FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks

Wenyan Lu†, Guihai Yan, Jiajun Li†, Shijun Gong†, Yinhe Han, Xiaowei Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
†*University of Chinese Academy of Sciences*
{*luwenyan, yan, lijiajun, gongshijun, yinhes, lxw*}*@ict.ac.cn*

***Abstract*—Convolutional Neural Networks (CNN) are very computation-intensive. Recently, a lot of CNN accelerators based on the CNN intrinsic parallelism are proposed. However, we observed that there is a big mismatch between the parallel types supported by computing engine and the dominant parallel types of CNN workloads. This mismatch seriously degrades resource utilization of existing accelerators. In this paper, we propose a flexible dataflow architecture (FlexFlow) that can leverage the complementary effects among feature map, neuron, and synapse parallelism to mitigate the mismatch. We evaluated our design with six typical practical workloads, it acquires 2-10x performance speedup and 2.5-10x power efficiency improvement compared with three state-of-the-art accelerator architectures. Meanwhile, FlexFlow is highly scalable with growing computing engine scale.**

1. Introduction

Convolutional Neural Network (CNN) is a promising algorithm for recognition, mining, and synthesis appications [13]. Recent years, more and more applications based on CNN models are spotlighted, such as handwriting recognition [18], face detection [9], [12], video surveillance [27], nature language analysis [10], [11], [24], and intelligent transportation [14], [28], [30].

CNN models hold intensive fine-grained parallelism in feature map (FP), neuron (NP), and synapse (SP) levels. These parallelisms provide attractive opportunities to accelerate the computations of CNN. Many researchers have proposed various dedicated large scale CNN accelerators [1], [2], [3], [6], [7]. The performance of these accelerators easily beat the general CPUs and GPUs.

However, the prior designs are still far from the perfect solution for CNN. Most of the prior solutions fall into one of the three representative architectures: Systolic [1], [7], 2D-Mapping [6], and Tiling [2], [3]. Due to the limitations of specific dataflow of each representative, most of existing accelerators can only support a very specific parallelism. For example, Systolic architectures [1], [7] can only exploit synapse parallelism, 2D-Mapping architectures [6] neuron parallelism, and Tiling architectures [2], [3] feature map parallelism.

However, given a practical CNN, the dominant parallel type varies dramatically with the changes in number of input feature maps, the number of output feature maps, the size of output feature map, and the size of kernel. Consequently, these accelerators fail to deliver the nominal performance as the design specification promised in many cases. Figure 1 lists the achievable performance normalized to nominal of three representative architectures running a typical workload LeNet-5 [15]. It's not uncommon that merely 10% GOPS (giga-operation per second) is achieved in practice. The key reason is the data path of the processing engines are designed with a specific "dominant parallelism" in mind. Unsurprisingly, they cannot support the various types of fine-grained parallelism simultaneously.

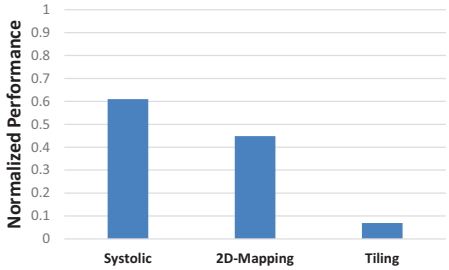


Figure 1. Nominal vs. Achievable Performance

To close the gap between achievable and nominal performance, an ideal architecture should be able to reap all types of the fine-grained parallelism. The principal challenge is how to support all types of data paths in each type of parallelism, without excessively complicated controls. We find the dataflows in synapse, neuron, and feature map parallelism differ sharply from each other. We abstract the combination of the three parallelism into eight dataflow styles (Section 2), and find the state-of-the-arts only cover three out of the eight styles. More importantly, we propose complementary parallelism principle to leverage the complementary between parallelisms to boost the resource utilization (Section 4), thereby mitigating the performance gap.

In particular, we propose a new dataflow (FlexFlow) architecture. It natively supports all parallelisms. Moreover, it can be adaptive to multiple mixtures of parallelisms to serve different layers efficiently. Specially, this paper makes the following contributions.

1. *Based on the analysis for prior designs, we propose complementary parallelism principle reaping multiple types of parallelism to improve the computing resource utilization.*
2. *We propose multiple strategies (RS, RA, IADP, IPDR) to optimize the dataflow from on-chip buffers to PEs, which can timely provide data for each mixture of parallelisms.*
3. *We implement a practical accelerator in* $65nm$ *process, and make comprehensive evaluations from computing resource utilization, performance, power, energy, and scalability.*

The rest of this paper is organized as follows. Section 2 introduces some CNN basics. In section 3, we will analyze three typical

Corresponding Authors: *Guihai Yan, Yinhe Han, and Xiaowei Li, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences; Phone:+8610-62600719; E-mail:*{*yan, yinhes, lxw*}*@ict.ac.cn.*

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.29

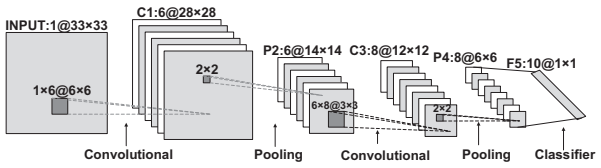


Figure 2. A Typical CNN (C: Convolutional, P: Pooling, F: Classifier)

computing architectures. In Section 4, the FlexFlow architecture is introduced. Section 5 describes the parallelism determination for each layer. We present experiments in Section 6.

2. Preliminary of CNN

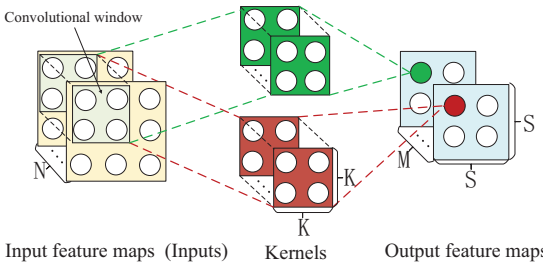
2.1. CNN Basics

A typical CNN consists of three types of operation layers: 1) convolutional layer (CONV), 2) pooling layer (POOL), and 3) classifier layer (FC), as exemplified in Figure 2. And six practical CNNs are listed in Table 1 (just one of two identical layer-parts of AlexNet is listed, otherwise specified). A CONV layer reads input feature maps or images from previous layer or data input ports, transforms these inputs using several groups of kernels into a series of output feature maps. Each CONV layer is usually followed by a POOL layer that subsamples the features from previous CONV layer. Finally, the FC layer generates the probability of each category implied in the initial input data.

Generally, there are four types of neural network data objects: feature map, neuron, kernel, and synapse. A more quantitative CNN specification can be described as follows. $\mathcal{I}^{(n)}$ denotes the n-th input feature map, $\mathcal{O}^{(m)}$ denotes the m-th output feature map, $\mathcal{K}^{(m,n)}$ is the kernel between n-th input and m-th output feature maps (all features map and kernels are two-dimensional, otherwise specified), $\mathcal{I}^{(n)}_{(r,c)}$ denotes the neuron at location (r, c) of $\mathcal{I}^{(n)}$, $\mathcal{O}^{(m)}_{(r,c)}$ denotes the neuron at location (r, c) of $\mathcal{O}^{(m)}$, and $\mathcal{K}^{(m,n)}_{(i,j)}$ is the synapse value at position (i, j) of $\mathcal{K}^{(m,n)}$.

TABLE 1. PRACTICAL CNNS (C_x: CONVOLUTIONAL LAYER)

Workloads	Layers	Kernels	Layer Size
PV [28]	Input		1@50 × 50
	C1	1 × 8@6 × 6	8@45 × 45
	C3	8 × 12@3 × 3	12@20 × 20
	C5	12 × 16@3 × 3	16@8 × 8
	C6	16 × 10@3 × 3	10@6 × 6
	C7	10 × 6@3 × 3	6@4 × 4
FR [5]	Input		1@32 × 32
	C1	1 × 4@5 × 5	4@28 × 28
	C3	4 × 16@4 × 4	16@10 × 10
LeNet-5 [16]	Input		1@32 × 32
	C1	1 × 6@5 × 5	6@28 × 28
	C3	6 × 16@5 × 5	16@10 × 10
HG [17]	Input		1@28 × 28
	C1	1 × 6@5 × 5	6@24 × 24
	C3	6 × 12@4 × 4	12@8 × 8
AlexNet [13]	Input		3@224 × 224
	C1	3 × 48@11 × 11	48@55 × 55
	C3	48 × 128@5 × 5	128@27 × 27
	C5	256 × 192@3 × 3	192@13 × 13
	C6	192 × 192@3 × 3	192@13 × 13
	C7	192 × 128@3 × 3	128@13 × 13
VGG-11 [25]	Input		3@224 × 224
	C1	3 × 64@3 × 3	64@222 × 222
	C3	64 × 128@3 × 3	128@109 × 109
	C5	128 × 256@3 × 3	256@52 × 52
	C6	256 × 256@3 × 3	256@50 × 50
	C8	256 × 512@3 × 3	512@23 × 23
	C9	512 × 512@3 × 3	128@21 × 21
	C11	512 × 512@3 × 3	512@8 × 8
	C12	512 × 512@3 × 3	512@6 × 6




```
for(m=0;m<M;m++){      //output feature maps
 for(n=0;n<N;n++){     //input feature maps
  for(r=0;r<S;r++){    //output neuron rows in a feature map
   for(c=0;c<S;c++){   //output neuron columns in a feature map
    for(i=0;i<K;i++){ //kernel rows operations
     for(j=0;j<K;j++){//kernel columns operations
     L:                //loop body (multiplication-accumulation)
       O(m)(r,c) += K(m,n)(i,j) × I(n)(r+i,c+j) ;
} } } } } }
```

Figure 3. The Logical Graph and Pseudo Code for CONV Operation

It's well-known that CONV layers are highly computation-intensive. For a typical CNN application, CONV layers take up more than 90% of the computation volume in both inference and training procedures. Fortunately, CONV computation exhibits intensive parallelism in feature map, neuron, and synapse levels (kernel level parallelism is intrinsically included in feature map and neuron levels).

