FCNNLib: An Efficient and Flexible Convolution Algorithm Library on FPGAs

Qingcheng Xiao, Liqiang Lu, Jiaming Xie, and Yun Liang†
Center for Energy-Efficient Computing and Applications, Department of CS, Peking University
Email: {walkershaw, luliqiang, jmxie, ericlyun}@pku.edu.cn

***Abstract*—Convolutions can be implemented with different algorithms, which are diverse in arithmetic complexity, resource requirement, etc. Multiple algorithms can share the FPGA resources spatially as well as temporally, introducing either reconfiguration overhead or resource underutilization. In this paper, we propose an efficient library `FCNNLib` to coordinate multiple convolution algorithms on FPGAs. We develop three scheduling techniques: spatial, temporal, and hybrid, which exhibit different trade-offs in latency and throughput. We also expose a set of interfaces to arm the users. Experiments using modern CNNs demonstrate `FCNNLib` achieves up to 1.315X latency improvement compared with dedicated accelerators and 1.755X energy efficiency improvement compared with cuDNN.**

I. Introduction

With the prevalence of deep convolutional neural networks (CNNs), there is an increasing demand for accelerating convolutions on hardware. Different algorithms for the essential convolution operation in CNNs have been studied [1]. These algorithms include conventional, general matrix-matrix multiplication (GEMM), Winograd, and Fast Fourier Transformation (FFT) algorithms. The conventional algorithm is performed on the original features, while the other three algorithms transform data into 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 Winograd algorithm in cuDNN, the number of multiplications in VGGNet [2] can be reduced to half of the conventional algorithm, leading to about 2.7X inference latency speedup. Due to the importance of convolution operations, highly optimized convolution libraries supporting different algorithms such as Arm Compute Library, MKL-DNN, and cuDNN are commonplace for CPU and GPU platforms and are widely used in deep learning frameworks, such as Tensorflow [3], PyTorch [4].

However, such systematic library support of different convolution algorithms is not quite here yet for FPGAs, in large part because FPGAs are highly reconfigurable and difficult to program. First, programmers do not yet have reliable intuition about how to choose and implement multiple convolution algorithms. 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; temporal sharing by reconfiguring the FPGAs to implement different algorithms over time. There exhibit different trade-offs in latency and throughput. The diversity of CNN models adds further complications as different models or different layers of the same models may favour different algorithms and call for different scheduling techniques. Second, FPGAs programming remains to be a significant challenge for library developers. To maximize algorithm performance, programmers need to extensively restructure the source code to realize the unique hardware features.

In this paper, we propose `FCNNLib`, an efficient and flexible convolution algorithm library for CNN inference on FPGAs. We first 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 under-utilization issues. Moreover, `FCNNLib` provides optimized algorithm IPs and high-level interfaces to facilitate the users to explore different algorithms and schedulings for a variety of CNN models. We make the following contributions.

- We propose a hardware library `FCNNLib`, which provides efficient and flexible implementations of multiple convolution algorithms for inference on FPGAs.
- We develop three multi-algorithm scheduling techniques, including spatial, temporal, and hybrid scheduling.
- We provide optimized IPs and a succinct set of interfaces to facilitate the users to explore the library.

