Zhuang YM, Hu X, Chen XB *et al.* DyPipe: A holistic approach to accelerating dynamic neural networks with dynamic pipelining. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(4): 899-910 July 2023. DOI: 10.1007/s11390-021-1161-y

DyPipe: A Holistic Approach to Accelerating Dynamic Neural Networks with Dynamic Pipelining

Yi-Min Zhuang[1, 2] (庄毅敏), Xing Hu[1] (胡　杏), *Member, CCF*, Xiao-Bing Chen[1, 2] (陈小兵), and Tian Zhi[1, *] (支　天), *Member, CCF*

[1] *State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

E-mail: zhuangyimin@ict.ac.cn; huxing@ict.ac.cn; chenxiaobing@ict.ac.cn; zhitian@ict.ac.cn

Received November 24, 2020; accepted May 30, 2021.

Abstract Dynamic neural network (NN) techniques are increasingly important because they facilitate deep learning techniques with more complex network architectures. However, existing studies, which predominantly optimize the static computational graphs by static scheduling methods, usually focus on optimizing static neural networks in deep neural network (DNN) accelerators. We analyze the execution process of dynamic neural networks and observe that dynamic features introduce challenges for efficient scheduling and pipelining in existing DNN accelerators. We propose DyPipe, a holistic approach to optimizing dynamic neural network inferences in enhanced DNN accelerators. DyPipe achieves significant performance improvements for dynamic neural networks while it introduces negligible overhead for static neural networks. Our evaluation demonstrates that DyPipe achieves 1.7x speedup on dynamic neural networks and maintains more than 96% performance for static neural networks.

Keywords dynamic neural network (NN), deep neural network (DNN) accelerator, dynamic pipelining

1 Introduction

Deep neural networks show their strength in a broad range of applications, such as image processing[1], natural language processing[2], and gaming[3]. Meanwhile, the continuously developing deep neural network (DNN) techniques raise new opportunities for domain-specific architectures[4–6] innovation. Many architectures and system studies[7–9] of machine learning accelerators contribute to accelerating the training and inference of DNNs for better computing capability with power efficiency.

Dynamic neural network (NN) techniques increasingly attract attention from researchers recently due to their powerful representation capability of complex network architectures with dynamic control flow and variable data sizes[10]. In observing their increasing importance in natural language processing[11] and semantic segmentation[12], the broadly-used frameworks start to support dynamic NN techniques. However, for DNN accelerators, a holistic system optimization for the execution efficiency of dynamic NNs is still missing.

Existing studies[13, 14] mainly contribute to optimizing static NNs with fixed input/output shapes and static computational graphs. For example, Auto TVM[15] builds a statistical cost model and designs an exploration module to search for the best configuration to run networks on the hardware. DNNVM[14] designs a cycle-accurate simulator to find the best execution strategy to fuse operations. Such optimiza-

Regular Paper

This work is partially supported by the Beijing Natural Science Foundation under Grant No. JQ18013, the National Natural Science Foundation of China under Grant Nos. 61925208, 61732007, 61732002 and 61906179, the Strategic Priority Research Program of Chinese Academy of Sciences (CAS) under Grant No. XDB32050200, the Youth Innovation Promotion Association CAS, Beijing Academy of Artificial Intelligence (BAAI) and Xplore Prize.

*Corresponding Author

tions require the knowledge of pre-defined fixed network architectures, which are hardly applied to dynamic NNs.

We analyze the inference process of dynamic NNs and observe that the dynamic tensor shapes and control flow hinder the scheduling optimization for better computing parallelism and hardware utilization. For static NNs, the shapes of inputs and outputs are known in advance and the structures of the networks are fixed. Therefore, compilers for DNN accelerators can optimize hardware utilization and computing throughput with software pipelining techniques according to the dependency analysis of static computational graphs[16]. For dynamic NNs, however, the control flow and the computation workload of tensors are determined during runtime. It is challenging to make the best scheduling beforehand during static analysis. Additionally, more contexts need to be recorded for dynamic NNs, which raises the burden of the register resources.

To address these issues, we propose DyPipe, a holistic approach to supporting dynamic NNs in enhanced DNN accelerators. DyPipe enables efficient dynamic pipelining and we also provide a programming interface so that programmers can specify the specialized stages for better computing parallelism. Concretely, our main contributions are as follows.

- We analyze differences between static and dynamic NNs and discover challenges of dynamic scheduling for dynamic NNs.
- We propose an architectural model with a context buffer (CB), which helps efficient pipelining.
- We propose a programming interface, which is feasible for programmers for stage optimization and provides high-level semantics for scheduling optimization.
- We design a scheduler to efficiently perform dynamic pipelining with the enhanced DNN accelerators.

The rest of this paper is organized as follows. In Section 2, we review the background of dynamic NNs and analyze the challenges of them. Section 3 introduces the details of the DyPipe system. We will then present the experimental methodology and experimental results in Section 4. Section 5 describes the inadequacy of existing work in DNN accelerators. Section 6 concludes this paper.

2 Background and Motivation

This section introduces the background of dynamic NNs and analyzes the challenges of running dynamic NNs in existing DNN accelerators, which motivate our design.

2.1 Static and Dynamic Neural Networks

DNN models generally use symbolic representation to represent the structure of the network computational graph. For example, TensorFlow[7] uses nodes and edges to describe the computational graph. Static NNs[17, 18] have fixed network structures and fixed tensor shapes. The definition phase of the computational graph is called static declaration. Static NNs make DNN models easy and efficient to deploy. Compilers can optimize network through complex optimization methods at compile time. Batching techniques can be used to improve the efficiency of multicore processors, such as GPU. Due to such advantages, static declaration is the domain programming paradigm for DNN compilers.

However, along with the continued advance for natural language processing and semantic segmentation, dynamic NNs are applied in more and more modern DNNs. In contrast to static NNs, dynamic NNs are with unfixed compute graphs which contain variable size[19], variable structure[20] and control flow[21]. Dynamic NN techniques, enabling variable network structures by dynamic declaration during runtime, emerge to empower applications that demand complex neural network structures. Specifically, dynamic NNs are commonly applied in the following scenarios.

