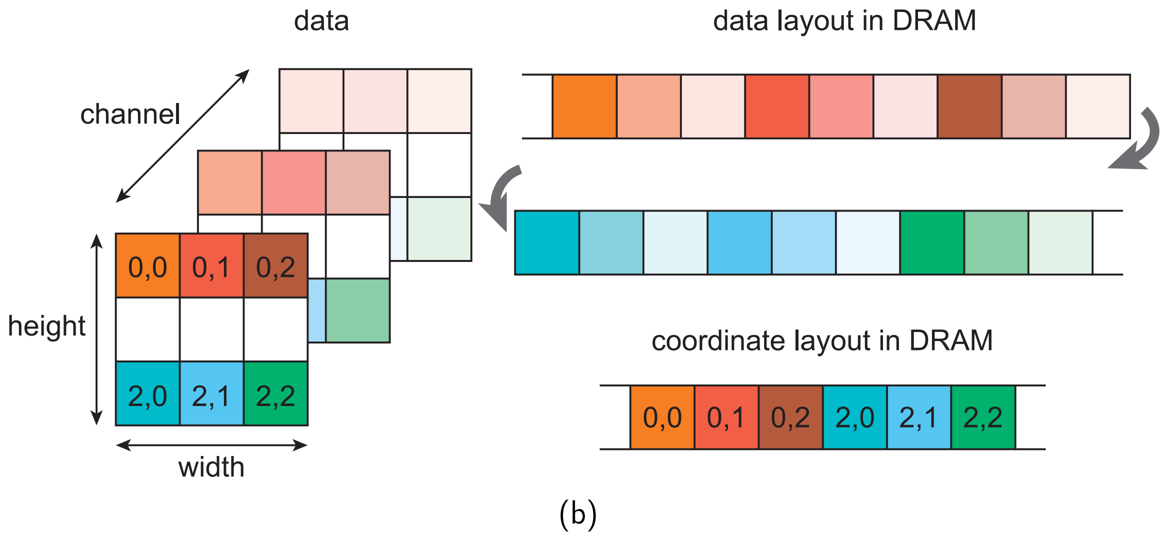


Figure 4-17: An illustration of row-major memory order for data and coordinate with a simple $3 \times 3 \times 3$ tensor. (a) Uncompressed storage. (b) Compressed storage (zero data are shown in white).

order to match the corresponding data (DiffFrame and OA) as shown in Fig. 4-17b. ConvMap is and stored as a 1D array of the set (pIA, pW, pOA) sorted by pOA, where pIA, pW, pOA are the positions of IA, W, and OA respectively.

4.2.6 Other Components and Overall Architecture

Fig. 4-18 shows the overall architecture of the proposed accelerator. It is designed with a DRAM bandwidth limit of 800 MB/s. A round-robin arbiter handles multiple

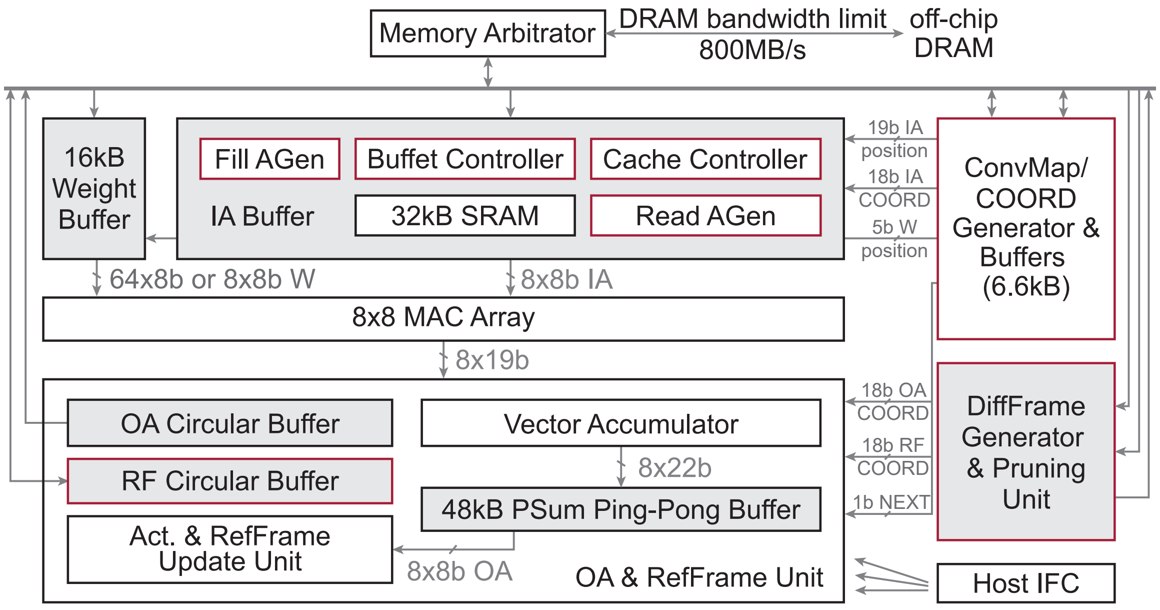
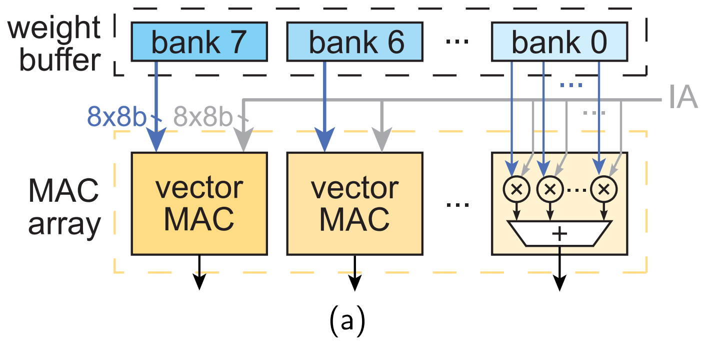


Figure 4-18: Overall system architecture. AGen: address generator; gen.: generator; PSum: partial sum; RF: RefFrame; IFC: interface; COORD: coordinate.

simultaneous DRAM requests from various buffers.

The convolution workhorse is a dual-mode 8×8 MAC array (composed of 8 vector MACs) for MobileNet-v2-based workload. As shown in Fig. 4-19a, the 8×8 MAC array is fully activated for standard convolution handling 8 input channels and 8 output channels in parallel. For DW convolution, only 8 multipliers are activated to handle 8 channels in parallel and the rest can be data gated as shown in Fig. 4-19b.

To provide enough bandwidth of weights and high utilization of SRAMs, a dual-mode weight buffer is proposed. It follows a Buffet-based EDDO [42] and contains 8 banks. Each has a read bandwidth of 8×8 bits and 256 entries. For standard convolution, all banks of the weight buffer are read every cycle providing a total read bandwidth of $8 \times 8 \times 8$ bits for the fully-activated 8×8 MAC array. The total weight buffer entry is 256. For depth-wise separable convolution, banks are reorganized so that only one bank of the weight buffer is read each cycle providing a total read bandwidth of 8×8 bits for the partially activated MAC array. The rest of the banks are also utilized to provide an equivalent total weight buffer entry of 256×8. Compared to only activating one bank and power gating the rest, our proposed dual-mode weight buffer achieves a larger storage size for depth-wise separable convolution.



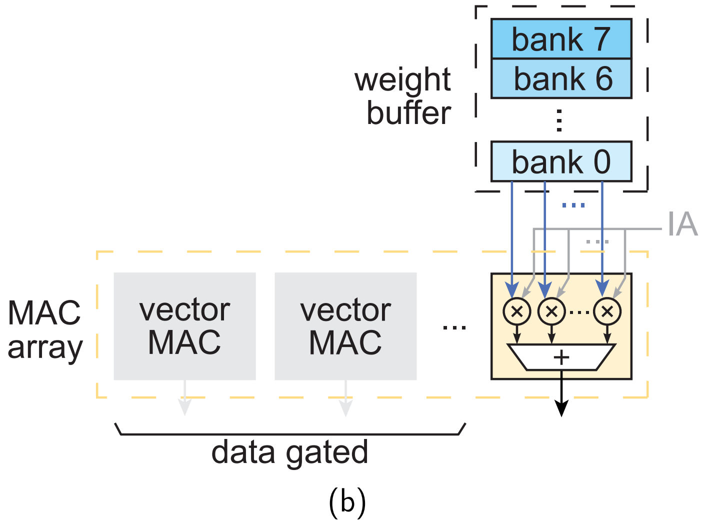


Figure 4-19: The dual-mode MAC array and weight buffer. (a) Standard convolution. (b) DW convolution.

The multi-mode IA buffer is presented in Section 4.2.4 providing 8×8 bits read bandwidth to the MAC array. A vector accumulator handles the temporal accumulation of partial sums at the output of the MAC array with a partial sum ping-pong buffer. The activation and RefFrame update unit applies the activation function and RefFrame addition and generates DiffFrame for the next layer. Circular buffers are used for RefFrame load and store, and OA store.

For standard convolution and PW convolution, OS dataflow is used between DRAM and on-chip buffers, and both OS and WS dataflows are supported between on-chip buffers and the MAC array. A custom mapping optimizer (discussed in Section 4.3.2) is designed to search for the best dataflow and tiling given a CNN layer and design metrics (such as EDP, throughput, etc.).

