FCNNLib: A Flexible Convolution Algorithm Library for Deep Learning on FPGAs

Yun Liang, *Senior Member, IEEE*, Qingcheng Xiao, Liqiang Lu, and Jiaming Xie

***Abstract*—Convolution features huge complexity and demands high computation capability. Among hardware platforms, field programmable gate array (FPGA) emerges as a promising solution for its substantial available parallelism and energy efficiency. Besides, convolution can be implemented with different algorithms, including conventional, general matrix–matrix multiplication (GEMM), Winograd, and fast Fourier transformation (FFT) algorithms, which are diverse in arithmetic complexity, resource requirement, etc. Different convolutional neural network (CNN) models have different topologies and structures, favoring different convolution algorithms. In response, software libraries such as cuDNN provide a variety of computational primitives to support these algorithms. However, supporting such libraries on FPGAs is challenging. First, multiple algorithms can share the FPGA resources spatially as well as temporally, introducing either reconfiguration overhead or resource underutilization. Second, FPGA implementation remains a significant challenge for library developers. It typically requires significant specialized hardware knowledge. In this article, we propose `FCNNLib`, an efficient and scalable convolution algorithm library on FPGAs. To coordinate multiple convolution algorithms on FPGAs, we develop three schedulings: 1) spatial; 2) temporal; and 3) hybrid, which exhibit different tradeoffs in latency and throughput. We explore these schedulings by balancing the reconfiguration overhead, resource utilization, and optimization objectives of the CNNs. Then, we provide efficient and tunable algorithm templates that allow performance tuning through performance and resource models. To arm the users, `FCNNLib` exposes a set of interfaces to support high-level application designs. We demonstrate the usability of `FCNNLib` with state-of-the-art CNNs. `FCNNLib` achieves up to 44.6× and 1.76× energy efficiency in various scenarios compared with software libraries for CPUs and GPUs, respectively.**

***Index Terms*—Convolution, deep learning, FPGAs, library.**

I. Introduction

ADVANCES in deep convolutional neural networks (CNNs) are leading to emerging applications, such as image recognition [1], semantic segmentation, and speech recognition [2], [3]. Convolutions are becoming the base of today's and future computing systems. With the prevalence of CNNs, there is an increasing demand for hardware acceleration as both CNNs training and inference demand a tremendous amount of computation. As a result, hardware accelerators, such as GPUs, field programmable gate arrays (FPGAs), and customized ASICs, have been employed to accelerate DNNs [4]–[11]. Among them, FPGAs emerges as a promising solution owing to its high available parallelism and flexibility [12]–[17].

On the other hand, different algorithms for the essential convolution operation in CNNs have been studied [18]. These algorithms include conventional, general matrix–matrix multiplication (GEMM), Winograd, and fast fourier transformation (FFT) algorithms. Conventional algorithm is performed on the original features, while the other three algorithms transform data to other domains and transform the results back after the computation. These algorithms are diverse in arithmetic complexity and dataflow. As a result, the performance and resource utilization of these algorithms may vary considerably, depending on the CNN models and layer parameters. For instance, by using the Winograd algorithm in cuDNN [19], the number of multiplications in VGGNet [20] can be reduced to half of the conventional algorithm, leading to about 2.7× inference latency speedup. GEMM and Winograd algorithms can collaborate in ResNet [21] and provide 1.4× latency speedup. Due to the importance of convolution operations, highly optimized convolution libraries supporting different algorithms, such as Arm Compute Library [22], math kernel library for deep neural networks (MKL-DNNs) [23], and cuDNN [19], are commonplace for CPU and GPU platforms. For instance, cuDNN provides up to eight different algorithms to perform convolutions on GPUs and is widely used in deep learning frameworks, such as Tensorflow [24] and PyTorch [25].

However, such systematic library support of different convolution algorithms is not quite here for FPGAs, in large part because FPGAs are highly reconfigurable and difficult to program. First, implementing multiple convolution algorithms is hard. Multiple algorithms can share the FPGA resources spatially as well as temporally. Spatial sharing is facilitated by configuring different portions of FPGAs for different algorithms; and temporal sharing by reconfiguring the FPGAs to implement different algorithms over time. Temporal sharing achieves high flexibility at the expense of extra reconfiguration overhead. Spatial sharing avoids the overhead but at the cost of potential resource underutilization. The application programmers do not yet have reliable intuition about which sharing or scheduling mechanism should be used. Furthermore, different models or different layers of the same

Manuscript received 14 September 2020; revised 26 June 2021; accepted 13 August 2021. Date of publication 26 August 2021; date of current version 19 July 2022. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2019B010155002, and in part by PKU-Baidu Fund under Project 2020BD024. This article was recommended by Associate Editor P. A. Beerel. *(Corresponding author: Yun Liang.)*

The authors are with the Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, Beijing 100871, China (e-mail: ericlyun@pku.edu.cn; walkershaw@pku.edu.cn; luliqiang@pku.edu.cn; jmxie@pku.edu.cn).

Digital Object Identifier 10.1109/TCAD.2021.3108065

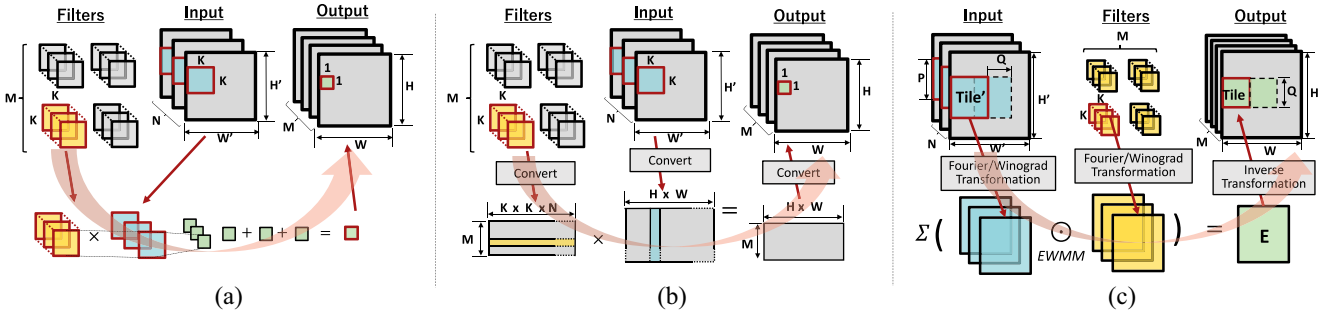


Fig. 1. Dataflow of different convolution algorithms. (a) Conventional algorithm. (b) GEMM algorithm. (c) Winograd and FFT algorithms.

models may favor different algorithms and call for different scheduling techniques.

Second, FPGAs programming remains to be a significant challenge for library developers. The wide adoption of the library may be hindered by low-level programming models. Recent advances in high-level synthesis (HLS) for FPGAs have lowered the programming barrier, which is evidenced by the wide adoption of commercial tools, including Intel FPGA SDK for OpenCL [26] and Xilinx Vivado HLS [27]. However, the original software implementation may not be suitable for hardware. To maximize algorithm performance, programmers need to extensively restructure the source code to realize the unique hardware features. Moreover, the HLS tools offer various optimization directives (e.g., loop unrolling, pipelining, and memory partitioning) with nontrivial performance and resource tradeoffs, which are difficult to explore.

In this article, we propose `FCNNLib`, an efficient and scalable convolution algorithm library for CNN inference on FPGAs. We propose three scheduling techniques to coordinate multiple algorithms on FPGAs. Temporal scheduling allows multiple algorithms to occupy FPGA resources over time. Spatial scheduling shares resources among multiple algorithms. Hybrid scheduling combines the benefits of spatial and temporal scheduling. We further improve these scheduling with the assistance of optimization algorithms to address reconfiguration overhead and hardware underutilization issues. Then, to accommodate efficient convolution implementation on FPGAs, we design an optimized template for each convolution algorithm in HLS. These templates provide configurable factors and are highly optimized with HLS directives. We also develop performance and resource models to find the optimal factors subject to resources constraints. Moreover, `FCNNLib` provides high-level library interfaces to facilitate the users to explore different algorithms and schedulings for a variety of CNN models.

Prior works mainly focus on implementing one convolution algorithm on FPGAs [28]–[31]. `FCNNLib` is the first to provide systematic library support of various convolution algorithms on FPGAs. A preliminary version of this article was presented in [32]. In [32], we propose and compare three multialgorithm scheduling techniques. In this extension, we provide details about library implementations. In particular, we show the loop optimizations made to the HLS-based algorithm templates. We also use resource and performance models to determine algorithm combinations and resource allocations. Overall, the key contributions are as follows.

1) We propose a hardware library `FCNNLib`, which provides efficient and scalable implementations of multiple convolution algorithms for inference on FPGAs.
2) We develop three multialgorithm scheduling techniques, including spatial, temporal, and hybrid scheduling.
3) We provide highly optimized algorithm templates, performance, and resource models for performance tuning.
4) We provide a succinct set of interfaces to facilitate the users to explore the library.

We evaluate `FCNNLib` with state-of-the-art CNN models on embedded and cloud platforms. The experiments demonstrate that designs offered by `FCNNLib` achieve better or comparable performance and efficiency results compared with dedicated FPGA accelerators. When compared with off-the-shelf software libraries for CPUs and GPUs, `FCNNLib` achieves up to 44.6× and 1.76× energy efficiency, respectively.

II. Background and Motivation

A. Convolution Algorithms

Convolution in CNNs is to shift a group of 3-D filters over an input tensor and outputs a result tensor. Assume the input is composed of N feature maps with size $H' \times W'$, while M filters all have a $K \times K \times N$ shape. Each filter convolves with the input tensor at stride S to obtain one feature map with size $H \times W$ in the output tensor so that features are extracted. In this way, after convolving all filters, M output feature maps are generated. The following equation details the convolution operation for each output element:

$$\text{out}_{m,h,w} = \sum_{n=1,r=1,c=1}^{N,K,K} \text{in}_{n,h\times S+r,w\times S+c} \times \text{filter}_{m,r,c,n}. \quad (1)$$

The basic convolution implementation is in line with the above formula using six nested loops, as shown in Fig. 1(a). We refer to it as *conventional algorithm*.

