CoExe: An Efficient Co-execution Architecture for Real-Time Neural Network Services

Chubo Liu*, Kenli Li*, Mingcong Song†, Jiechen Zhao†, Keqin Li‡, Tao Li†, Zihao Zeng*
*College of Information Science and Engineering, Hunan University, China
†Department of Electrical and Computer Engineering, University of Florida, USA
‡Department of Computer Science, State University of New York, USA
{liuchubo,lkl,zengzh}@hnu.edu.cn, {songmingcong,jiechen.zhao}@ufl.edu, lik@newpaltz.edu, taoli@ece.ufl.edu

***Abstract*—End-to-end latency is sensitive for user-interactive neural network (NN) services on clouds. For periods of high request load, co-locating multiple NN requests has the potential to reduce end-to-end latency. However, current batch-based accelerators lack request-level parallelism support, leaving the queuing time non-optimized. Meanwhile, naively partitioning resources for simultaneous requests suffers from longer execution time as well as lower resource efficiency because different applications utilize separate resources without sharing. To effectively reduce the end-to-end latency for real-time NN requests, we propose CoExe architecture, equipped with a pipeline implementation of a sparsity-driven real-time co-execution model. By leveraging the non-trivial amount of sparse operations during concurrent NNs execution, the end-to-end latency is decreased by up to 12.3× and 2.4× over Eyeriss-like and SCNN at peak workload mode. Besides, we propose row cross (RC) dataflow to reduce data movement cost, and avoid memory duplication.**

***Index Terms*—AI-cloud acceleration, Co-execution architecture, Real-time NN service, Sparsity-driven multi-context.**

I. Introduction

Nowadays, cloud-based machine learning products and solutions are quickly emerging as commonplace and are shaping our experiences in computing like no other time in history. Interactive speech (e.g., Alexa, Google Home, etc.), visual search and recommendation engines are just a few of the consumer applications that are available today. The impact of machine learning is getting broader with enterprise applications in health sciences (e.g., Dr. Watson), finance and security, forming the next wave that will drive the growth of public cloud. As a result, cloud providers are gearing up to offer a comprehensive stack that delivers AI as a Service [1].

Training NN models primarily demands throughput-oriented techniques (e.g., GPU and batching) and is insensitive to the latency of processing a single input. In contrast, inference (evaluating a pre-trained model) requests, such as real-time video processing for self-driving cars, are typically user-facing and require high-priority, low-latency, and smooth responses. For the cloud-based real-time inference services, requests often arrive one at a time and need to be processed upon arrivals (with batch size of 1). This results in lower resource utilization as well as energy efficiency on throughput oriented hardware such as GPU and specialized AI accelerators, which typically leverage batch to aggregate underutilized resources.

In this paper, we argue that although existing NN acceleration designs synergistically fuel NNs′ particular computation pattern and high loop-based concurrency (matrix-matrix or matrix-vector multiplication), request-level parallelism support has not been provisioned to reduce the queuing delay, which accounts for the dominant fraction (e.g., 80%-97%) of end-to-end latency [2]. Moreover, the cost-efficiency is worse considering that multi-request co-execution increases hardware complexity and reduces benefits of data reuse. In this paper, our goal is to enable request-level parallelism architecture support to minimize queuing time, while avoiding the significant increase of execution time when co-locating real-time NN requests.

To this end, we propose CoExe, which can effectively co-locate more than one latency-critical and asynchronous NN requests. Our proposed techniques are capable of harnessing the non-trivial amount of sparse cycles (i.e., the significant amount of zero-related multiply-accumulate operations are ineffectual) during each NN execution to process the effectual operations from other requests, increasing the inter-request resource sharing and mitigating the performance interference. Also, CoExe adopts a flexible and cost-efficient design with sparsity-driven execution model, which allocates hardware resources to process only the effectual multiply-accumulations (MACs) from more than one NN inference tasks. Thus, workloads consolidate resources, and higher utilization and cost-efficiency are achieved. Finally, we establish a row cross (RC) dataflow, which can significantly reduce data movement to improve efficiency by fully exploiting input/weight/sum reuse.

Our experimental results show that CoExe delivers improved latency over Eyeriss-like and SCNN architectures. Our proposed sparsity-driven execution model replenishes sparsity instead of eliminating them. Thus, data patterns are regular and quality of service (QoS) is more manageable under multi-context execution. We believe that our work will open opportunities for exploiting the request-level parallelism for cloud-based, real-time NN service acceleration.