1) *Sequence Language Models*[19]. The inputs of models are sentences which usually have variable length.

2) *Tree-Structured RNNs*[22]. For language models with sentiment analysis, the inputs are tree-structured and they are variable for different sentences.

3) *Neural Architecture Search* (*NAS*)[23]. NAS aims to find the optimal model for specific tasks by repeatedly testing the performance of different network architectures. The network structures constantly evolve during the execution of tasks.

In some cases, a dynamic NN can be simplified as a static NN[24]. For example, for a sequence language model with variable sentence length, all sentences can be aligned to the longest sentence by adding redundant paddings. But this leads to a large amount of redundant and unnecessary computations.

2.2 Existing Systems

Dynamic NNs are commonly adopted because they support complex tasks that exhibit diverse computational graphs during runtime. Envisioning their importance and ever-increasing demands, many toolkits and frameworks[10, 24–28] were proposed for better programmability and efficiency of dynamic NNs.

TensorFlow Fold[25] provides a high-level combinatory library in TensorFlow to perform dynamic batching. TensorFlow Fold rewrites the computational graphs of the given input into multiple depths and batches the same operation occurring at the same depth together by inserting additional concat and gather operation. MXNet[26] designs operators such as foreach, cond and while_loop to support dynamic control flow. However, these extension features introduce a large graph preprocessing overhead and are not consistent with the original programming models in these frameworks.

There are some imperative frameworks, such as DyNet[10], PyTorch[27]. Unlike the symbolic program which defines computation before the program runs, the imperative program performs computation when the program runs. Imperative frameworks are flexible to construct dynamic NNs since they can execute user expressions instantly. The computation graph construction and execution stages are coupled together in imperative frameworks, which raises the difficulty in adopting compiler optimization such as batching and operator fusion. Therefore, such a programming model is less popular due to this limitation.

2.3 Deep Neural Network Accelerators

DNN accelerators[4–6, 29] are domain-specific processors which are designed to improve computation and energy efficiency of DNN applications. The architectural characteristics of DNN accelerators are quite different from traditional CPU or GPU, which greatly affect the programming models and optimizations of compilers.

DNN accelerators generally have complex memory hierarchies which need to be explicitly managed by software. In addition, accelerators adopt very long instruction word (VLIW) to design their instruction set architectures (ISAs) for vector or matrix computation. Such ISAs need scheduling by software at compile time for better module parallelism. Based on these characteristics, software pipelining[14, 16] is one of the most effective optimization methods on DNN accelerators.

So far, existing toolkits and systems are based on CPU or GPU, without considering the architectural characteristics of DNN accelerators. They are not suitable for DNN accelerators because of the quite different programming models. Therefore, exploring efficient architecture and system optimizations for dynamic NNs in DNN accelerators is a timely topic.

2.4 Motivation

In this subsection, we analyze the architectural behaviors of dynamic NNs and summarize the challenges introduced by their dynamic features.

Dynamic NN toolkits enable dynamic declaration and varied neural networks according to the inputs. Due to these dynamic features, it is challenging for efficient task scheduling and computation pipelining in dynamic NNs. We illustrate the challenges with an inter-operation pipelining example in Fig.1. Here, we omit the intra-operation pipelining.

The static NNs often adopt software pipelining which simultaneously performs computation and data communication for better hardware utilization to improve the computing parallelism, as shown in Fig.1(a). Such software pipelining, implemented by statically scheduling the execution order during compile time, is simple and effective when the neural network model is fixed-structured.

While for dynamic NNs with dynamic control flow, the scheduling order cannot be determined beforehand, and thus the optimal statically scheduling can hardly be achieved. Fig.1(b) shows an example to illustrate the challenge with a dynamic control flow. In this computation graph, the computation of B or C depends on the value of condition check. It is unclear about whether B or C will be executed until the runtime. In this case, the simple software pipelining needs to insert a bubble which hinders the improvement of the computing parallelism.

Additionally, register management for dynamic NNs is much more complex. Contexts, such as tensor shapes, tensor addresses, and loop numbers, can be precomputed as immediate values when networks are static. But for dynamic NNs, they are variable values and all contexts should be saved in the registers. Such requirements make register allocation more challenging, especially for those processors with fewer regis-

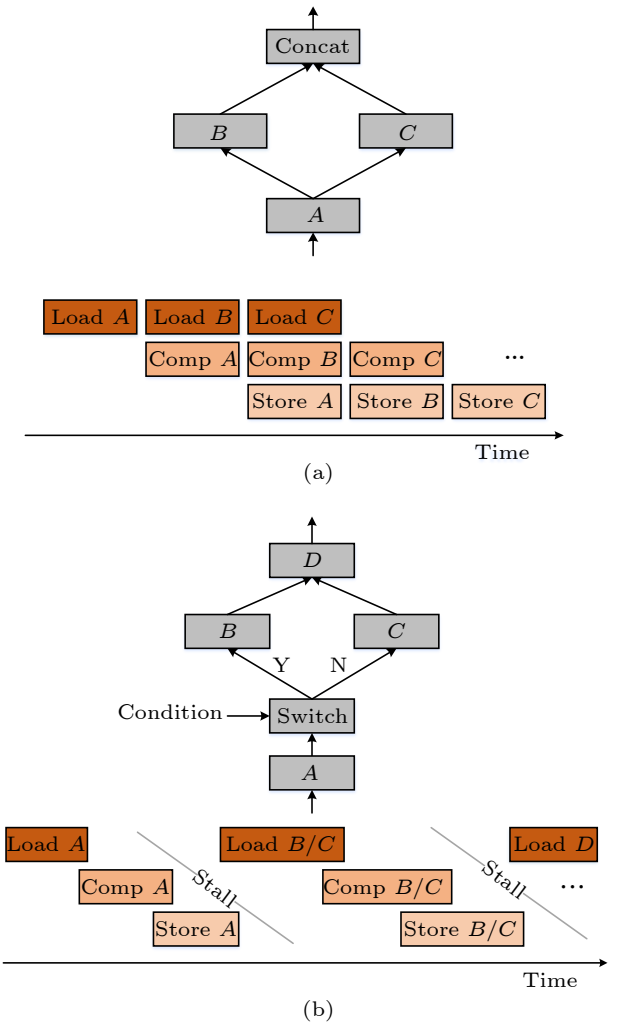


Fig.1. Software pipelining in two kinds of networks. (a) Network with a fixed structure. (b) Network with control flow. Comp means compute.

ters or without hardware memory management units.

