Light-OPU: An FPGA-based Overlay Processor for Lightweight Convolutional Neural Networks

Yunxuan Yu
ECE department, University of california, Los Angeles.
Los Angeles, CA

Tiandong Zhao
ECE department, University of california, Los Angeles.
Los Angeles, CA

Kun Wang*
ECE department, University of california, Los Angeles.
Los Angeles, CA

Lei He*
ECE department, University of California, Los Angeles.
Los Angeles, CA

ABSTRACT

Lightweight convolutional neural networks (LW-CNNs) such as MobileNet, ShuffleNet, SqueezeNet, *etc.,* have emerged in the past few years for fast inference on embedded and mobile system. However, lightweight operations limit acceleration potential by GPU due to their memory bounded nature and their parallel mechanisms that are not friendly to SIMD. This calls for more specific accelerators. In this paper, we propose an FPGA-based overlay processor with a corresponding compilation flow for general LW-CNN accelerations, called *Light-OPU*. Software-hardware co-designed *Light-OPU* reformulates and decomposes lightweight operations for efficient acceleration. Moreover, our instruction architecture considers sharing of major computation engine between LW operations and conventional convolution operations. This improves the run-time resource efficiency and overall power efficiency. Finally, *Light-OPU* is software programmable, since loading of compiled codes and kernel weights completes switch of targeted network without FPGA reconfiguration. Our experiments on seven major LW-CNNs show that *Light-OPU* achieves 5.5× better latency and 3.0× higher power efficiency on average compared with edge GPU NVIDIA Jetson TX2. Furthermore, *Light-OPU* has 1.3× to 8.4× better power efficiency compared with previous customized FPGA accelerators. To the best of our knowledge, *Light-OPU* is the first in-depth study on FPGA-based general processor for LW-CNNs acceleration with high performance and power efficiency, which is evaluated using all major LW-CNNs including the newly released MobileNetV3.

KEYWORDS

Lightweight CNN, FPGA acceleration, Processor, Compiler

ACM Reference Format:
Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. 2020. Light-OPU: An FPGA-based Overlay Processor for Lightweight Convolutional Neural Networks . In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375311

*Corresponding authors: Kun Wang (wangk@ucla.edu) and Lei He (lhe@ee.ucla.edu).

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375311

1 INTRODUCTION

Conventional convolutional neural network (CNN) acceleration on FPGA has drawn much attention in recent years [2, 4, 7, 16, 18, 21, 24, 28, 34, 37–39]. FPGA accelerators possess the advantages of high power efficiency, low latency, excellent flexibility and good computational capability. These features make it stand out especially in applications of deep CNNs on edge and embedded devices, *e.g.,* speech recognition on smart phones and visual object recognition in real-time on autonomous driving cars [8], where real-time speed and low power are needed.

With the development of deep learning algorithms, a new group of networks, called LightWeight CNNs (LW-CNNs) [6, 11, 12, 23, 40], emerge with the advantages of faster inference time and smaller model size compared with conventional CNNs. While LW-CNNs dramatically shrink down the model size, they also introduce new lightweight operations that cannot be handled well by conventional FPGA CNN accelerators. Moreover, reduction in latency on GPU is also limited. This indicates that lightweight operations do not fit in GPU acceleration architecture (or at least not as nicely as conventional CNN operations do). Therefore, accelerators tuned specifically for LW-CNNs is needed. Several work developed FPGA acceleration for LW-CNNs. [27] and [41] designed customized accelerators for MobileNet. However, separated or only partially shared acceleration engines are utilized for conventional convolution and depthwise convolution (DW-CONV). This causes the redundancy in resource utilization and further reduces the runtime efficiency. [1] deployed shared acceleration engine for different convolutions, but the architecture is designed for MobileNetV2 specifically. Moreover, some work tried to unify operations by modifying network architectures. [35] used 1×1 convolution and shift to get rid of DW-CONV, [32] and developed network architecture search (NAS) to enhance hardware efficiency for targeted model and dataset. [14] performed NAS with respect to hardware friendly templates and again, targeted dataset. However, modified models are not as universally adaptive to different datasets as the original model, and training cost for NAS is extremely high. In short, existing methods suffer from poor adaptivity to other models, limited operation types, inefficient resource utilization, and high cost of NAS.

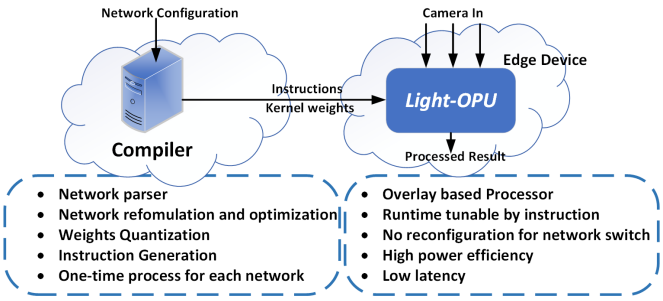

Figure 1: *Light-OPU* working flow.

To deal with these problems above, we propose *Light-OPU* as an FPGA-based general processor for LW-CNNs acceleration. We adopt part of the instruction and architecture design of our work on conventional CNN acceleration[37], then make major improvements to fit the acceleration need of LW-CNNs. More precisely, *Light-OPU* accelerates conventional convolution, DW-CONV and other lightweight operations with one single uniform computation engine. Meanwhile, an automatic compilation framework is provided for the support of general LW-CNNs. As shown in Fig. 1, the compiler takes the network architecture configuration from Tensorflow/Keras/ONNX as input, performs the network reformulation and optimization, along with the quantization for compression, then maps network operations to processor modules for instruction generation. Afterwards, the generated instruction sequence is sent to *Light-OPU* for execution. Consequently, fast deployment is enabled for officially published models without any network retraining due to architecture modification.

To be more specific, the features of our proposed *Light-OPU* are listed as follows:

- **Efficient adaptivity to Light-Weight operations**. Taking CNNs as input, *Light-OPU* slices and maps all types of convolutions, including DW-CONV and group convolution to a uniform acceleration framework. Moreover, irregular lightweight operations are either reformulated to fit in the primary computation engine or assigned to the specific acceleration module with low resource cost.
- **Flexible ISA for LW-CNNs**. Our instructions have optimized granularity to guarantee the generality of computation modules. Moreover, instruction based control enables dynamic pipelining of operations. This hides the communication latency and increases the overall efficiency.
- **Acceleration for state-of-the-art LW-CNNs**. We test a set of benchmarks of seven LW-CNNs on *Light-OPU* for performance evaluation. The benchmarks are composed of MobileNet series [9, 10, 23], including the newly released MobileNetV3, as well as Xception [6], DenseNet [11], ShuffleNet [40] and SqueezeNet [12]. All networks can be accelerated without any network architecture modification while achieving 1.3× to 8.4× better power efficiency and up to 172× lower latency compared with state-of-the-art designs [1, 17, 20, 27, 32, 33, 41].

The rest of the paper is organized as follows. Section 2 lists the motivation. Section 3 describes the *Light-OPU* instructions. Sections 4 and 5 explain the *Light-OPU* micro-architecture and the compiler, respectively. Section 6 presents our experiment results on various state-of-the-art LW-CNNs. Section 7 concludes the paper.

Table 1: Inference time (Batch=1) on NVIDIA Titan Xp GPU, model parameters and number of multiply-add operations. † indicates the ratio compared with that of VGG-19. Input size is 229×229 for Xception and 224×224 for others.

