A Convolutional Neural Network Accelerator for Power-Efficient Real-Time Vision Processing

Junghee Lee
School of Cybersecurity
Korea University, Seoul, Korea
Email: j_lee@korea.ac.kr

Chrysostomos Nicopoulos
Department of Electrical and Computer Engineering
University of Cyprus, Nicosia, Cyprus
Email: nicopoulos@ucy.ac.cy

***Abstract*—Deep Convolutional Neural Networks (CNN) constitute a promising framework for many applications. Such networks are often employed for vision processing algorithms, because CNNs offer better accuracy than traditional signal processing algorithms. However, it is challenging to apply high-accuracy deep CNNs for *real-time* vision processing, because they require high computational power and large data movement. Since general-purpose processors do not efficiently support CNNs, various hardware accelerators have been proposed. While it is required to support all the layers of the CNN for real-time vision processing, the large amount of weights (more than 100s of MB) limit the speedup of hardware acceleration, because the performance is largely bounded by memory access times. Recent CNN architectures, such as SqueezeNet and GoogLeNet, address this problem by employing narrow layers. However, their irregular architecture necessitates a re-design of hardware accelerators. In this paper, we propose a novel hardware accelerator for advanced CNNs aimed at realizing real-time vision processing with high accuracy.**

Keywords–Convolutional Neural Network; Hardware Accelerator; Scheduling.

I. INTRODUCTION

As unmanned vehicles and robotics keep evolving, there is a growing demand for power-efficient real-time vision processing. While deep Convolutional Neural Networks (CNN) offer high accuracy and are applicable to various vision processing algorithms, they are very challenging to employ for *real-time* vision processing, because of their high demand on computation and data movement. Various types of accelerators have been proposed based on Graphics Processing Units (GPU) [1], Multiprocessor Systems-on-Chip (MPSoC) [2], reconfigurable architectures [3], Field-Programmable Gate Arrays (FPGA) [4]–[6], in-memory computation [7], and dedicated hardware acceleration through Application Specific Integrated Circuits (ASIC) [8] [9].

A typical CNN architecture consists of a stack of convolutional and pooling layers, followed by classifier layers, as shown in Figure 1(a). To realize *real-time* vision processing, all layers of the CNN should run on an accelerator. Otherwise, the data transfer time between the host and the accelerator cancels out the acceleration in the computation itself. The challenge is in the processing of the classifier layer, where all neurons are fully connected. Award-winning high-accuracy CNNs (such as AlexNet [10], which won the 2012 ImageNet contest) usually require a huge number of weights (up to 100s of MB [7]) and weights are not reused.

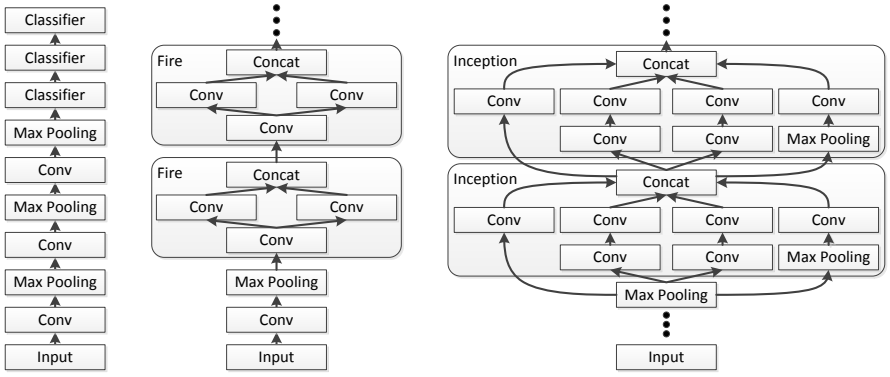
This challenge is being addressed by recent CNN architectures. Two representative examples are SqueezeNet [11] and GoogLeNet [12]. SqueezeNet offers comparable accuracy to AlexNet, but it uses 510 times fewer weights. GoogLeNet took the first place in the 2014 ILSVRC Classification contest. GoogLeNet employs narrow layers to minimize the number of weights, while offering high accuracy by using a large number of such narrow layers (more than 100). As shown in Figures 1(b) and (c), the SqueezeNet [11] and GoogLeNet [12] architectures are not as regular as the traditional CNN architecture of Figure 1(a).