Motivated by these two observations, we propose DyPipe, a holistic approach to addressing the inefficacy in dynamic NNs with enhanced hardware accelerator, programming interface, and dynamic scheduling.

3 DyPipe System

3.1 Overview

The DyPipe system proposes a holistic design to eliminate the inefficacy brought by the dynamic features, including an accelerator design with context buffer, a programming interface, and a runtime scheduler. The context buffer (Subsection 3.2) is an on-chip storage. Similar to other on-chip memory in accelerators, it is managed by software. The programming interface (Subsection 3.3) is provided for programmers to define different processes of pipelining. With the assistance of the CB and programming interface, the scheduler (Subsection 3.4) can achieve dynamic pipelining to improve computing parallelism at runtime even when the input graphs are variable with unfixed structures.

The key idea of DyPipe is to decouple dynamic structures and scheduling. By efficiently maintaining the structure of the computational graph on the CB, DyPipe transforms the dynamic graph into a deterministic graph and then the scheduler can achieve efficient pipelining. The overview of the DyPipe system is shown in Fig.2. At compile time when the completely graph is not determined, the operation behavior of each operation is split into several phases. The compiler then defines each phase into separate functions, which we call module functions. At runtime, when the dynamic information such as the structure of graph and tensor shape is determined, module functions of those operations that need to be executed will be saved into the context buffer. For example, we will save module functions of operations A, B and D into the CB if the condition is true. In this way, DyPipe transforms the dynamic graph into a deterministic graph which is saved in the CB. By scheduling the module functions in the CB, the scheduler can achieve high efficient pipelining without being affected by dynamic information. The scheduling process

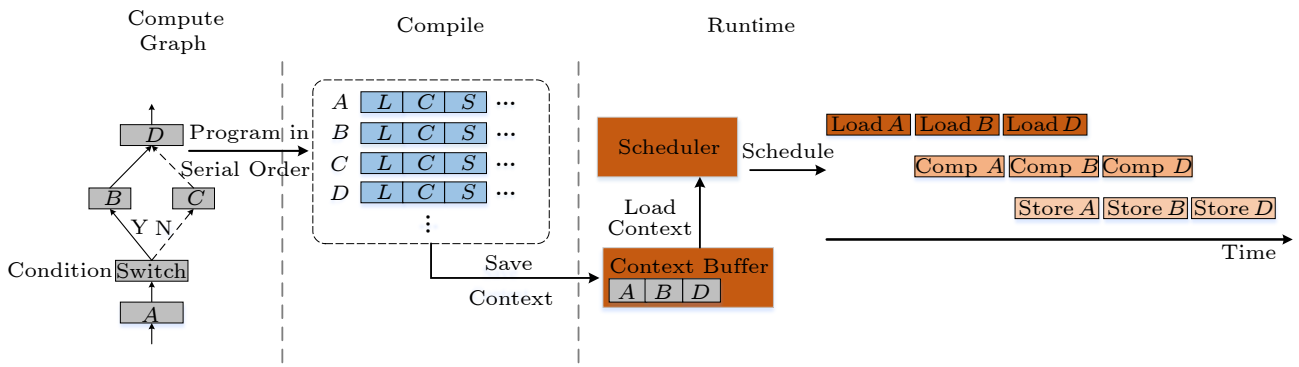


Fig.2. Overview of the DyPipe system. Comp: compute.

will be described in detail in Subsection 3.4.

3.2 DyPipe Context Buffer

DyPipe is established based on the DianNao[29] architecture and the simplified block diagram of DyPipe is shown in Fig.3. The block diagram includes a parallel function unit (PFU), a global buffer and a controller. The PFU is used to perform vector and matrix operations and the global buffer contains neuron buffer and synapse buffer. In addition, DyPipe is tailored specially for dynamic NNs with an additional context buffer, which can transform the original dynamic computational graph at compile time into a deterministic graph during runtime.

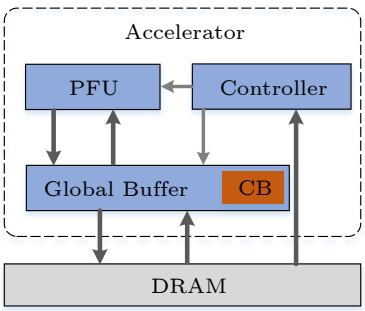


Fig.3. Block diagram of the accelerator with an additional context buffer.

Specifically, the context buffer has two-fold functionalities. 1) The context buffer stores the context information for the execution of operations. The context refers to all the execution information of an operation, including program counters (PCs) of module functions in this operation, tensor shapes, data addresses, computation configurations, etc. 2) The context buffer transforms the dynamic computational graph to a deterministic graph during runtime. To achieve this goal, the context buffer maintains module functions based on their PCs. The module functions describe the execution flow of a computational graph. During runtime, the module functions of the operation to be executed are stored in the context buffer, which construct a deterministic graph. Then, the scheduler manages the scheduling order of module functions according to the PCs in the CB. The details of scheduling are introduced in Subsection 3.4.

For efficient implementation, the context buffer is allocated in the global buffer that provides larger space and lower access latency, compared with allocating it in registers. Thus, the context buffer can reduce the complexity of maintaining contexts. During the optimization and execution of dynamic NNs, the compiler needs to maintain dozens of contexts at runtime. It requires spilling and refilling registers frequently to maintain such a large amount of contexts because of the limited register resources. Register allocation is an NP-complete problem and inappropriate register allocation may block the pipelining. Compared with limited register resources, the context buffer provides a large enough storage for storing these contexts.

