SynergyFlow: An Elastic Accelerator Architecture Supporting Batch Processing of Large-Scale Deep Neural Networks

JIAJUN LI, GUIHAI YAN, WENYAN LU, SHIJUN GONG, SHUHAO JIANG, JINGYA WU, and XIAOWEI LI, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, People's Republic of China

Neural networks (NNs) have achieved great success in a broad range of applications. As NN-based methods are often both computation and memory intensive, accelerator solutions have been proved to be highly promising in terms of both performance and energy efficiency. Although prior solutions can deliver high computational throughput for convolutional layers, they could incur severe performance degradation when accommodating the entire network model, because there exist very diverse computing and memory bandwidth requirements between convolutional layers and fully connected layers and, furthermore, among different NN models. To overcome this problem, we proposed an elastic accelerator architecture, called SynergyFlow, which intrinsically supports layer-level and model-level parallelism for large-scale deep neural networks. SynergyFlow boosts the resource utilization by exploiting the complementary effect of resource demanding in different layers and different NN models. SynergyFlow can dynamically reconfigure itself according to the workload characteristics, maintaining a high performance and high resource utilization among various models. As a case study, we implement SynergyFlow on a P395-AB FPGA board. Under 100MHz working frequency, our implementation improves the performance by 33.8% on average (up to 67.2% on AlexNet) compared to comparable provisioned previous architectures.

CCS Concepts: • **Computer systems organization → Neural networks**; **Data flow architectures**;

Additional Key Words and Phrases: Deep neural networks, convolutional neural networks, accelerator, architecture, resource utilization, complementary effect

ACM Reference format:
Jiajun Li, Guihai Yan, Wenyan Lu, Shijun Gong, Shuhao Jiang, Jingya Wu, and Xiaowei Li. 2018. SynergyFlow: An Elastic Accelerator Architecture Supporting Batch Processing of Large-Scale Deep Neural Networks. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 8 (December 2018), 27 pages.
https://doi.org/10.1145/3275243

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61572470, 61872336, 61532017, 61432017, 61521092, 61376043, and in part by Youth Innovation Promotion Association, CAS under grant No.Y404441000. The corresponding authors are Guihai Yan and Xiaowei Li.
Authors' addresses: J. Li, G. Yan, W. Lu, S. Gong, S. Jiang, J. Wu, and X. Li, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, No. 6 Kexueyuan South Road, Haidian, Beijing, 100190, China; email: {lijiajun, yan, luwenyan, gongshijun, jiangshuhao, wujingya, lxw}@ict.ac.cn.
Author's current address: The authors are also with University of Chinese Academy of Sciences.

1084-4309/2018/12-ART8 $15.00
https://doi.org/10.1145/3275243

NOMENCLATURE

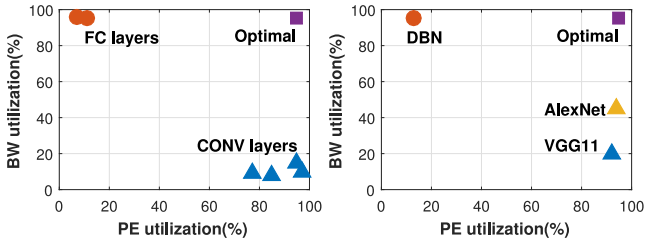
ATA	Asymmetric Tandem Architecture
BP	Back Propagation
BW	Memory Bandwidth
CCIB	C-Engine Computing Instruction Block
CMIB	C-Engine Memory Instruction Block
CNN	Convolutional Neural Network
CONV	convolutional layers
CTC ratio	Computation to Communication ratio
DNN	Deep Convolutional Neural Network
FC	fully-connected layers
FCIB	F-Engine Computing Instruction Block
FF	Feed Forward
FMIB	F-Engine Memory Instruction Block
GD	Gradient Descent
NN	Neural Network
STA	Symmetric Tandem Architecture

1 INTRODUCTION

Neural networks have been widely used in various fields such as image classification (Dahl et al. 2013; Farabet et al. 2009; Huang et al. 2013; Ji et al. 2013; Larochelle et al. 2007; Mnih and Hinton 2012). CNNs (Lecun et al. 1998) and DNNs (Bengio et al. 2013; Huang et al. 2014) have achieved state-of-the-art performance across a broad range of applications. With the networks going larger and deeper, there is a request for an unprecedented computation capacity, which poses a great challenge to the computing architectures. Thus, accelerator solutions have been shown to be highly promising in terms of both performance and energy efficiency (Chakradhar et al. 2010; Chen et al. 2014a, 2014b; Du et al. 2015; Esmaeilzadeh et al. 2012; Fan et al. 2009; Farabet et al. 2009; Hameed et al. 2010; Li et al. 2018a, 2018b; Liu et al. 2015; Qiu et al. 2016; Zhang et al. 2015).

Most of the prior accelerator solutions mainly focused on convolutional layers probably because of their sophistication and high computation intensity. However, prior accelerator-oriented solutions offer little insight into orchestrating batches of concurrent NN models. After all, executing one convolutional layer efficiently cannot necessarily guarantee the efficient execution of the whole NN model and, furthermore, a batch of models. The NN batches can be formed either when training one large-scale NN with batch learning algorithms (Vanhoucke et al. 2011) or multiple independent NN applications sharing the same hardware substrate (Silver et al. 2016). Such parallel processing of various NN models is a typical case of "Model-Level Parallelism." Google's practice (Dean et al. 2012) has shown that taking advantage of model-level parallelism can greatly boost the performance of training a large-scale NN using a cluster of computers. Rather than focusing on layer-wise optimization, our objective is to design a new accelerator that intrinsically supports such model-level parallelism in a "synergetic" way (i.e., rather than simply packing more accelerators into a "mega-accelerator").

There are two types of resources that attract the top consideration when designing an accelerator: (1) processing elements (PEs), which determine the nominal performance, usually measured in GOPS (Giga-Operations Per Second), and (2) memory bandwidth (BW), which largely determines the utilization of PEs, and ultimately the performance as well. However, striking an optimal balance between PEs and BW is never easy given the sophisticated diversity of NN applications.



(a) Layer-level diversity (b) Model-level diversity

Fig. 1. Utilization maps of generic accelerators.

The diversities of neural network models mainly come from (1) layer-level diversity and (2) model-level diversity. A deep neural network is usually composed of multiple layers, in which CONV and FC dominate the amount of computations. CONVs are computation intensive, while FCs are memory intensive, hence exhibiting distinct layer-to-layer diversity. The network models also differ in network depth, type, and scale of each layer. Accordingly, the oracle hardware resource allocation also spatially varies for different models and temporally changes even within the same model. Collectively, some models are relatively more computation intensive, while others may be relatively more memory intensive.

Properly handling these diversities is the key to designing an efficient accelerator. Since CONV layers are typically hungry for computing resources, while FC layers are hungry for memory bandwidth, the PEs will be quickly exhausted if too many CONV tasks are mapped to the accelerator simultaneously. At the same time, the BW may be underutilized due to the memory-nonintensive characteristic of CONV, causing poor BW efficiency. The efficiency can be clearly illustrated with a utilization map as shown in Figure 1(a). By contrast, if the BW is monopolized by FC tasks, the PEs will be rendered low utilized given the computation-nonintensive FC tasks. In such a scenario, the pending tasks, excluded by the jammed BW, have no way to access PEs even they are available.

Model-level diversity also causes a similar problem. We use AlexNet (Krizhevsky et al. 2012), VGG (Simonyan and Zisserman 2014), and DBN (Hinton et al. 2006) as examples. The result is shown in Figure 1(b). AlexNet and VGG make relatively good use of the PEs because they are more computation intensive, but the BW resources are relatively overprovisioning, especially VGG. By contrast, the more memory-intensive DBN shows the opposite.

An ideal solution should make high utilization of both types of resources, as located in the "sweet spot," the upper right corner in the utilization maps. Rather than blaming them for low efficiency, we would like to "dance with the diversity." Toward that sweet spot, the diversity can bring a significant complementary effect to boost the hardware utilization. For example, both the BW and PE utilization will be boosted by scheduling CONV tasks and FC tasks to work concurrently but in different PE regions. The underlying rationale is that the resource requirements of CONV and FC are complementary to each other. We find that properly leveraging model-level parallelism can unlock such a complementary effect.

However, there are two challenges: (1) how to determine the capacity of resource pools for various tasks and (2) how to orchestrate the diversified tasks to maximize the complementary effect. In this article, we propose a new accelerator architecture, called SynergyFlow, to tackle the above challenges. The key contributions are summarized as follows:

- We present an in-depth analysis of the characteristics of typical NN accelerator architectures and analyze the complementary effect between computation resource and memory bandwidth when running various models.

- We propose a tandem-engine architecture, SynergyFlow, which can smoothly orchestrate the CONV and FC computation. The resources allocated to each engine can be dynamically adjusted according to the workloads. By doing so, both the computation resource utilization and memory bandwidth utilization can be boosted.
- We present an analytical model for the resource pool configurations. This model is of low complexity and therefore supports dynamic resource allocation.

As a case study, we implement a SynergyFlow design on the P395-AB board, showing that SynergyFlow outperforms previous approaches in terms of both performance (up to 67.2% improvement) and energy consumption at almost the same hardware resources usage.

The rest of this article is organized as follows: Section 2 provides background and motivation. Section 3 presents the evolution of SynergyFlow, followed by design details presented in Section 4. Section 5 describes the implementation. Section 6 shows experiments, followed by related works discussed in Section 7. Section 8 concludes this article.

2 BACKGROUND AND MOTIVATION

2.1 Primer on Neural Networks

Layers in CNNs and DNNs. Deep learning includes both DNNs (Bengio et al. 2013; Huang et al. 2014) and CNNs. Usually, a CNN or a DNN contains four types of layers, i.e., CONV, Pooling, FC, and Normalization layers, where CONV and FC dominate the amount of computations. In terms of computing patterns, both CONV and FC layers are dominated by mainly matrix-vector-product (MVP) and vector-matrix-product (VMP). Though MVP and VMP can be considered as one computing pattern, we treat them differently for the accelerator design to eliminate energy-consuming matrix transposition (Gu et al. 2014).

Neural Network Training. The process of training is the critical operation for neural network applications (Li et al. 2014). Generally, training a neural network contains three steps: FF, error back-propagation (BP), and weight update. FF and BP can be considered as symmetric phases as they share almost the same computing patterns. The GD is one of the mostly used algorithms in weight updating, which is realized by adjusting the weights layer by layer (Gemulla et al. 2011).