To realize real-time vision processing, all layers of the CNN should run on the accelerator seamlessly. For example, Eyeriss [13] [8] requires reconfiguration of the accelerator for each layer. It takes 0.1 ms to configure one layer. If there are 100 layers, it takes 10 ms only for reconfiguration. ShiDianNao [9] addresses this by using hierarchical finite state machines. However, it is not proven with large-scale CNNs, such as SqueezeNet and GoogLeNet. Approaches using GPUs and FPGAs can execute all layers of the CNN quickly, but they consume an order of magnitude more power than ASIC designs. DaDianNao [14] offers low latency for all the layers of large-scale CNNs, but it consumes as much power as an FPGA, which may not be suitable for power-efficient vision processing. In general, an FPGA-based design cannot simply be implemented in an ASIC to boost power efficiency, due to the fundamental differences in the underlying design principles. Since the FPGA is programmable, the design can typically be customized to suit a particular CNN. This customization is not feasible in an ASIC. To support advanced CNNs like SqueezeNet and GoogLeNet in ASIC for real-time vision processing, we need a flexible – yet power-efficient – design that does not require run-time reconfiguration.

The proposed accelerator aims to achieve this goal by employing ***data-driven scheduling*** and ***modular design***. These two key features constitute *the novel contributions of this work*, since they enable the handling of *advanced CNNs without the need for reconfiguration*. The operation and destination of a Processing Element (PE) is determined at run-time upon receipt of data. The data is accompanied by metadata indicating the meaning of the data. By interpreting the metadata, a PE determines its schedule at run-time, which makes it easier to handle irregular CNN architectures. To achieve scalability, a modular design concept is employed with no shared resources and global synchronization being assumed. Each PE can only access its own local memory, and communicates only with its neighbors. Modular design facilitates deep pipelining, which enables further latency improvements by increasing the clock frequency. As a result, it is demonstrated by experiments that the proposed accelerator executes all layers of SqueezeNet and GoogLeNet in 14.30 and 27.12 million cycles with 64 processing elements. Assuming a 1 GHz clock speed, these latencies correspond to 14.30 ms and 27.12 ms, respectively, which is comparable to high-performance FPGA-based approaches (range of 1.06 ms to 262.9 ms [5] [6]). It is estimated that the proposed accelerator consumes 2.47 W and 2.51 W for SqueezeNet and GoogLeNet, respectively, which may be higher than power-efficient ASIC-based approaches (consuming 0.278 to 0.320 W [13] [9]), but it is significantly lower than FPGA-based approaches (that consume 8 to 18.61 W [4]) and DaDianNao [14] (that consumes 15.97 W).

After discussing related works in Section II, we present

 ISBN: 978-1-61208-748-1



(a) Traditional (b) SqueezeNet [11] (c) GoogLeNet [12]

Figure 1. Three different types of CNN architectures. The left one represents the traditional (generic) approach, while the other two represent two existing state-of-the-art approaches.

the proposed accelerator in Section III, and the details of the employed data-driven scheduling in Section IV. Section V provides experimental results, and Section VI concludes the paper.

II. Background and Related Work

Research in neural networks has a long history. Over the last several years, various types of approaches for the acceleration of CNNs have been studied. There is a trade-off between latency and power consumption among these accelerators. The GPU approach achieves 0.19 ms latency at 227 W [1], while FPGAs offer a range of 1.06 ms to 262.9 ms at 8 W to 18.61 W [4]–[6]. These values are measured under AlexNet [10] or VGG-16 (Visual Geometry Group 16) [15]. On the contrary, dedicated hardware accelerators implemented in ASIC target power-efficient implementations of small-scale CNNs, or the convolutional layers of large-scale CNNs [9]. For example Eyeriss [8] executes the convolutional layers of AlexNet [10] in 115.3 ms at 0.278 W [13].