Convolutions can be converted to matrix multiplications. As Fig. 1(b) shows, each filter is flattened into a row of filter matrix with length $K \times K \times N$. For the input matrix, each $K \times K \times N$ input feature map tile corresponding to an output element is also flattened into a column. By this way, multiplying a row and a column is equivalent to the convolution operation for an output element. In this article, we refer to

TABLE I
COMPARISONS OF CONVOLUTION ALGORITHMS

Algo.	Arithmetic Complexity	Memory Resource	Logic Resource	Accuracy	Adaptability
Conventional	High	Low	Medium	High	High
GEMM	High	Medium	Low	High	High
Winograd	Low	Medium	High	Low	Low
FFT	Medium	High	High	Medium	Medium

this implementation as the *GEMM algorithm*. *Winograd* and FFT algorithms are also known as fast algorithms, where they batch the computation of multiple output tiles by exploiting the structural similarity in an input tile. More concretely, they first transform the input tile and filter into Winograd and FFT domains, then perform elementwise multiplication (EWMM), and finally, transform the EWMM results back to the original output tile. Fig. 1(c) illustrates their dataflows.

B. Convolution Algorithms on FPGAs

In FPGAs, DSPs and look up tables (LUTs) are responsible for arithmetic and logic operations, respectively, while block RAMs (BRAMs) and flip-flops (FFs) are used for buffering data. In recent years, FPGAs have been demonstrated as a promising solution to accelerate CNNs [33]–[35]. Convolution algorithms implemented on FPGAs differ in arithmetic complexity, resource, accuracy, and adaptability, as Table I shows.

Arithmetic Complexity: The arithmetic complexity of a convolution algorithm represents the number of multiplications performed for each output element. Therefore, it is linear with DSP consumption on FPGAs. The arithmetic complexity of conventional and GEMM algorithms is $M \times N \times K \times K$, according to (1). Winograd and FFT algorithms replace partial multiplications with transformations. Hence, the complexity can be reduced to $M \times N \times P^2/Q^2$, where P^2 and Q^2 are the input and output tile sizes as shown in Fig. 1(c). A typical tile size ($P = 6$, $Q = 4$) reduces the complexity by 4X compared with conventional and GEMM algorithms.

Memory Resource: Memory resource represents both on-chip BRAMs and off-chip bandwidth. BRAM is used to buffer data, and its usage mainly depends on the total data size and data access pattern. The GEMM algorithm increases the data size by K^2 times since inputs are flattened into vectors. Winograd and FFT algorithms require to access the elements within a tile simultaneously by consuming more BRAMs.

Logic Resource: Logic resources of conventional and GEMM algorithms are mainly for building memory interconnect between processing elements (PEs) and BRAMs. Winograd and FFT algorithms use logic resources for the transformation functions. These functions use either constant matrix multiplication or Fourier transformations, which can be implemented using shift operations on FPGAs.

Accuracy: Using low-precision data in CNN applications is a common optimization on the FPGA platform. It has been demonstrated that 4–8 bit fixed point type can maintain high CNN accuracy [36]–[38] for conventional and GEMM algorithms [28]–[30]. However, Winograd and FFT algorithms introduce errors at their transformation stages. They require higher data precision to maintain CNN accuracy.

TABLE II
LAYER PREFERENCE ON CONVOLUTION ALGORITHMS. THE TARGET PLATFORM IS XILINX ZC706

Model	Layer Para. ($filterSize$, $stride$)	Oper. Ratio	Optimal Algorithm	Performance (GOPS)	
				Single Algo. Layer Peak	Single Algo. Overall
ResNet[21]	(7, 2)	1.0%	conven.	213.1	140.6
	(3, 1)	50.2%	Wino.	548.9	168.0
	(1, 1)	48.8%	GEMM	232.1	146.5
GoogLeNet [39]	(7, 2)	7.6%	conven.	173.1	77.8
	(5, 1)	8.5%	FFT	224.7	119.8
	(3, 1)	61.0%	Wino.	408.9	127.5
	(1, 1)	22.8%	GEMM	162.1	93.4
DenseNet [40]	(7, 2)	2.3%	conven.	213.1	151.2
	(3, 1)	39.1%	Wino.	548.9	178.6
	(1, 1)	58.6%	GEMM	232.1	175.5
DQN[41]	(5, 2)	99.9%	FFT	224.7	108.7
	(1, 1)	0.1%	GEMM	232.1	99.7

Adaptability: Conventional and GEMM achieve consistent performance for convolutions with different structures. However, the benefits of Winograd and FFT algorithms fade as the convolution strides increase. The reason is that data are organized as tiles in Winograd and FFT algorithms. When strides are larger than 1, Winograd and FFT algorithms process the convolutions as if their strides were 1 and screen out the valid outputs after computation.

Table II lists four real-world CNN models where layers have diverse parameters. We observe that different convolution layers favor different convolution algorithms on FPGAs. For instance, the optimal algorithm for one layer (filter size 3, stride 1) is Winograd, while the optimal algorithm for another layer (filter size 7, stride 2) is conventional. Using the best algorithm for a single layer provides the ideal peak performance. However, if one fixed algorithm is chosen for the entire model, the overall performance drops sharply compared to the layer peak performance. For instance, the overall performance drops more than 3× (548.9 versus 178.6) compared to the layer performance if we use the Winograd algorithm consistently for DenseNet. The drop indicates that using a single algorithm for all the layers or models will cause great sacrifice on performance. Different layers and models desire different convolution algorithms. Inspired by this, we design an efficient library that provides a variety of algorithms to implement convolution on FPGAs.

III. FCNNLIB OVERVIEW

`FCNNLib` employs three components: 1) multialgorithm scheduling; 2) algorithm templates; and 3) programming interfaces. The scheduling coordinates multiple convolution algorithms on FPGAs. The templates enable `FCNNLib` to generate efficient designs automatically. Finally, the programming interfaces ease the FPGA programming hurdle.

1) *Multialgorithm Scheduling:* We propose three scheduling techniques, as shown in Fig. 2. *Temporal scheduling* dynamically swaps algorithms at runtime according to the layer parameters. In the figure, we decide a

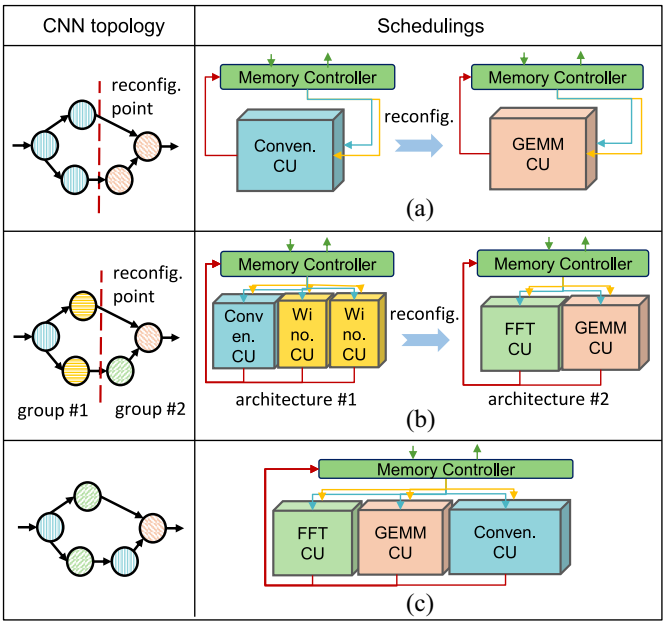


Fig. 2. Multialgorithm schedulings. (a) Temporal scheduling. (b) Hybrid scheduling. (c) Spatial scheduling.

reconfiguration point in the target CNN and change the original conventional algorithm into GEMM algorithm. To enable temporal scheduling, FPGAs have to be reconfigured to replace the implementation for one algorithm with another one. *Spatial scheduling* lets each algorithm occupy partial on-chip resources and maintains the same architecture through the whole CNN inference. Therefore, no reconfiguration is required. In Fig. 2, three algorithms are implemented in an architecture, and each algorithm CU handles partial layers. *Hybrid scheduling* partitions CNN models into several groups that each occupies the whole on-chip resources. Reconfiguration is triggered only when all workloads in a group are accomplished. In the figure, the CNN model is partitioned into two groups, resulting in two architectures. We discuss more details in Section IV.

2) *Algorithm Templates:* For all convolution algorithms, `FCNNLib` provides highly optimized HLS templates with parameters, such as data type, feature map shapes, filter size, stride, and parallelism. The templates can be used for performance tuning on different FPGAs. They form the architectures for the three schedulings (spatial architecture, temporal architecture, and hybrid architecture). We give more details in Section V.
3) *Interfaces:* We provide a set of high-level interfaces for library users. Users can specify the CNN model, FPGA platform, and a scenario. Then, `FCNNLib` will automatically map the model onto FPGAs by exploring various convolution algorithms and scheduling combinations. Grammars and examples are given in Section VI.

IV. Multialgorithm Scheduling

A. Spatial Scheduling

In spatial scheduling, we partition hardware resources (resource partitioning) for convolution algorithms so that each

Algorithm 1: Resource Partitioning Algorithm

Input: *CNN_model*, *R*, *max_iter*
Output: *final_PTN*
1 *iter* ← 0, *PTN* ← the initial resource partitioning solution
2 # *PTN* is a vector $\langle R_{conven.}, R_{GEMM}, R_{Wino.}, R_{FFT}\rangle$
3 **while** *iter* ≤ *max_iter* **do**
4 *PTN′* ← generate a new solution based on *PTN*
5 *arch′* ← build a spatial architecture based on *PTN′*
6 *ceiling_latency′* ← evaluate the ceiling inference latency of *arch′* when processing *CNN_model*
7 **if** *ceiling_latency′* < *current_latency* **then**
8 *PTN* ← *PTN′*
9 **else**
10 *PTN* ← *PTN′*, with certain possibility
11 *final_PTN* ← the solution with the min ceiling latency
12 **return** *final_PTN*

algorithm is implemented within its resource partitioning as a separated CU. The main challenge is the potential low utilization of spatial architectures: if we assign each convolution workload to a single algorithm CU, the CUs of other algorithms are idle. Hence, we also partition a convolution workload (workload partitioning) and assign subworkloads to all employed algorithms. Solving the resource partitioning and workload partitioning together results in enormous design space. Hence, we divide and conquer spatial scheduling through two stages.