II. Background and Related Work

A. Real-time AI-inference Service Features Make Traditional Architectures Inefficient

Extensive accelerators have been designed for NNs. Generally, the techniques can be divided into three categories (see Table I), i.e., ① Input/Weight Sharing [3], [4], ② Compression/Skipping Zero Operations [5]–[8], and ③ Memory Related Acceleration (e.g., processing in memory, hybrid

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

Techniques in Current Accelerators		
Input/Weight Sharing ①	**Compression/Skipping Zero Operations ②**	**Memory Related ③**
FlexFlow [3]	EIE [5], PERMDNN [11], Cambricon-X [6], Cambricon-S [7], SCNN [8]	PipeLayer [9], ZARA [12]
	LerGAN [10]	
ZFOST-ZFWST [4]		

Characteristics of Cloud AI-inference Service	
Small Batch Size and No Back-propagation	①②③
Asynchronous Arrival Pattern	①②③
Flexible Requests for Different NNs	①②③

TABLE I: Common Techniques of Current Accelerators and the Special Characteristics of Cloud AI-inference Service

memory cube) [9], [10]. However, almost all existing accelerators are designed for batch-based processing, which are not appropriate or can not efficiently handle real-time AI-inference services, which exhibit: small batch size and no-propagation, asynchronous arrival pattern, and flexible requests for different NNs.

For instance, different from batch-based AI processing which simultaneously processes a batch of inputs, the batch size of a real-time AI-inference request is usually one due to its asynchronous arrival pattern, e.g., finding a person′s identification by leveraging a camera photo. This prevents the coarse-grained input/weight sharing or compression among inputs. Also, there is no back-propagation update, which requires less memory. Last but not the least, in batched AI processing, extensive amount of inputs are used by one NN. However, in real-time AI-inference service, different requests may be served by different NNs due to accuracy or suitability. This further degrades the effectiveness of traditional accelerators.

B. Opportunities: Sparsity-Driven Co-Execution

Multiply-accumulate (MAC) operations account for more than 95% among the operations of NN execution [13]. As prior studies show, eliminating zero-related ineffectual MAC operations provide potential performance as well as energy efficiency enhancement due to the non-trivial ratio of ineffectual operations. For instance, VGGNet has about 70% of input zeros across all convolutional layers. Even applying the state-of-the-art pruning technique [7], the fraction of weight zeros is still up to 65%. Overall, the effectual MAC operations account for 15% total MAC operations on VGGNet.

In addition, we observed that the resource utilization of a NN for inferring a single input is low. The accelerator and memory utilizations range from 7% to 20% and 3% to 16% during the single inference processes of LeNet-5, AlexNet, and VGG16. Such a low resource utilization indicates another opportunity for co-executing NNs.

Unlike existing approaches that subtract unnecessary operations, we add multiple NN tasks into sparse cycles for concurrency and effectual throughput. Our goal is to reduce queuing time while achieving acceptable execution time and reasonable overhead, and consequently improve latency and accelerator efficiency in cloud-based real-time NN services. To this end, we leverage ineffectual (sparse) cycles to share

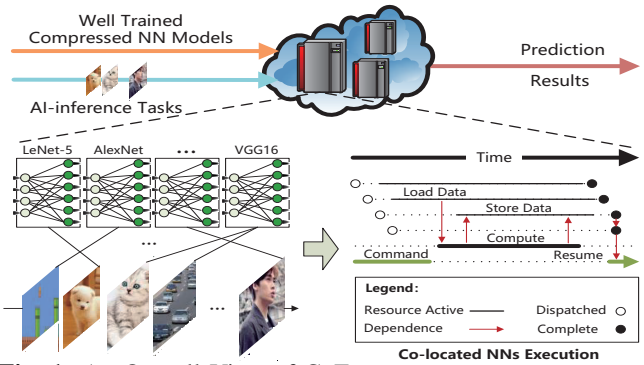


Fig. 1: An Overall View of CoExe

physical processing elements (PEs) among multiple asynchronous NN requests, and explore request-level co-execution architecture support.

III. An Overall View and Design of CoExe

Fig. 1 illustrates the overall view of our proposed CoExe for accelerating cloud-based real-time AI-inference service. To increase resource utilization while maintaining flexibility, NN models (weight matrixes) are compressed after well training and pruning, and each cloud server is equipped with multiple different NN models. AI-inference tasks arrive at different time and may choose different NN models for services. CoExe enables this kind of asynchronous execution of NNs by efficiently sharing storage and compute resources.