A CONV layer can be characterized by four object-related parameters: M, the number of output feature maps, N, the number of input feature maps, S, the output feature map size (the number of neurons), K, the kernel size (the number of synapses), as shown in Figure 3, where squares represent feature maps or kernels, circles denote neurons in feature maps or synapses in kernels. The computation can be described in a deep nested-loop as illustrated in lower part of Figure 3. Feature map related loops m and n index the output and input feature maps respectively. Additionally, neuron-related loops r and c index neurons of each feature map. Finally, synapse related loops i and j index synapses of each kernel.

2.2. Parallelism in CNN Computing

To acquire high throughput, the above deep nested-loop is usually unrolled and mapped to a parallel hardware substrate. The operations of a CONV layer can be unrolled as Figure 4 shown, where $\mathcal{T} =< T_m, T_n, T_r, T_c, T_i, T_j >$ is a set of unrolling factors to quantify the parallel degree for the six loops. The loops marked by the inner-box are unrolled to operate in parallel, and the operations in outer-box execute sequentially.

There are three types of parallelism according to different unrolling strategies for related loops.

- Feature map Parallelism (FP), the feature map related loops m and n are unrolled with factors $< T_m, T_n >$. T_m output feature maps and T_n input feature maps are processed at a time.
- Neuron Parallelism (NP), the neuron related loops r and c are unrolled with factors $< T_r, T_c >$. $T_r \times T_c$ neurons of one output feature map are processed at a time.
- Synapse Parallelism (SP), the synapse related loops i and j are unrolled with factors $< T_i, T_j >$. $T_i \times T_j$ synapses of one kernel are computed at a time.

Accordingly, the computing architectures can be divided into eight typical processing styles, and each reaps a certain combination of the three types of parallelism. For example, an architecture

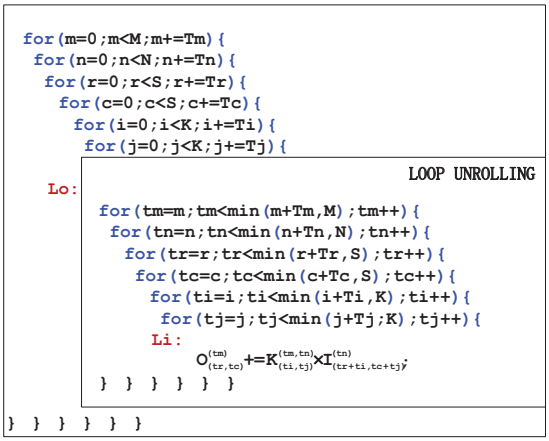


Figure 4. Loop Unrolling for One CONV layer

may handle single input feature map and single output feature map ($T_m = 1$ and $T_n = 1$), one neuron ($T_r = 1$ and $T_c = 1$) of each output feature map, but multiple synapses ($T_i > 1$ or $T_j > 1$) of each kernel at a time. We call this processing style as Single Feature map, Single Neuron, Multiple Synapses (SFSNMS). Similarly, we can build other seven possible processing styles: SFSNSS, SFMNSS, SFMNMS, MFSNSS, MFSNMS, MFMNSS, and MFMNMS.

An ideal architecture should be able to support all of the processing styles to accommodate different CNN models. However, it's not easy to fully exploit these parallelism in reality largely because of the distinct dataflow in different processing styles. The dataflow describes how the operands are dispatched, gathered, and updated among many distributed processing elements (PE). We find that most of prior architectures belong to SFSNMS, SFMNSS, and MFSNSS styles, which only support single parallel type (SP, NP, or FP) with a fixed parallel degree. We list some representative architectures in Table 2. The corresponding representative architectures are Systolic array, 2D-Mapping array, and Tiling array, respectively. In the next section, we analyze the dataflow of these architectures and discuss the advantages and disadvantages of each architecture. These analyses motivate our versatile FlexFlow architecture.

TABLE 2. THREE TYPICAL COMPUTING STYLES

Styles	Architectures
SFSNMS	DC-CNN [1], CNP [8], Neuflow [7]
SFMNSS	Diannao [2], DaDiannao [3], FPGA15 [29]
MFSNSS	ShiDiannao [6]

3. Architecture Comparison

In this section, we analyze three typical architectures (Systolic, 2D-Mapping, Tiling) in terms of parallelism, dataflow, data sharing, and specific limitation. Otherwise specified, we take $C3$ layer in Figure 2 to exemplify the working principle of these architectures.

3.1. SFSNMS: Systolic

Parallelism. This architecture reaps synapse parallelism (SP), and two synapse related loops i and j are thoroughly unrolled, as Figure 5(a1) shown. At each clock cycle, it handles single input feature map and single output feature map (i.e. Single Feature map), one neuron of each output feature map (i.e. Single Neuron), and all synapses (i.e. Multiple Synapses) of each kernel.

DataFlow. One systolic array with $K \times K$ ($K = 3$) PEs is shown in Figure 5(a2). PEs within each row are cascaded from left to right, and neighbor PE rows are linked through on-chip FIFOs. Each PE consists of one multiplier, one adder, and two registers (one holds a constant synapse, and another temporarily stores partial result). The whole array is a deep pipeline, and the depth of which roughly equals to the input feature map width times the kernel size (K). $K \times K$ output neurons are mapped to $K \times K$ PEs (pipeline stages), and shifted to next stages in next clock cycle. At every clock cycle, one input neuron is broadcasted into all PEs, then locally multiplied by the synapse stored in register and accumulated to one output neuron in each PE. After that, the output neuron is shifted to the next neighbor PE or switched to next PE row via on-chip FIFOs whose size equals to the input feature map width minus the kernel width (here is $12-3=9$). Eventually, one output neuron is integrated and pumped to external memory at the last stage of the pipeline. The number of clock cycles to complete one output feature map equals to the output feature map size, plus the pipeline filing time.

To further clarify the dataflow, we show a dataflow snapshot at loops ($m = 0, n = 1, r = 3, c = 1$) in Figure 5(a2). Firstly, neuron $\mathcal{I}^{(1)}_{(5,4)}$ is broadcasted into each PE, multiplied by synapses $\mathcal{K}^{(0,1)}_{(0,0)}$, $\mathcal{K}^{(0,1)}_{(0,1)}$,...,$\mathcal{K}^{(0,1)}_{(2,2)}$ and accumulated to output neurons $\mathcal{O}^{(0)}_{(5,4)}$, $\mathcal{O}^{(0)}_{(5,3)}$, ..., $\mathcal{O}^{(0)}_{(3,2)}$. Next, these output neurons are shifted to right neighbor PEs or FIFOs. Eventually, multiple partial results of neuron $\mathcal{O}^{(0)}_{(3,1)}$ are integrated and pumped to external memory in the final adder.

Data Sharing. In this computing style, multiple adjacent output neurons are mapped to different pipeline stages to spatially share input neurons and temporally reuse the synapses stored in local registers, till the same input feature map processing accomplished.

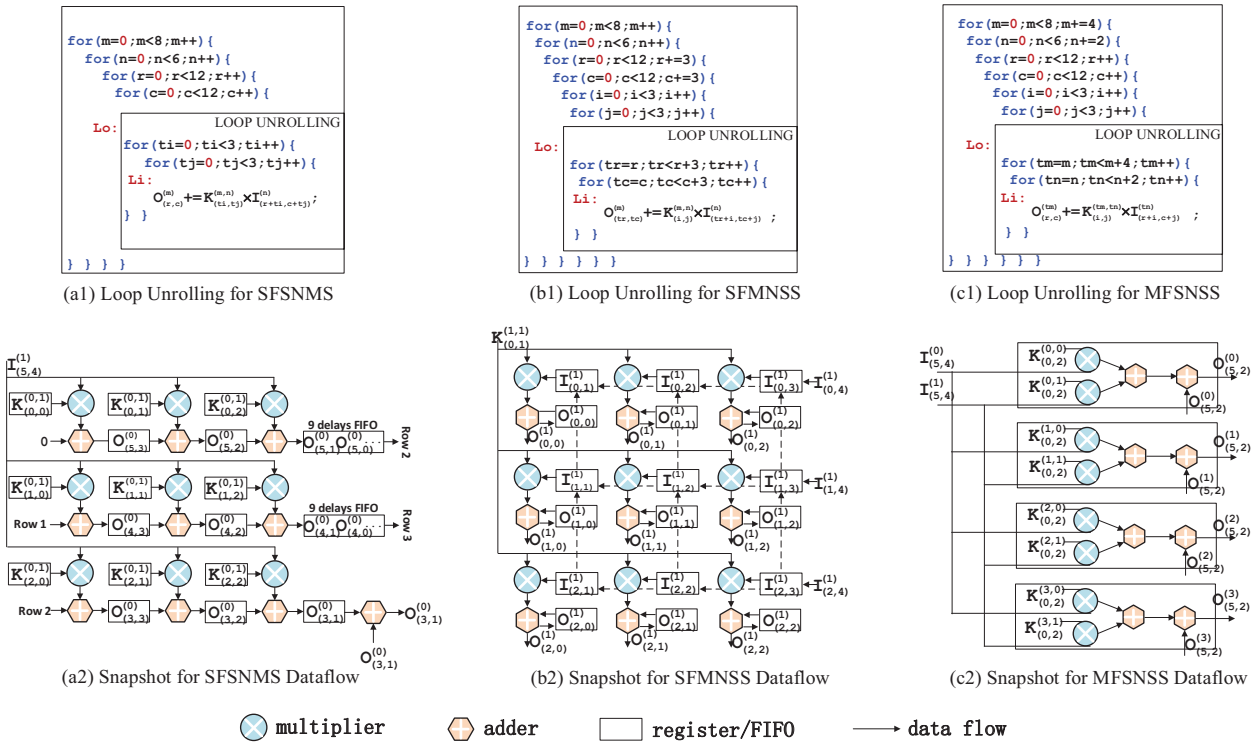
Specific Limitation. This computing style can only support synapse parallelism (SP), but not support the other two types of parallelism (NP and FP).