The context buffer is also flexible. The compiler is able to customize the recorded parameters and their storage format with the programming interface. The only thing that needs to ensure is the consistency of the processes of loading and storing contexts.

3.3 Programming Interface

Scheduling of pipelining should distinguish which codes can be executed in parallel on different hardware modules, which is extremely time-consuming at runtime for dynamic NNs. To eliminate this effort, DyPipe explicitly specifies the mapping between hardware modules and the code segments by splitting operations into several module functions at compile time. We illustrate it with an example shown in Fig.4. Adjacent codes executed in the same hardware module are combined into a composite function, i.e., module function. For example, the vector load instructions of bias, weight and input make up LoadFunction, ComputeFunction contains instructions of convolution, vector addition and activation, and StoreFunction contains the vector store instruction. Benefiting from the splitting of operations, the scheduler only needs to schedule different module functions,

```
1  for (int idx = 0; idx < LoopSize; idx++) {
2                                          → Load Function
3    if (idx == 0) {
4      LoadV.Onchip.Dram(bias, bias_ddr);
5      LoadV.Onchip.Dram(weight, weight_ddr);
6    }
7    LoadV.Onchip.Dram(in, in_ddr);
8                                          → Compute Function
9    Conv(out, in, weight, bias, kernel...);
10   AddV(out, bias);
11   Active(out, out);
12                                         → Store Function
13   StoreV.Dram.Onchip(out_ddr, out);
14 }
```

Fig.4. Splitting an operation into several module functions.

which largely simplifies scheduling and reduces the runtime scheduling overhead.

To maintain contexts, we need to rewrite the original program to save contexts. Fig.5 shows how LoadFunction saves contexts. When LoadFuncton is ready, all the variables involved are stored into the CB which include source addresses, destination addresses and sizes of each data. Besides, the PC of LoadFunction is also stored in the CB.

```
1  for (int idx = 0; idx < LoopSize; idx++) {
2    // save contexts and PCs of LoadFunction
3    // into the context buffer
4    // save DRAM addresses
5    SaveContext(CB, &in_ddr, &bias_ddr,
6            &weight_ddr);
7    // save on-chip addresses
8    SaveContext(CB, &in, &bias, &weight...);
9    SaveContext(CB, in_size, bias_size,
10           weight_size...);
11   SavePC(CB, &LoadFunction);
12
13   ...
14 }
```

Fig.5. Saving contexts and PCs into the CB.

As for the definition of module functions, DyPipe provides a special programming interface as shown in Fig.6. Compared with normal function definition, we add an additional keyword, i.e., Module Name, in the function definition. Module Name indicates the hardware module corresponding to the current function. The parameters of module functions come from the context buffer and they are prepared in advance. Module functions will first load parameters from the CB to registers and then perform the specific operations.

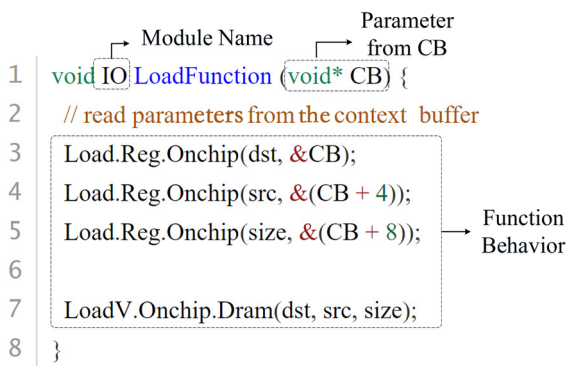


Fig.6. Example of the programming interface.

The proposed programming interface decomposes the program of an operation into two parts. The first part is to maintain contexts and PCs, which results in a flexible function call. The second part is to define different module functions with the indications of module name. Thus, the scheduler can take module functions as basic units to achieve dynamic pipelining with less runtime overhead.

3.4 Runtime Scheduler

We also propose the scheduler to support dynamic pipelining with the assistance of CB. The context buffer saves the module functions of those operations that need to be executed at runtime. The scheduler then schedules these module functions according to the module and data dependency, while retaining the hardware utilization. Such a process is referred to as dynamic pipelining in this study.

The specific process of dynamic pipelining is shown in Fig.7. For a given dynamic graph, the module functions of all relevant operations are defined at compile time according to the programming interface. As shown in Fig.7(a), the module functions of operations A, B, C and D are defined in advance. At runtime, only the module functions of those operations that need to be executed will be saved into the CB. For instance, if the condition of graph is true, the module functions of operations A, B and D are stored into the CB while that of operation C is discarded, as shown in Figs.7(b)–7(d). Therefore, a deterministic graph is constructed in the CB.

Then the runtime scheduler schedules these module functions in the CB according to dependency. The scheduler calls different module functions according to PCs saved in the CB and module functions load relevant contexts from the CB when implemented. Specifically, we explain the intra-operation scheduling and inter-operation scheduling as follows.

Intra-Pperation Scheduling. Take operation A as an example, as shown in Fig.7(b), where L_{ai} means the LoadFunction of operation A for the i-th data block, and C_{ai} means the ComputeFunction of operation A for the i-th data block. L_{a2} and C_{a1} should be implemented after L_{a1} due to module dependency and data dependency respectively, but L_{a2} and C_{a1} can be executed in parallel.