2.2 Limitation of State-of-the-Art CNN Accelerators

The computational intensity of the convolutional layers has sparked significant research interest in the design and optimization of accelerator structures (Chen et al. 2014b; Qiu et al. 2016; Suda et al. 2016; Zhang et al. 2015). Although prior solutions can deliver high computational throughput for convolutional layers, they could incur severe performance degradation when accommodating the entire network models, because convolutional layers and fully connected layers exhibit very diverse computing and memory bandwidth requirements. We observed that these architectures inevitably incur resource underutilization of either computing resource or memory bandwidth, causing performance degradation when processing the entire network model. As confirmed by the experimental results of work (Chen et al. 2014b; Qiu et al. 2016; Suda et al. 2016) demonstrated in Figure 2, the performance of Suda et al. (2016) on AlexNet merely reaches no more than 53.0% compared to their performance on the convolutional layers. Therefore, the above observation motivates building a highly efficient architecture to handle the diversities in CNNs and DNNs.

3 EVOLUTION OF SYNERGYFLOW

In this section, we present the overview of the SynergyFlow architecture by explaining how it evolves from the conventional NN accelerators to a more “thrifty” counterpart.

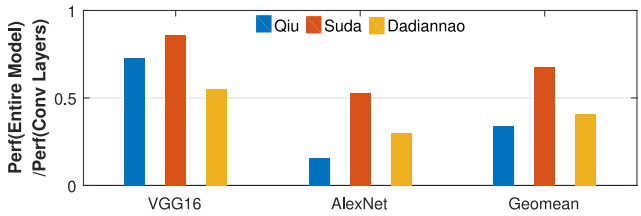


Fig. 2. The attainable performance on the entire NN model compared to the performance on CONV layers for generic accelerators.

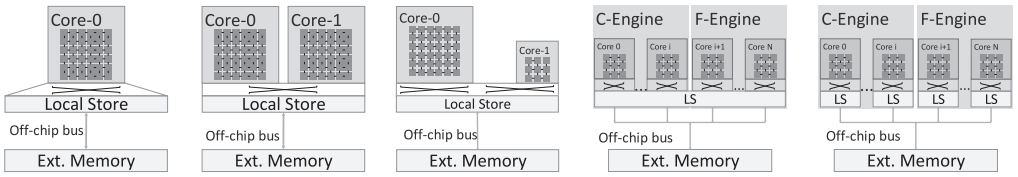


Fig. 3. NN accelerator architectures: (a) monolithic architecture; (b) symmetric tandem architecture; (c) asymmetric tandem architecture; (d) elastic architecture; (e) optimized elastic architecture.

3.1 Generic Architecture

A generic architecture of state-of-the-art accelerators (Chen et al. 2014a, 2014b; Zhang et al. 2015) can be modeled as in Figure 3(a). It comprises a core, a local store (LS), and an external memory (MEM). The core serves as the logic for arithmetic computations. It usually consists of many PEs organized into mesh-like topology. LS plays the role of cache in a traditional CPU but is commonly software managed to fully take advantage of application-specific features in NN applications, such as double-buffering techniques (Zhang et al. 2015). The on-chip interconnect is dedicated for data communication between the LS and PEs. The MEM is shared with other accelerators or host systems with high bandwidth meeting industry DDR standards. The accelerator is attached to a general-purpose host system that automatically translates network specifications (numbers of layers, kernel size, etc.) into code segments. The code segments can be mapped, scheduled, and executed on the accelerator. By assigning the code segments of different NN models to the accelerator, it can provide acceleration for these models, including inference and training phases.

First, we present a quantitative analysis for a general NN model in terms of computing and memory access. CONV layers and FC layers exhibit very diverse computing and memory bandwidth requirements. This layer-level diversity can be clearly identified by examining the metric *arithmetic intensity* (Williams et al. 2008; Williams et al. 2009), or *Computation-to-Communication (CTC)* ratio (Zhang et al. 2015), which is defined as the computation operations per memory access:

$$CTC = \frac{N_{op}}{N_d}, \tag{1}$$

where N_{op} denotes the total number of operations and N_d denotes the amount of external data accesses in the same time span. Intuitively, greater CTC implies more intensive computation or less memory bandwidth requirement; by contrast, the smaller CTC is correlated with less computation but higher memory intensity. The former case is usually interpreted as computation-intensive patterns, while the latter memory-intensive patterns. We find that the CTC ratio of FC layers is greatly dwarfed by that of CONV layers, as shown by the example in Figure 4. This result reveals

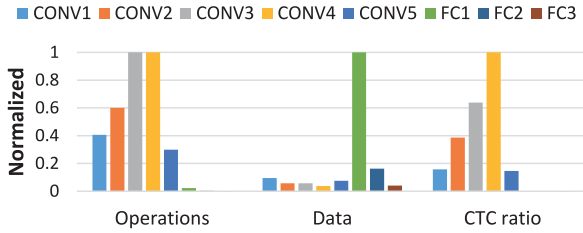
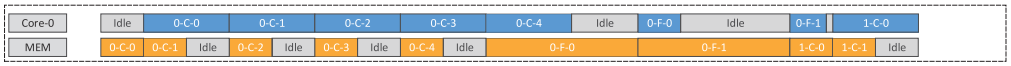
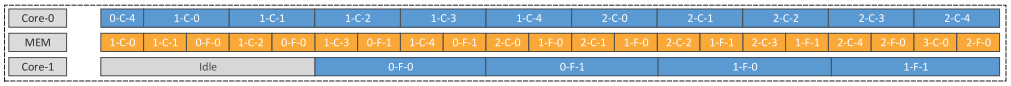


Fig. 4. Complexity analysis on VGG11 (the largest value of each item is set as the baseline).



(a) Monolithic architecture

(b) Symmetric tandem architecture



(c) Asymmetric tandem architecture and SynergyFlow

Fig. 5. Timing graph of NN accelerator architectures.

that CONV layers can do several orders of magnitudes more computing operations on each memory access than FC layers. The architectural implications can be reached from two perspectives:

1) *Memory-centric perspective*: If CONV and FC layers are fed with the same memory bandwidth, then to saturate the memory bandwidth, the computing capacity (e.g., the number of PEs in the computing engine) will be determined by CONV because its CTC ratio is much larger than that of FC. In such case, efficiency loss is inevitable because the memory-intensive FC does not have enough computations to saturate the computing capacity designed for the computing-intensive CONV.

2) *Computing-centric perspective*: The CONV layers and FC layers are served with the same computing capacity; then, to make full use of the capacity, the memory bandwidth will be determined by memory-intensive FC layers indicated by the much lower CTC ratio. Under the same computation volume, FC layers need much larger data volume than CONV layers. In such case, efficiency loss is also inevitable since CONV layers do not have enough memory access to saturate the memory bandwidth designed for FC layers.

In the following discussion, we will use the computing-centric perspective to uncover the limitations of conventional designs.

The efficiency loss cannot be avoided under the conventional architectures largely due to the intrinsically monolithic nature; i.e., the core for CONV and FC is the same one and cannot be spatially separated. The principle of "one fit to all" is challenged in this scenario. Figure 5(a) further demonstrates the limitation of monolithic architecture where a single core processes CONV and FC layers sequentially. The timing diagram consists of the states of two components, core and MEM, indicating the utilization of computation resource and memory bandwidth, respectively. The "idle" state indicates that the core or MEM is idle. The busy state of the core or MEM can be

specified by a triple (x-y-z), where x indicates the NN instance ID; y denotes the type of layer, "C" for CONV layer and "F" for FC layer; and z denotes the ID of layers. For example, (1-C-2) specifies the busy state of the second convolutional layer of the first NN instance. The MEM block is busy in either loading the data from external memory to the LS or writing the data in the LS back to external memory.

In monolithic architectures, to avoid a performance bottleneck, the core will be designed to meet the demand of computation-intensive CONV layers, while the memory bandwidth meets the demand of memory-intensive FC layers. When executing computing-intensive CONV layers, the memory bandwidth tends to be overprovisioning, causing efficiency loss. When executing memory-intensive FC layers, the computing capacity tends to be overprovisioning, again causing efficiency loss. In other words, monolithic architectures cannot efficiently deal with CONV layers and FC layers simultaneously, suffering underutilization of either computation resource or memory bandwidth.

3.2 Tandem Architectures

To overcome the limitations of monolithic architectures, we propose to spatially separate the hardware substrate into two parts, dedicated for CONV and FC layers, respectively. The basic principle can be explained with a Symmetric Tandem Architecture (STA), as shown in Figure 3(b). This architecture adopts two homogeneous cores, one for CONV layers (Core-0), another for FC layers (Core-1), and both share the same local store. This architecture exploits the complementary effect of CONV layers and FC layers: the surplus memory bandwidth of Core-0 can be reaped by Core-1, as long as the two cores can be independently managed. By properly scheduling the two cores, we can maintain a working principle of two-stage pipeline (CONV stage and FC stage), as illustrated in Figure 5(b). The underutilized memory bandwidth in processing CONV layers can be reaped by FC layers, boosting the utilization of memory bandwidth. Compared with monolithic architecture, the idle periods of "MEM" in CONV layers are gathered for FC layers.

However, this architecture can be improved further because Core-1 still suffers severe underutilization. Given that the total amount of operations in FC layers is usually magnitudes less than that in CONV layers, the computing capacity of Core-1 should be scaled down to fit the FC computing requirement. This scaling can be easily accomplished by leveraging the "big-little" architecture philosophy. We therefore reduce the capacity of Core-1 and get an Asymmetric Tandem Architecture (ATA), as shown in Figure 3(c). The idle periods of computation resource and memory bandwidth can be greatly diminished. As exemplified in Figure 5(c), the core and MEM idle periods are almost eliminated, except the beginning warmup phase.

3.3 SynergyFlow Architecture

The "big-little" architecture is still not an optimal design because of lack of adaptivity when processing various models. Because of model-level diversity, some models may contain relatively more FC computations but fewer CONV computations compared with other models, as the comparison between AlexNet and VGG11 shows in Figure 6(a). Intuitively, AlexNet needs a more powerful Core-1, while VGG11 prefers a more powerful Core-0. Thus, the unadaptable architecture for AlexNet will suffer underutilization of Core-1 when processing VGG11 and vice versa. Since the two cores are hardwired in hardware substrate and have static computing capacities, tandem architectures cannot maximize the utilization for various models.