3.2. SFMNSS: 2D-Mapping

Parallelism. It reaps neuron parallelism (NP), two neuron related loops r and c are unrolled ($T_r = 3, T_c = 3$), as Figure 5(b1) shown. At each clock cycle, it handles one single input feature map and one single output feature map (i.e. Single Feature map), $T_r \times T_c$ neurons (i.e. Multiple Neurons) of each output feature map, and one single synapse (i.e. Single Synapse) of each kernel.

DataFlow. A 2D-Mapping array with $T_r \times T_c$ PEs is shown in Figure 5(b2). Every PE is connected to its four neighbors through internal links. Each PE includes one multiplier, one adder, and two FIFOs temporarily storing input neurons for neighbor PEs reusing. Initially, $T_r \times T_c$ input neurons and one synapse are broadcasted to each PE. After that, at each clock cycle, T_r or T_c neurons are received and shifted from right to left or down to up across PEs, and one synapse is distributed to all PEs. Then neuron is locally multiplied by synapse and accumulated to one output neuron in each PE. Each PE keeps calculating part of one output neuron, till all partial results are accomplished. Then, the PE switches to another output neuron. It takes $K \times K$ clock cycles to complete $T_r \times T_c$ neurons.

The dataflow snapshot at loops ($m = 1, n = 1, r = 0, c = 0, i = 0, j = 1$) is shown in Figure 5(b2). Firstly, neurons $\mathcal{I}^{(1)}_{(0,4)}$, $\mathcal{I}^{(1)}_{(1,4)}$, and $\mathcal{I}^{(1)}_{(2,4)}$ are received and other neurons stored in internal FIFOs are shifted from right to left PEs, and synapse $\mathcal{K}^{(1,1)}_{(0,1)}$ is distributed to all PEs. Then neurons are multiplied by synapse



(a1) Loop Unrolling for SFSNMS (b1) Loop Unrolling for SFMNSS (c1) Loop Unrolling for MFSNSS

(a2) Snapshot for SFSNMS Dataflow (b2) Snapshot for SFMNSS Dataflow (c2) Snapshot for MFSNSS Dataflow

Figure 5. Typical Architectures

$\mathcal{K}^{(1,1)}_{(0,1)}$ and are locally accumulated with the partial results stored in local registers, yielding output neurons $\mathcal{O}^{(1)}_{(0,0)}$, $\mathcal{O}^{(1)}_{(0,1)}$,..., $\mathcal{O}^{(1)}_{(2,2)}$.

Data Sharing. In this computing style, multiple adjacent output neurons are mapped to the adjacent PEs, and the paths between neighbor PEs and the FIFOs give opportunity to temporally share input neurons across different PEs (output neurons). Meanwhile, the synapses are spatially shared with all PEs.

Specific Limitation. This computing style can only support neuron parallelism (NP), but not support the other two types of parallelism (SP and FP).

3.3. MFSNSS: Tiling

Parallelism. It reaps feature map parallelism (FP), and two feature map related loops m and n are unrolled ($T_m = 4, T_n = 2$), as Figure 5(c1) shown. At each clock cycle, it handles T_n input feature maps and T_m output feature maps (i.e. Multiple Feature maps), one neuron of each output feature map (i.e. Single Neuron), and one single synapse (i.e. Single Synapse) of each kernel.

DataFlow. One tiling array with T_m PEs is depicted in Figure 5(c2). Each PE consists of T_n multipliers and T_n adders ($T_n - 1$ adders are organized into one adder tree, and the rest one is used to accumulate partial results). At each clock cycle, T_n neurons and $T_m \times T_n$ synapses are loaded. Then T_n neurons are multiplied by $T_n \times T_m$ synapses, and the results of T_n multipliers are locally summed up to one output neuron in each PE. Each PE keeps calculating a single output neuron till all partial results completed, and then switches to another neuron. The number of clock cycles to complete T_m neurons equals to the kernel size $K \times K$.

The dataflow snapshot at loops ($m = 0, n = 0, r = 5, c = 2, i = 0, j = 2$) is shown in Figure 5(c2). Neurons $\mathcal{I}^{(0)}_{(5,4)}$, $\mathcal{I}^{(1)}_{(5,4)}$ and synapses $\mathcal{K}^{(0,0)}_{(0,2)}$, $\mathcal{K}^{(0,1)}_{(0,2)}$, ..., $\mathcal{K}^{(3,1)}_{(0,2)}$ are loaded and distributed to different PEs. Then two neurons are multiplied by the eight synapses, and the results are accumulated into four output neurons $\mathcal{O}^{(0)}_{(5,2)}$, ..., $\mathcal{O}^{(3)}_{(5,2)}$ in each PE.

Data Sharing. In this computing style, the links among PEs enable spatially share input neurons across different PEs (output neurons). But there is no local storage caching neurons and synapses, so it cannot support temporal neuron and synapse reusing. Compared with other styles, it acquires the poorest data sharing.

Specific Limitation. This computing style can only support feature map parallelism (FP), but not support the other two types of parallelism (SP and NP).

3.4. Limitations of Above Architectures

Rigid dataflow is the common characteristic for above architectures, which reap single type of parallelism (SP, NP, or FP) with fixed parallel degree to operate different CONV layers. Based on the above analysis, we ascribe the rigid dataflow to three aspects.

- **Fixed data direction.** The inflexible internal links among PEs provides directed dataflow between PEs, such as shifting input neurons right to left or down to up in 2D-Mapping, shifting output neurons left to right in Systolic.
- **Fixed data type.** The data in all PEs has same attribute, such as output neurons in different PEs belong to same feature map (Systolic, 2D-Mapping) or at same locations of multiple feature maps (Tiling).
- **Fixed data stride.** The operations for the operands (neurons or synapses) of all PEs are carried out synchronously. For example, in 2D-Mapping architectures, the input neurons are simultaneously fed to each PE, processed concurrently, and propagated to next PEs at a time.

In reality, the dominant parallel type changes greatly across different CONV layers. For existing architectures, the computing engine parameterized for one layer can get good performance, but the performance may drop sharply for other layers. We take $C1$ and $C3$ layers depicted in Figure 2 as examples to highlight the limitations of rigid architectures.

To carry out $C1$ layer efficiently, the architectures should be parameterized as following: $< T_m = 1, T_n = 1, T_r = 1, T_c = 1, T_i = 6, T_j = 6 >$ (Systolic), $< T_m = 1, T_n = 1, T_r = 28, T_c = 28, T_i = 1, T_j = 1 >$ (2D-Mapping), $< T_m = 6, T_n = 1, T_r = 1, T_c = 1, T_i = 1, T_j = 1 >$ (Tiling), the hardware utilization of three architecture is 100%. However, the dominant parallelism for $C3$ is different from $C1$ layer, the same architecture configuration as $C1$ will acquire poor hardware utilization: 25% (Systolic), 66.6% (2D-Mapping), 18.4% (Tiling). It is not a coincidence, and we further examine other four practical workloads listed in Table 1. The hardware utilization is listed in Table 3, where "C3 on C1-opt" means the hardware utilization when running C3 on the hardware optimized for C1. The utilization of 'C1 on C1-opt' is normalized to 100%. As the results shown in Table 3, only a few cases can achieve more than 50% utilization, and some even drop below 10%.

TABLE 3. HARDWARE UTILIZATION FOR THREE TYPICAL ARCHITECTURES ACROSS FOUR PRACTICAL WORKLOADS

Workloads	Optimization	Resource Utilization		
		Systolic	2D-Map.	Tiling
PV [28]	C3 on C1-opt	25	19	75
	C1 on C3-opt	100	56	8.3
FR [5]	C3 on C1-opt	80	12.7	100
	C1 on C3-opt	39	87	6.2
LeNet-5 [16]	C3 on C1-opt	100	12.7	88
	C1 on C3-opt	100	87	6.2
HG [17]	C3 on C1-opt	80	100	11
	C1 on C3-opt	39	100	8.3

4. FlexFlow: Flexible DataFlow Architecture

To overcome the limitations discussed above, we first modify the PE micro-architecture and the interconnection between PEs to support FP, NP, and SP simultaneously. Moreover, we exploit some complementary effects between those parallelisms by dynamically mapping different mixtures of parallelisms to occupy the computing engine. The key challenge is to design a flexible dataflow to timely provide data for each type of parallelism enabled.

Before delving into details, we first give an overview of proposed FlexFlow architecture, as shown in Figure 6. There are four key components: a convolutional unit, a pooling unit, three on-chip buffers (two neuron buffers and one kernel buffer), and an instruction decoder. The two computing units, a matrix of convolutional unit and an array of pooling unit, are referred to 2D convolutional unit and 1D pooling unit, respectively. The 2D convolutional unit comprises a group of PEs with a mesh-like organization. A PE consists of a multiplier, an adder, a neuron local store, a kernel local store, and a controller, as Figure 7(a) shown. The pooling unit is a series of lightweight ALUs, subsampling the immediate convolution results to reduce data transmission.