For DW convolution, OS dataflow is applied with a ConvMap guiding the com-

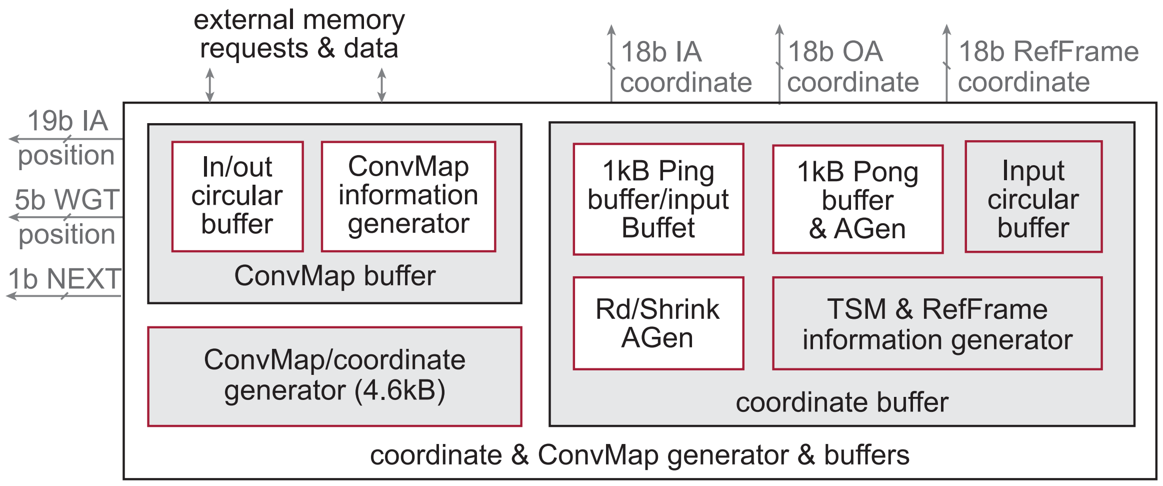


Figure 4-20: The block diagram of the ConvMap and coordinate generator and buffers. WGT: weight.

putation. A detailed block diagram of the ConvMap and coordinate generator and buffers is shown in Fig. 4-20. The design of ConvMap and coordinate generator is presented in Section 4.2.2. A circular buffer is used for the ConvMap load and store between DRAM. The ConvMap information generator outputs a NEXT signal to the partial sum buffer indicating when the vector accumulator needs to move forward to the next partial sum. A multi-mode coordinate buffer is designed to handle different buffering requirements for different layers. 1) Some layers directly load coordinates from off-chip DRAM and then use those coordinates to guide the load and store of RefFrame or shifted part for TSM. Thus, a Buffet-based buffer is needed. 2) The starting layer of our DiffFrame SparseConv directly receives coordinates from the ConvMap and coordinate generator, and needs to use them for RefFrame and then store them off-chip. A ping-pong buffer is used. 3) Some layers need to load coordinates from off-chip DRAM for the pruning unit and the coordinate generator to produce a new set of coordinates, use the new coordinate for computation and then store them in DRAM. In this case, a circular buffer is used to buffer input coordinates to the pruning unit and then a ping-pong buffer is used for the output coordinates from the coordinate generator.

As presented above, a lot of blocks are conditionally needed based on the layer type and operation mode. To reduce power consumption, user-configured block-level clock gating is applied to red blocks as shown in Fig. 4-18 and Fig. 4-20 based on the operational need of each layer. It brings 1.3x power reduction on MobileNet-v2 workload.

4.3 Evaluation and Test Setup

This section describes how we set up the experiments to evaluate the proposed algorithm and architecture in Section 4.2.

4.3.1 Model Preparation and Algorithm Evaluation Setup

We use MobileNet-v2-based models [46] as the backbone to evaluate the proposed algorithm. MobileNet-v2 achieves higher accuracy and lower runtime with fewer parameters than various NNs, e.g., MobileNet-v1 and ShuffleNet, on mobile devices on multiple applications, e.g. object detection, etc. [46]. Since our work targets edge applications, it is suitable to serve as a backbone.

Both a basic MobileNet-v2 model (MobileNet-v2-47) and a reduced MobileNet-v2 model (MobileNet-v2-34) were used during evaluation and measurements. Appendix B shows their model structures. The MobileNet-v2-34 is constructed from the ordinary MobileNet-v2 by 1) having a width multiplier of 0.68, a channel multiplier of 0.5, and a depth multiplier of 0.695; 2) advancing one downsampling layer to the beginning of the NN. For both NNs, the DiffFrame is first calculated at the input of the first inverted residual bottleneck block and our proposed method is then applied to the rest of the NN. By advancing the downsampling layer to the beginning, the metadata generation of our proposed sparsity handling method is reduced. Pruning is applied at the input of the first DiffFrame SparseConv layer to achieve higher sparsity in the input DiffFrame and the layers following all the downsampling layers as downsampling densifies the feature maps.

Our proposed algorithm is evaluated on DAC-SDC dataset [61] with ImageNet [9] pretraining. The DAC-SDC dataset includes 95 categories of video clips captured by unmanned aerial vehicles (UAVs) with moving objects inside and labels of bounding

(a)

(b)

(c)

Figure 4-21: A few samples of the DAC-SDC dataset [61].

boxes tracking the moving objects. Some samples are shown in Fig. 4-21. The accuracy of this object-tracking task is evaluated as intersection-over-union (IoU) [61], which is the ratio between the intersection of the predicted bounding box and the ground-truth bounding box and the union of them.

Quantization is applied to the NN during the evaluation. Standard 8-bit linear quantization with fine-tuning is used on weights, IAs, and OAs.

4.3.2 Architecture Evaluation and Chip Measurement Setup

The setup for architecture exploration (algorithm and architecture co-design) and evaluation is shown in Fig. 4-22. We use PyTorch to train and test the proposed algorithm, dump model parameters and intermediate data, and evaluate accuracy. We built a cycle-accurate and bit-accurate model in C++ to explore and evaluate the architecture. With an energy lookup table of major hardware components and the workload description, it logs the number of DRAM and on-chip buffer accesses and provides an estimated cycle count and energy consumption of the underlying architecture given the workload and mapping. A mapping optimizer searches the design space of the runtime parameters to find the optimum for the underlying architecture

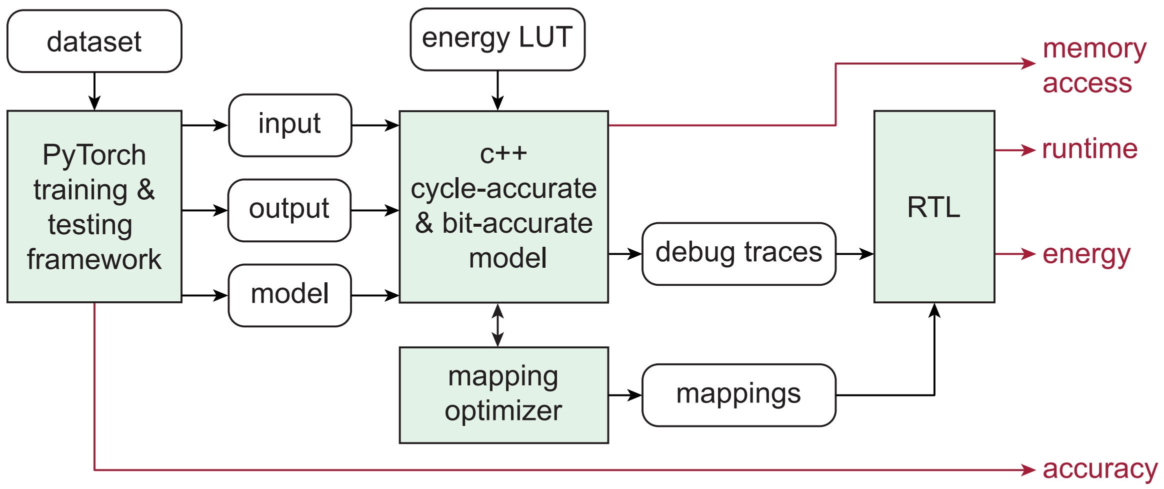
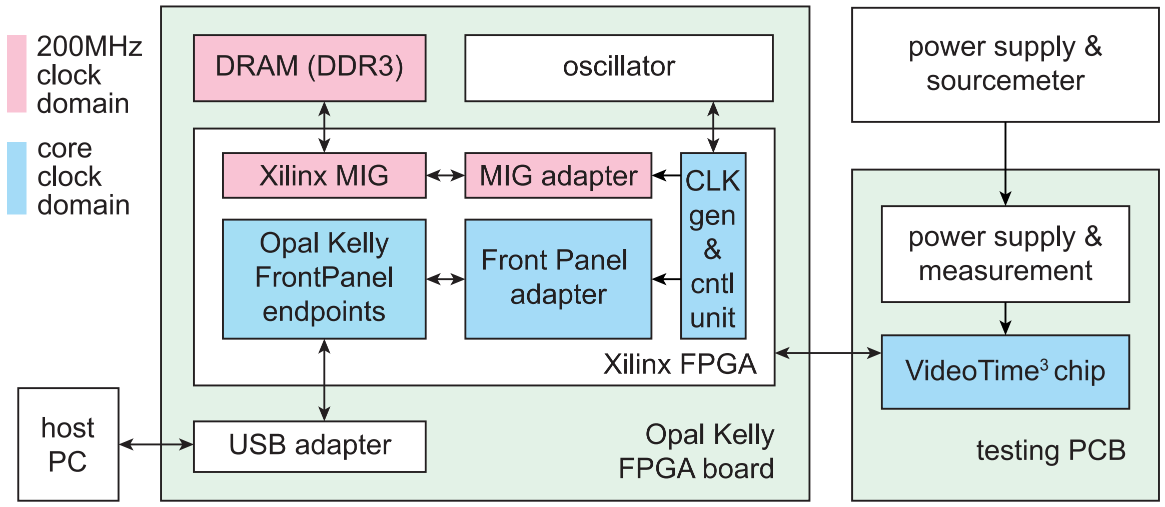