	Inference Time/ms	Speedup†	#Parameter Reduction†	#Operation Reduction†
VGG-19	5.50	1×	1× (138 M)	1× (20G)
SqueezeNetV1.1	1.60	3.43×	74.67×	57.58×
MobileNetV1	2.45	2.24×	32.62×	33.50×
MobileNetV2	3.34	1.65×	40.69×	66.89×
ShuffleNetV1	5.40	1.02×	74.67×	150.57×
Xception	6.44	0.85×	6.05×	4.37×
DenseNet-161	15.50	0.35×	4.85×	2.60×

2 MOTIVATION

2.1 Non-proportional operation reduction and speedup

Note that when running LW-CONV on GPU platforms, compared with conventional CNNs, the reduction on inference time of LW-CNNs is not proportional to their reduced number of parameters and multiply-add operations. Table 1 lists out the comparison of inference time, parameter number and operation number of LW-CNNs with conventional CNN VGG-19 [26]. It can be seen that the operation number of VGG-19 is 150.57× more than that of ShuffleNetV1, but their inference time on NVIDIA Titan XP GPU is basically the same. Moreover, MobileNetV1 has 33.5× fewer operation number but only gains a speedup of 2.24×. The possible reason is that light-weight operation, *e.g.,* DW-CONV, is more memory bounded than computation bounded. The operations per input element significantly drop compared with conventional convolution. However, CUDA cores are designated for computation-intensive workloads, and they cannot be efficiently utilized in such case. Despite DW-CONV, new lightweight operations still impede acceleration by GPU. As can be seen in Table 1, while MobileNetV2 has only 50% of the operation number compared with MobileNetV1, its execution time on GPU increases by 36%.

Table 2: Inference time on CPU with various number of cores.

CPU cores	1	2	4	6	8	10
V1 Latency (ms)	33.56	20.24	13.28	10.31	10.40	9.80
V2 Latency (ms)	27.05	17.27	13.39	11.14	11.35	10.72

As to multi-core CPU, MobileNetV2 has quickly diminished advantage for multi-core execution when compared with V1, because inverted residual operation and higher percentage of DW-CONV employed in V2 require extra memory accesses. This is shown

in Table 2, where the ratio of $Latency_{v1}/Latency_{v2}$ gradually decreases with the increase of CPU cores. In short, general acceleration platforms (*e.g.,* GPU and multi-core CPU) cannot handle LW operations efficiently. This calls for customized hardware architecture optimized for LW operations, and FPGA acceleration with low non-recurring engineering (NRE) cost is an appropriate candidate.

2.2 Uniform support for a variety of Models

Previous work accelerated LW-CNNs via optimizing hardware modules for different operations individually. For instance, [1] and [20] were specifically designed for MobileNetV2 and SqueezeNet, respectively. [27] applied separate modules for DW-CONV and conventional convolution without any resource sharing. Moreover, all intermediate feature maps (FMs) are stored on-chip to reduce expensive on-chip off-chip memory traffic, posing constraints on the size of intermediate FMs. DenseNet [11], with intensive concatenations of previous FMs, can introduce more than 10× on-chip memory overhead and cannot be fit in. Therefore, general support with efficient resource utilization for all special operations in LW-CNNs is required.

Light-OPU accelerates different lightweight operations under a unified hardware architecture. It also optimizes computation efficiency by our compilation framework.

3 INSTRUCTION SET ARCHITECTURE

Light-OPU is designed for general LW-CNN inference. We adopt the instruction framework from [37] with extra parameter settings for LW-operations specific. Moreover, improvements are made to the instruction execution mechanism for a more flexible and compact run-time execution.

We utilize a complex instruction set architecture, where each instruction can take up several hundreds of cycles to execute. Specifically, each instruction is composed of various number of 32-bit length short instructions. There are two types of short instructions: Conditional instruction (*C-type*) and Unconditional instruction (*U-type*). *C-type* instruction specifies target operations and sets operation trigger conditions. *U-type* instruction delivers corresponding operation parameters for its paired *C-type*. One *instruction unit* contains one *C-type* instruction with $0-n$ *U-type* instructions. One instruction block consisting of a number of basic units is fetched together and then distributed to various modules. The least significant bit of instruction indicates the end of current instruction block when its value is 0.

3.1 Instruction Types

C-type instruction contains operation (OP) code and trigger condition. OP code indicates the operation type and trigger condition defines the operation execution prerequisite.

We keep six main types of C-instructions defined in [37], *i,e.*, *Memory Read*, *Memory Write*, *Data Fetch*, *Compute*, *Post Process* and *Instruction Read*, then add extra control parameters for LW operations. Specifically:

- *Data Fetch* is improved to operate in two modes: (1) *FM reuse* mode corresponds to conventional convolution operation, where only the channel parallelism is explored. Fetched FM is reused for the computations of multiple output channels. The paralleled number of input and output channel can be run-time tuned; (2) Kernel weights reuse mode targets on DW-CONV operations. It explores intra-kernel parallelism to compensate for the limited channel parallelism in DW-CONV. Moreover, kernel weights are reused for computations of multiple FM windows.
- *Compute* controls all processing elements (PEs). One PE computes the inner product of two 1D vectors of length N. $N = 9$ is chosen for our *Light-OPU* (See section 4.1), which sufficiently guarantees the space exploration for different networks. Control parameters are added to switch *Compute* between *FM share* mode and *Kernel weights share* mode.
- *Post Process* takes care of extra non-computational-intensive operations, *e.g.,* Squeeze and Excitation (SE) block.

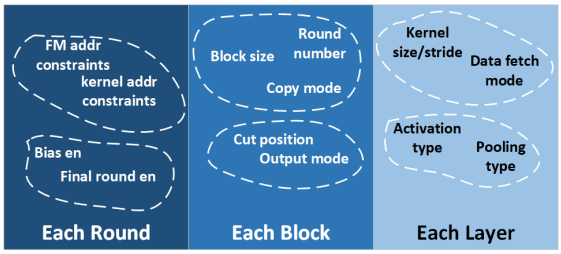

Figure 2: Grouping examples of parameters with different updating frequencies.

U-type instruction provides operation related parameters. In general, when operation pattern switches, only a subset of parameters are changed accordingly. For a certain *C-type* instruction, its corresponding parameters may have different updating rates. Therefore, as shown in Fig. 2, we group parameters with the similar updating rates into the same *U-type* instruction to minimize the total length of instruction sequences, which in turn reduces the memory access time and power consumption.

All the instructions are generated on an updating demand-based scheme, as a set of registers are provided to store the current parameters and trigger conditions until they get updated, which further reduces the length of instruction sequence.

3.2 Instruction Execution

We utilize dynamic pipeline fashion to organize our operations. Instead of fixing the instruction order within one layer, the order of our instruction units can be flexibly adjusted for different computation purpose. For efficient instruction control, we design a trigger condition list for each instruction, according to the dependency relationship among different operations under various operating patterns. Modifying the trigger condition index (TCI) by instruction at run-time sets the operation execution prerequisites. Using a dependency based execution strategy relaxes the order enforcement on instruction sequence, leaving enough room for the time uncertainty caused by memory related operations.