To tackle this problem, we therefore virtualize the asymmetric tandem architecture on a homogeneous core array substrate, as shown in Figure 3(d). By doing so, the computing capacity of Core-0 and Core-1 is not necessarily constant and hardwired, but only a logical organization. These cores are loosely coupled and can be dynamically allocated to the C-Engine (virtualized Core-0)

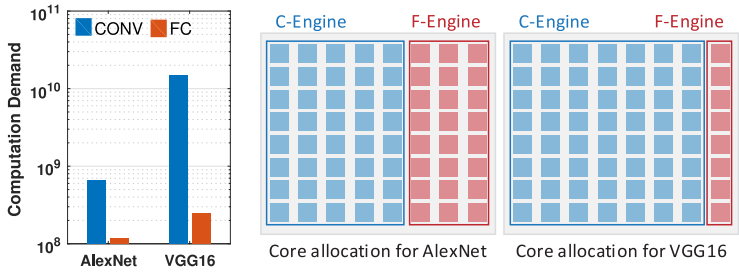


Fig. 6. Dynamic core allocation for C-Engine and F-Engine: (a) computation demand and (b) core allocation.

or F-Engine (virtualized Core-1) to maximally use the cores and memory bandwidth resources. Depending on the actual workloads, the scale of the two engines can be dynamically determined to match the resource requirements of workloads. We call this elastic architecture SynergyFlow.

SynergyFlow maintains the advantages of asymmetric tandem architecture: SynergyFlow orchestrates the CONV and FC computations by scheduling them to be executed in the same time slots, and thus the resource utilization of both computing resource and memory bandwidth can be boosted, thereby improving the performance. Furthermore, SynergyFlow can adaptively adjust the computing capacity of the C-Engine and F-Engine to better accommodate various NN models.

Figure 6(b) presents an example of dynamic core partitioning on an 8×8 homogeneous core array substrate. Since AlexNet needs a relatively more powerful F-Engine, more cores are allocated to the F-Engine compared with VGG11. The detailed allocation algorithm will be discussed in Section 4.2.

Shared or Private Local Store? Unlike the conventional architectures, we found that there exist little data dependencies between CONV layers and FC layers (only in the final CONV layer, the feature maps are flattened into a feature vector, serving as the input of the FC layers); thus, the shared local store between the C-Engine and F-Engine can be split into private local stores for each engine.

Inside the F-Engine, the cores share the feature vectors, which takes only a small share of the total amount of data; thus, further splitting the local store of the F-Engine into private local stores for each core hardly harms the data reusability. As the cores in the C-Engine share the feature maps, a shared local store for the cores in the C-Engine can indeed maximize the data reusability. However, it drastically increases the complexity of interconnects. It will also cause severe data access conflicts among the cores and cause a need for a complex LS management unit. Therefore, there exists a tradeoff between data reusability and design complexity. According to our experiments, private local stores increase the memory traffic between on-chip LS and off-chip memory by about 6% compared with a shared counterpart. Considering this affects the whole system little, we choose a private LS for the cores in the C-Engine for reducing design complexity, as shown in Figure 3(e), which in turn increases the flexibility of SynergyFlow. Furthermore, since the C-Engine and F-Engine are logically virtualized from a homogeneous core array substrate, we can use the same interconnect fabric for the C-Engine and F-Engine to transfer data between PEs and local stores.

To sum up, SynergyFlow has the following advantages over monolithic architectures: for layer-level diversity, SynergyFlow well orchestrates the computation of CONV layers and FC layers, and thus can almost fully utilize the computing resource and memory bandwidth; for model-level diversity, SynergyFlow can be adaptive to better accommodate various models.

4 SYNERGYFLOW ARCHITECTURE

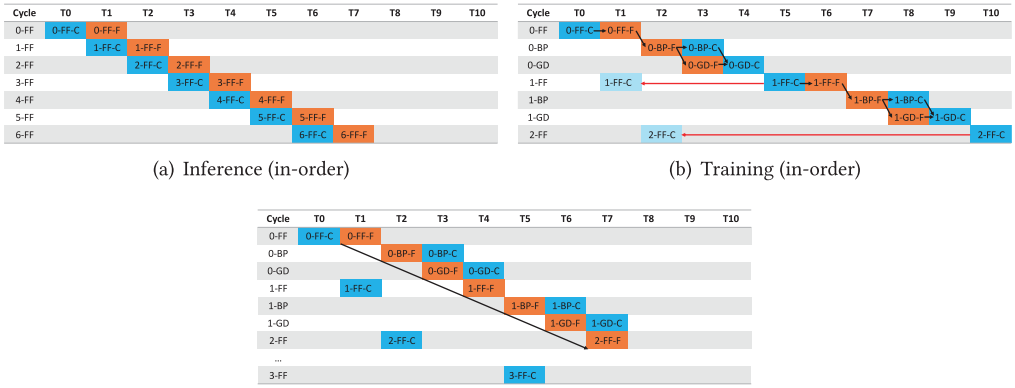
In this section, we will introduce the detailed designs of the SynergyFlow architecture. To support the parallel execution of the C-Engine and F-Engine, we first propose two key working principles, Out-of-Order Layer Scheduling and Preemptive Memory Bandwidth Allocation, and then present a Dynamic Core Allocation Algorithm guiding the resource pool configuration.

4.1 Working Principles

To better explain the working principles, we first present a formalization model to describe NN applications. Note that we primarily target the NN models containing both CONV and FC layers, since most NN applications contain both types of layers. Each phase (FF, BP, GD) can be considered as a type of *macro instructions*. The execution of each macro instruction usually contains two stages, the CONV stage and FC stage. We can encode any NN applications with these macro instructions. Hence, the processing of the layers can be described with the macro instruction scheduling in SynergyFlow.

In the FF procedure, the macro instruction uses the CONV stage first, followed by the FC stage. Accordingly, the C-Engine is invoked first to process convolutional computation, and then the F-Engine takes over to process the FC computation. In the BP procedure, due to the reversed data dependency compared with FF, the macro instruction uses the FC stage first, then the CONV stage. Thus, the F-Engine is enabled first, then the C-Engine. By contrast, since no explicit data dependency exists in the GD procedure, either engine can be enabled first in this procedure, depending on the instant resource availability. Interestingly, this flexibility of GD can help to balance the two-stage pipeline. If the C-Engine is overloaded while the F-Engine is relatively idle, then GD can use the FC stage first, or vice versa. To simplify discussion, we assume the maximal time period of processing a CONV task or FC task as one "logical" *cycle*, or simply *cycle* unless otherwise specified, to differentiate from the actual clock cycle of cores.

4.1.1 Out-of-Order Layer Scheduling. Because of the execution order of CONV layers and FC layers, the C-Engine and F-Engine often cannot work concurrently. For example, the F-Engine takes the output of the C-Engine as input, and thus the F-Engine cannot start to work until the C-Engine finishes the task, which violates the original intention of SynergyFlow. Thus, we apply an out-of-order (OoO) layer scheduling mechanism to keep both engines working concurrently. The concept is similar to the OoO execution model in classical superscalar processors (Dwyer and Torng 1992). Figure 7 illustrates the layer scheduling mechanism. In the "pipeline," each stage is denoted as (x-y-z), where x indicates the instance id, y indicates the phase type (FF, BP, GD), and z indicates the layer type ("C" for CONV, "F" for FC). We assume that the provided memory bandwidth is large enough to satisfy the computation, and thus the MEM stage is not shown in the figure. In the inference phase, the scheduling exploits the batch processing of instances to keep both stages busy; i.e., the C-Engine processes the CONV layers of the later instance, while the F-engine processes the FC layers of the current one. The data dependencies of the training phase are indicated by the arrow lines in Figure 7(b). If the tasks are executed in order, it will cause idles of either the C-Engine or F-Engine. The idles can be reduced or even eliminated by out-of-order layer scheduling; see Figure 7(c). Note that the term "out-of-order" implies that the CONV computations of later instances can be executed before the FC computations of the current instances. Correspondingly, "in order" implies that only after the computations of the current instance is totally completed can the computations regarding the next instance be processed. The implementation of out-of-order layer scheduling in SynergyFlow is much simpler than that in conventional superscalar processors. Because the data dependency is simple and explicit and can be obtained in advance (Section 5.1), it is unnecessary to implement a dedicated hardware such as scoreboards



(a) Inference (in-order)

(b) Training (in-order)

(c) Training (out-of-order)

Fig. 7. Out-of-order layer scheduling for SynergyFlow.

or reorder buffers. As shown in Figure 7(c), the execution of the F-Engine is in order, and we need only pre-execute some *FF instructions* of the training instances, among which there are no data dependencies. Through the layer scheduling, we can keep both stages busy without contention.

Note that the out-of-order processing may result in increased latency. However, since we target batch processing of NN models, the throughput of NN instances should be the preferred metric, rather than the latency of single instances. The rationale is very similar to that case in superscalar processors with out-of-order instruction execution. Hence, we take throughput as the primary optimization goal rather than the latency of individual models.

4.1.2 Preemptive Memory Bandwidth Allocation. As both engines share the memory bandwidth, an arbitration mechanism is needed to allocate memory bandwidth between them. We find that equally treating them with a straightforward round-robin style arbitrage is far from optimal. Giving the C-Engine higher priority benefits more in computing throughput, since CONV tasks contribute the lion's share of GOPS. The rational can be quantified by the following equation:

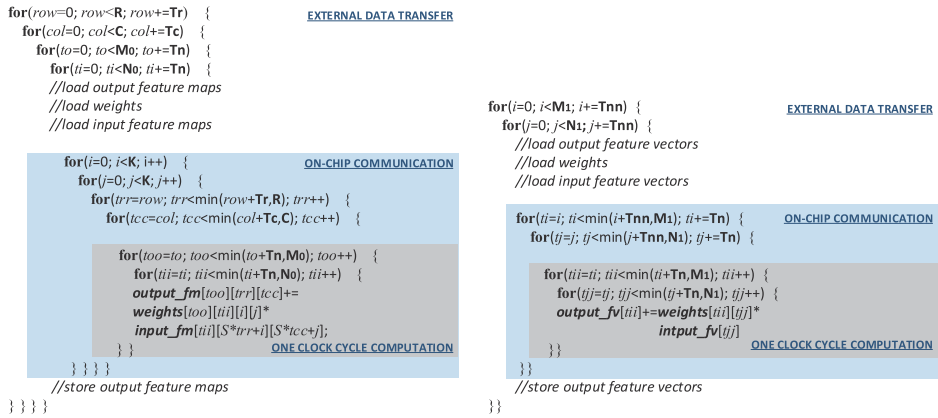
$$P_{\Delta t} = \eta_c \times BW \times \Delta t \times CTC_c + (1 - \eta_c) \times BW \times \Delta t \times CTC_f \propto \eta_c, \tag{2}$$

where $P_{\Delta t}$ represents the computing throughput in a small time span Δt, η_c is the memory bandwidth ratio that the C-Engine occupies in this time span, BW is the total memory bandwidth the platform provides, and CTC_c and CTC_f represent the CTC ratio of CONV layers and FC layers, respectively. Because CTC_c is much larger than CTC_f, $P_{\Delta t}$ is almost proportional to η_c. Thus, offering the C-Engine higher priority to memory bandwidth (larger η_c) will obtain higher computing throughput (larger $P_{\Delta t}$). Experiments also show that this preemptive mechanism improves the computing throughput by 12% on average.