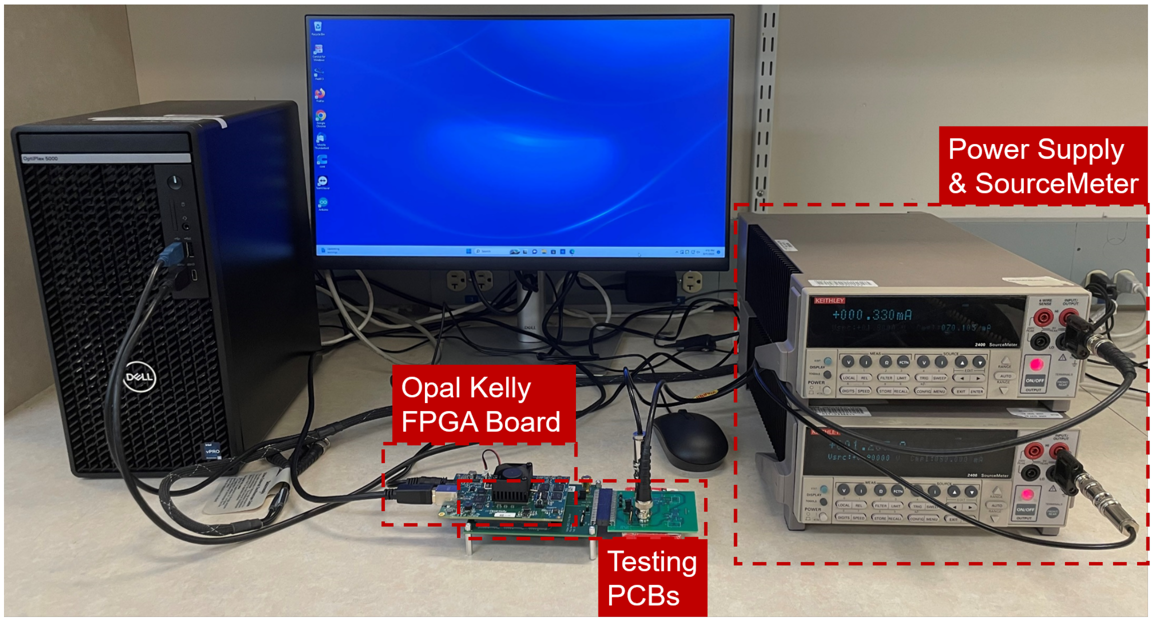
Figure 4-22: The setup for architecture exploration and evaluation. LUT: lookup table.

given a certain target, such as minimum DRAM traffic, highest throughput, or lowest energy-delay product. With the best mappings and data traces dumped from the bit-accurate model, we simulate RTL to get the accurate runtime and energy consumption of the accelerator. We iteratively tuned the architecture and its design-time and runtime parameters to achieve good accuracy and low memory access, runtime, and energy consumption.

We evaluate our proposed architecture with a chip called VideoTime^3 [57] taped out in TSMC 28 nm HPC+ technology. Fig. 4-23 shows the block diagram and a photo of our measurement setup. The host PC is connected to Opal Kelly FPGA board through a USB interface to send 1) the configuration bits to the chip via Opal Kelly FrontPanel endpoints and user-designed FrontPanel adapter, and 2) input data to DRAM via the Xilinx MIG and a user-designed MIG adapter. DRAM, the Xilinx MIG, and the MIG adapter operate at 200MHz, which is the highest clock frequency for DDR3. The rest components operate at the chip core frequency, which is adjustable during measurement. Our VideoTime^3 chip is attached to the test PCB using chip-on-board packaging and supplied with the Keithley power sources.



(a)



(b)

Figure 4-23: (a) The block diagram of the chip measurement setup. FPGA has two clock domains. One is the highest clock frequency for DDR3 and the other is for the chip. gen: generator; cntl: controller. (b) A photo of the chip measurement setup.

4.4 Evaluation and Measurements

Using the experimental setup discussed in the previous section, we evaluate the accuracy of our proposed algorithm and the DRAM traffic reduction, energy breakdown of our algorithm and architecture co-design. In the end, we present the chip measure-

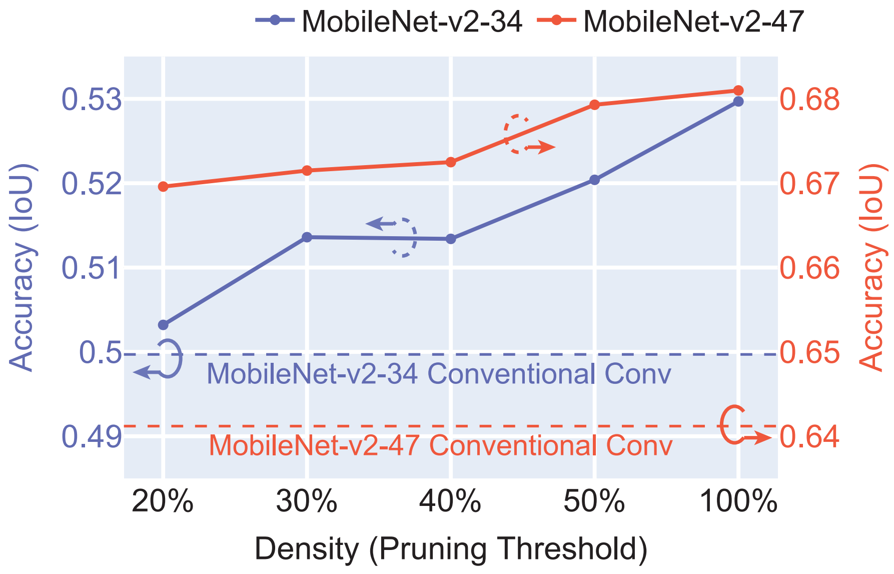


Figure 4-24: Comparison with conventional convolution and sensitivity of accuracy on the pruning threshold of the proposed real-time DiffFrame SparseConv with temporal modeling algorithm. The NN is not quantized. The case with a density of 100% is equivalent to convolution with TSM. The dash lines are plain conventional convolution.

ment and a comparison with prior work.

4.4.1 Accuracy

We compare our proposed real-time DiffFrame SparseConv with temporal modeling algorithm with conventional convolution and evaluate the sensitivity of accuracy on the pruning threshold of our algorithm. During this analysis, we use a density-based pruning threshold, which prunes the less significant pixels based on the L1 norm and keeps the pruned data at a specified density. During online processing, the user can use a value-based pruning threshold, which prunes data given a preset value. The pruning threshold is set according to user analysis and estimation of the input data given their target application. As shown in Fig. 4-24, our proposed algorithm improves the accuracy over conventional convolution even at 20% density (nonzeros). MobileNet-v2-47, which uses larger input resolution and has more parameters, is less

Table 4.4: Accuracy Analysis of the Proposed Method on Quantized MobileNet-v2 Neural Nets

	Conventional Convolution	Convolution with TSM	This Work
Real-Time DiffFrame SparseConv	X	X	✓
Temporal Shift Module	X	✓	✓
$\mathrm{IoU}_{\text{MobileNet-v2-34}}$ (%)	48.8	52.7	51.3
$\mathrm{IoU}_{\text{MobileNet-v2-47}}$ (%)	65.4	68.7	67.4

sensitive to pruning compared to MobileNet-v2-34.

Furthermore, we evaluate the accuracy of quantized MobileNet-v2 NNs at a density of 30% in three cases—conventional convolution, convolution with TSM, our proposed real-time DiffFrame convolution with temporal modeling. The results in summarized in Table 4.4. Although DiffFrame SparseConv and pruning lead to some information loss, overall accuracy (IoU) improves 2%–2.5% on various MobileNet-v2-based NNs on the object tracking dataset [61]. When SparseConv is directly applied to raw IA instead of DiffFrame, the accuracy drops to less than 30% at 40% density. The intuition may be that DiffFrames inform NNs which pixel locations have big changes between frames and are crucial for object tracking. The study emphasizes the importance of each component of our proposed real-time DiffFrame convolution with temporal modeling.