Without losing generality, we always use the example to demonstrate the engine working principle: a small scale 4×4-PE convolutional unit processing two CONV layers $C1$ ($M = 2$, $N = 1$, $S = 8$, $K = 4$) and $C2$ ($M = 2$, $N = 2$, $S = 4$, $K = 2$).

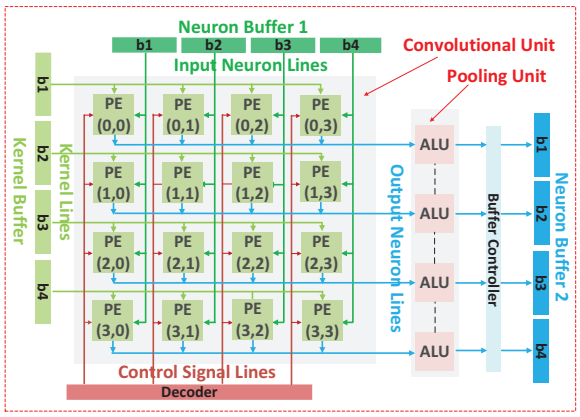
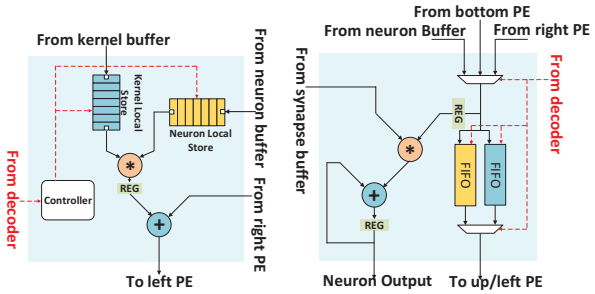


Figure 6. FlexFlow Architecture

4.1. FlexFlow's PE vs. Conventional PE

To support more types of parallelism, firstly, we should break the inflexible data path between PEs. We modified the PE microarchitecture and removed most of the fussy interconnections between PEs. Only the adders within each PE row are connected to form an adder tree, each PE row can complete one convolution and serve to one output neuron. The interfaces for exchanging operands with neighbor PEs are also removed. Compared with other designs, our design has more flexibility to cover multiple parallel types. For comparison, the PE of 2D-Mapping architecture is depicted in Figure 7 (b), the input neuron of this PE can only come from its left neighbor or down neighbor PEs, and consecutive neurons must belong to adjacent convolutional windows. Moreover, these neurons are shifted to left or up PEs sequentially in next few clock cycles since they are buffered in FIFOs. For PEs in FlexFlow (Figure 7 (a)), by contrast, operands are directly derived from on-chip buffers through vertical and horizontal buses to each PE, and buffered in randomly accessed local storages.



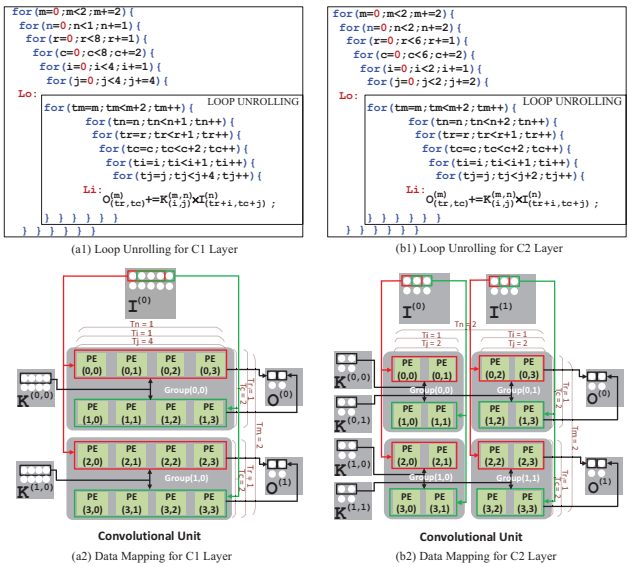
(a) (b)

Figure 7. PE in FlexFlow (a) vs. PE in 2D-Mapping Architecture (b)

4.2. Complementary Parallelism

By eliminating the dependency between adjacent PEs, the convolutional unit can support the comprehensive MFMNMS processing style which readily supports three types of parallelism:

- Neuron Parallelism (NP) is realized by allocating the N PE rows to N output neurons which belong to the same output feature map.
- Feature map Parallelism (FP) is realized by allocating N PE rows to N output neurons, but these neurons belong to different output feature maps, or allocating a PE row to N input neurons belonging to different input feature maps.



(a1) Loop Unrolling for C1 Layer (b1) Loop Unrolling for C2 Layer

(a2) Data Mapping for C1 Layer (b2) Data Mapping for C2 Layer

Figure 8. Complementary Parallelism

- Synapse Parallelism (SP) is realized by allocating a PE row to the input neurons of one convolutional window of a input feature map.

With the supports of these parallelism, we can dynamically enable a mix of FP and NP, or FP and SP, to exploit some kind of complementary effects in maximizing PE utilization. When a single type of parallelism cannot make high hardware utilization, multiple mixed parallelisms can complement each other to occupy more computing resources. For example, NP (output neurons belonging to the same output feature map) and FP (output neurons belonging to different feature maps) can complement each other to occupy the PE rows as many as possible, or "Inter-Row complementary effect". Meanwhile, SP (input neurons belonging to the same input feature maps) and FP (input neurons belonging to different feature maps) can complement each other to occupy the PEs within each PE row as many as possible, or "Intra-Row complementary effect".

The complementary effect is better explained with an example, as shown in Figure 8. For $C1$ layer, we use high SP ($T_j = 4$) to complement low FP ($T_n = 1$) occupying the intra-row PEs. And medium NP ($T_c = 2$) and FP ($T_m = 2$) are combined to occupy all PE rows. Overall the unrolling mode for $C1$ should be $< T_m = 2, T_r = 1, T_c = 2, T_n = 1, T_i = 1, T_j = 4 >$. Similarly, for $C2$ layer, the unrolling mode was configured to $< T_m = 2, T_r = 1, T_c = 2, T_n = 2, T_i = 1, T_j = 2 >$. By doing so, the PEs for both C1 and C2 are fully utilized.

Due to each mix of parallelism with a specific combination of inputs and outputs, the premise of enjoying the complementary effects is the flexible dataflow with high data "routability". We propose a hierarchical dataflow with low control overhead, as Figure 9 shown. The distribution layer can be viewed as an interconnection structure, which routes the data from on-chip buffer to PEs. The whole dataflow is divided into three sub-flows: DataFlow1, distribution layer to local stores in each PE, DataFlow2, from local store to operator (multiplier and adder), and DataFlow3, from neuron and kernel buffers to distribution layer. Next, we present the key optimizations into these sub-flows.

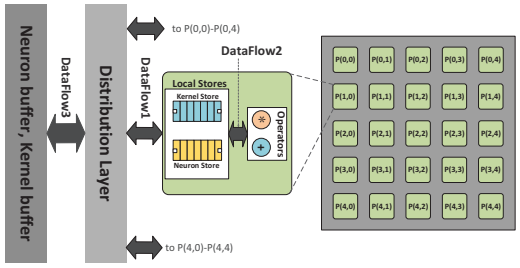


Figure 9. DataFlow

4.3. DataFlow1: Distribution Layer to Local Store

DataFlow1 is responsible for routing the data from the distribution layer to PEs. The interconnections should be simple enough to avoid the fixed data direction limitation (described in Section 3.4). To simplify the interconnection, the key is to minimize the volume of data repeatedly transmitted. We use horizontal (kernels) and vertical (neurons) common data buses (CDB) to broadcast data (neuron and kernel lines in Figure 6). CDBs are realized as simple, pipelined, data-only buses that do not dictate overhead for address decoding, or complex control, hence scalable and easy to route in layout. Accordingly, we proposed two data transfer optimizations to CDB: Relax Alignment (RA) and Relax Synchronization (RS).

Relax Alignment (RA). Based on complementary parallelism mechanism, different PE rows are allocated to output neurons at the same or adjacent locations of multiple output feature maps, the input neurons and kernels for these output neurons are significantly overlapped to each other. As Figure 8(a2) shown, input neurons partially and kernels totally are overlapped between the first and the second PE rows. However, the overlapping is not aligned horizontally or vertically. Neurons $\mathcal{I}^{(0)}_{(0,0)}$, $\mathcal{I}^{(0)}_{(0,1)}$, $\mathcal{I}^{(0)}_{(0,2)}$, $\mathcal{I}^{(0)}_{(0,3)}$ are assigned to PE(0,0), PE(0,1), PE(0,2), and PE(0,3), respectively. Neurons $\mathcal{I}^{(0)}_{(0,1)}$, $\mathcal{I}^{(0)}_{(0,2)}$, $\mathcal{I}^{(0)}_{(0,3)}$, $\mathcal{I}^{(0)}_{(0,4)}$ are assigned to PE(1,0), PE(1,1), PE(1,2), and PE(1,3), respectively. Obviously, the pairs of PE(0,1) and PE(1,0), PE(0,2) and PE(1,1), PE(0,3) and PE(1,2) receive identical neurons. It is not easy to utilize the data overlapping since PEs in each pair are not locate in the same column or the same row.