A. Sparsity-Driven Multi-Context Execution Model

Though zeros in NN models are avoided to a great extent, zeros still exist in convolutional layer′s computation. Fig. 2 shows the percentages of zeros in different convolutional layers′ inputs of LeNet-5, AlexNet, and VGG16. We can observe that on average the percentage of zeros is around 50% across all three NNs. To skip these zero operations, we propose a sparsity-driven multi-context (SD-MC) execution model described using a 2-context case as an example, which involves three cases.

Singular-Effectual (SE) Case: As shown in Fig. 3(a), if only one of the concurrent contexts is effectual, it directly issues the effectual context′s operands to the multiplier. Then, the calculated partial sum (psum) is written back into the memory corresponding to its context. During SE case, ineffectual context does not execute, yet their input (I) and weight (W) operand pair is still consumed in this cycle. Thus, counters of both contexts increase by 1 because both contexts make progress in this cycle.

Contention-Effectual (CE) Case: If more than one context is effectual, contexts are contending for computing resources (i.e., the multiplier in Fig. 3(b)). We select the executing context from the multiple contexts using a round-robin policy. After execution, the result is written back correspondingly. On the other hand, the non-executing context′s operand pair is stored in register files (RF), which works in FIFO manner. Each entry of RF has 1-bit tag, recording which context′s operands are stored. The store is in parallel with the execution of the selected context, meaning that it does not delay the execution.

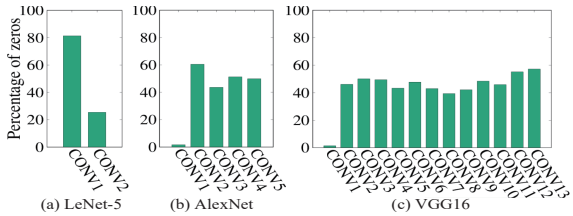


Fig. 2: Percentage of Zeros in Feature Inputs of NNs

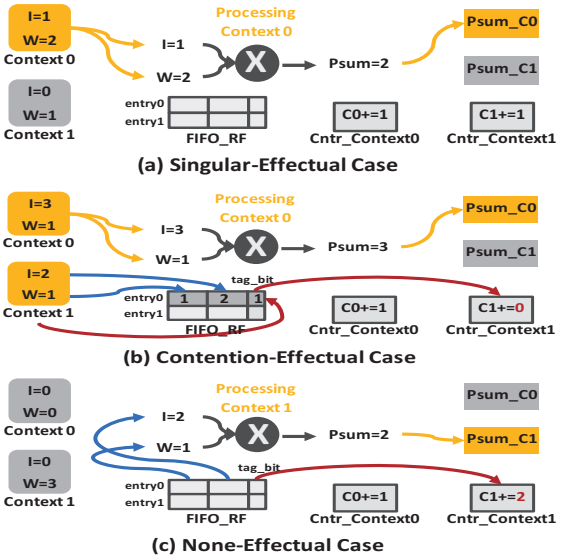


Fig. 3: Sparsity-Driven Execution Model

None-Effectual (NE) Case: If all contexts are ineffectual in this cycle (see Fig. 3(c)), one pre-stored operand pair in RF is issued to the multiplier, and the psum result is written into the psum context according to the tag. If all RF entries are filled, no load is performed but RF issues its first entry of operands for execution. The counter corresponding to the context issued from the RF increases by 2, denoting this context progresses one ineffectual MAC and one effectual MAC. The other counter increases by 1.

Note that for illustration purpose, we mainly discuss a {2-context, 1-multiplier, 4-entry FIFO_RF} execution model as well as its related dataflow and architecture across this paper. Nevertheless, our execution model could be flexibly extended to a general case of {*C*-context, *M*-multiplier and *E*-entry FIFO_RF} according to different AI service scenarios and their workload combinations.

B. Pipelined Sparsity-Driven PE Architecture

This section describes how a pipelined PE (PipePE) is designed based on the (SD-MC) execution model. Fig. 4 elaborates the design of each pipeline stage and its implemented micro-architectures are described as follows:

LOAD: In the 2-context PE architecture, each pipelined unit (PU) loads one pair of input and weight from each context′s memory. The PE first loads the compressed weights. Based on the weights, it selects the corresponding inputs from the input vector memory, and then loads the selected inputs.