For example, Fig. 3 shows a fragment of the instruction execution process for one FM block's computation, where several instructions executed at different time points can be grouped together and read at the same time. Several *Instruction Read* are performed for TCI

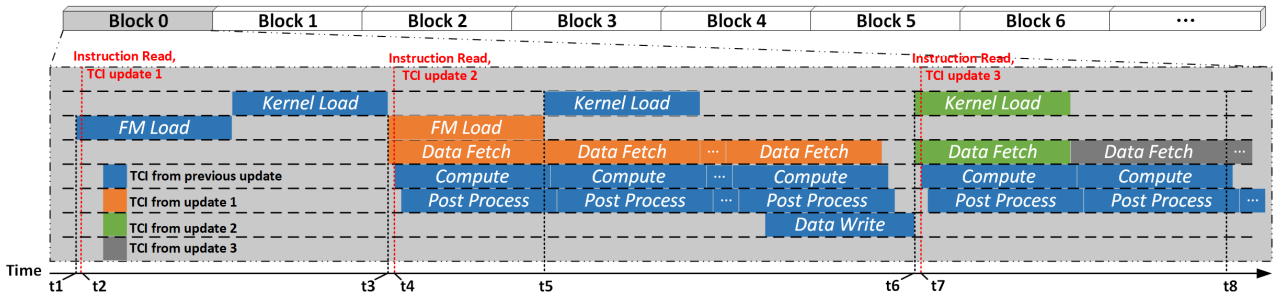

Figure 3: Instruction execution and TCI updates of time range $t1$ to $t8$. Red lines indicate TCI updates by *instruction read*. Each colored block shows the execution time range of one triggered instruction.

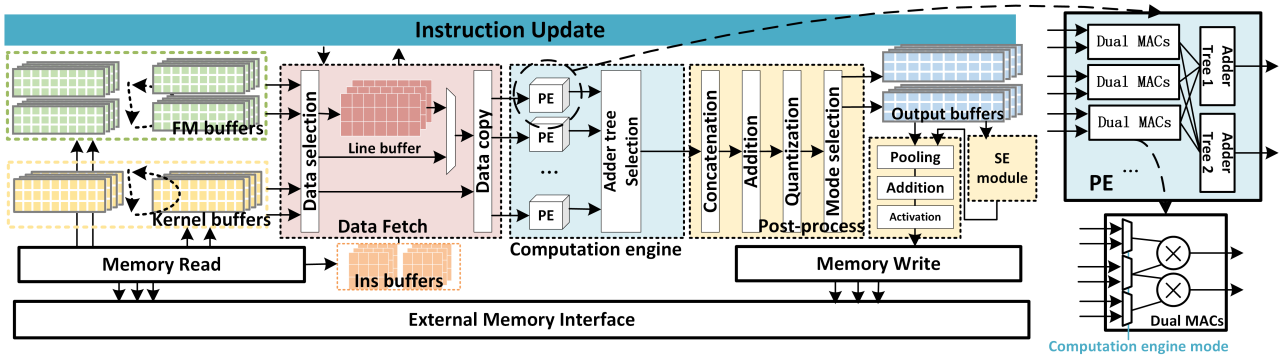

Figure 4: Overall micro-architecture and PE structure.

update, each labeled with one color. The color of instruction during execution process indicates the TCI it currently uses. Note that we update the next TCI right after the trigger of current TCI to make sure the operation will be triggered based on the new TCI next time. For example, *TCI update 1* is performed at time $t2$ right after the trigger of *FM load* at time $t1$. It can be seen that TCIs for *Compute*, *Post Process* and *Data Write* have not been updated within the time range plotted in the figure, where the instructions get executed multiple times whenever the preset condition is satisfied. Another example is *Kernel Load* operation, one mode of the *Memory Read* operations. For *Kernel Load* with TCI color blue, its trigger condition is the completion of *FM Load*. It gets executed twice until *TCI update 2*, labeled with color green, and updates its trigger condition to the completion of *Data Write*. Then at $t6$ *Kernel Load* labeled with green is triggered to pre-load kernel weights for the next round of computation that happens after $t8$.

4 MICRO-ARCHITECTURE

Hardware modules in *Light-OPU* are parameter tunable, which switch modes at run-time based on parameter registers updated by instructions. The computation engine is able to operate in different modes according to layer types in order to explore different combinations of parallelism.

As shown in Fig. 4, the *Light-OPU* micro-architecture is composed of *Memory Read*, *Memory Write*, *Data Fetch*, *Computation engine*, *Post-Process* and on-chip storage buffers. Each module accepts instruction updates from the *Instruction Update* control module. Micro-architectures only handle the computation of one sub-FM block. If the layer size is larger than the maximum block size allowed by hardware, the layer is sliced into sub-blocks by compiler to fit into hardware (See section 5).

4.1 Computation Engine

For layers in conventional CNN such as YOLO[22], GoogLeNet[30], VGG[26], ResNet[8], and Openpose[3], flattening the channel level computation guarantees enough parallelism for small to medium FPGA board resources. Moreover, the channel level parallelism is free from the architecture constraints posed by changeable kernel sizes, which ensures the generality of computation engine. However, the emerging LW-CNN comes with the wide application of DW-CONV, bringing challenges to the channel level parallelism based acceleration architectures. For a DW-CONV with n input channels and n output channels, each of the output FM channels is produced by one kernel channel convolving with only one input FM channel. Therefore, the explorable channel parallelism is reduced by $n\times$ compared with conventional convolution layer with the same input and output channel number. Considering the fact that conventional convolution layer is still widely used in LW-CNNs (*e.g.*, DenseNet, SqueezeNet and Xception), we develop two operation modes for the computation engine. With conventional mode targeting at traditional convolutional layers, channel parallelism is

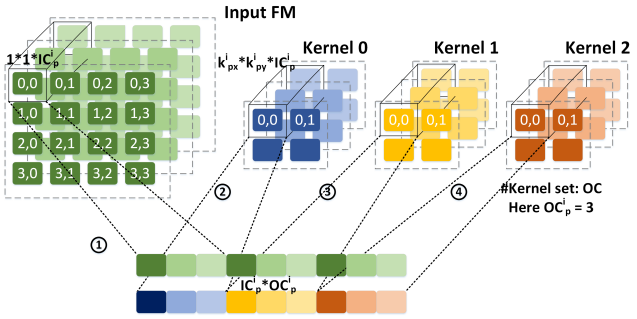

Figure 5: Conventional mode: Only the input and output channel parallelisms are explored. Kernel weights of size $k^i_{px} \times k^i_{py} \times IC^i_p$ are decomposed to $k^i_{px} \times k^i_{py}$ $1 \times 1 \times IC^i_p$ pointwise kernels. FM is copied for different output channel calculation.

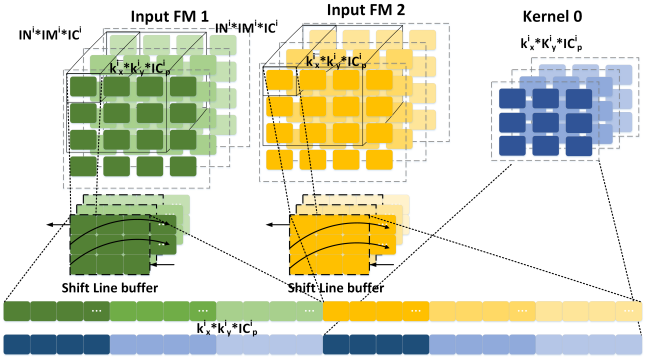

Figure 6: DW mode: three levels of parallelisms are explored. (1) Input&output channel level; (2) Intra-kernel level; (3) FM level, as input FMs are fetched from two FM blocks.

explored and FM gets reused. For DW-mode, multiple extra levels of parallelism are explored to handle the DW-CONV layer.