Relax alignment (RA) can exploit the non-aligned neurons by reordering the synapses. To better explain the RA, we show a snapshot of the data placement and accesses for PE Row0 and Row1 of $C1$ at clock cycle t and $t+1$, as Figure 10 shown, where dark colored arrows point to the data accessed in cycle t, and light color arrows point to data accessed in cycle $t+1$. The neurons can be assigned to the two PE rows in alignment way, i.e. forwarding identical neurons to PEs within one column, such as neuron $\mathcal{I}^{(0)}_{(0,1)}$ in PE(0,1) and PE(1,1). Then, at clock t, we can compute output neuron $\mathcal{O}^{(0)}_{(0,0)}$ by accessing neurons $\mathcal{I}^{(0)}_{(0,4)}$, $\mathcal{I}^{(0)}_{(0,1)}$, $\mathcal{I}^{(0)}_{(0,2)}$, $\mathcal{I}^{(0)}_{(0,3)}$ and synapses $\mathcal{K}^{(0,0)}_{(0,3)}$, $\mathcal{K}^{(0,0)}_{(0,0)}$, $\mathcal{K}^{(0,0)}_{(0,1)}$, $\mathcal{K}^{(0,0)}_{(0,2)}$ in the second PE row. In this way, the neurons $\mathcal{I}^{(0)}_{(0,1)}$, $\mathcal{I}^{(0)}_{(0,2)}$, $\mathcal{I}^{(0)}_{(0,3)}$ overlapped across the two PEs are exploited by reordering the synapses. To simplify the interconnections, the synapses of whole kernel are replicated to each PE, which also benefits the reusability of synapses.

Relax Synchronization (RS). Relax synchronization can minimize the repeated data transmission by pre-loading some data for PEs. Unlike prior architectures with fixed data stride limitation described in Section 3.4, in our design, the data simultaneously fed to all PEs can be processed asynchronously. For example, neuron $\mathcal{I}^{(0)}_{(0,4)}$ is broadcasted to PE(0,0) and PE(1,0) at a time. But it is

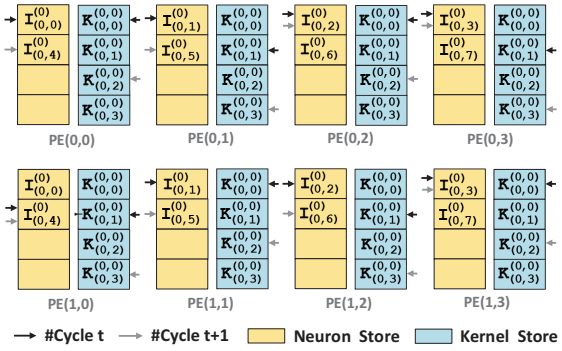


Figure 10. Local Store Access

processed at cycle t and $t+1$ in PE(1,0) and PE(0,0), respectively. In this way, the neurons can be preloaded in one PE along with the PEs located in the same column. Overall, RA and RS can drastically reduce data volume.

With the RA and RS mechanism, the data transmission becomes very regular. For the first and second PE rows of $C1$, the input neurons are divided to each PE column. Input neuron $\mathcal{I}^{(0)}_{(r,c)}$ is forwarded to PE(1:2, c mod 4). The synapses are broadcasted to all PEs of the two PE rows. Synapse $\mathcal{K}^{(0,0)}_{i,j}$ is forwarded to PE(0:1, 0:3). The output neurons are divided into each PE row. Output neuron $\mathcal{O}^{(0)}_{(r,c)}$ is mapped to PE Row(c mod 2). Moreover, the complementary parallelism principle logically divides the PE array into $T_m \times T_n$ groups, and each group includes $(T_i \times T_j) \times (T_r \times T_c)$ PEs. As shown in Figure 8(a2), Group(0,0) includes PE(0,0), ..., PE(1,3), and Group(1,0) includes PE(2,0), ..., PE(3,3). Each logical group has the identical data transmission format like Group(0,0) of $C1$. Generally, input feature map $\mathcal{I}^{(n)}$ is assigned to Group(:, n mod T_{pn}), and neuron $\mathcal{I}^{(n)}_{(r,c)}$ is assigned to PE(:, ((r mod T_{pi}) $\times$ T_{pj} + c mod T_{pj})) within one group (where notation ":" means "all rows"). Kernel $\mathcal{K}^{(m,n)}$ is assigned to Group(m mod T_{pm}, n mod T_{pn}), and synapse $\mathcal{K}^{(m,n)}_{(i,j)}$ is broadcasted to all PEs within one group. Also, Output neuron $\mathcal{O}^{(m)}_{(r,c)}$ is mapped to PE Row((m mod T_m)$\times T_r \times T_c +$ (r mod T_r)$\times T_c + c$ mod T_c)). Overall input neurons has column sharing characteristic, and synapse has block (one logical group) sharing characteristic.

4.4. DataFlow2: Local Store to Operator

Each PE has a neuron local store and a kernel local store. DataFlow2 is responsible for fetching data from local stores to operators. The local stores are not updated like FIFO buffers, but with some random-access capability to maximize data reusability. The overhead, fortunately, can be tackled well without sophisticated controls.

The neuron and kernel data preloaded through vertical and horizontal CDB may be accessed multiple times before being updated. To illustrate how the data is accessed and reused, we also use the example in Figure 10. At cycle t, input neurons $\mathcal{I}^{(0)}_{(0,0)}$, ..., $\mathcal{I}^{(0)}_{(0,3)}$ and synapses $\mathcal{K}^{(0,0)}_{(0,0)}$, ..., $\mathcal{K}^{(0,0)}_{(0,3)}$ are accessed to compute neuron $\mathcal{O}^{(0)}_{(0,0)}$ in the first PE row, and neuron $\mathcal{O}^{(0)}_{(0,1)}$ is computed in the second PE row. Next cycle, neurons $\mathcal{I}^{(0)}_{(0,2)}$ and $\mathcal{I}^{(0)}_{(0,3)}$ are reused in PE(0,2) and PE(0,3) to compute neuron $\mathcal{O}^{(0)}_{(0,2)}$. And neurons $\mathcal{I}^{(0)}_{(0,3)}$ and $\mathcal{I}^{(0)}_{(0,4)}$ are reused in PE(1,3) and PE(1,0) to compute neuron $\mathcal{O}^{(0)}_{(0,3)}$. Similarly, in next cycle, synapses $\mathcal{K}^{(0,0)}_{(0,0)}$, ..., $\mathcal{K}^{(0,0)}_{(0,3)}$ will be reused in the first PE row.

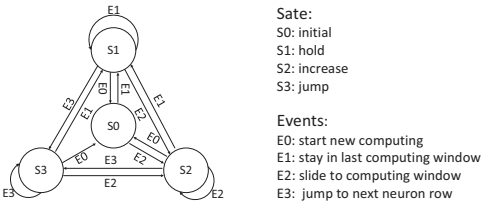


Figure 11. Control Finite State Machine

Obviously, the addressing mode for local stores is regular. For writing operation, the addressing mode is auto-increment. For reading operation, the addressing mode is regulated by four parameters: feature map size S, kernel size K, the counter step (T_c) and the current PE location within its group. Specifically, there are four reading addressing modes for the two local stores: $M0$, INIT, $M1$, INCR, $M2$, HOLD and $M3$, JUMP. INCR is increasing the address with a step. For the Group(0,0) of $C1$, the step for neuron local store is 1, and the step for kernel local store is 2. HOLD is holding the current address. JUMP is jumping to next neuron row.

The addressing modes for two local stores are governed by two identical four-state FSM (finite state machine), as shown in Figure 11. The FSM will jump to S0 when a new computation starts. Once one computing window (equaling to T_i) is completed, the FSM will jump to S2, and otherwise it stays in S1. The FSM will transit to S3 when one neuron row is completed.

4.5. DataFlow3: Neuron and Kernel Buffers to Distribution Layer

DataFlow3 is responsible for smooth data transmission between three D-banked buffers and distribution layer. We propose two optimizations to ease the data pressure of other sub-flows: 1) In-Advanced Data Placement (IADP), properly pre-layout data in on-chip buffers to take bank level parallelism. 2) In-Place Data Replication (IPDR), fullly utilizes the free bandwidth for data broadcasting to reduce internal connections.

In-Advanced Data Placement(IADP). To match the data format of vertical and horizontal CDBs , we need to arrange the data for each on-chip buffer in advance according to the logical grouping of PEs. The kernel buffer is shown in Figure 12(a). It is divided into T_m groups, each group further divided into T_r subgroups, and each sub-group holds T_c buffer banks. Each kernel is row-major concentrated in one area within a group (indicated in Figure 12(a) top-right bold arrows). Different kernels are stored across groups, indicated by the left bold arrows in Figure 12(a). Kernels loaded from external memory are directly stored in above format, then the reading controller can read one data from each group every clock cycle.