BRANCH: BRANCH stage determines corresponding effectual cases. As shown in Fig. 5, each Branch Logic is implemented by three AND gates, five MUXs, and a 4-entry FIFO_RF component. Each pair of input and weight enters an AND gate (AND0 and AND1) to obtain a signal acknowledging whether there exists a zero in the input or weight (0: **true**, 1: **false**). Then, the outputs of the AND0 and AND1 formulate four results, i.e., 00, 01, 10, and 11. The 11 (i.e., AND2 outputs 1) denotes the two contexts are effectual. Under this circumstance, MUX4 selects a pair of 01 (Context0) or 10 (Context1) to output by a round-robin way, and MUX3 outputs the output pair of MUX4. Meanwhile, MUX1 selects the output pair of the other context to write to the FIFO_RF. Otherwise (i.e., AND2 outputs 0), MUX3 selects the output pair of AND0 and AND1. Overall, the outputs of MUX3 have three results, i.e, 00, 01, or 10. Both 01 (Context0) and 10 (Context1) are used to control MUX0 to select the effectual pair of input and weight for execution. The 00 denotes both of the two input pairs are ineffectual. Under this circumstance, MUX0 selects an input pair from the FIFO_RF for output. MUX2 is used to pre-signal which context the PU is processing, and is also controlled by the output of MUX3.

CALCULATE: The CALCULATE stage processes multiplications. The multipliers are pipelined to increase throughput, which also hides the latency of TRACK stage preconfiguration.

TRACK: Note that in a single cycle, different PUs may process different contexts. One example is shown in Fig. 4, where PU0 and PU2 are processing context0 (orange background) while PU1 and PU3 are executing context1 (blue background). Thus, for ensuring no inter-context interference and computation incorrectness, we implement TRACK stage with 4 1-2 DEMUXs, each corresponding to identify each PU′s current context. The control signals of DEMUXs are derived from BRANCH stage before the calculated psum arrives. Therefore, the control signal′s timing constraints of setup/hold could be satisfied without slowing down the pipeline clock rate. After filtrating each context′s psum, the TRACK stage sends psums into two dedicated adders for accumulation.

ADD and **WRITE BACK**: Context-separate adders accumulate different psums from different PUs, and the accumulated results are written into isolated psum memory of each context.

C. Efficient Dataflow

1) Convolution Primitive: Row Cross (RC): Our dataflow is based on a basic convolution operation which we call row cross (RC). It is inspired by a basic observation shown in Fig. 6. Take the convolution of weight channel 1 and input channel 1 in Fig. 6(a) as an example, when calculating psum in orange square, rows 2-3 in input channel and rows 1-2 in weight channel can be further used to calculate partial psums in red and purple squares. The interactions between these two rows of inputs and two rows of weights are abstracted in Fig. 6(b). We take this RC convolution as a primitive. Each primitive operates on two rows of inputs and two rows of weights, and generates three rows of partial psums.

2) Primitive Mapping and Recombination: Our dataflow is separated into two steps: primitive mapping and recombination.

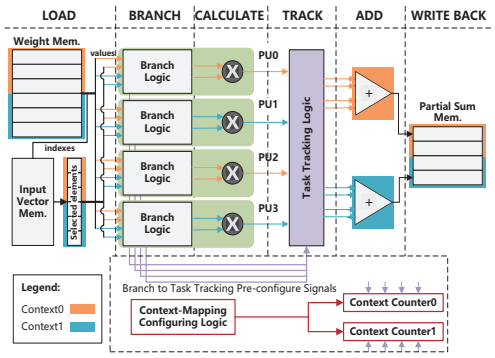
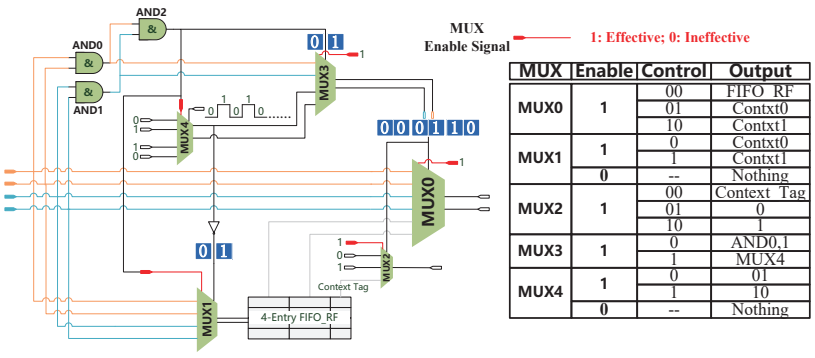


Fig. 4: Pipelined PE Architecture



MUX	Enable	Control	Output
MUX0	1	00	FIFO_RF
		01	Contxt0
		10	Contxt1
MUX1	1	0	Contxt0
		1	Contxt1
	0	--	Nothing
MUX2	1	00	Context_Tag
		01	0
		10	1
MUX3	1	0	AND0,1
		1	MUX4
MUX4	1	0	01
		1	10
	0	--	Nothing

