Efficient, Reconfigurable, and QoS-Aware Systems for Deep Neural Networks

by

Xiaowei Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Xiaowei Wang

xiaoweiw@umich.edu

ORCID iD: 0000-0002-5883-7327

DEDICATION

To my family.

ACKNOWLEDGMENTS

It would not be possible for me to finish this dissertation without the help of many people. First and foremost, I would like to express my sincere gratitude to my Ph.D. advisor and the dissertation committee chair, Reetuparna Das. She led me into the world of conducting research projects systematically, provided a tremendous amount of guidance to me on research with her insights in the field, and encouraged me to pursue intriguing and impactful research directions. I would also like to thank my dissertation committee members, Scott Mahlke, Lingjia Tang, and Zhengya Zhang. They provided many useful suggestions to improve the quality of dissertation. I would like to thank my collaborators for their inputs and brainstorming sessions that help to shape and improve the works in the dissertation: Jiecao Yu, Vidushi Goyal, Charles Eckert, Arun Subramaniyan, Jingcheng Wang, Valeria Bertacco, David Blaauw and Dennis Sylvester from University of Michigan; Andrew Boutros, Eriko Nurvitadhi, Charles Augustine and Ravi Iyer from Intel; Li Zhao from Alibaba. I would like to thank my fellow research group members and labmates for inspiring discussions and fun times. Thank you Daichi Fujiki, Arun Subramaniyan, Vidushi Goyal, Charles Eckert, Jack Wadden, Subarno Banerjee, Harisankar Sadasivan, Tim Dunn, Kush Goliya, Yufeng Gu and Alireza Khadem. I would also like to thank my fellow graduate student friends and the staff members at CSE for the conversations and help. I would like to thank the first responders who made it possible to conduct research activities throughout the pandemic times.

Finally, I would like to thank all my family members for their support. I would like to express my special thanks to my beloved wife, Yuqi Gu, who accompanied me, shared my happiness, and constantly gave me advice and encouragement throughout my Ph.D. journey.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Deep Neural Networks (DNN) are an important machine learning application which has high compute and memory bandwidth requirements to the underlying computer systems. Prior works have proposed domain specific accelerators for DNNs, and in-memory computing architectures are especially promising as they can provide high memory bandwidth and computing throughput at the same time.

The first part of the dissertation targets improving the efficiency of a recent SRAM-based in-memory DNN accelerator on sparse and low-bitwidth DNN models. In this system, the last level cache of a CPU is repurposed as a highly parallel bit-serial vector processor, achieved by the circuit modification to the SRAM array peripherals and a transposed data mapping. We propose two hardware/software co-design methods with customized model pruning algorithms to fully utilize the sparsity. A specialized bit-serial algorithm is developed for the operations on low bitwidth data.

While the above in-cache computing system is highly efficient, its hardware architecture lacks the flexibility to be optimally reconfigured for different DNN models. The second part of the dissertation proposes a reconfigurable in-SRAM computing DNN accelerator based on block RAMs (BRAM) on FPGAs. We propose circuit changes to the BRAM to enable bit-serial in-memory computing, which turns BRAMs as both bit-serial vector units and data storage. Building on the compute-capable BRAMs, we further propose customized accelerator instances for different DNN models, which outperforms a state-of-the-art DNN accelerator on FPGA.

DNN workloads can also be run on general purpose CPUs in datacenters. Cache compression is a technique to reduce the cache miss rate on CPU, which benefits DNNs as well as many other applications. In the third part of the dissertation, we present a novel method to compress cache data with efficient in-SRAM data comparison. Further, as datacenters frequently collocate multiple workloads to increase server utilization, the Quality-of-Service (QoS) of DNN workloads, such as latency, can be affected. The final part of the dissertation proposes a systematic approach to achieve the QoS of DNNs under collocation, with resource partition to reduce interference, and a proposed latency prediction model to choose the partition that satisfies the QoS requirement.

CHAPTER 1

Introduction

In the last decade, the Deep Neural Networks (DNN) has been pervasively used as a machine learning mechanism to solve many real-world problems that are traditionally difficult tasks for computers. The application domains span from computer vision tasks such as image classification (He et al., 2016) and object detection (Liu et al., 2016), to language processing tasks such as sentimental analysis (Devlin et al., 2018), machine translation (Vaswani et al., 2017), and speech recognition (Rao et al., 2017).

Along with the development of more and more DNN models is the increased attention on how to perform the necessary computation for the DNNs efficiently. DNN models can contain millions to billions of parameters and the computation is usually in vectors or matrices so the total number of arithmetic operations is huge. Fortunately, as there is a high level of parallelism in the operations, vector processor architectures can speed up the DNN computation. Further, as there are several key kernel operations that takes up most of the computation time such as convolution and matrix-vector multiplication, many works have analyzed the common computation and data movement pattern of such operations and built domain specific accelerators for DNNs, implementing the kernels in custom hardware (Chen et al., 2016, 2014). The TPU (Jouppi et al., 2017) is an example DNN accelerator, where the matrix-matrix multiplication kernel is implemented with a non-traditional hardware architecture of systolic arrays that has a high effective computing throughput.

An important and rapidly evolving technique for DNN hardware acceleration is in-memory computing. The main idea of in-memory computing is to perform computation near or within the memory structure. In-memory computing essentially improves the data movement bandwidth between the computing units and the data storage. As the DNN models usually have a large model size (more than tens of megabytes), the in-memory computing is helpful in alleviating the data movement bottlenecks. In addition, the large number of computing units can also speed up the parallel computing instructions of DNN. One recent work, Neural Cache (Eckert et al., 2018), utilizes the last-level cache of CPUs for SRAM-based in-memory computing to accelerate DNN. Their approach achieves a high overall computing throughput by enabling each bitline in the SRAM array as an individual computing unit. In first part of the dissertation, we aim at improve the above in-cache DNN accelerator with two model compression techniques, weight pruning and ultra-low precision quantization. We propose software/hardware co-designing methods to achieve this goal.

DNNs can also be accelerated on FPGAs, which offer a balance of adequate efficiency and high reconfigurability and therefore can adapt to new computation needs quickly. We observe that the in-memory computing opportunities on FPGAs have not been explored despite the large on-chip SRAM memory available. In the second part of the dissertation, we propose methods to adapt the on-chip memory of FPGA for in-SRAM computing, resulting in a boost of total computing throughput. We further show how this enhanced FPGA can improve RNN latency against Brainwave, a popular DNN accelerator on FPGA.

Multi-core CPUs can also serve as the platform for DNN computation, as this is the cost-sensitive option or where machine homogeneity is preferred in some datacenters. In the third part of the dissertation, we explore a specific type of microarchitectural optimization, cache compression, which helps improve the performance of cache-sensitive workloads. Specifically, we propose an efficient cache deduplication method based on in-memory data comparison. In datacenters, there are cases where the DNN workloads need to be collocated with other workloads on the same server during periods with high service request rates. Such collocation can negatively affect the Quality-of-Service (QoS) for the DNN workloads. In the final part of the dissertation, we propose a method to protect the QoS of DNNs under collocation based on workload latency prediction and resource partition.

In the remaining part of this chapter, we will provide a more detailed introduction on each of the research questions and the proposed solution.

1.1 Improving Compute-in-Cache DNN Accelerator with Model Compression

Neural Cache (Eckert et al., 2018), is one example of demonstrating the in-memory processing techniques for DNN inference. Neural Cache utilizes the SRAM arrays of last level cache of general purpose CPU and turn them into massively parallel computing units. The key idea is to map the data in a transposed way in the SRAM arrays, and then performing bit-serial operations on all the bit-lines in the SRAM arrays in parallel. The bit-serial operations are implemented with in-array dual-row simultaneous sensing and extra logic at the bit-line peripherals. Data on all the bit-lines can be processed in parallel, so the entire cache can be turned into a vector processor. In Convolutional Neural Networks (CNN), there are a massive number of Multiply-Accumulate (MAC) operations that can be parallelized, and they are mapped to the more than one million bit-lines on a 35 MB last-level cache. Thus the CNN inference is accelerated by the high throughput and the convolution specific data reuse strategy. Compared to other custom accelerators, the above solution is based on cache in general purpose processors, and the cache can still improve the performance of many other applications when not functioning as a CNN accelerator.

However, previous works have not explored the opportunity to leverage two important model compression techniques in CNN. First, the weight pruning technique for DNN can reduce the number of weights and thus the computation involving the pruned weights can be left out (Han et al., 2015; Yu et al., 2017). But the existing Neural Cache architecture is unable to skip such computation. This is because with the individually pruned weights scattered in location, the vector processor architecture cannot skip computation for a pruned weight when there are unpruned weights being processed at the same time. Second, low-precision values in DNN can also reduce computation time (Rastegari et al., 2016; Zhou et al., 2016; Zhu et al., 2016). Although the bit-serial algorithms can naturally take advantage of the low-precision formats, it is observed that bit-serial multiply-accumulate is inefficient when weights are ultra-low precision such as ternary or binary.

In the first part of the dissertation (Chapter 3), we present the solutions for accelerating DNNs with sparsity and low-precision values with in-SRAM bit-serial computation. For leveraging the data sparsity, the dissertation proposes two novel hardware/software co-design techniques. The first technique proposed is to create dense computation amenable for vector processors, by pruning the model by filter channels and then coalescing the non-zero filter channels. All the weights in a filter channel are mapped to one bit-line, so an entirely pruned filter channel may spare one processing unit. As coalescing the channels may complicate the input loading process, the dissertation proposes a second technique, which custom prunes the model so that several pruned filters can overlap with each other without location-conflicted non-zero values. Overall the above techniques skip the computation as well as data movement on the pruned weights, so as to improve the performance. For low-precision values, the dissertation redesign the process of multiply-accumulate for ternary/binary weights to combine the multiplication and accumulation in fewer cycles using logical operations.

The proposed sparsity-aware architecture achieves a 17.7×/3.7× speedup over server class CPU/GPU, and a 1.6× speedup compared to the relevant in-cache accelerator, with 2% area overhead each processor die, and no loss on top-1 accuracy for AlexNet. With a relaxed accuracy limit, the tunable architecture with data sparsity and low-precision values achieves higher speedups. This piece of work is published in Wang et al. (2019b).

1.2 Reconfigurable In-Memory Computing for DNN Acceleration

Reconfigurability of hardware is a desirable feature that amortizes the fabrication cost as reconfigurable hardware can usually be reused for a wide variety of applications efficiently. The in-cache DNN accelerator described in the previous chapter does not satisfy the reconfigurability as the hardware architecture of the cache is fixed.

In searching for the potential possibility of a reconfigurable in-memory DNN accelerator, a promising choice of the base hardware platform is FPGA. FPGAs have been widely deployed as domain-specific computation device (AWS; Caulfield et al., 2016), and they are known for their flexibility and reconfigurability.

The basic structure of FPGA hardware consists of several different types of building blocks serving different functions, including lookup tables (LUT) for building customized logic, block RAMs (BRAM) to store data, and digital logic processing (DSP) blocks for efficient arithmetic operations. We observe that state-of-the-art FPGA architectures have high capacity on-chip BRAMs, which, although are not conventionally used to form computation logic, provides opportunities for building compute-capable memory on FPGAs. For example, the largest monolithic Intel Stratix 10 device has 11,721 BRAMs that collectively provide 229 Mb of distributed on-chip memory (Int, 2019). Future generations of FPGA devices are also expected to have higher density of on-chip BRAMs to satisfy the ever increasing demands of memory-intensive datacenter workloads (Putnam et al., 2014).

If we apply the similar idea of bit-serial in-SRAM computing to the BRAM blocks, they can be turned into a total of about 1.5 million bit-serial compute units. Such compute-capable BRAM blocks would become an important computing resource in addition to the existing LUT and DSP blocks. In FPGAs, the gains of compute-capable BRAMs are compelling for these main reasons: (1) The large number of distinct memory blocks in an FPGA enables a significantly higher degree of parallelism. The BRAMs in the largest monolithic

Intel Stratix 10 FPGA can be re-purposed to over 1.5 million bit-serial SIMD compute lanes when enhanced with compute-capable peripherals. When not computing, BRAMs can still function as normal memory units. (2) The in-fabric distributed nature of the FPGA on-chip memory enables tighter integration of the compute-capable memories with the rest of the design logic. For operations that cannot be computed in-memory, the data from a compute-capable last-level cache in a CPU would still need to go through the different levels of the memory hierarchy and the fixed processor pipeline to reach the compute units, which is not the case for compute-capable FPGA BRAMs. (3) Performing the compute in BRAMs eliminates the data movement between memory and compute units which reduces the competition over the fabric's precious routing resources. This can potentially alleviate congestion and increase the routability of FPGA designs.

In the second part of the dissertation (Chapter 4), we demonstrate how to make modifications to the BRAMs to support the in-memory computing, tackling the challenge that the interface of BRAM to FPGA is different from the SRAM array's interface to the CPU cache. We present designs for the block-floating-point arithmetic with compute-capable BRAMs, dot-product acceleration modules built from an integration of compute-capable BRAMs and LUTs. We also show how to perform architectural customization when the DNN model varies.

In contrast to prior in-SRAM computing architectures which use fixed hardware, the proposed system fully utilizes the reconfigurability of FPGAs. The enhanced FPGA architecture can be reconfigured optimally for different DNN models, with the compute-capable BRAM being flexibly used as computing units or data storage. Further, the control structures are reconfigurable to support both fixed-point and block-floating-point numerical precision.

Our evaluation shows that enhancing a modern Stratix 10 FPGA architecture with compute-capable BRAMs can result in 1.6× and 2.3× increase in the device's peak multiply-accumulate (MAC) throughput for 8-bit integer and block floating-point precision, respectively. These gains are achieved with no change to the modes of operation or routing interface of existing BRAMs and at a minimal cost of 1.8% increase in the FPGA die size. We then evaluate the performance gains of our proposed architectural modification when accelerating a variety of real-time DNN workloads. Compared to Brainwave (Fowers et al., 2018), a state-of-the-art DNN accelerator on FPGA, the proposed architecture achieves an average 1.25× and 3× speedup for 8-bit integer and block floating-point numbers. In addition, the proposed accelerator can achieve an order of magnitude higher performance compared to a same-generation GPU. This piece of work is published in Wang et al. (2021b).

1.3 Efficient Cache Data Compression

During DNN computation, there can be a lot of duplicated values either due to the large number of zeros after model compression or because of quantization to a small number of bits leading to a small set of possible data values. In either case, an efficient way for data compression is crucial to improve the performance of DNNs running on CPUs.

Cache compression is an architectural optimization topic that has been extensively studied. In general, cache compression leverages the data redundancy within the cache, and stores the data in relatively fewer bits to increase the effective cache capacity. More cache lines can be cached and kept on chip with compression. The cache hit rate and the overall program performance, especially for the programs with a large working set, improve as a result. Cache compression also avoids adding more data arrays to the cache which is more energy-consuming.

Recently, compression methods that leverage the similarities across cache lines have been proposed (Tian et al., 2014; Ghasemazar et al., 2020). It is observed that duplicated or highly similar cache lines exist in the last level cache, leading to compressing opportunities where the duplicated part can be stored with one data copy and multiple pointers for the distinct tags. Such inter-cache-line compression exploits the data redundancies at a wider range across all the cache lines and therefore provide additional opportunity for compression beyond the data similarity within a cache line. However, these works treat the cache data array as a scratchpad and thus indexing the data is complicated. This causes delays in the decompression process, which is on the critical path of the memory read instructions in the case of cache hit. We discover that the above problem can be alleviated by storing the compressed data in segments that are conveniently indexed, with a case study on Intel Xeon last-level cache.

We also observe the cache organization in today's processors for more efficient compression. More specifically, in the last-level cache for a multi-core Xeon E5 processor (Huang et al., 2013), each core is connected to a cache slice. Within each slice, the SRAM arrays are arranged in columns, where each column has 16 8kB SRAM arrays and each column corresponds to one way in the set-associative cache. The 16 arrays in a way column are further partitioned into 4 banks, and the arrays within a bank are connected via an H-tree interconnect. Because the 4 banks within a way column all use separate data buses, we may use separate pointers and thus the deduplication compression happens independently for the 4 quads in a cache line, reaching a balance between high compression ratio and fast decompression.

The cache line similarity-based compression methods require comparison of new incoming cache lines with the existing ones to find similar pairs for compression. In a conventional architecture, it is necessary to read out the data in existing cache lines and transfer them to the comparison logic. We observe that a recent work on in-cache computation, Compute Caches (Aga et al., 2017), provides the ability to do data comparison in the data arrays in last-level cache efficiently. In Compute Caches, each SRAM array is equipped with an additional row decoder to activate two wordlines simultaneously. Then the differential sense-amplifiers at each bitline pair are modified to sense both bitline and bitline-bar separately, effectively calculating the logical AND and NOR of the two activated bitcells on that bitline. Other logical operations, including comparison, can be supported by a few additional logic gates, and making all the sense-amplifiers work concurrently. For instance, to support XOR operation, a NOR gate is added, taking the results of bit-line and bit-line complement (i.e., AND and NOR results) as inputs. To check whether two words are identical, first XOR is performed on the two data words, and then the XOR results go through a wired-NOR logic, to generate a binary matching result.

With the above in-SRAM comparison operation, the in-array data comparison directly works on the data stored at two word-lines in an SRAM array, without the need of transferring data out of the array. Such word-line level comparison can be done at multiple SRAM arrays in parallel. By comparing the cache line data in arrays, the comparison throughput can be improved thanks to the high parallelism. Further, while reading out a massive amount of data for comparison consumes energy in the data interconnects, the in-array comparison eliminates such energy consumption. We apply this idea and propose a cache compression design with low decompression latency (Chapter 5). We evaluate the proposed compression method on the SPEC2006 benchmarks. We achieve a 2.05× compression ratio, and 4.73% of speedup on average, compared to a conventional L3 cache without compression. This piece of work is published in Wang et al. (2021a).

1.4 Satisfying QoS of Collocated DNN Service

Now we will consider how to improve the DNN performance from a system perspective in the context of datacenters. In modern datacenters, it is crucial to provide a consistent Quality-of-Service (QoS) to the clients that meets the QoS targets set by the clients. The common QoS metrics are latency and/or throughput, which depend on the nature of the applications.

DNN inference is an important type of service provided by the datacenter servers recently (Hauswald et al., 2015). As DNN is widely used in artificial intelligence applications, a guaranteed QoS for DNN inference is a highly desirable feature in the datacenters, which further contributes to the guaranteed QoS of end-to-end artificial intelligence applications.

Even though many accelerators or customized hardware have been built for the DNN workloads, the CPU-based servers are still an important type of computing resource for machine learning inference. According to a report on machine learning inference at Facebook's datacenters (Hazelwood et al., 2018), the majority of inference services are run on the single-socket or dual-socket CPU servers, for a wide range of applications such as speech recognition, personalized recommendation, and face recognition. Running the inference on CPUs instead of dedicated accelerators can avoid the overhead of dedicated programming and compilation efforts, and also is favorable for a homogeneous datacenter environment for ease of management. In a datacenter with both CPU and GPU resources, it is more efficient to perform the training tasks on GPUs and assign the inference tasks to CPUs (Hazelwood et al., 2018).

With the DNN inference running on the CPU servers, it is a natural idea to collocate such DNN services with other services on the same multi-core CPU server to increase the overall server resource utilization and improve the request processing throughput. Another scenario where the task collocation would happen is when the request rate is at a high level and the datacenter is designed to avoid over-provisioning. Further, with the widely used software frameworks that can express different machine learning models, multiple machine learning services of different types can be run at the same server node, and thus the concurrent execution of different machine learning applications can also happen.

The workload collocation can have a negative impact on the performance of an individual workload and cause the violation of QoS. This is mainly caused by the contention of resources such as the processor cores, the memory capacity and the memory bandwidth, as well as the interference effects such as the thrashing of the cache contents. Although various previous works have studied the problem of contention caused by collocation of generic workloads on CPU-based servers, none of them directly targets the specific domain of DNN inference services.

In the final part of the dissertation (Chapter 6), we propose a mechanism to protect the QoS of DNN workloads under collocation. We observe that there is determinism in the computation of DNNs and therefore design a latency predictor. At runtime we generate a partition of different types of server resources across the latency-critical DNN workloads and other best-effort workloads, so that the predicted performance of the DNN workload meets the QoS target. When applied, the proposed approach improves the latency of DNN workloads by 85% and 26% when collocated with core and memory-bandwidth thrashing microbenchmarks, and by 11% when collocated with best-effort DNN workloads.

CHAPTER 2

Background and Related Works

This chapter describes the background knowledge and related works for the rest of the dissertation. We begin with a general description of the structure and computation pattern of common DNNs (Section 2.1), followed by an introduction to the in-memory acceleration of DNNs (Section 2.2), including an in-SRAM computing example in detail. Then we discuss two important types of algorithmic simplification opportunities in the DNN computation, data sparsity and quantization, and how the hardware can take advantage of them (Section 2.3). After that, we introduce the FPGA architecture and show how DNNs can be accelerated with FPGAs (Section 2.4). Finally, we provide background regarding topics of DNN performance on CPU, including the cache data compression techniques and how the DNN workloads achieve QoS targets (Section 2.5).

2.1 The Computation Pattern of DNNs

DNN is a general term for one family of machine learning models, which has many layers stacked together, and the output of one layer becomes the input of the next layer. One DNN layer usually includes a linear transformation on the input, and then is followed by a nonlinear function. There are trainable parameters in the DNN layers, which are also called weights. Usually, the linear transformation step involves a large number of weights and dominates the computation in inference.

The common DNN models include Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and Multi-Layer Perceptron (MLP).

Convolutional Neural Network (CNN) is a machine learning method that takes a 3D-array as input, performs convolution and non-linear functions on the array iteratively, and extracts the array information for further tasks such as classification. A typical CNN consists of many convolutional layers, a few pooling and fully connected layers, and non-linear functions between the layers. The overall procedure of a convolutional layer is shown in Figure 2.1. A convolutional layer takes in C channels of 2D activation maps as input, with activation of each channel having H pixels in height and W pixels in width. A batch of in total M 3D-filters are the network parameters of the layer. Each 3D-filter has C channels, each channel with $R \times S$ parameters called weights, where R is the filter height and S is the filter width. At convolution, the C channels of $R \times S$ filters are overlaid on the C channels of inputs. The $R \times S \times C$ input pixels are element-wise multiplied with their corresponding filter weights, and then the products are summed up across all C channels to produce one output pixel as shown in Figure 2.1. The $R \times S$ filter slides over the $H \times W$-sized input activation with stride U, generating $E \times F$ output pixels per 3D filter. Therefore the M filters create a total of $M \times E \times F$ output pixels for one convolutional layer and these pixels become inputs of the following convolutional layer. It has been shown that among the end-to-end CNN computation, most (90%) of the time is spent on convolutional layers (Chen et al., 2016).

Recurrent Neural Networks (RNN), is a type of DNN that is widely used in AI applications including machine translation and speech recognition. Many types of RNNs have been designed, including vanilla RNN, LSTM and GRU, with the main difference in the dependency of vectors and activation function used. RNNs have one shared major component in inference computation - the matrix-vector multiplication (MVM). For example, Figure 2.2 shows the algorithmic structure of a LSTM layer. The four MVM blocks consume the majority of the compute time. Note that there is no data dependency among the four MVM blocks, thus they can be combined into a larger MVM. This property holds for many different

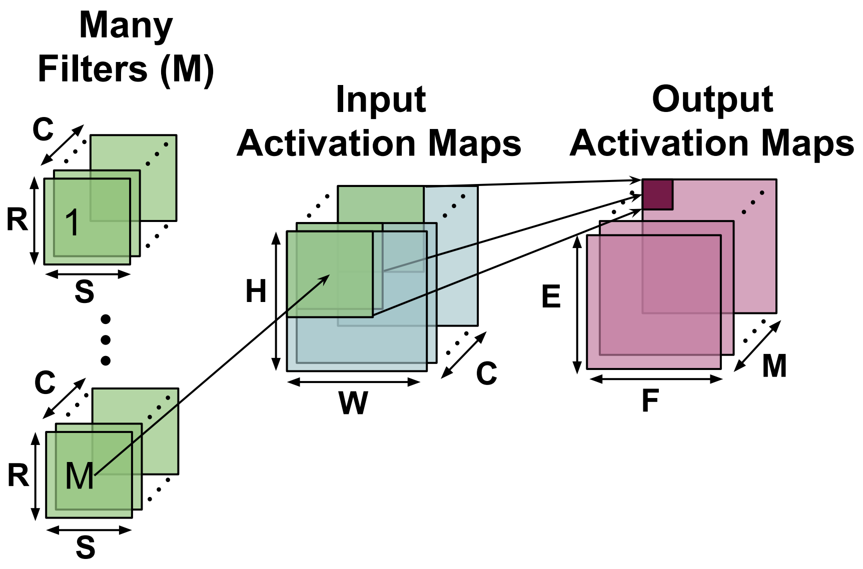


Figure 2.1: Computation of a Convolutional Layer.

types of RNN. In such a combined MVM, the inputs are a vector with C elements and a weight matrix of size $R \times C$. The output is a vector of R elements.

In MLP, the linear transformation is matrix-vector multiplication similar to the RNN. The main difference is that the RNN takes a sequence as an input while MLP has a single input vector.

In a DNN application, there are usually two computation stages, training and inference. The training stage generates the best weights for the given task, and the inference stage provides a prediction for the input with the already trained weights. Training is a one-time cost before the model deployment while inference usually is run many more times than training. In this dissertation, we mostly consider the inference stage as the target application.

2.2 In-Memory Computing for DNN

This section mainly discusses the in-memory computing techniques to accelerate DNNs. In fact, the DNN accelerators is a much wider research area than the in-memory DNN accelerations and there are numerous prior works (Chen et al., 2016, 2014; Du et al., 2015;

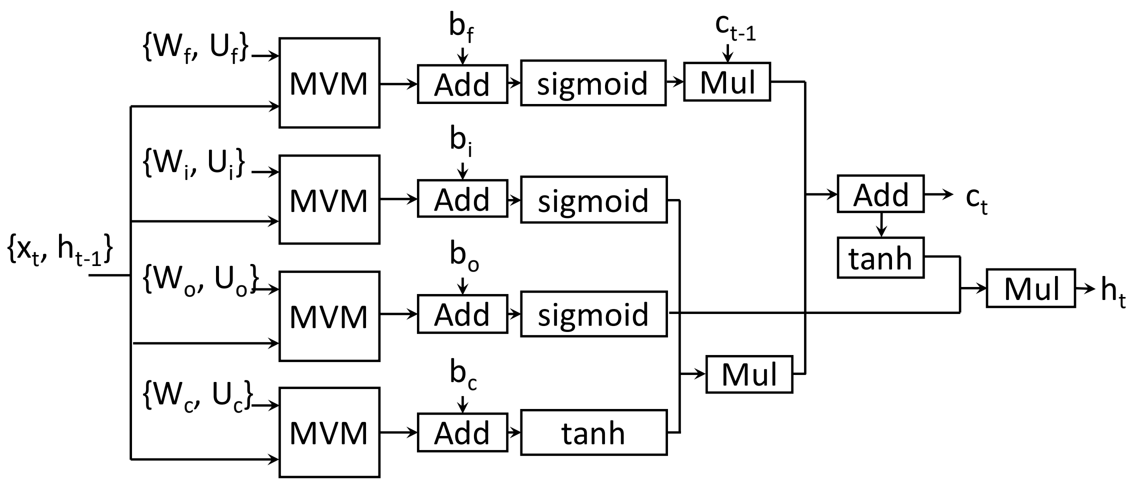


Figure 2.2: Overview of the algorithm for a LSTM layer. The computation bottleneck is in the Matrix Vector Multiplication (MVM).

Deng et al., 2019; Zhang et al., 2018; Hegde et al., 2018; Jouppi et al., 2017). TPU (Jouppi et al., 2017) is a typical DNN accelerator work that proposes a high-throughput systolic matrix multiply unit. Eyeriss (Chen et al., 2016) is another typical work, which explores different data flow options for CNN acceleration and reduces data-movement by maximizing weight reuse. Discussing all the DNN accelerators is beyond the scope of this work and Sze et al. (2020) provides a summary on this topic.

2.2.1 Overview of In-Memory DNN Acceleration

Many recent works have proposed to accelerate DNN applications with in-memory processing (Li et al., 2017; Seshadri et al., 2017; Li et al., 2018; Imani et al., 2019; Eckert et al., 2018; Ankit et al., 2019; Ji et al., 2019; Shafiee et al., 2016; Chi et al., 2016; Song et al., 2017; Kim et al., 2016). In-memory computing is a good fit for accelerating DNNs for multiple reasons. First, DNN applications usually have a high degree of parallelism as the computation involves large matrices and vectors. The parallelism can be well exploited by the vector-processor style architecture of in-memory processing. Second, in the inference stage of DNN, the model parameters are usually fixed and thus can be pinned to the memory, while the inputs are being streamed in. This is especially a desired behavior of in-memory processing as one of the operators (weights) is stationary and only the movement of the inputs needs to be orchestrated.

The in-memory DNN accelerators can utilize a range of different types of memory cells, including DRAM, SRAM, and non-volatile memory such as ReRAM. ReRAM-based in-memory DNN accelerators (Shafiee et al., 2016; Chi et al., 2016) use the analog circuit property of the ReRAM memory cells, where the model weights are programmed as memristance values into the cells. Then the inputs are converted into analog voltage values and applied to the ReRAM cells, leading to a generated current that represents the value of a MAC operation. There are other ReRAM-based in-memory computing architectures that stress flexibility with reconfigurable routing architecture (Ji et al., 2019), and support floating point arithmetic with digital in-memory computation (Imani et al., 2019). Overall, computing-in-ReRAM can achieve high compute density for MAC operations. But ReRAM devices are prone to operational variations which might affect its reliability, and the analog/digital converters also take up extra area and energy.