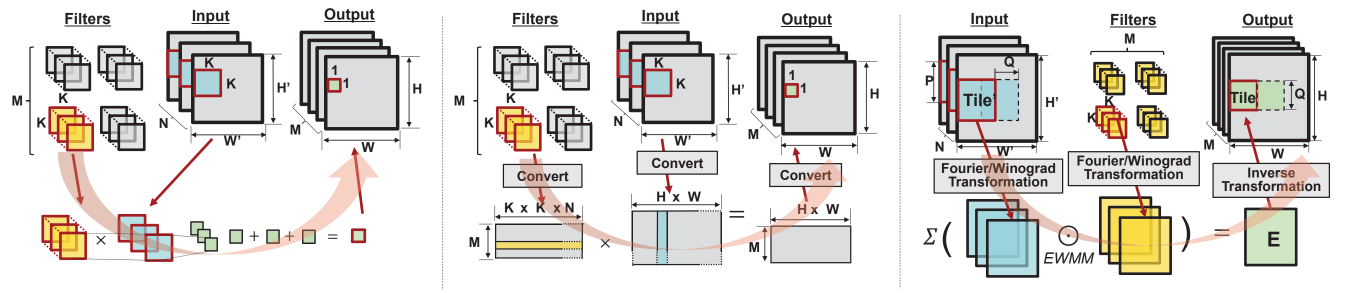
Experiments using state-of-the-art CNN models on Xilinx FPGAs demonstrate that designs offered by `FCNNLib` achieve up to 1.315X latency improvement and 1.292X DSP efficiency results compared with dedicated accelerators. Compared with cuDNN, `FCNNLib` provides up to 1.755X energy efficiency.

II. Background and Motivation

A. Convolution Algorithm Basics

Convolution in CNNs is to shift a group of 3D 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. To extract features, each filter convolves with the input tensor at stride S to obtain one feature map with size $H \times W$ in the output tensor. In this way, after convolving all filters, M output feature maps

†Corresponding Author



(a) conventional algorithm (b) GEMM algorithm (c) Winograd and FFT algorithms

Fig. 1: Dataflow of different convolution algorithms.

are generated. The following equation details the convolution operation for each output element:

$$out_{m,h,w} = \sum_{n=1,r=1,c=1}^{N,K,K} in_{n,h\times S+r,w\times S+c} \times filter_{m,r,c,n}$$

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

Convolutions can be converted to matrix multiplications. As Figure 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 corresponds to an output element. These tiles are also flattened into columns. In this way, multiplying a row and a column is equivalent to the convolution operation for an output element. In this paper, we refer to this implementation as **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 element-wise multiplication (EWMM), and finally transform the EWMM results back to the original output tile. Figure 1(c) illustrates their dataflows.

B. *Convolution Algorithms on FPGAs*

Convolution algorithms implemented on FPGAs are diverse in arithmetic complexity (the number of multiplications), resource (memory, logic, and compute resources), and adaptability to different workloads [5].

Table I lists convolution layers belonging to ResNet and DenseNet. We observe that layers favour different convolution algorithms on FPGAs. Conventional and GEMM achieve consistent performance for convolutions with different workloads. However, the benefits of Winograd and FFT algorithms fades as the strides increase. The reason is that Winograd and FFT algorithms organize data as tiles. When the strides are larger than 1, they process these convolutions as if their strides were 1 and screen out the valid outputs after computation. For instance, the optimal algorithm for one layer (kernel size 3, stride 1) is Winograd, while the optimal algorithm for another layer (kernel size 7, stride 2) is conventional. Using the best algorithm for a single layer provides the ideal peak performance. However, if we choose one fixed algorithm

TABLE I: Layer preference on convolution algorithms. The target platform is Xilinx ZC706.

Network	Layer Para. ($kernelSize$, $stride$)	Oper. Ratio	Optimal Algorithm	Performance (GOPS)	
				Single Algo. Layer Peak	Single Algo. Overall
ResNet [6]	(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
DenseNet [7]	(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

for the entire model, the overall performance drops sharply compared to the layer peak performance. For instance, the overall performance drops more than 3X (548.9 vs 178.6) compared to layer performance if we use 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. To this end, we design an efficient library that provides a variety of algorithms to implement convolutions on FPGAs.

III. Multi-algorithm Scheduling

One of the essential challenges of developing the library is the way to schedule multiple convolution algorithms. We propose three scheduling techniques. **Spatial scheduling** lets each algorithm occupy partial on-chip resources and maintains the same architecture through the whole CNN inference. **Temporal scheduling** dynamically swaps algorithms at runtime according to the layer parameters. To enable this scheduling, we reconfigure FPGAs to switch algorithm implementations. **Hybrid scheduling** partitions CNN models into several groups that occupy the hardware resources in time-sharing fashion. Within a group, we allocate partial resources to each layer for implementing its compute unit as in spatial scheduling. We discuss the scheduling algorithms and trade-offs as below.