Compared to two state-of-the-art CNN accelerators, the proposed accelerator offers lower latency and better scalability with the number of processing elements and clock frequency. Compared to Eyeriss [8], the proposed accelerator offers significantly lower latency through its modular design (that allows for higher clock frequencies), weight prefetching (optimized memory access patterns to Dynamic Random Access Memory (DRAM)), and by using larger on-chip memory. Additionally, the data-driven scheduling enables seamless execution of all layers without reconfiguration. ShiDianNao [9] also supports seamless execution of all layers, by storing all weights and feature maps in on-chip memory. However, the ShiDianNao [9] architecture was evaluated only with small-scale CNNs whose weights and feature map sizes fit into on-chip memory. Furthermore, both Eyeriss [8] and ShiDianNao [9] employ global shared memory, which renders their scalability questionable. In contrast, the modular design concept of the architecture proposed in this work enables high clock frequencies through pipelining. Even though the proposed accelerator requires more hardware and memory space to accommodate its data-driven scheduling and modular design, it is still significantly more power-efficient than FPGA-based approaches.

III. Overview of the Proposed Accelerator

A. Functional Requirements

The current implementation of the proposed accelerator supports three types of layers, and four types of layer connections. The four layers are: (1) convolutional layer, (2) max pooling layer, and (3) average pooling layer. The classifier layer can be implemented as a special case of the convolutional layer. SqeezeNet and GoogLeNet still use the classifier layer, even though it is not as big as those in traditional CNNs.

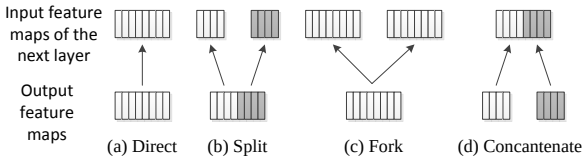


Figure 2. The 4 different types of layer connections supported by the proposed CNN accelerator that can be used to implement various CNN architectures.

To support a traditional/generic CNN, only one type of layer connection is enough, which is shown in Figure 2(a). To support more advanced CNN architectures, the proposed accelerator supports three other types of connections. The feature maps of a layer can be split and sent to different layers, as shown in Figure 2(b), and all feature maps can be sent to multiple layers, as shown in Figure 2(c). Finally, output feature maps of different layers can be concatenated as input feature maps of a layer, as shown in Figure 2(d).

The data-driven scheduling and modular design make it easy to support various types of layers and connections. Since the abovementioned three layers and four connections are enough to support SqueezeNet and GoogLeNet, the proposed accelerator only implements these for now, but it can be easily extended to cover other types of layers and connections. It is also possible to use heterogeneous PEs. These extension possibilities – and more – of the accelerator will be explored in our future work.

B. Architecture

For real-time vision processing, the speed of the feed-forward process is more important than that of the backward process, because the backward process is usually performed off-line during training. Thus, the proposed accelerator is focused on accelerating the feed-forward process.

Figure 3 illustrates the architecture of the proposed accelerator and presents the high-level details of one PE module. We assume that the accelerator is implemented as a separate chip. It receives inputs from and sends outputs to the host through

 ISBN: 978-1-61208-748-1

TABLE I. CONFIGURATION OF A LAYER TO BE STORED IN CONFIGURATION MEMORY.

Parameter	Description
R	Number of rows of an output feature map
C	Number of columns of an output feature map
M	Number of output feature maps
N	Number of input feature maps
K	Filter size
S	Stride
O	Number of next layers connected with this layer
T_n	The layer number of $n-th$ connected layer
F_n^{start}	Start feature map number of the $n-th$ connected layer
F_n^{end}	End feature map number of the $n-th$ connected layer
F_n^{shift}	Feature map number shift of the $n-th$ connected layer

a standard bus interface. It has its own main memory (e.g., DRAM), which is used to store weights.

The proposed accelerator consists of a number of PEs. All PEs are the same, but one of them is designated as an interface PE, which interacts with the host and memory. The PEs are connected by 1D rings. Two rings are used for data (activation) transfer, and the third ring is used for weight prefetching.