Inter-Operation Scheduling. We further explain the scheduling between operations, as shown in Fig.7(c). When operation B is ready, its module functions are also stored in the CB. Meanwhile, there are remain-

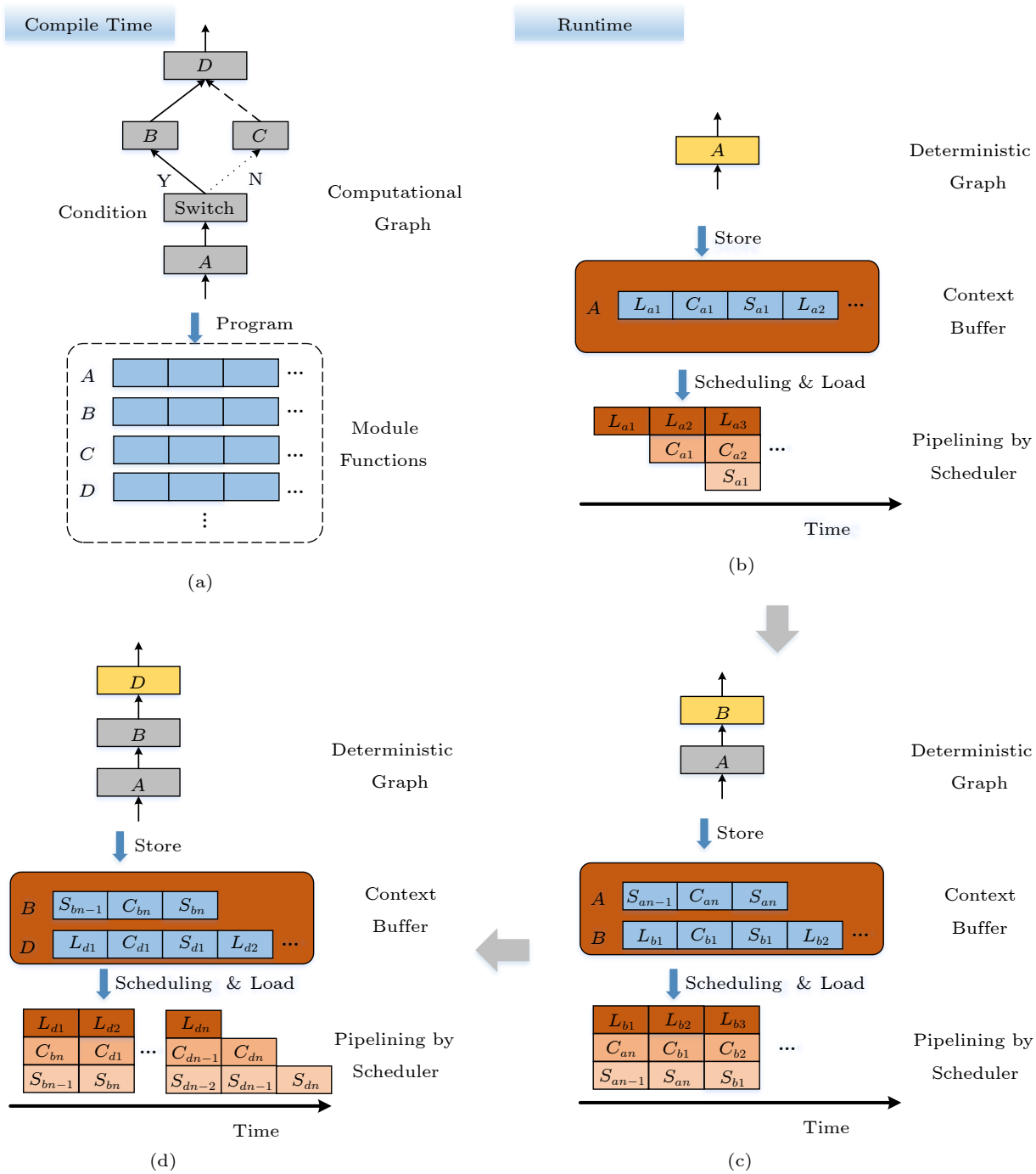


Fig.7. Specific process of the scheduler.

ing module functions of operation A, i.e., $S_{a(n-1)}$, C_{an} and S_{an}. In this situation, the scheduler will call L_{b1} as well as $S_{a(n-1)}$ and C_{an} to execute them in parallel. In this process, we have no need to add redundant bubbles for efficient pipelining.

4 Experimental Results

In this section, we introduce the experimental methodology and experimental results.

4.1 Experimental Methodology

Benchmarks. Three categories of models are used for the performance evaluation: 1) basic operation evaluation, including convolution, pool and fully connected layers; 2) complete static neural network models, including Alexnet[30], ResNet-18[17], GooGleNet[18]

and SqueezeNet[31]; 3) complete dynamic neural network models, generated based on RNN[32], LSTM[33] and GRU[34].

Setup. We implement a cycle-accurate performance simulator based on the DianNao architecture to evaluate the total execution time (cycles), due to the unbearable long duration of silicon implementation. The basic hardware configurations are as shown in Table 1.

Table 1. Configurations of Simulator

Type	Parameter
PE	32 × 64 (2 TOPS @ float16)
Frequency	1 GHz
Memory bandwidth	32 GB/s
Global memory size	1 008 KB
Context buffer	16 KB

Baselines. The experiment compares the execution time of DyPipe and the static pipelining. We also provide the performance without pipelining (NonePipe) to illustrate the performance improvement brought by software pipelining. Further, we analyze the overhead introduced by DyPipe for supporting dynamic pipelining at runtime.

4.2 Operation-Level Performance Evaluation

Initially, we analyze the performance of typical operations that are commonly used in neural networks. Such an experimental evaluation shows the basic overhead of DyPipe compared with static framework when dealing with simple and fixed operations. The operations with different tensor shapes are selected from the state-of-the-art networks[17, 18, 30]. The detailed configurations of them are shown in Table 2.

Table 2. Configurations of Experimental Operations

Layer	C (Output)	Height	Weight	C (Input)	Kernel	Stride
Conv1	256	27	27	96	5	1
Conv2	192	56	56	64	3	1
Conv3	64	56	56	128	1	1
FC1	1 000	-	-	512	-	-
FC2	4 096	-	-	4 096	-	-
FC3	1 000	-	-	1 024	-	-
Pool1	64	112	112	-	3	2
Pool2	512	27	27	-	3	2
Pool3	528	14	14	-	3	1

Note: -: there is no parameter; C: channel.