Many DRAM-based in-memory computing DNN accelerators (Kim et al., 2016; Gao et al., 2017) utilizes 3D-stacked memories, where the logic layer and the memory layers are vertically stacked to enable a high data movement bandwidth between these layers. The DNN computation is mapped to the above architecture with the consideration of reducing the data movements within the logic layer. Another line of work (Li et al., 2018) attaches ALU to each bitline in DRAM array for in-memory processing. An example of in-DRAM computing architecture for DNNs that modifies the DRAM array structure is DRISA (Li et al., 2017). The underlying technique is to activate three rows in a DRAM array and then performing bit-wise logic operation by exploiting the changes in capacitor charges. On the architectural level, DRISA builds a vector processor by activating multiple DRAM arrays simultaneously. Overall DRAM-based in-memory computing is a popular design due to the widespread use of DRAM as the main memory device.

The in-SRAM acceleration of DNNs can be based on either digital or analog approaches.

In the digital approach, elementwise logical operations are performed on the data of two rows in the same SRAM array, by activating two wordlines and sensing the currents on each bitline. We will discuss extensively one example of in-SRAM DNN accelerator in Section 2.2.2. The in-SRAM analog computing approach (Valavi et al., 2018; Srivastava et al., 2018) uses a similar approach as used in ReRAMs, where the weights are stored in memory cells and inputs are sent to all wordlines for MAC operations. Overall, in-SRAM computing has faster speed and lower energy consumption for each operation compared to other memory cell types, while its larger cell size makes the total capacity of SRAM smaller, and thus it is more complicated to handle large datasets.

2.2.2 In-SRAM Bit-Serial Computing with CPU Cache

Neural Cache (Eckert et al., 2018) is an architecture which repurposes the last level cache in general purpose processors to perform massively parallel in-SRAM computing for neural networks. Figure 2.3 shows the overall architecture of Neural Cache. The building block of Neural Cache is bitline computing enabled by SRAM array peripherals. In bitline computing, two wordlines of an SRAM array are activated at the same time and by sensing the shared bitline pairs (existing in Xeon LLC), logical operations (`and` and `nor`) on the cell data of the two wordlines can be performed (Aga et al., 2017; Jeloka et al., 2016).

To perform arithmetic operations such as addition and multiplication, a bit-serial architecture is utilized (Figure 2.3 (c)): data are mapped to a transposed layout where different bitlines hold data from different elements in the operand vector. Each n-bit element is stored across n wordlines, and thus each wordline holds one *bit-slice* from all the vector elements. The bits in each bit-slice are of the same bit position. The arithmetic operation is done bit-slice by bit-slice – for example, to compute $A + B$, bit-slice 0 of vector A and bit-slice 0 of vector B are first activated and added by the peripheral logic (generating sum and carry), then the same operation is done to bit-slice 1, 2, ..., $n - 1$ of the two operand vectors (carry taken from the previous bit-slice step). For n-bit integer addition, bit-serial computation

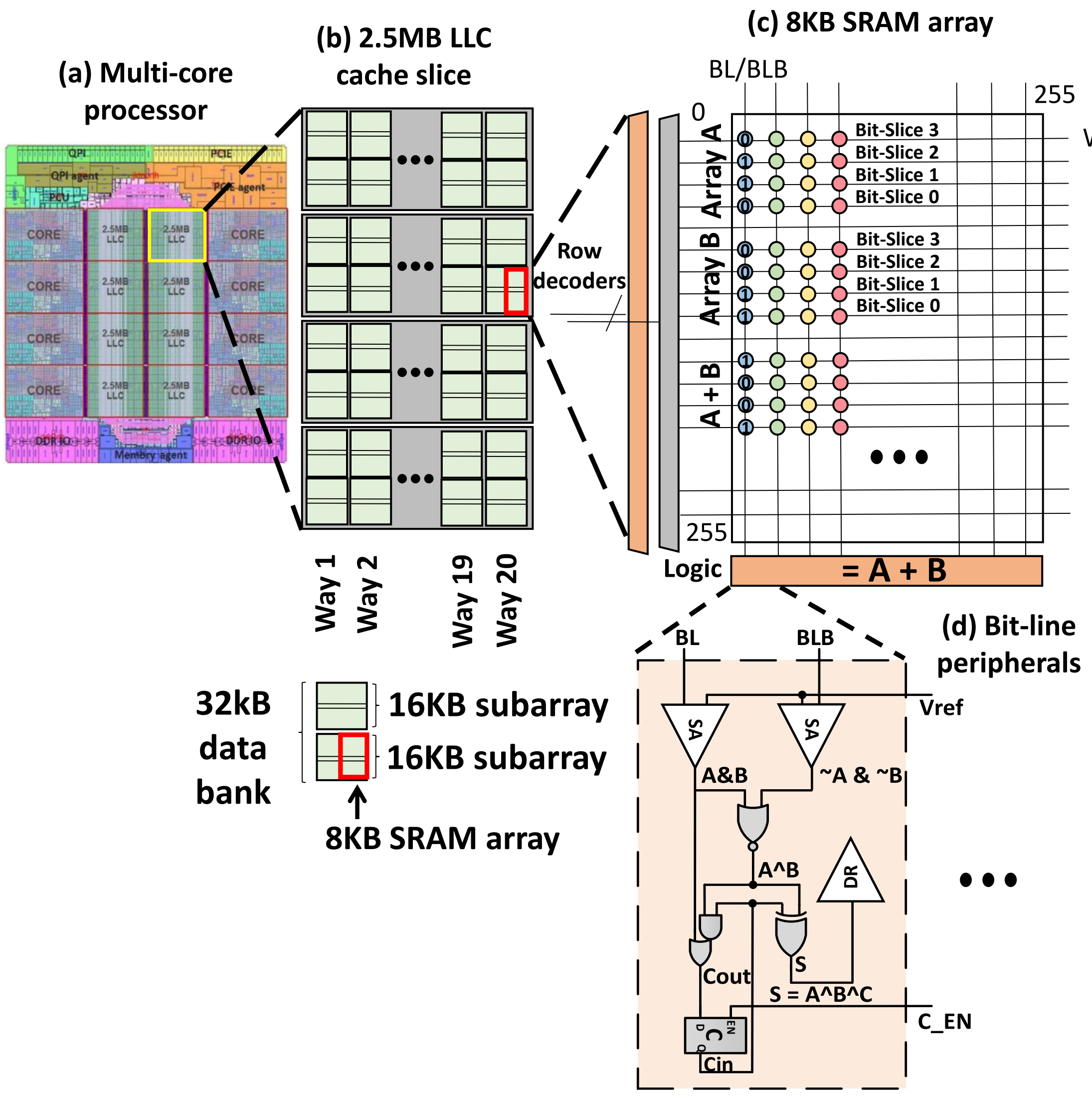
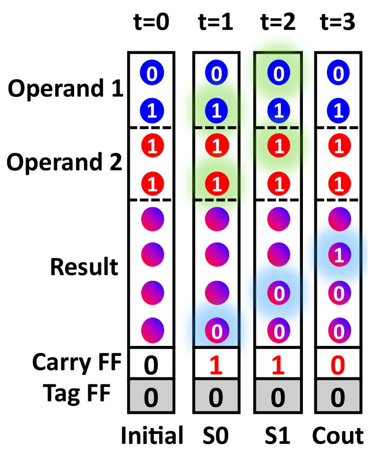


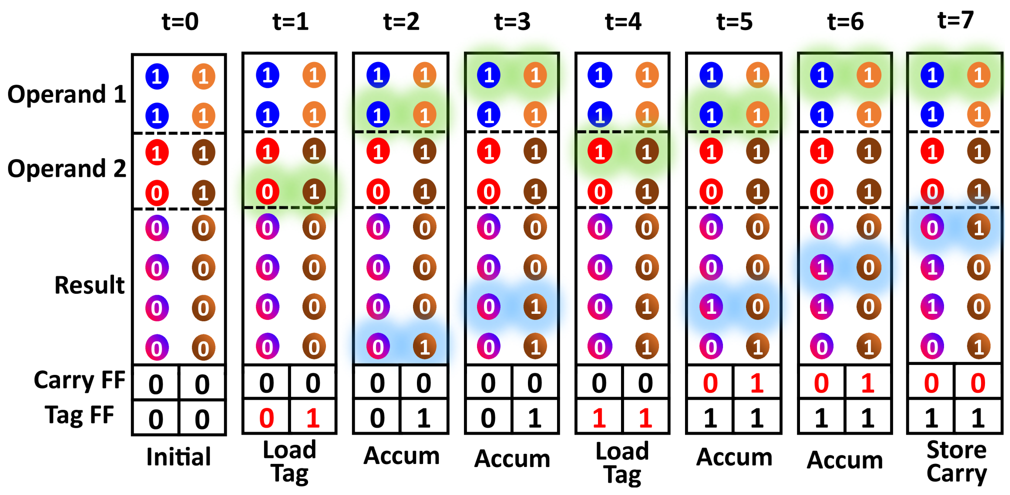
Figure 2.3: Neural Cache Architecture Overview.

takes n cycles; similarly, n-bit integer multiplication can be done in $n^2 + 3n - 2$ cycles. A step-by-step illustration of the bit-serial addition and multiplication are in Figure 2.4. The weights and activations are linearly quantized to 8-bit integers for bit-serial computation.

In the above architecture, 256 bitlines in one 8 kB SRAM array are turned into 256 ALUs in a vector unit. Xeon's 35 MB LLC can accommodate 4480 such 8 kB arrays. Thus up to 1,146,880 elements can be processed in parallel, while operating at a frequency of 2.5 GHz at runtime. By repurposing memory arrays, the above throughput is achieved for a cost of 7.5% area increase of an SRAM array and less than 2% area overhead for the entire Xeon



(a)



(b)

Figure 2.4: Steps for different in-SRAM bit-serial (a) addition, (b) multiplication operations. Green and blue shades correspond to read and write activated wordlines at each time step. The bits from a single data word are marked with the same color.

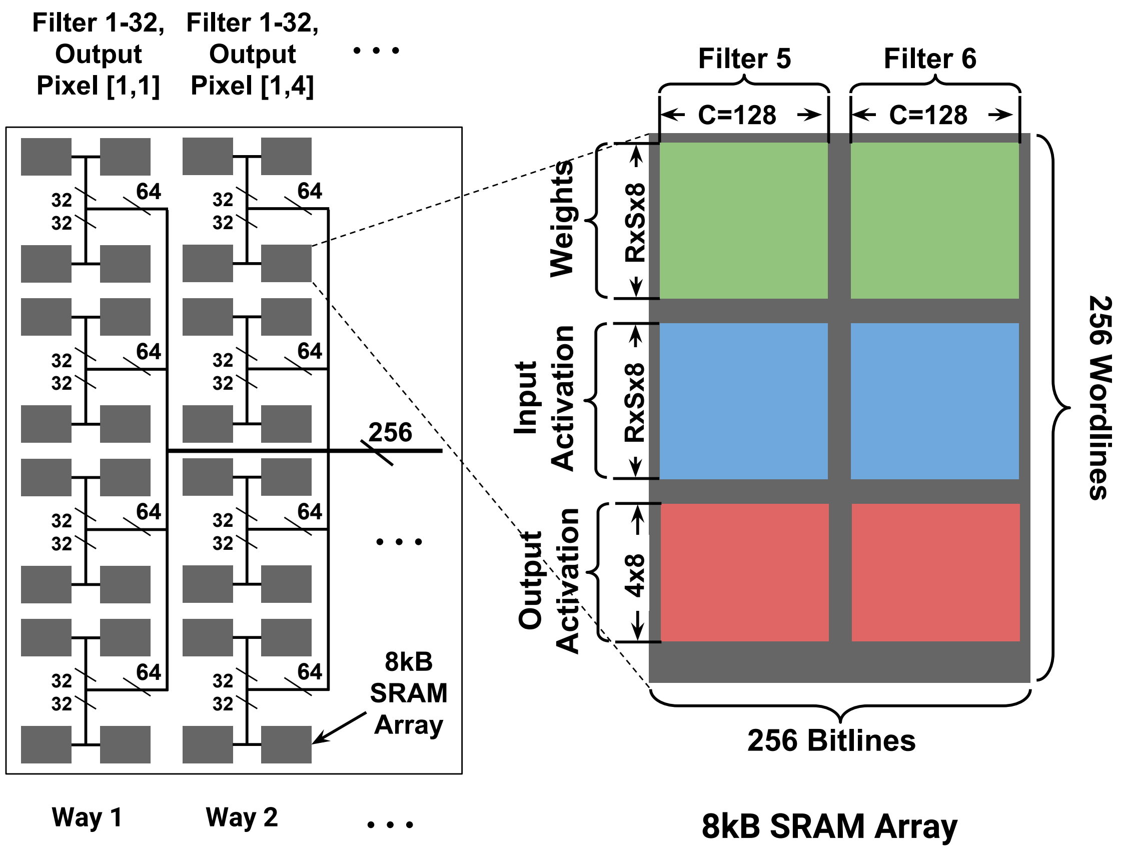


Figure 2.5: Neural Cache Data Mapping of a 2.5MB Slice.

processor die. Note, while a 35 MB LLC cache access from core takes 20-30 ns, the smaller 8 kB SRAM arrays can themselves operate at a frequency up to 4 GHz (Huang et al., 2013; Chen et al., 2013).

Neural Cache uses the cache structure of 2.5 MB Xeon LLC slice (Huang et al., 2013; Chen et al., 2013; Bowhill et al., 2016) for demonstrating its idea. In such a Xeon processor, multiple slices on a processor die are connected by a ring. Each slice has 20 columns which serve as 20 different ways of the set-associative LLC. Each way consists of 4 banks and each bank has 4 SRAM arrays of 8 kB. The 20 ways are connected by a 256-bit bus within the cache slice. The ways and banks in a slice are shown in the left hand side of Figure 2.5.

Figure 2.5 shows a typical data mapping scheme of one convolutional layer. In each 256×256 array, the filter weights are stored in $R \times S \times 8$ wordlines, and the input activations to be multiplied with weights are loaded to another $R \times S \times 8$ wordlines; each bitline corresponds to one input channel. The M filters span multiple arrays in the same or neighboring ways. One way of each slice is reserved for storing the outputs of the previous layer and another way reserved for system background processes. The other 18 ways hold the replicated weights for in-cache computation.

The in-cache convolution is performed in the five successive stages as described below:

1. Weight Loading: At the start of each layer, the filter weights are loaded from DRAM to the cache. Mapping a 2D filter to each bitline does not result in full utilization of all bit-serial compute units. Thus filters are replicated throughout all the ways and then slices to exploit parallelism. The inter-slice and intra-slice interconnect structures allow low-cost replication of weights using broadcasts. The weights from all the M filters are broadcasted to all the slices via the inter-slice ring, and then to all the ways via the intra-slice bus. After filter replication, the output pixels that still cannot be computed in parallel are computed in serial.

2. Input Loading: Each slice computes a tile (subset) of $E \times F$ output pixel positions across all M channels. The pixel positions with neighboring heights and widths are mapped to the same slice. Therefore, the output activations generated in one slice can be used as the input activation of the next layer. In input loading, the input activations are broadcasted from the reserved way to all the compute ways via the intra-slice bus. A small portion of the activations (at the border of tiles) are transferred through the inter-slice ring.

3. MAC (Multiply-ACcumulation): After data loading, at each array, the $R \times S$ weights multiply with the $R \times S$ inputs sequentially; after each multiplication, the product is accumulated into the partial sum in the reserved wordlines in the array.

4. Reduction: To sum up all the input channels of each filter, the partial sums at different bitlines are added up in the reduction stage. In reduction, at each array, partial sums of half the channels to be reduced are copied to another set of wordlines and aligned channel-wise with the other half of channels. Then the second half of partial sums are added into the first half. The copying and addition are called one round of reduction; the reduction rounds are conducted iteratively until the final reduction result is calculated.

5. Output Transfer: After reduction, the output activation maps at the compute arrays are transferred to the reserved array in the stage of output transfer. Then the inputs at a different height or width are loaded in, and the MAC, reduction, and output transfer repeat.

2.3 Sparsity and Quantization in DNNs

DNNs require a large amount of computation performed, as it usually consists of many layers and a large number of total weight parameters to obtain the high accuracy. Take CNN as example, the number of MAC operations required per convolutional layer is proportional to $R \times S \times C \times E \times F \times M$. This leads to efforts to reduce the total computation requirements by algorithmic changes to DNN inference itself.

Two types of algorithmic changes have been proposed in literature to reduce the computation requirements: 1) model pruning to reduce total number of weight parameters (Han et al., 2015); 2) represent the weight and/or feature maps with low-precision data formats (Rastegari et al., 2016; Zhu et al., 2016; Zhou et al., 2016).

In this section, we discuss two opportunities in DNN computation that lead to a reduction in total computation requiremetns, data sparsity and quantization, followed by a survey of hardware designs exploiting such opportunities.

2.3.1 Sparsity in DNNs

Data sparsity in DNNs especially means the zero values in the weight or feature maps. These zero values do not contribute to the result of MAC operation, so theoretically any MAC operations involving these zero values can be skipped.

Sparsity in weights: In a trained DNN model, the chance that a weight has exact zero value is low. The majority of the weight sparsity comes from an artificial optimization phase to increase the zero values in weights, called weight pruning.

Weight pruning selectively removes the weights in the DNN model. It is based on the observation that a typical DNN model has many weights whose values are close to zero (Han et al., 2015). If these near-zero values are collapsed to zero, then the sparsity is created and this intuitively does not affect the computation outcome significantly. The weight sparsity also leads to a reduced size of total model weights, which potentially results in reduced data loading time and energy.

A typical procedure of pruning (Han et al., 2015) is briefly described as follows. First, the weights to be pruned are determined - weights from the pre-trained model are evaluated by a type of regularization (for example, absolute value), and the weights with regularization less than a threshold are chosen to be pruned. Second, the pruned model is retrained to learn the values of the remaining weights in the network. The steps of weight pruning and retraining can be iteratively performed to increase model accuracy. After pruning, the network becomes sparse, and the percentage of the weights pruned is the *pruning rate*. Han et al. (2015) report an overall 63% pruning rate on convolutional layers of AlexNet.

Sparsity in feature maps:

The sparsity from feature maps mainly comes from a specific type of non-linear functions applied, called ReLU. The ReLU function returns zero for any negative inputs, and hence a significant portion of feature maps have zero value. The indices of the zero values are usually known only dynamically after an input is sent into the DNN model. In Rhu et al. (2018), it is shown that the activation sparsity is about 62% during the entire training period, on an average of eight CNN models.

2.3.2 Quantization in DNNs

It is known that on general-purpose hardware, integer operations are cheaper than floating point operations. Therefore, it would be beneficial to convert floating point values of weights and activations into discrete integers and compute with the integer representations. Such conversion is called quantization. In practice, 8-bit quantization of weights and activations usually has negligible effects on accuracy. For bit widths less than 8, researchers still achieve a reasonable accuracy with proper quantization schemes (Rastegari et al., 2016; Zhou et al., 2016; Zhu et al., 2016). A few ultra-low bit-width schemes have binary or ternary values for weights and/or input features, which effectively replace the arithmetic operations with a simple series of logical operations. There are also proposals that increase the number of weights while reducing the weight precision so as to reduce the overall computation complexity (Mishra et al., 2018).

Another type of quantization method is the block-floating-point (BFP) data format. In BFP, a contiguous part of the weight or feature map tensor share the same exponent bits but have their own mantissa and sign bits. The MAC operation can be performed on integer data within the exponent-sharing range.

2.3.3 Hardware Adaptations

Multiple recent works exploit the zero values in filter weights or input activations of DNNs and build ASICs based on such data sparsity. These architectures usually focus on efficiently fetching the non-zero data, as the indices of the zero values are often irregular, which is thus difficult to handle in parallel.

EIE (Han et al., 2016) is an ASIC for DNN inference which leverages sparsity in weights and activations. It uses a compressed model to fit the network in the on-chip SRAM. EIE focuses on the fully connected layers while the dissertation accelerates convolutional layers, which take up the majority of CNN inference time. SCNN (Parashar et al., 2017) proposes an accelerator for sparse CNNs with a dataflow that multiplies the non-zero weights and activations of the same channel while keeping them in a compressed format. Cnvlutin (Albericio et al., 2016) detects zeros in activations with hardware and skips ineffectual computation. Stripes (Judd et al., 2016) and Pragmatic (Albericio et al., 2017) apply variations of bit-serial computation and save computation time proportional to bit-width, where the necessary bit-width is determined by the network redundancy in each layer. Cambricon-X

(Zhang et al., 2016) is an accelerator with an indexing module for efficiently selecting the non-sparse weights and activations. The above works are dedicated ASICs for the purpose of sparse DNN acceleration. Compared to them, one advantage of the in-cache acceleration approach studied in the dissertation is being built as an extension to the commodity general purpose processors and thus more area-efficient.

There are works on software/hardware co-design for the sparsity in DNNs. Prior literature (Yu et al., 2017; Parashar et al., 2017) observed that, if pruning happens at the granularity of each individual weight, the weight sparsity cannot be easily exploited due to either the extra decoding stage of compressed weights or the low utilization rate of SIMD functional units. Many works therefore propose structured pruning at multiple levels (Yu et al., 2017; Wen et al., 2016), where the weights with one or more shared coordinate index in the $R \times S \times C \times M$ array are grouped together, and *each group either gets pruned entirely or remains entirely.* Such structured pruning is one example of software/hardware co-design for sparse DNN, and the methods proposed in the dissertation is similar, but catering to the case of in-SRAM computing in CPU last-level cache.

For the data quantization, the hardware customization usually focuses on implementing the ultra-low bitwidth computation efficiently. Our methods belong to this category, but customize for the bit-serial algorithms.

2.4 FPGAs for DNN

FPGA, or Field Programmable Gate Array, is a chip that consists of thousands of programmable logic blocks, memory blocks, and specialized computing blocks with a configurable interconnect among them. The FPGA developers describe the function with hardware description languages and the CAD tools of FPGA can automatically generate a mapping to the underlying blocks and interconnects with appropriate configurations. This leads to a faster developing and deployment cycle which allows for higher flexibility for new needs from workloads, compared to ASICs where chip fabrication is required before deploying. Additionally, FPGAs have the capability of performing DNN computation with low latency and high energy efficiency, thanks to its large number of computing blocks and flexible interconnects. In contrast, when DNNs are run on GPUs, although the peak computing throughput is also high, GPU has a fixed data path that is difficult to accommodate the DNNs' requirements, resulting in a higher single-input inference latency. With the above advantages above ASIC and GPU, the FPGA has been a popular choice for DNN acceleration.

2.4.1 FPGA Architecture

A FPGA chip consists of many small blocks with multiple functions, and the configurable interconnect among the blocks. The Stratix 10 GX 2800 FPGA (Int, 2019) contains 93312 LAB blocks (look-up tables), 11721 M20K blocks (BRAMs), and 5760 DSP blocks. All the blocks together form a large rectangle on the FPGA chip (432×273 blocks), with several irregularly scattered positions occupied by I/O banks or non-standard modules. Each column generally comprises one type of block. The LAB blocks take up most chip space; the columns with BRAM or DSP blocks are distributed roughly even on the chip.

The **M20K** block is based on an 8T SRAM array with 128 word-line pairs and 128 bit-line pairs for on-chip memory (extra 32 bit-lines are used for ECC bits). In each bit-cell, there are two wordlines that correspond to two separate ports to access the SRAM array. The wordline pair works concurrently and the dual-port mode can be implemented with M20K blocks. Bit-lines are multiplexed 4:1, so the read/write port is 32-bit wide. The output width can be reconfigured to less than 32 bits, and the real data are selected from the 32-bit data according to the result of a width configuration decoder. The SRAM array operates at 1.0 GHz.

A **LAB** (Logic Array Block) block is made up of 10 Adaptive Logic Modules (ALM), where each ALM contains two 6-bit look-up tables. A fraction of LAB blocks can also be configured as on-chip memory, by holding the data in the look-up table registers.

The **DSP** blocks serve as configurable ALUs supporting both fixed point and floating point arithmetics. For example, one DSP block can be configured as a single-precision floating-point adder.

2.4.2 DNN Acceleration on FPGA

The acceleration of DNN mainly relies on the large number of DSP blocks, which can be flexibly connected. The most common operation of DNN, MAC, can be implemented with one DSP block under floating-point precision. Previous works have proposed various methods to efficiently map the DNN computation into available compute resources on FPGAs.

There are many works that propose methods to efficiently map the DNN computations into available compute resources on FPGAs (Ma et al., 2017; Alwani et al., 2016; Li et al., 2019; Wang et al., 2018; Han et al., 2017; Lu et al., 2019; Shen et al., 2017a; Samragh et al., 2017; Colangelo et al., 2018; Rasoulinezhad et al., 2019; Shen et al., 2017b; Guan et al., 2017; Venieris and Bouganis, 2016), as FPGAs are naturally useful for building domain specific accelerators.

Brainwave (Fowers et al., 2018) is a DNN accelerator on FPGA focusing on reducing single-input latency. Its key programming abstracts include matrix-vector multiplication and element-wise vector options, which are the two major operation types for RNN, but can also be extended for other DNN models. The computation of a matrix-vector multiplication is tiled to dot-product modules, and there is also aggregation logic for the results from multiple matrix-vector multiplication units. The DNN model weights are pinned onto the on-chip SRAM memory on FPGA, to avoid additional data movement and obtain a high data accessing bandwidth. When there are multiple layers in an RNN model and the model weight size is larger than the BRAM capacity of FPGA, multiple FPGA boards are chained to pipeline the computation of the layers. Additionally, Brainwave uses block floating point as the data format which results in higher computing density as one DSP block can be configured to run two integer MACs per cycle. Brainwave achieves a magnitude of improvement in latency for RNN workloads over a GPU with a similar technology node. In this work, we re-architect Brainwave to showcase how our proposed compute-capable BRAMs can further benefit a DNN accelerator on FPGA with the additional computing throughput.

Different from weight pinning and partitioning in Brainwave, Other works have focused on mapping optimizations or model compression techniques to mitigate the problem of large size of model weight. Alwani et al. (2016), Shen et al. (2017a) aim to reduce off-chip data transfer by tiling multiple layers rather than processing layer-by-layer. Ma et al. (2017), Han et al. (2017), Samragh et al. (2017) use pruning/compression/encoding techniques for FPGA. Li et al. (2019) proposes a design methodology to determine best parameters (block-size) for compression techniques. Shen et al. (2017b) develops an efficient algorithm that computes a partitioning of the FPGA resources into multiple Convolution Layer Processors (CLPs) for higher utilization. On recent FPGA models Int (2019), there are embedded DRAMs on the FPGA chip to reduce off-chip data access.

In general, this dissertation work is orthogonal to all above works. We present a reconfigurable accelerator architecture that can flexibly balance between in-BRAM and DSP compute to achieve the highest performance. We enhance FPGA BRAMs with in-memory compute capabilities, a more fundamental change in the device architecture which can potentially benefit any accelerator architecture.

There are also prior works that change FPGA architecture for more efficient DNN acceleration. Boutros et al. (2018) and Rasoulinezhad et al. (2019) re-architect the DSP blocks to support higher density of low precision MACs for DNN inference. Arora et al. (2020) proposes integrating in-fabric tensor units for high-efficient matrix-matrix multiplications used in many DNN models. Boutros et al. (2019) and Eldafrawy et al. (2020) propose several ideas for logic block changes that can substantially increase the arithmetic density of the soft fabric. These works mainly focused on DSP and logic blocks, or seek to add new specialized blocks onto FPGA. Different from above approaches, the proposed work attempts to enhance the FPGA's on-chip memories with massively parallel computing capabilities.

It is worth noting that beyond FPGAs, there are other proposals to build in-memory DNN accelerators with a certain level of reconfigurability. Zha and Li (2018) proposes a ReRAM based reconfigurable fabric where each tile can be configured for compute, memory or interconnect mode. While the above work builds a new architecture based on ReRAM for reconfigurable in-memory computing, our proposed approach leverages the existing BRAMs on FPGA for computation with minimal modifications, maintaining the reconfigurability of FPGAs.

2.5 DNNs on CPU

After looking at how to accelerate DNNs with a variety of customized hardware, in this section we discuss some existing methods for running DNNs on CPUs efficiently.

While there are software and compiler optimizations targeted for DNNs (Liu et al., 2019), DNNs can also benefit from generic microarchitectural improvements of CPUs. In this dissertation we specially pay attention to cache compression, where a larger size of data can be stored in cache after compression techniques, leading to a reduction in cache miss rate and thus performance improvement. The DNN computation can benefit from the cache compression in the following ways. First, in the case of quantized computation, the total number of data values are limited (e.g., 256 for 8-bit quantization), so the data value duplication is common and thus can benefit from compression. Second, as the weight and feature map values are concentrated at zero, their exponent bit values (in 32-bit floating point format) are similar and thus has a higher chance of being duplicated.

Another scenario of interest is when DNN workloads are under collocation with other workloads, which is common in datacenters to increase machine utilization.

2.5.1 Cache Data Compression

Cache data compression is about leveraging data redundancies in cache contents to store them in fewer bytes. As cache is organized in the unit of cachelines, earlier works have explored compression methods for content of each cache line, which is also called intra-cacheline compression. More recently, researchers have discovered that there are also opportunities of compressible patterns across different cachelines, with the inter-cacheline compression methods being developed.

Intra-cache-line compression. Existing works have explored the data compression methods within a cache line to reduce the occupied space of each single cache line. FPC (Alameldeen and Wood, 2004) checks whether each data word belongs to some known pattern that can be stored in fewer bits (such as storing 8-bit value in 32-bit word), and uses a prefix to indicate the compressed patterns. C-Pack (Chen et al., 2009) detects the frequent patterns both statically and dynamically. BDI (Pekhimenko et al., 2012) compresses a cache line by representing it with a base value and a set of differences, as the data chunks within a cache line can have similar values. ZCA (Dusser et al., 2009) leverages the large amount of zeros in the cache data and designs a structure to represent zero blocks of different lengths.