4.1.1 ***Conventional Mode.*** For conventional convolutional layers, we leverage channel parallelism. Fig. 5 explains how it works. For layer i, at each clock cycle, a slice of input channel of depth IC^i_p 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 explored for $IC^i_p \times OC^i_p$. For kernel weights of position $(0, 0)$, input FM channel slice from position $(0, 0)$ to $(2, 2)$ will be fetched out and perform corresponding multiplication. Then we move to kernel weights of position $(0, 1)$. Moreover, our design of the computation unit provides flexible combinations of $[IC^i_p, OC^i_p]$ pairs to accommodate for different layer configurations. By adding selective adder trees after PE array, the computation engine is able to efficiently handle the computation of $[IC^i_p, OC^i_p] = \{[128, 8], [64, 16], [32, 32], [16, 64]\}$. This computation pattern guarantees a uniform data fetching logic for any kernel size or stride, which greatly simplifies the data fetch module, and enables higher design frequency with less resource consumption.

4.1.2 ***DW Mode.*** For a $[channel_{in}, channel_{out}] = [64, 64]$ DW-CONV, if we use *Conventional mode* for the computation, only 64 multiplications in total can be done in parallel. Therefore, the purpose of introducing *DW mode* for our computation engine is to ensure high run-time resource efficiency of DW-CONV while sharing the same set of PEs with conventional CONV. This can be achieved with an extra data management module. Among all of our target DW-CNNs, the DW layers have a uniform small kernel size of 3×3 (expect for a few layers with kernel size 5×5 in the newly released MobileNetV3). Therefore, we make use of this property and build a typical shift line buffer structure for 3×3 FM window data fetch. The 5×5 kernel can be decomposed into several 3×3 kernels for adaptation. This leads to only less than 3% extra computation time in MobileNetV3 compared with having another line buffer for 5×5 window. As shown in Fig. 6, the shift register based line buffer reuses previous values to expand the available FM bandwidth. In this way, the intra-kernel parallelism can be explored and the parallelable multiplications increase to $64 \times 9 = 576$.

Moreover, we decompose each Xilinx DSP48E1 into two 8×8 multipliers to fully utilize computation resources. However, these two decomposed multipliers require sharing of one input due to hardware constraints. For *Conventional mode*, we share the same FM channel data between two different output channels. While for *DW mode*, one input FM channel only corresponds to one output channel. To solve the sharing problem, we fetch FM data from two different FM blocks and share the same kernel weights, as shown in Fig. 6.

4.2 Other LW operations handling

Apart from DW-CONV, LW-CNNs such as DenseNet [11] and ShuffleNet [40] introduce several other irregular operations which require extra handling.

4.2.1 ***Channel Shuffle.*** Introduced to increase information sharing among group convolutions, *Channel Shuffle*, explained in Fig. 7 (a), performs an important role in ShuffleNet. We label the results from three group convolutions with different colors. The original *Channel Shuffle* operation selects channels separately from each result and recombines channels to form the input of DW-CONV. Then the result of DW-CONV will be fed into another set of group convolutions. This shuffle scheme breaks up the continuous data storage format in memory, thus requiring multiple extra memory read and write operations. To implement the same shuffle scheme in a hardware-friendly way, we reorganize the shuffled results, as shown in Fig. 7(b). For each new group, channels from the same original group are put together as smaller groups. Correspondingly, the channel position of kernel weights and biases from following DW-CONVs and following group convolutions gets switched to match the new input order. We label them as Weights-Switched (WS) DW-CONV and WS Group CONV. This reorganization does not change the original shuffle scheme, but greatly simplifies hardware operations. We directly compute the small groups separately and write them to adjacent destination addresses. Therefore, the shuffled results can be formed naturally without any extra memory manipulation operations.

4.2.2 ***Group Convolution.*** ShuffleNet utilizes *group convolution* to relieve the computation burden from an increased number of

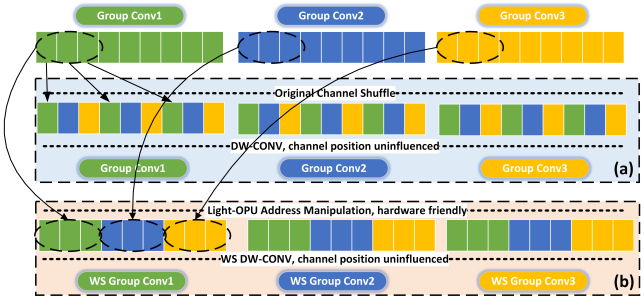

Figure 7: (a) *Channel Shuffle* operation and (b) its hardware-friendly implementation.

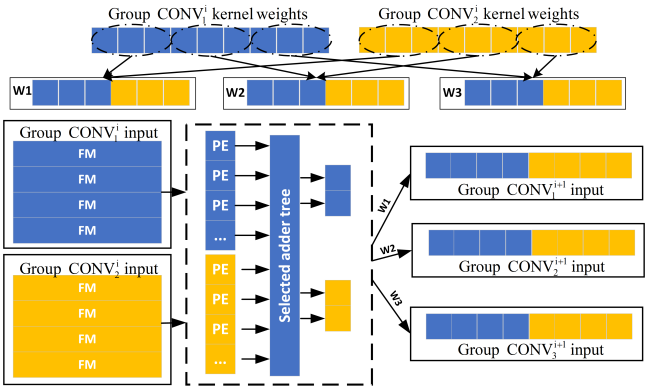

Figure 8: Calculation of two *group convolutions* in parallel.

channels. *group convolution* slices input FM into separate chucks in channel dimension and conducts individual convolution for each chuck, then concatenates the output FMs. Therefore, performing a *group convolution* can be simplified as calculating several conventional convolutions in sequence with input/output FMs address control, which helps fetch the input channel segments and concatenates output channels. However, as described in subsection 4.2.1, *group convolution* in ShuffleNet gets split into smaller convolutions for different output channel groups, which reduces the explorable parallelism and potentially leaves partial PEs idle.

To solve this issue, we fit two *group convolutions* into one round of computation. As shown in Fig. 8, we reorganize the kernel weights of *group* $CONV_1^i$ and *group* $CONV_2^i$ into $w1$, $w2$ and $w3$, each corresponding to one input of next set of three *group convolutions*, respectively. Meanwhile, input FMs for two *group convolutions* are fetched from different FM banks and sent to PE array together with reorganized weights. As a result, the output results get concatenated automatically and can be directly written back to memory. The parallelism of two *group convolutions* cuts the computation time in half without introducing extra memory manipulation operations.