4.4.2 DRAM Access

We compare the DRAM traffic between our proposed real-time DiffFrame SparseConv with temporal modeling and conventional convolution on our proposed hardware and evaluate how DRAM traffic changes with the input density of the proposed algorithm on two different NNs. Our hardware supports both the proposed algorithm and conventional convolution (this mode is used in the first few layers before the first DiffFrame layer). The optimal mappings for both the proposed algorithm and conventional convolution are used during the evaluation. As shown in Fig. 4-25, our

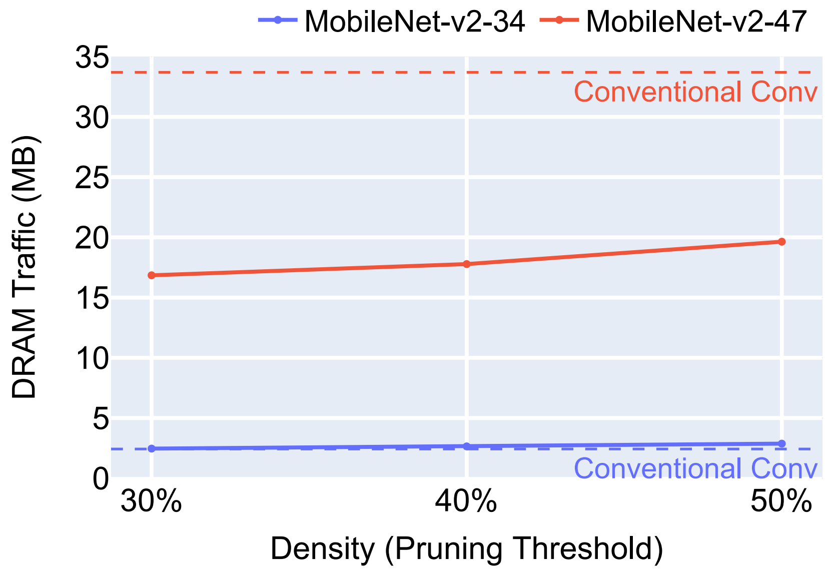


Figure 4-25: The total DRAM traffic vs. the pruning threshold (input density) of the proposed real-time DiffFrame SparseConv with temporal modeling algorithm and a comparison with conventional convolution. The target of the mapping optimizer is set to be minimizing DRAM traffic.

proposed method significantly reduces the DRAM traffic for MobileNet-v2-47 across various density levels. For MobileNet-v2-34, a higher level of pruning is needed to achieve DRAM traffic reduction with our proposed algorithm. The reason is the following. MobileNet-v2-34 has a much smaller channel size than MobileNet-v2-47 as shown in Appendix B. Our proposed algorithm features channel-wise sparsity and thus leads to less DRAM traffic reduction on MobileNet-v2-34 compared to MobileNet-v2-47. Nonetheless, a 1.8x reduction in the energy-delay product of the accelerator chip at 30% density is observed on MobileNet-v2-34 as shown in Section 4.4.4.

Fig. 4-26 shows a detailed breakdown of DRAM traffic and compares conventional convolution, estimated conventional frame-by-frame DiffFrame sparsity handling method, and our proposed method. The conventional frame-by-frame DiffFrame sparsity handling method is estimated by adding full RefFrame traffic, which is twice the size of total IAs, and the weight, IA, and OA DRAM traffic of our proposed method. The naive frame-by-frame DiffFrame sparsity handling method con-

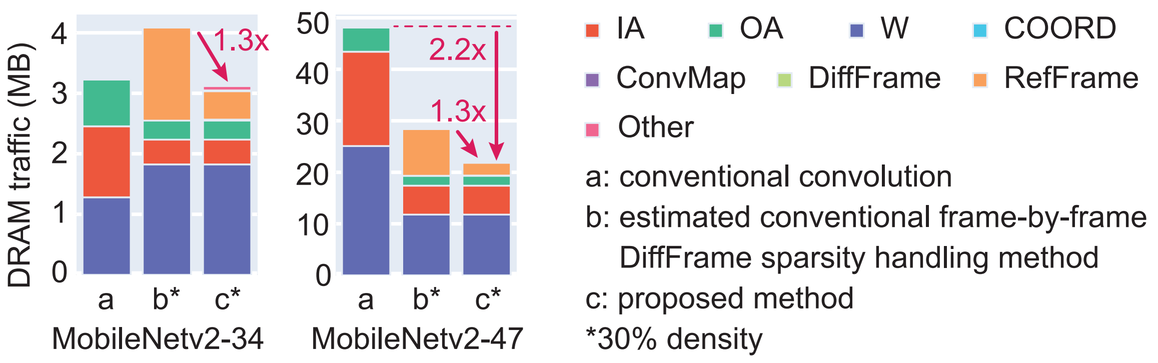


Figure 4-26: The proposed method reduces DRAM traffic by 1.3x and 2.2x compared to the prior sparsity handling method and conventional convolution on MobileNetv2-47 respectively. The mapping optimizer target is to lower the energy-delay product. MobileNetv2-34 has a small channel size; thus, our method featuring channel-wise sparsity leads to less DRAM traffic reduction. Nonetheless, a 1.8x reduction in the energy-delay product of the accelerator is observed. W: weight. COORD: coordinate.

sumes higher DRAM traffic than conventional convolution at 30% input density with MobileNetv2-34 workload due to the large overhead of RefFrame traffic. Our proposed method greatly reduced the RefFrame traffic and IA, OA, and weight DRAM traffic with very little overhead in sparsity handling metadata. It achieves 1.3x and 2.2x DRAM traffic reduction compared to the conventional sparsity handling method and conventional convolution respectively on the MobileNet-v2-47 workload.

4.4.3 Energy Breakdown

The energy breakdown of our accelerator running the proposed algorithm with the MobileNet-v2-34 backbone is shown in Fig. 4-27. The energy is obtained from post-PnR simulation with timing, switching activity and parasitic information annotated. The OA & RefFrame Unit (including PSum, OA and RefFrame buffers, the vector accumulator, and activation and RefFrame update unit as shown in Fig. 2-6a) and the weight buffer take up around 50% of the total energy consumption. Adding support to the OS-LWS dataflow proposed in Chapter 2 may further reduce their energy consumption. The energy overhead of having the ConvMap and Coordinate generator and buffer, DiffFrame pruning unit and the DiffFrame generator for our proposed algorithm is very small.

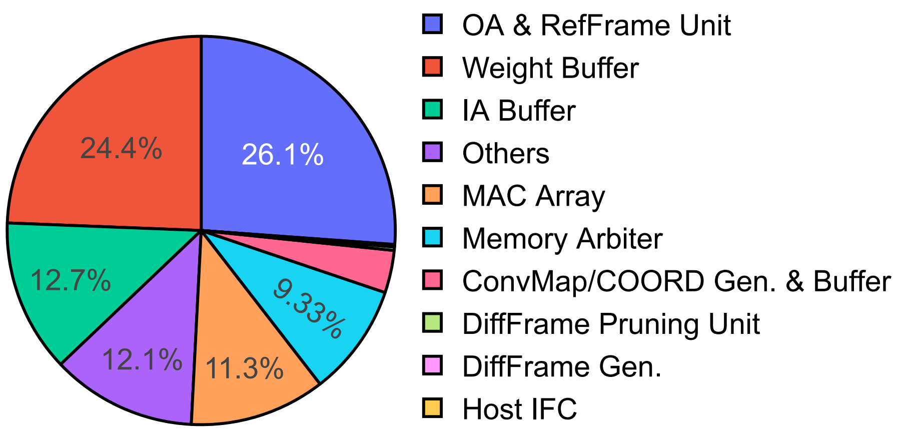


Figure 4-27: Energy breakdown of the accelerator chip (including IO drivers) running MobileNet-v2-34. DRAM access time and energy are not included. COORD: coordinate; Gen.: generator; IFC: interface.

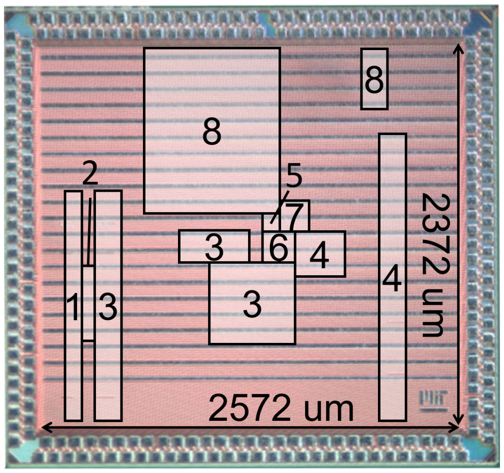
4.4.4 Chip Measurements and Comparison with Prior Work

The accelerator is fabricated in 28nm CMOS. The die micrograph is shown in Fig. 4-28a. We measure the accelerator chip running the proposed real-time DiffFrame convolution with temporal modeling algorithm with the MobileNet-v2-34 backbone. The voltage-frequency sweep is shown in Fig. 4-28. Achieving 50MHz at 0.6 V, it consumes 40 uJ/frame (DRAM excluded) with 1.01 TOPS/W and 38 FPS with an average IA sparsity of 63.1%. We also measure the accelerator chip running the conventional convolution with the same backbone. As shown in Fig. 4-29, our proposed techniques reduce the EDP by around 1.8x compared to conventional convolution. Furthermore, our work achieves a 6.3x improvement in energy efficiency (TOPS/W) compared to [64] and the lowest energy/frame compared to prior work as shown in Table 4.5.