Inter-cache-line compression. The inter-cacheline compression methods further leverage the data similarities across multiple cache lines, to further increase the range of potentially compressible data. SC^2 (Arelakis and Stenstrom, 2014) utilizes the Huffman coding to reduce the storage size of the frequent data in cache. Dedup (Tian et al., 2014) detects the completely duplicated cache lines with hashing functions, and uses doubly-linked lists to manage the mapping from the tags to the data. Our compression method performs data comparison in a finer granularity than Dedup. Thesaurus (Ghasemazar et al., 2020) utilizes locality-sensitive hashing to find similar cache-lines with few differed bits and thus compress on them. There are works that aim to reduce the tag overheads in cache compression by using one address tag for multiple cachelines (Sardashti and Wood, 2013; Sardashti et al., 2014). Recently there are works that propose cache compression based on program information

(Tsai and Sanchez, 2019).

The above cache compression works are unaware of the underlying hardware architecture when searching for data duplication. Instead, this work utilizes the cache geometry information to find more compression opportunities without sacrificing decompression latency, and also uses in-SRAM comparison technique for quick similarity search.

2.5.2 QoS for Collocated and DNN Workloads

As CPU-based servers are widely used in modern datacenters, many previous works have studied the problem of improving QoS of generic workloads on the CPU-based servers under collocation. They show that the resource contention can lead to a degradation of QoS on the workloads and apply the resource partition as an effective way to mitigate the problem. Heracles (Lo et al., 2015) develops isolation mechanisms of various resources such as CPU core, cache, memory and network bandwidth. Then a scheduler is designed to collocate latency critical workloads with best effort workloads. PARTIES (Chen et al., 2019) designs a scheduler to allocate resources dynamically to satisfy QoS targets of multiple concurrent latency-critical workloads. The scheduler adjusts the resource allocation based on the current QoS status of the workloads. There are other works that improve on PARTIES for reducing energy consumption with reinforcement learning (Nishtala et al., 2020), or increasing utilization with a systematic optimizer (Patel and Tiwari, 2020). SMiTe (Zhang et al., 2014) studies the resource contentions on multi-core CPUs, with a focus on the shared resources within a CPU core due to hyperthreading. Then a performance prediction model is developed based on the applications' contention properties, and a scheduler is proposed to safely co-locate workloads without QoS violation. Unlike this work, the above works do not specifically target the DNN workloads. We specifically characterize the DNN applications, and leverage the properties of the DNN workloads to build a customized prediction model. With the prediction model, we envision that our proposed resource manager can more quickly adapt to fluctuations in the requests, and also achieve higher overall system efficiency than these previous works.

With the popularity of DNN as a datacenter service, some other relevant works have explored the QoS problem specifically for DNN workloads. GrandSLAm (Kannan et al., 2019) studies the QoS for machine learning applications with multiple stages, on both CPU and GPU platforms. It proposes a model that predicts the execution time of a single stage, such as image classification, based on the size of a single input, the batch size, and the queueing delay. Then it dynamically batches the execution for identical stages from different applications, to improve system throughput without QoS violation. Kelp (Zhu et al., 2019) analyzes the sensitivity of machine learning workloads to the resource contention, on the accelerator and GPU platforms. They claim that the memory bandwidth is a critical resource for the workloads and thus propose mechanisms for isolating the memory bandwidth resources to improve QoS. Some recent works have focused on the QoS of machine learning under contention on dedicated accelerators. AI-MT (Baek et al., 2020) proposes a new design of DNN accelerator architecture for executing multiple DNN inference concurrently. It partitions the layers of DNN at compile time and then schedules the sub-layers efficiently to increase the resource utilization. The dissertation studies a different scenario for QoS of DNN, which is under collocation on CPU-based servers.

CHAPTER 3

Efficient In-Cache Acceleration of DNNs

In the previous chapter, we have seen that the in-SRAM computing is effective for accelerating CNNs with bit-serial arithmetics. We also see that the data sparsity, especially the sparsity in the CNN model weights, can lead to a lot of unnecessary MAC computation; the lower bit-precision formats can be applied to CNNs and they can benefit from specialized arithmetic operation.

In this chapter, we show how the existing in-SRAM acceleration architecture can be enhanced to leverage the sparsity in the DNN weights and the low-precision data formats. We propose hardware/software co-design techniques for tackling the above two problems. First, Section 3.1 describes the two distinct methods for leveraging the sparsity in model weights, namely coalescing the channels and overlapping the filters. For each method, we show how the modified pruning algorithm can benefit the in-SRAM computing architecture, and we describe the extra hardware units to support the pruned model. Then, Section 3.2 shows the customized design for reduced precision DNN. Section 3.3 describes the evaluation methodology and Section 3.4 demonstrates the evaluation results, on the achieved performance, energy, and a trade-off between performance and model accuracy.

3.1 Sparsity-Aware Architectures

The sparsity in filter channels can be leveraged to eliminate the energy spent on computation with zero-valued filters, and speed up the convolution by filling up the in-SRAM compute

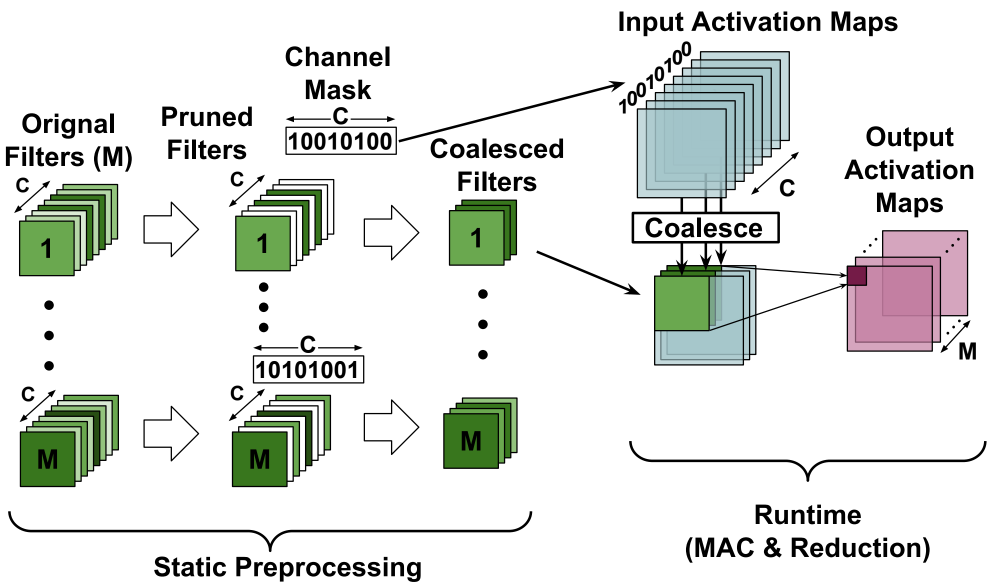


Figure 3.1: Sparse Convolution with Coalescing.

slots with *effective* computation. Neural Cache architecture does not leverage sparsity. Each SRAM array in Neural Cache is repurposed to function as a vector unit with 256 SIMD slots. Sparsity in filter channels, when stored in a dense format, leads to wasteful computation over bitlines because some slots in the vector unit are being utilized to compute zero. We propose two techniques to avoid this, as discussed below.

First, we propose to coalesce all non-zero filter channels into consecutive SIMD slots (i.e., bitlines). This data-mapping creates a dense structure from sparse filters. The filters can be coalesced statically apriori during the offline pruning/training phase. However, input channels or activations are not known until runtime and need to be coalesced dynamically. We propose an area efficient coalescing unit to dynamically coalesce the input channels. *Second*, we explore the idea of overlapping non-conflicting filter channels in different 3D filters. There are M possible 3D filters. The advantage of this technique is that input channels do not require coalescing or any additional encoding during runtime. The disadvantage is that the pruning rate achieved is lower due to additional restrictions imposed by the overlapping requirements.

Next, we describe the details of the above techniques, namely coalescing channels and

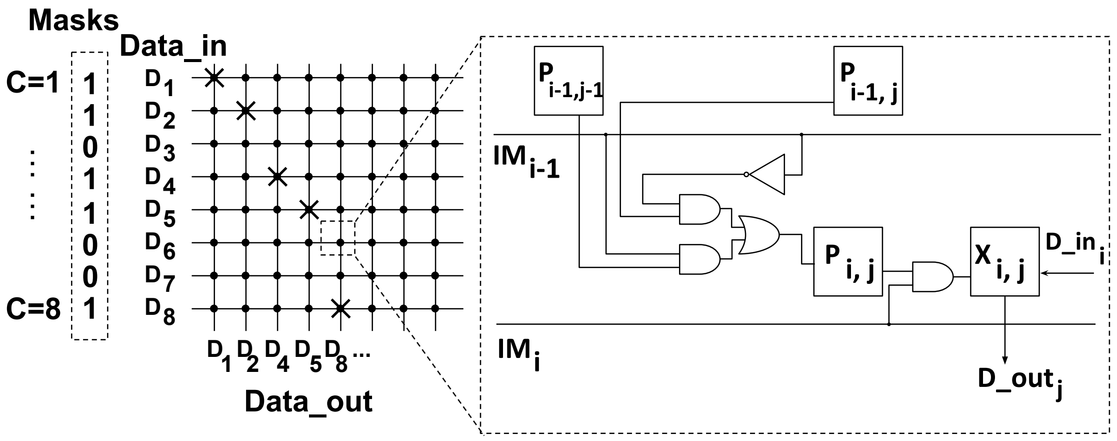


Figure 3.2: Architecture of Coalescing Unit (CU). Left: Crossbar and example data before coalescing (eight channels - D_1 to D_8) and after coalescing (D_1, D_2, D_4, D_5, D_8). Right: Reconfiguring Peripheral for one switch connectivity. $X_{i,j}$: Output of RP to Crossbar, whether the switch at (i, j) is connected. $P_{i,j}$: Boolean value of whether the position (i, j) is on the "path." IM_i: Channel mask for the i-th channel.

overlapping filters.

3.1.1 Coalescing Channels

Figure 3.1 provides an overview of the procedure of sparsity-aware computation by coalescing channels. We follow the structured sparsity approach (Yu et al., 2017) for pruning and use the granularity of a filter channel for pruning. Our pruning algorithm (described later in this section) selectively prunes unimportant $R \times S$ 2D-filters. The choice of 2D filter as a granularity of pruning finds the desired balance between achieving a high pruning rate, and keeping the encoding of the pruned filters simple. Each compressed 3D filter is equipped with a channel mask which distinguishes the non-zero channels from zero channels. Different 3D filters are permitted to have different sparsity patterns (and different channel masks) as shown in the figure.

We propose a dynamic coalescing process that helps to align input activations to heterogeneously pruned filters (conceptually shown in right hand side of Figure 3.1). Before coalescing, the input data has C channels with each of them being an $H \times W$ activation map.

There are M sets of different $R \times S \times C$ filters for one convolutional layer, and the inputs are coalesced according to the M filter pruning patterns. After pruning, each of the M filters is equipped with one channel mask vector of C bits, where each bit indicates whether the 2D $R \times S$ filter at the corresponding channel is pruned (0 for pruned channels, 1 for unpruned channels). During coalescing for the m-th filter, if Mask(c,m) = 1, then the c-th input channel is copied to the coalesced input data; otherwise the c-th input channel will not appear in the coalesced input data. Therefore, after coalescing, only the input channels that correspond to unpruned filter channels are preserved; such input channels and filter channels are automatically aligned. After coalescing, convolution is performed only on the dense channels of the filter and activation.

3.1.1.1 Dynamic Input Coalescing

To implement dynamic input coalescing, we propose the design of *Coalescing Unit* (CU) and place the CUs in the reserved way. Each CU is connected to one 16 kB subarray, and is placed between the sense-amplifier and the intra-slice data bus connecting multiple ways in a slice. In total 8 CUs are required per L3 slice. Note, input activations are broadcasted from the reserved way to all the compute ways via the data bus in Neural Cache. Thus placing CUs only in the reserved way is sufficient. The activation data are coalesced right before being transferred on the bus, making the incoming input data at compute arrays already aligned with the coalesced filters, with a minimal hardware overhead of CUs. The Coalescing Unit consists of a 256 bit × 256 bit crossbar, reconfiguring peripheral (RP) which determines which inputs of the crossbar are connected to which outputs, and a FIFO buffer.

Crossbar: At the input interface, 256 bits from 256 input channels are fed to 256 data_in ports. The 256×256 switch connectivity configuration is initialized once per layer by RP. With the configured switch, the output of crossbar is an ordered gather of the input data bits with corresponding mask value 1. With the crossbar, the output can be any combination and arrangement of the 256 bits, so the inputs can be coalesced under any sparsity pattern.

Figure 3.2 shows the structure of a Coalescing Unit with a smaller 8-bit input/output vector size, as well as the circuits of RP for one switch value output. In this example, among the 8 input channels, the channels 3, 6, 7 are pruned, so the outputs are data from the channels 1, 2, 4, 5, 8. Under some scenarios where input channel number is small or prune rate is high, after pruning, there are more than one 3D-filter mapped to one 8 kB array. The crossbar can still handle this situation since each input can be fed to multiple outputs at arbitrary position.

Reconfiguring Peripheral (RP): The RP can be implemented in combinational logic. The right side of Figure 3.2 shows how one switch value produced by RP is related to the neighboring switch values; such circuit is repeated at each cross-point to generate the 256×256 switch value array. The key idea is: if one switch is set, then the switch to its *right-down* may be set; else, the switch to its *down* may be set. We call such a dependency chain a *path*, and the switches on the path have the potential to be set. Whether a switch on the path is set or not depends on whether the corresponding channel mask is 1. In Figure 3.2, $P_{i,j}$ denotes whether a switch is on the path; $X_{i,j}$ is the actual switch value of the position (i, j) produced by RP. IM_i is the input mask bit of the channel i, representing whether the input channel i should be passed to the crossbar output.

Formally, the logic of $P_{i,j}$ and $X_{i,j}$ is:

$$P_{i,j} = (P_{i-1,j} \wedge \overline{IM_{i-1}}) \vee (P_{i-1,j-1} \wedge IM_{i-1})$$

$$X_{i,j} = P_{i,j} \wedge IM_i$$

$P_{0,j}$ (the first row) do not depend on any other switches and will be initialized with preset value. $P_{i,0}$ (the first column) can also be overwritten with initial value besides depending on the switch $P_{i-1,0}$.

When there are multiple 3D-filters mapped to the same array after pruning, the input masks of different filters are passed to the RP sequentially and the switch values are updated sequentially for each filter. Also, the starting point of the path is set according to the actual starting position of the current filter in the output. For example, if the first filter has 32 unpruned channels, then when the RP calculates switch values for the second filter, $P_{0,32}$ is set to 1 as the first row initialization. The number of unpruned channels for each filter can be stored alongside the filters (8 bits for a 256-channel filter). The upper triangle of the crossbar is used in this case, to coalesce the activations starting from the second filter mapped to the array. If a filter spans more than one array, on the second array, $P_{j,0}$ is set to 1 where j is the first input channel on the path of the second array. Note that this whole RP configuration process is done once per layer. This is because even with multiple output pixels to compute in serial, only the input pixel position changes; the indices of the effective input channels remain the same.

FIFO buffer: Due to the column muxing for the bitlines, 32 bits of data are read out of the SRAM array, and the data bus for each array is also 32-bit wide. However, the interface of the crossbar is 256-bit wide. Therefore, the FIFO buffer serves as a bridge for balancing the different bandwidths between the data bus and the crossbar, and between the crossbar and the SRAM array. It collects every 32 bits from array sense-amps and sends in 256-bit data to the crossbar. At the output of the crossbar, the FIFO buffer transmits the 256-bit data to the bus in 32-bit chunks.

The crossbar size of 256×256 can handle filters with up to 256 channels. To accommodate filters with more than 256 channels, we simply split the filters into smaller ones, each with only 256 channels, and treat them as multiple sequential filters. The normal convolution steps can proceed until the reduction ends. The reduction results of each 256 input channels can be summed up in the end.

3.1.1.2 Convolution Operation

With the coalesced input channels, a few modifications to the convolution procedure are proposed regarding the irregularity in data mapping.

(a) Weight Loading: The filter weights of the pruned model can be coalesced offline after training, before the inference process starts. All the weights in the pruned channels are eliminated so they are no longer loaded from DRAM to the cache. The weight loading time is therefore reduced proportionally to the percentage of channels pruned.

(b) Input Loading: With the input-loading aware structured pruning design, to compute output pixels in one position, it is sufficient to load the coalesced inputs *once* from the reserved way to all the compute ways via the intra-slice bus. The total time for input loading will remain the same as baseline, mainly because the total amount of transferred data does not change. The incurred overhead to input loading includes CU latency and the time to load the masks for coalescing. The CU latency is short when compared to bus transfer; the data size of masks ($C \times M$ bits are required per layer) is small compared to weights.

(c) MAC: The total number of output pixels computed serially is reduced because fewer input channels need computing. Hence, more output pixels can fit in the same number of bitline computing slots, reducing the output pixels computed in serial. Thus the total time on MAC is reduced. Note, the standard MAC algorithm is unaffected by coalescing, since the filters are pruned at the granularity of each channel, instead of individual weight.

(d) Reduction: The reduction stage needs modification because the boundaries for filters are irregular now. We design a "preparing round" to tackle this difficulty. The core idea of preparing round is to perform initial reductions within each filter until there is enough space to fit in all the filters. Two sets of fresh wordlines are allocated for performing copying and addition. The eight 32-bit segments in a 256-bit wordline are sequentially copied to the newly allocated wordlines, where they match up for addition. The 32-bit segment size is a result of 8-way column multiplexing of SRAM arrays. When copying a segment consisting of weights from more than one filter, the 32-bit segment is first AND-ed with a mask to selectively copy the weights only from the desired filters. After the preparing round, the filters are lined-up ready for normal reduction. The reduction is done in parallel for all the filters within an array. For filters that span more than one array, the reduction within each array is first done, before intra-array results are added up finally.

For example, in AlexNet layer conv3, C=256. In one array, there are 3 different filters for reduction, and the corresponding bitlines are BL1-BL96, BL97-BL160, BL161-BL256. Each partial sum takes up wordlines WL1-WL32. In the preparing round, 64 new wordlines (WL33-WL96) are allocated, and the partial sums of the n-th filter are copied to these 64 wordlines of the n-th bitline quarter. For instance, for the first filter, BL1-BL64 are copied to WL33-WL64 of BL1-BL64; BL65-BL96 are copied to WL65-WL96 of BL1-BL32. Then the second half of new wordlines are added into the first half. After such starting round, each filter takes up 64 bitlines and the remaining reduction can be done as in Neural Cache.

3.1.1.3 Weight Pruning Method

Similar to the previous pruning techniques (Han et al., 2015; Yu et al., 2017), we prune the weights that would have the least impact on final inference results. We use the L2-norm (sum of squares of all elements) of each 2D filter to measure its importance. All the 2D-filters with an L2-norm lower than a given threshold are pruned. This criterion is straightforward and also effective in our experiments. Other criteria for weight importance measurement can also be directly applied to prune networks at the same granularity.

The pruned models will be fine-tuned to regain the accuracy. A $[C \times M]$-bit mask is generated for each convolutional layer to indicate the pruned 2D filters. All masked weights are fixed to 0 during the retraining. The pruning and retraining steps will be performed layer by layer to achieve the best accuracy. Also, the pruning rate will be gradually increased until the targeting accuracy cannot be met.

To avoid the irregularity in input activation loading, we propose a different, more structured, criterion for pruning. To minimize the number of on-bus data transfers, it is desired to have all the SRAM arrays connected to the same data bus to get the exact same channels after pruning. Note that in each 16 kB sub-array, the two 8 kB arrays share the same sense-amplifiers, and thus using the same pruning pattern for such two arrays is beneficial. We denote the eight 8 kB arrays within the same way and at the orthogonal dimension to the dimension of the shared sense-amps as a *half-way*, and the channel masks need to be the same for all the half-ways.

The procedure for the more structured pruning is as following: **(1)** According to the pruning rate, calculate the number of channels in all filters after pruning. Dividing this total channel count by the bitlines (channels) in a half-way, we can get the number of half-ways, Y, required for one output pixel position. **(2)** Divide the M filters equally into Y half-ways. Group the channels that have the same position within a half-way together. Then there are $C \times M/Y$ such channel groups. **(3)** Calculate the L2-norm of each channel group and order them. The groups with the lowest L2-norm will be pruned until the pruning rate is met.

The above pruning procedure is necessary for efficient dynamic coalescing. Without the customized pruning approach, the weight sparsity rate is so low that there is little opportunity to coalesce. The pruning and retraining process is offline, so the inference performance is not affected.

3.1.2 Overlapping Filters

Overlapping is another technique we design to leverage sparsity. The key insight is that, if different filters are sparse at different channels as a result of customized pruning, then they can be overlapped as one filter when doing MAC operations. Figure 3.3 provides an overview of the convolution process with filter overlapping. During offline training, the filters are grouped and pruned so that multiple filters can be combined into one. At runtime, the compressed filters directly perform a 2D-MAC with the input activation maps – channels are naturally aligned; at reduction, the partial sums from different filters are gathered correctly according to the channel mask vector.

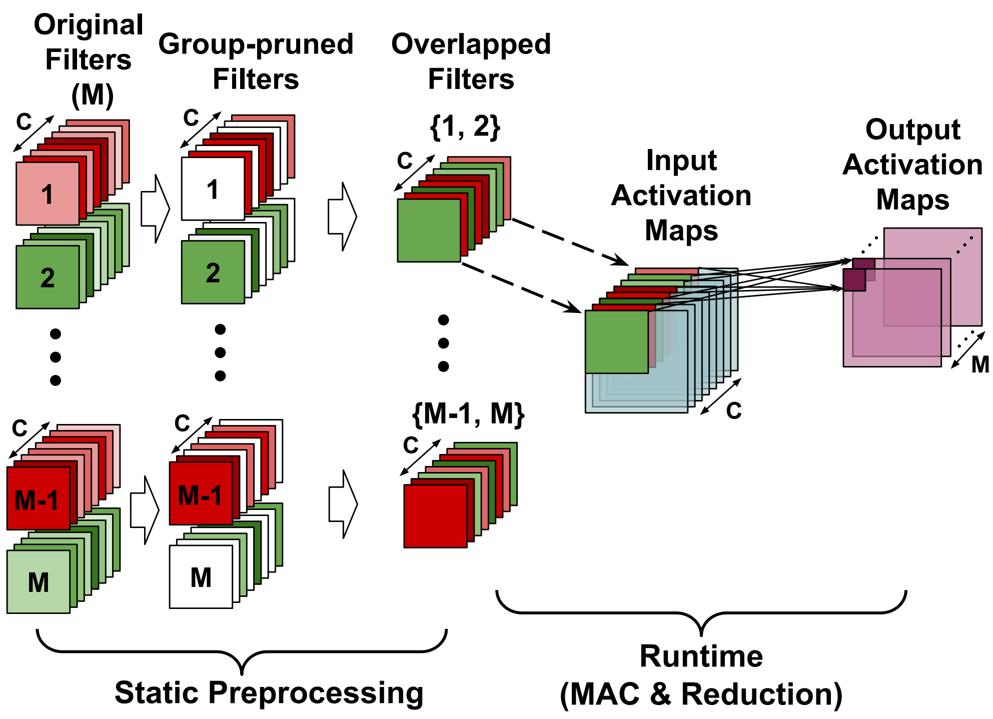


Figure 3.3: Sparse Convolution with Overlapping.

3.1.2.1 Convolution Operation

After the filters are overlapped in preprocessing, the weight loading, input loading, and MAC stages for convolution are the same as the baseline. The reduction stage is different from baseline. Similar to reduction with coalescing, an additional preparing round is required before the normal reduction operation, due to the interleaving of MAC results computed with multiple 3D-filters. In the preparing round, two sets of fresh wordlines are allocated (the number of wordlines of each set equals the bit width of partial sum). For an array with N filters, the bitlines are partitioned into groups of $256/N$ bitlines, and the partial sum from each filter is copied to its own group of $256/N$ bitlines to perform reduction in the preparing round. Copying of partial sums is done selectively with channel masks. After the preparing round, reduction for the N filters in the same array can be done in parallel. For example, in AlexNet layer conv3, C=256. In the first 8kB array, there are 32 wordlines (WL1-WL32) storing the partial sums of the overlapped filters (M=1,2). In the preparing round for reduction, the BL1-BL128 of WL1-WL32 are selectively copied to another set of wordlines, WL33-WL64, based on channel-mask for C1-C128 of M1. Then, BL129-BL256 of WL1-WL32 are selectively copied to BL1-BL128 of WL65-WL96, based on channel-mask for C129-C256 of M1. Next, the partial sums for M2 are similarly copied to BL129-BL256 of WL33-WL96. Then, addition is done between WL33-WL64, and WL65-WL96. After the above starting round, the partial sums of filter M1 are in BL1-BL128, and those of M2 in BL129-BL256. So the remaining reduction can proceed as in original Neural Cache. The total latency for reduction will drop thanks to the reduced number of output pixels computed in serial. The extra overhead includes the time for loading the channel masks, consisting of $C \times M$ bits per layer, as well as the extra preparing round.

3.1.2.2 Weight Pruning Method

Similar with coalescing, the filter weights are pruned at the granularity of each $R \times S$ 2D-filter. With the overlapping scheme, it is desired that the M different filters are pruned at different channels. Such pruning generates remaining filters that can overlap with each other. To achieve this, we can explicitly control which channels are pruned by carefully designing the weight masks. All the M 3D filters are divided equally into "overlappable groups," where each group has N filters. The N filters in each group are masked such that at each channel, only weights from one filter are unpruned, while others being pruned. The remaining filter must have the highest L2-norm of its weights over other filters at the channel. For example, if N=2, then we take the 1st and 2nd filters, compare the L2-norm of the 2D-filter weights of the two filters iteratively from the 1st channel to the last (C-th) one. At the channel c, if L2-norm$(c, 1)$>L2-norm$(c, 2)$, then Mask$(c, 1)$=1, Mask$(c, 2)$=0. Otherwise, Mask$(c, 1)$=0, Mask$(c, 2)$=1. The masks for the 3rd and 4th, ..., (M-1)-th and M-th filters are generated with the same rule. With this pruning procedure, the pruning rate is $1 - \frac{1}{N}$. It is possible to prune the filters in other patterns for overlapping, which might yield better accuracy. Such exploration is left for future work. After the masks are generated, the network is fine-tuned layer by layer as described in Section 3.1.1.3. To adjust the target pruning rate, the parameter N can be tuned on a layer-by-layer basis.

3.2 Reduced Precision Architecture

Reducing the bit width of filter weights and activation maps, while maintaining a reasonable inference accuracy, is well studied in literature. Due to the bit-serial characteristic of the in-SRAM computation, it is natural to leverage the reduced bit width to accelerate computation. Specifically, using ternary and binary weights can further replace the requirement of multiplication simply with addition.

With fewer bits in the weights and activations, it is possible to fit more filter weights along a bitline and avoid unnecessary filter splitting (where one 2D filter is split and mapped to multiple bitlines), thus reducing the total number of reduction needed. Also, the maximum possible bit width for reduction result decreases, saving the cycles for copying and addition in reduction. Finally, less time and energy will be spent on data movement for loading weights and input activations - the data size to load is proportional to the bit width.

3.2.1 Ternary Networks

Ternary networks have weight values from the set {0, 1, -1}. Each weight can be encoded with 2 bits – one for sign (bit 0 for positive; 1 for negative), the other for magnitude (bit 0 for zero; 1 for one). To simplify computation, we require a zero be encoded as positive zero. The multiplication of ternary weights with quantized activations can be achieved with supported arithmetic operations. First, each bit-slice of activation is AND-ed with the magnitude bit to perform the multiplication with zero-valued weights. The AND-ed results are written to a new set of wordlines for the product. Second, each bit-slice of the product is XOR-ed with the sign bit, to compute the 1's complement of the product for those channels with weight values -1. Finally, the products are added to the partial sums for accumulating the products from the same channel, after being sign-extended to match the partial sum bit width. The sign

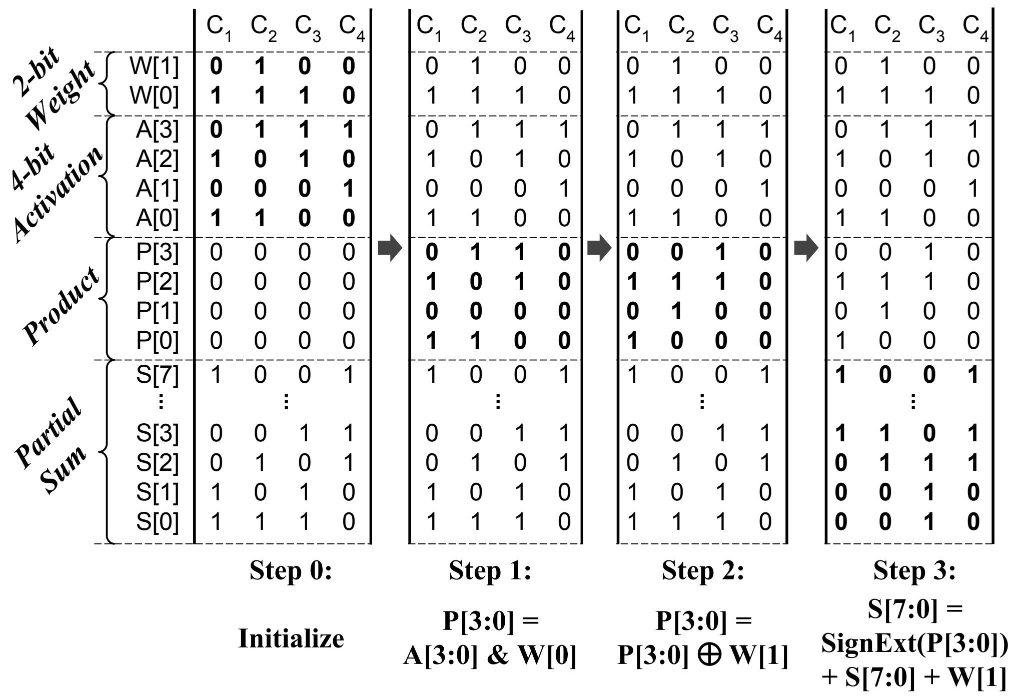


Figure 3.4: MAC step of Ternary Network.

bit of weight is used as carry in the addition, and therefore the 1's complement generated in the previous step will become 2's complement here. Collocating the conversion to 2's complement with the partial sum accumulation saves one addition operation.