4.2.3 ***Dense block***. *Dense block* requires channel concatenation, which is a typical operation in CNN algorithms. However, the implementation complexity of channel concatenation can be influenced by its position. In the hardware implementation, we conduct *Layer Grouping* (See section 5.1) to reduce off-chip memory access

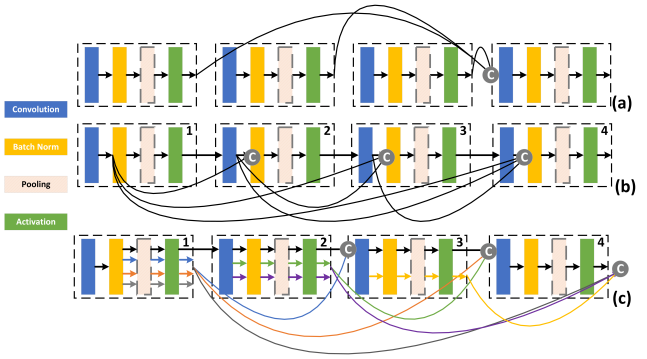

Figure 9: (a) Regular inception module concatenation. (b) *Dense block* concatenation. (c) Hardware-friendly *Dense block* concatenation implementation.

latency. Layers are grouped into one computation block that includes operations as [*memory_read* - *convolution* - (*Batch_Norm*) - *pooling* - (*SE*) - (*residual*) - *activation* - *memory_write*], which is performed in a data streaming fashion. Note that *Batch_Norm* can be merged, the position of *residual* can be flexibly adjusted, and all the operations except for *memory_read* and *memory_write* can be skipped using instruction control. In a typical inception module from GoogLenet [29], concatenation happens at the end of computation blocks, as shown in Fig. 9(a). Therefore, the address arrangement of the *memory_write* can perform concatenation.

However, in the *Dense Block* case, the sources of concatenation come within the computation block, as shown in Fig. 9(b). Moreover, the concatenation is also performed within the computation block instead of in the external memory. As a result, multiple extra memory writes and reads are required to fetch intermediate data, store it temporarily in memory, then send it to another computation block. To solve this issue, we adjust the order of computation within *Dense block* to make it hardware-friendly. For example, in original *Dense block* implementation, as shown in Fig.9(b), before getting read as the input of computation block 3, intermediate result from computation block 1 needs to go through [*memory_write* - *memory_read* - *Batch_Norm* - *pooling* - *activation* - *memory_write*]. While in our hardware-friendly implementation shown in Fig. 9(c), we move the computation in block 2 to block 1, then the same chunk of data only needs [*Batch_Norm* - *pooling* - *activation* - *memory_write*]. Benefiting from our flexible ISA and system control, multiple different *Batch_Norm* can be performed consecutively with one computation block in advance. The computation order adjustment of *Dense block* reduces 66% of the memory access operations.

4.2.4 ***Squeeze and Excitation (SE) block***. In MobileNetV3 [9], *SE block* is applied to weight channels for accuracy improvement. The computation increment brought by *SE block* is limited. However, ShuffleNet with *SE blocks* inserted is evaluated in [15], leading to 26% slow-down in GPU speed compared with original version. This indicates that the irregular structure of *SE block* could degrade computation efficiency of GPU. Therefore, a specific acceleration module is needed. We find that sharing the main computation

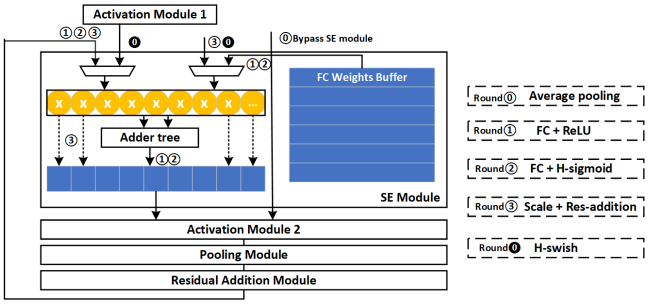

Figure 10: Calculation process of *SE block*.

engine for *SE block* leads to high memory access cost due to imbalanced computation cost and data requirement. Therefore, we insert a hardware *SE module* for the computation of *SE block* into the on-chip data flow, which avoids the off-chip data communication with small hardware resource cost. Calculation in *SE block* is shown in Fig. 10, where the circled number labels different data sources for different rounds of calculation. For example, when computing *FC (fully connected layer) + ReLU* ①, two inputs for the multiplier array are the results of average pooling and FC weights from the buffer. When computing round ③, one of the inputs is the FC results as the scaling factor, while the other input switches to the intermediate results kept in on-chip BRAM. For one *SE block* with input channel number $ch_{in} = 40$, input FM size $fm_{in} = 56$ and reduction ratio $r = 4$, the calculation takes 6324 cycles with no memory access latency (the weights for FC operation are pre-loaded during previous layer's calculation). Meanwhile, if we calculate the *SE block* using main computation engine, the calculation takes 6324 cycles with 6322 extra memory access latency for intermediate results write and read between rounds. Moreover, the multiplier array in SE module can be shared for the computation of activation function *H-swish* introduced by MobileNetV3, represented as follows:

$$out = \frac{x \times relu6(x+3)}{6}, \tag{1}$$

where two arrays of multipliers are needed. Thereby, the multiplier array in the SE module calculates $x \times relu6(x+3)$ as indicated by round ⓪ (black), and following *activation 2* module takes care of $\times\frac{1}{6}$.

5 COMPILER

In this section, we propose a compiler as the bridge between network configuration representation and *Light-OPU*'s hardware inference execution. The flow of compiler is shown in Fig. 11, mainly accomplishing two goals: (1) **Network Reformulation** that reformulates the network computation into hardware-friendly operations. **Network Reformulation** consists of the steps from *Network Configuration Parsing* to *Operation reordering*; (2) **Hardware Mapping** that maps the reformulated network into hardware with minimum execution latency. **Hardware Mapping** covers the steps from *Network slicing* to *Instruction generation*. The details of each target are discussed as follows.

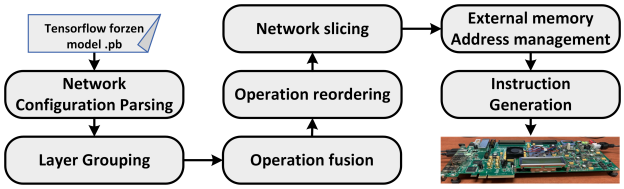

Figure 11: Compiler Flow.

5.1 Network Reformulation

Network Configuration Parsing extracts network structure related information, with input as the frozen model file generated by Tensorflow/Keras/ONNX. Layer parameters and connections are fetched and compressed for easy representation.

Layer Grouping is conducted to link adjacent layers into computation blocks. Each computation block is led by one convolution or fully connected layer, then followed by pooling / activation / residual layers. External memory access only happens between computation layers to reduce the communication latency.

Operation Fusion is a typical operation in hardware acceleration compilers [5]. Layers such as Batch Normalization can be completely merged into preceding convolution layers in some networks. Moreover, we merge the padding operation into the following convolution layer, where padding can be accomplished by zero data selection in *Data Fetch* module.

Operation Reordering is performed in section 4.2.1 and 4.2.4, where computation order arrangement is sometimes required to make the operation more hardware-friendly. Therefore, kernel weights reorganization and operation order switches are performed to handle the irregular operations introduced by LW-CNNs.

5.2 Hardware Mapping

In this stage, an automatic optimizer is applied to explore optimal slicing scheme that maps current architecture to overlay with maximum throughput.

Network Slicing. There are two levels of *Network slicing*, *i.e.,* 2D block slicing and channel slicing. For 2D block size slicing, the block size is constrained by the on-chip buffer size. For channel slicing, the $[Channel_{in}, Channel_{out}]$ combination is limited by the on-chip PE resources. Suppose an individual layer i is sliced into p^i blocks. Then each block is defined as $(IN_j^i, IM_j^i, IC_j^i, OC_j^i)$, with $j \in [0, p^i)$, where IN_j^i, IM_j^i, IC_j^i, and OC_j^i represent input block width, height, input channel number, and output channel number, respectively. Note that one sliced block is the FM input for one round of overlay computation, and kernel weights input can be calculated by parameter $[IC_j^i, OC_j^i]$.