Resource Partitioning Stage: The objective of this stage is to build a good spatial architecture based on a resource partitioning solution. Here, we need to answer: 1) how to express a partitioning solution; 2) how to evaluate a solution (spatial architecture); and 3) how to find a good solution.

First, we define the hardware resource constraints R of FPGAs as a vector $\langle \#BRAMs, \#DSPs, \ldots\rangle$, where each element represents the total amount of a type of resources. For a convolution algorithm *algo*, its resource partitioning R_{algo} is also a vector similar to R. Accordingly, a partitioning solution can be represented as a long vector $\langle R_{\text{conven.}}, R_{\text{GEMM}}, R_{\text{Wino.}}, R_{\text{FFT}}\rangle$.

Second, a reasonable latency upper bound of a spatial architecture is the execution time when each workload is processed by its favorite algorithm. We term this upper bound as the ceiling inference latency and use it to evaluate resource partitioning solutions. The best solution minimizes the ceiling inference latency.

Finally, we explore different partitioning solutions using a simulated annealing algorithm as shown in Algorithm 1. For each partitioning solution *PTN′*, we build a spatial architecture (line 5) with our resource models described in Section V-B. Then, we calculate the ceiling inference latency of the architecture with our performance models (line 6). If the ceiling latency of *PTN′* is the current lowest, we accept *PTN′* as the current solution. Otherwise, we accept *PTN′* with a possibility (lines 7–10). In this way, the solution moves toward the global lowest ceiling latency. We generate and evaluate a new partitioning solution based on the current solution *PTN* (line 4). More clearly, we randomly change the vector values of *PTN* to simulate resource increase or decrease. The

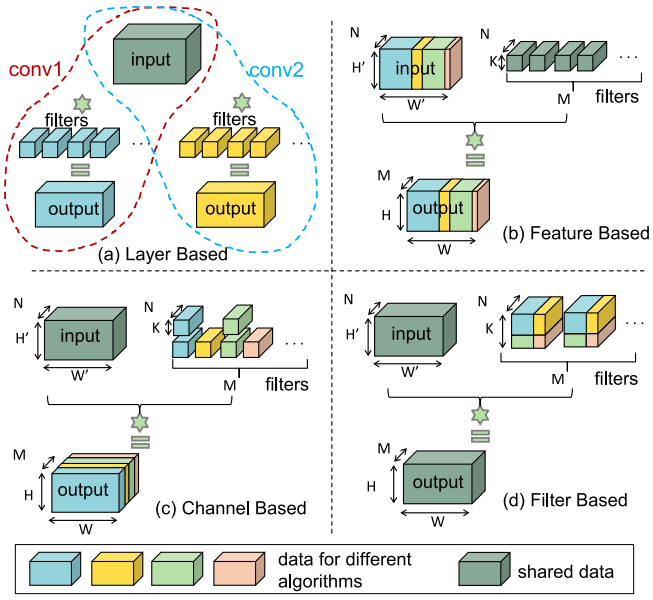


Fig. 3. Workload partitioning methods. (a) Layer based. (b) Feature based. (c) Channel based. (d) Filter based.

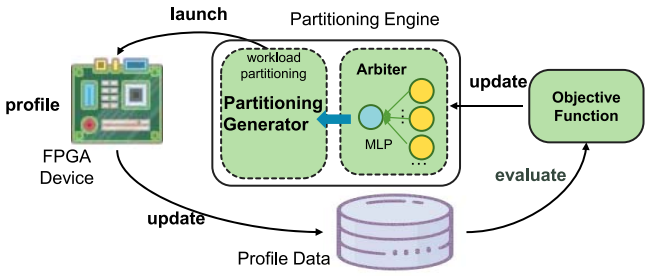


Fig. 4. Workflow of the ML-based partitioning engine.

algorithm continues till the ceiling latency converges, or the iteration number exceeds the given maximum *max_iter*. In the end, Algorithm 1 returns the current best partitioning solution with which we generate a spatial architecture.

Workload Partitioning Stage: The goal of this stage is to balance the workloads among convolution algorithms with a good partitioning. First, we identify four partitioning methods: 1) layer-based; 2) feature-based; 3) channel-based; and 4) filter-based methods, as shown in Fig. 3. Layer-based method assigns each algorithm with the workload of an entire layer, which is a coarse-grained workload unit. More importantly, algorithms can work concurrently only when there is no data dependency between the assigned layers as shown in Fig. 3(a). However, independent workloads are usually limited in CNN models, especially in sequential CNNs. The feature-based method divides each output feature map to several tiles, each of which needs to be produced by an algorithm. Similarly, the channel-based method divides output channels instead of output feature maps. Feature-based and channel-based methods all generate nonoverlapped output regions, as shown in Fig. 3(b) and (c). To achieve the workload balance, they require either large feature maps or substantial number of channels. The filter-based method performs fine-grained partitioning to each filter so that each algorithm produces the partial result of the whole outputs. For example, a 5×5 filter can be divided into one 3×3, two 3×2, and one 2×2 filters. However, as shown, using this method, all algorithms would simultaneously update the whole output feature maps and may lead to memory conflicts.

We employ a hybrid of feature-based and channel-based methods for workload partitioning in `FCNNLib`. The insight is that within a CNN model, the shallow layers usually have large feature maps and a few channels, while the deep layers are on the opposite. Using this hybrid partitioning method, the feature-based method and channel-based method are able to work in a complementary manner.

Then, we develop a machine-learning-based engine to decide the partitioning strategy (method and ratio) specific to each workload. As shown in Fig. 4, the engine consists of a partitioning generator and an arbiter. The generator proposes multiple partitioning strategies for a workload. The arbiter selects one out of the proposed strategies. Specifically, the arbiter is implemented as a multilayer perceptron (MLP). Its inputs include workload parameters and the partitioning strategy to be evaluated. It outputs the relative execution time of the input strategy, as we only care about the relative merits of partitioning strategies proposed by the generator. The relative time prediction is enabled by a rank loss objective function [42]. The objective function *obj* is

$$\text{obj} = \sum_{a,b} \log\left(1 + e^{-\text{sign}(\text{lat}_a - \text{lat}_b) \times (\text{pred}_a - \text{pred}_b)}\right)$$

where a and b are two partitioning strategies, *sign* is the Signum function, *lat* is the actual execution latency, and *pred* is the relative latency predicted by the MLP.

Since the MLP is architecture specific, we train it with runtime statistics. We let the generator propose partitioning strategies. Then, we launch them on the spatial architecture and profile their actual execution time, which is used to train the MLP with the rank loss function. The profiling and training process only takes hours as we have already fixed the spatial architecture in the resource partitioning stage.

B. Temporal Scheduling

Temporal scheduling enables multiple algorithms based on the FPGA reconfigurability. Naturally, layer boundaries are potential reconfiguration points. A reconfiguration benefits the following layer as the layer can use the optimal algorithm. However, it also incurs extra overhead, which is the time to reprogram the FPGA arrays. Taking this tradeoff into account, we develop a dynamic programming algorithm to determine necessary reconfiguration points. The algorithm insight is that when implementing a group of layers, we can either use a single algorithm for all these layers or find an intermediate point to switch to another algorithm. Thus, we formulate the recursion formula as follows:

$$T(i,j) = \min\left\{\min_{i \le k < j}\left\{T(i,k) + T(k+1,j) + \frac{T_{\text{reconf}}}{SZ_{\text{batch}}}\right\}, T_{\text{one}}(i,j)\right\} \quad (2)$$

Algorithm 2: Group Algorithm

Input: i, j, R
Output: *arch*

```
1  opt_arch = INIT(none_unit, max_latency)
2  current_arch = INIT(none_unit, no_latency)
3  Config(current_arch, i, j)
4  return opt_arch
5  Function Config(current_arch, i, j)
6      if i > j then
7          if current_arch.lat < opt_arch.lat then
8              opt_arch = current_arch
9          return
10     foreach convolution algorithm algo do
11         foreach algorithm parameters p for layer i under
             hardware resource constraints do
12             res = ResourceModel(algo, p, layer[i])
13             lat = PerfModel(algo, p, layer[i])
14             new_arch.lat = Max(current_arch.lat, lat)
15             new_arch.res = current_arch.res + res
16             if new_arch.lat ≥ opt_arch.lat then
17                 break
18             if MeetConstraints(new_arch.res) then
19                 Config(new_arch, i + 1, j)
```

where $T(i,j)$ represents the minimal latency for layers i to j after considering reconfiguration. $T_{\text{one}}(i,j)$ is the latency of using a single convolution algorithm without reconfiguration. T_{reconf} is the overhead and SZ_{batch} is the input batch size. FCNNLib amortizes T_{reconf} over batched CNN inputs to improve throughput. To derive $T_{\text{one}}(i,j)$, we enumerate processing layers i to j with the four convolution algorithms. For each algorithm, we first build a compute unit subject to the platform resource constraints. Then, we evaluate the latency when processing layers i to j with the compute unit according to algorithm performance models. Among the four algorithms, the minimal evaluated latency is set as $T_{\text{one}}(i,j)$.

C. Hybrid Scheduling

In our hybrid architecture, each layer in the given CNN is processed by a dedicated CU. The problem lies in that the limited hardware resources cannot accommodate all the hundreds of layers in modern complex CNNs such as ResNet. Hence, we have to partition the CNN layers into multiple groups. For each group, FCNNLib generates an individual architecture on the target FPGA. The reconfiguration is triggered only when all workloads in a group are accomplished. Furthermore, layers within a group are organized as a fine-grained pipeline [43] to improve the overall throughput.