4.2 Dynamic Core Allocation Scheme

To accommodate various NN models, a dynamic core allocation algorithm is proposed to allocate the computing resource in the C-Engine and F-Engine. Here we present the details of this analytical model.

4.2.1 Workload Characterization. First, we present a workload characterization on NN models through the optimized codes of both CONV and FC layers, which contain heavily nested loops, as shown in Figure 8.



(a) Layer-level diversity (b) Model-level diversity

Fig. 8. (a) Pseudo-code of CONV layers in feed-forward phase. (b) Pseudo-code of FC layers in feed-forward phase.

Table 1. Workload Characterization for NN Models

Layer	Workload Parameters	Loop Iterators	Tiling Factors
CONV	M_0: number of output feature maps	to	T_n
	N_0: number of input feature maps	ti	T_n
	R: height of output feature maps	row	T_r
	C: width of output feature maps	col	T_c
	K: kernel size	i, j	-
	S: stride	-	-
FC	M_1: number of output feature vectors	i, ti	T_{nn}, T_n
	N_1: number of output feature vectors	j, tj	T_{nn}, T_n

The execution performance is determined by specific loop tiling methods aiming to fully exploit the data locality (Zhang et al. 2015). The code tiling structure can be described with a set of parameters, called tiling factors, which govern the stride of memory accesses and guide computing engine design. The workload characteristics and the tiling factors are shown in Table 1. T_n is used in both CONV and FC layers, and thus the "one clock cycle computation" of both CONV layers and FC layers can be defined as matrix (size of $T_n \times T_n$) and vector (length of T_n) operations, which provides the possibility to design a general computing engine for both layers. Note that loop iterators (i, j) in CONV layers are not tiled due to the relatively small size of kernels.

4.2.2 Key Parameters of SynergyFlow. Table 2 lists the principal design parameters of SynergyFlow. The "Static parameters" are the platform specifications determined in the design phase. The "Dynamic parameters" can be reconfigured according to the workload characteristics. To dynamically balance core resources between the C-Engine and F-Engine, the key problem is how to determine the number of cores required by each engine. We propose an analytical model to fulfill this purpose.

Table 2. Key Parameters of SynergyFlow

Static parameters	$\boldsymbol{N}$: total amount of cores
	$\boldsymbol{BW}$: total provided external memory bandwidth
	$\boldsymbol{T_n}$: the scale of PE array in each core
	$\boldsymbol{freq}$: clock of each core
	$\boldsymbol{DW}$: data width
Dynamic parameters	$\boldsymbol{N_c}$: total amount of cores in C-Engine
	$\boldsymbol{N_f}$: total amount of cores in F-Engine

4.2.3 Analytical Model for SynergyFlow. Given the workload characterization and platform specifications, we can roughly evaluate the system performance by theoretical estimation. We start with a neural network composed of a single CONV layer and a single FC layer; the total computation demand and memory access demand of the network is estimated as follows:

$$\begin{aligned} C_{op} &= 2 \times R \times C \times M_0 \times N_0 \times K \times K \\ C_{mem} &= \alpha_{in} \times B_{in} + \alpha_{wght} \times B_{wght} + \alpha_{out} \times B_{out}, \end{aligned} \tag{3}$$

where

$$\begin{aligned} B_{in} &= T_n(ST_r + K - 1)(ST_c + K - 1) \\ B_{wght} &= T_n T_n K^2 \\ B_{out} &= T_n T_r T_c \\ \alpha_{in} &= \alpha_{wght} = \frac{M}{T_n} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c} \\ \alpha_{out} &= \frac{M}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c}. \end{aligned} \tag{4}$$

C_{op} denotes the total operations of the layer, while C_{mem} denotes the total external memory access.

Similarly, for FC layers, we have

$$\begin{aligned} F_{op} &= 2 \times M_1 \times N_1 \\ F_{mem} &= \frac{M_1}{T_{nn}} \times \frac{N_1}{T_{nn}} \times T_{nn} + \frac{M_1}{T_{nn}} \times \frac{N_1}{T_{nn}} \times T_{nn} \times T_{nn} + \frac{M_1}{T_{nn}} \times T_{nn} \\ &= \frac{M_1 \times N_1}{T_{nn}} + M_1 \times N_1 + M_1. \end{aligned} \tag{5}$$

Suppose that we have achieved an optimal state under these design parameters; then we can estimate the computation time of both layers as follows:

$$\begin{aligned} C_{time} &\approx \left\lceil \frac{\lceil M_0/T_n \rceil \times \lceil N_0/T_n \rceil}{N_c} \right\rceil \times R \times C \times K \times K \times \frac{1}{freq} \\ F_{time} &\approx \left\lceil \frac{\lceil M_1/T_n \rceil \times \lceil N_1/T_n \rceil}{N_f} \right\rceil \times \frac{1}{freq}. \end{aligned} \tag{6}$$

Given the memory access demand and the computation time, we can further calculate the memory bandwidth requirement:

$$
\begin{aligned}
C_{bw} = \frac{C_{mem}}{C_{time}} &\approx \frac{freq \times M_0 \times N_0}{\left\lceil \frac{\lceil M_0/T_n \rceil \times \lceil N_0/T_n \rceil}{N_c} \right\rceil} \times \left(\frac{1}{T_r \times T_c} + \frac{S^2}{T_n \times K^2} + \frac{1}{N \times K^2} \right) \\
&\approx freq \times N_c \times T_n^2 \times \left(\frac{S^2}{T_n \times K^2} + \frac{1}{T_r \times Tc} \right) \\
F_{bw} = \frac{F_{mem}}{F_{time}} &\approx \frac{freq \times M_1 \times N_1}{\left\lceil \frac{\lceil M_1/T_n \rceil \times \lceil N_1/T_n \rceil}{N_f} \right\rceil} \approx freq \times N_f \times T_n^2.
\end{aligned} \tag{7}
$$

A feasible solution must meet some constraints: (1) BW constraint: the total memory bandwidth of both engines cannot exceed the maximal bandwidth that the platform can provide; (2) core constraint: the total amount of cores in the C-Engine and F-Engine cannot exceed the provided amount of cores; (3) time constraint: the execution time of the F-Engine is close to the time of the C-Engine to avoid the stages of pipeline filled with bubbles. Thus, the design space of all the feasible solutions can be confined by the following constraints:

$$
\begin{cases}
C_{bw} + F_{bw} \le BW/DW \\
N_c + N_f \le N \\
F_{time} \approx C_{time}.
\end{cases} \tag{8}
$$

Substituting Equation (6) and (7) into the above constraints, we finally get the model:

$$
\begin{cases}
N_c = min\left(\left\lceil \frac{BW/DW}{freq \times T_n^2 \times \left(\frac{1}{T_r \times T_c} + \frac{S^2}{T_n^2 \times K} + \frac{1}{R_{ctf}} \right)} \right\rceil, \left\lceil \frac{R_{ctf} \times N}{1 + R_{ctf}} \right\rceil \right) \\
N_f = min\left(\left\lceil \frac{BW/DW}{f \times T_n^2 \times \left(\frac{R_{ctf}}{T_r \times T_c} + \frac{S^2 \times R_{ctf}}{T_n^2 \times K} + 1 \right)} \right\rceil, N - N_c \right),
\end{cases} \tag{9}
$$

where

$$
R_{ctf} = \frac{C_{op}}{F_{op}}, \tag{10}
$$

which indicates the ratio of computation amount of CONV tasks to that of FC tasks.

Note that the model is throughput based rather than latency based because we target batch processing of NN models. In such cases, the throughput is the first concern, rather than the latency of single instances. For example, when training neural networks, the overall throughput is the most important, rather than the latency of single instances. Given the platform specifications, the model indicates that (N_c, N_f) are mainly determined by workload characteristics. In fact, given workloads composed of multiple layers or multiple models, we can estimate the parameters for the workloads. For R_{ctf}, we can easily estimate the total operations of CONV layers and FC layers by Equation (3) and Equation (5). For "K," as the kernel size of CONV layers usually lies in a small range, we can use the average value of each layer weighted by the operations of the layer. "S" is 1 in most cases.

As a case study, we use this model to estimate the SynergyFlow configuration for AlexNet ($R_{ctf} \approx 5.68, K \approx 5, S \approx 1$). Given the platform specification $\{N = 64, T_n = 4, BW = 34.2GB/s, f = 100MHz, DW = 16bits\}$, we can obtain SynergyFlow configuration $\{N_c = 54, N_f = 10\}$. To verify the effectiveness of the model, we further enumerate all the feasible configurations and search the optimal configuration from the design space. Figure 9 shows that the configuration from the analytical model falls into the optimal configuration region obtained by exhaustive searching. Additionally, more explanatory examples with different CTC ratios and CONV-to-FC ratios can be found in Table 6.

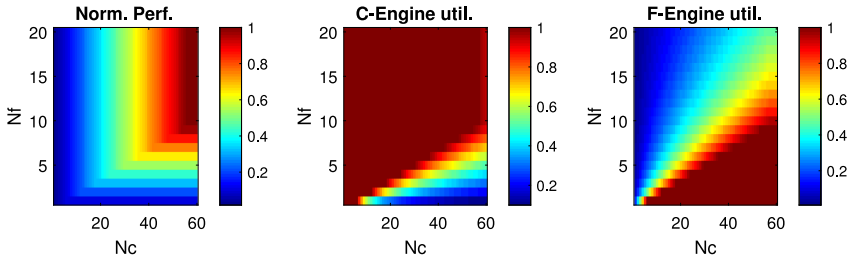
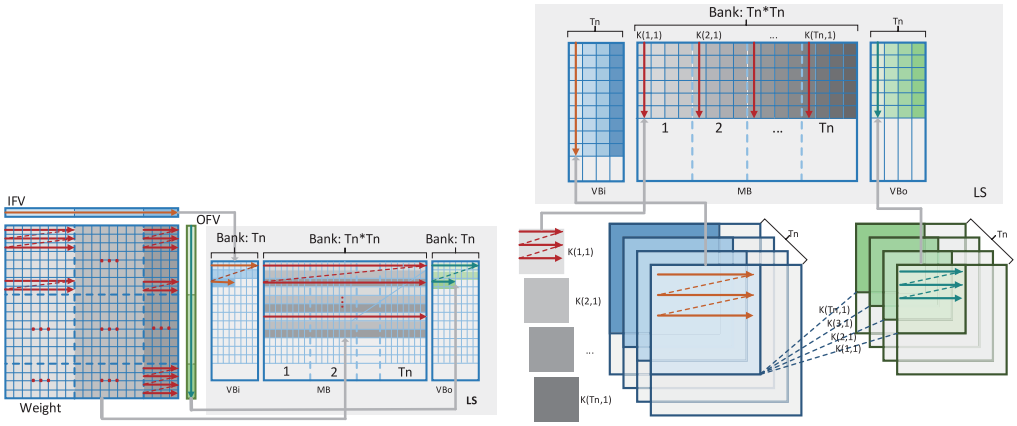


Fig. 9. Design Space Exploration for AlexNet.