If the layer type is conventional convolution, the inference latency L_j^i of one round's computation can be calculated by

$$memory_j^i = IN_{j+1}^i \times IM_{j+1}^i \times \left\lceil \frac{IC_{j+1}^i}{Bandwidth} \right\rceil + k_x^i \times k_y^i \times \left\lceil \frac{OC_{j+1}^i \times IC_{j+1}^i}{Bandwidth} \right\rceil + ON_{j+1}^i \times OM_{j+1}^i \times \left\lceil \frac{OC_{j+1}^i}{Bandwidth} \right\rceil$$

$$compute_j^i = (k_x^i \times k_y^i) \times ON_j^i \times OM_j^i \times \left\lceil \frac{\left\lceil \frac{IC_j^i}{MAC_{PE}} \right\rceil \times OC_j^i}{PE_{num}} \right\rceil,$$

$$L_j^i = max(memory_j^i, compute_j^i), \quad (2)$$

where ON_j^i and OM_j^i indicate the width and height of output block. $Bandwidth$ represents the off-chip memory bandwidth, which takes value 64 under current hardware platform and frequency. MAC_{PE} indicates the number of MACs implemented within one PE unit, which takes 9. PE_{num} indicates the number of PEs, which takes 128. $memory$ is the memory access time, including FM data reading and writing as well as kernel weights reading. Note that the kernel weights reading is only required at the first block of the whole layer. $compute$ is the computation time required for current block. The overall latency is defined as the maximum of $memory$ and $compute$, as we use computation time to hide memory access time.

If the layer type is DW convolution, we modify Eq. (2) into

$$memory_j^i = (IN_{j+1}^i \times IM_{j+1}^i + k_x^i \times k_y^i + ON_{j+1}^i \times OM_{j+1}^i) \times \left\lceil \frac{IC_{j+1}^i}{Bandwidth} \right\rceil$$

$$compute_j^i = \alpha \times \left\lceil \frac{ON_j^i \times OM_j^i \times IC_j^i}{2 \times PE_{num}} \right\rceil + IN_j^i \times 2 + 2,$$

$$L_j^i = max(memory_j^i, compute_j^i), \quad (3)$$

where α represents the kernel size adjustment coefficient, and takes value 1 for 3×3 kernel and 4 for 5×5 kernel. The 2 in the denominator of $compute_j^i$ indicates the two FM banks calculated in parallel (See section 4.1.2), and the $IN_j^i \times 2 + 2$ term represents pre-loading time for line buffers.

Therefore, the slicing optimization target can be represented as

$$\min_{\omega} \sum_i^{\hat{m}} (\sum_j^{p^i} L_j^i + memory_0^i)$$

$$\text{s.t.} \quad IN_j^i \times IM_j^i <= depth_{thres}$$

$$IC_j^i, OC_j^i <= width_{thres}, \quad (4)$$

where $depth_{thres}$ and $width_{thres}$ stand for the depth and width limit of on-chip BRAM, respectively. $memory_0^i$ represents the memory pre-loading time. $\hat{m}$ indicates the total number of layers after *Network reformulation*. ω represents a set of slicing scheme configurations, where each scheme defines p^i sliced block parameters for layer i, including both 2D block slicing and channel slicing.

We use an example to illustrate our slicing strategy. Suppose for a conventional convolution layer, we have $channel_{input} = 96$, $channel_{output} = 48$, $fm_{size} = 48 \times 48$ and $ker_{size} = 2 \times 2$. The constraints are set as $depth_{thres} = 2048$ and $width_{thres} = 64$.

For channel slicing, Fig. 12 shows the number of computation rounds required for different slicing schemes. For example, if we slice the layer channel into 2 blocks as $\{[64, 48] \times 2\}$, *i.e.*, reading 64 input channels for each block, we compute partial results of 48 output channels. The computation engine has mode [64, 16], thereby it takes $48/16 = 3$ rounds to finish one block's computation. In total, 6 rounds are required to complete the computation. Similarly, if we slice the channel as {[64, 48], [32, 64]}, it takes 3 rounds to compute

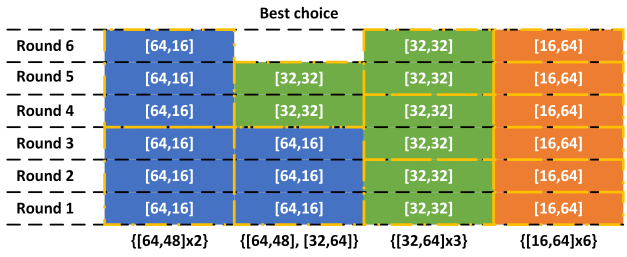

Figure 12: Channel slicing example. Each column represents one slicing strategy and each row represents one computation round. Labels on block indicates the number of $[channel_{in}, channel_{out}]$ calculated in this round.

[64, 48] via computation engine mode [64, 16], and 2 rounds to compute [32, 64] using different mode [32, 32]. In total, only 5 rounds are needed for the computation, which is the best choice.

For 2D block size slicing, apart from the intuitive rule of filling up the on-chip buffer, we also need to keep all the block size balanced to hide the memory access latency. As the sum of computation latency stays the same for different slicing strategies, only the memory access time leads to extra latency. From the constraints, we know that at least 4 blocks need to be sliced as FM size $48 \times 48 > 2048$. Suppose we apply the best channel slicing of {[64, 48], [32, 64]}. This layer will be sliced into $4 \times 2 = 8$ blocks in total, with each block size slicing corresponding to each channel slicing. If we slice the block size as $\{[3 \times 3], [45 \times 3], [3 \times 45], [45 \times 45]\}$, where minimal sized $[3 \times 3]$ block is calculated first since the FM and kernel load latency of the first block cannot be hidden. Then during the computation of block $\{block : [3 \times 3], channel : [64, 48]\}$, we need to load all the data of $\{block : [45 \times 3], channel : [64, 48]\}$. We have $computation = 108\ cycles$ and $memory = 270\ cycles$ based on Eq. (2). An extra 162 *cycles* of memory access latency cannot be hidden by the computation. In total, 3033 extra memory latency is induced. However, if we choose a balanced block slicing $\{[22 \times 22], [22 \times 23], [23 \times 22], [23 \times 23]\}$, only the pre-load memory access latency of 916 from the first block is required. Moreover, for balanced slicing scheme, the extra memory access latency stays the same regardless of block number. While for imbalanced slicing scheme, the extra latency increases with the increment of block number. In summary, the minimum latency slicing strategy of this example layer should be {[64, 48], [32, 64]} for channel slicing and $\{[22 \times 22], [22 \times 23], [23 \times 22], [23 \times 23]\}$ for 2D block size slicing.

6 EXPERIMENTS

We implement *Light-OPU* on Xilinx XC7K325T FPGA in a customized board with resource utilization shown in Table 3. The power consumption of FPGA board is measured using a PN2000 electricity usage monitor. A PC with Xeon 5600 CPU is used for

Table 3: FPGA resource utilization.

	LUT	FF	BRAM	DSP
Utilization	173522(85.14%)	241175(59.16%)	193.5(43.48%)	704(83.81%)

Table 4: Network benchmark statistics.