Similar to temporal scheduling, there exists a tradeoff between group number and performance. More groups mean better algorithm customization and performance optimization opportunities but incur more reconfiguration overhead at the same time. To address this tradeoff, we use the same dynamic programming algorithm used in temporal scheduling, as (2) shows. However, here $T_{\text{one}}(i,j)$ denotes the latency of layers from i to j when they are organized as a group. To obtain $T_{\text{one}}(i,j)$, we develop a branch-and-bound algorithm as shown in Algorithm 2. The algorithm explores the architecture for the group composed of layer i to j subject to hardware resource constraints R, as shown in Algorithm 2. Starting from the ith layer, we enumerate various algorithms and parameters for each layer in a depth-first fashion (lines 5–19). We evaluate the latency and resource usage of each layer with the performance and resource models (lines 12 and 13). Since layers form a fine-grained pipeline, the group latency equals approximately to the latency of the slowest layer within the architecture (line 14). Once the jth layer is reached, we update the current best group latency and architecture if necessary. $T_{\text{one}}(i,j)$ is the final group latency. Two constraints bound the search space. For one thing, the total resource usage of all layers is constrained by the on-chip resource (line 18). For another, we use the best historical total latency to bound the following traversal (line 16). If the current group latency already exceeds the best latency, we skip the following layers and try another implementation for the current layer.

```
1  FFT_conv<...>(input, output, filters){
2   #pragma HLS DATAPACK
3   in_buf[N_s][H'_s][W'_s];
4   out_buf[M_s][H_s][W_s];
5   fltr_buf[M_s][K_s][K_s][N_s];
6   #pragma HLS ARRAY_PARTITION
7   for m = 0 to M, m += M_s
8     for n = 0 to N, n += N_s
9       for c = 0 to W , c += W _s
10        for r = 0 to H , r += H _s {
11          #pragma HLS DATAFLOW
12          load<...>(input, in_buf, filters,
                fltr_buf)
13          compute<...>(in_buf, fltr_buf, out_buf);
14          store<...>(out_buf, output) }
15 }
16
17 compute<...>(in_buf, fltr_buf, out_buf){
18     for tc = 0 to W_s, tc += W_p
19     #pragma HLS PIPELINE
20      for tm = 0 to M_s, tm += M_p
21      #pragma HLS UNROLL
22         for tn = 0 to N_s, tn += N_p
23         #pragma HLS UNROLL   {
24 compute the partial result of an output tile
25             for ttm = 0 to M_p, ttm ++
26             ...  }
27 }
```

Listing 1. FFT algorithm template.

V. Algorithm Templates

In this section, we first introduce the optimizations made to algorithm templates. Then, we provide resource and performance models to determine template parameters.

A. HLS-Based Loop Optimization

Convolution algorithms can be written as nested loops in C-based HLS templates. For each convolution algorithm, we design templates by employing four loop optimizations, including tiling, interchange, pipelining, and unrolling. We enable these optimizations through code restructure and HLS directives. Listing 1 gives the code of optimized FFT algorithm template.

Tiling: One of the common performance bottlenecks in convolution algorithms is the high off-chip data access latency. Paralleled computation in convolutions requires

Tb/s bandwidth while FPGAs usually have Gb/s off-chip bandwidth. We employ a two-level tiling to decouple data transfer and parallel computation. Accordingly, we refer to these two-level tiling as storage tiling and parallelism tiling, respectively. Each loop in convolution algorithm is split into three subloops through two-level tiling. For instance, in Listing 1, the loop associated with output channel M is split into three loops in lines 7, 19, and 24. The loop for storage tiling (line 7) determines the total number of elements stored on-chip, while the loop for parallelism tiling (line 19) determines the number of elements involved in parallel computation each time. After this two-level tiling, we use a ping-pong buffer between the storage loop (line 7) and the parallelism loop (line 19) to overlap the data transfer and computation.

Different convolution algorithms have distinct data access and compute patterns. Therefore, they have distinct optimal tiling factors. Assume a loop before performing tiling has trip count *DIM*. We use DIM_s and DIM_p to represent the tiling factors of storage loop and parallelism loop, respectively. Taking Listing 1 for instance, the storage factor and parallelism factor of the M loop are M_s (line 7) and M_p (line 19), respectively. With this denotation, we summarize the tiling factors for each convolution algorithm in Table III.

Then, we determine the optimal value for each tiling factor. Among all factors, some can be predetermined according to compute and data access patterns of algorithms. As illustrated in Fig. 1, conventional and GEMM algorithms calculate each output element individually, while FFT and Winograd algorithms use tiles as the basic computation units. We set parallelism factors for H and W dimension loops based on this compute pattern. We set H_p and W_p as 1 in the conventional algorithm so that the partial results of one output element is calculated. For FFT and Winograd algorithms, we set H_p and W_p as their output tile sizes. In the FFT algorithm, we determine the tile size from two perspectives. First, we use Cooley-Turkey FFT, which restricts the input tile size to the power of 2. Second, using 8×8 input tile and 6×6 output tile leads to $3.45\times$ reduced arithmetic complexity with little transformation overhead [18]. Larger tile sizes require more on-chip memory and significant transformation overhead, while smaller tile sizes bring less arithmetic saving. Similarly, in the Winograd algorithm, we employ a 4×4 output tile size and set H_p and W_p as 4, as shown in Table III. On the other hand, storage factors can be determined based on the on-chip memory size of the target platform. In this way, we fix all storage factors and most parallelism factors in the table by analyzing the algorithm details. There also exist some parallelism factors that cannot be constrained. We leave them to be explored in Section IV.

Interchange: After decoupling data transfer and computation, we reorder all the subloops to ensure that elements of off-chip inputs, outputs, and filters are accessed in burst mode. More concretely, we let all storage loops be the outer loop in the template (lines 7–10 in Listing 1). They are all responsible for off-chip data transfer. Accordingly, all parallelism loops become the inner loops and are abstracted as the compute function (line 16–26), taking charge of the core computations of each algorithm.

TABLE III
TILING FACTORS ON ZC706. H, W, M, N, K, AND K ARE WORKLOAD DIMENSIONS IN LINE WITH (1)

Tiling Factors	H'/H		W'/W		N		M		K	
	s	p	s	p	s	p	s	p	s	p
Conventional	**3 / 1**	**3 / 1**	*160*	**3 / 1**	*32*	◊ [1]	*64*	◊	**3**	**1**
GEMM[2]	-		-		*128*	◊	*128*	◊	*256*	◊
FFT	**8 / 4**	**8 / 4**	*160*	**8 / 4**	*32*	◊	*32*	◊	**5**	**5**
Winograd	**6 / 4**	**6 / 4**	*160*	**6 / 4**	*32*	◊	*32*	◊	**3**	**3**

[1] Factors to be explored.
[2] We denote the matrix multiplication size as $K \times N \times M$.

Then, three phases are naturally formed for all convolution algorithms: 1) data load; 2) computation; and 3) data store (line 12–14). Data fetched in the load phase are stored off-chip after computation. Since data dependency exists among these three phases, we organize them into a pipeline manner to overlap data transfer with computation, improving the overall throughput. In this way, the data transfer time can be successfully hidden for compute-bounded workloads. This phase-level pipelining is enabled by *DATAFLOW* directive.

Pipelining and Unrolling: Finally, we exploit the parallelism for the computation loops, which is the core computation. We choose to pipeline the most outer parallelism loop (lines 17 and 18) and unroll the rest parallelism loops (lines 19–22). Loop unrolling creates multiple instances for the loop body so that they can be processed in parallel. Loop pipelining is to schedule all operations within the loop in a pipeline manner. These two operations are enabled by *PIPELINE* and *UNROLL* directives. Furthermore, parallel computing requires massive data access. To maximize off-chip bandwidth, we pack sequential elements in filters or feature maps into a single wide scalar (line 2). To sustain sufficient on-chip bandwidth, we split a buffer into multiple memory banks, each of which has two access ports. The off-chip and on-chip bandwidth optimization are enabled by *DATAPACK* and *PARTITION* directives.

B. Resource and Performance Models

To enable performance tuning for different tiling factors, we develop accurate resource and performance models for each algorithm, as summarized in Table IV.

Resource Models: Resource models are used to model the occupied resource given tiling factors. Among all the on-chip resources of FPGAs, DSPs and logic resources are mostly spent on performing convolution computation, while on-chip memories are used as buffers. We model each type of resource for each algorithm as follows.