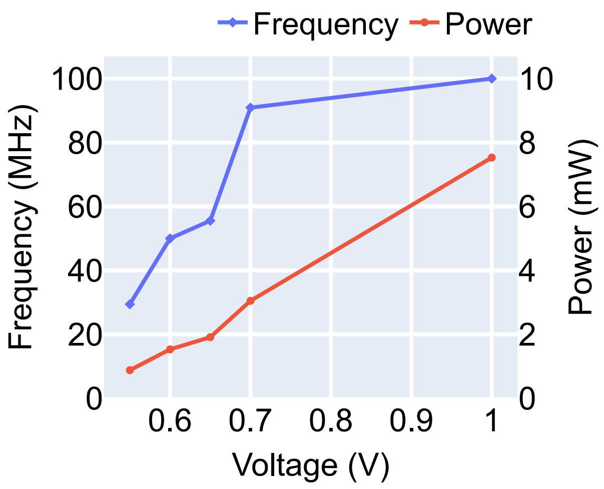
4.5 Further Discussions on CNN Sparsity Handling

In Section 4.2.2, we present how our work handles sparsity in the proposed real-time DiffFrame convolution. In this section, we analyze the existing sparsity handling techniques and describe how our work is different from prior work.

Sparse data can be utilized in various ways. Based on existing works and analy-



(a)



(b)

Figure 4-28: (a) Die micrograph (1: 16 kB Weight buffer; 2: 8 × 8 MAC array; 3: 32 kB IA buffer; 4: 44 kB OA RefFrame unit; 5: DiffFrame generator; 6: DiffFrame pruning; 7: ConvMap buffer; 8: ConvMap generator and coordinate buffer). (b) Frequency and power measurements.

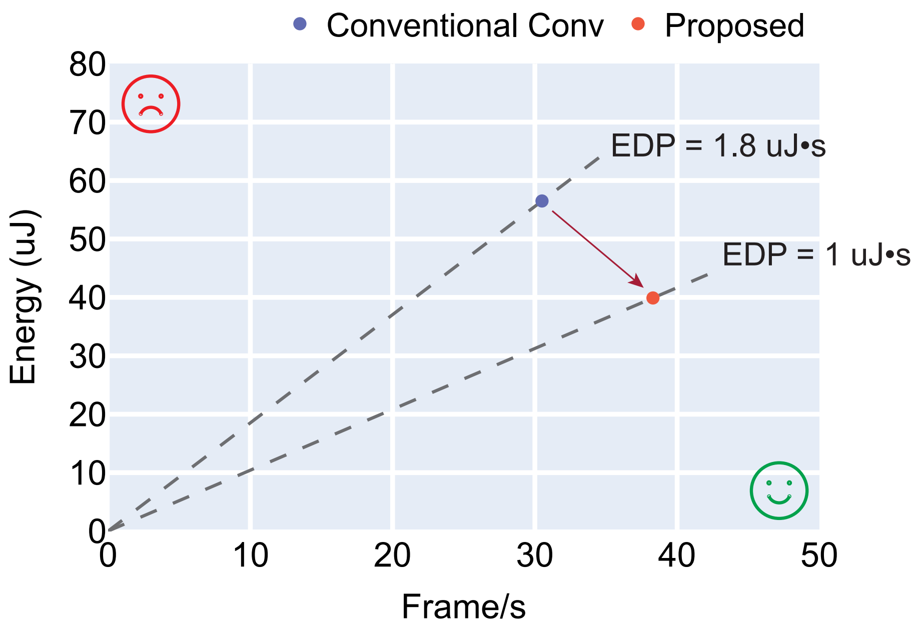


Figure 4-29: Comparison of the proposed real-time Diffframe convolution and conventional convolution in terms of energy, frame rate and energy-delay product (EDP) on our accelerator chip.

Table 4.5: Comparison to Prior Work

	JSSC'18 [1] 1—3,5)	ISSCC'20 [3] 1—5)	ISSCC'21 [2] 1,2,4,5)	This work 1—5)
Temporal modeling?	N	N	N	Y
Temporal redundancy utilized?	N	Y	N	Y
Single-frame latency?	N	N	Y	Y
Technology	65 nm	65 nm	28 nm	28 nm
Core vol. (V)	0.7~1.1	0.52~0.99	0.6~0.9	0.55~1.0
Clock freq.	10~200 MHz	50 MHz	100~470 MHz	29~100 MHz
Precision	8b W & IA	8b W, 4b/8b IA	8b W & IA	8b W & IA
DRAM BW limit	not mentioned	800MB/s	not mentioned	800MB/s
Workload	AlexNet	MNet-v1-27	ResNet	MNet-v2-34
Dataset	ImageNet	custom dataset	ImageNet	DAC-SDC
Accuracy	56.9% (top-1)	not mentioned	76.92% (top-1)	51.3% IoU
Throughput (GOPS)	not mentioned	1.32~1.97	not mentioned	0.9~3.1
Frame/sec.	20 ~ 346	44 ~ 67	40	23 ~ 77
Energy/frame	310 ~ 838 uJ	183 ~ 278 uJ6)	1120 uJ	39 ~ 98 uJ

1) 1MAC=2OP; 2) BatchNorm, softmax not included; 3) MLP not included; 4) DRAM access time not included; 5) DRAM access energy not included; 6) The minimum energy/frame reported in [3] is 24.7uJ with a tiny MNet-v1-16 (3.2x fewer operations compared to our MNet-v2-34). Since [3] does not report the corresponding throughput, we note this case here for the clarity of the table. *MNet: MobileNet.

sis [51, 6], we classify them into two categories based on their target components. One is the computation-related techniques shown in Table 4.6. A basic approach is to gate computation with zero weight and/or IA to reduce the dynamic energy of computing units. An example can be found in Eyeriss [6]. A more aggressive approach skips the computation cycle along with computation when weight and/or IA is zero, such as Cnvlutin [1] and SCNN [41]. The other category is the memory-related techniques shown in Table 4.7. Some work, such as Eyeriss [6], skips memory access of an input when the other input is zero for the computation. As skipping memory access only applies to low memory level(s) close to the computing units, it does not save major data delivery costs. Other work compresses the sparse data in part of or all of the memory

Table 4.6: Computation-Related Sparsity Handling Techniques

	Computation Energy	Computation Cycles
Gate Computation	⬇	▬
Skip Computation	⬇	⬇

Table 4.7: Memory-Related Sparsity Handling Techniques

	Memory Access	Data Delivery	Storage Size
Skip Memory Access	⬇	▬	▬
Compress Data	⬇	⬇	⬇

levels so that associated memory access, data delivery and storage size are reduced. A common approach is to compress the data traffic between the off-chip DRAM and the accelerator chip as shown in Eyeriss [6] and Envision [38]. Another well-studied approach is to convolve with compressed weights only, such as Cambricon-X [65]. Since weights stay the same during inference, they can be compressed in advance according to the need of the designer. Thus, the sparsity pattern of weights is known in advance and metadata can be generated offline. That simplifies the inference accelerator design compared to handling sparsity in activations as activations are generated on-the-fly and their metadata needs to be processed online. In the following analysis, we focus on activation sparsity handling with the more aggressive technique in each category—skipping computation and data compression—as they potentially provide more benefits.

Table 4.8 summarizes the ineffectual computation skipping and activations and PSums compression in different memories in prior work. Some prior works explored the handling of compressed IAs in all memory levels for convolution, such as Cnvlutin [1] and SCNN [41]. However, prior work does not consider the sparsity handling of PSums/OAs across all memory levels. The reason is that PSums/OAs have a high probability to be dense in conventional convolution due to dilation. In contrast, our work proposes real-time DiffFrame convolution eliminating dilation (with specific

Table 4.8: Comparison to Prior Work in Terms of Sparsity Handling in Activations and Partial Sums

		Conventional Convolution	Prior Work			This Work
Skip zero computation?		X	✓	✓	✓	✓
Is compressed in DRAM?	IA	X	X	✓	✓	✓
	OA/PSum	X	X	✓	✓	✓
Is compressed in SRAM?	IA	X	X	X	✓	✓
	OA/PSum	X	X	X	X	✓

rules between nonzero input and output coordinates) to introduce sparsity in PSums and OAs. With algorithm and architecture co-design, our work handles convolution with compressed PSums/OAs in all memory levels. That expands the sparse CNN accelerator design space.

Furthermore, our work introduces decoupled metadata generation and map-guided sparse convolution to efficiently convolve compressed PSums/OAs. The definition and explanation of decoupled metadata generation is presented in Section 4.2.2. In contrast, we classify the prior sparsity handling architectures as coupled metadata generation and coordinate-guided sparse convolution. Coupled metadata generation refers to the design in which the processing sequence of metadata is associated with that of compressed data. The reasons prior work used coupled metadata generation are that 1) prior work needs coordinates to index uncompressed PSums during accumulation for convolution whereas our work directly uses position to index PSums, 2) the PSum coordinate sequences are different for every layer providing no reuse opportunities, and thus they need to be generated with PSums to guide accumulation (whereas metadata are shared across the layers with a stride equal to 1 in our work). To analyze prior work, a sparse fiber tree representation is proposed, which uses coordinates as indices to present sparse dataflow as an abstraction of the CNN accelerator architecture [51]. To represent our work more straightforwardly, we illustrate our map-guided convolution and decoupled metadata generation as shown in the pseudo-codes of Fig. 4-12 and 4-13a, where positions are used to index data during convolution and metadata generation is fully decoupled from convolution. That provides an alternative way of representing sparsity handling architectures of NN accelerators.