For neuron buffers, there are two types of IADP procedures, one is data loading from external memory to neuron buffer, and the other is reading and writing of intermediate data between two neuron buffers. Considering first procedure, one neuron buffer is shown in Figure 13(a), which is divided into T_n groups, each group is further divided into T_i subgroups, and each subgroup includes T_j buffer banks. Each input feature map is concentrated in one group, and each neuron row of a input feature map is concentrated in one subgroup (indicated in Figure 13(a) top-right bold arrows). Different feature maps are stored across groups indicated in Figure 13(a) left arrows. Neurons loaded from external memory are directly stored in above format, so D data can be read out from D buffer

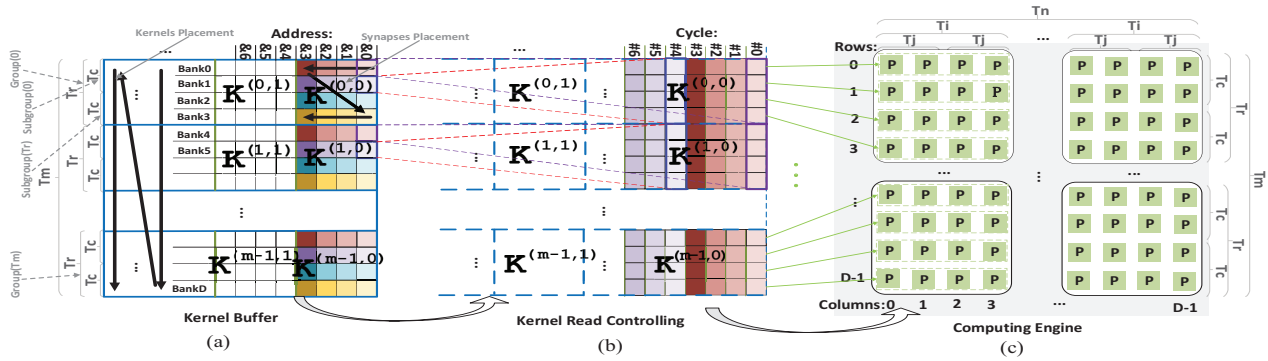


Figure 12. Kernel Buffer Data Placement and Data Transmission Pattern

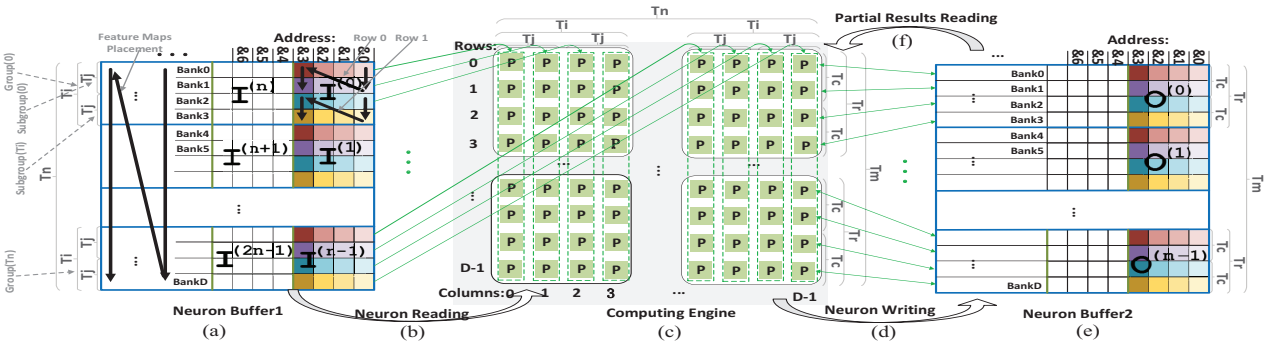


Figure 13. Neuron Buffer Data Placement and Data Transmission Pattern

banks and fed to corresponding PE columns through vertical buses (Figure 13(c)). To facilitate the mixtures of parallelisms switching and reduce data congestion between two layers, the results of one layer are always written to another neuron buffer with the form of next layer. It means that the factors $< T_m, T_r, T_c >$ of current layer always equal to factors $< T_n, T_i, T_j >$ of next layer. So another buffer is partitioned by factors $< T_m, T_r, T_c >$ (Figure 13(e)). Besides, due to the limitation of the size of computing engine, a complete convolution requires multiple computations, the data written back are partial results. In this situation, these data are needed to read back and accumulated to other partial results (Figure 13(f)).

In-Place Data Replication (IPDR). To support RA mechanism, kernel is broadcasted to all PEs within one logical group. The high reusability of kernels provides abundant free bandwidth for horizontal CDB. Consequently, we can broadcast some data out of the computing engine to simplify the internal interconnections. Free horizontal buses can be fully utilized to transmit identical data. As shown in Figure 12(b), every data read in reading controller is replicated $T_r \times T_c$ times. Then these data are fed to all PEs within one group through horizontal buses (Figure 12(c)). Otherwise specified, $T_r \times T_c$ is no larger than the PE scale D. And, the reading controller is very simple, which only needs to route data to up or down direction busses. So, the area and energy overhead is very low. There is no IPDR mechanism for neuron transmission since no group broadcasting for neuron data required.

5. Determining Parallelism

In this section, we describe how to determine the parallel type and degree for each CONV layer, i.e. to determine the unrolling factors $< T_m, T_n, T_r, T_c, T_i, T_j >$. Given a CONV layer, assume that the number of output feature maps is M, the number of input feature maps is N, the size of one output feature map is S, the size of one kernel is K, the kernel size of next CONV layer is K', and the pooling window size of next POOL is P. According to IADP, we should pre-layout data in on-chip buffers with the format of next CONV layer to avoid data congestion. Hence, the factors $< T_m, T_r, T_j >$ of current layer always equal to factors $< T_n, T_i, T_j >$ of next CONV layer. Besides, there is always a POOL layer between two CONV layers. So parameters T_r and T_c are limited by next CONV layer and POOL layer ($0 < T_r \leq P \times K'$, $0 < T_c \leq P \times K'$). If mapping the CONV layer to a convolutional unit with $D \times D$ PEs, the space of all feasible factors should meet the following constraints.

$$\begin{cases} 0 < T_m \leq M,\ 0 < T_n \leq N,\ 0 < T_i \leq K,\ 0 < T_j \leq K \\ 0 < T_r \leq P \times K',\ 0 < T_c \leq P \times K' \\ (T_n \times T_i \times T_j) \leq D,\ (T_m \times T_r \times T_c) \leq D \end{cases} \tag{1}$$

Given a CONV layer and a convolutional unit, we can get the computing resource utilization under any feasible unrolling factors.We use PE cycle to portray the resource utilization. The number of PE cycles for computation to the total number of PE cycles ratio, features the computing resource utilization. For simplicity, we compute PE row utilization (U_r) and PE column utilization (U_c) of the convolutional unit, which are calculated by Equation 2 and Equation 3. The total utilization (U_t) can be calculated by $U_r \times U_c$.

$$U_r = \frac{N \times K \times K}{\lceil \frac{N}{T_n} \rceil \times \lceil \frac{K}{T_i} \rceil \times \lceil \frac{K}{T_j} \rceil \times D} \tag{2}$$

$$U_c = \frac{M \times S \times S}{\lceil \frac{M}{T_m} \rceil \times \lceil \frac{S}{T_r} \rceil \times \lceil \frac{S}{T_c} \rceil \times D} \tag{3}$$

Our goal is to maximize U_r and U_c, subject to the Constraint 1. We compute the utilization for each set of candidate unrolling factors, and select the factors with maximal hardware utilization. We have developed a specialized compiler including a workload analyzer, which determines the unrolling factors for each layer and produces assemble language code to configure the FlexFlow. Given a 16×16 PEs convolutional unit, for four practical workloads (list in Table 1), the unrolling factors of each CONV layer are listed in Table 4.

TABLE 4. UNROLLING FACTORS FOR FOUR WORKLOADS

Workloads	Layers	T_m	T_n	T_r	T_c	T_i	T_j
PV [28]	C1	8	1	1	2	2	6
	C3	3	8	1	5	1	2
FR [5]	C1	4	1	1	4	3	15
	C3	16	4	1	1	1	4
LeNet-5 [16]	C1	3	1	1	5	3	5
	C3	16	3	1	1	1	5
HG [17]	C1	3	1	1	5	3	5
	C3	4	2	1	4	2	4

6. Evaluation

In this section, we evaluate FlexFlow architecture from computing resource utilization, performance, power, energy, and area. We compare our design with three typical architectures (Systolic, 2D-Mapping, Tiling) using six practical workloads. We also examine the scalability of FlexFlow in terms of resource utilization, power, and area with growing scales of computing engine.

6.1. Experimental Methodology

6.1.1. Baselines

To make fair comparison, we reimplement the following baselines with the same scale. All architectures use 16-bit fixed point data type.

Systolic. The unrolling factors are $< T_i = 6, T_j = 6 >$ to cover all workloads except AlexNet (with the unrolling factors $< T_i = 11, T_j = 11 >$). We use 7 identical 6×6 arrays to match the computing scale with other architectures. Different arrays work in a Tiling-like mode (DC-CNN [1]).

2D-Mapping. The unrolling factors are $< T_r = 16, T_c = 16 >$, which can process 256 output neurons at a time.

Tiling. The unrolling factors are $< T_m = 16, T_n = 16 >$, which can process 16 input features and 16 output feature maps at a time.

FlexFlow. We choose a 16×16-PE convolutional unit, in line with the computing scale of above architectures.

TABLE 5. PARAMETERS SETTING OF FOUR BASELINES

	FlexFlow	Others
Neuron store:	256B	-
Kernel store:	256B	-
Neuron buffer:	32KB	32KB
Kernel buffer:	32KB	32KB

All baselines are equipped with same size of on-chip memories as listed in Table 5. We implement all baselines in Synopsys design flow on TSMC 65nm technology: simulating with Synopsys Verilog Compile Simulator (VCS), synthesizing with Synopsys Design Compiler (DC), analyzing power with Synopsys PrimeTime (PT), and placing them with Synopsys IC Compiler (ICC).

6.1.2. Workloads

Table 1 lists six workloads selected from recent published papers. PV (pedestrains and vehicles recognition) [28] proposed a CNN model to recognite pedestrians and cars. FR (face recognition) [5] implemented a face recognition model. LeNet-5 [16], the most famous handwriting recognition model. HG (hand gesture recognition [17]) model is used to recognize hand gestures of human. AlexNet [13] and VGG [25] are two most famous CNN models for image classification.

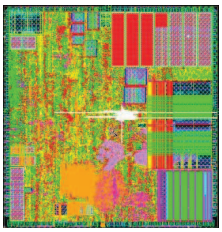


Figure 14. Layout of FlexFlow

6.1.3. Metrics

Besides the commonly used power, energy, and performance metrics, we also evaluate the data reusability and scalability.