(a) Feature vector and weight layout

(b) Feature map and kernel layout

Fig. 10. Data layout in local store.

5 IMPLEMENTATION

This section presents the implementation details of SynergyFlow. As computing resources and memory bandwidth resources are both fine-grained allocated, we implement two types of instruction sets for computation and external memory access, respectively. The cycle-by-cycle control signals are encoded into the instructions, and the resource orchestration is realized by controlling the execution order of the instructions.

5.1 Dataflow Control Instructions

The on-chip storage structures of the accelerator are implemented with scratchpad memory. Since the computation patterns are based on matrix and vector operations, we separate the buffers into a Matrix Buffer (MB) and Vector Buffers (VBi and VBo for input and output vectors). Figure 10 illustrates the data layout in corresponding buffers. Specifically, the MB is dedicated for the synapses, including the kernels in CONV layers and weights in FC layers. The MB is designed with $T_n \times T_n$ banks to support the bank-parallel data access of each core. VBi and VBo have T_n banks, storing the feature maps and feature vectors. According to the access patterns, the feature maps and kernels are stored in a column-first manner, while feature vectors and weights are stored in a row-first manner.

We use "memory instructions" to precisely allocate the memory bandwidth for the C-Engine and F-Engine. These instructions control the communication between buffers and external memory. We can easily encode the data locality information into instructions by matching the data

Table 3. MMU Instruction Flow

W/r	Mem	LS	Stride Access Parameters		
Mode	Addr	Addr	Length	Stride	Repetition
LR	4096	0	4	28	4
SR	0	0	8	8	8
LC	1000	100	32	224	32
SC	1000	100	32	224	32

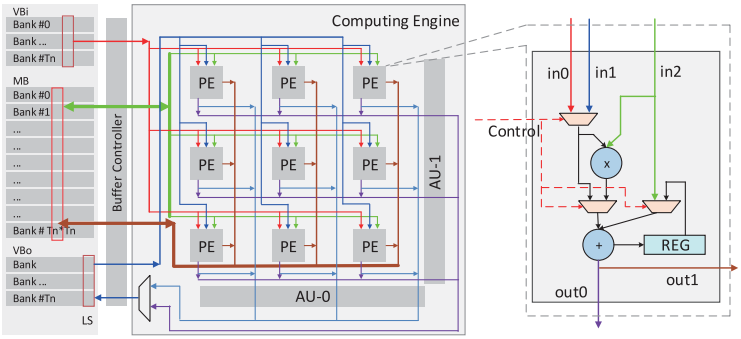


Fig. 11. Architecture of each core and PE architecture.

layout with access patterns. Table 3 presents an example. Each memory instruction has four slots, corresponding to the write/read mode, the external memory address, the LS address, and the parameters for stride access. Take "LS" as an example: "L" indicates this instruction load data from external memory to LS, and "R" indicates the data placed in a row-first manner, for "weights" or "feature vectors." "Mem addr" along with "Stride access parameters" determines the data location in external memory. Specifically, "length" determines the length of each memory segment, "stride" stands for the address stride for the next segment, and "repetition" stands for the total number of segments. For the first instruction, the source data consist of four segments (repetition = 4), each segment consists of four data elements (length = 4), the stride between two segments is 28. Hence, the location of the source data is &0 − &3, &28 − &31, &56 − &59, &84 − &87). The source data are stored in &4096 of MB in a row-first manner.

According to the pseudo-code in Figure 8, a layer is partitioned into multiple sublayers by the tiling factors (the outermost loop), and each sublayer has data transfer with external memory. We package the memory instruction flow for each sublayer into an instruction block. We can control the issuing of these instruction blocks, thereby fine-tuning the memory bandwidth allocation.

5.2 Computation Control Instructions

Figure 11 shows the architecture of the core in the C-Engine and F-Engine, which is composed of multiple PEs in a mesh organization and two Aggregation Units (AUs). The PE mesh is in a square pattern to efficiently support both the inference phase and training phase as they can be considered as symmetric phases. The mesh size is $T_n \times T_n$ corresponding to the tiling factors in Table 1. Each PE can execute fixed multiply-accumulation operations. The AU is composed of adder trees and REGs to accumulate the partial products for a row or a column of PEs. The core can simultaneously read vectors and matrices from VBs and the MB through the interconnects and distribute them to different PEs. Specifically, the core can support the main types of computation patterns in neural

Table 4. Computation Instruction Flow

Inst ID	Compute Mode	Input 0	Input 1	Output	Acc. Flag
inst-0	prodmv	0	4096	-	0
inst-1	prodmv	1	4097	-	1
inst-2	prodmv	28	4098	-	1
inst-3	prodmv	29	4099	20480	1
inst-4	prodmv	1	4096	-	0
inst-5	prodmv	2	4097	-	1
inst-6	prodmv	29	4098	-	1
inst-7	prodmv	30	4099	20481	1

networks. For VMP, the core reads a vector from VBi and a matrix from the MB. The vector is distributed in a row-sharing manner (PEs in a row share the same data). AU-0 is used to aggregate the partial products for each column and stores the results to VBo. Similarly for MVP, the core also reads a vector and matrix, but the vector is distributed in a column-sharing manner, and AU-1 is used to aggregate the partial products. For VVP, the core reads two vectors for VBs, one distributed in a row-sharing manner, another in a column-sharing manner; "out1s" in each PE are grouped as a matrix and stored back to the MB. In a nutshell, the core can support the typical computation patterns in both CONV and FC layers.

Prior work has claimed that the performance of CONV layers depends heavily on the dataflow design (Chen et al. 2017), which has motivated many accelerator designs to optimize the performance of CONV layers by exploiting flexible dataflow designs, such as Eyeriss (Chen et al. 2017) and FlexFlow (Lu et al. 2017). In SynergyFlow, the dataflow for CONV layers is similar to Zhang et al. (2015) and falls into "SFMNSS" (the category described in FlexFlow (Lu et al. 2017)). Although using this rigid dataflow may degrade the performance of CONV layers, it can still verify the effectiveness of the key idea of SynergyFlow: orchestrating CONV and FC layers to boost the performance of entire NN models. More importantly, the dataflow optimizations are orthogonal to SynergyFlow since SynergyFlow can also enable them to orchestrate CONV and FC layers to boost resource utilization. In other words, the prior flexible dataflow optimizations can also be leveraged by SynergyFlow to further improve the performance of CONV layers. We omit the dataflow optimizations of CONV layers for brevity in this work.

The core logic is controlled by computation instructions. Each instruction has three slots, corresponding to the computing mode, the three operands, and an "accumulative" flag. Table 4 presents an example of a convolutional operation with a kernel size of 2×2 and input feature map size of 28×28. The details of the instructions are omitted due to space limitations.

The computation instructions are also organized in blocks for each sublayer. The memory instructions and computation instructions are grouped in pairs for each sublayer. The computation instruction cannot be executed until the corresponding memory instructions are finished. We developed a dedicated code generator on Matlab, which is responsible for mapping different neural networks to the accelerators. Hence, SynergyFlow can provide acceleration for different NN models. The code generator takes the network structure as input and generates Memory Instructions and Computation Instructions for both the C-Engine and F-Engine. There are four main types of instruction blocks: CONV Memory Instruction Blocks (CMIBs), FC Memory Instruction Blocks (FMIBs), CONV Computing Instruction Blocks (CCIBs), and FC Computing Instruction Blocks (FCIBs). The MIBs are responsible for the memory accesses between off-chip and on-chip, while the CIBs are responsible for the on-chip computations. As the C-Engine and F-Engine share the external memory bandwidth, there exists a resource contention between FMIBs and CMIBs. Thus,

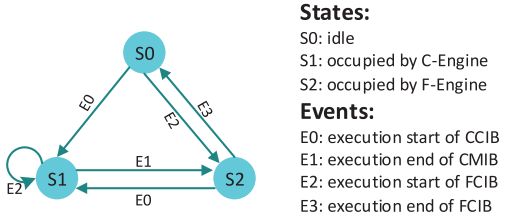


Fig. 12. Finite state machine for instruction block control.

a dedicated Finite State Machine (FSM) is designed to control the execution flow of the instruction blocks, as shown in Figure 12. The C-Engine and F-Engine share the bandwidth in a time-division-multiplexing manner, and the FSM offers execution priority to CMIB, thus giving memory access priority to the C-Engine. Specifically, three states for the memory bandwidth occupation (i.e., $S0$, $S1$, and $S2$) and four events for the instruction executions are defined in the FSM. For example, if the current state is S_0, the memory bandwidth is currently idle and no MIBs are being executed. It jumps to $S1$ when a CCIB starts to be executed (E0), to prepare the data for the next CCIB. After the current CMIB is finished (E1), it jumps to $S2$ to prepare data for the F-Engine. The other details of the FSM are similar and omitted due to limited space. Note that the state of CIBs will invoke the corresponding MIBs to prepare the data for next CIB due to the double-buffering techniques mentioned in Section 3.1. However, the exception is that when the current state is $S1$, the execution starts of FCIB (E2) do not change the current state, thereby ensuring that the C-Engine can always be fed with data in a timely manner.

6 EVALUATION

6.1 Experimental Setup

Accelerator Simulator. We developed a cycle-accurate C++ simulator, DeepWater (already open-sourced on GitHub at https://github.com/top-speed/DeepWater) for the architecture exploration. The simulator takes the instruction flow as input and generates cycle-accurate execution flow. The simulator is also used as the specification for FPGA implementation.