A. *Spatial Scheduling*

In spatial scheduling, we partition hardware resources (resource partition) for convolution algorithms. CNN models have distinct convolution workloads and call for customized spatial scheduling designs. If we process CNN workloads layer by layer and assign a convolution workload to only one convolution algorithm, the compute units of other algorithms are idle, leading to potential low utilization. Hence, we also partition a convolution workload (workload partition) and assign sub-workloads to all employed algorithms. Solving the resource partition and workload partition problems together results in enormous design space. Hence, we divide and conquer spatial scheduling through two stages.

Resource Partition. A resource partition solution results in an architecture where each algorithm is implemented with its resource partition. Convolution workloads with various strides and filter sizes may favour different convolution algorithms. When each workload in the CNN is processed by its favourite algorithm compute unit in the architecture, the execution time represents a reasonable inference latency upper-bound. We denote this upper-bound as the ceiling inference latency.

Given a target CNN, we explore different resource partition solutions and fix the architecture with the best resource partition. We evaluate each partition solution with the ceiling inference latency of the architecture associated with the partition solution. We use a simulated annealing algorithm to search the minimal ceiling inference latency, namely the best partition solution, as shown in Algorithm 1. We first denote a partition solution in the algorithm. Since FPGA resources are multi-dimensional, including BRAMs, DSPs, LUTs, etc., we denote the hardware resource constraints R as a vector $\langle \#BRAMs, \#DSPs, \dots \rangle$ where each element represents the total amount of a type of resources. For convolution algorithm $algo$, its resource partitioning R_{algo} is also a vector similar to R. Accordingly, a partition solution PTN' can be represented as a long vector $\langle R_{conven.}$, R_{GEMM}, $R_{Wino.}$, $R_{FFT}\rangle$. Then we model an architecture where each algorithm is implemented within its resource partitioning (line 5). We calculate the ceiling inference latency of this architecture based on performance models (line 6). If the ceiling latency is less than the previous one, we accept the partition solution PTN' as the current solution with a possibility (line 7-9). A new partition solution is generated based on the current solution and evaluated (line 4). This process continues until the ceiling latency converges, or the iteration number exceeds the given maximum max_iter. In the end, Algorithm 1 returns the best partition solution with which we build a multi-algorithm accelerator.

Workload Partition. The goal of partitioning convolutions is to balance the workloads among different algorithms. We employ feature-based and channel-based partition methods, as shown in Figure 2(a) and (b). 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. Our 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. The feature-based method and channel-based method can work in a complementary manner. There also exist other partition methods, such as assigning each algorithm with the workload of an entire layer. However, this method is hard to achieve workload balance and is not general to traditional sequential CNNs.

Algorithm 1: Resource partition algorithm.

```
   Input: $CNN\_model$, $R$, $max\_iter$
   Output: $final\_PTN$
1  $iter \leftarrow 0$, $PTN \leftarrow$ initial resource partition solution
2  # $PTN$ is a vector $\langle R_{conven.}, R_{GEMM}, R_{Wino.}, R_{FFT}\rangle$
3  while $iter \leq max\_iter$ do
4      $PTN' \leftarrow$ generate a new solution based on $PTN$
5      $arch' \leftarrow$ model an architecture associated with $PTN'$
6      $ceiling\_latency' \leftarrow$ evaluate the ceiling inference
         latency of $arch'$ when processing $CNN\_model$
7      if $ceiling\_latency' < ceiling\_latency$ then
8          # $ceiling\_latency$ is associated with $PTN$
9          $PTN \leftarrow PTN'$, with certain possibility
10 $final\_PTN \leftarrow$ the solution with the min ceiling latency
11 return $final\_{P}TN$
```