Data Reusability. We use the volume of data transmission as the proxy of data reusability one computing architecture can provide. Given a CNN model, the architectures dictate less volume of data transmission can be extrapolated maintaining high data reusability.

Scalability. We use the sensitivity between the utilization and the scale of computing resources to characterize the scalability of one architecture. For example, the computing resource utilization ratio of a scalable architectures should be insensitive to the scale of computing engines. In addition, we also evaluate the scalability in area and power.

6.2. Experimental Results

6.2.1. Layout and Area

We layout the four baselines with the parameters listed in Table 5, and the layout of FlexFlow is shown in Figure 14. The total area of four baselines (Systolic, 2D-Mapping, Tiling, and FlexFlow) are $3.52mm^2$, $3.46mm^2$, $3.21mm^2$ and $3.89mm^2$. The area of FlexFlow is slightly larger than other baselines since the local stores equipped in each PE dictating part of area budget. Systolic and 2D-Mapping also equip local stores. Meanwhile, Tiling and 2D-Mapping are suffered fussy interconnection. By contrast, FlexFlow simplifies most of the interconnections between PEs. Moreover, with the growing scales of the computing engine, interconnecting wires will take up more share of the total chip area. By contrast, the simplified interconnections in FlexFlow will shown great merit in area scalability. This trend will be shown in Section 6.2.6.

6.2.2. Computing Resource Utilization

Computing resource utilization of each baseline is shown in Figure 15. FlexFlow obtains over 80% resource utilization across all workloads. Retaining flexible dataflows which reaping mixture of parallelisms directly contribute to the superior resource utilization. By contrast, the other baselines only acquire less than 60% (most of them less than 40%) resource utilization. Moreover, the computing resource utilization of the baselines is also volatile across different workloads. This result is not a coincidence. For Systolic, the kernel sizes (3, 4) for workloads PV, FR, AlexNet

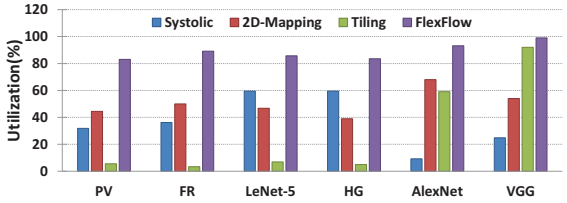


Figure 15. Computing Resource Utilization

and VGG are less than the systolic array size (6), so the resource utilization is less than that on workloads LeNet-5 and HG. For Tiling, the small quantity of input feature maps and output feature maps attributes to the low resource utilization for former four applications. The number of feature maps of most layers in AlexNet and VGG are coincidentally multiple times of tiling factor (16), so Tiling acquires high resource utilization for this two workloads. Even so, the decent Tiling is still left behind by FlexFlow. For 2D-Mapping, the feature map size of the second or later layers of these workloads is smaller than computing array, which wastes computing resources. Without flexible dataflow supports, the low utilization is inevitable.

6.2.3. Performance

The high utilization of FlexFlow makes a leading position in the performance competition. The performance of four baselines is shown in Figure 16. FlexFlow can constantly acquire over 420GOPs performance with $1GHz$ working frequency. FlexFlow provides more than 2x performance speedup over Systolic and 2D-Mapping, and 10x over Tiling for some cases. The performance of these baselines is proportional to the computing resource utilization in 2D-Mapping and Tiling, but not Systolic. Unlike 2D-Mapping, the Systolic occupies more computing resource, but it acquires low performance because Systolic needs a long initialization phase to fill its deep pipeline.

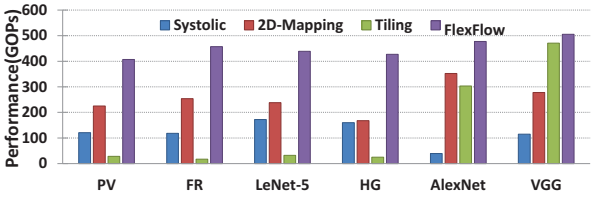


Figure 16. Performance for Different Baselines

6.2.4. Data Reusability

The total volume of the data transmitted is depicted in Figure 17. FlexFlow imposes the least data volume across the six workloads. The novel dataflow mechanisms, RS, RA, and random local store mechanisms, show great "retainability" in data locality. RS and RA can smoothly forward the neurons and synapses to the most suitable PEs. And the local store mechanism maximizes the reusing of these data. The Tiling architecture dictates huge volume of data transmission because, as the analysis in Section 3, it enables the poorest data sharing: the synapses are private in each PE, and $T_m \times T_n$ synapses are loaded at each clock cycle. For Systolic, it acquires high data spatial sharing (neurons) and data reusing (synapses), so it needs a small volume of data transmission. For 2D-Mapping, it exploits local FIFOs to reuse neuron data along PE rows and columns, and therefore achieve decent data reusability as well. 2D-Mapping is slightly worse than Systolic, largely because the input feature maps are still needed to be read multiple times corresponding to different output feature maps. By contrast, the Systolic can enable some input feature maps sharing between the 7 systolic arrays. Even so, the decent Systolic and 2D-Mapping are still left behind by FlexFlow.

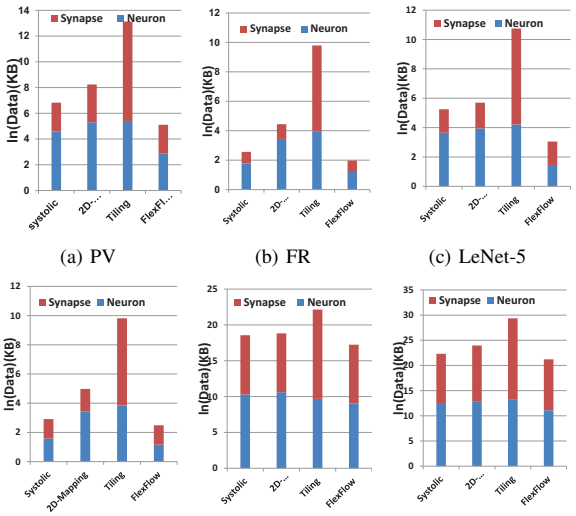


Figure 17. Data Reusability

6.2.5. Power and Energy

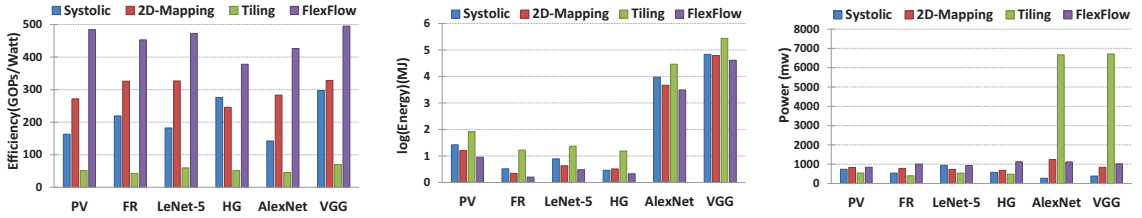
Computing logics. FlexFlow still leads in the power efficiency comparison. Power efficiency (a.k.a performance per watt) is defined as GOPs/Watt. Figure 18(a), shows the efficiency result. FlexFlow gets the highest power efficiency. It gets about 1.5-2.5x power efficiency speedup over Systolic and 2D-Mapping, and even 10x over Tiling. Higher power efficiency implies that accomplishing a task takes less energy, so we also illustrate the energy in Figure 18(b), which is totally in line with the power efficiency statistics.

Although simply compare power is meaningless, it's a good indicator for utilization. Figure 18(c) illustrates that the power for FlexFlow is higher than the other baselines, the reason is twofold: 1) high computing resource utilization contributes to power dissipation, 2) the internal local stores consume overmuch power. Table 6 lists the power dissipation for each component of FlexFlow architecture across six workloads. From Table 6, we can find that the three on-chip buffers occupy very small part power budget (less than 20%), the computing engine including neuron local store and kernel local store occupies a large share of the total power. Although the power is higher than other baselines, FlexFlow still consumes far less energy to complete the four workloads due to its high performance.

Interconnection. We also compare the power of FlexFlow with ideal routing network and the practical one to investigate the power

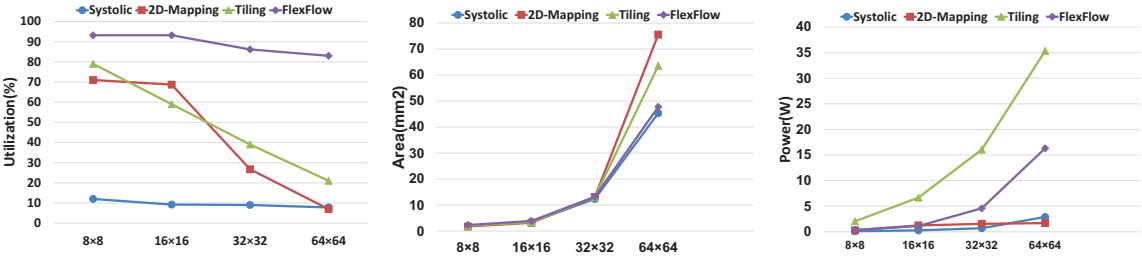
TABLE 6. POWER BREAKDOWN BY COMPONENT

Workloads	P_{nein} $(mw)(\%)$	P_{neout} $(mw)(\%)$	P_{kerin} $(mw)(\%)$	P_{com} $(mw)(\%)$
PV	48(5.7)	66(7.9)	15(1.8)	711(84.6)
FR	61(6.1)	75(7.4)	25(2.5)	847(84.0)
LeNet-5	49(5.3)	72(7.8)	28(3.0)	779(83.9)
HG	54(4.8)	94(8.3)	79(7.0)	900(79.9)
AlexNet	58(5.1)	75(6.7)	27(2.4)	958(85.8)
VGG	50(4.9)	86(8.4)	23(2.2)	860(84.5)



(a) Power Efficiency (b) Energy (c) Power

Figure 18. Power Efficiency, Energy, and Power



(a) Utilization (b) Area (c) Power

Figure 19. Scalability

of routing network. We find that the power percent of routing network gradually declines with the increasing of PE scale: 28.34% for 16x16, 25.97% for 32x32, and 21.32% for 64x64. The power proportion of interconnection keep stable with the growing scale of computing engine. This is because FlexFlow removes most of the fussy interconnections between PEs and uses the vertical and horizontal common bus (CDB) feeding data. Unlike radically growth in routing complexity as other baselines, the routing complexity grows much linearly with the scale of PEs.

