OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks

Yunxuan Yu, Chen Wu, Tiandong Zhao, Kun Wang, *Senior Member, IEEE*, and Lei He, *Senior Member, IEEE*

***Abstract*— Field-programmable gate array (FPGA) provides rich parallel computing resources with high energy efficiency, making it ideal for deep convolutional neural network (CNN) acceleration. In recent years, automatic compilers have been developed to generate network-specific FPGA accelerators. However, with more cascading deep CNN algorithms adapted by various complicated tasks, reconfiguration of FPGA devices during runtime becomes unavoidable when network-specific accelerators are employed. Such reconfiguration can be difficult for edge devices. Moreover, network-specific accelerator means regeneration of RTL code and physical implementation whenever the network is updated. This is not easy for CNN end users. In this article, we propose a domain-specific FPGA overlay processor, named OPU to accelerate CNN networks. It offers software-like programmability for CNN end users, as CNN algorithms are automatically compiled into executable codes, which are loaded and executed by OPU without reconfiguration of FPGA for switch or update of CNN networks. Our OPU instructions have complicated functions with variable runtimes but a uniform length. The granularity of instruction is optimized to provide good performance and sufficient flexibility, while reducing complexity to develop microarchitecture and compiler. Experiments show that OPU can achieve an average of 91% runtime multiplication and accumulation unit (MAC) efficiency (RME) among nine different networks. Moreover, for VGG and YOLO networks, OPU outperforms automatically compiled network-specific accelerators in the literature. In addition, OPU shows 5.35× better power efficiency compared with Titan Xp. For a real-time cascaded CNN networks scenario, OPU is 2.9× faster compared with edge computing GPU Jetson Tx2, which has a similar amount of computing resources.**

***Index Terms*— Convolutional neural network (CNN) overlay processor, field-programmable gate array (FPGA) acceleration, hardware–software codesign.**

I. Introduction

Field-programmable gate array (FPGA) acceleration for convolutional neural networks (CNNs) has drawn much attention in recent years [1]–[11]. Well-designed FPGA accelerator for CNN can leverage the full capacity of parallelism in network computation to achieve low latency and high performance. Moreover, FPGA comes with the advantage of reconfigurability that enables fast adoption to new CNN architectures. In addition, FPGA has higher energy efficiency compared to CPU or GPU.

Manuscript received February 22, 2019; revised June 29, 2019; accepted August 15, 2019. *(Corresponding authors: Kun Wang; Lei He.)*

The authors are with the Department of Electrical and Computer Engineering, University of California at Los Angeles, Los Angeles, CA 90095 USA (e-mail: yunxuan.yu@hotmail.com; chenwu1989@g.ucla.edu; zhaotiandong@ucla.edu; wangk@ucla.edu; lei.hexun@gmail.com).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2019.2939726

Implementing a high-performance FPGA accelerator can be time-consuming as it normally involves parallel architecture exploration, memory bandwidth optimization, area and timing tuning, as well as software–hardware interface development. This leads to the development of automatic compilers for FPGA CNN accelerators [8]–[11], where the hardware description code of target accelerators can be generated automatically based on parametric templates, and the design space exploration is simplified to parameter optimization with regard to network structure and hardware resource constraints.

However, some disadvantages still remain. While RTL code can be generated as the final output of autocompilers, it still takes logic synthesis, placement, and routing to obtain the final bitstream. Furthermore, the resulting design may fail timing requirements. Instead of fixing timing failing paths as in regular FPGA design process, in autocompiler process, one can only adjust module parameters or relax timing constraints at the expense of performance degradation. Moreover, complex deep learning tasks usually involve cascaded network flow. It would be inefficient, and sometime impossible for edge computing to constantly reburn FPGA for different networks during runtime.

In this article, we propose an RTL-based hand-coded FPGA overlay domain-specific processor unit (OPU) with software-programmability and fast compilation time, targeting at general CNN accelerations. The processor accepts network architecture from deep learning framework such as Tensorflow [12]. Each time a new network configuration is given, instead of regenerating a new accelerator, we compile the network into instructions to be executed by OPU. OPU has fine-grained pipe-line, and it explores parallelism of different CNN architectures. This ensures an average of 91% runtime utilization of computing resources as shown by experiments on nine different network architectures, including YOLO [13], VGG [14], GoogleNet [15], and ResNet [16]. Moreover, superior power efficiency compared with Titan GPU (both batch $=$ 1 and batch $=$ 64) is observed for all network cases. In addition, for cascaded CNN networks to detect car license plate, OPU is 2.9× faster compared with edge computing GPU Jetson Tx2 with a similar amount of computing resources.

More specifying, the proposed overlay processor OPU has the following features.

1) *CPU/GPU Like User Friendliness:* As shown in Fig. 1, CNN network is compiled into instructions. This is done once for each network. Then, instructions are executed by OPU which is implemented on FPGA and fixed for

Fig. 1. OPU working flow.

all networks. The CNN algorithm developer does not need to deal with FPGA implementation.

2) *Domain-Specific Instruction Set:* Our instructions have optimized granularity, which is smaller than that in [17] to ensure high flexibility and computational unit efficiency, while a lot larger than those for CPU/GPU to reduce the complexity of compiler.
3) *FPGA-Based High-Performance Microarchitecture:* These architectures are optimized for computation, data communication and reorganization, which are controlled by parameter registers set directly by instructions.
4) *Compiler With Comprehensive Optimization:* Independent of microarchitecture, operation fusion is performed to merge or concatenate closely related operations to reduce computation time and data communication latency. Data quantization is also conducted to save memory and computation resources. Related to microarchitecture, compiler explores multiple degrees of parallelism to maximize throughput by slicing and mapping the target CNN to overlay architectures.

The rest of this article is organized as follows. Section II reviews the existing deep learning accelerator work. Section III describes the OPU instructions. Sections IV and V explain OPU microarchitecture and compiler, respectively. Section VI presents our experiment results on various state-of-the-art CNNs. Section VII concludes this article.

II. Related Work

Deep learning CNN acceleration by FPGA has been extensively studied, started with developing customized hardware accelerators for specific networks. Farabet *et al.* [1] used FPGA as a vector-based arithmetic unit, and implemented CNN mainly on a 32-bit soft processor. Ovtcharov *et al.* [2], Cadambi *et al.* [3], and Chakradhar *et al.* [4] designed specific accelerators for each layer of CNN. Zhang *et al.* [5] implemented Alexnet [18] on a VC707 board using high-level synthesis (HLS) and [6] hand-coded an RTL accelerator for VGGnet [19]. Suda *et al.* [7] used HLS to design accelerators for both Alexnet and VGGnet. The aforementioned work demonstrated FPGA's capability as a high performance CNN accelerator platform, but manually designing accelerator for each CNN was inefficient.

More recent work developed automatic compiler to implement CNN accelerators to FPGA. Sharma *et al.* [8] and Ma *et al.* [9] mapped CNN algorithms to a network of hand-optimized design templates, and gained performance comparable with hand-crafted accelerators. Zhang *et al.* [10] developed an HLS-based compiler with bandwidth optimization by memory access reorganization. Wei *et al.* [11] applied an systolic array architecture to achieve higher clock frequency. However, they all generate specific individual accelerators for each CNNs. This has high reengineering effort when the target CNN changes.

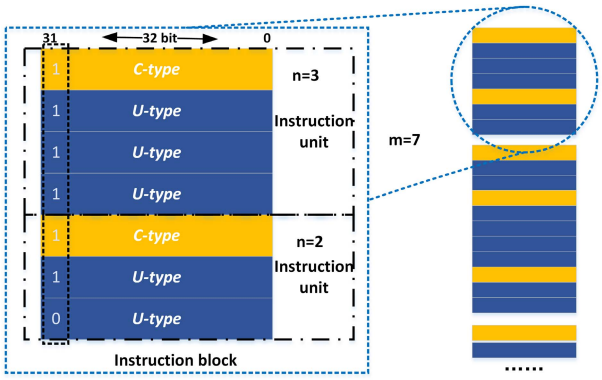


Fig. 2. Configuration of instruction block and instruction unit.