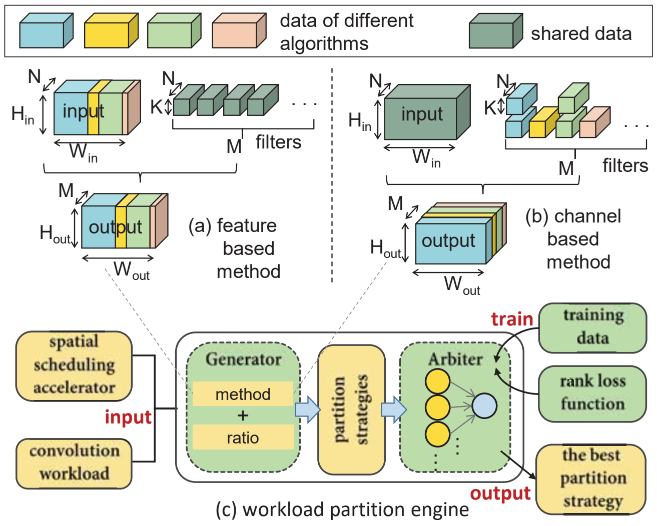


Fig. 2: Workflow of the workload partition engine.

The chosen partition method and ratio depend on both workload itself and the accelerator generated for spatial scheduling. Building performance models for all generated accelerators is impossible. Hence, we develop a machine-learning-based engine to partition workloads, as shown in Figure 2(c). The engine consists of a partition generator and an arbiter. Given a workload, the generator proposes multiple workload partition strategies (method and ratio combinations). The arbiter selects a strategy among them according to their relative execution time. Here we use the relative execution time instead of the absolute runtime latency, as we only care about the relative merits of partition strategies proposed by the generator.

Specifically, we implement the arbiter with a multi-layer perceptron (MLP). Its inputs are the convolution workload and the partition strategy to be evaluated. It outputs the relative execution time of the input strategy. The relative time prediction is enabled by employing a rank loss function [8] as the training objective function. The loss function obj is:

$$obj = \sum_{a,b} log(1 + e^{-sign(lat_a - lat_b)\times(pred_a - pred_b)})$$

where a and b are two partition strategies, $sign$ is the Signum function, lat is the actual execution latency, and $pred$ is the relative latency predicted by the MLP. We collect the training data (actual execution latency results) by launching a batch of randomly partitioned convolutions to the accelerator.

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 trade-off 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:

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

where $T(i,j)$ represents the minimal latency for layers i to j after considering reconfiguration. $T_{one}(i,j)$ 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 $T_{one}(i,j)$, 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 $T_{one}(i,j)$.

C. Hybrid Scheduling

Hybrid scheduling employs a compute unit for each layer in the given CNN. The compute unit area, in general, depends on the layer compute complexity. However, the limited hardware resources cannot accommodate all the hundreds of layers in modern complex CNNs like 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 [9] to improve overall throughput.

Similar to temporal scheduling, there exists a trade-off 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 trade-off, we use the same dynamic programming algorithm used in temporal scheduling, as Equation 1 shows. However, here $T_{one}(i,j)$ denotes the latency of layers from i to j when they are organized as a group. To obtain $T_{one}(i,j)$, 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 i_{th} 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

Algorithm 2: Group architecture 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 $new\_arch.lat \ge opt\_arch.lat$ then
17                 break
18             if MeetConstraints($new\_arch.res$) then
19                 Config($new\_arch$, $i+1$, $j$)
```

with the performance and resource models (lines 12-13). 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 j_{th} layer is reached, we update the current best group latency and architecture if necessary. $T_{one}(i,j)$ 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.

IV. FCNNLib Implementation

We develop IPs in high-level synthesis and optimize them with loop transformations, including tiling, interchange, pipelining, and unrolling. Directives are properly placed to maximize algorithm performance. `FCNNLib` also provides performance and resource models for each IP.

Listing 1: Example of deploying ResNet with `FCNNLib`.