FPGA Implementation. The FPGA implementation is built on the board Nallatech P395-AB. It consists of the Altera Stratix V GXAB, 4×8 *GB* DDR3 SDRAM. Our implementation uses 16-bit fixed-point numbers ($DW = 16bits$), which is good enough for neural network training and inference (Qiu et al. 2016). We implement a 64-core system ($N = 64$), and each core contains a 4×4 PE array ($T_n = 4$). The working frequency is 100MHz. In the current implementation on the Nallatech development board, the bandwidth of the external memory bus is limited to 34.2GB/s (BW = 34.2GB/s) since we use dual-channel DDR3-2133. Detailed resource utilization is presented in Table 5.

GPU Implementation. The GPU platform is Nvidia K20M (2,496 CUDA cores with 5GB GDDR5 320-bit memory, 208GB/s memory bandwidth, 3.52TFlops peak, 28nm technology), and the GPU can also report its power usage. The experiments are implemented on Caffe (Jia et al. 2014).

Architectural Baselines. Monolithic architectures are selected as the baselines. In fact, the implementation of SynergyFlow can also work as a monolithic architecture (MonoArch) by enabling all the cores acting as the C-Engine or F-Engine for CONV layers or FC layers, respectively. Similarly, SynergyFlow can also work as an ATA, by statically partitioning the cores to the C-Engine and F-Engine. We implement five ATAs with typical configurations, denoted as ATA(N_c, N_f): ATA-1(32,32), ATA-2(48,16), ATA-3(56,8), ATA-4(60,4), ATA-5(63,1). Because MonoArch and ATAs use almost the same hardware resources with SynergyFlow, we can make a fair comparison among these baselines. Because the DRAM bandwidth provisioning of prior solutions is widely ranged,

Table 5. FPGA Resource Report

Resource	Logic Elements	DSP Blocks	M20K RAMs
Used	246k	292	1,385
Total	952k	352	2,640

Table 6. Benchmarks

Workload	R_{ctf}	K	SF Config.	Description
AlexNet	5.68	5	(54,10)	CNN for ImageNet (Krizhevsky et al. 2012), 100 instances
VGG11	60.08	3	(62,2)	CNN for ImageNet (Simonyan and Zisserman 2014), 100 instances
DBN-M	0	0	(0,11)	DBN for MNIST (Hinton et al. 2006), 10,000 instances
Batch1	10.87	4	(58,6)	VGG11 (100 instances) + AlexNet (2,000 instances)
Batch2	2.35	5	(26,11)	AlexNet (100 instances) + DBN-M (5,000 instances)
Batch3	16.29	3	(60,4)	VGG11 (100 instances) + DBN-M (20,000 instances)

it is unfair to make straightforward comparisons by simply comparing their experimental results. Moreover, there are many other optimization techniques in prior solutions that are hard to reimplement. To make the comparison fair and clear, we use the idealized monolithic architecture to represent the prior approaches.

Benchmarks. We use the representative VGG11 (Simonyan and Zisserman 2014), AlexNet (Krizhevsky et al. 2012), and DBN (Hinton et al. 2006) as the workloads. AlexNet (Krizhevsky et al. 2012), consisting of five CONV layers and three FC layers, is the winner of ILSVRC 2012, achieving the top 5 accuracy at 84.7%.

The VGG model achieved accuracy at 92.6% and won first place in the image classification task of ILSVRC 2014 (Simonyan and Zisserman 2014). VGG models contain five CONV layer groups and three FC layers. A series of VGG models include VGG11, VGG13, VGG16, and VGG19, differentiated by the number of layers.

The Deep Belief Network (Hinton et al. 2006) is formed by stacked RBMs and trained with a greedy policy. DBNs are graphical models that learn to extract a deep hierarchical representation of the training data. The DBN model in Hinton et al. (2006) achieves 1.25% errors for the MNIST test set without resorting to any knowledge of geometries.

Since the main arena of SynergyFlow is in the application scenarios with intensive model-level parallelism, we also group the models as batch workloads to mimic such scenarios. The above benchmarks are ideal ingredients to build the synthetic workloads because these models contain both layer-level diversity and model-level diversity. The characteristics of the workloads are shown in Table 6, along with the configuration of SynergyFlow for these workloads.

In this work, we mainly focused on two kinds of diversity in neural network models: model-level diversity and layer-level diversity. Under this condition, the diversities can be modeled by the R_{ctf}. Hence, we select the representative workloads with different R_{ctf} ranging from 0 (DBN-M) to 60.08 (VGG11). These workloads exhibit the diversities that we focused on.

Training. The training phase is also based on the above benchmarks. Since it takes days, even weeks, to train large neural networks such as VGG and AlexNet, we only clip an interval in the training phase (epoch = 10) to make the time manageable. The training batch size is set as 50.

6.2 Experimental Results

Since SynergyFlow is evolved from monolithic architectures, we first make the comparison between them.

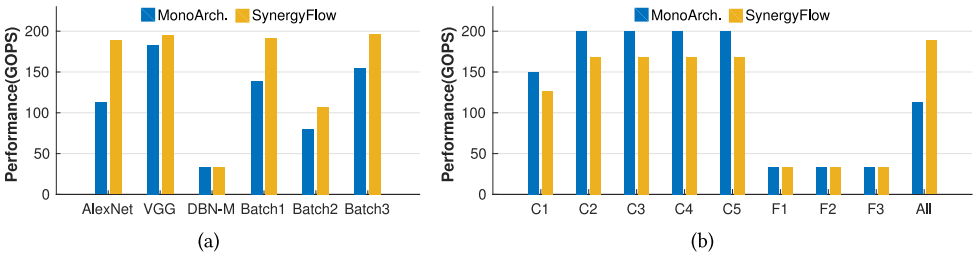


Fig. 13. Performance comparison with monolithic architectures: (a) performance on various models (inference); (b) performance on AlexNet (inference).

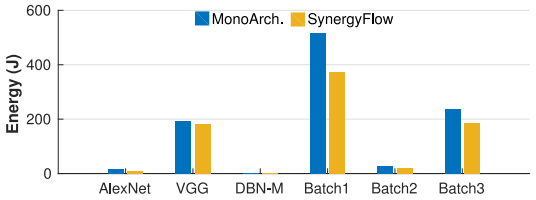
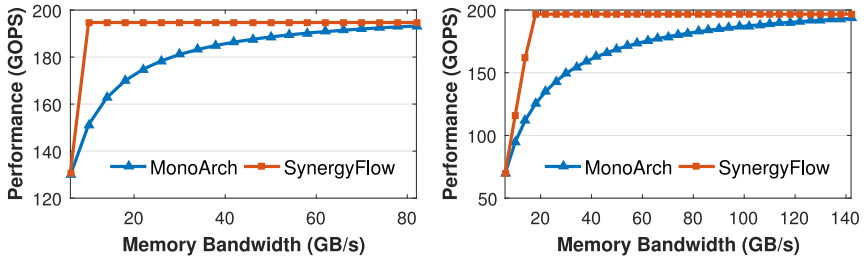


Fig. 14. The energy comparison between SynergyFlow and MonoArch.

6.2.1 Performance Comparison with Monolithic Architectures. Compared with monolithic architectures, SynergyFlow achieves a performance improvement of 33.78% on average with the same hardware resources, as shown in Figure 13(a). Specifically, SynergyFlow achieves up to 67.41% performance improvement on AlexNet, which mainly comes from the overlapping execution of CONV and FC layers. SynergyFlow achieves little performance improvement on CONV-dominant workloads (7.02% for VGG11) and FC-dominant workloads (DBN), since there only exists limited layer-level diversity for orchestration. However, if leveraging model-level diversity (Batch2), SynergyFlow can also achieve 33.50% performance improvement.

Figure 13(b) further illustrates the performance improvement over MonoArch by analyzing the layer-wise performance of AlexNet. MonoArch wins over SynergyFlow on the performance of each layer. However, without orchestration, MonoArch suffers underutilization of BW when processing CONV layers and underutilization of PEs when processing FC layers, causing a relatively poor overall performance. By contrast, by overlapping CONV layers and FC layers, SynergyFlow boosts both PE utilization and BW utilization and significantly outperforms MonoArch. Note that the performance on $C1$ is dwarfed by other CONV layers. The reason is that $C1$ is too "small" (from the number of feature maps perspective, three input feature maps to 64 output feature maps) in contrast to a relatively larger PE array (4×4), which causes underutilization of PEs. But this "corner" case does not affect much of the overall NN performance.

6.2.2 Energy Consumption Comparison with Monolithic Architectures. In Figure 14, we report the energy comparison between SynergyFlow and MonoArch. On average, SynergyFlow reduces the energy consumption by 23.1% on average over MonoArch across the workloads. The most striking result is that SynergyFlow achieves an energy reduction by 40.2% on MonoArch. The reduction of energy varies widely across the workloads depending on R_{ctf}. The energy reduction of SynergyFlow mainly stems from the improvement of performance; SynergyFlow achieves higher performance and largely reduces the processing time.



(a) Performance on VGG11 (b) Performance on Batch3

Fig. 15. The impact of memory bandwidth on SynergyFlow(inference).

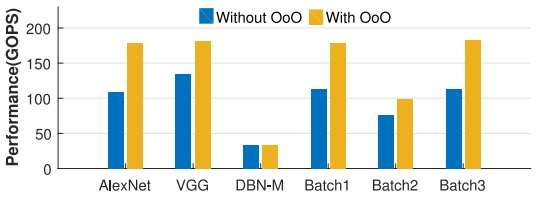


Fig. 16. The impact of OoO execution on SynergyFlow (training).

6.2.3 The Impact of Memory Bandwidth. Since BW is a main specification of SynergyFlow, we tested the performance of the accelerator under different BW configurations through simulation, as shown in Figure 15.

For VGG11, SynergyFlow can maintain its high performance even with surprisingly low provisioning of BW. By contrast, the performance of MonoArch degrades gradually and significantly. We can see a clear "knee point" in the performance curves, even though the knee points are located in different BW. Below this point, SynergyFlow's performance benefit quickly diminishes, even still above MonoArch. However, beyond the knee points, SynergyFlow exhibits attractive robustness to BW configurations. By contrast, MonoArch is much more sensitive to the change of BW. SynergyFlow accomplishes this robustness through a more sophisticated BW allocation scheme. Furthermore, the different BW on knee points also implies that an appropriate BW specification of SynergyFlow should be designed with respect to the workloads of interest. The specific value can be obtained through, for example, Monte Carlo simulation, which is beyond the scope of this article. Moreover, it turns out that the bandwidth requirement for MonoArch can go up to 140GB/s to reach the peak on Batch3, which is hard, if not totally impossible, to be satisfied in a single DRAM setup with current DRAM technology. Note that it is quite hard to precisely control the DRAM bandwidth provisioning because it dictates the hardware modifications to DRAM, which requires intensive engineering work. Hence, we take a suboptimal alternative by using the simulation results.