A PE consists of a communication interface, matching logic, functional units (multiplier and adder), an output Finite State Machine (FSM), and local memories for weights and feature maps. The matching logic determines whether the incoming activation is assigned to the PE or not. The matching logic makes a decision based on the mapping information, which is presented in the next section (subsection IV-A). If the incoming activation is accepted, it is pushed to a queue and processed by the functional unit. If the queue is full, the incoming activation cannot be accepted, even though it is destined to this PE. By interpreting the metadata accompanied by the activation, the corresponding functional unit is triggered. The result is stored in the local feature map memory, and transferred to other PEs when the computation is done.

IV. DATA-DRIVEN SCHEDULING

The heart of the proposed accelerator and its key novelty is *data-driven scheduling*. It enables the execution of advanced CNN architectures without reconfiguration. Each PE determines whether to accept an activation and the subsequent schedule of operations, based on metadata and the CNN's configuration. The metadata is accompanied by the activation coming from the interconnection network. The CNN configuration is transferred from the host through the interface PE, and stored in the local configuration memory.

Figure 4 shows examples of the metadata. The format of the metadata depends on the type of data. For example, for activations, the metadata includes the layer, feature map, and the position (row and column) of the activation. The position of an activation in the input feature map is denoted as `y` and `x`, that of a neuron in the output feature map is denoted as `row` and `col`, and that of a weight in a filter is denoted as `i` and `j` throughout this paper.

The configuration of layers is broadcasted to all PEs at initialization time, and it is stored in the local configuration memory of each PE. The configuration of one layer is shown in Table I.

The parameters R, C, M, N, K, and S are basic parameters of the CNN. Specifically, O and F are used to specify the connection, while F^{start} and F^{end} are used to support splits, and F^{shift} is used to support concatenation. For example, if a layer has 64 output feature maps, and 32 of them are sent to layer 1, and the remaining 32 are sent to layer 2, then O=2, T_0=1, F_0^{start}=0, F_0^{end}=31, F_0^{shift}=0, T_1=2, F_1^{start}=32, F_1^{end}=63, and F_1^{shift}=-32. In this case, F_1^{shift} is used to convert the feature map numbers 32–63 of the current layer to the feature map numbers 0–31 of the next layer. In a similar way, when feature maps of multiple layers are concatenated, the feature map numbers can be adjusted to become linear, by using the F^{shift} parameter.

A. Mapping

In the proposed accelerator architecture, the granularity of mapping is a feature map. A PE processes all neurons in its assigned feature maps. In this way, we can *avoid the sharing of weights among PEs*, which facilitates modular design. In other words, if a PE processes all the neurons of its assigned feature maps, it can store their weights in its local memory and other PEs do not need to access them.

Feature maps are assigned as a combination of input and output feature maps. As a toy example, let us suppose a layer has 2 input feature maps (`ifm0` and `ifm1`), and 2 output feature maps (`ofm0` and `ofm1`). If there are 2 PEs, one PE is assigned to `ifm0-ofm0` and `ifm1-ofm0`, and the other PE is assigned to `ifm0-ofm1` and `ifm1-ofm1`. In other words, each PE processes all input feature maps of its assigned output feature map. If there are 4 PEs, feature maps are spread out as PE0 to `ifm0-ofm0`, PE1 to `ifm1-ofm0`, PE2 to `ifm0-ofm1`, and PE3 to `ifm1-ofm1`. PE0 and PE1 produce partial sums of neurons for `ofm0`, and one of them must accumulate them. In the proposed accelerator, the PE processing the last input feature map of an output feature map is responsible to collect the partial sums from other PEs that are assigned to the same output feature map. In our toy example, PE0 should send its partial sums to PE1, so that PE1 can collect them and generate the final `ofm0`, while PE2 should send its partial sums to PE3, so that PE3 can generate the final `ofm1`.