In Fig.8, we report the execution time of these operations. Compared with NonePipe, both static pipelining and DyPipe have a significant improvement in the accelerator. The results also indicate that DyPipe achieves good performance, which is close to static pipelining by a factor of 0.96x on geometric mean (GeoMean). The performance reduction comes from two major reasons. The first is the additional control flow to support dynamic software pipelining in runtime. The second is the extra memory access to save and reload contexts from the CB.

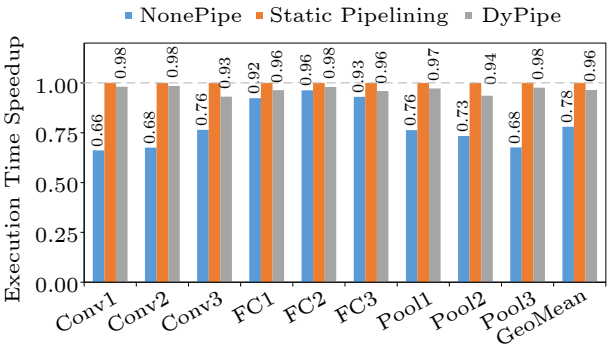


Fig.8. Execution time comparison on operations.

We further take an in-depth analysis of the operation of Conv3, which has nearly 7% performance reduction in DyPipe. We observe that it has a very small channel size and a kernel size. Thus, the execution time for each loop iteration is short and the effect of control flow is more significant. Fortunately, operations with both the small channel size and the kernel size occupy a very small proportion in the whole networks in general.

4.3 Performance Evaluation on Static Neural Networks

In the further step, we make the performance comparison between DyPipe and static pipelining for the state-of-the-art static neural networks, covering ResNet-18[17], AlexNet[30], GoogLeNet[18] and SqueezeNet[31]. The detailed results are shown in Fig.9 across

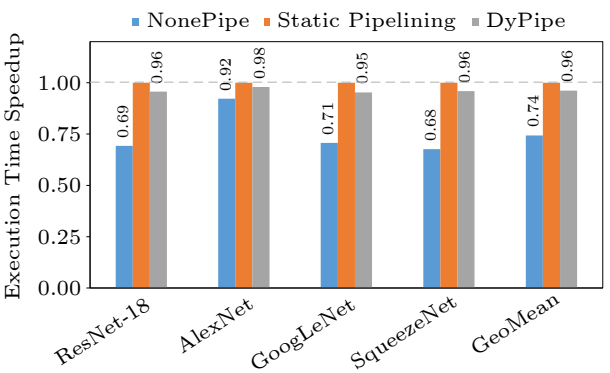


Fig.9. Execution time comparison on the static network.

these four neural networks with different scales. Overall, DyPipe achieves 96% performance on GeoMean compared with static pipelining. However, DyPipe is more flexible since it does not need to recompile and optimize when the network is dynamic and it comes at less than 5% performance penalty. These results show that DyPipe supports normal networks while introducing very low dynamic pipelining overhead.

4.4 Performance Evaluation on Dynamic Neural Networks

We further evaluate the performance advantages of DyPipe for dynamic structures. We produce the dynamic scenarios under the natural language processing domains. Specifically, we generate randomized inputs with variable length, ranging from 1 to 20, for these sequence-to-sequence models. The results are shown in Fig.10. DyPipe outperforms static pipelining by a factor of 1.75x on GeoMean. For static pipelining, short sentences need to be completed by adding paddings so that they can be aligned to the longest sentence, which leads to a large amount of redundant and unnecessary computations. DyPipe eliminates such unnecessary computations by supporting inputs with variable sizes and thus achieves better performance.

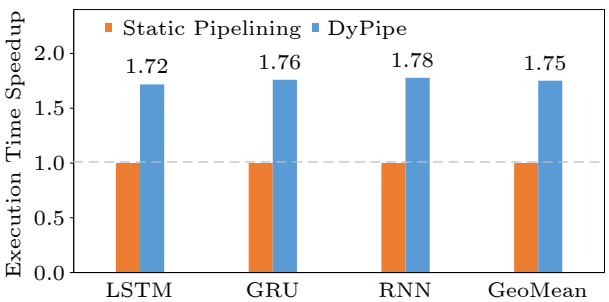


Fig.10. Execution time of dynamic networks for DyPipe compared with static pipelining.

For other dynamic structures such as Tree-structured RNN, the inputs have different structures but the operations involved are similar. Thus, DyPipe can also support such dynamic NNs by defining module functions in advance. However, static pipelining cannot handle dynamic data structures; hence no result is available for comparison.

4.5 Execution Time Breakdown

We break down the execution time into two parts. The first is computation and memory access with DRAM (COM & IO). The second is control flow and overhead of maintaining contexts (CF & CTX) introduced by DyPipe. We choose two static networks and two dynamic networks as the representatives and the results are shown in Fig.11. The results show that the extra effort introduced by DyPipe is small (less than 8%). The main reason is because DNN accelerators can perform hundreds of data in one instruction. Therefore, the extra control flow and memory access from the CB is negligible. Besides, by software pipelining, a part of extra consumption can be covered by computation or memory access through parallelism computing.

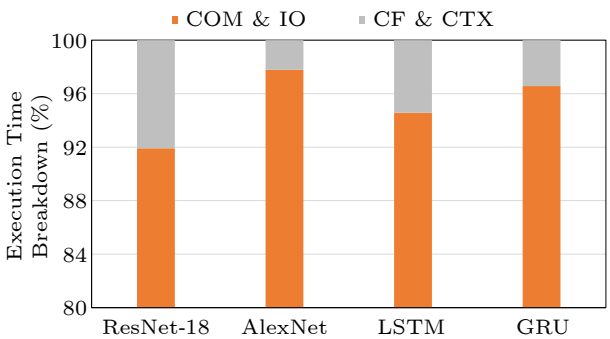


Fig.11. Execution time breakdown.