	MobileNetv1	MobileNetv2	MobileNetv3	SqueezeNetV1.1	DenseNet-161	ShuffleNetV1	Xception
Input size	224 × 224	224 × 224	224 × 224	224 × 224	224 × 224	224 × 224	229 × 229
Kernel size	including 1 × 1, 3 × 3, 5 × 5, 7 × 7						
Kernel stride	including 1 × 1, 2 × 2						
#Conventional CONV layer	15	36	50	26	160	2	40
#DW-CONV layer	13	17	15	0	0	16	34
#Group CONV layer	0	0	0	0	0	31	0
Activation Type	ReLu	ReLu	H-swish, H-sigmoid	ReLu	ReLu	ReLu	ReLu

Table 5: Network quantization accuracy (Top-1). Performance evaluated on ImageNet LSVRC-2012 dataset.

Network	32*bit* float-point	8*bit* fixed-point
DenseNet-161	77.1%	76.6%
MobileNetV3	68.4%	66.7%

Table 6: GPU and FPGA data sheets.

	Jetson TX2	*Light-OPU*
Technology(nm)	16	28
Frequency(MHZ)	1300	200
Peak GOPS	1300	460.8
Bit-width	32*bit* float-point	8*bit* fixed-point

Peak GOPS:Theoretical Peak GOPS when all PEs are used.

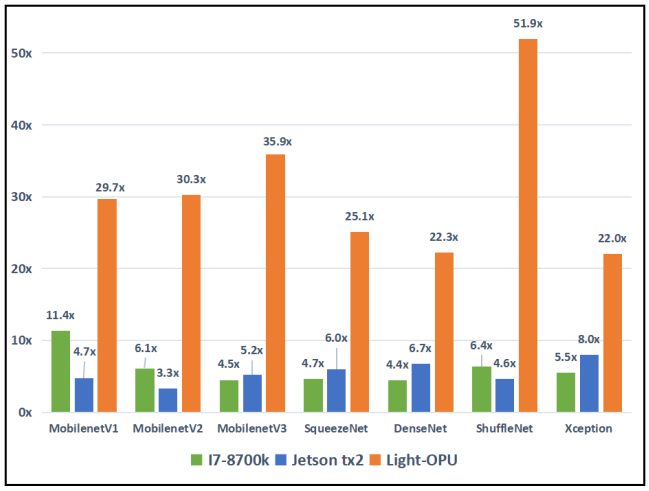

Figure 13: Latency comparison (Normalized over embedded CPU ARM Cortex-A57 data).

off-line software compiler. For hardware comparison, we use Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz, with power measured by Stress Terminal UI. We also compare our *Light-OPU* with edge GPU Jetson TX2. Inferences on GPU use batch = 1 mode for latency dominating evaluation, with power consumption all averaged over 500 runs.

6.1 Network Benchmarks

We use seven LW-CNNs including all major lightweight CNNs for a comprehensive evaluation. They are MobileNetV1 [10], MobileNetV2 [23], MobileNetV3 [9], SqueezeNet [12], DenseNet [11], Xception [6] and ShuffleNet [40], with statistics shown in Table 4. Different kernel sizes (1×1, 3×3, 5×5, 7×7), strides (1×1, 2×2), layer types (Conventional-CONV, DW-CONV, group-CONV) are covered. Irregular operations such as channel shuffle, residual addition and dense block concatenation are also included.

6.2 Network Quantization

With existing 8*bit* quantization for SqueezeNet [13, 19, 31], MobileNetV1 [25], MobileNetV2 [19], ShuffleNetV1 [31] and Xception [13], we quantize DenseNet-161 and newly released MobileNetV3 into 8*bit* using typical dynamic fixed-point quantization scheme in this paper and present accuracy in Table 5. Below, we use quantized networks for our experiments for FPGA acceleration. Note that CPU and GPU use floating point in our experiments.

6.3 Comparison with CPU and GPU

With GPU and FPGA information in Table 6, we compare the latency in Fig. 13 with all latency normalized with respect to ARM A57. Compared with CPUs, *Light-OPU* shows 30.6× and 4.9× speedup over embedded ARM and i7-8700k Intel on average. For GPU comparison, *Light-OPU* has on average a speedup of 5.5× compared with edge GPU TX2, which has 2.8× higher Peak GOPS. The latency advantage of *Light-OPU* over GPUs comes from its domain-specific ISA and micro-architecture tailored for CNN operations, especially LW-CNN operations. For example, *Light-OPU* shows 25.1/6.0 = 4.18× speedup over GPU Jetson TX2 when running SqueezeNet, which contains only conventional 3 × 3 and 1 × 1 convolutions. Meanwhile for MobileNetV2, speedup increases to 30.5/3.3 = 9.18×, because acceleration of DW-CONV layers and residual addition in MobileNetV2 is specifically optimized in *Light-OPU* (See section 4.1.2). Moreover, the weights reorganization and *group convolution* parallelism (See section 4.2.1 and 4.2.2) reduce the memory access latency and the number of operations for ShuffleNet, enabling *Light-OPU* to achieve 11.3× speedup compared with Jetson TX2.

We compare the power efficiency in Fig. 14, where the number of useful multiplication/ addition per Watt is utilized to evaluate the **power efficiency** of different networks, and the plotted performance is normalized with respect to ARM A57. *Light-OPU* shows

Table 7: Comparison with customized FPGA accelerators.

	[27]	[41]	[1]	***Light-OPU***			[32]	[33]	***Light-OPU***	[17]	[20]	***Light-OPU***
Year	2018	2018	2018	**2019**			2018	2019	**2019**	2018	2018	**2019**
Device	XCZU9EG	Stratix V 5SGSD8	Arria 10 Soc	**XCK325T**			Zynq7045	4×VCU118	**XCK325T**	XC7Z020	DE-10	**XCK325T**
Network	RR-MobileNet	MobileNetV1	MobileNetV2	**MobileNetV1**	**V2**	**V3-Large**	DenseNet-161			SqueezeNetV1.1		
Bit-width	8/4	16	8	**8**			16	16	**8**	32	8 float	**8**
DSP Utilization	1452	1641	1278	**704**			816	4×4993	**704**	192	<336	**704**
Frequency(MHZ)	150	133	150	**200**			125	200	**200**	100	100	**200**
FPS	127.4	231.2	266.2	**264.6**	**325.7**	**332.6**	11.69	153.8	**24.1**	2.6	9	**420.9**
Throughput/DSP (GOPS)	0.03	0.13	0.06	**0.21**	**0.14**	**0.12**	0.17	0.18	**0.25**	0.02	0.02	**0.19**
Power efficiency GOPS/W	NA	NA	4.9	**17.9**	**11.5**	**9.9**	38.9[a]	31.1[a]	**52.5[a]**	1.89	6.49	**15.83**

a: The power data with note (a) is subtracted with Idle power to match the evaluation method of reference power data.

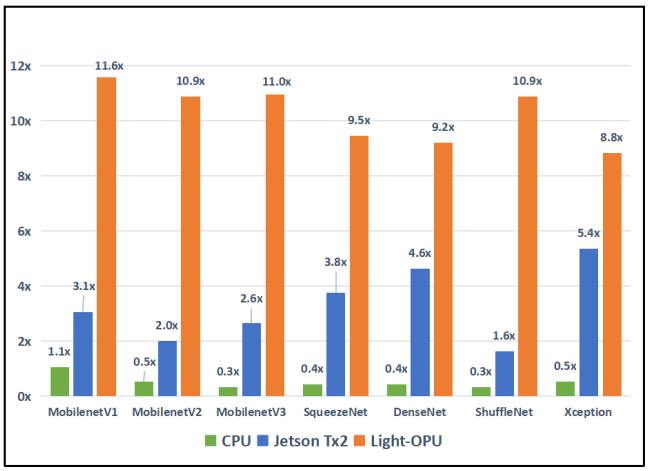

Figure 14: Power efficiency comparison (with power normalized over embedded CPU ARM Cortex-A57).

an average of 3.0× better power efficiency compared with GPU Jetson TX2. The superior performance of *Light-OPU* on both latency and power efficiency makes it ideal for various embedded real-time edge computing tasks, *e.g.*, detection, tracking and classification on robotic systems.