To generalize this concept, we compute a feature map index for each combination of input and output feature maps, and a range of indices is assigned to PEs. The feature map index is computed as `index = ifm + ofm × M`, where `ifm` denotes the input feature map number, `ofm` is the output feature map number, and `M` is the total number of input feature maps. In the above toy example, the index of `ifm0-ofm0` is 0, `ifm1-ofm0` is 1, `ifm0-ofm1` is 2, and `ifm1-ofm1` is 3. If there are 2 PEs, PE0 is assigned to the range of indices from 0 to 1, and PE1 to indices from 2 to 3. If there are 3 PEs, PE0 is assigned to 0 and 1, PE1 to 2, and PE2 to 3. Thus, feature maps are not evenly distributed. If there are 4 PEs, each PE is assigned to each index.

The matching logic accepts an incoming activation, if its feature map falls within the range of the assigned indices. Recall that an activation is accompanied by metadata that includes the input feature map number, as shown in Figure 4. The pseudo code in Figure 5 shows how to determine if an activation, whose index is `ifm`, should be accepted or not, given a range of indices from `index_start` to `index_end`. Again, `M` indicates the total number of input feature maps.

Even if the activation is accepted, it should be forwarded to the next PE, because it may be used by the next PE. In fact, if there is a high enough number of output feature maps, as compared to the number of PEs, all PEs would need all input feature maps. Coming back to the toy example,

 ISBN: 978-1-61208-748-1

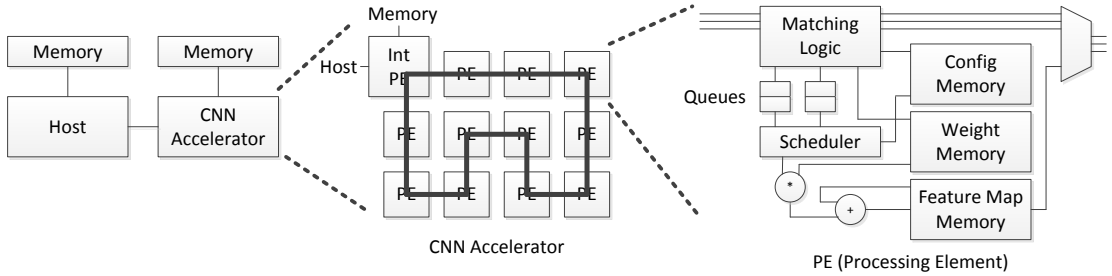


Figure 3. The architecture of the proposed accelerator and a high-level overview of one processing element. Note that the memory connected to the CNN accelerator on the leftmost diagram is connected only to the interface PE, i.e. it is not a shared memory. The pseudo codes of the 'Matching Logic' and the 'Scheduler' modules are presented, respectively, in Figure 5 and Figure 7.

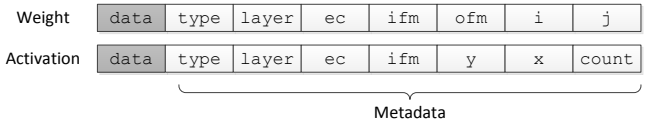


Figure 4. Examples of message formats, including the pertinent metadata. [ec: Escape channel; ifm: Input feature map number; ofm: Output feature map number.]

```
ofm_start =
  index_start % M <= ifm ?
  index_start / M : index_start / M + 1;
ofm_end =
  ifm <= index_end % M ?
  index_end / M : index_end / M - 1;
if(ofm_end >= ofm_start)
  activation accepted;
```

Figure 5. The pseudo code of the matching logic. The code determines if an activation should be accepted or not.

let us suppose there are 2 PEs. PE0 processes `ifm0-ofm0` and `ifm1-ofm0`, while PE1 processes `ifm0-ofm1` and `ifm1-ofm1`. Thus, both PE0 and PE1 need all input feature maps (`ifm0` and `ifm1`). Therefore, we designed the accelerator in such a way that activations are broadcast, and PEs determine if they are to be accepted. This is in contrast to sending activations to specific target destinations.

Due to resource constraints, an activation may not be accepted, even if it is destined to the particular PE. Because of this, we need to maintain two types of counters. One counter is to determine when the activation should be removed from the network. When the activation is injected into the network, the total number of output feature maps is attached to the metadata. Whenever a PE accepts the activation, it decrements this counter by the number of assigned output feature maps and forwards it to the next PE. When this counter reaches zero, it is no longer forwarded (i.e., it is removed from the network).