Fig. 5: Branch Logic Architecture

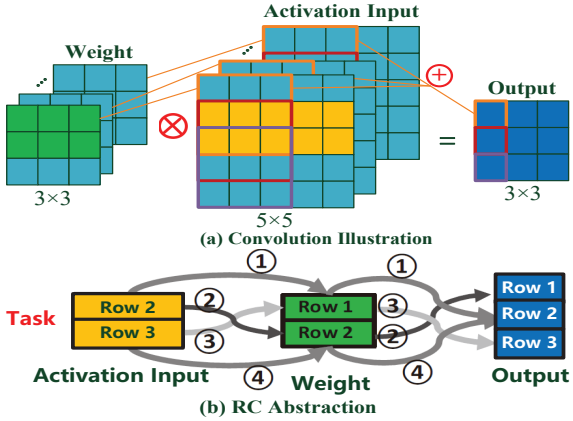


Fig. 6: RC Dataflow

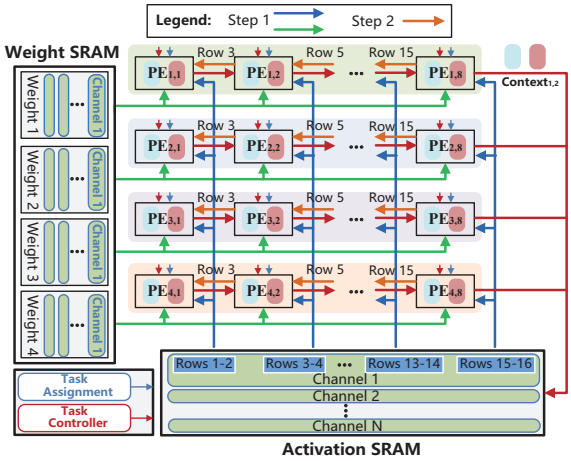


Fig. 7: Two-context CoExe Architecture

The former deploys the primitives into a PE array. Then the latter tries to increase inter-PE data reuse.

Primitive Mapping: We map primitives channel by channel and each primitive is mapped to one PE. Fig. 7 illustrates our PE array, where each weight channel is horizontally shared and each two-row of inputs in a channel is shared vertically, respectively. Hence, each row of PEs collaboratively calculate a channel of psums.

Primitive Recombination: Notice that in Fig. 7, rows in input channel are disjointedly processed by PE columns. For example, primitives involving rows 1-2, rows 3-4, rows 5-6, and so on, are respectively processed. However, primitives involving rows 2-3, rows 4-5, and so on, still need processing. To improve efficiency, we design a link between inter PEs (orange arrows in Fig. 7). Each PE just needs to receive an input row from adjacent PE for processing another primitive.

D. CoExe Architecture

Based on the SD-MC PE architecture and dataflow, we design our CoExe architecture shown as Fig. 7, which consists of 32 PEs with each row containing 8 PEs. Since weight filters are shared among PEs inside one row, a multicast bus connects PEs on the same row for sending filter weights (shown by green lines), and different filter weights are distributed across multiple PE rows. On the other hand, inputs of each channel are shared vertically and broadcasted by a bus shown as vertical blue lines. Since each row of PEs calculate psums for a same output channel, a bus connecting inter-PE in the same row is added (shown by red lines). There are global task assignment control units. Task assignment unit connects to the config. unit (shown in Fig. 4) in every PE. This assignment could control how tasks are mapped onto contexts and make counters in each PE track the progress of each context. The controller collects tasks′ progress from all PEs.

1) Hierarchical Data Sharing/Reuse: With our CoExe design, inputs, weights and psums sharing/reuse can be extensively and simultaneously exploited at different levels. We abstract the hierarchical data sharing/reuse opportunities in Fig. 8.

Task Process Level (T_P): Each process corresponds to a context and multiple processes can collaboratively work for a same task. Each process processes a partial of input channels and psums are reused for accumulating final outputs.

Task Channel Level (T_C): As illustrated in Fig. 7, all PEs process inputs from a same input channel, and inputs are vertically shared in this level.

Task Primitive Level (T_{PR}): Each primitive is mapped to a PE, and a weight channel is shared among a row of PEs (primitives). For inputs, as shown in Fig. 7, around half of input rows are directly reused from the adjacent PEs in Step 2. Also, since a row of primitives calculate for a same output channel, the psums within a row are reused for accumulating final results. Hence, in this level, the reuse/sharing of inputs, weights, and psums is simultaneously exploited.