Most recently, Abdelfattah *et al.* [17] used FPGA overlay to implement CNN accelerators. While instructions are used to decrease the control logic overhead, they still reconfigure the overlay architecture to maximize performance for a specific CNN. Moreover, the granularity of their instructions is larger than that for our OPU. In [17], one block of about ten instructions is used for a whole subgraph defined as a list of chained functions, which is normally a single convolution layer and an optional pooling layer. In contrast, our OPU has the instruction set with smaller granularity (to be discussed in Section III) than that in [17]. Each typical operation in CNN inference is mapped to a specific type of instruction, resulting in high runtime efficiency. Moreover, different CNNs can be compiled and then executed without FPGA reconfiguration.

III. Instruction Set Architecture

Instruction set architecture (ISA) is the key to a processor. Our OPU is specific for CNN inference. We identify all the necessary operations during CNN inference and group them into different categories. Each category maps to one type of instruction with adjustable parameters for flexibility. Our instructions are 32 bit long and have complicated functions and variant runtimes (up to hundreds of cycles). CNN inference can be executed by OPU without a general-purpose processor such as CPU.

There are two types of instructions defined: Conditional instruction (C-type) and Unconditional instruction (U-type). C-type instruction specifies target operations and sets operation trigger conditions, while U-type instruction delivers corresponding operation parameters for its paired C-type. As shown in Fig. 2, one instruction unit contains one C-type instruction with $0 - n$ U-type instructions. This instruction block consisting of a number of basic units is fetched together and then is distributed to processing element (PE) modules. The least significant bit of instruction indicates the end of current instruction block when its value is 0.

A. Conditional Instruction

Conditional- or C-type instructions contains operation (OP) code and trigger condition. OP code identifies the target operation, while trigger condition defines when operation is ready to execute.

Six types of C-instructions are defined in the following, and each operates on a slice of data block.

1) *Memory Read* transforms the data from external memory to onboard memory. It operates in two modes to accommodate for different data read patterns. Received data will be reorganized and distributed to three destination buffers corresponding to the feature map (FM), kernel weighs, and instructions, respectively.
2) *Memory Write* sends the block of computational results back to the external memory.
3) *Data Fetch* performs data read from onboard FM and kernel buffer, then feeds to computation engine. Its working pattern can be flexibly adjusted by placing constraint parameters on row and column address counters, read strides, and data reorganize modes.
4) *Compute* controls all processing units (PEs). One PE computes the inner product of two 1-D vectors of length N, which is set to 16 in current microarchitecture implementation based on widely used CNNs families' architectures. This sufficiently guarantees the design space exploration for different networks. Results of PEs can be summed up in different modes based on parameter setting.
5) *Post Process* includes pooling, activation, data quantization, intermediate result addition, and residual operations. Selected combination of before-mentioned operations is executed when postprocess is triggered.
6) *Instruction Read* reads a new instruction block from instruction buffer and directs it to target operation modules.

Each aforementioned instruction leads one instruction unit. Instead of linking all the operation modules together in a fixed pipeline, we organize our operations in a dynamic pipeline fashion and each module is controlled by an individual instruction unit for more flexibility. For example, after one memory read is called for FM loading, multiple data fetch, and compute may be called to reuse loaded FM data. Then, at certain point during computation, memory read can be called again to replace kernel weights data (in the case where kernel size is large). When residual layer is encountered, memory read is called one extra time to load FM data from short cut [16] for post process. Individual control of each module by instruction greatly simplifies overall hardware control frame and enhances the architecture applicability to different network configurations.

To realize efficient instruction control, the trigger condition is employed, so instruction is not executed immediately upon read. In CNN inference flow, each operation depends on previous operations based on different operating patterns, which have limited variations. We design a trigger condition list for individual operation, then modify trigger condition index (TCI) by instruction at runtime to set module initiation dependence. Moreover, using a dependency-based execution strategy relaxes the order enforcement on instruction sequence. Cause memory related operations have the uncertainty in the execution time due to extra refresh latency. The resulting system has a simple instruction update scheme. Several instructions

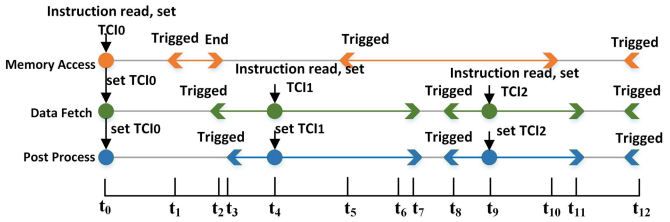


Fig. 3. Instruction execution order controlled by TCI.

executed at different time points can be put into the same instruction block and read at the same time. As shown in Fig. 3, after the first instruction read, initial TCI_0 is set at t_0 for all three operations. At t_1, memory access is triggered then executes for $t_1 - t_2$. Data fetch is triggered upon finishing memory operation and post process is trigged at t_3. Next instruction read that updates TCI_1 for Data fetch and post process can be performed at any time point between t_3 and t_7. Moreover, we store current TCI to avoid setting the same condition repeatedly when modules operate in one pattern consecutively (at time t_0 and t_5, memory access is triggered by the same TCI). This shortens the instruction sequence over $10\times$.

B. Unconditional Instruction

Unconditional- or U-type instruction provides operation related parameters and is generated on an updating demand-based scheme, as parameters are stored to reduce the total length of instruction sequence.

Several U-type instructions combined can update the complete parameters list for one C-type operation. However, in general, when operation pattern switches, only a subset of parameters is changed accordingly. Flexible combinations of U-type instructions can update necessary parameters with the minimum instruction cost. We group parameters that are closely related to each other and have similar updating rates in one U-type instruction. This reduces the possibility of loading futile instruction sections, thus reducing both storage space and communication power.

IV. Microarchitecture

Another challenge in OPU design is overlay microarchitecture design. The overlay microarchitecture needs to incur as less control overhead as possible while maintaining easily runtime adjustable and functionality. We design our modules to be parameter customizable, and switch modes at runtime based on parameter registers that directly accepts parameters provided by instructions. The computation engine explores multiple level of parallelisms that generalize well among different kernel sizes. Moreover, CNN operations categorized into the same group are reorganized and combined (discussed in Section III-A) so they can be accomplished by the same module to reduce overhead.

As shown in Fig. 4, the overlay microarchitecture can be decomposed into six main modules following the instruction architecture definition. Each module can be controlled by instruction to accomplish functionalities defined in Section III-A. In addition, four storage buffers (i.e., input FM buffer, kernel buffer, instruction buffer, and output buffer) are

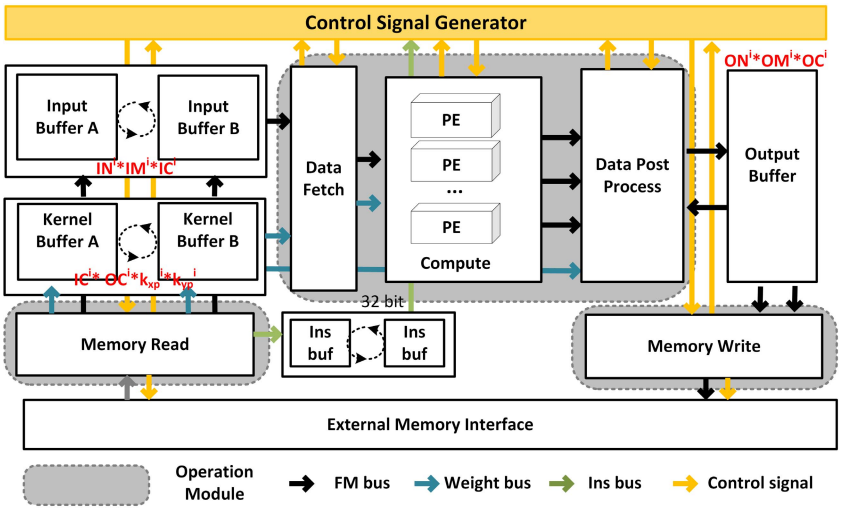


Fig. 4. Overview of microarchitecture.