6.2.4 The Impact of Out-of-Order Layer Scheduling. Regarding the layer scheduling mechanism proposed in (Section 4.1.1), we evaluate the impact of this mechanism on SynergyFlow, as shown in Figure 16. On average, the scheduling mechanism improves the training performance by 41.2% on average, and up to 63.5% for AlexNet. Due to data dependency, SynergyFlow without OoO execution suffers idles of either the C-Engine or F-Engine, while OoO execution can keep both engines busy and contention-free.

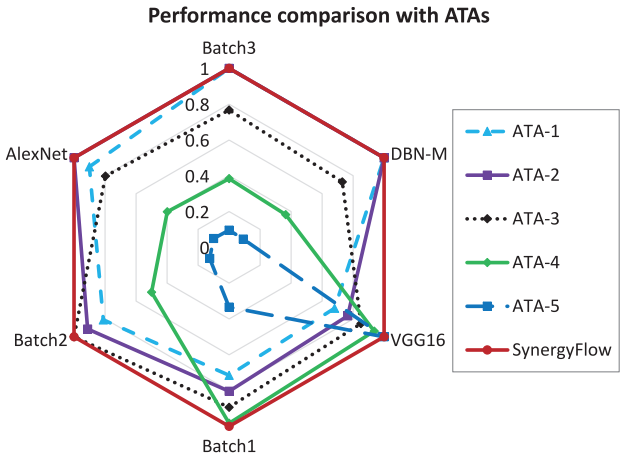


Fig. 17. The impact of the dynamic core allocation scheme (normalized to the performance of SynergyFlow).

6.2.5 The Impact of Dynamic Core Allocation. As a key feature, SynergyFlow supports dynamic core allocation as proposed in Section 4.2. The benefit can be highlighted by comparing it with asymmetric tandem architectures, as shown in Figure 17.

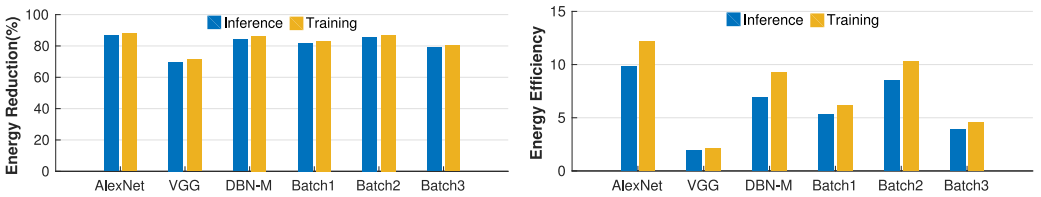
SynergyFlow can achieve the best performance in all the workloads, while ATA can only obtain the highest performance on specific workloads. Specifically, ATA-1 and ATA-2 can achieve high performance on FC-dominant workloads since they have a relatively more powerful F-Engine compared with other ATAs, but they perform relatively poor on CONV-dominant workloads due to the relatively weak C-Engine; i.e., ATA-2 suffers 23.4% performance loss on VGG11 compared to SynergyFlow. By contrast, ATA-4 and ATA-5 have relatively powerful C-Engines and weak F-Engines, and thus achieve high performance on CONV-dominant and poor performance on FC-dominant workloads. By contrast, SynergyFlow maintains a high performance level among various models, because SynergyFlow can dynamically reconfigure the computing capacity of the C-Engine and F-Engine according to workload characteristics. Overall, the dynamic allocation scheme helps SynergyFlow to well handle the model-level diversity, which again puts SynergyFlow in a winning position over ATAs.

6.2.6 Comparison with GPU-Based Solutions. GPUs have been proven to be promising for NN applications. The comparison between SynergyFlow and GPU-based solutions is shown in Table 7. In the inference phase, unsurprisingly, the performance of GPU on CONV-dominant workloads (i.e., with large R_{ctf}, such as VGG11, Batch 1) is higher than SynergyFlow. For example, GPU delivers a 1.7× performance speedup over SynergyFlow in VGG11, a typical CONV-dominant benchmark. However, this result cannot deny the merit of SynergyFlow dealing with diversity. For example, GPU suffers severe performance degradation on AlexNet, a benchmark with modest layer diversity. By contrast, SynergyFlow outperforms GPU by 31.9% on AlexNet. This result again confirms that SynergyFlow is designed for diversified NN applications, not only convolutional layers. In fact, this objective also differentiates SynergyFlow from many prior convolution-oriented solutions.

As for training, SynergyFlow can maintain its high performance compared to the inference phase, because SynergyFlow intrinsically supports MVP and VMP, eliminating the need for weight transpose in training phase. For GPU, however, the matrix transpose consumes some time, thus degrading the overall performance by about 8.1%. Note that we test GPU using the default 32-bit

Table 7. Performance Comparison with GPU

Workload		Inference			Training (Epoch=10)		
		GPU	SF	Speedup	GPU	SF	Speedup
AlexNet	time(s)	0.51	0.39		17.77	12.22	
	GOPS	144.20	188.52	1.31x	123.14	179.09	1.45x
VGG11	time(s)	4.17	7.22		141.86	233.03	
	GOPS	337.51	194.70	0.58x	297.44	181.07	0.61x
DBN-M	time(s)	1.00	0.91		35.72	28.21	
	GOPS	31.00	33.94	1.09x	26.00	32.92	1.27x
Batch1	time(s)	14.28	14.93		497.19	481.46	
	GOPS	200.61	191.96	0.96x	172.87	178.52	1.03x
Batch2	time(s)	1.01	0.83		35.63	26.67	
	GOPS	87.96	106.93	1.22x	74.44	99.45	1.34x
Batch3	time(s)	6.16	7.46		213.30	240.80	
	GOPS	238.21	196.72	0.83x	206.53	182.95	0.89x



(a) Energy reduction compared to GPU baseline

(b) Energy Efficiency w.r.t. GPU baseline

Fig. 18. Energy efficiency comparison with GPU.

floating point because of convenience of implementation. Using 16-bit fixed-point numbers on GPUs dictates massive labor-intensive engineering work, if not being totally impossible. Furthermore, the performance of GPU will not be doubled since there exist lots of nonarithmetic operations that cannot benefit from the shorter data width. Even so, the performance of our implementation is still comparable with GPU scaled to 16-bit fixed-point numbers. We further normalize their performance to their peaks; SynergyFlow can achieve up to 95% of its peak, while the GPU reaches merely 10% of its peak, which is consistent with the results reported by Chen et al. (2014a, 2014b).

SynergyFlow achieves up to 81.2% energy reduction compared to the GPU baseline in the inference phase, as shown in Figure 18(a). The minimum energy reduction is 69.3% for VGG16, while the highest energy reduction is 86.7% for AlexNet, which also reflects that SynergyFlow is designed for diversified NN applications. The energy reduction is also significant (82.7% on average) in the training phase.

Furthermore, SynergyFlow beats GPU in terms of energy efficiency, as shown in Figure 18(b). The minimum and best energy efficiency improvement in the training phase also lies in VGG16 (1.9×) and AlexNet (9.8×), respectively. SynergyFlow achieves 6.1× and 7.4× higher energy efficiency on average over GPU in the inference and training phase, respectively.

6.2.7 Comparison with Existing Approaches. The comparison between SynergyFlow and previous approaches is shown in Table 8. Since the target applications and platforms differ among these

Table 8. Comparison with Previous Implementations

	Zhang	Qiu	Suda	DianNao	DaDianNao	Ours
Precision	32 bits float	16 bits fixed	8–16 bits fixed	16 bits fixed	16 bits fixed	16 bits fixed
Frequency	100MHz	150MHz	120MHz	0.98GHz	606MHz	100MHz
Bandwidth (GB/s)	12.8	4.2	6.0	250	25.6	34.2
Platform	Virtex7 VX485T	Zynq XC7Z045	Stratix-V GSD8	ASIC	ASIC	Stratix-V GXAB
NN models	AlexNet	VGG-16-SVD	VGG &AlexNet	General	General	General
Perf (CONV only)	61.62GOPS	187.80GOPS	136.5GOPS	130~195x[‡] (w.r.t. SIMD)	160x[§] (w.r.t. GPU)	198.3
Perf (entire NN)	NA	136.97 (V*) 5.13[†] (A*)	117.8 (V) 72.4 (A)	NA	63x[§] (w.r.t. GPU)	194.7 (V) 188.5 (A)
Degradation ratio	NA	27.1% (V) 97.3% (A)	13.9% (V) 47.0% (A)	NA	60.6%[§]	**1.8% (V)** **5.0% (A)**

*"V" short for VGG-16, "A" short for AlexNet.
[†]The performance of Qiu (Qiu et al. 2016) on AlexNet is estimated by its CONV layer performance and FC layer performance.
[‡]The speedup is based on specific layers.
[§]We use the four-node version of DaDianNao (Chen et al. 2014b). Since the performance in Chen et al. (2014b) is quantified by the speedup over GPU, which also suffers performance degradation on entire NNs, we give a conservative estimation on its degradation ratio (the real value is even higher).

solutions, it is hardly possible to make an apple-to-apple comparison. Even so, we think comparing the reported statistics is still informative to make a qualitative evaluation.

Also, we would like to remind that previous approaches often suffer performance degradation when processing entire CNN models. To quantify the performance degradation of processing entire NNs, we define a metric:

$$Degradation\ Ratio = 1 - \frac{Perf_{CONV}}{Perf_{NN}}, \tag{11}$$

where $Perf_{CONV}$ and $Perf_{NN}$ denote the performance of CONV layers and entire NN, respectively. Prior implementations suffer at least 47% performance loss on AlexNet, while our approach can maintain a relatively constant high performance.

7 RELATED WORK

Many neural network accelerators focus on optimizing CONV layers from a computing perspective or memory perspective. Some implementations (Cadambi et al. 2010; Chakradhar et al. 2010; Farabet et al. 2009; Sankaradas et al. 2009) focus on optimizing computation engines. They implement complete CNN applications on FPGA but exploit different parallelism opportunities. Some work (Peemen et al. 2013) chooses to maximize data reuse and reduce bandwidth requirement to the minimum, but has to reconfigure FPGA for different CONV layers. Some work (Zhang et al. 2015) focuses on the balance between computing resources and memory bandwidth for CONV layers. FusedCNN (Alwani et al. 2016) proposes a CNN accelerator architecture that focuses on cross-convolutional layer optimizations over FPGAs but provides little insight for the optimizations on FC layers.