```
Step 1: Generate a hardware design on ZC706 FPGA
params = getParams(ZC706_resource, ResNet, Spatial)
design = configIPs(params)
Step 2: Schedule multiple algorithms on FPGAs
for wl in ResNet:# wl: workload
# partition each workload and assign them to algorithms
 wl.sub_wls = balanceWorkload(wl, design)
foreach input image:
 for wl in ResNet: # execute the model layer by layer
      with balanced sub-workloads
  wl.output = scheduleAlgo(design, wl.sub_wls,
                 wl.input, Spatial)
```

We design a set of high-level interfaces for library users. When using `FCNNLib`, there are two steps: hardware design generation and multi-algorithm 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 IP. With

TABLE II: Comparison with Previous FPGA accelerators.

Model	ResNet-152				DenseNet-161		DQN
Work	[12]	[13]	FCNNLib spatial		FCNNLib spatial		FCNNLib spatial
Platform	ZC706	VU9P	ZC706	VU9P	ZC706	VU9P	ZC706
Frequency (MHz)	125	200	200	200	200	200	200
Precision	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed
conventional	✓	✓					
GEMM			✓	✓	✓	✓	✓
Winograd			✓	✓	✓	✓	
FFT							✓
Latency (ms)	156.4	17.34	118.9	14.6	68.5	14.4	0.05
Throughput (GOPS)	188.18	1463	190.19	1547.84	209.17	996.6	108.8
DSP Efficiency (GOPS / DSP)	0.209	0.357	0.270	0.228	0.298	0.229	0.273
Power (W)	-	-	5.92	32.2	5.92	25.2	5.55

these parameters, users can instantiate and integrate IPs 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 partition engine. Moreover, we provide *autoScheduling* interface, which automatically explores the algorithm and scheduling combinations and returns a design with the best performance.

Listing 1 is an example of deploying ResNet with spatial scheduling on the Xilinx ZC706 board. Subject to resource constraints, *getParams* returns algorithm parameters leading to the highest performance. The resource constraints are obtained by Algorithm 1 in spatial scheduling. FCNNLib generates a ResNet accelerator after instantiating IPs (line 3). For each convolution workload in ResNet, *balanceWorkload* interface partitions it into sub-workloads specific to the ResNet accelerator (line 7). Last, *scheduleAlgo* interface launches the sub-workloads on the accelerator and collects results. As for temporal and hybrid schedulings, *scheduleAlgo* would also reconfigure FPGAs when necessary.

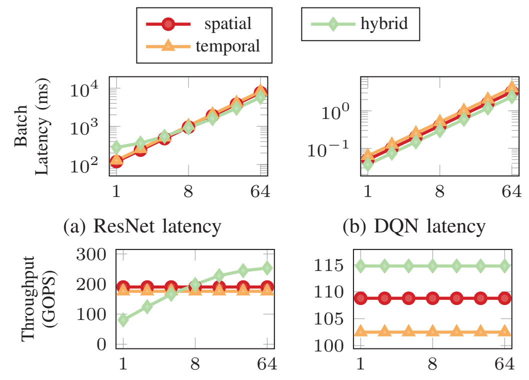
V. Experiments

A. Experimental Setup

To demonstrate the efficiency of FCNNLib, we integrate it into PyTorch [4] and evaluate it with widely used CNNs, including ResNet, DenseNet, and DQN [10]. We treat fully connected layers as convolutions with 1×1 filters and fuse activation functions with convolutions. We employ FPGAs deployed in both embedded and cloud scenarios. Xilinx ZC706 board is an embedded SoC platform, consisting of one XC7Z045 FPGA chip, dual ARM Cortex-A9 CPUs, and 1 GB DDR3 memory. VU9P board is a PCIe-based board that has been used in AWS F1 instance. For both platforms, we set the frequency as 200 MHz and use a 16-bit fixed-point data type. We use Xilinx Vivado SDx(v2018.2) [11] for design synthesis.

B. FPGA Accelerator Comparison

Prior techniques [12], [13] shown in Table II proposed dedicated optimization for one convolution algorithm. They solely employ conventional convolution algorithm and focus on optimizing inference latency with batch size as 1. For fair comparisons, we use spatial scheduling in FCNNLib to generate designs targeting low latency.



(a) ResNet latency (b) DQN latency
(c) ResNet throughput (d) DQN throughput

Fig. 3: Scheduling comparisons on ZC706. X-axis: batch size.