4.6 Summary and Conclusions

This section presents an algorithm and hardware co-design of a deep learning accelerator, VideoTime^3, optimized for state-of-the-art video understanding applications. The chip is innovative as it achieves all three features—is capable of temporal modeling for higher accuracy, utilizes temporal redundancy to improve energy efficiency, and achieves single-frame latency for real-time applications, such as autonomous vehicles and AI drones. Our work has four key contributions: 1) propose real-time DiffFrame convolution achieving 2.2x DRAM access reduction on MobileNet-v2 workload at single-frame latency compared to conventional convolution; 2) introduce decoupled metadata processing and map-guided convolution to efficiently handle compressed PSums/OAs and IAs resulting from algorithm co-design and design sorter-free architecture for the metadata generator; 3) enable temporal modeling and have 2% – 2.5% accuracy (IoU) improvement with real-time DiffFrame convolution on DAC-SDC object tracking dataset [61]; 4) optimize data buffering to remove DRAM traffic overhead for temporal modeling and reduce 55% – 79% IA DRAM traffic in depthwise layers. The chip consumes 40 uJ/frame with 38 frames/second at 0.6 V in 28nm CMOS.

We focus on the efficient handling of activations for real-time video understanding applications considering redundancy and information between successive video frames in our analysis. However, the proposed techniques can also be applied to other applications that have similarities between multiple inputs, such as data from MRI machines. Also, they can be applied with well-known weight sparsity handling methods to boost energy savings and speed even further.

Chapter 5

Conclusions and Future Directions

Addressing the challenge of the immense demand for the computing power while maximizing potential of deep learning has been a crucial and highly sought-after focus in both the research community and industry. Designing across the boundaries of circuit, architecture and algorithm, this thesis provides unique insights and presents novel techniques to tackle this challenge.

5.1 Summary of Contributions

- Hybrid dataflows and memory hierarchy co-design

  This thesis shows that maximizing the reuse of a single data type, such as fully weight/output/input stationary, may not lead to the highest energy efficiency. We argue that the designer should explore all possible storage elements provided by a chosen technology for each data type and consider balancing the reuse across different data types given the energy and access time of different storage elements.

  We propose hybrid dataflows that balance the reuse of weight and partial sums and introduce a new level of the memory hierarchy—the collector—between the on-chip buffer and compute units. Our proposed output stationary with local weight stationary dataflow improves the energy efficiency of two NN accelerators analyzed in this thesis by 1.5x – 2x. This technique is also adopted in a more recent accelerator for tranformers and significantly reduces the overall proportion of power used for memory access and data delivery compared to prior work. It can be generalized to accelerators for various applications, such as NN training and general tensor algebra, and various types of accelerators, such as accelerators on the edge and accelerators on the cloud.

- Weight tuning algorithm and datapath/SRAM co-design

  We looked into the data statistics in the accelerator and investigated how they affect the overall energy consumption. As the designer has a full control over weights for inference, we propose a weight tuning algorithm that reduces the toggle count of weight sequence by tweaking the bit representation of weights. The reduction in toggle count of weight leads to dynamic energy reduction of weight delivery and computation with the co-designed datapath. Adopting a custom SRAM with conditional precharge, this technique also reduces the energy consumption of the weight buffer.

- Algorithm and architecture co-design for real-time video understanding with temporal redundancy reduction and temporal modeling

  When designing an accelerator for a specific application, investigating the processed data and co-designing the application-specific algorithm bring more flexibility and provide powerful design knobs to improve energy efficiency. Following this design methodology, we propose a real-time DiffFrame convolution with temporal modeling algorithm for video understanding, which captures the useful information in video data for higher application accuracy while utilizing redundancy in the data for energy savings.

- Decoupled metadata processing and map-guided sparse convolution with compressed OA

  The proposed real-time DiffFrame convolution for video understanding provides unique sparsity handling opportunities—sparse PSum and OA. We propose decoupled metadata processing and map-guided sparse convolution to efficiently enable compressed PSum/OA across all memory hierarchies. Compared to conventional sparse convolution with uncompressed PSum and OA, our method delivers around 3x reduction in storage size for PSum/OA at 30% density and achieves a significant reduction in overall memory access and data delivery. We expand the taxonomy of sparsity handling architectures for convolution with our decoupled metadata generation and map-guided sparse convolution. Besides video, this method can be applied to other data that present some similarities or correlations across batches, such as images from MRI.

- Test chips

  We taped out two test chips in this thesis to evaluate the proposed techniques, including 1) the NN accelerator in 40-nm technology supporting the proposed weight tuning algorithm for small footprint NNs, and 2) the VideoTime^3 accelerator in 28-nm technology for video understanding with our proposed techniques.

- Evaluation of eMRAM

  We explore the application of emerging eMRAM on DL accelerators in this thesis. After comparing the characteristics of eMRAM, DRAM and SRAM, we used eMRAM for the last level weight storage to keep all the weights on chip. Through dataflow and buffer scheme optimization, the energy consumption of eMRAM is less than 6% of the total energy consumption of the NN accelerator.

5.2 Future Directions

Given the significant advancements in DL applications, such as generative AI and autonomous vehicles, and their large computation demand, there is increasing demand and exciting opportunities for efficient DL accelerators. Here are some possible extensions of this work:

- **More Efficient Sparsity Handling**: The activation sparsity handling techniques developed in this work can be applied along with existing efficient weight sparsity handling techniques. That will enable compression in all data types involved in convolution, leading to higher utilization of sparsity and thus more opportunities to achieve higher energy efficiency.

- **Generalization to Other Applications**: It would be interesting to explore the generalization of the techniques proposed in this thesis to other applications, such as generative AI and graph processing, which also involve NNs or sparse tensor algebra.

- **New Technologies**: While all our chips were fabricated in CMOS technology, we also explored the application of the emerging eMRAM technology on DL accelerators with an analytical model and obtained promising results. It would be useful to design and fabricate the chip with emerging technologies. Moreover, while our techniques are designed for a single chip, it would be beneficial to incorporate them into DL accelerator systems with chiplet, which is a promising technology to further push the limit of energy efficiency.

- **Compiler Design and Software Ecosystem**: While our work explores algorithm and hardware co-design for a given application and develops custom mappers for our proposed algorithms and hardware, it would be useful to investigate how to jointly develop the compiler and build a better software ecosystem along with the DL accelerator design in a systematic way.

Appendix A

List of Abbreviations

DL	deep learning
GPT	generative pre-trained Transformer
IA	input activation
OA	output activation
PSum	partial sum
FMap	feature map
EDP	energy-delay product
NN	neural network
WS	weight stationary
IS	input stationary
OS	output stationary
eMRAM	embedded magnetoresistive RAM
OS-LWS	output stationary with local weight stationary
WS-LOS	weight stationary with local output stationary
PE	processing element
SCM	standard cell memory
KWS	keyword spotting
CNN	convolution neural network
NoC	network on chip
FIR	finite impulse response

AR	augmented reality
VR	virtual reality
AI	artificial intelligence
TSM	temporal shift module
PW	point-wise (convolution)
DW	depth-wise (convolution)
UAV	unmanned aerial vehicle
IoU	intersection-over-union

Appendix B

Custom NN Model structures

We present the structures of custom NN models used in this thesis.

Table B.1: The Original and Decomposed Structure of CNN80 (The input size of CNN80 is $H = 10$ and $W = 49$.)

Original							Decomposed						
Layer	FH	FW	ICHN	OCHN	SH	SW	Layer	FH	FW	ICHN	OCHN	SH	SW
CONV1	4	10	1	28	1	1	CONV1-1	1	10	1	6	1	1
							CONV1-2	4	1	6	9	1	1
							CONV1-3	1	1	9	28	1	1
CONV2	4	10	28	30	2	1	CONV2-1	1	1	28	18	1	1
							CONV2-2	4	10	18	21	2	1
							CONV2-3	1	1	21	30	1	1
LIN	1	1	1920	16	NA	NA	LIN-1	1	1	1920	12	NA	NA
							LIN-2	1	1	12	12	NA	NA
							LIN-3	1	1	12	16	NA	NA
FC1	1	1	16	128	NA	NA	FC1	1	1	16	128	NA	NA
FC2	1	1	128	12	NA	NA	FC2	1	1	128	12	NA	NA

* LIN: linear layer; FC: fully-connected layer; SW: stride in width; SH: stride in height

Table B.2: The Structure of MobileNet-v2-47 (The input size is $H = 160$ and $W = 360$. The proposed real-time DiffFrame convolution with temporal modeling algorithm is applied to layer 4 and the following layers. Metadata need to be generated for layer 4, 16, 17, 37 andd 38. The last layer, which is a fully connected layer, is not included in our evaluation.)