Figure 3.4 demonstrates an example of 4-bit activations convolving with ternary weights. The figure shows one multiplication and addition in the MAC step, where 4 channels are computed in parallel. W[1] is the sign bit and W[0] is the magnitude bit. In Step 1, A[3:0] is bitwise AND-ed with W[0] and the results are written to P[3:0]. In Step 2, P[3:0] is bit-wise XOR-ed with W[1] and the results update P[3:0]. In Step 3, P[3:0] is accumulated into partial sum S[7:0], while being sign-extended with the weight sign bit W[1] and also using W[1] as the carry. In this example, 8 cycles are spent on multiplication (Step 1-2), and another 8 cycles are spent on addition (Step 3).

3.2.2 Binary Networks

Binary networks have weight values of either 1 or -1. Each weight can be encoded with only one sign bit. MACs of weights and activations can therefore be converted to addition and subtraction of activations. For the in-cache compute, the algorithm is similar to the one for ternary networks, but without the first step of AND-ing with magnitude bit. For 4-bit activations, it takes 4 cycles to do one multiplication, and a number of partial sum bit width cycles for one accumulation.

3.3 Evaluation Methodology

We evaluate the latency, energy, and the model accuracy on the following configurations: CPU and GPU baseline (CPU-base, GPU-base), Neural Cache baseline (N$-base), sparsity-aware architecture with coalescing (SPR-coal)/ overlapping (SPR-olap), and low-precision architecture with 4-bit activation, ternary (LPR-2b)/ binary (LPR-1b) weights.

CPU/GPU Baseline: The dual-socket Intel Xeon E5-2697 v3 serves as the CPU baseline platform; Nvidia Titan Xp as the GPU baseline. The hardware specifications are summarized in Table 3.1. We use TensorFlow v1.9 and its profiler as the software framework and the tool for measuring the latency of CPU/GPU baseline. For CPU energy and power measurement, we use RAPL tools (David et al., 2010). Nvidia-SMI is used for measuring GPU power and energy. Note that the baseline CPU chip has the exact same L3 cache structure as the model for the Neural Cache architecture.

Neural Cache Baseline: We use an in-house cycle-accurate simulator for estimating the time and energy for Neural Cache arithmetic. For fair comparison, we use the same cycle latency and energy model as reported in the paper (Eckert et al., 2018). The in-SRAM compute frequency is at 2.5 GHz. The energy per cycle is 15.4 pJ and 8.6 pJ for compute and read/write cycles in SRAM, respectively.

The time of data loading (filter weights and input activations) depends on the latency

Table 3.1: Baseline CPU & GPU Configuration

CPU	**Intel Xeon E5-2697 v3**
Base Frequency	2.6 GHz
Cores/Threads	14/28
Process	22 nm
TDP	145 W
Cache	32 kB i-L1 per core, 32 kB d-L1 per core, 256 kB L2 per core, 35 MB shared L3
System Memory	64 GB DRAM, DDR4

GPU	**Nvidia Titan Xp**
Frequency	1.6 GHz
CUDA Cores	3840
Process	16 nm
TDP	250 W
Cache	3MB shared L2
Graphics Memory	12 GB DRAM, GDDR5X

of transferring data between DRAM and LLC, as well as on the LLC inter-slice ring. To model this, we develop a micro-benchmark in C measuring the latency of loading data from DRAM to the CPU. The micro-benchmark loads data to the exact cache sets as the data loading for in-cache computation. Note that for input loading, the data are loaded from one reserved LLC slice, so we exclude the DRAM bounded time reported by Intel VTune. We use RAPL tools (David et al., 2010) to measure the CPU energy of the micro-benchmark, for estimating the data loading energy.

Proposed architecture: The base CPU architecture for in-cache computing is the same as used in Neural Cache, as described in Section 2.2.2. For sparsity-aware pruning, we develop a tool-chain in TensorFlow. The cycle-accurate simulator is used to model our proposed sparsity and low-precision architecture with proper modifications. The latency, energy and area overheads of the proposed circuit are modeled by synthesis. For the Coalescing Unit, we use IBM 45 nm soi12s0 cell library and Synopsys Design Compiler for modeling the Reconfiguring Peripheral. With scaling to 22 nm, the area of each RP is 0.05 mm^2. We conservatively do not scale for the power and latency estimation. The estimated power of each RP, including leakage power and dynamic power, is 325 mW. The RP has a latency of 29.2 ns, which is equivalent to 73 cycles at 2.5 GHz. The 256×256 crossbar is modeled conservatively at 28 nm. Each crossbar is 0.032 mm^2, and it transmits 256 input bits in a latency of 163 ps and an energy of 49 pJ. In total, the 8 Coalescing Units in the reserved way translate to less than 7% area overhead to a cache slice and 1.9% area overhead to the processor die.

CNN models: We use AlexNet (Krizhevsky et al., 2012) and Inception v3 (Szegedy et al., 2016) as the network model for evaluating performance and accuracy. For evaluating the performance of CPU and GPU baseline, the network weights are floating point numbers (single-precision for CPU, half-precision for GPU) because the quantized network has worse performance on CPU and GPU platforms due to lack of optimized software libraries. To fairly compare with CPU/GPU baselines, we apply state-of-art pruning techniques of Scalpel (Yu et al., 2017) - a pruning method for CPUs and GPUs. For evaluating the accuracy of Neural Cache baseline and sparse architecture, we perform 8-bit linear quantization on weights and inputs, based on the dynamic range of all the weights/inputs in one layer. Note that for the sparsity architectures, the quantization happens after pruning and fine-tuning the network. Quantization for ternary network is done with the approach described in WRPN (Mishra et al., 2018), where the weights are first clipped by the range [-1, 1] and then linearly quantized. For binary network quantization, we follow the approach in DoReFa-Net (Zhou et al., 2016).

3.4 Results

3.4.1 Latency

Figure 3.5 shows the latency of all the convolutional layers of the AlexNet with corresponding configurations. The CPU and GPU baseline have significantly higher latencies of 6.89 ms and 1.43 ms. All the accelerator architectures benefit from in-situ computation and a large

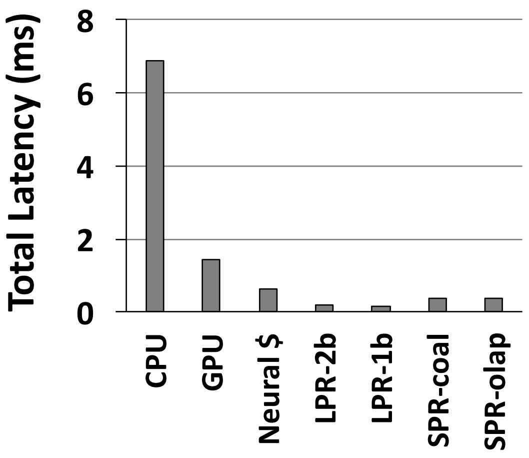


Figure 3.5: Total Convolution Latency Across Layers.

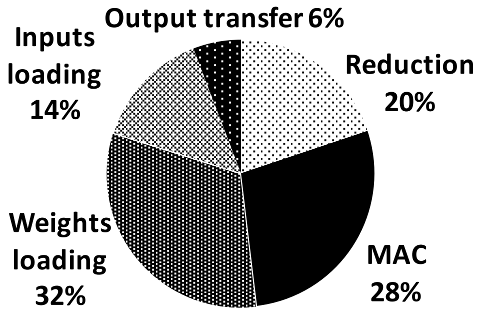


Figure 3.6: Latency Breakdown of SPR-Coal Configuration.

number of SIMD slots. The Neural Cache baseline is 0.619 ms. The LPR configurations achieve the best latency at the cost of accuracy loss, 0.199 ms and 0.158 ms for ternary and binary models (3.1× and 3.9× speedup over N$-base). This is because bit-serial computation scales with bit-width intrinsically. The SPR-coal and SPR-olap achieve 0.375 ms and 0.390 ms of latency.

Figure 3.7 compares the layer latency (break down into 5 stages) of SPR/LPR configurations with Neural Cache. For SPR configurations, we see a significant drop in weight loading time, thanks to the reduced size of total weights. The MAC time goes down because the number of serial computation required is smaller. The gain of input loading time savings is reduced by the large input size of the first layer, where pruning is not applied for maintaining accuracy. The reduction time goes down slightly as expected, where the benefits of fewer

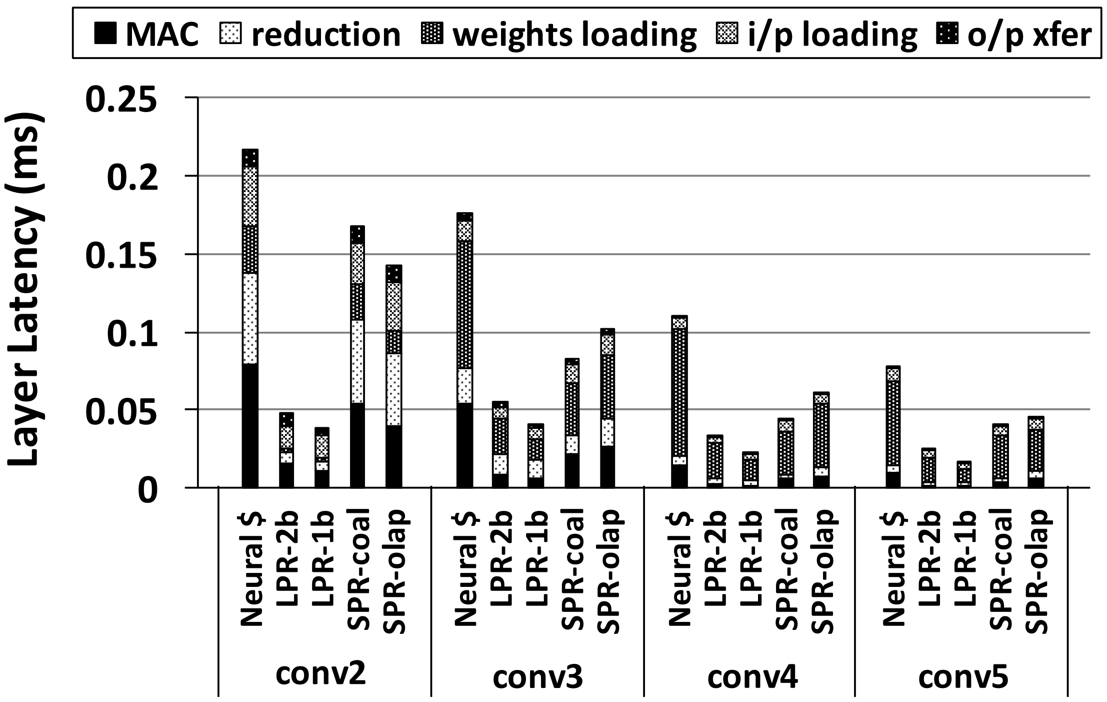


Figure 3.7: Latency of Each Convolutional Layer.

channels are offset by the overheads of the preparing round computation and mask loading.

The performance is also affected by the parameters of layers. The layers with larger filter sizes, more operations and smaller input data sizes enjoy a better speedup since the weight loading and MACs can be better accelerated. Also, a higher pruning rate leads to a higher speedup for SPR configurations.

Figure 3.6 further breaks down the overall latency of SPR-coal. The computation (MAC and reduction) consists about half of the latency, with 28% for MAC and 20% for reduction. The weight loading also takes up a significant portion (32%), which is in part due to the long latency of DRAM communication. The latency for all the convolutional layers and the relative speedup over CPU baseline, for Inception v3, are shown in Table 3.4.

3.4.2 Accuracy

An overview of accuracy vs. performance for AlexNet under all configurations is presented in Figure 3.8. The top-1 accuracy on the validation set is chosen as the accuracy metric,

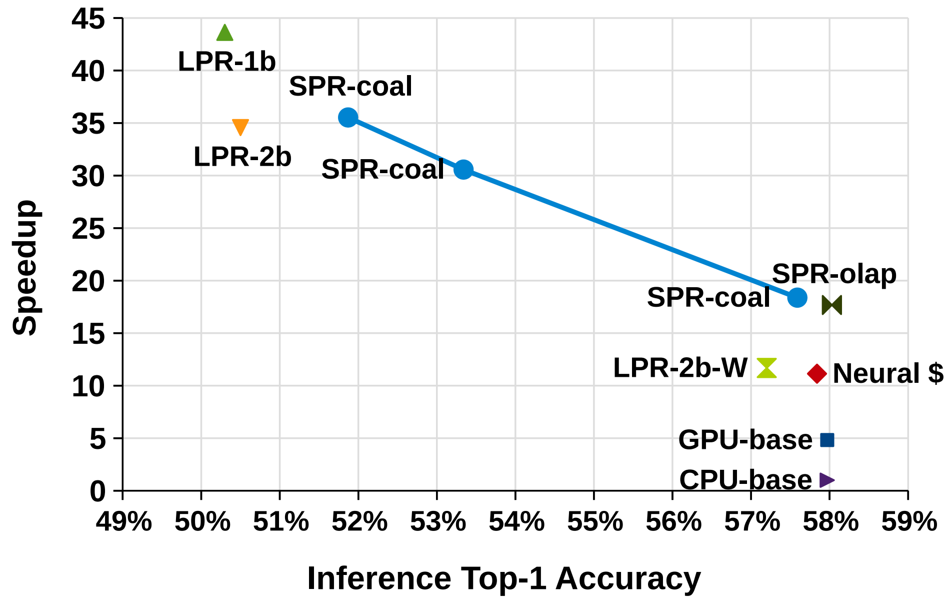


Figure 3.8: Accuracy vs Performance for AlexNet.

and performance is quantified as the speedup over baseline CPU. The CPU baseline has an accuracy of 57.97% and N$-base is 57.84% with *8-bit* quantized weights and activations. The SPR configurations may achieve zero (SPR-olap, 58.03%) or less than 0.5% accuracy loss (SPR-coal, 57.59%). There are three connected data points for SPR-coal on the graph, which is achieved by varying the sparsity at structured pruning: the two points with higher sparsity have the accuracy of 53.34% and 51.87%. Note that the SPR configurations are quantized to 8-bit weights and activations as described in experiment methodology.

Overall, SPR methods can maintain minimal accuracy loss while offering up to 18.4× speedup over CPU baseline. SPR-coal also enables flexible trade-off between accuracy and performance. The point SPR-coal (51.87%, 35.5×) has better speedup than LPR-2b, also with a 1.37% higher accuracy. LPR-1b achieves the highest speedup (43.6×), at the cost of a 7.67% accuracy loss to CPU-base. If future research improves the accuracy for LPR models, the LPR data points will shift right in the figure, making them a more competitive option. The LPR-2b-W data point is experimented with the same configuration with LPR-2b except that the channel number is doubled for all the layers to improve accuracy, as described in WRPN (Mishra et al., 2018). Although the accuracy is close to the baseline,

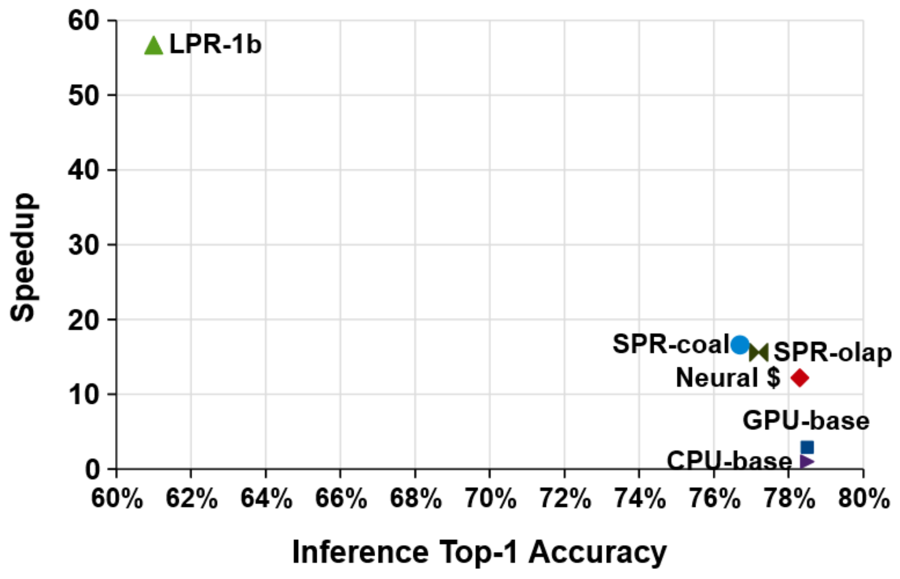


Figure 3.9: Accuracy vs Performance for Inception V3.

the speedup is lower than SPR configurations because doubling channels incurs overheads to all stages. For LPR-2b-W, the widening of input channels C by 2× also requires widening of output channels M by 2×. The number of weights is proportional to $C \times M$, so it increases by 4× with widening, offsetting the 4× reduction from 8-bit weight baseline to 2-bit LPR configuration. The time for ternary weight MAC does improve but in reduction stage the total number of channels increases by 4×. Overall the speedup of LPR-2b-W is worse than SPR and 5% better than Neural Cache baseline.

The results of the trade-off between accuracy and acceleration ratio with Inception V3 are shown in Figure 3.9. The SPR-olap method still achieves a 1.3× speedup over Neural Cache, with a 1.1% of accuracy loss.

3.4.3 Energy

Figure 3.10 plots the estimated energy consumption of convolutional layers of AlexNet for all the configurations. Baseline CPU has the highest energy at 0.512 J. N$-base consumes 0.024 J. LPR-1b consumes the least energy at 0.007 J. SPR-coal and SPR-olap have the

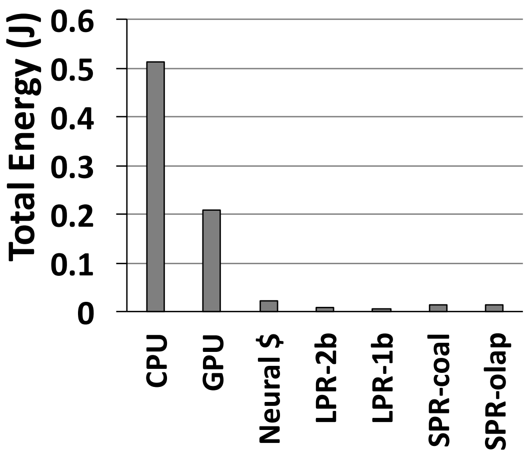


Figure 3.10: Total Energy for Convolution.

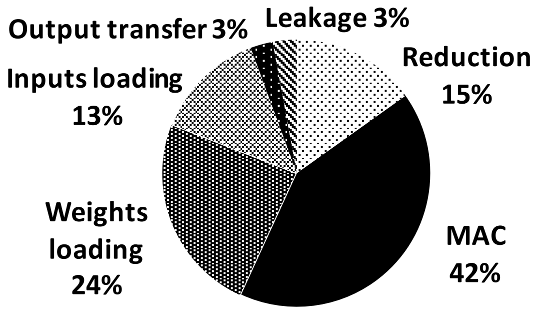


Figure 3.11: Energy Breakdown of SPR-Coal Configuration.

energy consumption of 0.0151 J and 0.0150 J, respectively. The layer-by-layer energy consumption is plotted in Figure 3.12. Leakage indicates the background leakage energy on CPU and DRAM. In N$-base the most significant portion is MAC, and the LPR and SPR both successfully reduce MAC cycles and thus energy. LPR-1b is 3.4× more energy efficient than N$-base while SPR-olap achieves 1.59× improvement. Figure 3.11 shows the energy breakdown of SPR-coal, where MAC dominates energy consumption at 42%. Less than 40% of energy is spent on data movement. Table 3.4 shows the energy consumption of all the convolutional layers for Inception v3 and the energy efficiency improvements over the CPU baseline.

The savings in energy can be attributed to the reduced amount of necessary computation and data movement, with the removal of redundant filters. The energy trend is similar to

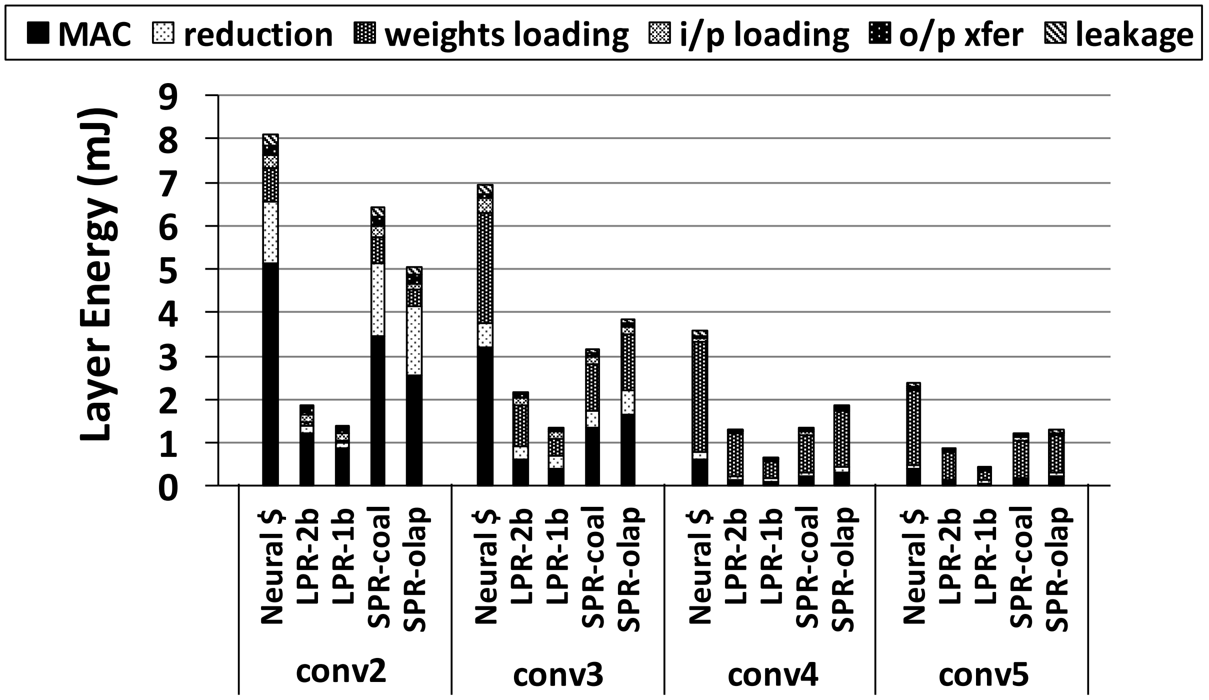


Figure 3.12: Energy of Each Convolutional Layer.

the latency, since at computation the SIMD slots are highly utilized.

We further propose the following techniques to reduce energy. First, based on the observation that most of the unpruned 8-bit quantized weights are centered around the value 128, we can use fewer bits to encode the weight values that are close to 128. In MAC, for those encoded weights, it takes fewer cycles since the bit width is reduced (for an n-bit weight, the multiplication needs 11n-2 cycles). Then, the input activations corresponding to encoded weights are added up and then shifted to correct the offset in weight encoding. Second, during reduction, the column peripherals of the bitlines that do not have active partial sum data can be turned off. For example, in the addition phase of the first reduction round, half of the total bitlines do not need activating. Third, we can power-gate the unused CPU cores to save static energy. It is sufficient to leave one core running to control the Neural Cache operations, while other cores are power-gated.

With zeros in the activations, there can be additional savings by selectively disabling the bitlines at the write-back cycle of the in-SRAM compute. The zeros in the activations can

Table 3.2: Pruning Rate of SPR-coal and SPR-olap

Layer	conv2	conv3	conv4	conv5	Overall
SPR-coal	27%	60%	55%	42%	50%
SRP-olap	50%	50%	50%	50%	49%

Table 3.3: Model Size Comparison

Neural $	LPR-2b	LPR-1b	SPR-coal	SPR-olap
2285 kB	602 kB	321 kB	1147 kB	1163 kB

be detected by sensing the shared bitline: for an 8-bit activation, for instance, it is cheap to sense in analog whether the AND of the 8 bitlines is zero. If so, then this activation must be zero and a mask bit is set to skip the computation on this bitline at the MAC stage. This can lead to an additional 8% of the total energy savings on average, which are not included in the reported results.

3.4.4 Model Size

In Table 3.3, the total sizes of the pruned/quantized models for AlexNet are compared. For low-precision models, the model size decreases linearly with the bit width. For pruned models, the model size is determined by the pruning rate (summarized in Table 3.2), with a small overhead of mask vectors. The SPR-coal and SPR-olap achieve 49.8% and 49.1% pruning rate, respectively. The model size of LPR-2b is reduced by 4× and LRP-1b by 8×, which is the direct result of using low numerical precision of weights. For Inception v3, in both SPR-coal and SPR-olap, 50% of weights are pruned for the layers we prune.

3.4.5 Comparing to ASICs

To demonstrate the efficiency of this work over other accelerators, in Table 3.5, we perform a comparison with recent works on sparsity-aware DNN accelerators. Eyeriss (Chen et al., 2017) is a low-power DNN accelerator. Cambricon-X (Zhang et al., 2016) optimizes for sparse weights. SCNN (Parashar et al., 2017) optimizes for both sparse weights and activations. We

Table 3.4: Efficiency of Inception v3 Model

Architecture	latency (ms)	energy (J)	top-1 accuracy	speedup	relative energy efficiency
CPU	56.8	6.14	78.5%	1.0	1.0
GPU	19.9	2.73	78.5%	2.9	2.2
Neural $	4.66	0.18	78.3%	12.2	35.2
SPR-coal	3.43	0.12	76.7%	16.6	50.1
SRP-olap	3.64	0.13	77.2%	15.6	45.8
LPR-1b	1.00	0.03	61.0%	56.7	183.6

Table 3.5: Comparison with ASICs. Area scaled to 22 nm.

Architecture	Area (mm^2)	Latency (ms)	Inferences /($s{\cdot}mm^2$)	Inferences /($s{\cdot}mm^2{\cdot}J$) - with DRAM energy
SPR-coal	9.8	0.38	272	17,977
SPR-coal (w/o RP)	4.2	0.47	511	29,242
Eyeriss (Chen et al., 2017)	1.4	115.3	24	1,511
Cambricon-X (Zhang et al., 2016)	0.7	4.97	275	23,226
SCNN (Parashar et al., 2017)	14.9	0.76	88	N/A

use the SPR-coal configuration in comparison. The area overhead includes the modifications required by Neural Cache and the additions for this work. To optimize for area efficiency, we propose another design, where the crossbar values of the Coalescing Unit are directly loaded from DRAM instead of being generated from the Reconfiguring Peripheral. This design incurs a higher latency, as shown in the table. We do not synthesize or layout other ASICs. Instead, the area overheads are scaled to 22 nm process node, according to the relationship between area and technology node for recent processors (Huang et al., 2011). The energy and frequency are as reported in the original papers and not scaled. We compare the throughput under the unit area and energy. The results are shown include energy consumed by DRAM. The latencies of Cambricon-X and SCNN are not directly provided, so they are estimated based on the cycle count and frequency. The accelerator energy is estimated based on the reported average power and the estimated latency. The energy of SCNN is not estimated because the average power for SCNN was not reported in the paper. The DRAM energy of Eyeriss and Cambricon-X is estimated based on the DDR4 power model of Micron (Micron Technology Inc., 2017).

Compared to the other ASICs, our design has the shortest latency thanks to the high parallelism. The area-optimized design has a small increase in latency because the coalescing configuration is read from DRAM, but it still outperforms other accelerators in terms of throughput per area. Comparing the throughput per area per energy (not including DRAM), our design is within 1.8× of Cambricon-X. However, when DRAM energy is included, our design is better in this metric than Cambricon-X. In contrast to the in-memory architecture of this work, Cambricon-X has a small on-chip buffer, and hence requires a lot of off-chip data transfer between DRAM and the accelerator chip. Our architecture achieves the stated improvements based on minor modifications to a general purpose processor, without requiring a dedicated ASIC.

3.5 Summary

Our proposed in-cache architecture improves latency and energy efficiency by skipping sparse weights with two schemes – coalescing and overlapping. In coalescing, the effective input channels are coalesced from the original activation map efficiently by the Coalescing Unit. Therefore the vector slots will not be wasted on ineffective computation. In overlapping, the filters are overlapped with each other where the non-zero channels do not conflict. Hence, the utilization of vector units is increased due to more filters being convolved with inputs at the same time. We also develop novel bit-serial MAC algorithms for low-precision CNNs with binary and ternary weights. Overall the proposed methods improve speed and energy, as well as offering the speed-accuracy trade-off. We achieve a 1.6× speedup over the original in-cache DNN accelerator with no loss of top-1 accuracy for AlexNet, with 2% area overhead of a CPU processor.