For less regular modern CNNs, FCNNLib prefers to integrate a high-performance algorithm (Winograd or FFT) with a more general one such as conventional or GEMM algorithm. As shown, Winograd and GEMM algorithms are employed for ResNet and DenseNet, while FFT and GEMM algorithms are used for DQN. In ResNet and DenseNet, the most common convolution workloads are 3×3 convolutions and 1×1 convolutions. Winograd algorithm shows high performance for the 3×3 convolutions, and GEMM is the optimal algorithm for 1×1 convolutions. As a result, FCNNLib spatial scheduling lets Winograd collaborates with GEMM. It determines the resource partition solutions for ResNet and DenseNet, respectively, and balances the runtime workloads of Winograd and GEMM algorithm. Overall, for ResNet, FCNNLib achieves 118.9 ms latency and 190.19 GOPS throughput on the ZC706 board, while 14.6 ms latency and 1547.84 GOPS throughput on VU9P device. Hence, by leveraging multiple algorithms in ResNet, FCNNLib spatial scheduling achieves up to 1.315X latency improvement compared with [12], 1.292X DSP efficiency improvement compared with [13]. DQN consists of three 5×5 convolutions with strides being 2 and a final fully connected layer. FFT algorithm is optimal for the 5×5 convolutions. FCNNLib spatial scheduling achieves 0.05 ms latency and meets real-time requirements.

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 Figure 3.

For ResNet, spatial scheduling generates a design that is 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. Hence it generates a ResNet design employing only Winograd algorithm. Its latency results are also linear with the batch size and are higher than spatial scheduling latency. However, hybrid scheduling for ResNet always performs reconfiguration since ResNet consists of hundreds of layers and requires hundreds of compute units. Such many compute units cannot

TABLE III: Cross-platform comparison.

CNN	Platform	Device	Library	Precision	Latency (ms)	Throughput (GOPS)	Power (W)	Energy EFF. (GOP/J)
DenseNet, batch size = 16	FPGA	VU9P	FCNNLib hybrid	16-bit fixed	198.24	1158.3	26.4	43.88
	GPU	P100	cuDNN	16-bit float	108.7	2112.14	84.5	25.00
ResNet, batch size = 1	FPGA	ZC706	FCNNLib spatial	16-bit fixed	118.9	190.19	5.92	32.13
	GPU	TX1	cuDNN	16-bit float	118.11	191.44	9.8	19.54

be implemented together on a device. It takes six reconfigurations in hybrid scheduling and leads 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 Figure 3(c). DenseNet shows latency and throughput results similar to ResNet. 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 compute unit with customized algorithms and parameters.

Depending on a series of factors including model topology, resources and reconfiguration overhead of the platform, available batch size, the three scheduling techniques vary in latency and throughput improvements and require careful selection.

D. Library Comparison

Then we compare our hardware library `FCNNLib` for FPGA platforms with the software library cuDNN 9.0 for GPUs. We let `FCNNLib` and cuDNN automatically select algorithms through *autoScheduling* and *cudnnGetConvolutionForwardAlgorithm* interfaces, respectively. We use 16-bit data types for FPGAs and GPUs. For DenseNet, we use the VU9P FPGA board and NVIDIA P100 GPU. For ResNet, we use the ZC706 FPGA board and NVIDIA Jetson TX1 GPU board. We list the comparison results in Table III. `FCNNLib` provides consistently better energy efficiency in the two cases compared with cuDNN. `FCNNLib` chooses hybrid scheduling in the DenseNet case and achieves 1.755X energy efficiency compared with cuDNN. Spatial scheduling is chosen for the ResNet case. `FCNNLib` provides comparable throughput and 1.644X energy efficiency compared with cuDNN.

VI. Related Work