Intra-T_{PR}: Inside a primitive, we adopt RC scheme. As shown in Section III-C1, RC is used to simultaneously calculate three rows of psums, in which the data reuse of the two rows of inputs and two rows of weights is fully exploited.

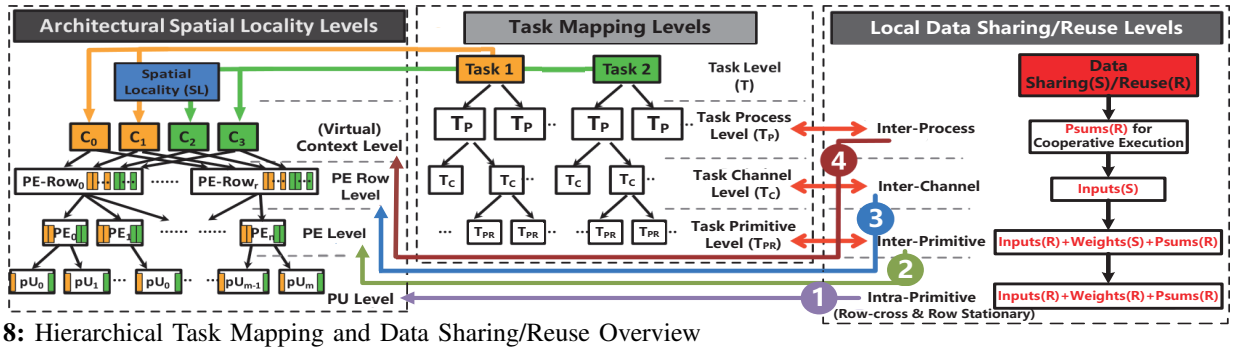


Fig. 8: Hierarchical Task Mapping and Data Sharing/Reuse Overview

Furthermore, since each RC involves four row pairs of inputs and weights′s calculation, they can be performed in parallel, and since they all calculate for the same three-row of psums, psums are also reused for accumulation. Therefore, in this level, the reuse/sharing of inputs, weights, and psums is also simultaneously exploited.

2) Memory in PE: We design our PE memory based on that of Eyeriss [14]. Though we consider multiple contexts, the memory overhead for a two-context PE is not doubled.

The cost saving stems from the following reasons: First, due to our RC design, each input/weight value is more efficiently utilized compared to Eyeriss. Second, to avoid doubling input/weight memory, we exploit one input channel at a time instead of unrolling four channels as the case of Eyeriss PE. Hence, we can derive the input memory $iMEM_{\text{COEXE}} = C \times \frac{RI_{\text{COEXE}}}{RI_{\text{Eyeriss}}} \times \frac{CN_{\text{COEXE}}}{CN_{\text{Eyeriss}}} \times iMEM_{\text{Eyeriss}} = 2 \times \frac{2}{1} \times \frac{1}{4} \times iMEM_{\text{Eyeriss}} = iMEM_{\text{Eyeriss}}$, where C, RI and CN denote the numbers of contexts, input rows, and input/weight channels in each PE. By doing so, more local psum reuse is exploited within a row of PEs, because they all calculate for a same output channel and psums are accumulated locally. We can derive psum memory as $pMEM_{\text{COEXE}} = C \times \frac{RP_{\text{COEXE}}}{RP_{\text{Eyeriss}}} \times pMEM_{\text{Eyeriss}} = 2 \times \frac{3}{1} \times pMEM_{\text{Eyeriss}} = 6 \times pMEM_{\text{Eyeriss}}$, where RP denotes the number of simultaneously calculated psum rows in each PE. Though psum memory increases, this saves the energy of psum on-chip data movement, which accounts for about 25% energy per operation [14] during AlexNet execution. Moreover, the weight SRAM rather than the psum RF accounts for most of the area in Eyeriss PE. Our design keeps the same weight SRAM size while increasing RF size reasonably to accommodate psums. As the number of contexts further scales, we increase the storage correspondingly based on the two-context design. Table II compares the memory overhead between the two designs.