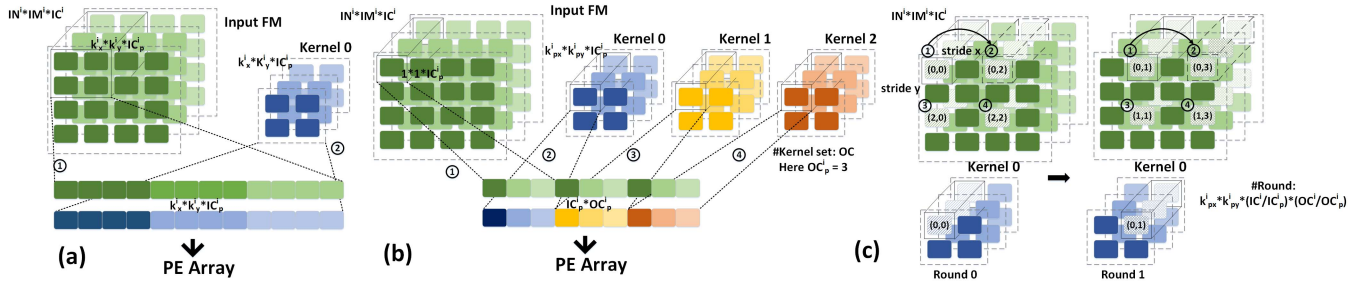


Fig. 5. (a) Conventional intrakernel-based parallelism. (b) OPU input–output channel-based parallelism. (c) FM data fetch pattern of OPU.

placed to cache local data for fast access. With most of the control flow embedded in instruction, overlay only handles the computation of one sub-FM block. If layer size is larger than the maximum block size allocated, the layer will be sliced into subblocks by compiler to fit into the hardware. The optimization of slicing scheme is discussed in Section V.

A. Computation Unit

How to utilize the same set of PE structures to accommodate for different layers is dominant in the design of CNN acceleration architecture. Conventional designs tend to explore parallelism within a single 2-D kernel, which is straightforward but comes with two disadvantages, i.e., complex FM data management and poor generalization among various kernel sizes. As shown in Fig. 5(a), expanding a $k_x * k_y$ kernel sized window of FM requires multiple data read from row and column directions within single clock cycle (step ①). This poses challenge on limited block ram bandwidth and generally requires extra complicated data reuse management (like line buffer) to accomplish. Furthermore, data management logic designed for one kernel size cannot be efficiently applied to another one. Similarly, PE architecture optimized for certain kernel size $k_x * k_y$ may not fit other sizes very well. This is why many conventional FPGA designs optimize their architecture on popular $3*3$ kernel and perform the best only on networks with pure $3*3$ layers.

To address this issue, we explore higher level of parallelism and leave 2-D kernel being computed sequentially. Fig. 5(b) explains how it works: at each clock cycle, a slice of input channel of depth IC_p^i with width and height as $1*1$ is read along with corresponding kernel elements. This fits natural data storage pattern and requires much smaller bandwidth. Parallelism is then implemented within input channel slice IC_p^i and output channel slice OC_p^i. Fig. 5(c) further shows the computation process. For round 0 cycle 0, input FM channel slice from position $(0, 0)$ is read. Then, we jump stride x ($x = 2$ is used as example here) and read position $(0, 2)$ in next cycle. Read operation continues until all pixels corresponding to kernel position $(0, 0)$ is fetched out and computed. Then, we enter round 1 and read starting from position $(0, 1)$ to get all pixels corresponding to kernel position $(0, 1)$. To finish computing this data block of size $\mathrm{IN}^i * \mathrm{IM}^i * \mathrm{IC}^i$ with current slice of kernel with size $k_{px}^i * k_{py}^i * \mathrm{IC}_p^i * \mathrm{OC}_p^i$ in kernel buffer, $k_{px}^i * k_{py}^i * (\mathrm{IC}^i/\mathrm{IC}_p^i) * (\mathrm{OC}^i/\mathrm{OC}_p^i)$ rounds are needed.

One implementation of the computation unit is shown in Fig. 6. One single multiplier unit (MU) computes two 8×8 multiplication with one of the inputs kept the same. This constraint comes from the DSP decomposition implementation. Each PE consists of 16 MUs followed by an adder tree structure, thus each PE equals 32 multiplication and accumulation units (MACs). For one of our implementations of OPU, we implement 32 PEs within the computation unit, and an adder tree with switch is implemented to sum up results from different group sizes of PEs. The outputs number choices include $[64, 32, 16, 8, 4, 2]$. This allows the computation unit

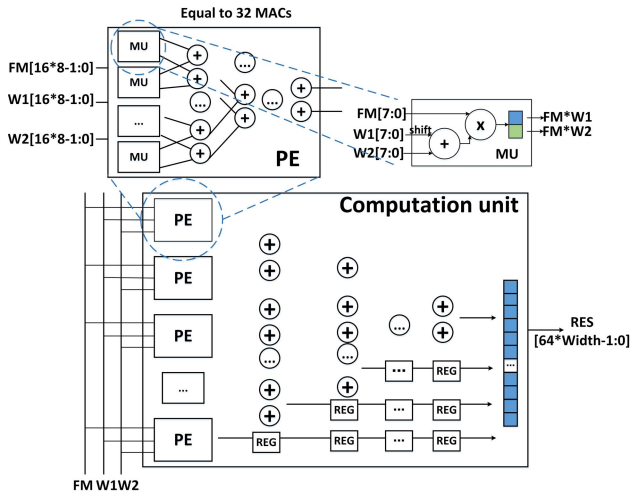


Fig. 6. Computation unit.

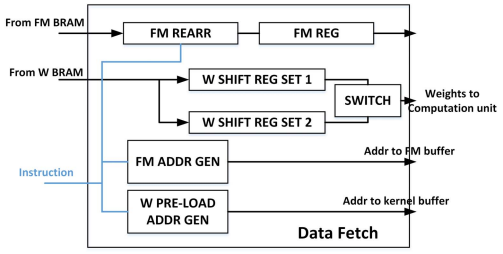


Fig. 7. Data fetch module.

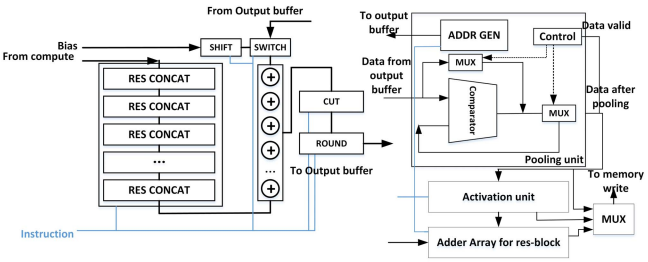


Fig. 8. Data post process module.

to flexibly fit into the needs of different combinations of input–output channels. For example, current implementation supports 1024 multiplications, which is able to handle six $[\text{in}_{\text{channel}}, \text{out}_{\text{channel}}]$ pairs: [512, 2], [256, 4], [128, 8], [64, 16], [32, 32], and [16, 64].

Our computation pattern guarantees the uniform data fetching pattern for any kernel size or stride. This greatly simplifies the data management stage before compute operation and enables higher design frequency with less resource consumption. Moreover, we leverage both the input and output channel level parallelisms in a flexible way. This provides higher flexibility for resource utilization and promises reasonable generalization performance.

B. Data Fetch and Postprocess

The data fetch module reads FM and kernel data from on-chip buffer, rearranges the data and then sends to the computation unit. As shown in Fig. 7, for input FM read, FM ADDR GEN takes control parameters from instruction and produce FM buffer read address at each clock cycle. The parameters include [Xmin, Xmax, Ymin, Ymax, Xstride, Ystride, Xsize, Ysize]. The FM data read from buffer will be selected and copied by the FM REARR to fit the target computation pair of computation unit. For kernel weights, the bandwidth requirement can be as high as 8192 bit/cycle, since all the parallelisms are explored on the kernel side (input channel/output channel). We choose a computation pattern that shares the same set of kernel weights during one round of computation. Accordingly, kernel weights are only preloaded once from buffer for each round. We use W PRE-LOAD ADDR GEN to generate weights address for buffer, and each weight takes 32 cycles to load. This loading time is overlapped with the previous round of data fetch process. A pair of local shift registers using ping-pong structure [W SHIFT REG SET 1, W SHIFT REG SET 2] is used to cache the weights.

The data postprocess module performs data quantization, partial sum addition, pooling, activation, and residual addition. Since pooling, activation, and residual are only conducted once for one memory write, they are concatenated with the data memory write module to reduce extra on-chip data movement. As shown in Fig. 8, a data concatenation block is placed right after the input port to collect computation output (the output data number can be [2, 4, 8, 16, 32, 64] based on different computation patterns) and formulate a 64-word long array for later process. A set of adders is used for bias addition and partial sum addition. Then, the data quantization is achieved by data shift, cut and round modules that work based on parameters provided by instructions. When the complete output is ready, another part of the post process will be called and conduct pooling, activation, and residual addition. The detailed corresponding module structure is shown in the right half of Fig. 8.