6.4 Comparison with FPGA Accelerators

To the best of our knowledge, all existing FPGA accelerators are designed specifically for a particular LW-CONV network, therefore we compare our general accelerator *Light-OPU* with customized accelerators. In Table 7, we use **frame per second (FPS)** for latency evaluation as all FPGA designs are running at batch = 1 mode. The **Throughput/DSP** is employed for the evaluation of run-time computation resource efficiency, which reflects the percentage of useful computation conducted by DSP on average during run-time. The **Throughput/DSP** is adjusted based on data-width for fair comparison, where values for 8*bit* system are multiplied with 0.5. For MobileNetV1/V2, *Light-OPU* performs 1.6× and 2.3× better in Throughput/DSP compared with existing customized designs, and also gains 2.3× higher power efficiency. For DenseNet-161, compared with [32], *Light-OPU* (with compression of data width) doubles FPS and power efficiency. Multiple-FPGA design in [33] has 153.8/24.1 = 6.4× higher FPS compared with *Light-OPU*, but it utilized 28× more DSP slices. Therefore, it has significantly worse power efficiency and per DSP performance. For SqueezeNet, *Light-OPU* exhibits up to 8.4× higher power efficiency compared with existing designs [17, 20]. Overall, *Light-OPU* has 1.39× to 8× improvement in terms of throughput per DSP.

Advantages of *Light-OPU* over existing accelerators are due to the following reasons: (1) Flexible instruction and control enables dynamic pipelining, which reduces off-chip communication time and latency; (2) 8*bit* data representation helps to fully utilize on-chip resources, improve throughput and reduce power consumption; (3) Flexible computation engine design and special handling of various operation in LW-CONV greatly improve the performance and power efficiency.

7 CONCLUSIONS

We have proposed *Light-OPU*, an FPGA-based overlay processor to accelerate a variety of lightweight CNNs (LW-CNNs). *Light-OPU* performs two levels of optimization: (1) Software-level network reformulation, including layer grouping, operation fusion and operation reordering, eliminates redundant memory access and reduces number of operations in LW-CNN; (2) Hardware-level micro-architecture is specifically designed for LW-CNN operations. Meanwhile, the micro-architecture can be used for conventional convolutional layer computation since it keeps all hardware features such as those from [36] for conventional CNNs. The flexible acceleration engine guarantees high run-time resource efficiency, and thereby leads to low latency and high power efficiency. *Light-OPU* achieves 5.5× better latency and 3.0× better power efficiency compared with edge computing targeted GPU Jetson TX2, and obtains 1.39× to 8× better throughput per DSP and 5× to 8.4× better power efficiency compared with recent FPGA accelerators for LW-CNNs. Moreover, *Light-OPU* is fully software programmable, and no FPGA reconfiguration is required for network and application switches. In contrast, existing FPGA accelerators are all designed for specific LW-CNNs.

REFERENCES

[1] Lin Bai, Yiming Zhao, and Xinming Huang. 2018. A CNN accelerator on FPGA using depthwise separable convolution. *IEEE Transactions on Circuits and Systems II: Express Briefs* 65, 10 (2018), 1415–1419.
[2] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. 2010. A programmable parallel accelerator for learning and classification. In *PACT*. ACM, 273–284.
[3] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. 2017. Realtime multi-person 2d pose estimation using part affinity fields. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7291–7299.
[4] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks.

In *ACM SIGARCH Computer Arch. News*, Vol. 38. 247–257.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[6] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1251–1258.

[7] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. 2009. Cnp: An fpga-based processor for convolutional networks. In *FPL*. IEEE, 32–37.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[9] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244* (2019).

[10] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[11] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. 2014. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869* (2014).

[12] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[13] Jun Haeng Lee, Sangwon Ha, Saerom Choi, Won-Jo Lee, and Seungwon Lee. 2018. Quantization for rapid deployment of deep neural networks. *arXiv preprint arXiv:1810.05488* (2018).

[14] Mohammad Loni, Masoud Daneshtalab, and Mikael Sjödin. 2018. ADONN: adaptive design of optimized deep neural networks for embedded systems. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 397–404.

[15] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 116–131.

[16] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 1–8.

[17] Panagiotis G Mousouliotis, Konstantinos L Panayiotou, Emmanouil G Tsardoulias, Loukas P Petrou, and Andreas L Symeonidis. 2018. Expanding a robot's life: Low power object recognition via FPGA-based DCNN deployment. In *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 1–4.

[18] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper* 2 (2015).

[19] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. 2018. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 580–595.

[20] Kathirgamaraja Pradeep, Kamalakkannan Kamalavasan, Ratnasegar Natheesan, and Ajith Pasqual. 2018. EdgeNet: SqueezeNet like Convolution Neural Network on Embedded FPGA. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 81–84.

[21] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 26–35.

[22] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. *arXiv preprint* (2017).

[23] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[24] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 17.

[25] Tao Sheng, Chen Feng, Shaojie Zhuo, Xiaopeng Zhang, Liang Shen, and Mickey Aleksic. 2018. A quantization-friendly separable convolution for mobilenets. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE, 14–18.

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

[27] Jiang Su, Julian Faraone, Junyi Liu, Yiren Zhao, David B Thomas, Philip HW Leong, and Peter YK Cheung. 2018. Redundancy-reduced MobileNet acceleration on reconfigurable logic for ImageNet classification. In *International Symposium on Applied Reconfigurable Computing*. Springer, 16–28.

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

[29] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *CVPR 2015*.

[30] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[31] Frederick Tung and Greg Mori. 2018. Deep neural network compression by in-parallel pruning-quantization. *IEEE transactions on pattern analysis and machine intelligence* (2018).

[32] Stylianos I Venieris and Christos-Savvas Bouganis. 2018. fpgaConvNet: mapping regular and irregular convolutional neural networks on FPGAs. *IEEE transactions on neural networks and learning systems* 30, 2 (2018), 326–342.

[33] Deguang Wang, Junzhong Shen, Mei Wen, and Chunyuan Zhang. 2019. An efficient design flow for accelerating complicated-connected CNNs on a multi-FPGA platform. In *Proceedings of the 48th International Conference on Parallel Processing*. ACM, 98.

[34] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 29.

[35] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, et al. 2019. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 23–32.

[36] Yunxuan Yu, Chen Wu, Xiao Shi, and Lei He. 2019. Overview of a FPGA-Based overlay processor. In *2019 China Semiconductor Technology International Conference (CSTIC)*. 1–3. https://doi.org/10.1109/CSTIC.2019.8755623

[37] Yunxuan Yu, Chen Wu, Tiandong Zhao, Kun Wang, and Lei He. 2019. OPU: An FPGA-Based overlay processor for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2019).

[38] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design*. ACM, 12.

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

[40] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6848–6856.

[41] Ruizhe Zhao, Xinyu Niu, and Wayne Luk. 2018. Automatic optimising CNN with depthwise separable convolution on FPGA:(Abstact only). In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 285–285.