Note that Systolic and 2D-Mapping have similar dissipation of power and energy. Systolic has a higher data reusability than 2D-Mapping, so the power of Systolic is less than 2D-Mapping in most cases. But the performance of 2D-Mapping is higher than Systolic. Overall, the energy efficiency of 2D-Mapping is higher than Systolic in many cases. Tiling takes the bottom of power comparison because of low computing resource utilization, and the poorest energy efficiency due to low performance and poor data reusability.

6.2.6. Scalability

We evaluate the scalability of the four baselines using AlexNet workload, given it is the most complicated in the benchmarks. The trend of computing resource utilization, power, and chip area of various computing engine scales (8×8 PEs, 16×16 PEs, 32×32, and 64×64 PEs) is depicted in Figure 19. Generally, with the computing engine scaling up, the computing resource utilization for the former three baselines drops drastically. However, FlexFlow can constantly maintain high resource utilization. The adaptivity of FlexFlow again shows a great merit missed in the other baselines.

In term of area scalability, FlexFlow acquires slower growth of chip area than 2D-Mapping and Tiling, largely because the interconnection of FlexFlow is much simplified than that of other baselines. The power scalability highly depends on the resource utilization. Reasonably, FlexFlow and Tilling shows relatively linear power growth (note the X-axis is in exponential scale), but the slop of FlexFlow is much mild than that of Tiling. The Systolic and 2D-Mapping show to be flat power growth does not imply they retain superior power scalability, but because of the incompetent to exercise the computing resources, as confirmed in Figure 19(a).

6.3. Comparison with Other Accelerators

Due to differences in technology, hardware resources and system setup, it's hard to make an apple to apple comparison between different implementations. But we will list some recent works for qualitative reference. We only list some available data about two most representative implementations in Table 7. Some missing spec is denoted as NA. From the DRAM Accesses Per Operation (Acc/Op), we can see FlexFlow should be more efficient than Eyeriss in terms of data reusability.

TABLE 7. COMPARISON OF ACCELERATORS

	DianNao	Eyeriss	FlexFlow
Process	65nm	65nm	65nm
Num of PEs	256	168	256
Local Store/PE	NA	512B	512B
Buffer Size	36KB	108KB	64KB
Area	$3.02mm^2$	$16mm^2$	$3.89mm^2$
DRAM Acc/Op	NA	0.006	0.0049

7. Related Work

There are commonly three ways - programming (CPU, GPU), recompiling (FPGA) and reconfiguring (ASIP) - to optimize the computation about CNN.

CPU, GPU: General purpose CPUs can be flexibly adapted to various network structures [26], but the flexibility comes at the expense of inherent instruction handling inefficiencies. CPUs can't cope well with the large computations of CNNs. GPUs have become one of most popular platforms for CNNs implementation [19], [23] because of its high performance and flexibility. Unfortunately, it is difficult to overcome the shortcomings of high energy consumption.

FPGA: Recently, many accelerators for CNNs have been proposed based on FPGAs because of the merits of good performance,

high energy efficiency and sufficient recompilability. Work [8] presents a self-contained FPGA implementation of CNN . Work [22] implements a static 2D computing array based on systolic system to optimize the convolutional operation. In [20], a flexible memory hierarchy is proposed to support different kinds of parallelism and to improve the efficiency of on-chip memory. Work [21] is focused on the FC layer. There are two main reasons for developing custom hardware: the low working frequency and long development cycle of FPGA platforms.

ASIP: The application-specific instruction-set processors (ASIP) can provide high performance and flexibility concurrently. So, researchers pay more and more attention to this direction. Work [1] proposed a coprocessor for CNNs, which can dynamically configure "inter-output" and "intra-output" parallelism. In [7], a runtime reconfigurable 2D dataflow computing engine is proposed, which can implement all kinds of operations about CNN. The above two designs are built around systolic implementations. Work [2] and work [3], use loop tiling to optimize computation and communication. In [6], a instruction programmable accelerator is proposed, which maps 2D feature map on 2D processing array to improve efficiency. It belongs to 2D-Mapping computing architecture. Eyeriss [4] proposed a row stationary (RS) data flow, which maps each row of output feature map to one PE. PEs are organized in a 2D-Mapping like way, and the diagonal PEs are also linked together. So it also suffers some limitations of conventional architectures. Moreover, the output path varies with the PE set changing, which will attribute huge control and routing overhead.

8. Conclusion

We designed a flexible dataflow architecture (FlexFlow) for CNN applications, which can exploit parallelism complementary effects to dynamically mix different types of parallelism along with different CNN workloads. It greatly mitigates the mismatch between computing architecture and CNN workloads. Tested by six typical workloads, our design provides more than 2-10x and 2.5-10x performance and power efficiency speedup over three typical baselines. Besides, our design has high scalability, it can be scaled up in different sizes with stable resource utilization.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61572470, 61532017, 61522406, 61432017, and 61376043, in part by Youth Innovation Promotion Association, CAS under grant No.Y404441000.

References

[1] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi. A dynamically configurable coprocessor for convolutional neural networks. *SIGARCH Comput. Archit. News*, 38(3):247–257, June 2010.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *SIGARCH Comput. Archit. News*, 42(1):269–284, Feb. 2014.

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

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

[5] S. Dawwd and B. Mahmood. A reconfigurable interconnected filter for face recognition based on convolution neural network. In *Design and Test Workshop (IDT)*, pages 1–6, Nov 2009.

[6] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *International Symposium on Computer Architecture(ISCA)*, pages 92–104. ACM, 2015.

[7] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116, June 2011.

[8] C. Farabet, C. Poulet, J. Han, and Y. LeCun. Cnp: An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37, Aug 2009.

[9] S. S. Farfade, M. J. Saberian, and L.-J. Li. Multi-view face detection using deep convolutional neural networks. In *Proceedings of the 5th ACM on International Conference on Multimedia Retrieval*, ICMR '15, pages 643–650, New York, NY, USA, 2015. ACM.

[10] X. Hu, X. Lu, and C. Hori. Mandarin speech recognition using convolution neural network with augmented tone features. In *Chinese Spoken Language Processing (ISCSLP), International Symposium on*, pages 15–18, Sept 2014.

[11] J. Huang, J. Li, and Y. Gong. An analysis of convolutional neural networks for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4989–4993, April 2015.

[12] M. Khalil-Hani and L. S. Sung. A convolutional neural network approach for face verification. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pages 707–714, July 2014.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[14] M. M. Lau, K. H. Lim, and A. Gopalai. Malaysia traffic sign recognition with convolutional neural network. In *Digital Signal Processing (DSP), 2015 IEEE International Conference on*, pages 1006–1010, July 2015.

[15] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[17] H. I. Lin, M. H. Hsu, and W. K. Chen. Human hand gesture recognition using a convolution neural network. In *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 1038–1043, Aug 2014.

[18] C. Liu, J. Liu, F. Yu, Y. Huang, and J. Chen. Handwriting character recognition with sequential covolutional neural network. In *Proceedings of the 2013 International Conference of Machine Learning and Cybernetics*, pages 291–296. IEEE, July 2013.

[19] Q. Liu, Z. Huang, and F. Hu. Accelerating convolution-based detection model on gpu. In *Estimation, Detection and Information Fusion (ICEDIF), 2015 International Conference on*, pages 61–66, Jan 2015.

[20] M. Peemen, A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 13–19, Oct 2013.

[21] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. ACM.

[22] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. Graf. A massively parallel coprocessor for convolutional neural networks. In *Application-specific Systems, Architectures and Processors*, pages 53–60, July 2009.

[23] D. Scherer, H. Schulz, and S. Behnke. Accelerating large-scale convolutional neural networks with parallel graphics multiprocessors. In K. Diamantaras, W. Duch, and L. Iliadis, editors, *Artificial Neural Networks C ICANN 2010*, volume 6354 of *Lecture Notes in Computer Science*, pages 82–91. Springer Berlin Heidelberg, 2010.

[24] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil. Learning semantic representations using convolutional neural networks for web search. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 373–374, Republic and Canton of Geneva, Switzerland, 2014. International World Wide Web Conferences Steering Committee.

[25] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[26] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *in Deep Learning and Unsupervised Feature Learning Workshop, NIPS*, 2011.

[27] C. Wang, H. Zhang, L. Yang, S. Liu, and X. Cao. Deep people counting in extremely dense crowds. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, pages 1299–1302. ACM, 2015.

[28] R. Wang and Z. Xu. A pedestrian and vehicle rapid identification model based on convolutional neural network. In *Proceedings of the 7th International Conference on Internet Multimedia Computing and Service*, ICIMCS '15, pages 32:1–32:4, New York, NY, USA, 2015. ACM.

[29] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.

[30] J. Zheng, Y. Wang, and W. Zeng. Cnn based vehicle counting with virtual coil in traffic surveillance video. In *Multimedia Big Data (BigMM), 2015 IEEE International Conference on*, pages 280–281, April 2015.