5 Discussion

Dynamic NNs are applied in many modern deep neural networks and many existing approaches are proposed to support dynamic NNs[10, 24, 25, 28]. However, these approaches are GPU-based optimization, which are not well adapted to DNN accelerators. We take TensorFlow Fold and DyNet as examples to explain the reasons. TensorFlow Fold implements dynamic NNs by rewriting a computational graph into a static control flow graph[25]. However, TensorFlow Fold only applies to the tree-structured networks and cannot support others. Therefore, TensorFlow Fold has limited application scenarios without general support for dynamic NNs. DyNet is an imperative framework to support dynamic NNs[10]. However, the imperative framework cannot support the compile optimizations and is less popular than the declarative framework. To the best of our knowledge, DyPipe is the first study to generally support dynamic neural network models in DNN accelerators. We propose a holistic approach to efficiently supporting dynamic NNs based on a context buffer, a programming interface, and a scheduler. Specifically, DyPipe brings the following benefits. 1) It transforms a dynamic graph

into a deterministic graph, which implements dynamic pipelining. 2) It reduces the overhead of maintaining contexts. 3) It supports flexible and efficient scheduling by calling different module functions based on PCs.

6 Conclusions

This paper presented DyPipe, a holistic approach to optimizing dynamic neural network inferences in enhanced deep neural network accelerators. DyPipe supports dynamic pipelining with a context buffer, a programming interface and a well-designed scheduler. It avoids the substantial overhead of resource management and runtime scheduling. Experimental results showed that DyPipe maintains high performance in static neural networks while efficiently executing dynamic models. It achieves 1.7x speedup on dynamic models.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Xie S N, Girshick R, Dollár P, Tu Z W, He K M. Aggregated residual transformations for deep neural networks. In *Proc. the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp.5987–5995. DOI: 10.1109/cvpr.2017.634.

[2] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A N, Kaiser Ł, Polosukhin I. Attention is all you need. In *Proc. the 31st International Conference on Neural Information Processing Systems*, Dec. 2017, pp.6000–6010.

[3] Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, Hubert T, Baker L, Lai M, Bolton A, Chen Y T, Lillicrap T, Hui F, Sifre L, van den Driessche G, Graepel T, Hassabis D. Mastering the game of go without human knowledge. *Nature*, 2017, 550(7676): 354–359. DOI: 10.1038/nature24270.

[4] Jouppi N P, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, Boyle R, Cantin P L, Chao C, Clark C, Coriell J, Daley M, Dau M, Dean J, Gelb B, Ghaemmaghami T V, Gottipati R, Gulland W, Hagmann R, Ho C R, Hogberg D, Hu J, Hundt R, Hurt D, Ibarz J, Jaffey A, Jaworski A, Kaplan A, Khaitan H, Killebrew D, Koch A, Kumar N, Lacy S, Laudon J, Law J, Le D, Leary C, Liu Z Y, Lucke K, Lundin A, MacKean G, Maggiore A, Mahony M, Miller K, Nagarajan R, Narayanaswami R, Ni R, Nix K, Norrie T, Omernick M, Penukonda N, Phelps A, Ross J, Ross M, Salek A, Samadiani E, Severn C, Sizikov G, Snelham M, Souter J, Steinberg D, Swing A, Tan M, Thorson G, Tian B, Toma H, Tuttle E, Vasudevan V, Walter R, Wang W, Wilcox E, Yoon D H. In-datacenter performance analysis of a tensor processing unit. In *Proc. the 44th Annual International Symposium on Computer Architecture*, Jun. 2017. DOI: 10.1145/3079856.3080246.

[5] Chen Y H, Krishna T, Emer J S, Sze V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 2017, 52(1): 127-138. DOI: 10.1109/JSSC.2016.261657.

[6] Alwani M, Chen H, Ferdman M, Milder P. Fused-layer CNN accelerators. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016. DOI: 10.1109/micro.2016.7783725.

[7] Abadi M, Barham P, Chen J M, Chen Z F, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray D G, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X Q. Tensorflow: A system for large-scale machine learning. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*, Nov. 2016, pp.265–283.

[8] Rotem N, Fix J, Abdulrasool S, Catron G, Deng S, Dzhabarov R, Gibson N, Hegeman J, Lele M, Levenstein R, Montgomery J, Maher B, Nadathur S, Olesen J, Park J, Rakhov A, Smelyanskiy M, Wang M. Glow: Graph lowering compiler techniques for neural networks. arXiv: 1805.00907, 2018. https://arxiv.org/abs/1805.00907, August 2023.

[9] Vasilache N, Zinenko O, Theodoridis T, Goyal P, DeVito Z, Moses W S, Verdoolaege S, Adams A, Cohen A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. arXiv: 1802.04730, 2018. https://arxiv.org/abs/1802.04730, August 2023.

[10] Neubig G, Dyer C, Goldberg Y, Matthews A, Ammar W, Anastasopoulos A, Ballesteros M, Chiang D, Clothiaux D, Cohn T, Duh K, Faruqui M, Gan C, Garrette D, Ji Y F, Kong L P, Kuncoro A, Kumar G, Malaviya C, Michel P, Oda Y, Richardson M, Saphra N, Swayamdipta S, Yin P C. DyNet: The dynamic neural network toolkit. arXiv: 1701.03980, 2017. https://arxiv.org/abs/1701.03980, August 2023.

[11] Devlin J, Chang M W, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proc. the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jun. 2019, pp.4171–4186. DOI: 10.18653/v1/n19-1423.

[12] Kirillov A, Wu Y X, He K M, Girshick R. PointRend: Image segmentation as rendering. In *Proc. the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020, pp.9796–9805. DOI: 10.1109/cvpr42600.2020.00982.

[13] Chen T Q, Moreau T, Jiang Z H, Zheng L M, Yan E, Cowan M, Shen H C, Wang L Y, Hu Y W, Ceze L, Guestrin C, Krishnamurthy A. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proc. the 13th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2018, pp.579–594.

[14] Xing Y, Liang S, Sui L Z, Jia X J, Qiu J T, Liu X, Wang Y S, Shan Y, Wang Y. DNNVM: End-to-end compiler leveraging heterogeneous optimizations on FPGA-based CNN accelerators. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 2020, 39(10): 2668-2681. DOI: 10.1109/tcad.2019.2930577.