Data Type (16-bit Precision)	PE (Eyeriss)	PE (CoExe)	
	Size (B)	Size (B) (2-Context)	Size (B) (4-Context)
Input (RF)	24	24	48
Weight (SRAM)	448	448	896
Psum (RF)	48	288	576

TABLE II: PE Memory Overhead Comparison

IV. Evaluation

We evaluate our design using a set of popular NNs, i.e. AlexNet, VGGNet, ResNet and RNN. Our CoExe architecture is configured as 32 4-context PipePEs. Each PipePE has 4 PUs and 8 multipliers. We implement a custom cycle-accurate simulator for simulating SD-MC execution model when different benchmarks are co-running on multiple contexts. For comparison, we implement a dense RS [14] dataflow architecture called Dense-RS-like, with 16×16 PE array to match the same processing throughput. Configurations are listed in Table III. Besides, to evaluate the response time over one of state-of-the-art sparse accelerators SCNN [8], we scale our configuration to process 1024 MACs/cycle. To measure the power and area, we implement the RTL of the SD-MC architecture in Verilog. Then we synthesize the hardware at 0.99V and 667 MHz to obtain gate-level netlist as well as power, area and timing statistics by using Synopsys Design Compiler (DC) under CMOS 40nm process standard cell library with worst case PVT corner. The area and power of SRAM are calculated by CACTI [15].

Architecture Configurations		
Design	**CoExe (4-context)**	**Eyeriss-like**
Technology	40 nm	40 nm
# of PE	4×8	16×16
# of multiplier	256	256
GB-weight (SRAM)	8KB×4; 512-bit/cycle read	8KB; 256-bit/cycle read
GB-activation (SRAM)	100KB×4; 32-bit/cycle read; 16b-bit/cycle write	100KB; 64-bit/cycle read; 64-bit/cycle write
Area (mm^2)	**3.02**	**2.82**
PE Configurations		
iMem (Register)	48B; 128-bit/cycle	24B; 16-bit/cycle
wMem (SRAM)	896B; 128-bit/cycle	448B; 16-bit/cycle
pMem (Register)	576B; 64-bit/cycle	48B; 16-bit/cycle
Area (um^2)	**27994**	**8645**

TABLE III: Architecture Configuration v.s. Eyeriss-like

Design	CoExe (4-context)	SCNN
# of PE	128	64
# of multiplier	1024	1024
SRAM (MB)	1.7	1.0
Technology	40 nm	16 nm
Area (mm^2)	12.1	7.9

TABLE IV: Architecture Configuration Compared with SCNN

A. Response Time Comparison

We simulate three offered load modes, i.e., 10%, 50%, and 100%, to demonstrate that our response time optimization is resulted from the concurrency support. We compare with Eyeriss-like and SCNN in Fig. 9. First, as the number of concurrent tasks growing, the response time of Eyeriss-like baseline degrades by 69%, 72%, and 73% in three service modes, respectively. This is mainly stemmed from its inter-task waiting time explosion because it fails to support concurrency. Meanwhile, SCNN could achieve shorter response time than

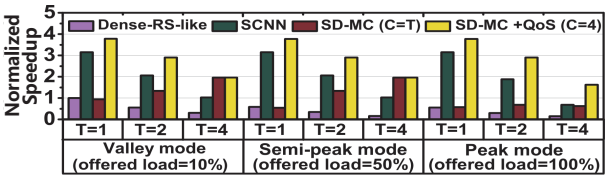


Fig. 9: Response Time Speedup Comparison. T, C: Numbers of Concurrent Tasks and Activated Contexts

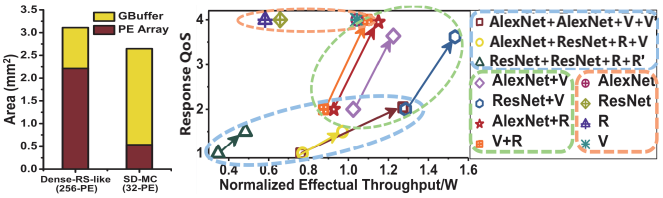
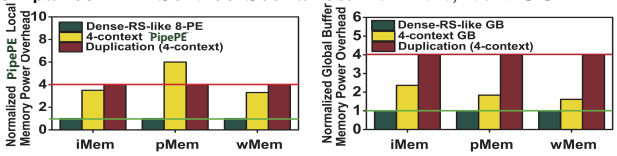


Fig. 10: Area Comparison

Fig. 11: Response QoS-ET/W for Different Service Scenarios. R: RNN; V: VGG



(a) Local Memory Power Overhead: 1-PipePE v.s. 8-Eyeriss-like-PE (b) Global Memory Power Overhead: CoExe v.s. Eyeriss-like

Fig. 12: Power Breakdown for Memories

Eyeriss-like, since it minimizes the execution time. However, its response time still degrades by 67%, 67%, and 78%, respectively. On the other hand, our CoExe architecture could respond services 6.4×, 12.3×, and 11.1× faster than Eyeriss-like, and 1.9×, 1.9×, and 2.4× faster than SCNN. In particular, in peak mode where concurrent workloads request services at a higher frequency, we could handle services faster. The speedup over SCNN is still 1.58× for single task acceleration. This is achieved since we consolidate more resources in the four contexts, while SCNN has non-trivial amount of inter-PE and intra-PE fragmentation.