C. Memory Management

One crucial issue for CNN acceleration on FPGA is off-chip communication latency. Roof-line model [5] reveals the relationship between bandwidth utilization and computational roof performance. Bandwidth can easily be the bottleneck of performance. Therefore, we utilize a ping-pong structure-based caching memory management system to hide off-chip communication latency. While one buffer's data is being fetched, the other buffer can get refilled and updated, which maintains the maximum bandwidth utilization.

Another key point in memory management is data storage format in both local onboard buffer and external memory. For onboard storage format, data from the same channel slice are stored under the same address so that they can be fetched in one clock cycle. The limit of channel slice depth is set to be the width of on-chip buffer to guarantee such memory arrangement. Benefited from computation pattern, enough data bandwidth is provided and no extra memory latency is caused during the computing stage.

Moreover, data storage format in external memory influences data transmission efficiency and complexity of memory access unit. We thereby employ a channel sliced scheme for FM storage. As shown in Fig. 9, data from one channel slice

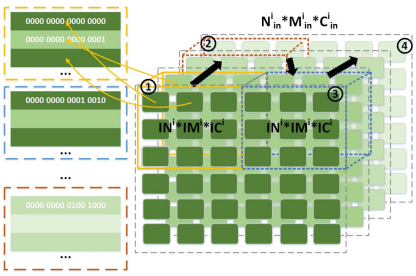


Fig. 9. Channel-based memory storage management.

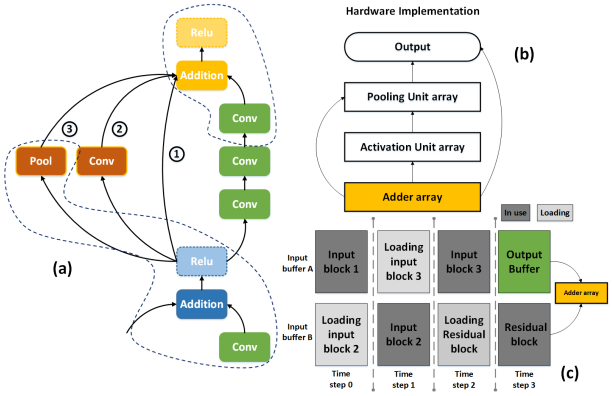


Fig. 10. (a) Residual module type description. (b) Embedding addition operation into postprocess data streaming pipeline. (c) Loading time management for residual data.

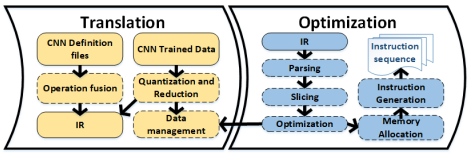


Fig. 11. Two steps flow of compiler.

with shape $1*1*IC^i$ is stored adjacently. In this way, data can be streamed onboard in burst mode to fully utilize bandwidth. During the computation process, data chunk loading order is ① − > ② − > ③ − > ④. Each chunk of data represents one sub-FM block.

D. Irregular Operation Handling

ResNet [16] and GooglNet [15] exhibit excellent performance with reduced computational requirements compared with VGGNet [14] and AlexNet [18]. However, they also introduce new nonconvolution operations that require extra attention.

1) Inception Module: Channelwise input concatenation is required for inception module. Taking advantage of our memory storage pattern, we manipulate output memory addresses of preceding layers to place their outputs adjacently. Therefore, input concatenation can be achieved without any extra computational operation or latency cost. The arranged memory can be loaded for the next layer in burst mode for efficient off-chip memory transmission.

2) Residual Module: The operation of residual module is matrix elementwise addition, which is not computational intensive but may cause extra stage of off-chip data communication. We embed the addition operation into postprocess pipeline and cover the additional block loading stage with computation time, so residual operation is executed with negligible latency increment. Fig. 10 illustrates how this implementation can realize all types of residual modules. As shown in Fig. 10(a), three kinds of residual paths are labeled out. Moreover, an activation module after elementwise addition is optional. We group elementwise addition, pooling, and activation together and embed to the previous convolution layer's operation pipeline. Fig. 10(b) shows that an adder array is inserted at the first stage of postprocess hardware implementation. All function modules are implemented with bypass logic. With the combination of different bypass and functions, all types of residual layer can be computed. Fig. 10(c) shows the content of ping-pong input buffer at different time steps. At time step 2, current block's computation is at the final stage and uses data from input buffer A. Meanwhile, residual source is being loaded to input buffer B. At time step 3, both matrices for elementwise addition are ready and postprocess begins with no extra latency incurred.

V. Compiler

We develop a compiler to perform operation fusion, network slicing, and throughput optimization on input CNN configuration. There are two stages during the operation of compiler: Translation and Optimization, as shown in Fig. 11. Translation extracts necessary information from model definition files and reorganizes them into a uniform intermediate representation (IR) we defined. During this process, operation fusion (introduced in Section V-A) is conducted to combine closely related operations. Another aspect of translation stage is data quantization and arrangement. Fast data quantization is performed for Kernel Weights/Bias/FMs with negligible accuracy loss. Generated dynamic fixed-point representation system is then merged into IR. Moreover, processed weights get rearranged based on network slicing and optimization results from the optimization stage. Optimization stage parses Translation generated IR and explores the solution space of network slicing to maximize throughput. For this stage, an optimizing scheme for slicing is developed, which will be introduced in Section V-D. In the end, the optimization solution is mapped to an instruction sequence.

A. Operation Fusion

Conventional CNNs contain various types of layers that are connected from top to bottom to form a complete flow. In order to avoid the off-chip memory communication between layers as to reduce computation requirements, operation fusion is employed to merge or concatenate related layer operations. Here, we define two types of fusions, p-fusion and r-fusion. p-fusion indicates operation fusion that only contributes to off-chip memory access reduction, and r-fusion not only avoids communication latency but also reduces the total number of operations and inference time. Among them, p-fusion and r-fusion-I are hardware microarchitecture independent, while r-fusion-II is related to actual implementation scale.

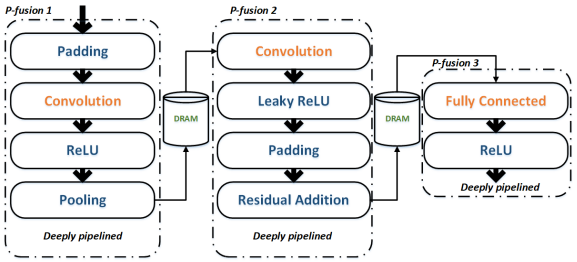


Fig. 12. p-fusion example on a sequence of layers.

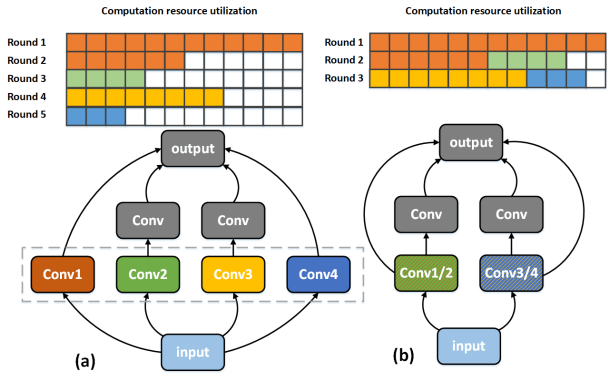


Fig. 13. (a) Original inception module. (b) Merged inception module.

1) p-Fusion: We perform p-fusion to concatenate different layers into the same data stream pipeline, which prevents saving intermediate results back to off-chip memory. Convolution and Fully connected (FC) layers are major layers. Pooling, Padding, Activation, Residual, and Output concatenation layers are treated as affiliated layers. Fig. 12 shows how this is applied on a set of layers. Originally off-chip memory access is required between each pair of adjacent layers, such as $\{Padding, Convolution\}$, $\{Convolution, ReLU\}$, and $\{ReLU, Pooling\}$. After employing p-fusion $1-3$, memory access is reduced to only two times. The fused layers are called a layer group.