Layer IDX	OCHN	ICHN	FH	FW	Stride	Group
1	32	3	3	3	2	1
2	32	32	3	3	2	1
3	128	32	3	3	2	1
4	128	128	3	3	1	128
5	24	128	1	1	1	1
6	128	24	1	1	1	1
7	128	128	3	3	1	128
8	32	128	1	1	1	1
9, 12	256	32	1	1	1	1
10, 13	256	256	3	3	1	256
11, 14	32	256	1	1	1	1
15	256	32	1	1	1	1
16	256	256	3	3	2	256
17	64	256	1	1	1	1
18, 21, 24	512	64	1	1	1	1
19, 22, 25	512	512	3	3	1	512
20, 23, 26	64	512	1	1	1	1
27	512	64	1	1	1	1
28	512	512	3	3	1	512
29	96	512	1	1	1	1
30, 33	512	96	1	1	1	1
31, 34	512	512	3	3	1	512
32, 35	96	512	1	1	1	1
36	512	96	1	1	1	1
37	512	512	3	3	2	512
38	160	512	1	1	1	1
39, 42	1024	160	1	1	1	1
40, 43	1024	1024	3	3	1	1024
41, 44	160	1024	1	1	1	1
45	1024	160	1	1	1	1
46	1024	1024	3	3	1	1024
47	320	1024	1	1	1	1
48	1280	320	1	1	1	1

Table B.3: The Structure of MobileNet-v2-34 (The input size is $H = 54$ and $W = 122$. The proposed real-time DiffFrame convolution with temporal modeling algorithm is applied to layer 3 and the following layers. Metadata need to be generated for layer 3, 12, 13, 27, and 28. The last layer, which is a fully connected layer, is not included in our evaluation.)

Layer IDX	OCHN	ICHN	FH	FW	Stride	Group
1	32	3	3	3	2	1
2	64	32	3	3	2	1
3	64	64	3	3	1	64
4	16	64	1	1	1	1
5	64	16	1	1	1	1
6	64	64	3	3	1	64
7	16	64	1	1	1	1
8	128	16	1	1	1	1
9	128	128	3	3	1	128
10	16	128	1	1	1	1
11	128	16	1	1	1	1
12	128	128	3	3	2	128
13	32	128	1	1	1	1
14, 17	256	32	1	1	1	1
15, 18	256	256	3	3	1	256
16, 19	32	256	1	1	1	1
20	256	32	1	1	1	1
21	256	256	3	3	1	256
22	48	256	1	1	1	1
23	256	48	1	1	1	1
24	256	256	3	3	1	256
25	48	256	1	1	1	1
26	256	48	1	1	1	1
27	256	256	3	3	2	256
28	80	256	1	1	1	1
29	512	80	1	1	1	1
30	512	512	3	3	1	512
31	80	512	1	1	1	1
32	512	80	1	1	1	1
33	512	512	3	3	1	512
34	160	512	1	1	1	1
35	1280	160	1	1	1	1

Appendix C

Dataflows

We present all the dataflows (besides OS-LWS-a, OS-LWS-b dataflows shown in Chapter 2) used in the design space of System 2 in Chapter 2.

```
Level 2
1.  For each OCHN/ochn_tile:
2.    For each OH/oh_tile:
3.      For each OW/ow_tile:
4.        For each ICHN/ichn_tile:
Level 1
5.          ia_l2_buf.read()
6.          ia_l1_buf.write()
7.          For each ochn_tile/ochn_parallel:
8.            For each FH:
9.              For each FW:
10.               For each ichn_tile/ichn_parallel:
11.                 weight_l2_buf.read()
12.                 weight_l0_reg.wr_rd()
13.                 For each OH:
14.                   For each OW:
15.                     ia_l1_buf.read()
16.                     ia_l0_reg.wr_rd()
17.                     psum_l1_ping_buf.read()
18.                     Parallel_for ichn_parallel:
19.                     Parallel_for ochn_parallel:
20.                       MAC & psum_l0_reg.accum()
21.                     psum_l1_ping_buf.write()
Level 2
22.       psum_l1_pong_buf.read()
23.       psum_l2_buf.write()
```

Figure C-1: WS-a dataflow (inner-level). No weight L1 buffer is needed.

```
Level 2
For each OCHN/ochn_tile:
  weight_l2_buf.read()
  weight_l1_buf.write()
  For each OH/oh_tile:
    For each OW/ow_tile:
      For each ICHN/ichn_tile:
Level 1
        ia_l2_buf.read()
        ia_l1_buf.write()
        For each ochn_tile/ochn_parallel:
          For each FH:
            For each FW:
              For each ichn_tile/ichn_parallel:
                weight_l2_buf.read()
                weight_l0_reg.wr_rd()
                For each OH:
                  For each OW:
                    ia_l1_buf.read()
                    ia_l0_reg.wr_rd()
                    psum_l1_ping_buf.read()
                    Parallel_for ichn_parallel:
                    Parallel_for ochn_parallel:
                      MAC & psum_l0_reg.accum()
                    psum_l1_ping_buf.write()
Level 2
      psum_l1_pong_buf.read()
      psum_l2_buf.write()
```

Figure C-2: WS-b dataflow (inner-level).

Level 2

```
For each OCHN/ochn_tile:
  For each OH/oh_tile:
    For each OW/ow_tile:
      For each ICHN/ichn_tile:
```

Level 1

```
        ia_l2_buf.read()
        ia_l1_buf.write()
        For each ochn_tile/ochn_parallel:
          weight_l2_buf.read()
          weight_l1_buf.write()
          For each oh_tile:
            For each ow_tile:
              psum_l1_ping_buf.read()
              For each FH:
                For each FW:
                  For each ichn_tile/ichn_parallel:
                    ia_l1_buf.read()
                    ia_l0_reg.wr_rd()
                    weight_l1_buf.read()
                    weight_l0_reg.wr_rd()
                    Parallel_for ichn_parallel:
                    Parallel_for ochn_parallel:
                      MAC & psum_l0_reg.accum()
              psum_l1_ping_buf.write()
```

Level 2

```
      psum_l1_pong_buf.read()
      psum_l2_buf.write()
```

Figure C-3: OS-a dataflow (inner-level).

```
Level 2
For each OCHN/ochn_tile:
  weight_l2_buf.read()
  weight_l1_buf.write()
  For each OH/oh_tile:
    For each OW/ow_tile:
      For each ICHN/ichn_tile:
Level 1
        ia_l2_buf.read()
        ia_l1_buf.write()
        For each ochn_tile/ochn_parallel:
          For each oh_tile:
            For each ow_tile:
              psum_l1_ping_buf.read()
              For each FH:
                For each FW:
                  For each ichn_tile/ichn_parallel:
                    ia_l1_buf.read()
                    ia_l0_reg.wr_rd()
                    weight_l1_buf.read()
                    weight_l0_reg.wr_rd()
                    Parallel_for ichn_parallel:
                    Parallel_for ochn_parallel:
                      MAC & psum_l0_reg.accum()
              psum_l1_ping_buf.write()
Level 2
    psum_l1_pong_buf.read()
    psum_l2_buf.write()
```

Figure C-4: OS-b dataflow (inner-level).

```
Level 2
1.  For each OCHN/ochn_tile:
2.   For each OH/oh_tile:
3.    For each OW/ow_tile:
4.     For each ICHN/ichn_tile:
Level 1
5.      ia_l2_buf.read()
6.      ia_l1_buf.write()
7.      For each ochn_tile/ochn_parallel:
8.       For each FH:
9.        For each FW:
10.        For each ichn_tile/ichn0:
11.         weight_l2_buf.read()
12.         weight_l0_collect.write()
13.         For each OH:
14.          For each OW:
15.           psum_l1_ping_buf.read()
16.           For each ichn0/ichn_parallel:
17.            weight_l0_collect.read()
18.            ia_l1_buf.read()
19.            ia_l0_reg.wr_rd()
20.            Parallel_for ichn_parallel:
21.            Parallel_for ochn_parallel:
22.             MAC & psum_l0_reg.accum()
23.           psum_l1_ping_buf.write()
Level 2
24.    psum_l1_pong_buf.read()
25.    psum_l2_buf.write()
```

Figure C-5: WS-LOS-a dataflow (inner-level). No weight L1 buffer is needed.

```
Level 2
For each OCHN/ochn_tile:
  weight_l2_buf.read()
  weight_l1_buf.write()
  For each OH/oh_tile:
    For each OW/ow_tile:
      For each ICHN/ichn_tile:
Level 1
        ia_l2_buf.read()
        ia_l1_buf.write()
        For each ochn_tile/ochn_parallel:
          For each FH:
            For each FW:
              For each ichn_tile/ichn0:
                weight_l1_buf.read()
                weight_l0_collector.write()
                For each OH:
                  For each OW:
                    psum_l1_ping_buf.read()
                    For each ichn0/ichn_parallel:
                      weight_l0_collector.read()
                      ia_l1_buf.read()
                      ia_l0_reg.wr_rd()
                      Parallel_for ichn_parallel:
                      Parallel_for ochn_parallel:
                        MAC & psum_l0_reg.accum()
                    psum_l1_ping_buf.write()
Level 2
      psum_l1_pong_buf.read()
      psum_l2_buf.write()
```

Figure C-6: WS-LOS-b dataflow (inner-level).

Bibliography

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News*, 44(3):1–13, 2016.

[2] Brett W Bader and Tamara G Kolda. Efficient matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2008.

[3] Mariusz Bojarski, Philip Yeres, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Lawrence Jackel, and Urs Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. *arXiv preprint arXiv:1704.07911*, 2017.

[4] Anantha P Chandrakasan and Robert W Brodersen. Minimizing power consumption in digital CMOS circuits. *Proceedings of the IEEE*, 83(4):498–523, 1995.

[5] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[6] 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*, 52(1):127–138, January 2017.

[7] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne 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*, 9(2):292–308, 2019.