[15] Chen T Q, Zheng L M, Yan E, Jiang Z H, Moreau T, Ceze L, Guestrin C, Krishnamurthy A. Learning to optimize tensor programs. In *Proc. the 32nd International Conference on Neural Information Processing Systems*, Dec. 2018, pp.3393–3404.

[16] Xiao Q C, Liang Y, Lu L Q, Yan S E, Tai Y W. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *Proc. the 54th Annual Design Automation Conference*, Jun. 2017, Article No. 62. DOI: 10.1145/3061639.3062244.

[17] He K M, Zhang X Y, Ren S Q, Sun J. Deep residual learning for image recognition. In *Proc. the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp.770–778. DOI: 10.1109/cvpr.2016.90.

[18] Szegedy C, Liu W, Jia Y Q, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In *Proc. the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015. DOI: 10.1109/cvpr.2015.7298594.

[19] Lan Z Z, Chen M D, Goodman S, Gimpel K, Sharma P, Soricut R. ALBERT: A lite BERT for self-supervised learning of language representations. arXiv: 1909.11942, 2019. https://arxiv.org/abs/1909.11942, August 2023.

[20] Zoph B, Le Q V. Neural architecture search with reinforcement learning. arXiv: 1611.01578, 2016. https://arxiv.org/abs/1611.01578, August 2023.

[21] Sutskever I, Vinyals O, Le Q V. Sequence to sequence learning with neural networks. In *Proc. the 27th International Conference on Neural Information Processing Systems*, Dec. 2014, pp.3104–3112.

[22] Tai K S, Socher R, Manning C D. Improved semantic representations from tree-structured long short-term memory networks. In *Proc. the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, Jul. 2015, pp.1556–1566. DOI: 10.3115/v1/p15-1150.

[23] Zoph B, Vasudevan V, Shlens J, Le Q V. Learning transferable architectures for scalable image recognition. In *Proc. the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2018, pp.8697–8710. DOI: 10.1109/cvpr.2018.00907.

[24] Shen H C, Roesch J, Chen Z, Chen W, Wu Y, Li M, Sharma V, Tatlock Z, Wang Y D. Nimble: Efficiently compiling dynamic neural networks for model inference. arXiv: 2006.03031, 2020. https://arxiv.org/abs/2006.03031, August 2023.

[25] Looks M, Herreshoff M, Hutchins D, Norvig P. Deep learning with dynamic computation graphs. arXiv: 1702.02181, 2017. https://arxiv.org/abs/1702.02181, August 2023.

[26] Chen T Q, Li M, Li Y T, Lin M, Wang N Y, Wang M J, Xiao T J, Xu B, Zhang C Y, Zhang Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv: 1512.01274, 2015. https://arxiv.org/abs/1512.01274, August 2023.

[27] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z M, Gimelshein N, Antiga L, Desmaison A, Köpf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J J, Chintala S. PyTorch: An imperative style, high-performance deep learning library. In *Proc. the 33rd International Conference on Neural Information Processing Systems*, Dec. 2019, Article No. 721.

[28] Xu S Z, Zhang H, Neubig G, Dai W, Kim J K, Deng Z J, Ho Q, Yang G W, Xing E P. Cavs: An efficient runtime system for dynamic neural networks. In *Proc. the 2018 USENIX Conference on Usenix Annual Technical Conference*, Jul. 2018, pp.937–950.

[29] Chen T S, Du Z D, Sun N H, Wang J, Wu C Y, Chen Y J, Temam O. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Feb. 2014, pp.269–284. DOI: 10.1145/2541940.2541967.

[30] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. In *Proc. the 25th International Conference on Neural Information Processing Systems*, Dec. 2012, pp.1097–1105. DOI: 10.1145/3065386.

[31] Iandola F N, Han S, Moskewicz M W, Ashraf K, Dally W J, Keutzer K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 MB model size. arXiv: 1602.07360, 2016. https://arxiv.org/abs/1602.07360, August 2023.

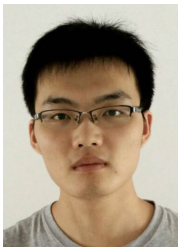
[32] Werbos P J. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 1990, 78(10): 1550-1560. DOI: 10.1109/5.58337.

[33] Hochreiter S, Schmidhuber J. Long short-term memory. *Neural Computation*, 1997, 9(8): 1735-1780. DOI: 10.1162/neco.1997.9.8.1735.

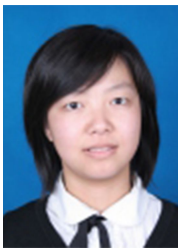
[34] Chung J, Gulcehre C, Cho K H, Bengio Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv: 1412.3555, 2014. https://arxiv.org/abs/1412.3555, August 2023.

Yi-Min Zhuang received his B.S. degree in electronic engineering from University of Science and Technology of China, Hefei, in 2016. He is currently a Ph.D. candidate in University of Chinese Academy of Sciences, Beijing. His research interests include deep learning and compiler of neural network accelerator.

Xing Hu received her B.S. degree in computer science and technology from Huazhong University of Science and Technology, Wuhan, and her Ph.D. degree in computer architecture from University of Chinese Academy of Sciences, Beijing, in 2009 and 2014, respectively. She is currently an associate professor of State Key Laboratory of Processors, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. Her current research interests include domain-specific hardware architectures.



Xiao-Bing Chen received his B.S. degree in information security from Wuhan University (WHU), Wuhan, in 2016. He is currently a Ph.D. candidate in University of Chinese Academy of Sciences, Beijing. His research interests include deep learning and compilation optimization.



Tian Zhi received her B.E. degree in biomedical engineering from Zhejiang University, Hangzhou, in 2009, and her Ph.D. degree in information engineering from Institute of Electronics, Chinese Academy of Sciences, Beijing, in 2014. She is currently an associate professor at the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. Her research interests include computer architecture and computational intelligence.