2) r-Fusion: Fusion that decreases the amount of hardware arithmetic computations by merging adjacent operations is also conducted. There are mainly two kinds of r-fusions.

r-fusion-I is Batch normalization [20] elimination. A batch normalization operation can be represented by

$$y = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{1}$$

where γ, μ, σ, β, and ϵ are fixed values during inference time. x and y represent the input and output matrices of one channel. Equation (1) is essentially a linear function of input matrix, which can be merged to preceding or succeeding convolutional layers based on network structure. This merging avoids the separate computation of batch normalization, thus reducing the inference time. The merged convolutional layer works with modified weights and bias that incorporate batch normalization coefficients.

r-fusion-II is input sharing. In many cases, layers that share identical inputs can be computed by the same round to fully utilize computational resources and reduce memory communication time. As shown in Fig. 13(a), Conv1-Conv4 share the same input layer, but the numbers of their output channels are either small or cannot be evenly divided by available computing resources. Therefore, it takes five rounds to compute the four layers sequentially with relatively low runtime efficiency of PEs. To make use of idle resources, we add an optimization step into compiler that identifies input sharing layers and reassembles them based on their individual resource consumption, as shown in Fig. 13(b). Fig. 14 shows the improvement of computation resource runtime efficiency after applying input sharing r-fusion to different inception modules. Different groups indicate different scales of computing resource, which increase from left to right. More resources available means higher possibility of resource idling, thus providing more room for input sharing optimization.

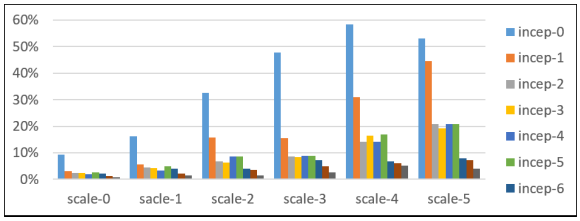


Fig. 14. Throughput improvements by applying r-fusion-II on inception modules.

B. Data Quantization

It has been proven that CNNs are robust against precision reduction [6], [21]. To reduce memory footprint and save computational resources, we use limited precision fixed-point values during computation, data type can be set to fixed-4/8/16 bits based on platform constraint and network accuracy requirements. Taking general network precision redundancy and hardware architecture complexity into consideration, 8-bit system is chosen as our data quantization standard for both FM and kernel weights. We employ a fast yet effective stationary quantization method. Dynamic quantization scheme (similar as [6] and [22]) has been employed for better accuracy. In such scheme, each layer's kernel weights and FMs have their own range for higher precision. The process of finding the best range for each trunk of data is described as follows:

$$\underset{\text{floc}}{\operatorname{argmin}} \sum (\text{float} - \text{fix}(\text{floc}))^2 \tag{2}$$

where float is the original single precision representation of kernel weights or the FMs, and fix(floc) is the value after the float is cut into fixed-point based on certain fraction length floc. Table I shows the quantization accuracy evaluation of nine experiment networks, where the accuracy loss is within 1% on average.

C. Intermediate Representation (IR)

The IR is defined based on layers after the p-fusion, which we call layer groups, and each layer group gets represented by a set of coefficients, as given in Table II. The representation is easy to expand by adding more terms to accommodate for new network features.

IR contains all the operations included in the current layer groups. Layer index is the sequential number assigned to

TABLE I
8-bit QUATIZATION EVALUATION FOR DIFFERENT NETWORKS

	Classification[a]							Detection[b]	
	VGG16	VGG19	Inception V1	Inception V2	Inception V3	resenet V1	resnet V2	YOLO V2[c]	Tiny YOLO[d]
Float 32 bit	89.8%	85.2%	87.29%	90.30%	93.15%	92.9%	93.7%	85.93%	90.8%
Fixed 8 bit	89.4%	84.3%	85.49%	89.87%	91.41%	92.3%	93.2%	86.19%	89.5%

a: Reported accuracy are top-5 accuracy evaluated on Imagenet ILSVRC2012 validation set
b: Reported value are mAP. c: Evaluated on a private subway x-ray dataset. d: Evaluated on a private traffic view dataset.

TABLE II
IR CONTENT FOR SINGLE LAYER GROUP

Layer Type	Fully connected(0), Conv(1), extra pool(2)			
Input idx Output idx	Input/Output layer index of length l_{in}, l_{out} $l_{in} > 1$ for inception case $l_{out} > 1$ for residual case			
Input size Output size	3 Dimensional Triple			
Kernel info	Kernel size		Kernel stride	
Pool info	Pool enable	Pool type	Pool size	Pool stride
Padding info	Pad enable	Pad size	pad bf pool info	
Activation type	Relu and Leaky Relu			
Dump out loc	Intermediate FM from either major or affiliated layer can be dumped out to DRAM			
Element-wise op loc	Residual addition can be performed between any two conventional layers			
Quantization info	Dynamic fix point design for Weights/FM			

each conventional layer. Single layer group may have multiple layer indexes for input in the case of inception module, where various previous outputted FMs are concatenated to form the input. Meanwhile, multiple intermediate FMs generated during layer group computation can be used as other layer groups' residual or normal input sources. Dump out location specifics which set of FM to dump out to the off-chip memories. Elementwise operation location is used to flexibly adjust elementwise residual addition position in layer groups, so any combination of pool/activation/elementwise addition can be accommodated.

D. Slicing and Allocation

In this stage, IR generated by Translation stage is parsed to get network architecture after operation fusion and quantization. Then, an automatic optimizer is applied to explore optimal slicing scheme that maps current architecture to overlay with maximum throughput.

Suppose an individual layer i is sliced into p^i blocks. One slicing scheme for layer i can be defined as a vector of parameter groups $\vec{P}$: $[(\mathrm{IN}^i_{j_n}, \mathrm{IM}^i_{j_m}, \mathrm{IC}^i_{j_c}, \mathrm{OC}^i_{j_c}) | j_n \in [0, p^i_n), j_m \in [0, p^i_m), j_c \in [0, p^i_c)]$. Each parameter group $(\mathrm{IN}^i_{j_n}, \mathrm{IM}^i_{j_m}, \mathrm{IC}^i_{j_c}, \mathrm{OC}^i_{j_c})$ decides one round of overlay computation, where $\mathrm{IN}^i_{j_n}$, $\mathrm{IM}^i_{j_m}$, $\mathrm{IC}^i_{j_c}$, and $\mathrm{OC}^i_{j_c}$ represent input block width, height, input depth, and output depth, respectively. Then, we have

$$N^i_{\mathrm{in}} = \sum_{j_n}^{p^i_n} \mathrm{IN}^i_{j_n}, \quad M^i_{\mathrm{in}} = \sum_{j_m}^{p^i_m} \mathrm{IM}^i_{j_m}$$
$$C^i_{\mathrm{in}} = \sum_{j_c}^{p^i_c} \frac{\mathrm{IC}^i_{j_c}}{\#C^i_{\mathrm{out}} \text{ slices}}, \quad C^i_{\mathrm{out}} = \sum_{j_c}^{p^i_c} \frac{\mathrm{IC}^i_{j_c}}{\#C^i_{\mathrm{in}} \text{ slices}} \tag{3}$$

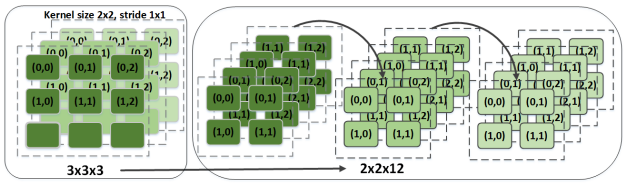


Fig. 15. Input image rearrangement: channel dimension filling.

TABLE III
FPGA RESOURCE UTILIZATION

		LUT	FF	BRAM	DSP
OPU1024	XC7K325T	94763(46.50%)	150848(37.01%)	165(37.08%)	516(61.43%)
OPU2048	XC7K325T	129927(63.75%)	233996(57.41%)	165(37.08%)	817(97.26%)
OPU4096	XC7Z100	154516(55.70%)	337651(60.86%)	337(44.64%)	1986(98.32%)

and

$$p^i = p^i_n \times p^i_m \times p^i_c. \tag{4}$$

The inference latency L^i_j of one round computation can be represented by

$$L^i_j = \left(k^i_x \times k^i_y + 2\right) \times \mathrm{ON}^i_{j_n} \times \mathrm{OM}^i_{j_m}, \quad j \in [n0, p^i) \tag{5}$$

where $\mathrm{ON}^i_{j_n}$ and $\mathrm{OM}^i_{j_m}$ indicate the output block width and height, and the $+2$ term accounts for initial memory read and final memory write latency. Then, we define the throughput of inferencing a network as

$$T = \frac{\sum_i^{\hat{m}} N^i_{\mathrm{out}} \times M^i_{\mathrm{out}} \times \left(2 \times C^i_{\mathrm{in}} \times k^i_x \times k^i_y - 1\right) \times C^i_{\mathrm{out}}}{\sum_i^{\hat{m}} \sum_j^{p^i} L^i_j} \tag{6}$$

where $\hat{m}$ represents the number of layers after fusion.