1) *DSP Model:* The number of multiplication is linear with the parallelism factors for all algorithms. We model DSP usage as the product of parallelism factors. Taking conventional algorithm for example, partial results of M_p outputs are generated by $N_p \times K_p \times K_p$ inputs each time. Thus, the DSP usage is $M_p \times N_p \times K_p \times K_p$.
2) *Memory Model:* We model input, output, and filter buffers individually. Since these buffers are all partitioned into banks, we consider both bank number and

TABLE IV
RESOURCE AND PERFORMANCE MODELS. α IS THE DSP CONSUMPTION PER MULTIPLICATION. β IS THE BRAM SIZE. γ AND δ ARE THE COEFFICIENTS IN LINEAR REGRESSION. *Depth*, *II*, AND *Freq* ARE THE PIPELINE DEPTH, ITERATION INTERVAL, AND FREQUENCY, RESPECTIVELY

Algorithm			GEMM	conventional	Winograd	FFT
Resource Models	DSP		$\alpha\ M_p\ N_p\ K_p$	$\alpha\ M_p\ N_p\ K_p\ K_p$	$\alpha\ (H'_p)^2\ M_pN_p$	$\alpha\ 3H'_p(\lfloor H'_p/2 \rfloor + 1)\ M_pN_p$
	BRAM	Input	$2(M_p * K_p)(\lceil M_s/M_p \rceil\ \lceil K_s/K_p \rceil/\beta)$	$(H'_s + H_s)(H'_p\ W'_p\ N_p)(\lceil H'_s/H'_p \rceil\ \lceil W'_s/W'_p \rceil\ \lceil N_s/N_p \rceil/\beta)$		
		Filter	$2(K_p \times N_p)(\lceil K_s/K_p \rceil\ \lceil N_s/N_p \rceil/\beta)$	$(K_p^2\ M_p\ N_p)(\lceil K_s/K_p \rceil^2\ \lceil M_s/M_p \rceil\ \lceil N_s/N_p \rceil/\beta)$		
		Output	$2(M_p \times N_p)(\lceil M_s/M_p \rceil\ \lceil N_s/N_p \rceil/\beta)$	$2(H_p\ W_p\ M_p)(\lceil H_s/H_p \rceil\ \lceil W_s/W_p \rceil\ \lceil M_s/M_p \rceil/\beta)$		
	Logic		$\gamma\ log(M_p) + \delta\ log(N_p)$		$\gamma\ M_p + \delta\ N_p$	
Performance Models	Compute(T_{cpt})		$(\lceil M_s/M_p \rceil\ \lceil N_s/N_p \rceil\ \lceil K_s/K_p \rceil\ II + Depth)/Freq$	$(\lceil M_s/M_p \rceil\ \lceil N_s/N_p \rceil\ \lceil H_s/H_p \rceil\ \lceil W_s/W_p \rceil\ II + Depth)/Freq$		
	Transfer(T_{comm})		$max(M_sK_s, K_sN_s, M_sN_s) \times Bits/BW$	$max(N_s\ H'_s\ W'_s, M_s\ H_s\ W_s, M_s\ N_s\ K_s\ K_s) \times Bits/BW$		
	# repeat(NUM_{rpt})		$\lceil M/M_s \rceil \lceil N/N_s \rceil \lceil K/K_s \rceil$	$\lceil M/M_s \rceil \lceil N/N_s \rceil \lceil H/H_s \rceil \lceil W/W_s \rceil \lceil K/K_s \rceil^2$		
	Overall Latency(T)		$NUM_{rpt}\ max(T_{cpt}, T_{comm})$			

the memory usage for each bank. Taking the output buffer in conventional CU for instance, it is partitioned into $H_p \times W_p \times M_p$ banks and each consumes $\lceil H_s/H_p \rceil \lceil W_s/W_p \rceil \lceil M_s/M_p \rceil/\beta$ BRAMs, as shown.

3) *Logic Resource Model:* As aforementioned, algorithms have distinct usage for logic resources. For conventional and GEMM algorithms, we use logarithmic functions to model LUT usage for memory interconnects. For Winograd and FFT algorithms, we use linear regression to model LUT usage in transformation and inverse transformation.

Performance Models: Since the core computation and data transfer are overlapped, the latency is bounded by the maximum of data transfer time and computation time. For computation time T_{cpt}, since all operations are organized into a pipeline manner, we model the cycle number using iteration interval, iteration number, and pipeline depth. Taking the Winograd algorithm as an example, the iteration number is the product of the total input tile number and channel iteration ($\lceil M_s/M_p \rceil \times \lceil N_s/N_p \rceil$). The iteration interval and pipeline depth are platform-dependent constants. For data transfer time T_{comm}, it is bounded by loading input feature maps, filters, or storing output feature maps. We use the maximal transfer size to model transfer time by modeling the bitwidth of each element and the platform bandwidth.

Data transfer and computation are iterated NUM_{rpt} times to accumulate all partial sums and generate final outputs. NUM_{rpt} is determined by the original buffer size and on-chip buffer size. Putting it all together, we model the overall latency using T_{cpt}, T_{comm}, and NUM_{rpt}, as shown in Table IV.

To validate our models, we compare the predicted and actual results on different FPGAs. On average, our resource and performance models achieve 84.6% and 93.8% accuracy, respectively. FPGA synthesis may cause extra resources for on-chip bus and register, which is not modeled. The accuracy loss of latency results may come from the discrepancy of actual and peak bandwidth and DRAM access latency.

```
1  Step 1: Generate a hardware design on ZC706 FPGA
2  config = getConfig(ZC706, scenario, ResNet,
       Spatial)
3  design = configureTemplt(config)
4  Step 2: Schedule multiple algorithms on FPGAs
5  for wl in ResNet:
6  # partition each workload and assign them to
       algorithms
7   wl.sub_wls, wl.asgmt = balanceWorkload(wl,
        design)
8  foreach input image for ResNet:
9   for wl in model:
10   # execute the model layer by layer with
         balanced sub-workloads
11   wl.output = scheduleAlgo(design, wl.sub_wls,
12                wl.asgmt, wl.input, Spatial)
```

Listing 2. Example of deploying ResNet with FCNNLib.

VI. FCNNLIB INTERFACES

We also design a set of high-level interfaces, as listed in Table V. When using FCNNLib, there are two steps: 1) hardware design generation and 2) multialgorithm scheduling. The design generation step is to generate a CNN accelerator employing multiple convolution algorithms. We provide *getParams* interface to determine parameters for each algorithm template. With these parameters, users can instantiate and integrate algorithm CUs to form a design via *configIPs* interface. The scheduling step is to schedule multiple algorithms on the generated accelerator. Users feed the design with input data and get inference results through *scheduleAlgo* interface. Specific to spatial scheduling, *balanceWorkload* interface is provided to balance workloads among algorithms with the help of the ML-based partitioning engine. Moreover, we provide *autoScheduling* interface, which automatically explores the algorithm and scheduling combinations and returns a design with the best performance.

Listing 2 is an example of deploying ResNet with spatial scheduling on the Xilinx ZC706 board. Subject to resource constraints, *getParams* returns algorithm parameters leading to

TABLE V
PROGRAMMING INTERFACES IN FCNNLIB

Interface	Usage
getParams	Based on the target $platform$, $model$, and $scheduling$, find out $parameters$ using simulated annealing to initialize an architecture.
configIPs	Generate a $design$ by configuring templates with parameters ($params$). For temporal and hybrid scheduling, there may exist multiple configured $design$s.
scheduleAlgo	Run the $workload$ on the $design$ to generate inference $result$s.
balanceWorkload	For a given $design$, partition a $workload$ into $sub-workloads$ and $assign$s them to algorithms.
autoScheduling	Given the $platform$ and $model$, select one $scheduling$ automatically and generate the corresponding $design$.
customizeConfig	Customize a $configuration$ based on an $algorithm\ list$ and the $parameter\ list$ for each algorithm.

TABLE VI
COMPARISONS WITH PREVIOUS FPGA ACCELERATION WORKS

Model	AlexNet			VGG-16			ResNet-152				GoogLeNet			DenseNet-161		DQN
Work	[44]	[45]	spatial	[46]	[47]	spatial	[29]	[30]	FCNNLib spatial		DPU[48]	FCNNLib spatial		FCNNLib spatial		spatial
Hardware Language	HLS	RTL	HLS	RTL	HLS	HLS	HLS	HLS	HLS		RTL	HLS		HLS		HLS
Platform	ZC706	GXA7	ZC706	KU115	Virtex690t	ZC706	ZC706	VU9P	ZC706	VU9P	ZCU102	ZC706	VU9P	ZC706	VU9P	ZC706
Frequency (MHz)	167	100	200	200	150	200	125	200	200	200	281	200	200	200	200	200
Datatype	16-bit fixed	8-16 bits fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	8-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed
conventional		✓		✓	✓		✓	✓			✓					
GEMM			✓						✓	✓		✓	✓	✓	✓	✓
Winograd	✓					✓			✓	✓		✓	✓	✓	✓	
FFT			✓													✓
Latency (ms)	-	12.75	4.91	18.051	65.13	48.24	156.4	17.34	118.9	14.6	-	38.24	6.95	68.5	14.4	0.05
Throughput (GOPS)	202.8	114.5	254.3	1702.3	354	637.8	188.18	1463	190.19	1547.84	541.12	133.89	736.69	209.17	996.6	108.8
DSP Efficiency (GOPS / DSP)	0.223	0.447†	0.290	0.383	0.125	0.708	0.209	0.357	0.270	0.228	0.228	0.322	0.221	0.298	0.229	0.273
Power (W)	9.4	19.5	6.12	-	26	7.6	-	-	5.92	32.2	-	5.00	24.1	5.92	25.2	5.55
Energy Efficiency (GOP / J)	21.4	5.87	41.55	-	13.6	83.92	47.04	-	32.13	48.07	-	26.78	30.57	35.33	39.35	19.60

the highest performance. The resource constraints are obtained by Algorithm 1 in spatial scheduling. FCNNLib generates a ResNet accelerator after instantiating CUs (line 3). For each convolution workload in ResNet, *balanceWorkload* interface partitions it into subworkloads specific to the ResNet accelerator (line 7). Finally, *scheduleAlgo* interface launches the subworkloads on the accelerator and collects results. As for temporal and hybrid schedulings, *scheduleAlgo* would also reconfigure FPGAs when necessary.

VII. EXPERIMENTS

A. Experimental Setup

Benchmarks: We have integrated FCNNLib into PyTorch [25]. To evaluate its efficiency, we use the state-of-the-art CNNs in Table II; VGGNet [20], and AlexNet [49]. AlexNet, VGGNet, GoogLeNet, ResNet, and DenseNet are widely used in classification tasks, while DQN is an emerging reinforcement learning method that provides impressive results for game applications. VGGNet is more regular than the other five CNNs as VGGNet only uses 3×3 filters. We treat fully connected layers as convolutions with 1×1 filters and fuse activation functions with convolutions.

Methodology: We first compare designs generated by FCNNLib with dedicated FPGA accelerators. Then, we compare the three multialgorithm schedulings as the batch size scales. We further compare FCNNLib with software libraries on CPU and GPU platforms in Table VII. Finally, we test the accuracy and scalability of FCNNLib.