B. Area Overhead Breakdown

Fig. 10 shows the entire architecture area comparison between CoExe and Eyeriss-like. We have 4× larger global weight and activation SRAM memory than Eyeriss-like. However, the PE array area decreases by 59% with equivalent parallelism. This is determined by the area efficiency and consolidation in each PipePE. Overall, the area overhead over Eyeriss-like is 1.07×.

C. Power Overhead Breakdown

Fig. 12(a) illustrates that local input/weight memory power (i.e., read + write) increases 3.5× and 3.3×, respectively, over an 8 Eyeriss-like PE. Psum memory is over-duplicated up to 6×. However, this saves global-to-local data movements and enables more local data sharing/reuse. Fig. 12(b) shows that global input/weight memory read power increases by 2.36× and 1.6×, respectively, less than that of local memory. This is benefited from our dataflow which has more local data reuse than Eyeriss-like dataflow. Specifically, psum memory power increases only by 1.84×.

D. Flexible Architectural Support for Response QoS and Throughput Efficiency

We show the results of different workload combinations simultaneously executed in CoExe architecture in Fig. 11. In the green circle, 2-task scenarios increase response QoS by 1.89×, and effectual throughput (ET)/W by 1.22×, respectively. The response QoS improvement is derived from leveraging inter-context underutilized resources, while the ET/W improvement comes from the working mode tuning to share more contexts on a single task. In 4-task scenarios, response QoS and ET/W are improved by 1.68× and 1.44×, respectively. Besides, the response QoS and ET/W improvements are also influenced by workload combinations. For example, ResNet+ResNet +RNN+RNN improves only 1.3×, 1.4× QoS and ET/W, respectively, while AlexNet+AlexNet+VGG+VGG improves 2.1× and 1.7×. This is mainly due to the fact that AlexNet and VGGNet have larger ineffectual operation ratios than ResNet and RNN. With the sparsity adaptation method, response QoS and ET/W improvements could be tuned along with real-time requirements in our design.

V. Conclusion

This paper presents a SD-MC architecture for enabling high concurrency, cost efficiency, high utilization for real-time NN services. Our experiment results show that the proposed design achieves up to 12.3× and 2.4× faster responses than Eyeriss-like and SCNN accelerators. Heterogeneous services with different NN models and real-time requirements could be co-executed to consolidate dataflow architecture resources. We expect that the techniques proposed in this paper will contribute to building truly heterogeneous and high efficiency next-generation real-time AI clouds.

References

[1] L. Song, et al., "Hypar: Towards hybrid parallelism for deep learning accelerator array," in *HPCA*, 2019, pp. 56–68.

[2] L. A. Barroso, et al., "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, 2013, vol. 8, no. 3, pp. 1–154.

[3] W. Lu, et al., "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *HPCA*, 2017, pp. 553–564.

[4] M. Song, et al., "Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning," in *HPCA*, 2018, pp. 66–77.

[5] S. Han, et al., "Eie: efficient inference engine on compressed deep neural network," in *ISCA*, 2016, pp. 243–254.

[6] S. Zhang, et al., "Cambricon-x: An accelerator for sparse neural networks," in *MICRO*, 2016, pp. 1–12.

[7] X. Zhou, et al., "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *MICRO*, 2018, pp. 15–28.

[8] A. Parashar, et al., "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017, pp. 27–40.

[9] L. Song, et al., "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA*, 2017, pp. 541–552.

[10] H. Mao, et al., "Lergan: A zero-free, low data movement and pim-based gan architecture," in *MICRO*, 2018, pp. 669–681.

[11] C. Deng, et al., "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *MICRO*, 2018, pp. 189–202.

[12] F. Chen, et al., "Zara: A novel zero-free dataflow accelerator for generative adversarial networks in 3d reram," in *DAC*, 2019, pp. 1–6.

[13] H. Sharma, et al., "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ISCA*, 2018, pp. 764–775.

[14] Y.-H. Chen, et al., "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016, pp. 367–379.

[15] T. Shyamkumar, et al., "Cacti 5.0," 2007.