Therefore, the target of slicing optimization can be represented as

$$\begin{aligned} &\max_P \ T \\ &\text{s.t. } \mathrm{IN}^i_{j_n} * \mathrm{IM}^i_{j_m} <= \mathrm{depth}_{\mathrm{thres}} \\ &\quad \mathrm{ceil}\left(\frac{\mathrm{IC}^i_{j_c}}{V_{\mathrm{PE}}}\right) * \mathrm{OC}^i_{j_c} <= N_{\mathrm{PE}} \\ &\quad \mathrm{IC}^i_{j_c}, \ \mathrm{OC}^i_{j_c} = \mathrm{width}_{\mathrm{thres}} \end{aligned} \tag{7}$$

where $\mathrm{depth}_{\mathrm{thres}}$ and $\mathrm{width}_{\mathrm{thres}}$ stand for on-chip BRAM depth and width limit, respectively.

TABLE IV
NETWORK INFORMATION

	YOLOv2	tiny-YOLO	VGG16	VGG19	InceptionV1	InceptionV2	InceptionV3	Resnet-50	Resnet-101
Input size	608×608	416×416	224×224	224×224	224×224	224×224	299×299	224×224	299×299
Kernel size	1×1,3×3	1×1,3×3	3×3	3×3	1×1,3×3,5×5,7×7	1×1,3×3	1×1,3×3,5×5,1×3,3×1,1×7,7×1	1×1,3×3,7×7	1×1,3×3,7×7
Pool size/Pool stride	2×2	2×2	2×2	2×2	3×2,3×1,7×1	3×2,3×1,7×2	3×2,3×1,8×2	3×2,1×2	3×2,1×2
Conv layer number	21	9	13	16	57	69	90	53	53
Activation Type	Leaky ReLU	Leaky ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU
Operations (GOP)	54.07	5.36	30.92	39.24	2.99	3.83	11.25	6.65	12.65

TABLE V
RME OF OPU1024 FOR DIFFERENT NETWORKS

	YOLOV2	tiny-YOLO	VGG16	VGG19	InceptionV1	InceptionV2	InceptionV3	Resnet-50	Resnet-101
Frequency (MHZ)	200								
RME	95.51%	89.21%	97.18% (B)	97.30% (B)	90.38% (B)	90.48% (B)	91.10 (B)%	84.48%	86.92%
Conv RME	95.51%	89.21%	97.01%	97.75%	90.45%	90.75%	90.90%	84.48%	86.92%
Frame/s	7.23	68.32	12.18 (B) / 11.28	9.72 (B) / 9.4	112.48 (B) / 104.47	89.77 (B) / 84.60	30.01 (B) / 27.28	54.36	27.05

B: Evaluated in batch mode with batch size 8.

Fig. 16. Evaluation board and runtime results for classification network VGG16 and detection network YOLO.

E. Extra Efficiency Improvements

In most cases, combining two levels of parallelism $IC^i * OC^i$ utilizes a large portion of PE resources. However, rare cases exist where C^i_{in} and C^i_{out} are too small and offer limited parallelism. This usually happens in the first layer, where C^1_{in} is fixed to 3 and C^i_{out} is usually below 64. To accommodate for this situation, our proposed compiler further rearranges input FMs. Pixels in the kernel window are moved to fill the channel dimension to increase the parallelism availability channelwise, as shown in Fig. 15. This rearrangement is able to gain over 10× speedup for the first layer computation on average.

VI. EXPERIMENT RESULTS

We implement three OPU versions with different MAC numbers on Xilinx XC7K325T FPGA and XC7Z100 FPGA. Corresponding resource utilization is given in Table III. For OPU1024, all the MACs are implemented with DSP. For OPU2048 and OPU4096, part of the MACs is implemented with LUT since the number of DSPs is not enough. A PC with Xeon 5600 CPU is used for our compiler program. Result interface and device are shown in Fig. 16.

A. Network Description

To evaluate the performance of OPU, nine CNNs of different architectures are mapped, including YOLOv2, tiny-YOLO, VGG16, VGG19, Inceptionv1/v2/v3, Resnetv1-50, and Resnetv1-101. Among them, YOLOV2 and tiny-YOLO are object detection networks and the rest is image classification networks. Detailed network architectures are given in Table IV. Different Kernel sizes from square kernel $(1 \times 1, 3 \times 3, 5 \times 5, 7 \times 7)$ to sliced kernel $(1 \times 7, 7 \times 1)$ are used, in addition to various pooling $(1 \times 2, 2 \times 2, 3 \times 1, 3 \times 2, 7 \times 1, 7 \times 2, 8 \times 2)$ sizes and activation types (ReLU and Leaky ReLU). Irregular operations, such as inception module and residual module, are included as well. All networks are quantized to 8-bit precision for both kernel weights and FM to achieve higher efficiency.

B. Runtime MAC Efficiency

OPU is designed to be a domain-specific processor for a variety of CNNs. Therefore, runtime MAC efficiency (RME) for different CNN's is an important metric for both hardware efficiency and performance. RME is calculated by the actual throughput achieved during runtime, divided by the theoretical roof throughput (TTR) of design. The TTR can be calculated as follows:

$$\text{TTR} = \text{MAC}_{\text{num}} \times 2 \times f \tag{8}$$

where MAC_{num} indicates the number of MACs and f represents the design frequency. For instance, in our implementation of OPU1024 running with 200 MHZ, 1024 MAC units are utilized for PE array. Therefore, we have $\text{TTR}_{\text{opu1024}} = 1024 \times 2 \times 200$ Mhz $= 409.6$ GOPS. The actual throughput achieved by running the convolution part of VGG network on the OPU is 397 GOPS, so the RME for VGG (CONV) is $(397/409.6) = 97.79\%$. High RME indicates all computational resources of hardware are well-utilized and processor is efficient for the current network. Note that for FC layers, a high RME is normally difficult to be obtained under nonbatch mode due to large number of weights and relatively small computational requirements. Therefore, for networks that contain FC layers, we report the overall RME under batch mode, and frames per second under both nonbatch and batch mode for better evaluation. Processor is running under 200 MHZ. As given in Table V, on average an overall RME of 91.44% on average is achieved for all test networks. This is even higher than

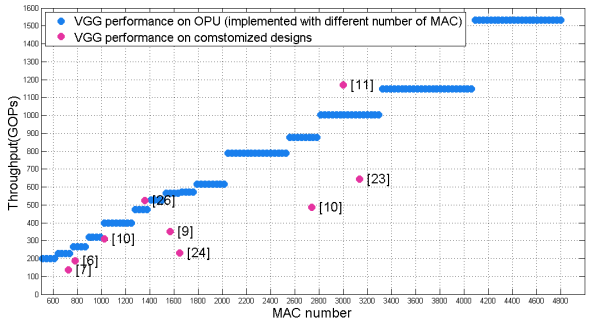


Fig. 17. Performance comparison of OPU implemented on similar number of MACs with reference designs.

the start-of-the-art customized implementations (discussed in Section VI-C).