Platforms: We use FPGAs designed for both embedded and cloud scenarios. ZC706 board is an embedded SoC platform, consisting of one XC7Z045 FPGA chip, dual ARM Cortex-A9 CPUs, and 1-GB DDR3 memory. Xilinx VU9P board is a PCIe-based board with substantial on-chip resources and has been used in AWS F1 instance. For both platforms, we set all implementation frequency as 200 MHz and use a 16-bit fixed-point data type. We use Xilinx Vivado SDx(v2018.2) [27] for design synthesis. We measure the dynamic power of the whole ZC706 SoC with a meter and estimate the VU9P power through Xilinx Power Estimator [50].

B. FPGA Accelerator Comparison

Prior techniques [29], [30], [44]–[48] shown in Table VI use a single convolution algorithm for CNN inference scenarios where the batch size is 1. For fair comparisons, we use spatial scheduling in FCNNLib to generate designs, which are also given in the table. For VGGNet, FCNNLib only uses the Winograd algorithm showing high performance for the 3×3 convolutions. For other irregular CNNs, FCNNLib integrates a

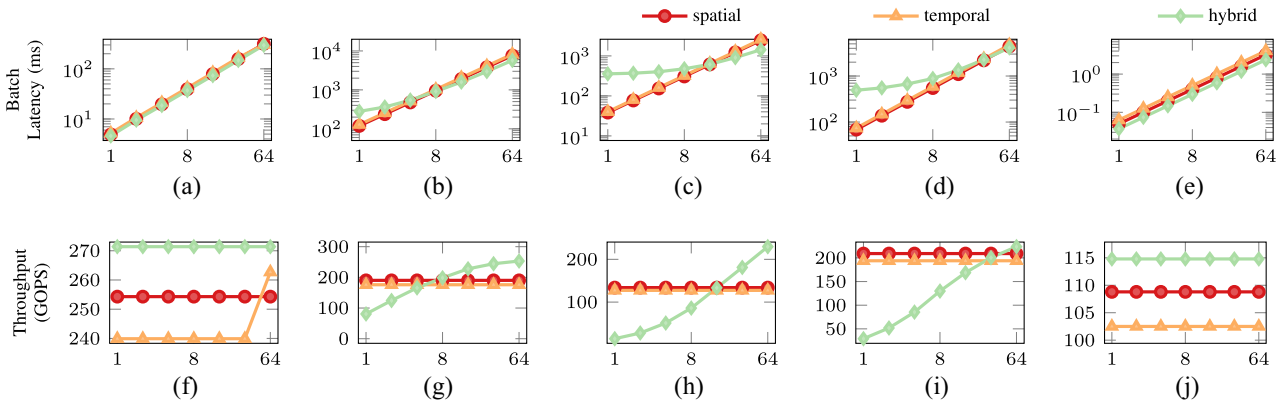


Fig. 5. Comparisons among three schedulings on ZC706. The *x*-axis is batch size ranging from 1 to 64. (a) AlexNet latency. (b) ResNet latency. (c) GoogLeNet latency. (d) DenseNet latency. (e) DQN latency. (f) AlexNet throughput. (g) ResNet throughput. (h) GoogLeNet throughput. (i) DenseNet throughput. (j) DQN throughput.

high-performance algorithm (Winograd or FFT) with a more general one (conventional or GEMM). As the table shows, Winograd and GEMM algorithms are employed for ResNet, GoogLeNet, and DenseNet, while FFT and GEMM algorithms are used for AlexNet and DQN.

We use ResNet, AlexNet, and DQN to demonstrate algorithm choices.

1) In ResNet, 3×3 convolutions and 1×1 convolutions are the most common workloads. Winograd and GEMM are the most efficient algorithms for the two types of workloads, respectively. As a result, `FCNNLib`'s spatial scheduling lets Winograd collaborates with GEMM. It determines the resource partitioning and balances the runtime workloads of the two algorithms. Overall, `FCNNLib` achieves 118.9-ms latency and 190.19 GOPS throughput for ResNet on the ZC706 board, while 14.6-ms latency and 1547.84 GOPS throughput on VU9P device. Previous works [29], [30] employ only the conventional algorithm. By combining multiple algorithms for ResNet, `FCNNLib` spatial scheduling achieves up to 1.315× latency improvement compared with [29], 1.292× DSP efficiency improvement compared with [30].
2) In AlexNet, the first convolutional layer employs 11×11 filters with stride as 4, which is extremely unfriendly to Winograd or FFT algorithms. Therefore, `FCNNLib` uses the GEMM algorithm for this layer and fully connected layers. In addition, `FCNNLib` uses the FFT algorithm to efficiently process other 3×3 and 5×5 layers. Previous work [44] accelerates AlexNet using the Winograd algorithm, which cannot efficiently process the 11×11, 5×5, and 1×1 convolutions. Compared with [44], `FCNNLib` achieves 1.254× throughput and 1.942× energy efficiency.
3) DQN consists of three 5×5 convolutions with strides being 2 and a final fully connected layer. The FFT algorithm is optimal for 5×5 convolutions. `FCNNLib`'s spatial scheduling achieves 0.05-ms latency and meets real-time requirements.

Among the FPGA accelerators listed in Table VI, works [45], [46], and Xilinx DPU [48] are developed in RTL codes, while the others and `FCNNLib` are in HLS. `FCNNLib` can achieve comparable or even better DSP efficiency compared with the RTL designs. Though RTL enables a fine-grained control for memory and data transfer, HLS can already achieve a high DSP utilization such as RTL. Thus, HLS-based designs are efficient enough for convolutions bounded by computations. More importantly, the benefits brought by different convolution algorithms significantly surpass the architecture improvements. For conventional and GEMM algorithms, the theoretical upper bound of their DSP efficiency is $2 \times$ *Frequency* (0.4 GOPS/DSP in our settings). Fast algorithms reduce the arithmetic complexity and enhance the DSP efficiency. Through scheduling convolution algorithms, `FCNNLib` improves the DSP efficiency by 1.849× to 5.664× for VGGNet. In addition, `FCNNLib` provides high productivity, good platform portability, and high performance for different FPGAs and CNN models. Xilinx DPU supports different models by an AI library. Other FPGA accelerators cannot be easily ported to modern neural networks, such as GoogLeNet and DenseNet.

C. Scheduling Comparison

We vary the batch size from 1 to 64 and compare batch latency and throughput results of the three scheduling in `FCNNLib` on ZC706 FPGA, as illustrated in Fig. 5.

For AlexNet, hybrid scheduling can implement all layers without reconfiguration. The low latency of AlexNet only allows temporal scheduling to perform one reconfiguration between the second and third convolutional layers when the batch size is 64. Therefore, the throughput of temporal scheduling remains the same until the 64 batch size. In the DQN case, since DQN consists of only four convolution workloads, all schedulings generate designs without reconfiguration. Hybrid scheduling achieves constantly better throughput results since each layer is processed by a dedicated CU with customized algorithms and parameters.

ResNet, GoogLeNet, and DenseNet are all highly structured CNNs and show similar patterns in Fig. 5. Here, we take

[1]One DSP slice of GXA7 platform is equivalent to two DSP slices of ZC706 platform when performing 16×8 fixed-point multiplications.

TABLE VII
CROSS-PLATFORM COMPARISONS. THERMAL DESIGN POWER IS USED FOR INTEL XEON E5-2630

Scenario	Embedded						Cloud					
Batch Size	1						16			64		
Model	DQN			ResNet			DenseNet			GoogLeNet		
Platform	FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU
Device	ZC706	ARM-A57	TX1	ZC706	ARM-A57	TX1	VU9P	E5-2630	P100	VU9P	E5-2630	P100
Library	FCNNLib spatial	ACL	cuDNN	FCNNLib spatial	ACL	cuDNN	FCNNLib hybrid	MKL -DNN	cuDNN	FCNNLib hybrid	MKL -DNN	cuDNN
Datatype	16-bit fixed	32-bit float	16-bit float	16-bit fixed	32-bit float	16-bit float	16-bit fixed	32-bit float	16-bit float	16-bit fixed	32-bit float	16-bit float
Top-1 Accuracy (%)	-	-	-	77.52	77.55	77.54	77.8	78.0	78.0	68.63	68.92	68.4
Batch Latency (ms)	0.05	0.81	0.04	118.9	2688.89	118.11	198.24	2465.0	108.7	157.1	2176.98	33.0
Throughput (GOPS)	108.8	6.7	123.2	190.19	8.41	191.44	1158.3	93.15	2112.14	2085.4	150.52	9929.6
Power (W)	5.55	7.8	9.2	5.92	9.2	9.8	26.4	85	84.5	25.8	85	91.0
Energy Efficiency (GOP/J)	19.60	0.86	13.39	32.13	0.91	19.54	43.88	1.10	25.00	78.99	1.77	109.12

ResNet for a detailed analysis. The building block of ResNet is residual blocks consisting of one 1×1 convolution, one 3×3 convolution, and one 1×1 convolution. Specifically, ResNet-152 uses 50 residual blocks to extract features. In spatial scheduling, FCNNLib generates a design (Winograd and GEMM algorithms) independent of the batch size and processes input images in sequence. Hence, the batch latency results of spatial scheduling are linear with the batch size. Temporal scheduling chooses to perform no reconfiguration to avoid the overhead. It generates a ResNet design employing only the Winograd algorithm. Thus, the latency results of temporal scheduling are also linear with the batch size and higher than spatial scheduling latency. In contrast to spatial or temporal schedulings, hybrid scheduling always performs reconfiguration. Table VIII gives the group information and algorithm choices. Hybrid scheduling partitions ResNet into 17 groups. Due to the repeating residual blocks, some partitioned groups have the same convolution workloads and can share an architecture. Thus, FCNNLib generates five individual architectures as shown in the table and takes six reconfigurations, leading to around 197.4-ms overhead. As the batch size increases, the overhead is better amortized, improving the throughput from 80.67 GOPS to 253.2 GOPS, as shown in Fig. 5(g). Hybrid scheduling surpasses spatial scheduling for several reasons. First, though the two schedulings almost use the same algorithm combination (Winograd and GEMM), spatial scheduling fixes the parallelism and storage of algorithm CUs, while hybrid scheduling generates different CUs for different workloads. As Table VIII shows, groups 2–16 are all composed of residual blocks but differ in the number and size of feature maps. Hybrid scheduling generates three architectures with different parallelism factors and storage factors to improve performance. Second, data transfer can be the bottleneck of 1×1 convolutions in residual blocks. Hybrid scheduling organizes the workloads in an architecture as a pipeline such that data transfer bottlenecks are eliminated. In addition, hybrid scheduling uses the FFT algorithm for the first 7×7 convolution to enhance the performance further.