CHAPTER 4

DNN Acceleration with Compute-Capable BRAM

In the previous chapter we discussed how to design an efficient in-SRAM DNN accelerator based on model compression techniques. While the proposed method supports different CNN architectures, the underlying hardware is fixed. In this chapter, we focus on an approach to enable designing DNN accelerators with reconfigurable hardware, such that more computing kernels can benefit from customized hardware acceleration, and also the architecture can be overall more future-proof. More specifically, we propose modifications on the Block RAM (BRAM) on the FPGA to enable in-memory computing for kernel computation of DNNs in BRAMs, and evaluate the overhead of such modifications (Section 4.1). Then we demonstrate how the compute-capable BRAMs can be used in tandem with other logic blocks and specialized-computing blocks in FPGAs to build a reconfigurable accelerator for DNNs (Section 4.2), where we discuss the details of how to perform efficient arithmetic operations on different data formats, and also how to accelerate the dot-product computation kernel. We show how the reconfigurability of FPGAs helps with accelerating different model variants by an example for RNN with design space exploration (Section 4.3). The evaluation methodology is described in Section 4.4 and the results are shown in Section 4.5.

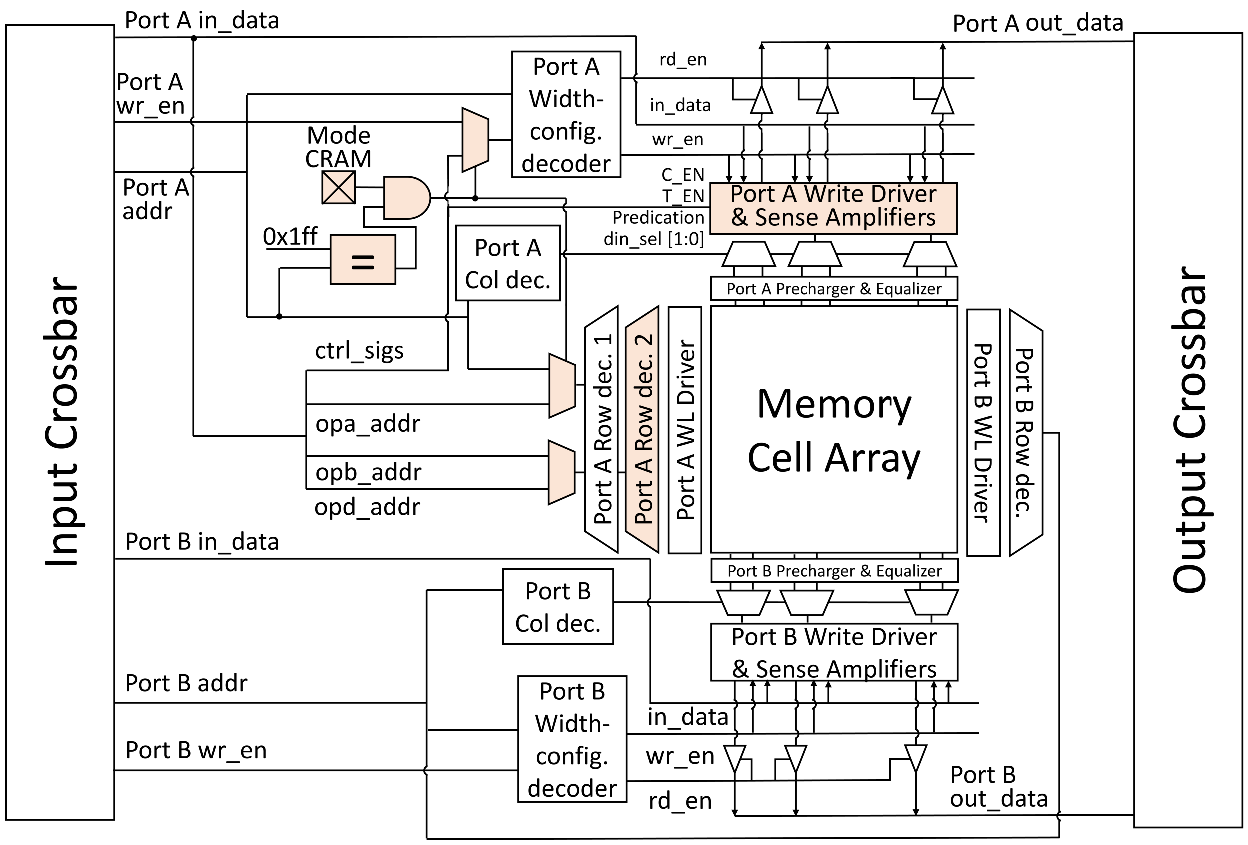


Figure 4.1: Block diagrams for internal architecture of the compute-capable dual-port single-bank BRAM. The added/modified circuitry is highlighted in color.

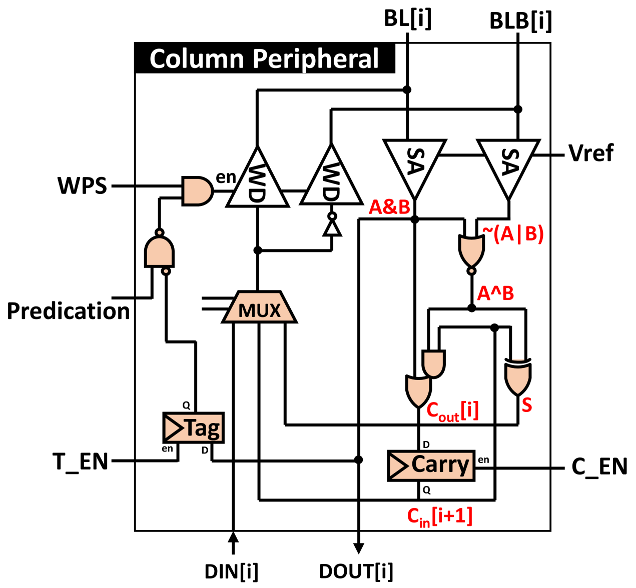


Figure 4.2: Block diagrams for peripheral logic circuitry added for each bit-line of the memory cell array. The added/modified circuitry is highlighted in color.

4.1 Compute-Capable Block RAMs for FPGAs

4.1.1 Enhanced BRAM Architecture

Fig. 4.1 shows the proposed BRAM architecture following the dual-port 1-bank SRAM array design from Yazdanshenas et al. (2017). Blocks added or modified to enhance the FPGA BRAM with the in-memory compute capabilities are highlighted in a different color. One of our main design targets is to implement all the necessary changes while reusing the same existing BRAM interface (i.e. keep the same number of input/output ports) to minimize the area overhead of the proposed changes and avoid stressing the routing from/to the BRAMs. Therefore, as highlighted in Fig. 4.1, all the changes in the proposed BRAM architecture are limited to the block's internals.

The two major changes are the addition of: (1) a second row decoder to one of the two BRAM ports to allow the simultaneous activation of two memory cells that share the same bitline and (2) the bit-serial compute logic to the peripherals of each bitline. Fig. 4.2 shows the circuitry of the modified bitline peripheral. On the read path, the outputs of the sense amplifiers (SA) go through a few logic gates to create the sum and carry bits for adding the two activated memory cells. The carry bit is stored in the carry flip-flop (FF) to be used in the following cycle's computation. The sum bit is routed to the write path to be stored back to the memory array in the location reserved for the results. The output of the SA can be passed directly to the output for a normal read operation, or stored in the tag FF used in the bit-serial multiplication operation. On the write path, a 3:1 multiplexer is added to choose between different sources for a memory cell write: input data bit (for a normal write operation), sum bit (for addition), or carry bit (for the final carry out value). We also add logic circuitry to allow enabling/disabling the write drivers (WD) based on the tag value, which is key for implementing the bit-serial multiplication algorithm described in Section 2.2.2. Our SPICE simulations using 28 nm process technology at 0.9V supply voltage show that the added circuitry results in a 1.6× increase in the BRAM cycle time only when it is operating in the compute mode. In the memory mode, the 3:1 multiplexer in Fig. 4.2 is the only circuitry added on the normal write path. However, this added negligible delay and did not affect the BRAM speed in the conventional memory mode.

The proposed compute-capable BRAM can function in the conventional *memory mode* or the new *compute mode*. The mode of operation is determined by an additional configuration SRAM (CRAM) cell for each BRAM tile, which is configured during programming the FPGA. These added configuration bits represent a negligible 12 Kb increase to the 577 Mb compressed bitstream of the largest Stratix 10 device. When the compute-capable BRAM is configured in memory mode, its operation is exactly the same as that of a conventional BRAM and the designer can flexibly configure the number of ports and the width/depth of the BRAM. On the other hand, when configured in compute mode, the BRAM is automatically configured to its maximum width to maximize the read/write throughput for populating the memory array with input data and reading the final results in transposed format. In addition, a special address (`0x1ff`) is reserved; any data word written to the reserved address is treated as an in-memory compute instruction, while accesses to other addresses are processed normally. In this case, the row address, the SRAM array access control signals (including `wr_en`), and the bitline peripheral control signals are all packed into the incoming instruction. To support that, we insert several multiplexers (shown in Fig. 4.1) to select the set of right control signals based on the operation mode, and a comparator to determine if the input address is equal to the reserved one. These instructions are generated by a control finite-state machine (FSM) implemented in the soft logic and supplied to the `in_data` port of the BRAM. The control FSM is lightweight; it utilizes less than ∼300 look-up tables (LUTs), which can be amortized by sharing the FSM across multiple BRAMs working in lockstep.

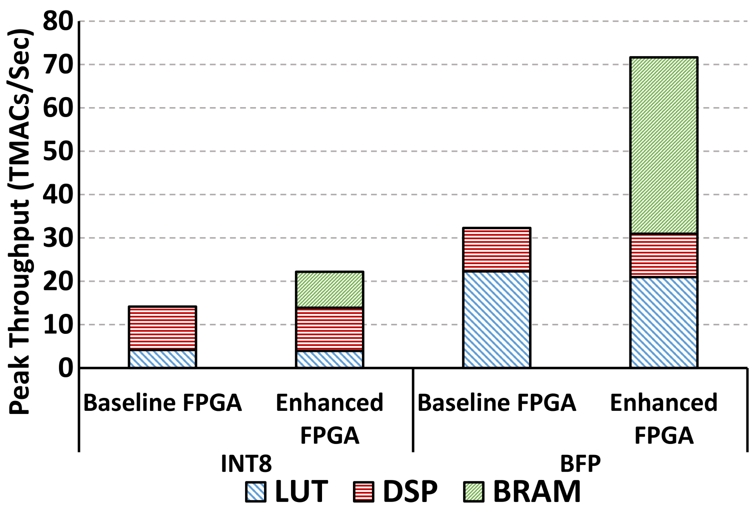


Figure 4.3: Peak MAC throughput gains when enhancing a baseline Stratix 10 GX2800 FPGA with compute-capable BRAMs for 8-bit integer and block floating-point precisions.

4.1.2 Peak MAC Throughput Gain

In this subsection, we evaluate the peak MAC throughput gains achieved by enhancing the largest Stratix 10 (GX 2800) FPGA architecture with our proposed compute-capable BRAMs. This study gives the workload-agnostic performance gains before evaluating the achievable gains with real benchmarks. We experiment with two numerical precisions: 8-bit integer (INT8) and block floating-point (BFP8) with 1 sign bit, 2 mantissa bits, and 5 exponent bits (1s.2m.5e), which is used in various DNN workloads (Fowers et al., 2018). For the INT8 experiments, we assume 27-bit accumulation as in Nurvitadhi et al. (2019).

To calculate the peak throughput of the soft logic, we synthesize, place and route one MAC unit to LUTs to determine its operating frequency and resource utilization. We then calculate the total LUT throughput by optimistically assuming that we can fill all the available LUTs with MAC units at the same operating frequency. This is an unrealistic assumption as it does not consider the routability and frequency degradation as we fill the device, but serves the purpose of this peak throughput study. In addition, we run simple Quartus experiments to determine the the maximum operating frequencies for DSP blocks in the $2\times$ integer MACs mode and BRAMs in the simple dual-port mode, which are found to be 866 MHz and 998

MHz, respectively. This means that the BRAMs would run at a maximum frequency of 624 MHz when configured in the compute mode (1.6× slower than memory mode). A single MAC operation implemented using the in-BRAM bit-serial algorithms takes 113 and 23 cycles in case of INT8 and BFP8, respectively.

Fig. 4.3 shows the peak throughput in TMACs/sec for the baseline and enhanced Stratix 10 FPGA architectures for the two studied precisions. It shows that compute-capable BRAMs can deliver an additional **8.3 INT8 TMACs/sec** and **40.7 BFP8 TMACs/sec**, enhancing the peak device throughput by a factor of **1.6**× and **2.3**× for INT8 and BFP8, respectively. These gains come with minor degradation to the contribution of LUTs to the peak device throughput since, with every 64 BRAMs sharing the same control FSM, all the compute-capable BRAMs in the device utilize ∼55k LUTs (less than 6% of the device resources).

4.1.3 Area Overhead and CAD Support

The area overhead due to the added row decoder and column peripherals for bitline computing is estimated to be 7.5% for a 64 Kb SRAM array in 28 nm process technology (Eckert et al., 2018). This was then verified by a fabricated prototype chip in Wang et al. (2019a). Since the BRAM's interface to the programmable routing is not changed at all when adding the in-memory compute capabilities, we simply rely on the data produced by the COFFE automatic transistor sizing tool for FPGA circuitry (Tatsumura et al., 2018) to quantify the FPGA BRAM tile area overhead. According to this data, a 64 Kb SRAM array in an FPGA BRAM tile is $11,016\ \mu m^2$ in a similar 22 nm process technology and therefore the area of added compute circuitry would be $826\ \mu m^2$ for its 256 wordlines and 256 bitlines. Since the added circuitry size scales linearly with the number of wordlines (the extra decoder) and bitlines (the column peripherals), then the overhead for an M20k block in a Stratix 10 architecture with 128 wordlines and 128 bitlines (excluding ECC bits) would be $413\ \mu m^2$. This represents a **7.4% increase in the BRAM tile area** according to the data generated by

COFFE (Tatsumura et al., 2018). With BRAMs occupying ~25% of the die size of modern FPGAs with traditional compositions (Tatsumura et al., 2016), this overhead corresponds to only **1.8% increase in the FPGA die size**.

In addition, enhancing BRAMs with in-memory compute capabilities does not require any significant CAD support. In an FPGA design tool like Quartus, an RTL designer usually instantiates a memory block from the library of vendor-supplied IPs (Int, 2020). For compute-capable BRAMs, a new IP is added to the IP catalog that maps to one or more BRAMs. The designer only needs to instantiate the new IP wherever necessary in the design. Then, the synthesis engine can directly map the instantiated IP to BRAMs and implements its associated control FSM in soft logic.

4.2 RIMA: DNN Reconfigurable In-Memory Accelerator

In this section, we introduce our DNN reconfigurable in-memory accelerator, or RIMA for short. RIMA utilizes our proposed compute-capable BRAMs to achieve higher DNN inference throughput. It also exploits the FPGA reconfigurability by customizing the workload balance between compute-capable BRAMs and DSPs to further optimize performance for each workload.

4.2.1 Target DNN Workloads

RIMA is targeted for accelerating recurrent neural networks (RNNs). These networks process sequence inputs such as speech samples or sentences and are typically used in natural language processing and machine translation. They consist of multiple matrix-vector multiplications followed by vector operations known as *gates*. Different variations of RNNs include vanilla RNNs, gated recurrent units (GRUs), and long short-term memories (LSTMs) with 2, 6, and 8 vector-matrix multiplications per time step. The multiple matrix-vector multiplications typically consume the majority of the compute time of an RNN. With no data dependencies between the matrix-vector multiplications of the same time step, they can be combined into a larger matrix-vector multiplication, in which the input is a vector of C elements, the weight matrix is of size $R \times C$, and the output is a vector of R elements.

4.2.2 Accelerator Architecture

Our accelerator adopts a similar architecture as that of the Brainwave neural processing unit (NPU), a state-of-the-art FPGA overlay for DNN acceleration (Fowers et al., 2018; Nurvitadhi et al., 2019). However, RIMA has two key differences compared to the NPU: (1) it utilizes compute-capable BRAMs as massively parallel SIMD lanes in the matrix-vector multiplication engine, and (2) it customizes its architecture parameters and instruction sequences for each specific workload instead of being a *one-size-fits-all* software programmable overlay. In this work, we apply per-workload architecture customization to maximize the overall performance by balancing between in-BRAM and DSP compute. Fig. 4.4a illustrates the top-level organization of our accelerator. It consists of five pipeline stages: the matrix-vector multiplication unit (MVU), the external vector register file (eVRF) for skipping the MVU when necessary, two identical multi-function units (MFUs) for vector elementwise operations (e.g. activation, addition, subtraction), and finally the loader (LD) which writes back to any of the VRFs. RIMA uses the same eVRF, MFUs, and LD blocks as in Nurvitadhi et al. (2019), but re-designs the MVU (the key compute complex) to exploit the compute-capable BRAMs in our proposed enhanced FPGA architecture.

The MVU, as illustrated in Fig. 4.4a, consists of T tiles followed by an inter-tile global reduction tree to generate the final MVU output. As shown in Fig. 4.4b, each tile consists of dot product engines (DPEs) that compute dot product operations between a portion of the input vector and several rows of the weight matrix. There are two types of DPEs in each tile; the logic DPEs (L-DPEs) that implement an array of multipliers and an adder tree to accumulate the products using LUTs and DSP blocks, and memory DPEs (M-DPEs) that

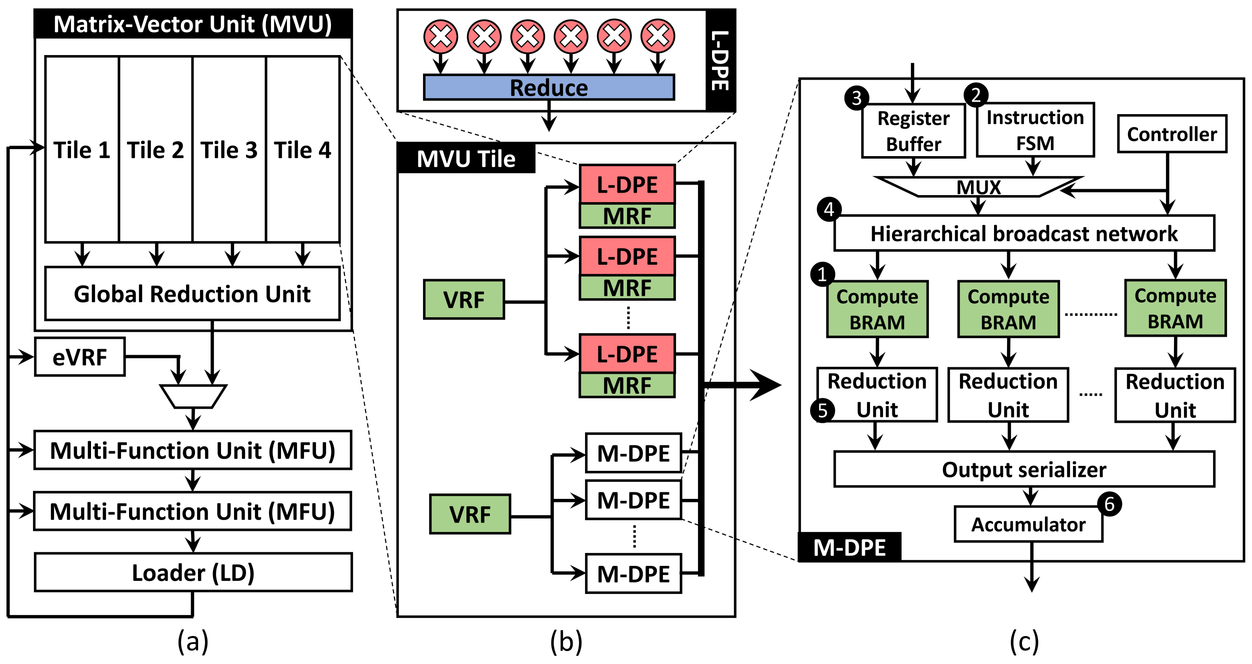


Figure 4.4: Block diagrams showing: (a) overview of the RIMA architecture, (b) our proposed hybrid MVU tile, and (c) the architecture of an M-DPE, with compute-capable BRAMs.

perform the same operation using compute-capable BRAMs. The two sets of DPEs work in tandem to fully exploit the computational resources in our proposed FPGA architecture with in-BRAM compute capabilities. To further optimize performance, the M-DPEs are designed such that they belong to a different clock domain than the rest of the architecture, since the M-DPEs can perform in-memory computations at a much faster clock speed due to their reduced routing utilization and simple control logic. The values of the weight matrix used by each L-DPE are pinned in a tightly coupled matrix register file (MRF) implemented using conventional memory-mode BRAMs. The M-DPEs do not need an MRF since the data is stored and processed in-place in the compute-capable BRAMs. Additionally, each tile has two VRFs (instead of one in the baseline NPU) that store the same part of the input vector in different layouts and supply inputs to the L-DPEs and M-DPEs at different rates due to their different compute styles.

4.2.3 Memory Dot-Product Engines

Fig. 4.4c depicts the internal architecture of an M-DPE. The core of the M-DPEs is an array of compute-capable BRAMs (❶). Unlike an L-DPE that computes a single dot product operation between part of the input vector and a single matrix row, multiple dot product operations with multiple matrix rows are mapped to an M-DPE to maximize the degree of parallelism on the BRAM bitlines. Each M-DPE has one Instruction FSM (❷) that sequentially generates the required bitline compute instructions. Instructions are broadcast to all the BRAMs in an M-DPE, as they all act as SIMD lanes executing the same instructions in lockstep. We implement a set of registers (❸) that buffer a complete data word coming from the VRF while the controller sequentially loads the appropriate portions of it to the input-reserved locations in the BRAMs. We use a pipelined tree interconnect network (❹) to broadcast instructions and input vector values to all BRAMs in an M-DPE without stressing the FPGA routing with such high fanout connections. This broadcast network is time-shared between the register buffer and instruction FSM, as the data loading and computation never occur simultaneously. The output of each BRAM is fed to a reduction unit (❺) where the results on multiple bitlines are added together as detailed later in Section 4.2.4. Finally, the results from all reduction units are serialized and passed to an accumulator (❻) that performs any necessary additions across different BRAMs to produce the final results of the M-DPE.

4.2.4 Transposed Integer Arithmetic

We discuss our implementation of the dot product operation of two vectors in compute-capable BRAMs for INT8 and BFP formats in this subsection and the next, respectively.

4.2.4.1 Data Mapping

In the transposed data mapping, each INT8 value is mapped to 8 memory cells attached to the same bitline (i.e. across 8 consecutive memory words), and different values in an operand vector are mapped to consecutive bitlines. Once all bitlines in the SRAM array are filled, the remaining values in the operand vector can be mapped to another set of 8 wordlines. The elements of the other operand vector are mapped similarly such that each two corresponding elements (i.e. elements in the same position) of the two vectors are mapped to memory cells on the same bitline. For example, for a memory cell array with 128 bitlines and a dot product between input and weight vectors of length 256 elements, the first 128 weights are mapped to all bitlines of wordlines 1-8 and weights 129-256 are mapped to wordlines 9-16. Similarly, the 256 inputs are mapped to word-lines 17-32.

4.2.4.2 Operand Loading

Before any computation starts, the weight matrix and input vector need to be loaded to the memory array. The weights remain unchanged in the on-chip memory during inference. Therefore, they can be transposed and loaded into the memory array offline in a preprocessing step. On the other hand, the input values are supplied by the user in the MVU VRFs and loaded to the compute-capable BRAMs from the register buffer (see Fig. 4.4c). In our implementation, the register buffer holds 32 consecutive values in the input vector (one VRF word) that, when loaded to the compute-capable BRAM, goes through an 8:1 bit-slice selector before broadcasting. Then, the selected bit slice of the 32 values maps to 32 cells on consecutive bitlines and the same wordline.

4.2.4.3 In-BRAM MAC & Reduction

After loading the operands into the BRAMs, the corresponding weights and inputs are multiplied with the bit-serial algorithm, and the products are accumulated in each bitline with bit-serial addition. After the MAC, the accumulated partial sums in all bitlines are iteratively reduced until there are 16 partial sums left. Performing further in-BRAM reduction significantly degrades the BRAM compute throughput as only a successively smaller portion of the BRAM bitlines are actively performing compute during each reduction iteration. We

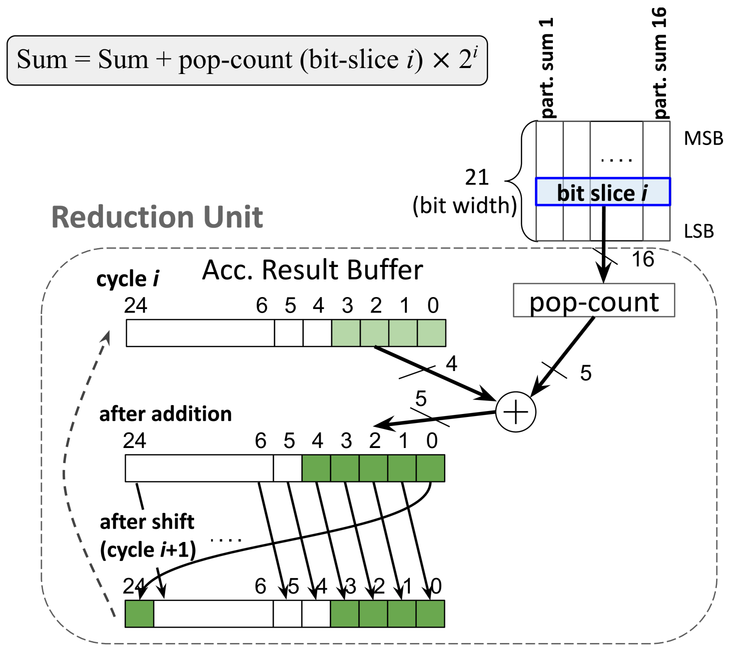


Figure 4.5: External reduction unit for summing up 16 partial results. Every cycle a bit slice from all values is accumulated with pop-count, addition, and shift.

empirically choose to reduce down to 16 partial results as this design point achieves a good balance between the BRAM throughput and the resources of the external reduction unit.

4.2.4.4 External Reduction

Then, the 16 remaining partial results are read out from the BRAM, one bit slice at a time (starting from the least significant bit), and sent to the external reduction unit shown in Fig. 4.5 to obtain the final sum. The formula for accumulating with bit-slices is $\sum_{i=0}^{N} \text{pop-count}(\textit{bit-slice } i) \times 2^i$, where N is the bitwidth of the partial results. At each cycle, 16 bits from the same position of 16 different values first go through pop-count logic (counting the number of 1's). Then, the pop-count result is left-shifted by i bits and added to the accumulated sum. Practically, a barrel shifter (i.e. with variable offset) is expensive to implement on an FPGA, so we designed the external reduction unit using a circular shift register instead, as shown in Figure 4.5. In each cycle, the pop-count result (5 bits representing a value between 0 and 16) is added to the last 4 bits of the current accumulation

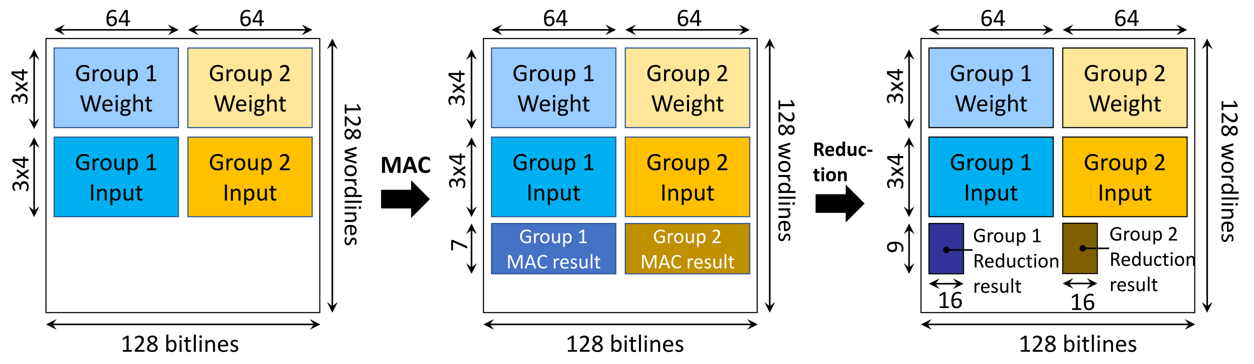


Figure 4.6: In-memory operations for dot-product with block floating-point in a compute-capable BRAM. Vector has 512 elements and is partitioned into 2 exponent-sharing groups in BFP.

value. After addition, the bits are circularly right-shifted by 1 bit to prepare for the next cycle addition. This addition-and-shift operation repeats until the most significant bit slice is processed. The external reduction unit is pipelined with the addition in the last iteration of the in-BRAM reduction to hide its latency. The external reduction unit reads out and operates on bit slice i, while the in-BRAM reduction generates bit slice $i + 1$ at the same time since the FPGA BRAMs have separate read and write wordlines.

4.2.4.5 Bit Slicing

We further implement bit slicing, an optimization for achieving higher utilization of bitline SIMD lanes. First, each weight/input value is partitioned into multiple *slices* (e.g. an 8-bit value is split into higher 4 bits and lower 4 bits). Then, different slices are mapped to different compute-capable BRAMs to extract a higher degree of parallelism and reduce the bit-serial processing latency. Finally, the results from different slices are shifted to the appropriate bit position and summed up in the soft logic to produce the combined results.

4.2.5 Transposed Block Floating-Point Arithmetic

The BFP data format is mainly used in DNN acceleration due to its computational efficiency and adequate accuracy (Fowers et al., 2018; Aydonat et al., 2017; Drumond et al., 2018).