Another class of accelerators focuses on entire NN models. Diannao (Chen et al. 2014a) designed a general NN accelerator that focuses on memory bandwidth utilization. Though adaptive

for various NN layers, they use a monolithic architecture for CONV and FC layers, and thus cannot maximize resource utilization due to layer-level diversity and model-level diversity. Alternatively, they require very high external memory bandwidth to maintain its high performance. FPGA-Qiu (Qiu et al. 2016) focused on both CONV and FC layers. Also as a monolithic architecture, they use SVD to reduce the complexity of FC layers. FPGA-Suda (Suda et al. 2016) proposed a methodology to find the optimal accelerator design for any CNN model implementation. As an asymmetric tandem architecture, they use different computing engines for CONV and FC layers. FlexFlow (Lu et al. 2017) demonstrates a flexible dataflow that can support all types of fine-grained parallelism of CONV layers. DNA (Tu et al. 2017) optimized the dataflow by leveraging weight, input, and output reuse in the same fabric. Eyeriss (Chen et al. 2017) proposed a row-stationary dataflow that minimizes the data movement energy of convolutional layers on a spatial architecture. MAERI (Kwon et al. 2018) proposed to enable flexible dataflow mapping over NN accelerators via reconfigurable interconnects. These accelerators are mainly optimized for CONV layers and improve the performance of CONV layers and FC layers individually; hence, the performance degradation is inevitable when they accommodate the entire NN model because of the diverse resource requirements. These works are complementary to SynergyFlow, since SynergyFlow can enable them to orchestrate CONV and FC layers to boost resource utilization.

Additionally, we mention the recurrent neural network (RNN), which is also an important class of artificial neural network and has shown high performance in many applications. SynergyFlow primarily focused on CNNs but did not consider RNNs because of the main difference between them: in each neuron of RNNs, the output of the previous time step is fed as input of the next time step, and the data dependency of RNN layers is much more complicated than for CNNs. Hence, the acceleration of CNNs and RNNs usually requires independent compute units, such as DNPU (Shin et al. 2017). It may dictate complicated hardware support to exploit the complementary effect of resource requirements, which may not be totally possible. Considering this situation, we would like to forgo the new exploration to RNN but leave it as future work.

Sparse NNs have emerged as an effective solution to reduce the amount of computation and memory accesses while maintaining high accuracy. Cambricon-X (Zhang et al. 2016) and Cnvlutin (Albericio et al. 2016) remove ineffective computations from zero weights and activations, respectively. SCNN (Parashar et al. 2017) leverages the sparsity in both weights and activations. EIE (Han et al. 2016) is an energy-efficient inference engine that performs inference on the compressed FC layers and accelerates the resulting sparse matrix-vector multiplication. The aim of SynergyFlow is not to deal with sparsity, but rather to design a CNN accelerator for the batch processing of NN models.

Partitioning a processing array among different tasks has been advocated in the field of massively parallel chip multiprocessors (Li 2015). Ma et al. (2016) proposed an analytical framework for estimating scale-out and scale-up power efficiency of heterogeneous many cores. However, there are fundamental differences between SynergyFlow and these works. For general-purpose processors, the partitioning among the tasks is trivial; for domain-specific accelerators, the partitioning is hardly applicable for monolithic architectures because they are designed for fine-grained data or thread parallelism. It is infeasible for the PEs in a monolithic architecture to run different tasks because they are tightly coupled.

Additionally, we mention the work that investigates the codesign models of algorithm and hardware. Czechowski et al. (2011) proposed a balance principle of how to design computational algorithms for particular hardware architectures and vice versa and presented a prediction of how future systems should be codesigned. He et al. proposed techniques for approximate NN acceleration (He et al. 2018a, 2018b; Ke et al. 2018).

To sum up, our work investigates the diversities in neural network models and takes advantage of the complementary effect in resource demand, thus boosting the performance and resource utilization.

8 CONCLUSION

In this work, we proposed an elastic accelerator architecture called SynergyFlow for neural network acceleration. SynergyFlow can efficiently handle the diversities in neural network models and intrinsically support model-level parallelism. SynergyFlow boosts the resource utilization by exploiting the complementary effect of resource demand in neural network models and can dynamically reconfigure itself for various models. We also present an analytical model for resource pool configuration of SynergyFlow according to the workload characteristics. Under the frequency of 100MHz, SynergyFlow improves the performance by 33.78% on average compared with previous approaches using the same hardware resources.

REFERENCES

J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 1–13.

Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

Y. Bengio, A. Courville, and P. Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 8 (Aug. 2013), 1798–1828. DOI: https://doi.org/10.1109/TPAMI.2013.50

Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. 2010. A programmable parallel accelerator for learning and classification. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 273–284.

Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 247–257.

Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014a. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *Architectural Support for Programming Languages and Operating Systems* 49, 4 (2014), 269–284.

Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014b. DaDianNao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47'14)*. IEEE Computer Society, 609–622.

Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

Kent Czechowski, Casey Battaglino, Chris McClanahan, Aparna Chandramowlishwaran, and Richard W Vuduc. 2011. Balance principles for algorithm-architecture co-design. *HotPar* 11 (2011), 9–9.

George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 8609–8613.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, and Quoc V. Le. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*. 1223–1231.

Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.

Harry Dwyer and Hwa C. Torng. 1992. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. *ACM SIGMICRO Newsletter* 23, 1–2 (1992), 272–281.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM MICRO*. IEEE Computer Society, 449–460.

Kevin Fan, Manjunath Kudlur, Ganesh Dasika, and Scott Mahlke. 2009. Bridging the computation gap between programmable processors and hardwired accelerators. In *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*. IEEE, 313–322.

Clément Farabet, Cyril Poulet, Jefferson Y. Han, and Yann LeCun. 2009. CNP: An FPGA-based processor for convolutional networks. In *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 32–37.

Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'11)*. ACM, New York, 69–77.

Junli Gu, Maohua Zhu, Zhitao Zhou, Feng Zhang, Zhen Lin, Qianfeng Zhang, and Mauricio Breternitz. 2014. Implementation and evaluation of deep neural networks (DNN) on mainstream heterogeneous systems. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys'14)*. ACM, 12.

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 37–47.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 243–254.

Xin He, Liu Ke, Wenyan Lu, Guihai Yan, and Xuan Zhang. 2018a. AxTrain: Hardware-oriented neural network training for approximate inference. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'18)*. ACM, New York, Article 20, 6 pages. DOI: https://doi.org/10.1145/3218603.3218643

X. He, W. Lu, G. Yan, and X. Zhang. 2018b. Joint design of training and hardware towards efficient and accuracy-scalable neural network inference. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2018), 1–1. DOI: https://doi.org/10.1109/JETCAS.2018.2845396

Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural Computation* 18, 7 (2006), 1527–1554.

Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*. ACM, 2333–2338.

W. Huang, H. Hong, G. Song, and K. Xie. 2014. Deep process neural network for temporal deep learning. In *2014 International Joint Conference on Neural Networks (IJCNN'14)*. IEEE, 465–472. DOI: https://doi.org/10.1109/IJCNN.2014.6889533

S. Ji, W. Xu, M. Yang, and K. Yu. 2013. 3D convolutional neural networks for human action recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 1 (Jan. 2013), 221–231. DOI: https://doi.org/10.1109/TPAMI.2012.59

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia (MM'14)*. ACM, New York, 675–678. DOI: https://doi.org/10.1145/2647868.2654889

Liu Ke, Xin He, and Xuan Zhang. 2018. NNest: Early-stage design space exploration tool for neural network inference accelerators. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'18)*. ACM, New York, Article 4, 6 pages. DOI: https://doi.org/10.1145/3218603.3218647

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Williamsburg, 461–475.

Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th ICML (ICML'07)*. ACM, New York, 473–480.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (Nov. 1998), 2278–2324. DOI: https://doi.org/10.1109/5.726791

B. Li, Y. Wang, Y. Wang, Y. Chen, and H. Yang. 2014. Training itself: Mixed-signal training acceleration for Memristor-based neural network. In *2014 19th Asia and South Pacific (ASP-DAC'14)*. IEEE, 361–366. DOI: https://doi.org/10.1109/ASPDAC.2014.6742916

J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. 2018a. CCR: A concise convolution rule for sparse neural network accelerators. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE'18)*. IEEE, 189–194. DOI: https://doi.org/10.23919/DATE.2018.8342001

J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. 2018b. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE'18)*. IEEE, 343–348. DOI: https://doi.org/10.23919/DATE.2018.8342033

Keqin Li. 2015. Optimal partitioning of a multicore server processor. *Journal of Supercomputing* 71, 10 (2015), 3744–3769.

Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. Pudiannao: A polyvalent machine learning accelerator. In *SIGARCH Comput. Archit. News*, 43, 1 (2015), 369–381.

Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, 553–564.

J. Ma, G. Yan, Y. Han, and X. Li. 2016. An analytical framework for estimating scale-out and scale-up power efficiency of heterogeneous manycores. *IEEE Trans. Comput.* 1 (2016), 1–1.

Volodymyr Mnih and Geoffrey E Hinton. 2012. Learning to label aerial images from noisy data. In *Proceedings of the 29th ICML (ICML'12)*. 567–574.

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

Maurice Peemen, Arnaud A. A. Setio, Bart Mesman, and Henk Corporaal. 2013. Memory-centric accelerator design for convolutional neural networks. In *31st IEEE International Conference on Computer Design (ICCD'13)*. Institute of Electrical and Electronics Engineers (IEEE).

Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *Field Programmable Gate Arrays*. 26–35.

Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. 2009. A massively parallel coprocessor for convolutional neural networks. In *2009 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 53–60.

D. Shin, J. Lee, J. Lee, and H. J. Yoo. 2017. 14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In *2017 IEEE International Solid-State Circuits Conference (ISSCC'17)*. IEEE, 240–241.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, and Marc Lanctot. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv Preprint arXiv:1409.1556* (2014).

Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-optimized openCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 16–25.

F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. 2017. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration Systems (VLSI'17)* 25, 8 (2017), 2220–2233.

Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Proceeding of the Deep Learning and Unsupervised Feature Learning NIPS Workshop*, Vol. 1. Citeseer, 4.

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for foating-point programs and multicore architectures. *Communications of the Association for Computing Machinery*.

Samuel Webb Williams. 2008. *Auto-Tuning Performance on Multicore Computers*. University of California, Berkeley.

Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Field Programmable Gate Arrays*. 161–170.

Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

Received January 2018; revised September 2018; accepted September 2018