Depending on a series of factors, including model topology, resources and reconfiguration overhead of the platform, and available batch size, the three scheduling techniques vary in latency and throughput improvements and require careful selection. We also compare the multialgorithm schedulings with single algorithm designs. Designs generated by FCNNLib achieve up to 2.28× latency speedup in the embedded scenario and 2.95× throughput speedup in the cloud scenario.

D. Library Comparison

Then, we compare our hardware library for FPGA platforms, FCNNLib, with software libraries ARM Compute Library v19.02 [22], MKL-DNN v0.18 [23], and cuDNN 9.0 [19] for CPUs and GPUs, respectively. We let these libraries automatically select algorithms through get_convolution_method, convolution_auto, and cudnnGetConvolutionForwardAlgorithm interfaces, respectively. We conduct comparisons in both embedded and cloud scenarios. For the embedded scenario, we compare the Xilinx ZC706 FPGA board, ARM-A57 CPU, and NVIDIA Jetson TX1 GPU board. For the cloud scenario, the Xilinx VU9P FPGA board, Intel Xeon E5-2630 CPU, and NVIDIA P100 GPU are compared. We use 16-bit data types for FPGAs and GPUs, while a 32-bit floating-point type for CPUs. The reason is that either the platform (ARM-A57) or the library (MKL-DNN) does not support 16-bit floating-point types. Other configurations are also listed in Table VII.

As the table shows, FCNNLib provides constantly better energy efficiency for DQN, ResNet, and DenseNet compared with software libraries. FCNNLib with hybrid scheduling can also achieve comparable performance with cuDNN for GoogLeNet. Overall, FCNNLib provides up to 44.6× and 1.76× energy efficiency compared with MKL-DNN and cuDNN, respectively.

E. Accuracy

Currently, FCNNLib supports 8- to 16-bit fixed-point and 32-bit floating-point datatypes. The performance of FCNNLib improves as the bitwidth decreases. Take accelerating ResNet

TABLE VIII
DETAILS OF HYBRID SCHEDULING FOR RESNET WHEN THE BATCH SIZE IS 64

Group	Workloads (filter, M)	Arch.	Algorithms	Latency (%)
1	[7 × 7, 64] [residual block, 256] × 3	1	FFT, GEMM, Winograd	8.63
2-3	[residual block, 512] × 4	2	Winograd, GEMM	15.18
4-15	[residual block, 1024] × 3	3	Winograd, GEMM	67.84
16	[residual block, 2048] × 2	4	Winograd, GEMM	3.31
17	[residual block, 2048] [1 × 1, 1000]	5	Winograd, GEMM	1.58

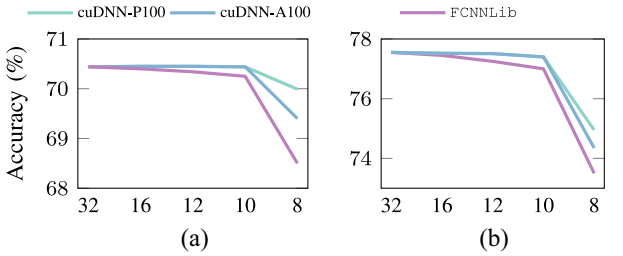


Fig. 6. Top-1 accuracy using different bitwidths and libraries. The *x*-axis represents bitwidths. (a) VGG-16. (b) ResNet.

in embedded scenarios for instance. FCNNLib provides 78.41 GFLOPs to 410.23 GOPs as the datatype varies from 32-bit floating point to 8-bit fixed point.

The inference accuracy of FCNNLib is also sensitive to datatypes. In Fig. 6, we test the inference accuracy of cuDNN and FCNNLib as the bitwidth scales. For P100 GPU, cuDNN uses FP32 CUDA Cores to process both 16- and 32-bit floating point. We convert data less than 16 bits into 16-bit floating point and call cuDNN. Thus, the accuracy loss comes from CNN models. For A100 GPU, cuDNN can directly process 8-bit fixed-point (INT8) data with tensor cores. Then, the accuracy gap between P100 and A100 is from hardware. In contrast, FCNNLib customizes hardware for different bitwidths.

As Fig. 6 shows, FCNNLib restricts the accuracy loss within 0.4% till we use an 8-bit fixed point. The accuracy loss is negligible compared with cuDNN. However, the accuracy results of cuDNN and FCNNLib drop dramatically when using 8-bit data. The reason is threefold. First, overflows happen more and more frequently as data precision decreases. Also, the data transformations of fast algorithms introduce extra errors. In addition, the INT8 accuracy gap between FCNNLib and A100 is caused by our pretrained model. Though we can further fine-tune our 8-bit model or apply FCNNLib to 6-bit or other quantized models, the quantization process is beyond the scope of this work.

F. Scalability

To test the scalability, we set eight different resource constraints that lie in the range from small-scale embedded SoC, large-scale embedded MPSoC, to cloud devices, as shown in Fig. 7. These designs all employ spatial scheduling. FCNNLib

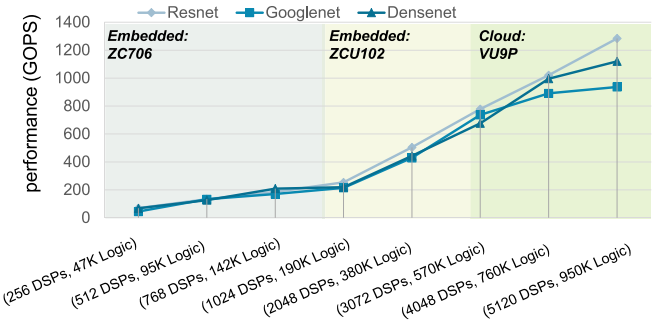


Fig. 7. Scalability results of FCNNLib.

can generate designs that provide performance nearly linear with available resources.

VIII. RELATED WORKS

The implementation of convolution algorithms on FPGAs has been studied in a number of previous work. Xiao *et al.* [43] and Lu *et al.* [44] implemented the Winograd algorithm on FPGAs. Zhang and Prasanna [51] have implemented the FFT algorithm for both CNN training and inference. Suda *et al.* [52] applied the GEMM algorithm to CNN acceleration. As for the conventional algorithm, most of the previous FPGA CNN accelerators are based on it [34]. CHaiDNN [53] and FP-DNN [54] are HLS-based DNN Libraries, which only implement the conventional algorithm. No library has been developed to integrate all these algorithms and to reduce programming difficulty when scheduling multiple algorithms like FCNNLib does. More recent dedicated FPGA accelerators focus on improving each convolution algorithm by optimizing the data movement and parallelism strategy [28], [31]. They are orthogonal to FCNNLib's multialgorithm scheduling techniques.

Convolution libraries for CPU and GPU platforms have been developed and optimized for years. Vendors usually provide their libraries. Intel MKL-DNN [23] is a general library that includes DNN building blocks optimized for Intel CPUs and GPUs. Arm Compute Library [22] includes a collection of low-level functions optimized for Arm Cortex-A CPUs and Arm Mali GPUs, targeting computer vision, and machine learning applications. NVIDIA cuDNN [19] is a GPU-accelerated library that provides highly optimized implementations for standard DNN routines. AMD MIOpen [55] is a machine learning library that provides similar functions to cuDNN. There also exist third-party DNN libraries such as [56]. Most of these libraries include heterogeneous convolution algorithms and provide primitives that help in algorithm selection. In addition, several of previously proposed effective optimizations for machine learning kernels on CPUs and GPUs [57]–[70] can be potentially integrated into libraries.

IX. CONCLUSION

In this article, we proposed a convolution algorithm library FCNNLib consisting of highly optimized algorithm templates, three multialgorithm schedulings, and programming interfaces.

We designed and optimized tunable templates for all convolution algorithms. Then, we explore spatial, temporal, and hybrid schedulings assisted with optimization algorithms. `FCNNLib` also provides a series of programming interfaces to ease the FPGA programming hurdle. The experiments using state-of-the-art CNNs demonstrate that `FCNNLib` achieves up to 44.6× and 1.76× energy efficiency compared with software libraries for CPUs and GPUs, respectively.

REFERENCES

[1] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, "REQ-YOLO: A resource-aware, efficient quantization framework for object detection on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2019, pp. 33–42.

[2] Y. Zhang, W. Chan, and N. Jaitly, "Very deep convolutional networks for end-to-end speech recognition," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, New Orleans, LA, USA, 2017, pp. 4845–4849.

[3] J. Liu, G. Wang, P. Hu, L.-Y. Duan, and A. C. Kot, "Global context-aware attention LSTM networks for 3D action recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 7. Honolulu, HI, USA, 2017, p. 43.

[4] R. Prabhakar *et al.*, "Plasticine: A reconfigurable accelerator for parallel patterns," *IEEE Micro*, vol. 38, no. 3, pp. 20–31, May/Jun. 2018.

[5] W. Wahby, T. Sarvey, H. Sharma, H. Esmaeilzadeh, and M. S. Bakir, "The impact of 3D stacking on GPU-accelerated deep neural networks: An experimental study," in *Proc. IEEE Int. 3D Syst. Integr. Conf. (3DIC)*, San Francisco, CA, USA, 2016, pp. 1–4.

[6] S. Yin *et al.*, "A high throughput acceleration for hybrid neural networks with efficient resource management on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 678–691, Apr. 2019.

[7] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. 45th Annu. Int. Symp. Comput. Archit.*, Los Angeles, CA, USA, 2018, pp. 764–775.