Unlike the traditional floating-point format, where each value has its own sign, mantissa and exponent, the BFP format shares the same exponent across a *block* of values. Therefore, to compute the dot product of two BFP vectors, only the mantissa bits need to be multiplied and accumulated as integer values, while the new shared exponent is calculated based on the shared exponents of operands.

As the mantissa of each BFP value has fewer bits, multiple weights and one input can be mapped to each bitline, so that the input can be reused for multiplying with different weights. In addition, each memory array is partitioned into groups of bitlines, and the elements with a shared exponent are mapped within a group for efficient accumulation as shown in Fig. 4.6. Bit-serial reduction is done in parallel for all the groups, and then the external reduction unit handles results from all groups sequentially. The reduction operation beyond the block size is performed using DSP blocks in floating-point mode, after the results of the external reduction unit are converted to the normal single-precision floating-point format.

4.3 Architecture Modeling & Customization

4.3.1 Load Partitioning and Execution

Fig. 4.7 illustrates how the matrix-vector multiplication is partitioned into sub-problems and mapped to the tiles and DPEs of the MVU in RIMA. The matrix and input vector are equally split (horizontally) into T *column blocks*, such that each tile is responsible for a matrix-vector multiplication sub-problem of dimensions $R \times (C/T)$. Within a sub-problem, each matrix column block is split (vertically) into several *row blocks*, such that each row block is mapped to a DPE. The number of matrix rows in a row block (i.e. row block size) differs depending on whether the row block is mapped to an L-DPE or an M-DPE. Within an M-DPE, the weights in one matrix row are mapped to one or more compute-capable BRAMs as described in Section 4.2.4. The number of L-DPEs and M-DPEs in a tile, as well as the portion of the matrix mapped to each type, are architectural parameters specified by the designer and can be different for each workload. All the DPEs in the same tile, regardless to their type, use the same portion of the input vector stored in the tile's VRFs. The outputs from all DPEs in a tile are concatenated to form an R-element partial result vector, then different partial results from different tiles are summed up in the global reduction unit (see Fig. 4.4) to produce the final output.

The execution of an RNN dataflow graph on RIMA proceeds as follows. The weights are loaded into the M-DPEs and the MRFs of L-DPEs offline as a pre-processing step. Different parts of the user-supplied input vector are first dispatched to their corresponding VRFs in different tiles. Then, all the DPEs in all tiles process their own sub-problem concurrently. The M-DPEs perform input loading, MAC, and reduction in parallel. Each M-DPE then performs across-BRAM accumulation, and sends out the results sequentially to reduce data transfer bandwidth. The L-DPEs load partial matrix rows and input vector from MRFs and VRF respectively, and perform the dot product computations in a pipelined fashion. Finally, after the complete output vector of the matrix-vector operation is produced, the subsequent MFUs consume it in chunks of 40 elements each to perform the vector elementwise operations depending on the workload dataflow graph. These steps are repeated again for the next time step of the RNN. For simplicity, these execution steps assume no overlap between different stages of the RIMA pipeline. For example, the first MFU does not start processing until the MVU is completely done. This is definitely sub-optimal and can be avoided by passing a part of the MVU result to the MFU while generating the next part. However, we leave this optimization for future work as we focus more on showcasing the gains of FPGA in-memory compute capabilities, rather than building the most efficient accelerator architecture.

4.3.2 Design Space Exploration & Architecture Customization

To achieve the optimal performance within the available FPGA resources, it is necessary to change the M-DPE configuration as well as the workload partitioning between the L-DPEs and M-DPEs for different problem sizes. Assigning a large portion of a large workload

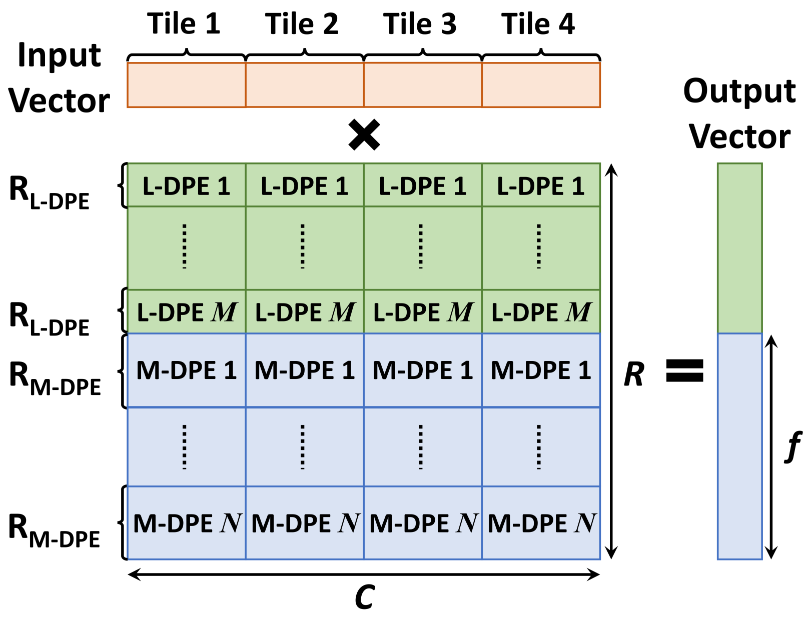


Figure 4.7: Partitioning of a matrix-vector multiplication operation to an example RIMA architecture with 4 tiles, M L-DPEs and N M-DPEs.

matrix to M-DPEs will cause all the model weights not to fit within the total chip BRAM capacity, since a portion of the BRAMs configured in compute mode must be reserved for the intermediate and final computation results. On the other hand, assigning only a small portion of the matrix to M-DPEs can cause significant underutilization of the BRAM compute throughput.

Since our RIMA architecture is very deterministic (e.g. with no pipeline overlaps or multi-threading), it is possible to estimate a given workload's performance using a detailed analytical model for a given architecture configuration. Therefore, we implement a design space exploration tool that uses an analytical model to find the optimal combination of architecture parameters that minimizes the processing latency for each specific workload. The analytical model includes the following configurable architecture parameters: the fraction of elements in the output vector that are computed by the M-DPEs (f), the number of MACs performed serially in each bitline (m), the operands' bitwidth after applying bit slicing (b), and the number of MVU tiles (T).

The optimal configuration of these parameters minimizes the MVU latency of a given workload with dimensions $R \times C$ under specific resource constraints. The overall MVU latency is the higher of L-DPE and M-DPE latencies, and the resource constraint is that the number of BRAMs used by L-DPEs (for MRFs) and M-DPEs, and the number of DSPs used by the L-DPEs do not exceed 85% of the total number of available BRAMs and DSP blocks in the target FPGA device. This leaves enough BRAMs and DSPs to implement the rest of the fixed RIMA pipeline. Although it is hard to capture the logic block utilization in the analytical model, it was never the design's bottleneck for all the RIMA instances that we experimented with in our study.

4.3.3 Reconfiguration Process of FPGA

The FPGA is reconfigured with different bitstreams for different RNN models. The Stratix 10 FPGA provides two fast methods for reconfigurations: via a proprietary protocol called Avalon, and via PCIe. The 32-bit Avalon can achieve a data rate of 500 MB/s, loading the bitstream in 0.37 seconds. With PCIe 3.0 x8, the data bandwidth reaches 8GB/s, thus transferring the bitstream in 0.023 seconds. The reconfiguration also covers the process of weight pinning. Note that the reconfiguration is infrequent, as the proposed architecture targets the cloud servers for DNN service providers.

4.4 Evaluation Methodology

4.4.1 Platforms and Benchmarks

To highlight the gains of using our enhanced FPGA architecture with compute-capable BRAMs for DNN acceleration, we first evaluate the performance of the RIMA architecture in comparison to state-of-the-art FPGA-based accelerators for RNNs. To ensure a fair comparison, we compare the BFP and INT8 versions of RIMA to the Microsoft Brainwave architecture from Fowers et al. (2018) which uses the same (1s.2m.5e) BFP format (BW-BFP) and the INT8 Intel NPU architecture from Nurvitadhi et al. (2019) (NPU-INT8), respectively. Both BW-BFP and NPU-INT8 use the same Stratix 10 GX 2800 FPGA device, and RIMA assumes a similar device (in resource count) in which the BRAMs are enhanced with in-memory compute capabilities. We also compare RIMA's performance to that of the same-generation Nvidia Titan V GV100 on the same set of workloads using the official Nvidia persistent CuDNN kernels. Although this GPU can perform half-precision (FP16) computations, the official kernels for these workloads only support single-precision (FP32). All our experiments use the RNN, GRU and LSTM models from the DeepBench (Baidu) benchmark suite.

4.4.2 FPGA Implementation and Validation

We implement the MVU and MFU of the RIMA architecture in SystemVerilog RTL. The MVU is the main compute complex of the architecture and the only pipeline stage of the original NPU architecture that we re-architect to use in-BRAM compute capabilities. According to the results in Nurvitadhi et al. (2019), the other NPU pipeline stages (i.e. eVRF, and LD) utilize less than 10% of the FPGA's LUTs and BRAMs, and they are never the bottleneck for the NPU's operating frequency. For the compute-capable BRAMs, we write an RTL module that maps to a normal BRAM block since the interface of compute-capable BRAMs to the programmable routing remains unchanged. We use Intel Quartus Prime Pro 17.1 to synthesize, place and route the RIMA architecture with four tiles and necessary number of M-DPEs/L-DPEs to obtain the frequency and resource utilization results for each instance customized for each specific benchmark. Then, we add in the resources utilized by the rest of the RIMA architecture components as previously mentioned. We also validate the functional correctness of our RIMA architecture by performing RTL cycle-accurate behavioral simulations using ModelSim. To simulate the compute-capabilities of our enhanced BRAMs, we write a black-box simulation model for the BRAM that supports both the memory and compute modes of operation. The evaluation on the RIMA architecture accounts for the overheads of routing and on-chip data movement, demonstrating the realistic effects of the improved computing throughput on FPGA.

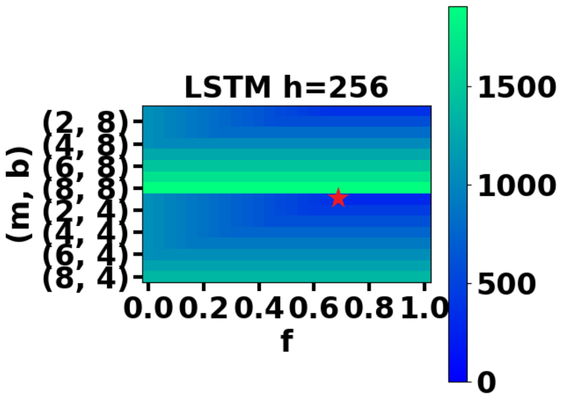
4.4.3 Performance Modeling

As existing FPGAs do not have the needed circuitry for in-BRAM compute, we rely on cycle-accurate RTL simulation with our behavioral simulation model of the compute-capable BRAM to obtain the cycle count for the RIMA MVU, which constitutes the majority of processing cycles. Then we combine that with an analytical model latency estimation for the deterministic MFU vector element-wise operations, assuming no overlap between the MVU and MFU processing. We obtain the maximum operating frequencies of the M-DPEs and the rest of the design from the Quartus timing reports. The BRAM in compute mode has a 1.6× lower frequency upper bound than an unmodified BRAM; in memory mode, the BRAM maximum frequency remains unchanged. The overall performance of each RIMA instance is then calculated based on the obtained frequencies and number of processing cycles for each benchmark.

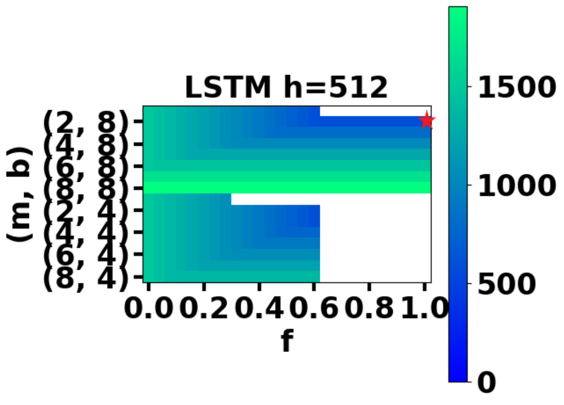
4.5 Results

4.5.1 RIMA Architecture Design Space Exploration

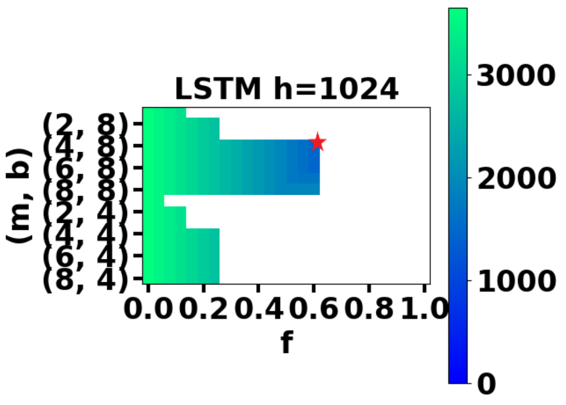
The heatmaps in Fig. 4.8 show the latency results obtained by the analytical model during the design space exploration process for the RIMA-INT8 variation for four different LSTM benchmarks. In this experiment, we fix the number of MVU tiles and lanes of the L-DPEs to be 4 and 40 respectively, similar to that used in Nurvitadhi et al. (2019). Each pixel in the heatmap represents one architecture configuration. The pixel color indicates the estimated latency for one time step in nanoseconds, a lighter color represents a higher latency and a darker color represents a lower latency (i.e. the darker the color is, the better). The white parts of the heatmaps indicate that the corresponding configuration violates the resource constraints of the analytical model. On the horizontal axis, the parameter f (fraction of



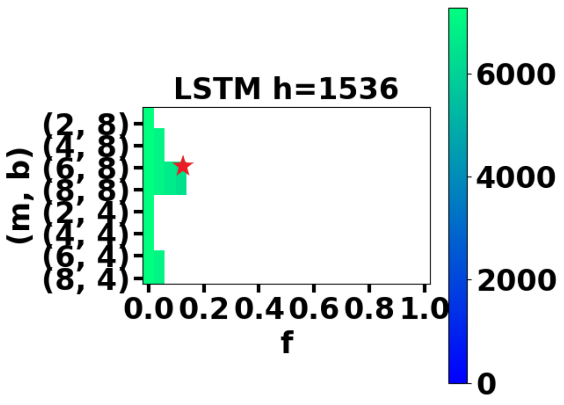
(a)



(b)



(c)



(d)

Figure 4.8: Design space exploration for the RIMA architecture parameters with number of tiles and L-DPE lanes fixed at 4 and 40, respectively. Each pixel is one architecture configuration and the color represents the processing latency. White points are invalid configurations violating the resource constraints.

compute mapped to M-DPEs) changes from 0% to 100%. The vertical axis represents 16 different combinations of the parameters (b, m), where b (bits per slice) takes the value 8 or 4, and m (MACs in serial on the same bit-line) is swept from 1 to 8.

With infinite resources, a larger f and smaller m would increase the degree of extracted parallelism and therefore lead to better performance. However, as shown in the figure, the larger the model size gets, the less fraction of compute can be mapped to the M-DPEs to ensure that the complete model still fits in the on-chip BRAMs. The optimal configurations for these models (highlighted by a red star on the heatmaps) are different, which highlights the importance of hardware customization. Our experiments show that the per-workload customization of the RIMA architecture offers an average 18% (up to 40%) performance improvement compared to a fixed RIMA instance for all workloads. For the largest LSTM model ($h = 1536$), the optimal architecture configuration has $m = 6$, $b = 8$, and $f = 0.15$. This model has 13.5 MB of weight data, which imposes tighter constraints on the fraction of computations that can be offloaded to the M-DPEs (only 15%). To offload more computation to the M-DPEs, six bit-serial MACs ($m = 6$) are performed sequentially on the same bitline, and no bit slicing is applied. On the other hand, for the smallest LSTM ($h = 256$), the optimal architecture configuration has $m = 1$, $b = 4$, and $f = 0.68$. This small model has a 512×1024 weight matrix, which enables us to map 68% of the compute to the M-DPEs, perform only 1 MAC operation per bitline (fully unrolled) and apply the bit slicing to extract a high degree of parallelism.

For RIMA-BFP, we found that for all the studied workloads, we were able to map all the computations to M-DPEs, which provides superior performance than any hybrid configurations. This highlights that in-BRAM compute on FPGAs achieves the best performance gains for lower precisions (3-bit operations in BFP8), which are becoming more widely adopted for DNN inference.

Table 4.1: FPGA implementation results of RIMA instances customized for each workload. The operating frequencies are for the M-DPE clock domain.

Benchmark	**Precision**	**Logic**	**BRAMs**	**DSPs**	**Freq. (MHz)**
RNN h=1152	**INT8**	87%	72%	50%	328
	BFP	59%	46%	44%	333
RNN h=1792	**INT8**	70%	65%	50%	417
	BFP	79%	64%	60%	250
LSTM h=256	**INT8**	60%	55%	50%	455
	BFP	49%	46%	40%	345
LSTM h=512	**INT8**	74%	69%	50%	417
	BFP	63%	42%	39%	323
LSTM h=1024	**INT8**	73%	69%	50%	313
	BFP	63%	42%	39%	323
LSTM h=1536	**INT8**	89%	93%	50%	278
	BFP	84%	57%	60%	303
GRU h=512	**INT8**	74%	69%	50%	417
	BFP	82%	57%	60%	303
GRU h=1024	**INT8**	70%	65%	50%	417
	BFP	82%	57%	60%	303
GRU h=1536	**INT8**	85%	89%	50%	296
	BFP	82%	57%	60%	303

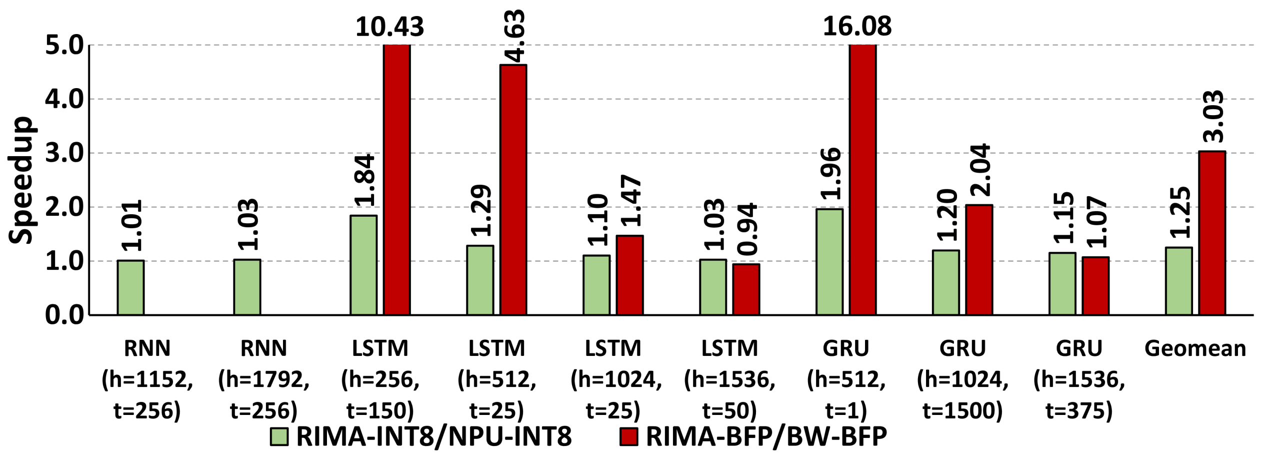


Figure 4.9: Speedup achieved by RIMA INT8 and BFP compared to NPU-INT8 and BW-BFP, respectively.

4.5.2 RIMA FPGA Implementation Results

The FPGA resource utilization results of the per-workload customized RIMA instances are shown in Table 4.1. The results show that the implemented RIMA instances achieve a high degree of the FPGA resource utilization in all cases. Models with a larger weight size generally have a higher BRAM usage, as more weights need to be pinned on chip. The M-DPE clock frequencies for different workloads also vary due to the effect of architecture parameter customization on the external reduction units and accumulators. In RIMA-BFP, the whole architecture operates at the same frequency reported in Table 4.1, while in RIMA-INT8, the M-DPEs operate at a higher frequency (shown in the table) than the 278 MHz clock driving the rest of the architecture.

4.5.3 Performance Results

Fig. 4.9 presents the speedup achieved by RIMA-INT8 compared to the NPU-INT8, and RIMA-BFP compared to the BW-BFP. RIMA-INT8 and RIMA-BFP achieve average speedups of **1.25× and 3× over the baseline NPU-INT8 and BW-BFP** that do not use compute-capable BRAMs, respectively. As previously discussed, the gains of adding BRAM compute-capabilities are higher when used to accelerate narrower precisions. This

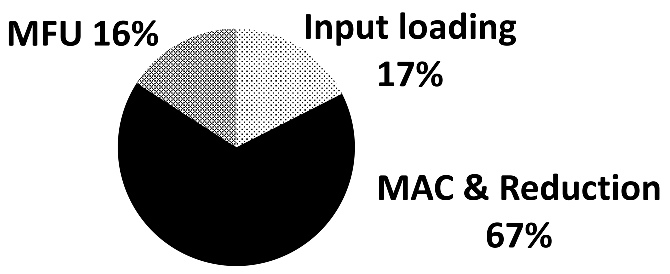


Figure 4.10: Latency breakdown of RIMA-INT8 for the LSTM ($h = 1024$) benchmark.

Table 4.2: Processing latency comparison between RIMA and Nvidia Titan V GV100 GPU. All numbers are in milliseconds.

Workload	**RNN**		**LSTM**			**GRU**	
h	**1152**	**1792**	**256**	**512**	**1536**	**1024**	**1536**
t	**256**	**256**	**150**	**25**	**50**	**1500**	**375**
GPU (FP32)	1.0	1.38	0.44	0.15	5.7	12.5	29.94
RIMA (INT8)	0.21	0.42	0.06	0.02	0.24	2.63	1.26
RIMA (BFP8)	0.19	0.39	0.04	0.02	0.15	1.86	0.89

can be attributed to two main reasons: (1) bit-serial multiplication latency grows quadratically with operands bitwidth, and (2) narrower bitwidths allow sharing the same memory array bitline among different weight values that are all multiplied by the same input. Therefore, the RIMA architecture has the potential to accelerate other low-bit-width formats such as INT4; the detailed analysis for other precisions is left for future work. The figure also shows higher speedups for smaller workloads that impose more relaxed constraints on the amount of workload computation that can be mapped to M-DPEs. The smaller the amount of weights that need to be kept persistent in the on-chip memories, the higher the degree of compute parallelism that can be extracted from the compute-capable BRAMs.

For some of the workloads in Fig. 4.9, the speedup achieved by RIMA exceeds the peak performance gains of compute-capable BRAMs (1.6× and 2.3× in Section 4.1). This is because the baseline NPU-INT8 and BW-BFP architectures suffer from significant underutilization and padding overheads due to their fixed overlay architecture for all workloads. In contrast, RIMA best exploits the FPGA reconfigurability by customizing the architecture parameters to achieve the optimal performance for each given workload.

As an example, Fig. 4.10 shows the breakdown of the processing latency of RIMA-INT8 for the LSTM benchmark with 1024 hidden units ($h = 1024$). It shows that the matrix-vector multiplication in M-DPEs constitutes 84% of the processing time, where the majority of this time is spent in bit-serial MAC and reduction operations. The latency of the external reduction unit is fully overlapped with the in-BRAM computation. Loading the inputs to the compute-capable BRAMs takes 17% of the cycles, while the remaining time is spent on the vector operations in the MFUs.

Table 4.2 compares the processing latencies of different workloads running on the RIMA architecture and the Nvidia Titan GV100 GPU. The results show that the RIMA architecture **outperforms the GPU by 8.1× and 10.6×** when using the INT8 and BFP numerical precisions, respectively.

4.6 Summary

The continuous increase in the capacity of FPGA on-chip memories offers a great opportunity to also enhance the device's compute throughput by supporting in-BRAM compute capabilities. Our proposed architectural change can enhance the peak MAC throughput of a large Stratix 10 device by a factor of 1.6× and 2.3× for 8-bit integer and block floating-point precisions, respectively. This comes at a cost of 7.4% increase in the BRAM tile area, which corresponds to only 1.8% increase in the total FPGA die size. In addition, our proposed compute-capable BRAM does not change the existing BRAM modes or interface to the programmable routing, and requires minimal CAD support which further simplifies its adoption in commercial architectures. We also evaluate the effect of enhancing FPGAs with compute-capable BRAMs on deep learning inference performance. To do that, we implement a reconfigurable in-memory accelerator architecture, RIMA, which uses compute-capable BRAMs and exploits the FPGA reconfigurability to perform per-workload architecture customization. The RIMA architecture outperforms the state-of-the-art Brainwave overlay by

1.25× and 3× for 8-bit integer and block floating-point precisions respectively on a variety of real-time memory-bound RNN workloads. It also achieves an order of magnitude higher performance compared to same-generation GPUs. This study shows promising results for incorporating compute-capable BRAMs in modern FPGA architectures, especially for narrower data precisions which are becoming widely adopted in deep learning inference tasks. The proposed FPGA architecture enables a reconfigurable in-memory DNN accelerator, while may also accelerate other compute intensive applications with the abundant bit-serial computing units created.

CHAPTER 5

Cache Compression with In-SRAM Data Comparison

In this chapter, we study how the in-SRAM computation can be leveraged for performing a generic type of architectural optimization - cache data compression, which is useful for DNN application under data quantization as well as many other application domains. This chapter is organized as follows: Section 5.1 provides background knowledge on the data addressing and organization in cache. Section 5.2 contains the details of the proposed compression mechanism, including the location of the compressed data, the procedure of compression, decompression, and data replacement. Section 5.3 describes the methodology for evaluation and Section 5.4 presents the results on compression ratio and performance.

5.1 Cache Organization and Data Addressing

We demonstrate our compression design based on the Intel Xeon last-level cache (Huang et al., 2013) which has one cache slice per core. Figure 5.1 (a) shows the high level architecture of a cache slice. Each cache slice is organized in multiple columns, and each column serves as one way in the set-associative cache. By changing the number of columns in a cache slice, the cache capacity may vary for different CPU models. We use 1 MB as cache slice size, which is investigated in previous works (Pekhimenko et al., 2012). There are 8 way columns in the 1 MB cache slice. Each way is organized in 4 banks, and each bank has 4

SRAM arrays of size 256×256. The identical bank across all the ways together constitute a *quad*. In each way, a cache line is mapped to distributed locations in multiple SRAM arrays for parallel data access. Specifically, the 64B cache line is split into 8 segments of 64 bits each, and are mapped to 8 arrays, either on the left half or right half of the way column. In one SRAM array, the 64 bits from a cache line occupies one-fourth of a word-line, and there are in total 1024 such 64-bit data segments from 1024 cache sets. The mapping of cache sets in a bank is shown in Figure 5.1 (b).

The data are read out from (and written to) the cache with the data buses connecting all the ways. There is one 64-bit data bus in each quad, and the 4 data buses can transmit 256 bits per cycle. The 512-bit cache line is accessed in 2 cycles. The data access pattern within a bank is shown in Figure 5.1 (b). In a data reading, the upper and lower half of the bank each provides 32 bits and they concatenate to transmit on the quad data bus. In the upper (lower) half-quad, the 32 bits are selected from the 2048 sets by the set ID. More specifically, the row-decoder and column-mux selector of a SRAM array chooses the 32-bit data segment to read from the array, and an extra mux selects between the two arrays in the half-quad.

5.2 Compression Design

5.2.1 Overview of Compression Opportunity

In our proposed compression method, we examine how the data of the cache lines are distributed across the slice according to the cache architecture. Then we determine the ranges of cache lines for detecting similarity based on the layout. In this way the decompression procedure remains simple and fast.

While the entirely or partially duplicated cache lines can reside anywhere in the cache, we target the duplications that are in the structurally neighboring positions in the cache. This reduces the decompression latency, as only light modifications are needed to read the data at the correct position.

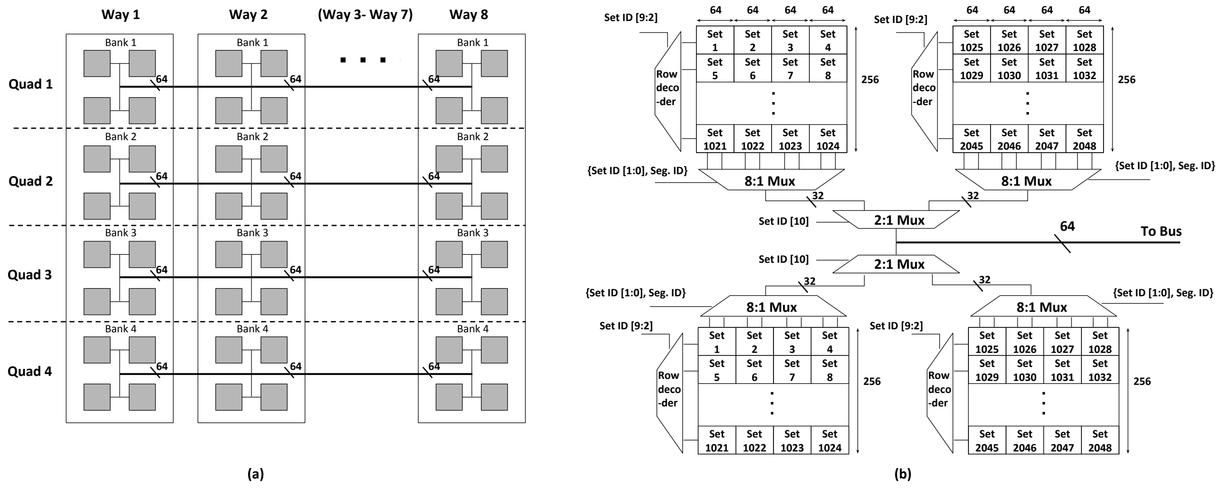


Figure 5.1: Cache slice organization for 1MB Intel Xeon LLC (Huang et al., 2013). (a) High level structure of the ways, banks and data arrays. Data bus connects banks in different ways of the same quad. (b) The set mapping and data accessing logic within a bank. The sets are interleaved within an array, and the data of a cache line is distributed. The row, column, and left-right-array selector chooses the accessed data location for a given Set ID. Seg ID is a 1-bit value to select one of the two 32-bit segments of the 64 bits from the same cache line in an array.