C. Comparison With Existing FPGA Accelerators

In this section, we compare the performance of OPU with autocompiler generated network-specific accelerators. Table VI lists out customized accelerators designed for networks VGG16 or YOLO, which are implemented on FPGAs of similar scales for fair comparison. We use throughput and RME as the comparison criteria. The number of MACs and design frequency decide the TTR that can be achieved by certain design. RME shows how well the accelerator utilizes all its available resources actually, which can be directly transformed to real throughput based on TTR. When estimating the TTR of reference design, we also take the data format into consideration. One 25-bit $\times$ 18-bit DSP can be decomposed to handle two 8-bit $\times$ 8-bit multiplications, while it can only handle one 16-bit $\times$ 16-bit multiplication. Therefore, for the same number of DSPs, 8-bit system has twice overall computational capability compared with 16-bit system. A coefficient α is used to account for the influence of data format. The RME of different designs can be computed as follows:

$$\text{RME} = \frac{T}{\alpha \times \text{DSP}_{\text{num}} \times f \times 2} \tag{9}$$

where T represents the real throughput achieved by the design, DSP_{num} indicates the total number of DSP utilized. Here, $\alpha = 1$ for 16-bit system, and $\alpha = 2$ for 8-bit system. The $\times 2$ term is used to compute both multiplication and addition operations. Compiling VGG and YOLO for OPU takes much less time compared to generating one network-specific accelerator by automatic compile. Yet, OPU achieves better RME compared with automatically compiled network-specific accelerators. For example, OPU has 86% RME, while the RME for other VGG-specific accelerators ranges from 58% to 69% in Table VI.

Direct comparison of throughput without taking the number of utilized MACs into consideration is not fair, so we scale our design to match the number of MACs in different reference designs. As shown in Fig. 17, the blue dots represent the simulated real performance of OPU implemented with a different number of MACs running VGG16. Increase in the MAC number leads to the improvement of the real throughput. But their relationship is not linear, as MAC number that cannot evenly divide the available parallelism of network very well may have low RME. In such cases, extra MACs come with no additional throughput (as indicated by the horizontal lines formed by the blue dots). Purple dots indicate the throughput of reference designs. Most of them locate in the area below OPU "line," which means the similar number of MAC provides lower throughput compared with OPU. Among all the reference designs, implementations in [11] and [24] have higher performance compared with OPU. The performance of [11] is achieved by its high design frequency (231 MHz), which is enabled by the systolic PE matrix architecture that shares inputs with neighboring PEs to provide simple routing. However, this architecture is easily constrained by the specific network structure, thus requires the change of PE matrix shape for every new network to achieve claimed performance. (PE matrix shape of [11, 14, 8] and [8, 19, 8] are used for Alexnet and VGG separately [11]). Zhang *et al.* [24] create their own version of pruned VGG to improve performance. Moreover, they use layer-pipe-lined design that accelerates several layers of one network independently. It requires large modifications of FPGA implementation for different networks ([#DSP, #BRAM] resource of [680, 542] and [808, 303] are used for Alexnet and VGG, respectively).

Normally, the design parameters of customized accelerators are tuned specifically to fit the configuration of target network. Therefore, they are able to gain better performance. For example, an accelerator design for VGG can be specifically tuned to fit only the 3×3 kernel size and cannot perform well with YOLO where 1×1 kernel size is included. Moreover, the control flow and data buffer need to be changed to fit different target networks. We summarize our performance advantages over customized accelerators into two points. First, input and output channel parallelisms have less variation compared with kernel/input FM pixel parallelism. Combined with the first layer rearrangement, it can provide enough parallelism to fully utilize the computation resources and gain high throughput. For example, if only paralleling the kernel and input channel, the computation resource can easily fall underutilized when kernel size is too small. Second, our PE array is designed to fit various types of [input–output] pairs, which cover the majority of the configurations in CNNs. Therefore, we can achieve higher throughput than designs with fixed number of input and output channels.

D. Power Comparison

We compare the power efficiency of OPU with other FPGA designs as well as GPU and CPU. Table VII lists out the comparison results of different hardware platform running VGG16. We measure the power consumption of OPU using a PN2000 electricity usage monitor. The reported power includes static and dynamic powers consumed by the whole board. FPGA designs generally have better power efficiency compared with CPU, around $2\times$ better power efficiency compared with GPU with batch $= 1$ and no obvious advantage compared with GPU with larger batch in most cases [10], [21], [25]. Considering the fast development speed of GPU

TABLE VI
COMPARISON WITH CUSTOMIZED ACCELERATORS (VGG AND YOLO)

	[7]	[6]	[23]	OPU1024	[21]	OPU1024	
Device	XC7Z045	XC7Z045	XC7Z045	XC7K325T	XC7Z020	XC7K325T	XC7K325T
Network	VGG16				tiny-YOLO		YOLOv2
DSP Utilization	727 (900)	780 (900)	824 (900)	516 (840)	190 (220)	516 (840)	
Data format (bit)	16	16	8	8	200	200	200
Frequency (MHZ)	120	150	100	200	214	200	200
TTR (GOPS)	174	234	329	412	162	412	412
Throughput(GOPS)	118 / 137 (C)	137 / 188 (C)	230	354 / 397 (C)	62.9	366	391
RME	67% / 78 (C)%	58% / 79% (C)	69%	**86% / 97%(C)**	39 %	**89%**	**95 %**

C: Convolutional layer only

TABLE VII
POWER COMPARISON AMONG CPU, GPU, AND FPGA DESIGNS

	CPU	GPU				FPGA							
Device	i7-8700	GTX 780	GTX 1080 [a]		Titan Xp		[10] KU060	[10] KU060	[10] VX690t	[21] XC7Z045	[21] XC7Z020	[25] Stratix-V GXA7	OPU1024 XC7K325t	OPU4096 XC7Z100
Technology	14 nm	28 nm	16 nm		16 nm		20nm	20nm	28nm	28nm	28nm	28nm	28nm	
Power(W)	120	228	180	180		180	25	25	26	9.63	3.5	19.5	16.5	17.7
batch	1	64	1	64	1	64	1	1	1	1	1	1	1	1
Throughput (GOPS)	26	1563	1310	3788	628.53	10620	266	1171	354	137	61.8[b]	114.5[c]	354	1218
Throughput/power (GOPS/W)	0.21	6.85	7.28	21.04	3.49	51.80	10.65	46.84	13.61	14.22	17.65	5.87	21.45	68.81
power efficiency	1×	3.73×	3.96×	11.47×	1.91x	28.25×	5.8×	25.54×	7.42×	7.75×	9.63×	3.20×	**11.7**×	**37.53**×

a Data taken from [10]
b [22] only reports convolutional layer throughput(84.3 GOPs), while all the other designs are compared with the overall network throughput. We compute the equivalent overall network throughput based on the throughput relationship of another design implemented by the same paper: $84.3 \times \frac{137}{187} = 61.8 GOPs$
c This performance is obtained with running Alexnet

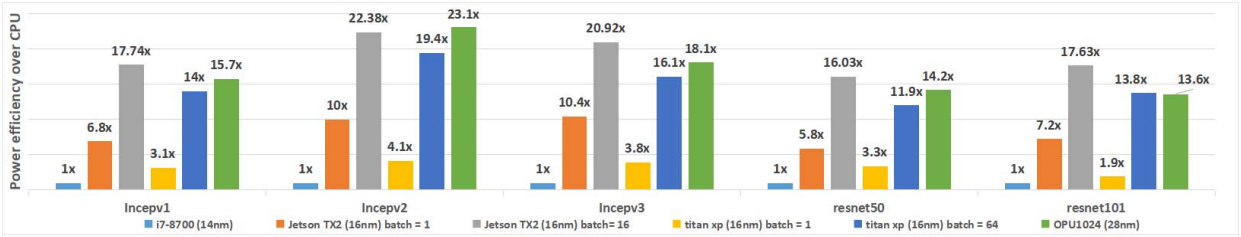


Fig. 18. Power efficiency (GOPS/W) comparison of CPU/GPU/OPU using CPU as baseline.

devices, we include the comparison results of three GPUs of different technologies. GTX 780 uses 28 nm (same as OPU), and GTX 1080 and Titan Xp use 16 nm.

From Table VII, we can see that the power efficiency of OPU1024 running a VGG16 network is 11.7× better than 14-nm technology CPU, on average 4.5× better for batch = 1 GPU devices, and 3.6× to 1.2× better compared with other FPGA designs. The only FPGA design that has better power efficiency is from [10], where they use LUT to implement all the MACs and leave on-chip DSPs idle. They implemented over 4000 MACs (the exact MACs number is not specified by [10]), which brings large throughput advantage compared with OPU1024 with only 1024 MACs implemented. Therefore, we implement OPU4096 with 4096 MACs on another board with Zynq7z100 for fair comparison. The result shows that OPU4096 has 1.5× better power efficiency compared with [10]. Moreover, OPU4096 shows 3.27× and 1.33× better power efficiency comparing with GTX 1080 and Titan Xp running at batch = 64. This is because static power after bitstream download, which takes a big portion of the overall power, does not tend to increase in proportional with board size. Therefore, using a larger board normally can acquire better power performance, on condition that the performance does not degrade due to design size change.