[8] L. Lu *et al.*, "TENET: A framework for modeling tensor dataflow based on relation-centric notation," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Valencia, Spain, 2021, pp. 720–733.

[9] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang, "HASCO: Towards agile HArdware and software co-design for tensor computation," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Valencia, Spain, 2021, pp. 1055–1068.

[10] Y. Liang, L. Lu, and J. Xie, "Omni: A framework for integrating hardware and software optimizations for sparse CNNs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 8, pp. 1648–1661, Aug. 2021.

[11] Y. Liang *et al.*, "An efficient hardware design for accelerating sparse CNNs with NAS-based models," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Mar. 17, 2021, doi: 10.1109/TCAD.2021.3066563.

[12] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei, "Aggressive pipelining of irregular applications on reconfigurable hardware," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Toronto, ON, Canada, 2017, pp. 575–586.

[13] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "FA3C: FPGA-accelerated deep reinforcement learning," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 499–513.

[14] J. Lambert, S. Lee, J. Kim, J. S. Vetter, and A. D. Malony, "Directive-based, high-level programming and optimizations for high-performance computing with FPGAs," in *Proc. Int. Conf. Supercomput.*, 2018, pp. 160–171.

[15] S. Wang *et al.*, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2018, pp. 11–20.

[16] X. Wei, Y. Liang, P. Zhang, C. H. Yu, and J. Cong, "Overcoming data transfer bottlenecks in DNN accelerators via layer-conscious memory managment," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2019, p. 120.

[17] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on FPGAs," in *Proc. IEEE 27th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, San Diego, CA, USA, 2019, pp. 17–25.

[18] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 4013–4021.

[19] NVIDIA Corporation. (2019). *NVIDIA cuDNN*. [Online]. Available: https://developer.nvidia.com/cudnn

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014. [Online]. Available: arXiv:1409.1556.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Las Vegas, NV, USA, 2016, pp. 770–778.

[22] Arm Limited. (2019). *Arm Compute Library*. [Online]. Available: https://www.arm.com/why-arm/technologies/compute-library

[23] Intel Corporation. (2019). *Intel MKL-DNN*. [Online]. Available: https://software.intel.com/mkl

[24] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Conf. Oper. Syst. Design Implement.*, 2016, pp. 265–283.

[25] A. Paszke *et al.*, "Automatic differentiation in pytorch," in *Proc. 31st Conf. Neural Inf. Process. Syst.*, 2017, pp. 1–4.

[26] Intel Corporation. (2019). *Intel FPGA SDK for OpenCL*. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html

[27] Xilinx Inc. (2019). *Xilinx Vivado High-Level Synthesis*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[28] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Washington, DC, USA, 2019.

[29] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 326–342, Feb. 2019.

[30] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, "TGPA: Tile-grained pipeline architecture for low latency CNN inference," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, San Diego, CA, USA, Nov. 2018, pp. 1–8.

[31] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J.-S. Seo, "End-to-end scalable FPGA accelerator for deep residual networks," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Baltimore, MD, USA, 2017, pp. 1–4.

[32] Q. Xiao, L. Lu, J. Xie, and Y. Liang, "FCNNLib: An efficient and flexible convolution algorithm library on FPGAs," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2020, pp. 1–6.

[33] H. Sharma *et al.*, "DNNWEAVER: From high-level deep network models to FPGA acceleration," in *Proc. Workshop Cogn. Archit.*, 2016, pp. 1–6.

[34] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, Taipei, Taiwan, 2016, p. 22.

[35] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A framework for generating high throughput CNN implementations on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2018, pp. 117–126.

[36] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Taipei, Taiwan, 2016, pp. 1–12.

[37] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *Proc. 55th Annu. Design Autom. Conf.*, San Francisco, CA, USA, 2018, p. 20.

[38] B. McDanel, S. Teerapittayanon, and H. T. Kung, "Embedded binarized neural networks," in *Proc. Int. Conf. Embedded Wireless Syst. Netw.*, 2017, pp. 168–173.

[39] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Boston, MA, USA, 2015, pp. 1–9.

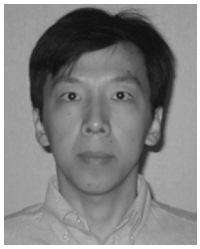
[40] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Honolulu, HI, USA, 2017, pp. 4700–4708.

[41] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[42] C. Burges *et al.*, "Learning to rank using gradient descent," in *Proc. 22nd Int. Conf. Mach. Learn. (ICML)*, 2005, pp. 89–96.

[43] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. Design Autom. Conf.*, Austin, TX, USA, 2017, pp. 1–6.

[44] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, Napa, CA, USA, 2017, pp. 101–108.
[45] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-S. Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integration*, vol. 62, pp. 14–23, Jun. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167926017304777
[46] X. Zhang *et al.*, "DNNExplorer: A framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator," in *Proc. 39th Int. Conf. Comput.-Aided Design*, 2020, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3400302.3415609
[47] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 11, pp. 2072–2085, Nov. 2019.
[48] Xilinx Inc. (2021). *DPU for Convolutional Neural Network*. [Online]. Available: https://www.xilinx.com/products/intellectual-property/dpu.html
[49] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Cuuran, 2012, pp. 1097–1105.
[50] Xilinx Inc. (2021). *Xilinx Power Estimator*. [Online]. Available: https://www.xilinx.com/products/technology/power/xpe.html
[51] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2017, pp. 35–44.
[52] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2016, pp. 16–25.
[53] Xilinx Inc. (2019). *Xilinx Power Estimator*. [Online]. Available: https://github.com/Xilinx/CHaiDNN
[54] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE Int. Symp. Field Program. Custom Comput. Mach.*, Napa, CA, USA, 2017, pp. 152–159.
[55] Advanced Micro Devices Inc. (2019). *AMD MIOpen*. [Online]. Available: https://gpuopen.com/compute-product/miopen
[56] J. Fang, H. Fu, W. Zhao, B. Chen, W. Zheng, and G. Yang, "swDNN: A library for accelerating deep learning applications on sunway taihulight," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Orlando, FL, USA, 2017, pp. 615–624.
[57] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proc. 24th Symp. Principles Pract. Parallel Program.*, 2019, pp. 300–314.
[58] C. Hong *et al.*, "Efficient sparse-matrix multi-vector product on GPUs," in *Proc. 27th Int. Symp. High Perform. Parallel Distrib. Comput.*, 2018, pp. 66–79.
[59] L. Ning and X. Shen, "Deep reuse: Streamline CNN inference on the fly via coarse-grained computation reuse," in *Proc. ACM Int. Conf. Supercomput.*, 2019, pp. 438–448.
[60] L. Ning, H. Guan, and X. Shen, "Adaptive deep reuse: Accelerating CNN training on the fly," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Macao, China, 2019, pp. 1538–1549.
[61] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet another SPMV framework on GPUs," *Acm Sigplan Notices*, vol. 49, no. 8, pp. 107–118, 2014.
[62] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "GRNN: Low-latency and scalable RNN inference on GPUs," in *Proc. 14th EuroSys Conf.*, 2019, p. 41.
[63] Y. Hu *et al.*, "Bitflow: Exploiting vector parallelism for binary neural networks on CPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Vancouver, BC, Canada, 2018, pp. 244–253.
[64] F. Kjolstad, W. Ahrens, S. Kamil, and S. Amarasinghe, "Tensor alge-bra compilation with workspaces," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, Washington, DC, USA, 2019, pp. 180–192.
[65] P. Roy, S. L. Song, S. Krishnamoorthy, A. Vishnu, D. Sengupta, and X. Liu, "NUMA-Caffe: NUMA-aware deep learning neural networks," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 2, p. 24, 2018.
[66] D. Grubic, L. Tam, D. Alistarh, and C. Zhang, "Synchronous multi-GPU training for deep learning with low-precision communications: An empirical study," in *Proc. EDBT*, 2018, pp. 145–156.
[67] R. Dathathri *et al.*, "CHET: An optimizing compiler for fully-homomorphic neural-network inferencing," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, 2019, pp. 142–156.
[68] R. Istrate, A. C. I. Malossi, C. Bekas, and D. Nikolopoulos, "Incremental training of deep convolutional neural networks," 2018. [Online]. Available: arXiv:1803.10232.
[69] J. Kang, K. Chung, Y. Yi, and S. Ha, "NNsim: Fast performance estimation based on sampled simulation of GPGPU kernels for neural networks," in *Proc. 55th Annu. Design Autom. Conf.*, 2018, p. 176.
[70] J. Liu, X. He, W. Liu, and G. Tan, "Register-aware optimizations for parallel sparse matrix–matrix multiplication," *Int. J. Parallel Program.*, vol. 47, no. 3, pp. 403–417, 2019.



Yun Liang (Senior Member, IEEE) received the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2010.

He is an Associate Professor (with tenure) with the School of EECS, Peking University (PKU), Beijing, China. He worked as a Research Scientist with the University of Illinois at Urbana–Champaign, Champaign, IL, USA, before he joins PKU. His research focuses on heterogeneous computing (GPUs, FPGAs, and ASICs) for emerging applications, such as AI and big data, computer architecture, compilation techniques, programming model and program analysis, and embedded system design.

Qingcheng Xiao received the B.S. degree from the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, in 2016, and the Ph.D. degree from the Center for Energy-Efficient Computing and Applications, Peking University, in 2021.

He has built several automated tools, including a co-design framework for agile co-design, a hardware convolution library, and a hardware generator for DNNs. His research interest is architecture and software co-design for AI chips.

Liqiang Lu received the B.S. degree from the Institute of Microelectronics, Peking University, Beijing, China, in 2017, where he is currently pursuing the Ph.D. degree with the School of EECS.

His research focuses on algorithm-level and architecture-level optimizations of FPGA for machine learning applications.

Jiaming Xie received the B.S. degree from the Institute of Microelectronics, Peking University, Beijing, China, in 2018, where he is currently pursuing the Ph.D. degree with the School of EECS.

His research focuses on GPU programming and system-level optimization for FPGA cluster.