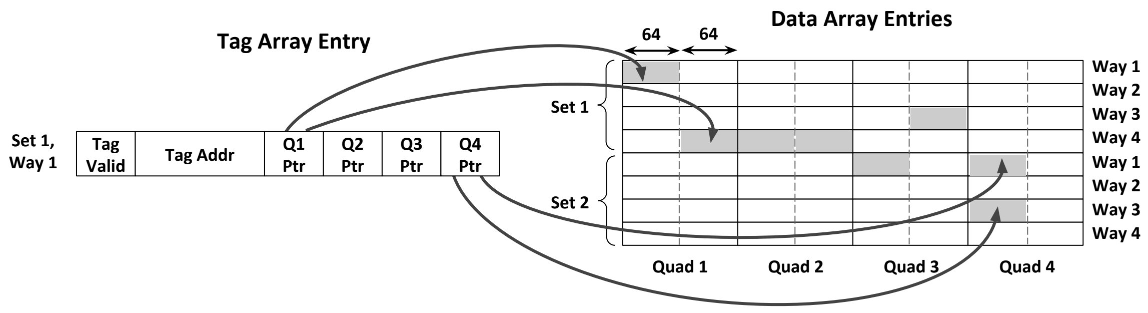


Figure 5.2: The mapping from tag entry to data arrays after compression. Left: The format of an entry in the tag array (Set 1, Way 1) with the proposed compression method. Right: Logical structure of the partial data array that the tag entry may point to. Each cache line is partitioned into 4 quads, each further partitioned into 2 segments of 64 bits. The pointers for each quad include the address of the two segments in the pointer. The pointers for quad 2 and 3 are not shown for simplicity. The shaded boxes represent the actual data segments that correspond to this tag entry.

Specifically, we consider two levels of structurally neighboring cache lines. First, we target the cache-lines in different ways in the same set, and compress the duplicated cache-lines by storing only one copy of data. In original cache, the way is chosen by the matching status of tag address; with the proposed compression design, a pointer indicates the data from which way should be read out. Second, we extend the range of cache-lines to a group of neighboring sets. To select the correct data from the set group, one needs to change the mux selection signal from the set ID to a pointer. The set group size is tunable to balance the compression ratio and the energy efficiency.

We search for data duplications at a finer granularity than a cache line to enable more opportunities for compression. First, the 4 quads are searched independently for duplications, and they may find duplications at different ways. Second, within a quad, we observe that the data are read out from 2 positions in two cycles. Therefore, we can further partition the cache line data in a quad (128 bits) into two segments, and compare with the 2 positions separately.

5.2.2 Compressed Data Format and Decompression Procedure

This section describes the organization of the cache with the proposed compression strategy. On a high level, although the modified cache is still set-associative, the tag array and the data array of the compressed cache are decoupled, as the mapping from the tag to data is not fixed. Extra bookkeeping structures are utilized to maintain the mapping between the tags and the data.

The tag array is still organized with sets and ways. To hold more cache lines in the cache, the number of ways is increased for the tag array. In our design, the number of ways is 16, which is $2\times$ of the original associativity. The total number of sets remains the same at 2048. In each tag entry, the tag valid information and the address bits are the same as the conventional cache. Extra pointers to the data array are appended to each tag entry, for retrieving the correct data in the read operation. As 4 quads do compression independently,

Table 5.1: Remapping of control signals to retrieve data after compression within Quad 1.

Control Signal Usage	**Original Cache**	**Compressed Cache**
Column Mux Sel.	{Set ID [1:0], t}	Q1_Ptr[t][2:0]
Row Decoder	Set ID [9:2]	{Set ID [9:3], Q1_Ptr[t][3]}
Array Sel.	Set ID [10]	Set ID [10]
Way Sel.	Way ID [2:0]	Q1_Ptr[t][6:4]

the four quads and two segments in each quad have separate pointers. Each pointer specifies the location of the 64-bit segment in the set group, including the way ID (3 bits for 8 ways), the in-group set ID (n bits for a 2^n-set group), and the segment order (1 bit). Figure 5.2 shows the modified tag entry with data pointers, in an illustrative example of 4-way data array and 2-set data comparison range. It is shown that the two segments of each quad can be stored at any two slots in the quad. In the case when the two segments in the same quad have identical data, they may also be mapped to the same data array slot (not shown in the example). Different tags mapping to the same data array slot effectively form a compression.

For a cache hit, to generate the data for the requester, the cache controller simply follows the data pointers to fetch the data. Table 5.1 describes the difference between the control signals for accessing the data, between the original and the compressed cache, for a configuration of 8 sets in a group for comparison. The table describes the signal for quad 1, and the other quads follow the similar remapping mechanism. The 1-bit signal t represents the timestep and varies between 0 and 1. With compression, the way selection, the column mux selection, and the lower bits of the row decoder signals are replaced by the pointers to choose the correct data to access. The set group size determines how many lower bits of the row decoder are replaced, as each set group may include one or more rows in an array. The data access is still done in parallel for the 4 quads, and in two cycles, the entire cache line can be read out. The decompression only incurs a one-cycle extra latency to access the data pointers.

The data array still has the same 1 MB capacity as the original one. The mapping of the sets and the ways to the cache geometry is also unchanged. An extra structure is designed to monitor the status of the data. Specifically, for each segment in the data (64 bits, atomic unit for duplication detection), there is a status register indicating how many cache-lines are actually using this segment. This helps to identify non-occupied data segments during the data replacement (details in Section 5.2.4).

5.2.3 Compression Procedure: Data Matching

Compression is performed when a new cache line arrives at the cache. The new cache line is compared with existing data in the cache to search for duplications. We leverage the bit-line peripherals to compare with existing cache content in the arrays. Note that the compression is not in the critical path, because the data is first served to the requester before the cache is searched for duplications.

The data at the 4 quads are compared separately. As discussed in Section 5.2.1, the compression granularity is 64-bit data segments, and each quad contains two such segments. During compression, at each quad, the two segments are written to the arrays for comparison with peripherals. They are written to either the left two arrays or the right two arrays depending on the set ID of the incoming cache line. Each array receives 32 bits from the first segment, and another 32 bits from the second segment. Then in an array, the 32 bits of each segment is compared to all the 32-bit segments within the set group. For example, if the new cache line belongs to set 7, and the set group size is 4, then the data from set 5, 6, 7 and 8 are compared to in an array.

In the data comparison, the bit-wise XOR is performed on the existing cache line and the new cache line, and then the XOR results go through a wired-NOR logic to generate the final results of whether there is a match. We use sense-amp cycling (Subramaniyan et al., 2017) to accelerate the comparison on multiple existing cache data. This is possible because the data to be compared are in the contiguous locations in the array, so the multiple comparisons in the same word-line can be accelerated by sense-amp cycling. As the data at multiple locations in the array are compared sequentially, a counter can record the information of which location results in a match.

The new cache line is sent to all ways (via the data bus) for comparison. The comparison for multiple ways are done in sequential, and once a match is found, the searching process terminates, and thus saves energy.

The comparison energy can be further reduced by an adaptive strategy for comparing. The idea is that for some benchmarks which have few duplications in data, we may skip the comparison as the compression is not effective on these benchmarks. We first check the fraction of successful compressions over a sampling period. Then if the compression successful rate is higher than a threshold, in the subsequent strategy executing period, we will always check for compression opportunities with in-SRAM comparison. Otherwise, the existing data will not be compared to the new data; instead, the cache lines are evicted based on the replacement policy to make space for the new cache line (details in Section 5.2.4. The compression successful rate is calculated as the fraction of incoming cache lines that are at least partially duplicated to the existing cache data so the compression is effective. The threshold and the sampling and executing period lengths can be empirically decided.

5.2.4 Data Replacement Procedure

The entries in the tag and data arrays need to be replaced by new cacheline due to capacity conflicts. As the tag and data entries do not maintain a one-to-one mapping with the compression, simple conventional cache replacement strategy cannot be directly applied.

When a cache miss happens, the tag entry in the set that is least-recent-used (LRU) is evicted, then we check whether the new data can fit in the data array with either compression or empty data slots. If so, then we record the data location and the replacement procedure finishes; otherwise, we evict other tag entries within the set group, also based on LRU policy. Such eviction can release data space in the set group and lead to the fitting of the new cache-

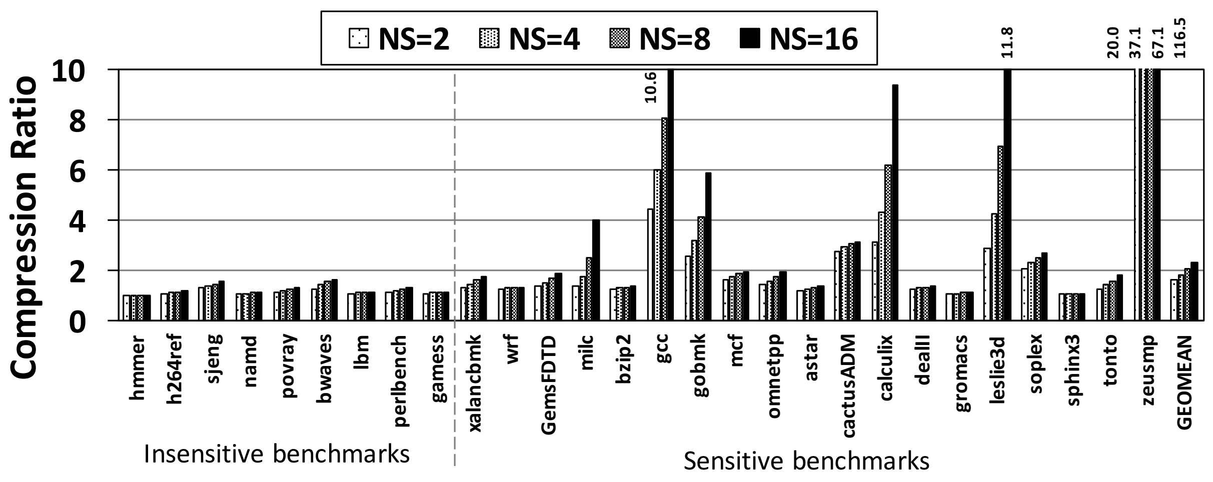


Figure 5.3: Compression Ratio for all benchmarks (higher is better), defined as the data space taken on a conventional cache divided by the data space taken on cache with compression. For each benchmark the compression ratio is an average over the entire program execution time.

line data. If eviction once does not create enough space, additional tag entries will be evicted, until the data can fit.

Note that whether the new cache line can fit in the cache is determined together by the data fit status of each individual quad. Only when all four quads can fit the corresponding two segments (either data match or empty slot), the entire cache line is considered as successfully inserted.

In an eviction, the tag entry itself is invalidated, and the data segments of this cache-line need to be notified. The data pointers are used to find all such segments. Then the status counters of these segments decrease by one, indicating that the number of tag entries using the segment is reduced by one.

For a write that modifies the data in the cache, we first remove the segments in the cache line itself that have changed values from the previous data. Then if the new data cannot fit due to the removed segments having multiple tag entries point to it, we evict other tag entries to make space.

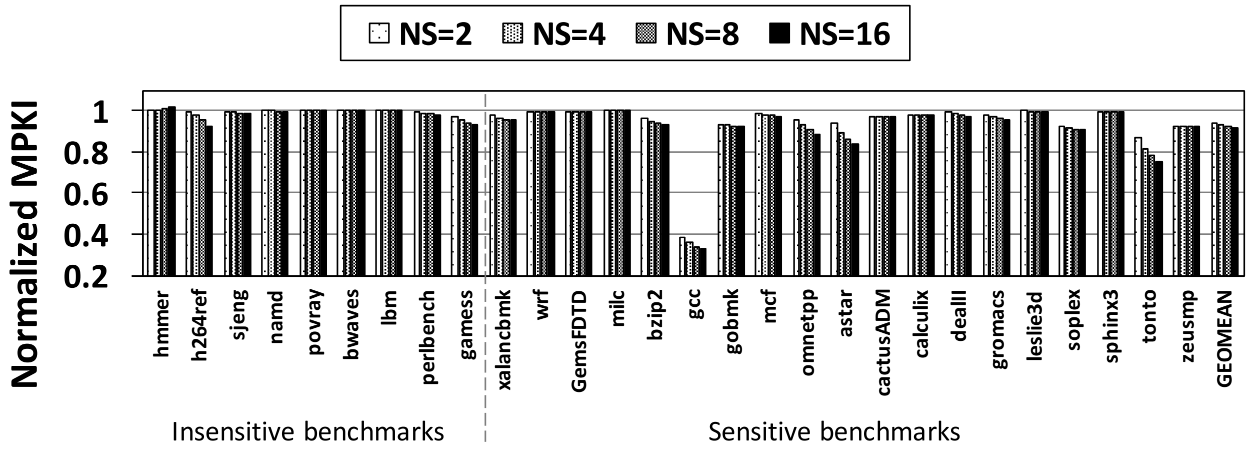


Figure 5.4: The relative MPKI (miss per kilo instruction) of the compressed cache, normalized to a baseline conventional cache. Lower is better.

5.3 Evaluation Methodology

We evaluate on the SPEC2006 benchmarks to demonstrate the effects of the proposed cache compression method. We use SimPoints (Hamerly et al., 2005) to generate the representative program regions for each benchmark, to enable faster performance simulation. At most 10 representative intervals are generated for a benchmark, each interval having 30 million instructions for detailed performance modeling, and 100 million instructions as a warm-up stage.

We extract the memory access traces, including the data content, with an instrument tool based on Pin (Luk et al., 2005). We use an in-house cache simulator to estimate the cache behavior with compression, based on the memory access trace. The cache behavior trace (cache hit information) is fed to the Sniper Simulator (Heirman et al., 2012) to estimate the performance. The CPU model specifications are listed in Table 5.2. In baseline CPU, the 1MB last-level cache has the structure described in Section 5.1, with 8 ways and 35-cycle access latency.

We evaluate the proposed compression method with multiple configurations. The number of sets in a set group (NS) of 2, 4, 8, and 16 are evaluated, to show the sensitivity of compression effect to the range of compared data.

Table 5.2: Specification of the Simulated CPU Model

Core	single-core, 2.66 GHz, x86-64, out-of-order
L1-i	4-way, 32 kB, 4-cycle access latency, LRU
L1-d	8-way, 32 kB, 4-cycle access latency, LRU
L2	8-way, 256 kB, 12-cycle access latency, LRU
LLC	8-way, 1 MB, 35-cycle access latency, LRU

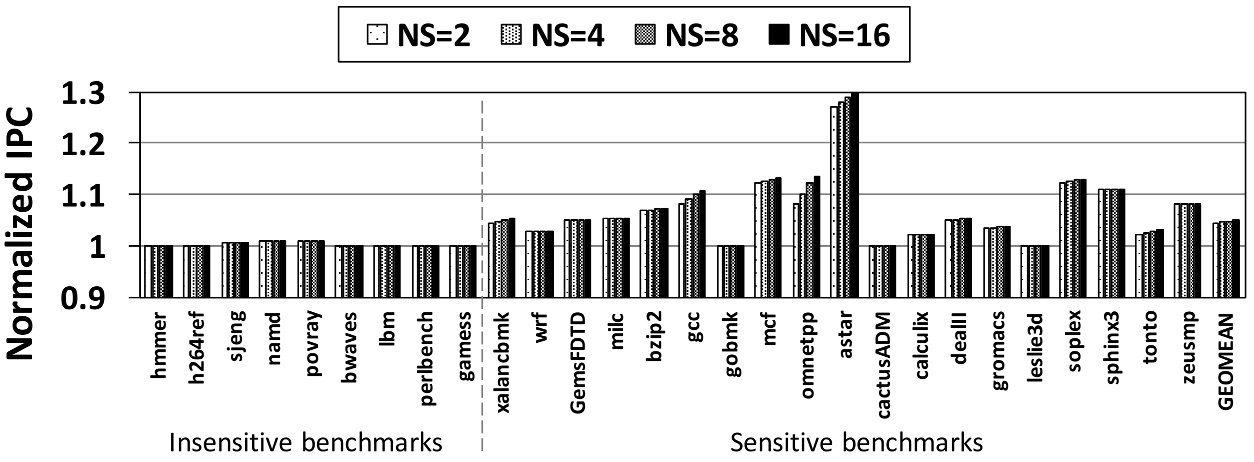


Figure 5.5: The relative IPC of the benchmarks with the compressed cache, normalized to a baseline conventional cache. Higher is better.

5.4 Results

5.4.1 Compression Ratio

Figure 5.3 shows the compression ratio achieved by the proposed compression method, with different number of sets per group. The compression ratio is defined as the ratio between the data array space that the data would have taken on a conventional cache (the valid tag entries multiplied by the cache-line size), and the actual space that the data take in the compressed cache (data segments being pointed to by at least one tag entry multiplied by segment size). The compression ratio can represent the level of data duplications actually leveraged. Multiple benchmarks (gcc, calculix, leslie3d, zeusmp) show a high compression

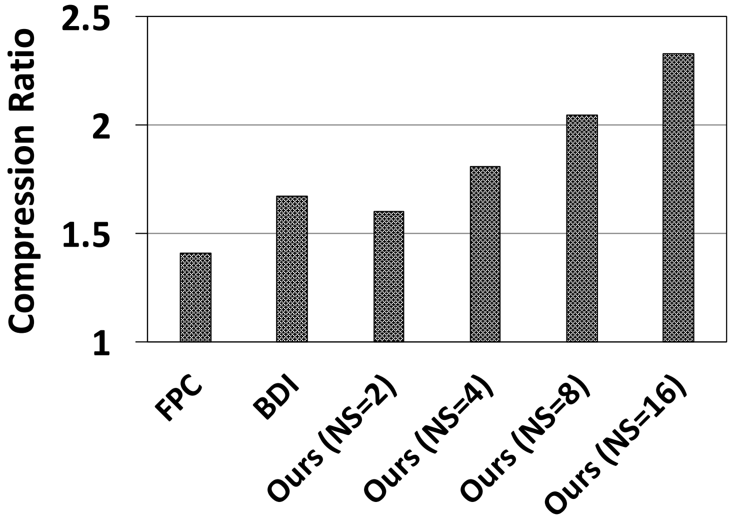


Figure 5.6: The average compression ratio on SPEC2006 benchmarks for FPC, BDI and our compression method, all with 2048-set 1MB data array.

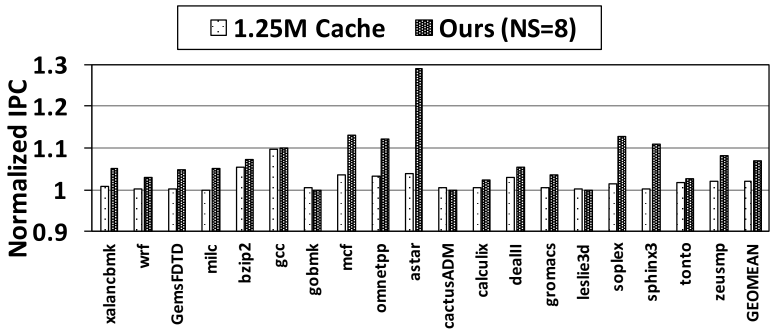


Figure 5.7: The IPC on compression-sensitive benchmarks for 1.25 MB cache and our compression method (NS=8).

ratio, potentially due to the high similarity of the data values of the cache lines.

Note that the compression ratio for a benchmark can be higher than 2, although the tag array capacity is only 2× of the baseline scenario. For example, consider a naive case, where the baseline cache has 4 entries, and the compression-enabled new cache has 8 tag entries and 4 data entries. Then if all 8 tag entries, as well as only 2 data entries are occupied, the compression ratio is calculated as $8/2 = 4$.

5.4.2 Miss Rate

We show the normalized data miss rate (misses per kilo instructions, or MPKI) in Figure 5.4. The normalized MPKI is defined as the MPKI of a baseline configuration, where the L3 cache is a conventional cache with 1 MB data arrays, and the MPKI of the cache with the proposed compression mechanism. The benchmark gcc achieves a significant reduction in miss rate, indicating that the compression mechanism effectively keeps more data in the cache. The miss rate reduced by 8.2% for NS=16, and 7.6% for NS=8 on average.

5.4.3 Performance

The relative instruction per cycle (IPC) of the compressed cache over the baseline 1 MB cache are shown in Figure 5.5. The performance of some benchmarks such as astar, gcc, mcf, and omnetpp are more sensitive to the reduction of cache miss, where we see a higher speedup. On average the compression with NS=16 achieves 4.87% of speed improvement, and 4.73% for NS=8.

5.4.4 Discussion

5.4.4.1 Comparison with Other Cache Compression Methods

To demonstrate the effectiveness of the proposed compression method, we compare with two related work of cache compression, FPC (Alameldeen and Wood, 2004) and BDI (Pekhimenko et al., 2012). FPC detects whether a data word in the cache matches a pre-determined compressible pattern, and stores it in compressed format if so. BDI stores a cache-line in a format of a base value and many delta values, when the components in a cache-line are similar to each other. We test these two compression mechanisms with the identical tag and data space: 1MB data arrays and 2× tag entries of the original cache (16 ways per set).

The geometric mean of the compression ratio on all the benchmarks are shown in Figure 5.6. The FPC and BDI achieve 1.41× and 1.67× compression ratio, respectively. With NS=8, our compression method has a 2.05× compression ratio. The higher compression ratio of the proposed method can be mainly attributed to the wider range of data comparison than within a cache-line.

5.4.4.2 Energy Consumption

The proposed data compression method incurs an energy overhead of the in-SRAM data comparison. Note that in-SRAM comparisons are an order of magnitude cheaper than LLC cache access (Aga et al., 2017; Subramaniyan et al., 2017). Majority of LLC cache access energy is consumed by data-movement between core and LLC data arrays. However, as the compression reduces the total number of cache misses, the expensive off-chip DRAM access energy can be saved. The in-SRAM operation energy is further reduced by the adaptive compression strategy (described in Section 5.2.3). Finally, as the CPU itself consumes energy through leakage power, the reduction in total execution time will also reduce the CPU energy, leading to an overall energy saving of the system.

The benchmarks with high compression ratio (e.g., zeusmp) have a large portion of the unoccupied data arrays, as many tag entries are pointed to the same data. This can lead to further energy savings, as these data arrays may be turned into idle state with low leakage power (Flautner et al., 2002).

5.4.4.3 Area Overhead

The area overheads for compression mainly consist of two parts. First, there are pointers from the tag entry to the data arrays. They are required for all the tag entries and all quads, and take up 112 kB of space (for NS=8). Second, the data entries need to maintain the number of linked tags (see Section 5.2.2), requiring 64 kB of space (for NS=8). They combined equal an 17% of area overhead of the 1MB data. To demonstrate the area efficiency for the compression, we compare with the conventional cache with extra ways with area used as additional data arrays (2 extra ways, 1.25 MB of data). Figure 5.7 shows the IPC comparison on the compression-sensitive benchmarks. On average our method is 4.9% better than the cache with increased data arrays.

5.5 Summary

In this chapter, we proposed a cache compression mechanism for compressing across cache lines with similarities. We leverage the in-SRAM bit-line computing to efficiently compare for duplications in the data arrays. Our decompression procedure is simple and fast with a remapping of the data selection control signals. The compression leads to a 2.05 $\times$ compression ratio (up to 67$\times$), and a 4.73 % of IPC improvement (up to 29%), over the SPEC2006 benchmarks.

CHAPTER 6

QoS-Aware Collocation for Deep Neural Networks

In this chapter, we study the problem of guaranteeing DNN latency QoS under the collocation scenario. We first present the unique opportunity for DNNs to achieve QoS by resource partition (Section 6.1). Then we propose the design of the system, called DQG, which consists of the stages of offline latency prediction and online resource partition (Section 6.2), followed by the design of the latency predictor (Section 6.3). Finally we present the evaluation methodology (Section 6.4) and the experimental results (Section 6.5).

6.1 Opportunity for Collocated DNN to Satisfy QoS

A common method to ensure workload QoS under collocation is by resource partition and isolating the workloads so that the interference is minimized. Many CPU-based server systems support the partitioning of different types of resources (such as processor cores, cache capacity, and memory bandwidth) and allocating each partition to a different user process. These resource partitioning mechanisms include hardware and software implementations and some representative partitioning techniques are summarized in Table 6.1. For example, `taskset` can make a process run on a given set of logical cores (a subset of all the available cores); the Intel Cache Allocation Technology restricts the last-level shared cache capacity to a certain process, by only allowing it to use a certain group of ways in the set-associative cache.

Table 6.1: Different types of server resources and their isolation mechanisms.

Resource type	**Isolation mechanism**
Processor core	CPU core affinity tools (Linux taskset)
Last level cache capacity	Intel Cache Allocation Technology
Memory capacity	Linux memory cgroups
Disk bandwidth	Linux blkio cgroups

As the workloads only use their own partition of resource, the interference caused by resource contention is reduced. Then it becomes possible for the workloads to meet their QoS targets as their performance is not interfered by other collocated ones.

How to allocate the partitioned resource to the workloads is a key question in the above approach of partitioning and isolation. Many previous works (Lo et al., 2015; Chen et al., 2019) have taken an adaptive approach to adjust the resource allocation, where the resource for one workload is increased if its performance is below the QoS target and vice versa. One drawback of this adaptive approach is that it takes time to test the latency of a workload during the actual serving and thus the workload performance may not reach the QoS target when the transition is ongoing. This problem exacerbates when the latency of a service is relatively longer, which is true for many of the DNN inference services.

DNN inference has one unique property that can be potentially leveraged to avoid the aforementioned transition overhead - the predictability of its latency. This is because many DNNs have a fixed computation path for the varied inputs. Take the example of ResNet, a convolutional neural network. It can be observed that for any given input image, the computation flow and the memory accessing pattern is the same across all the layers of the network. This leads to a potential of predicting the runtime latency based on offline profiling, which enables a fast transition to a desired resource partition during service time.

Towards this opportunity, we propose DQG (DNN QoS Guard), which predicts the latency of DNN workloads under different resource partitions and thus making the DNN workloads meet the QoS target under collocation.

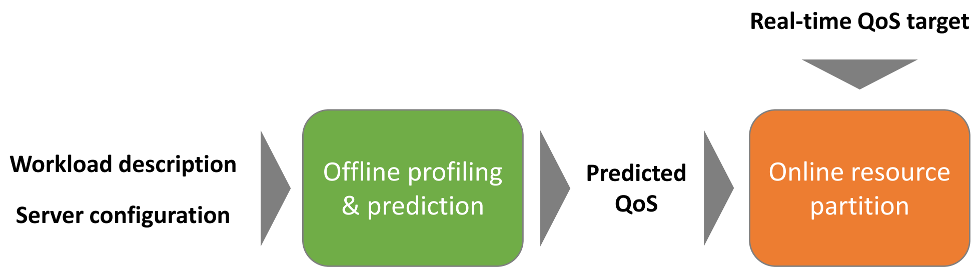


Figure 6.1: Overview of the Workflow of DQG.

6.2 Design of DQG

DQG mainly consists of two parts: the offline stage of profiling and performance prediction for each workload; and the online stage to determine the resource partition given a specific workload combination and the QoS requirements. The overall procedure of DQG is summarized in Figure 6.1.

6.2.1 Offline Performance Prediction

DQG targets the scenario where the machine learning workloads are known, and can be profiled on the servers with the exact same configuration as the actual runtime servers. This is consistent with the real-world deployment of machine learning services, as a fixed DNN model is often deployed to process a massive amount of inputs for an extensive period of time. Also, there are usually a large array of homogeneous servers in the datacenter, so it is possible to perform the profiling procedure on an identical server.

In the profiling stage, we vary the amount of resources allocated to the machine learning workload and measure the average latency under each resource level. The different types of resource (e.g., cores and cache ways) are swept in an orthogonal way. To reduce the time of profiling, we pick a subset from the total set of resource level configurations and the predictor (details described in Section 6.3) can generate a predicted latency for all the other unmeasured configurations. The profiling and prediction is done for each of the workload separately, as the different DNN workloads are sensitive to different types of resources, and at different levels. The predicted latency results for each workload is stored in a table, which will be accessed during the online resource partition decision procedure.

6.2.2 Online Resource Partition

At runtime, DQG determines how to allocate the partitioned resources to the multiple collocated workloads, depending on their QoS targets and priority levels. DQG needs the information of workload types, priority levels, and QoS targets, as the inputs at runtime.

DQG supports two levels of priority for the workloads: latency-critical (LC) and best-effort (BE). The latency-critical workloads have their specific QoS target on latency, while the best-effort workloads do not have any specific performance goal to achieve. DQG attempts to find a partition scheme that satisfies all the QoS targets of LC workloads. A searching algorithm is deployed for this purpose. The search space is all the possible ways of partitioning different types of resources into the workloads. For each partition, by looking up the table of predicted latency of each LC workload and comparing it with the latency target, DQG can determine whether this workload's QoS target is met under the partition. Within all the QoS-satisfying partitions, the algorithm chooses the one with most resources for BE workloads, to potentially improve the performance of BE workloads as well.

If a partition to meet all the QoS targets is deemed impossible by the searching algorithm, then it is up to the cluster-wise workload scheduler to determine how to migrate some workloads to other servers. When any of the workload information changes, a new resource partition needs to be generated. The incurred timing overhead includes the searching algorithm execution time and the time to change the resource allocation by the isolation tools.