[8] Inc. Cloudera. An introduction to video understanding: Capabilities and applications, 2021.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255. IEEE, 2009.

[10] C. Duan, A. Gotterba, M. E. Sinangil, and A. P. Chandrakasan. Reconfigurable, conditional pre-charge SRAM: Lowering read power by leveraging data statistics. In *Proceedings of IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pages 177–180, November 2016.

[11] Jamil Fayyad, Mohammad A Jaradat, Dominique Gruyer, and Homayoun Najjaran. Deep learning sensor fusion for autonomous vehicle perception and localization: A review. *Sensors*, 20(15):4220, 2020.

[12] Luciano Floridi and Massimo Chiriatti. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.

[13] Paula Fraga-Lamas, Lucía Ramos, Víctor Mondéjar-Guerra, and Tiago M Fernández-Caramés. A review on IoT deep learning UAV systems for autonomous obstacle detection and collision avoidance. *Remote Sensing*, 11(18):2144, 2019.

[14] William J Gallagher, Eric Chien, Tien-Wei Chiang, Jian-Cheng Huang, Meng-Chun Shih, CY Wang, Christine Bair, George Lee, Yi-Chun Shih, Chia-Fu Lee, et al. Recent progress and next directions for embedded MRAM technology. In *IEEE Symposium on VLSI Circuits*, pages T190–T191, 2019.

[15] J. S. P Giraldo, Steven Lauwereins, Komail Badami, Hugo Van Hamme, and Marian Verhelst. 18uW SoC for near-microphone keyword spotting and speaker verification. In *IEEE Symposium on VLSI Circuits*, pages C52–C53, 2019.

[16] J. S. P. Giraldo and Marian Verhelst. Hardware acceleration for embedded keyword spotting: tutorial and survey. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(6), oct 2021.

[17] JSP Giraldo and Marian Verhelst. Laika: A 5uW programmable LSTM accelerator for always-on keyword spotting in 65nm CMOS. In *Proceedings of IEEE European Solid State Circuits Conference (ESSCIRC)*, pages 166–169, 2018.

[18] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence Beyond the Edge: Inference on intermittent embedded systems. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–213. ACM, 2019.

[19] Benjamin Graham and Laurens Van der Maaten. Submanifold sparse convolutional networks. *arXiv preprint arXiv:1706.01307*, 2017.

[20] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv:1604.03168 [cs]*, April 2016.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W Fletcher. Morph: Flexible acceleration for 3D CNN-based video understanding. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 933–946, 2018.

[23] Mark Horowitz. Computing's energy problem (and what we can do about it). In *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, 2014.

[24] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[25] Vikram Jain, Sebastian Giraldo, Jaro De Roose, Linyan Mei, Bert Boons, and Marian Verhelst. TinyVers: A tiny versatile system-on-chip with state-retentive eMRAM for ML inference at the extreme edge. *IEEE Journal of Solid-State Circuits*, 2023.

[26] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. A survey of deep learning-based object detection. *IEEE Access*, 7:128837–128868, 2019.

[27] Licheng Jiao, Ruohan Zhang, Fang Liu, Shuyuan Yang, Biao Hou, Lingling Li, and Xu Tang. New generation deep learning for video object detection: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 33(8):3195–3215, 2022.

[28] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[29] Nitin Kasturi. Power reducing algorithms in FIR filters. Master's thesis, Massachusetts Institute of Technology, 1997.

[30] Ben Keller, Rangharajan Venkatesan, Steve Dai, Stephen G. Tell, Brian Zimmer, Charbel Sakr, William J. Dally, C. Thomas Gray, and Brucek Khailany. A 95.6-TOPS/W deep learning inference accelerator with per-vector scaled 4-bit quantization in 5 nm. *IEEE Journal of Solid-State Circuits*, 58(4):1129–1141, 2023.

[31] Brucek Khailany, Rangharajan Venkatesan, Jason Clemons, Joel S Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, et al. A modular digital vlsi flow for high-productivity soc design. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[32] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv:1511.06530 [cs]*, November 2015.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 2012.

[34] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2018.

[35] Ji Lin, Chuang Gan, and Song Han. TSM: Temporal shift module for efficient video understanding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7083–7093, 2019.

[36] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. PointAcc: Efficient point cloud accelerator. In *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–461, 2021.

[37] Huiyu Mo, Wenping Zhu, Wenjing Hu, Guangbin Wang, Qiang Li, Ang Li, Shouyi Yin, Shaojun Wei, and Leibo Liu. A 28nm 12.1 TOPS/W dual-mode CNN processor using effective-weight-based convolution and error-compensation-based prediction. In *IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 146–148, 2021.

[38] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. 14.5 Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 246–247, 2017.

[39] OpenAI. GPT-4 technical report, 2023.

[40] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[41] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 27–40, 2017.

[42] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel 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*, pages 137–151, 2019.

[43] Michael Price, James Glass, and Anantha P Chandrakasan. 14.4 A scalable speech recognizer with deep-neural-network acoustic models and voice-activated power gating. In *IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers*, pages 244–245, 2017.

[44] Davide Rossi, Francesco Conti, Manuel Eggiman, Alfio Di Mauro, Giuseppe Tagliavini, Stefan Mach, Marco Guermandi, Antonio Pullini, Igor Loi, Jie Chen, et al. Vega: A ten-core SoC for IoT endnodes with DNN acceleration and cognitive wake-up from MRAM-based state-retentive sleep mode. *IEEE Journal of Solid-State Circuits*, 57(1):127–139, 2021.

[45] T Sainath and Carolina Parada. Convolutional neural networks for small-footprint keyword spotting. In *Proceedings of Interspeech*, 2015.

[46] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, 2018.

[47] Mohit Shah, Jingcheng Wang, David Blaauw, Dennis Sylvester, Hun-Seok Kim, and Chaitali Chakrabarti. A fixed-point neural network for keyword detection on resource constrained hardware. In *IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2015.

[48] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–27, 2019.

[49] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 188–200. ACM, 2004.

[50] Soumya Sudhakar, Vivienne Sze, and Sertac Karaman. Data centers on wheels: Emissions from computing onboard autonomous vehicles. *IEEE Micro*, 43(1):29–39, 2022.

[51] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

[52] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 2020.

[53] Kodai Ueyoshi, Kota Ando, Kazutoshi Hirose, Shinya Takamaeda-Yamazaki, Mototsugu Hamada, Tadahiro Kuroda, and Masato Motomura. QUEST: Multi-purpose log-quantized dnn inference engine stacked on 96-MB 3-D SRAM using inductive coupling technology in 40-nm CMOS. *IEEE Journal of Solid-State Circuits*, 54(1):186–196, 2018.

[54] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. MAGNet: A modular accelerator generator for neural networks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[55] Miaorong Wang. Algorithms and low power hardware for keyword spotting. Master's thesis, Massachusetts Institute of Technology, 2018.

[56] Miaorong Wang and Anantha P Chandrakasan. Flexible low power CNN accelerator for edge computing with weight tuning. In *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pages 209–212. IEEE, 2019.

[57] Miaorong Wang, Yujun Lin, Zhekai Zhang, Ji Lin, Song Han, and Anantha P. Chandrakasan. Videotime^3: A 40-uJ/frame 38 FPS video understanding accelerator with real-time diffframe temporal redundancy reduction and temporal modeling. *IEEE Solid-State Circuits Letters*, 6:169–172, 2023.

[58] P. Warden. Speech command: A public dataset for single-word speech recognition., 2017.

[59] Paul N Whatmough, Sae Kyu Lee, David Brooks, and Gu-Yeon Wei. DNN engine: A 28-nm timing-error tolerant sparse deep neural network processor for iot applications. *IEEE Journal of Solid-State Circuits*, 53(9):2722–2731, 2018.

[60] Di Wu, Nabin Sharma, and Michael Blumenstein. Recent advances in video-based human action recognition using deep learning: A review. In *International Joint Conference on Neural Networks (IJCNN)*, pages 2865–2872, 2017.

[61] Xiaowei Xu, Xinyi Zhang, Bei Yu, Xiaobo Sharon Hu, Christopher Rowen, Jingtong Hu, and Yiyu Shi. DAC-SDC low power object detection challenge for UAV applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(2):392–403, 2019.

[62] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, et al. DNN dataflow choice is overrated. *arXiv preprint arXiv:1809.04070*, 6:5, 2018.

[63] Shouyi Yin, Peng Ouyang, Shixuan Zheng, Dandan Song, Xiudong Li, Leibo Liu, and Shaojun Wei. A 141 uW, 2.46 pJ/neuron binarized convolutional neural network based self-learning speech recognition processor in 28nm CMOS. In *IEEE Symposium on VLSI Circuits*, pages 139–140, 2018.

[64] Zhe Yuan, Yixiong Yang, Jinshan Yue, Ruoyang Liu, Xiaoyu Feng, Zhiting Lin, Xiulong Wu, Xueqing Li, Huazhong Yang, and Yongpan Liu. A 65nm 24.7 μJ/frame 12.3 mW activation-similarity-aware convolutional neural network video processor using hybrid precision, inter-frame data reuse and mixed-bit-width difference-frame data CODEC. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 232–234. IEEE, 2020.

[65] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[66] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello Edge: Keyword spotting on microcontrollers. *arXiv:1711.07128 [cs, eess]*, November 2017.

[67] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.