We also compare the power performance of OPU with GPU/CPU when running other networks, such as inception series and residual-net series. We use edge computing targeted GPU platedform Jetson Tx2 and server targeted Titan Xp for evaluation. As shown in Fig. 18, OPU1024 with 28-nm technology has over 15× better power efficiency compared with 14-nm i7-8700 CPU. For 16-nm Jetson Tx2 and Titan Xp running with batch = 1, the power efficiency of OPU1024 is 2.13× and 5.35× better. For Titan Xp running with batch = 64, OPU1024 is 1.25× better. For Jetson Tx2 running with batch = 16, OPU1024 shows 89% power efficiency. However, big batch size indicates over 6× larger latency compared with batch = 1 mode. For edge computing targeted device, the real-time batch = 1 mode with short latency performance is more important.

E. Case Study of Real-Time Cascaded Networks

To further evaluate the real-time performance on cascaded networks of OPU, we implement the task on OPU1024 to recognize car license plate from street-view pictures. It is composed of three networks: car-YOLO (trained based on

TABLE VIII
REAL-TIME CASCADED NETWORK EVALUATION COMPARISON WITH JETSON

	Frequency (MHZ)	TTR (TOPs)	car-YOLO	plate-tiny-YOLO	cr-network	Speed
Jetson Tx2	845	0.45	188 ms	47 ms	16 ms	1×
OPU1024	200	0.41	64 ms	19 ms	1 ms	2.9×

YOLO for car detection), plate-tiny-YOLO (trained based on YOLO for plate detection), and a character recognition network (cr-network). For single picture input, the car-YOLO network runs first to label all cars. Then, plate-tiny-YOLO and cr-network run to detect the plate numbers or characters for each car.

As given in Table VIII, we compare the performance for OPU1024 and Jetson Tx2. Tx2 is running with batch = 5 and the speed data are computed by total time between input to output divided by 5.

It can be seen that OPU is faster in executing all three networks compared to Jetson. Overall, OPU is 2.9× faster than Jetson. With similar computation capability, the higher speed achieved by OPU comes from the higher PE utilization rate enabled by our domain-specific architecture and compiler.

VII. CONCLUSION AND DISCUSSION

In this article, we proposed OPU, a domain-specific FPGA-based overlay processor system targeting at general CNN acceleration. It is software-programmable with short compilation time. We developed a set of instructions with granularity optimized for PE efficiency. Our compiler performs operation fusion to reduce inference time and conducts network slicing to maximize overall throughput and RME. OPU exhibits high flexibility and generality for a variety of CNNs with average RME around 91%. It has a higher RME on VGG16 and YOLO networks compared with the state-of-the-art network-specific autocompiler generated accelerators. For power consumption, OPU of different scales show 1.2×–5.35× better power efficiency compared with GPU (batch = 1, batch = 16, and batch = 64) and other FPGA design. Moreover, for cascaded CNN networks to detect car license plate, OPU is 2.9× faster compared with edge computing GPU Jetson Tx2 with a similar amount of computing resources.

Our future work will develop better microarchitecture and more compiler optimization mechanisms, extend OPU for both CNN and recurrent neural network (RNN), and also apply and optimize OPU for different deep learning applications, particularly for 3-D medical images.

REFERENCES

[1] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An FPGA-based processor for convolutional networks," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug./Sep. 2009, pp. 32–37.

[2] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Res., Redmond, WA, USA, White Paper 2, 2015.

[3] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Sep. 2010, pp. 273–284.

[4] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, 2010.

[5] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2015, pp. 161–170.

[6] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2016, pp. 26–35.

[7] 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*, Feb. 2016, pp. 16–25.

[8] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2016, pp. 1–12.

[9] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–8.

[10] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2016, pp. 1–8.

[11] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. 54th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2017, pp. 1–6.

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

[13] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," Dec. 2016, *arXiv:1612.08242*. [Online]. Available: https://arxiv.org/abs/1612.08242

[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," Sep. 2014, *arXiv:1409.1556*. [Online]. Available: https://arxiv.org/abs/1409.1556

[15] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 2818–2826.

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

[17] M. S. Abdelfattah *et al.*, "DLA: Compiler and FPGA overlay for neural network inference acceleration," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 411–418.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2015.

[20] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," Feb. 2015, *arXiv:1502.03167*. [Online]. Available: https://arxiv.org/abs/1502.03167

[21] K. Guo *et al.*, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.

[22] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 45–54.

[23] 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. 54th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2017, pp. 1–6.

[24] X. Zhang *et al.*, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2018, Art. no. 56.

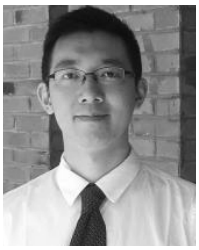
[25] Y. Ma, N. Suda, Y. Cao, J.-S. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of convolutional neural networks onto FPGA," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug./Sep. 2016, pp. 1–8.

Yunxuan Yu received the B.S. degree in microelectronics from Peking University, Beijing, China, in 2014, and the M.S. degree in ECE from the University of California, Los Angeles (UCLA), Los Angeles, CA, USA, in 2017, where she is currently working toward the Ph.D. degree at EDA lab.

Her current research interests include deep learning FPGA acceleration.

Chen Wu received the B.S. degree from the University of Electronic Science and Technology of China, Chengdu, China, in 2012, and the M.S. degree from Tsinghua University, Beijing, China, in 2015. He is currently working toward the Ph.D. degree in electrical and computer engineering at the University of California, Los Angeles (UCLA), Los Angeles, CA, USA.

His current research interests include computer architecture, deep learning, VLSI design, and hardware acceleration.



Tiandong Zhao received the B.S. degree from Fudan University, Shanghai, China, in 2017. He is currently working toward the M.S. degree in electrical and computer engineering at the University of California, Los Angeles (UCLA), Los Angeles, CA, USA.

His research interests include architecture and compiler optimization for hardware acceleration.

Kun Wang (M'13–SM'17) received the Ph.D. degrees in computer science from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2009, and from the University of Aizu, Aizuwakamatsu, Japan, in 2018.

He was a Postdoctoral Fellow at the University of California, Los Angeles (UCLA), Los Angeles, CA, USA, from 2013 to 2015, where he is currently a Senior Research Professor. He was a Research Fellow at the Hong Kong Polytechnic University, Hong Kong, from 2017 to 2018, and a Professor at the Nanjing University of Posts and Telecommunications. His current research interests are mainly in the area of big data, wireless communications and networking, energy Internet, and information security technologies.

Dr. Wang is a recipient of the IEEE GLOBECOM 2016 Best Paper Award, the IEEE TCGCC Best Magazine Paper Award 2018, the IEEE TCBD Best Conference Paper Award 2019, the CBD 2019 Best Student Paper Award, and the IEEE ISJ Best Paper Award 2019. He serves/served as an Associate Editor of the IEEE ACCESS, an Editor of the *Journal of Network and Computer Applications*, and a Guest Editor of the IEEE NETWORK, the IEEE ACCESS, *Future Generation Computer Systems*, *Peer-to-Peer Networking and Applications*, *IEICE Transactions on Communications*, *Journal of Internet Technology*, and *Future Internet*.

Lei He (M'03–SM'08) received the Ph.D. degree in computer science from the University of California, Los Angeles (UCLA), CA, USA, in 1999.

He was a Faculty Member with the University of Wisconsin, Madison, WI, USA, from 1999 to 2002. He consulted Cadence Design Systems, Cisco, Empyrean Soft, HewlettPackage, Intel, and Synopsys, and was a Founding Technical Advisory Board Member for Apache Design Solutions and Rio Design Automation. He is a Co-Founder of Silicon Cloud International and the Founder of NxEco, Inc. He is currently a Professor with the Electrical and Computer Engineering Department, UCLA, and a Guest Chair Professor with Fudan University, Shanghai, China. He has authored or coauthored one book and more than 200 technical papers. His current research interests include modeling and simulation, very-large-scale integration circuits and systems, Internet-of-things, and artificial intelligence.

Dr. He was a recipient of many best paper nominations and awards, including the 2010 ACM Transactions on Electronic System Design Automation Best Paper Award and the 2011 IEEE Circuit and System Society Darlington Best Paper Award.