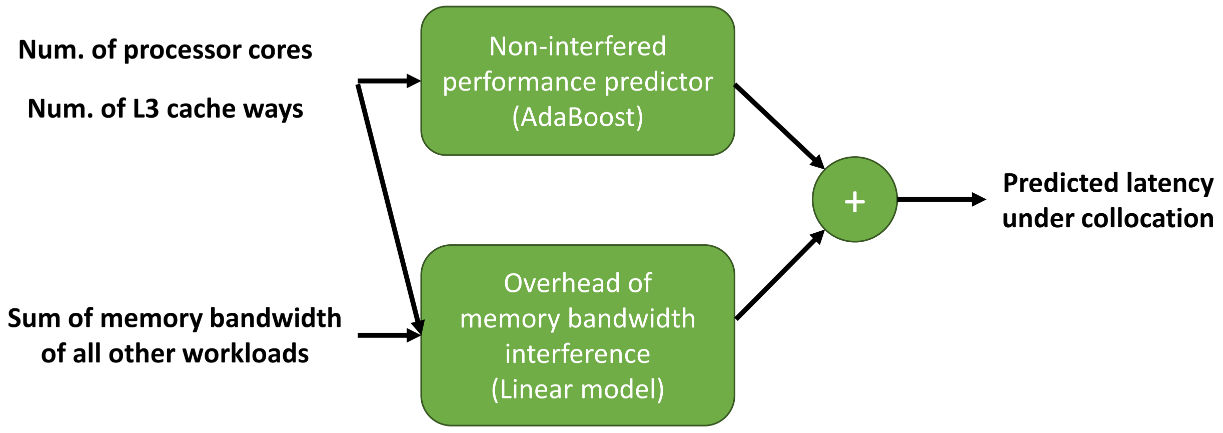

Num. of processor cores
Num. of L3 cache ways
Sum of memory bandwidth of all other workloads
Non-interfered performance predictor (AdaBoost)
Overhead of memory bandwidth interference (Linear model)
+
Predicted latency under collocation

Figure 6.2: Performance predictor for a DNN workload.

6.3 Predictor for Latency

6.3.1 Prediction Model

The performance predictor generates the predicted latency of a DNN inference, given the inputs of the allocated resource and the information from the collocated workloads. The overall structure of the prediction model is in Figure 6.2.

In the first stage of the predictor, an AdaBoost regressor model takes in the inputs of the number of cores and the number of ways in the L3 cache allocated to the workload, and outputs the predicted latency of this workload under this specific core and cache resource configuration. The prediction in this stage assumes a perfect isolation mechanism that prevents any interference by other collocated workloads so no information from other workloads is needed. The AdaBoost regressor model is empirically chosen for this stage of prediction, mainly because it can capture the relationship between the latency and the resource with relative high accuracy, even with a significant portion of missing data.

The second stage of the predictor is to adjust the predicted latency by taking account of the interference on memory bandwidth caused by collocated workloads. In contrast to the core and cache resources, there are no available mechanisms on commercial hardware to completely isolate the memory bandwidth across multiple processes. Therefore it would be not accurate to predict the latency of one DNN workload without considering the other collocated workloads in terms of memory bandwidth.

The intrinsic mechanisms on how the memory requests by one workload affects other ones are complicated. The level of performance degradation caused by memory request conflicts can depend on the exact timings of the memory requests from all the collocated workloads, which is neither practical to capture nor to predict entirely. But one metric, the average memory bandwidth, can approximately describe the level of memory request interference caused by a workload.

Therefore, the second stage takes in the input of the aggregated memory bandwidth of all other collocated workloads. Then we adopt a simple linear model, to predict the extra latency with regard to the total interfering memory bandwidth based on empirical observations. Note that to reduce the time spent on the profiling procedure, we use the sum of memory bandwidth of each individual workload under no collocation to replace the actual aggregated memory bandwidth during collocation. The interfering memory bandwidth is capped at the maximum system memory bandwidth. The coefficient of the linear model is to be learned by the training procedure. Note that this coefficient would vary for different resource configuration: when the other resources (core and cache) are abundant, the performance is usually less likely to be negatively affected by the contention on memory bandwidth.

The performance predictor described above has the assumption that the latency does not vary for different inputs. Some sequence-based DNN models, such as RNN-Transducer and Transformer, may have significantly varied inference latency due to the varying length of the input. While this scenario is not covered by the proposed performance predictor, it can be potentially accommodated by an updated version: under the otherwise same conditions, the latency of the sequence-based models has an approximately linear relationship to the length of the input sequence. This updated version of the prediction model and its effects on resource allocation is a future direction of improvement.

6.3.2 Training Procedure

The training of the prediction model is done sequentially for the two stages.

First, we train the AdaBoost regressor by providing training samples obtained from profiling. Each training sample consists of these fields: the number of cores, the number of cache ways, and the measured average latency. The number of training samples can be tuned based on a design trade-off: a larger number of training samples leads to higher accuracy, at the cost of more profiling time. Practically, one rule can be applied when choosing the configuration for profiling: to include denser data points where the resource level is relatively low, as in this case a change in the resource count causes a larger change in the latency.

Second, we train the linear model for the additional latency related to memory bandwidth interference. For this stage of training, we need profiling samples that measure the latency of the target workload under different interfering memory bandwidths. The collocated workload should include a wide range of memory bandwidths to make the estimated coefficients more accurate. Then a linear regression is performed on the training sample data of interfering memory bandwidth and the measured latency, and the estimated variable coefficient is taken to our prediction model.

Note that the above training procedure should be done separately for different core and cache configurations. An alternative procedure is to use a representative coefficient for a range of core and cache resource level, so that the number of required profilings is reduced.

6.4 Evaluation Methodology

6.4.1 Hardware and Software Framework Setup

We evaluate DQG on a server-client setup. We use a dual-socket CPU machine as the server, with its detailed hardware configurations specified in Table 6.2. We use additional CPU nodes as the clients, which send DNN inference requests to the server. The service latency

Table 6.2: Server configuration

Component	Specification
CPU model	Dual-socket Intel Xeon Platinum 8259CL
frequency	2.5GHz
Number of logical cores	48
L3 cache size	36MB, 11-way set associative
Memory capacity	396 GB
Operating System	Linux, kernel 5.15.0-1017-aws

is measured from the client side. We use TensorFlow Serving as the software framework to support the DNN inference service on the server machine.

6.4.2 Workloads

The DNN models used as latency-critical workloads are taken from the MLPerf benchmark suite (Reddi et al., 2020). Table 6.3 lists the DNN workloads used for experiments. The best-effort workloads consist of resource-thrashing microbenchmarks and DNN models (and variants) used simultaneously as latency-critical workloads. The processor-core-thrashing microbenchmark is from the stress testing suite stress-ng (King, 2017) that keeps busy all the 96 logical cores; the memory bandwidth-thrashing microbenchmark is from the iBench suite (Delimitrou and Kozyrakis, 2013), and we set it up to reach the maximum achievable memory bandwidth on the tested server. For simplicity, we set a unified target QoS for all the latency-critical workloads for each best-effort workload setting. These target QoS and the standalone memory bandwidth for BE workloads, are summarized in Table 6.4. We randomly choose from a set of representative inputs for the DNN workloads. Note that for the sequence-based models, we choose to use inputs with similar length to avoid extra uncertainty in performance.

Table 6.3: List of DNN models as latency-critical workloads

Application domain	**Model name**	**Model architecture/components**
Image classification	ResNet-50	convolutional neural network
Object detection	Single Shot Detector	convolutional neural network
Natural Language Processing	BERT	general matrix multiplication
Machine Translation	Transformer	general matrix multiplication

Table 6.4: Experiment Configuration by BE workloads

BE Workload(s)	**Total memory bandwidth** (GB/s)	**QoS Target of LC workloads** (extra latency percentage)
DNN (ResNet + Single Shot Detector)	75.5	20%
Processor core stressor	1.2	5%
Memory bandwidth stressor	203.1	40%

6.4.3 Settings for Profiling and Partitioning

In profiling, we measured for 12 different numbers of cores (out of 24 possible) and 6 different numbers of cache ways (out of 11 possible). To train for the memory bandwidth interference coefficient, we use the ResNet-50 (with input batch size 8) and the Transformer as the background interfering workloads. We use 3 representative memory bandwidth interference coefficients for each workload, with a medium-level cache way number, and high-, medium- and low-level core number, respectively.

The server CPU has 24 physical cores (48 logical cores) each socket. The granularity for core partitioning is 4 logical cores, which is composed of two logical cores from one physical core at each socket. The server CPU has 11 ways in last-level cache and the granularity for cache partitioning is 1 way.

We only consider the partition of core and cache way resources as other resources including memory capacity and disk bandwidth do not cause contention for the tested workload combination. In that case, the additional factors will be included in the predictor model and the profiling procedure.

6.5 Results

6.5.1 Collocation with Other DNN Workloads

Figure 6.3 shows the results of DQG, compared against a baseline of unmanaged collocation, and the predicted latency results under the partition. In the unmanaged collocation, all the LC and BE workloads are run simultaneously without resource isolation. The results are represented as the speedup against the non-collocated case, i.e. when only a single workload is running on the server. The speedup is less than 1 as the collocation increases the latency.

The DQG improves the latency by an average of 11% across the tested DNN workloads, benefiting from the resource isolation that guarantees resource for LC workloads. The average prediction error is 5%, and 100% of the workloads meet their QoS targets. Note that the predicted result does not always precisely correspond to the target QoS, as there is not always an exact match within the prediction results from all the partitions. Among the four workloads, an average of 86% of requests satisfied the QoS target. This ratio can be further improved by using tail latency to build the predictors, as the target currently set is for average latency. The BE workloads receive a performance penalty compared to non-collocated case (3.9× on average for the DNN mix measured). But DQG is still effective in improving BE workloads when compared with the partition that gives only a minimum amount of resource to the BE workloads. On average, in DQG, the ResNet BE workload achieves a 1.9× speedup over the above fixed partitioning strategy, and the Single Shot Detector BE workload achieves a 8.4× speedup.

6.5.2 Collocation with Core Stressing Microbenchmarks

Figure 6.4 shows the effects of DQG when core thrashing microbenchmarks are used as BE workload. DQG achieves a 1.9× speedup over baseline. Compared to DNN workloads, core thrashing benchmarks take up even more core resources when running as BE workloads, which significantly reduces the amount of processor time allocated to the LC workloads.

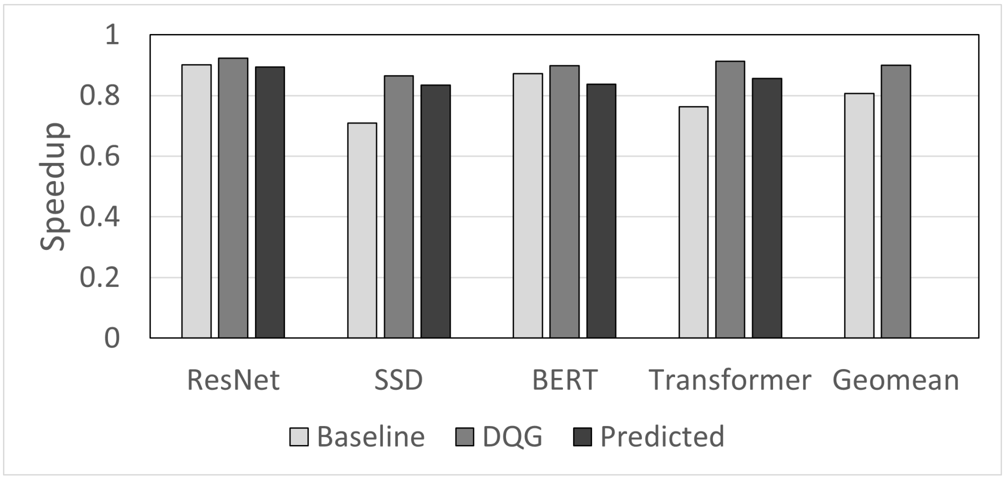


Figure 6.3: DQG Result with DNNs as BE workloads.

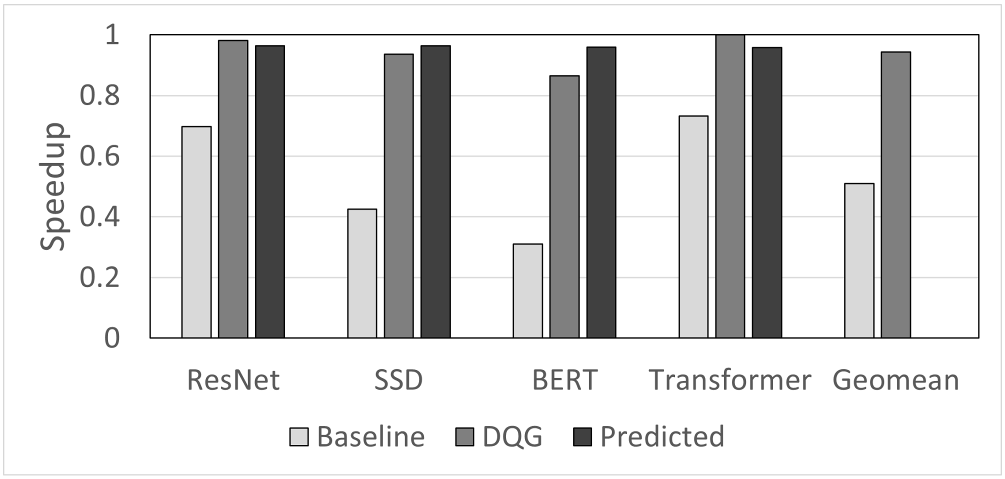


Figure 6.4: DQG Result when collocated with core-thrashing microbenchmarks.

Therefore, core resource partitioning brings more advantages in protecting LC workloads.

6.5.3 Collocation with Memory-Bandwidth Stressing Microbenchmarks

The results with memory-bandwidth thrashing microbenchmarks as BE workloads are shown in Figure 6.5. The average improvement of DQG over baseline is 26%. We notice that the accuracy of latency prediction here is relatively lower, with an error rate of 17%. This is mainly due to the fact that the coefficients for memory bandwidth interference are not

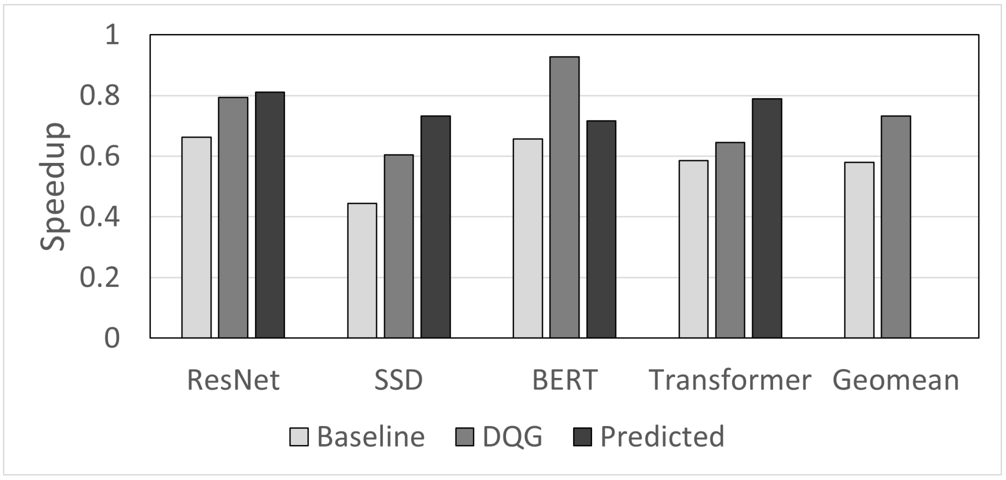


Figure 6.5: DQG Result when collocated with memory-bandwidth-thrashing microbenchmarks.

estimated accurately. Combined with high interfering memory bandwidth, the final predicted latency can have a larger error. This can be mitigated by collecting more coefficients for different core and cache way combinations.

6.6 Summary

We propose a mechanism to protect the QoS of DNN workloads under collocation, based on the latency prediction of DNNs. We design a DNN latency predictor which takes in factors of CPU core and cache resources as well as the interfering workloads' memory bandwidth. During runtime, the proposed mechanism chooses the resource partition that is predicted to meet the QoS target of DNN workloads, while not taking up resources excessively. The evaluated results show a 11% of average speed improvements for the DNN workloads, compared to the OS default management.

CHAPTER 7

Conclusion

Deep Neural Networks have been a highly popular application for modern computing systems and the frontier of DNNs keeps evolving with new models being proposed. Today the DNN workloads in datacenter run on a variety of hardware platforms such as CPU, GPU, ASIC, and FPGA. The different platforms have distinct strengths to improve the performance of the workloads: CPUs have a very large memory capacity; GPUs have a high computing throughput for SIMD-patterned computation; ASICs have the highest efficiency for several specialized types of computation; FPGAs provides a customized logic to map algorithms and can be easily reconfigured. Beyond traditional architectures, in-memory computing is a special data-centric computing diagram that reduces data movement by bringing the computing units closer to the data storage. It is a promising candidate for accelerating DNN workloads as the weights can be pinned to memory and the it offers massive parallelism for the highly-parallelizable compute kernels. There are also software methods to improve the speed of DNN. The model compression, which mainly includes weight pruning and data quantization, is practically popular due to their relatively easy implementation. This dissertation observes challenges and opportunities within the intersections of hardware platforms, computing diagrams, and software model compression techniques, with the end goal of improving DNN computing systems.

First, we improve on an existing in-SRAM DNN accelerator which reuses the last level cache of CPUs. We observe its inefficiency in handling sparse model and low-bitwidth data format. We take a software/hardware co-design approach to tackle the sparse data efficiency problem, by proposing channelwise weight pruning methods that match the underlying vectorized architecture. We further develop specialized bit-serial algorithms for the low-bitwidth data. This method results in an even more efficient in-SRAM DNN accelerator with a reduced weight size while also offering the flexibility in the accuracy-speed trade-off with different model compression techniques and different levels of pruning rates. We believe this work can be inspiring for other in-memory computing accelerators with bit-line ALUs, in supporting model compression techniques.

Second, we enable the BRAMs in FPGAs with computing capabilities so that these compute-capable BRAMs can be used for DNN acceleration on FPGAs. Taking the approach of in-SRAM bit-serial computing, we present how the BRAM blocks can be modified without interface change in a small area footprint. We build DNN accelerators using the dual-functional BRAM along with other LUT and DSP blocks on FPGA. This work bridges the in-memory computing to the FPGAs, and further enables a more reconfigurable platform for in-SRAM computing. We have shown that the proposed reconfigurable in-memory accelerator can further speedup on the Brainwave DNN instances, to bring more performance benefits of using FPGAs as the platform. For the FPGA community, this work also changes the traditional way of how BRAMs are used, and such additional computing throughput on FPGAs has the potential to accelerate workloads with other compute patterns, even beyond DNNs.

Third, we propose a novel cache compression technique based on in-memory computing for data comparison. The quick comparison enables wide range of compression opportunity and the decompression procedure is short thanks to the awareness of the cache microarchitecture. This work benefits any cache-sensitive workloads on CPU including many of DNN workloads.

Finally, we propose a mechanism to protect the QoS of DNN workloads on CPU servers. We design a performance predictor for DNN latency under resource partition, and accordingly adjust the resource at runtime. This leads to the protected QoS of DNNs with other best-effort workloads collocated, which outperforms the OS default workload scheduler. This proposed mechanism would encourage more use cases of running DNN workloads on CPU-based servers when the lowest possible latency is not a priority. We envision that future works can develop more complicated performance predictors that support more types of DNN models, more performance-related metrics, and more flexible of runtime settings such as different request rates. Also, for DNN running on heterogeneous platforms including GPU/ASICs, similar performance prediction techniques can be applied to determine an optimal allocation of the concurrent workloads onto different platforms. One interesting extension is for the type of in-cache DNN accelerator described in previous chapters of this dissertation. The cache slices and ways can be partitioned to different DNN models for collocated computing. After such partition, the SRAM-array computing resources are isolated across DNNs but the memory bandwidth can still cause interference. The proposed performance predictor needs to be modified in two ways. First, for the non-interfered latency prediction, there is no need to profile for different cache allocation configurations as the latency can be directly generated by the analytical model. Second, the memory bandwidth interference should include two parts: the off-chip data access bandwidth for loading the weights, and the data access bandwidth within each cache slice for duplicating the input activations. Both can be obtained from analytical model given the DNN architecture. A complete design for such extension is left for future work.

In summary, this dissertation improves the efficiency of DNN workloads running on multiple platforms: in-memory computing accelerator, FPGA, and multi-core CPU. To design better platforms for DNNs under different goals is a multi-faceted problem calling for further research works.

BIBLIOGRAPHY

S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute caches. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 481–492. IEEE, 2017.

A. Alameldeen and D. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2004.

J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 1–13. IEEE, 2016.

J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 382–394. ACM, 2017.

M. Alwani, H. Chen, M. Ferdman, and P. Milder. Fused-layer cnn accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 22. IEEE Press, 2016.

A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 715–731. ACM, 2019.

A. Arelakis and P. Stenstrom. Sc 2: A statistical compression cache scheme. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 145–156. IEEE, 2014.

A. Arora et al. Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020.

AWS. Amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1. Accessed: 2019-11-22.

U. Aydonat et al. An OpenCL Deep Learning Accelerator on Arria 10. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

E. Baek, D. Kwon, and J. Kim. A multi-neural network acceleration architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 940–953. IEEE, 2020.

Baidu. Baidu deepbench. https://github.com/baidu-research/DeepBench. Accessed: 2019-11-22.

A. Boutros et al. Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

A. Boutros et al. Math Doesn't Have to be Hard: Logic Block Architectures to Enhance Low-Precision Multiply-Accumulate on FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

W. J. Bowhill, B. A. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, O. Mendoza, C. Morganti, et al. The xeon® processor E5-2600 v3: a 22 nm 18-core product family. *J. Solid-State Circuits*, 51(1):92–104, 2016.

A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengil, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 7. IEEE Press, 2016.

S. Chen, C. Delimitrou, and J. F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

W. Chen, S.-L. Chen, S. Chiu, R. Ganesan, V. Lukka, W. W. Mar, and S. Rusu. A 22nm 2.5 mb slice on-die l3 cache for the next generation xeon® processor. In *VLSI Technology (VLSIT), 2013 Symposium on*, pages C132–C133. IEEE, 2013.

X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *IEEE transactions on very large scale integration (VLSI) systems*, 18(8):1196–1208, 2009.

Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Computer Architecture (ISCA), ACM/IEEE 43rd International Symposium on*, pages 367–379. IEEE, 2016.

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, 2017.

P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 27–39. IEEE Press, 2016.

P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis. Exploration of low numeric precision deep learning inference using intel® fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–80. IEEE, 2018.

H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: memory power estimation and capping. In *Low-Power Electronics and Design, 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE, 2010.

C. Delimitrou and C. Kozyrakis. ibench: Quantifying interference for datacenter applications. In *2013 IEEE international symposium on workload characterization (IISWC)*, pages 23–33. IEEE, 2013.

C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan. Tie: energy-efficient tensor train-based inference engine for deep neural network. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 264–278. ACM, 2019.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

M. Drumond et al. Training DNNs with Hybrid Block Floating Point. *Advances in Neural Information Processing Systems (NeurIPS)*, 31:453–463, 2018.

Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *Proceedings of the 23rd international conference on Supercomputing*, pages 46–55, 2009.

C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 383–396. IEEE Press, 2018.

M. Eldafrawy et al. FPGA Logic Block Architectures for Efficient Deep Learning Inference. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(3):1–34, 2020.

K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 148–157. IEEE, 2002.

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

M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764, 2017.

A. Ghasemazar, P. Nair, and M. Lis. Thesaurus: Efficient cache compression via dynamic clustering. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 527–540, 2020.

Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159. IEEE, 2017.

G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

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

S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. IEEE, 2015.

K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 674–687. IEEE Press, 2018.

W. Heirman, T. Carlson, and L. Eeckhout. Sniper: Scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, pages 91–94, 2012.

M. Huang, M. Mehalel, R. Arvapalli, and S. He. An energy efficient 32-nm 20-mb shared on-die L3 cache for intel® xeon® processor E5 family. *J. Solid-State Circuits*, 48(8): 1954–1962, 2013.

W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro*, 31(4):16–29, 2011.

M. Imani, S. Gupta, Y. Kim, and T. Rosing. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 802–815. ACM, 2019.

Intel® Stratix® 10 GX/SX Device Overview. Intel Corporation, 2019.

Intel® Quartus® Prime Pro Edition User Guide: Design Recommendations. Intel Corporation, 2020.

S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 51(4):1009–1021, 2016.

Y. Ji, Y. Zhang, X. Xie, S. Li, P. Wang, X. Hu, Y. Zhang, and Y. Xie. Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 733–747. ACM, 2019.

N. P. Jouppi, C. Young, N. Patil, D. Patterson, 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. ACM, 2017.

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

R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *Proceedings of ISCA*, volume 43, 2016.

C. I. King. Stress-ng. *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)*, 2017.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 288–301. ACM, 2017.

S. Li, A. O. Glova, X. Hu, P. Gu, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie. Scope: A stochastic computing engine for dram-based in-situ accelerator. In *MICRO*, pages 696–709, 2018.

Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, and Y. Wang. E-rnn: Design optimization for efficient recurrent neural networks in fpgas. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 69–80. IEEE, 2019.

W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, 2019.

D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang. An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–25. IEEE, 2019.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54. ACM, 2017.

Micron Technology Inc. Calculating memory power for ddr4 sdram. Technical Report TN-40-07, 2017.

A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr. WRPN: Wide reduced-precision networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=B1ZvaaeAZ.

R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179. IEEE, 2020.

E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, R. Krishnamurthy, S. Girbok, B. Pasca, M. Langhammar, D. Marr, and A. Dasu. Why compete when you can work together: Fpga-asic integration for persistent rnns. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 199–207. IEEE, 2019.

A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of 44th International Symposium on Computer Architecture*, pages 27–40. ACM, 2017.

T. Patel and D. Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 377–388. IEEE, 2012.

A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014. doi: 10.1109/ISCA.2014.6853195.

K. Rao, H. Sak, and R. Prabhavalkar. Exploring architectures, data and units for streaming end-to-end speech recognition with rnn-transducer. In *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 193–199. IEEE, 2017.

S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. Leong. Pir-dsp: An fpga dsp block architecture for multi-precision deep neural networks. In *2019 IEEE 27th Annual International*

Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 35–44. IEEE, 2019.

M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

M. Samragh, M. Ghasemzadeh, and F. Koushanfar. Customizing neural networks for efficient fpga implementation. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 85–92. IEEE, 2017.

S. Sardashti and D. A. Wood. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 62–73. IEEE, 2013.

S. Sardashti, A. Seznec, and D. A. Wood. Skewed compressed caches. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 331–342. IEEE, 2014.

V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 273–287. ACM, 2017.

A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 14–26. IEEE Press, 2016.

Y. Shen, M. Ferdman, and P. Milder. Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–100. IEEE, 2017a.

Y. Shen, M. Ferdman, and P. Milder. Maximizing cnn accelerator efficiency through resource partitioning. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 535–547. IEEE, 2017b.

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

P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag. Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56. IEEE, 2018.

A. Subramaniyan, J. Wang, E. R. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das. Cache automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 259–272. ACM, 2017.

V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

K. Tatsumura et al. High Density, Low Energy, Magnetic Tunnel Junction Based Block RAMs for Memory-Rich FPGAs. In *IEEE International Conference on Field-Programmable Technology (FPT)*, 2016.

K. Tatsumura et al. Enhancing FPGAs with Magnetic Tunnel Junction-Based Block RAMs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(1):1–22, 2018.

Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh. Last-level cache deduplication. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 53–62, 2014.

P.-A. Tsai and D. Sanchez. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–242, 2019.

H. Valavi, P. J. Ramadge, E. Nestler, and N. Verma. A mixed-signal binarized convolutional-neural-network accelerator integrating dense weight storage and multiplication for reduced data movement. In *2018 IEEE Symposium on VLSI Circuits*, pages 141–142. IEEE, 2018.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

S. I. Venieris and C.-S. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47. IEEE, 2016.

J. Wang et al. 14.2 A Compute SRAM with Bit-Serial Integer/Floating-Point Operations for Programmable In-Memory Vector Acceleration. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2019a.

S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 11–20. ACM, 2018.

X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das. Bit prudent in-cache acceleration of deep convolutional neural networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 81–93. IEEE, 2019b.

X. Wang, C. Augustine, E. Nurvitadhi, R. Iyer, L. Zhao, and R. Das. Cache compression with efficient in-sram data comparison. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–8. IEEE, 2021a.

X. Wang, V. Goyal, J. Yu, V. Bertacco, A. Boutros, E. Nurvitadhi, C. Augustine, R. Iyer, and R. Das. Compute-capable block rams for efficient deep learning acceleration on fpgas. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 88–96. IEEE, 2021b.

W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082, 2016.

S. Yazdanshenas et al. Don't Forget the Memory: Automatic Block RAM Modelling, Optimization, and Architecture Exploration. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 548–560. IEEE, 2017.

Y. Zha and J. Li. Liquid silicon-monona: A reconfigurable memory-oriented computing fabric with scalable multi-context support. In *ACM SIGPLAN Notices*, volume 53, pages 214–228. ACM, 2018.

S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *Microarchitecture (MICRO), 49th International Symposium on*, pages 1–12. IEEE, 2016.

X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu. Towards memory friendly long-short term memory networks (lstms) on mobile gpus. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 162–174. IEEE, 2018.

Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 406–418. IEEE, 2014.

S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez. Kelp: Qos for accelerated machine learning systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 172–184. IEEE, 2019.