The implementation of convolution algorithms on FPGAs has been studied in a number of previous works [14], [15]. [5] implement Winograd algorithm on FPGAs. [16] has implemented FFT algorithm for both CNN training and inference. [17] applies GEMM algorithm to CNN acceleration. As for conventional algorithm, most of the previous FPGA CNN accelerators are based on it [18]. CHaiDNN [19] is an HLS based DNN Library implementing conventional algorithm solely. 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 data movement and parallelism strategy [20]–[22]. They are orthogonal to our multi-algorithm scheduling efforts made to `FCNNLib`.

VII. Conclusion

In this paper, we propose a convolution algorithm library `FCNNLib` consisting of three multi-algorithm schedulings, highly optimized algorithm IPs, and programming interfaces. We explore spatial, temporal, and hybrid schedulings with the help of scheduling algorithms. We use resource partition and workload partition for spatial scheduling. For temporal and hybrid schedulings, we develop dynamic programming algorithms to determine the schedule timing and method. `FCNNLib` also provides highly optimized algorithm IPs and a series of programming interfaces to ease the FPGA programming hurdle. Experiments using modern CNNs demonstrate that `FCNNLib` achieves significant energy efficiency and latency improvement compared with dedicated accelerators.

Acknowledgment

This work is supported by Beijing Natural Science Foundation (No. JQ19014, L172004) and Beijing Academy of Artificial Intelligence (BAAI).

References

[1] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *CVPR*, 2016.
[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[3] M. Abadi and et al., "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.
[4] A. Paszke and et al., "Automatic differentiation in pytorch," in *NIPS-W*, 2017.
[5] L. Lu and et al., "Evaluating fast algorithms for convolutional neural networks on fpgas," in *FCCM*, 2017.
[6] K. He and et al., "Deep residual learning for image recognition," in *CVPR*, 2016.
[7] G. Huang and et al., "Densely connected convolutional networks," in *CVPR*, 2017.
[8] C. Burges and et al., "Learning to rank using gradient descent," in *ICML*, 2005.
[9] Q. Xiao and et al., "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas," in *DAC*, 2017.
[10] V. Mnih and et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015.
[11] Xilinx Inc. (2019) Xilinx vivado high-level synthesis. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html
[12] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas," *IEEE Trans. Neural Netw. Learn. Syst.*, no. 99, 2018.
[13] X. Wei and et al., "Tgpa: Tile-grained pipeline architecture for low latency cnn inference," in *ICCAD*, 2018.
[14] Q. Xiao and Y. Liang, "Fune: An fpga tuning framework for cnn acceleration," *IEEE Design & Test*, 2019.
[15] Q. Xiao and et al., "Zac: Towards automatic optimization and deployment of quantized deep neural networks on embedded devices," in *ICCAD*, 2019.
[16] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *FPGA*, 2017.
[17] N. Suda and et al., "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA*, 2016.
[18] M. Alwani and et al., "Fused-layer cnn accelerators," in *MICRO*, 2016.
[19] Xilinx Inc. (2019) Xilinx deep neural network library. [Online]. Available: https://github.com/Xilinx/CHaiDNN
[20] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in dcnn accelerators," in *HPCA*, 2019.
[21] Y. Ma and et al., "End-to-end scalable fpga accelerator for deep residual networks," in *ISCAS*, 2017.
[22] S. Yin and et al., "A high throughput acceleration for hybrid neural networks with efficient resource management on FPGA," *TCAD*, 2018.