The other type of counter is for determining if the activation has already been accepted, or not. Because a ring is used as a communication fabric in the proposed accelerator, the same activation may arrive at the PE more than once, if it is not removed from the network. To check for this, a PE maintains a counter for each input feature map of a layer. The activations of an input feature map are accepted in a pre-determined order. In our implementation, all columns of a row are accepted in an increasing order of their column index, and those of the next rows are accepted in the same way. The counter counts how many activations of the input feature map have been accepted. Since activations are accepted in a specific order, if a PE knows how many have been accepted, the PE can determine

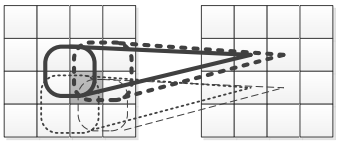


Figure 6. Illustration of how an activation is used for multiple filters.

what should come next. The activation is accepted only if the incoming activation is what the PE is expecting. In this way, the PE avoids accepting the same activation more than once.

In case of the max and average pooling layers, the number of input and output feature maps is always the same. An output feature map only needs one corresponding input feature map. Thus, those PEs that generate the final output feature map of the previous layer (which is the input feature map of the pooling layer) are assigned to process the corresponding output feature map of the pooling layer. In this way, we can eliminate unnecessary activation transfers.

B. Scheduling

Once an activation is accepted, all operations that need the activation are scheduled. To compute a neuron, its neighboring activations are required. The exact number of required activations depends on the size of a filter. In other words, an activation should be used by multiple filters.

Figure 6 shows an example. Let us suppose the filter size is 2 by 2 and the stride is 1. To compute a neuron at (1,1) of an output feature map, we need activations (neurons of input feature map) at (1,1), (1,2), (2,1), and (2,2). Similarly, neurons at (1,2), (2,1), and (2,2) of the output feature map need the same activation at (2,2) of the input feature map. If multiple output feature maps are assigned to the PE, neurons in other feature maps also need the incoming activation.

The pseudo code in Figure 7 shows how Multiply-And-Accumulate (MAC) operations are scheduled for an incoming activation. The `ofm_start` and `ofm_end` parameters are computed as shown in Figure 5. As shown in Figure 4, the position of the activation is given by `y` and `x`. The same mechanism is used for pooling layers. Instead of MAC operations, comparison (max pooling) or accumulation (average pooling) operations are scheduled.

The pseudo code is implemented as an FSM in the functional units. The FSM pops an activation from the queue located in-between the functional units and the matching logic in Figure 3. Once the FSM finishes all the scheduled operations, it pops the next activation from the queue. A functional unit

 ISBN: 978-1-61208-748-1

```
for(ofm=ofm_start; ofm<=ofm_end; ofm++)
  for(row=MIN(y/S, R-1); row>(y-K)/S && row>=0; row--)
   for(col=MIN(x/S, C-1); col>(x-K)/S && col>=0; col--) {
    i = y-row*S;
    j = x-col*S;
    feature_map[layer][ofm][row][col] +=
     weights[ofm][ifm][i][j] *
     activation
   }
```

Figure 7. The schedule of operations when an activation is accepted. [R: Number of rows of the output feature map; C: Number of columns of the output feature map; K: Filter size; S: Stride. All of the R, C, K, and S are of the current layer.]

TABLE II. THE DEFAULT SIMULATION PARAMETERS USED IN ALL EXPERIMENTS.

Parameter	SqueezeNet	GoogLeNet
Number of PEs	64	
Average memory access cycle	1	
Pipeline stages of communication channel	1	
Pipeline stages of functional units	1	
Queue depth	16	
Number of rings	3	
Configuration memory size	0.021 MB	0.092 MB
Weight memory size	1.289 MB	4.119 MB
Feature map memory size	9.132 MB	3.333 MB
Bit width of one activation ring	68	71
Bit width of the weight ring	58	61
Number of escape channels	10	46

accesses the weight memory and the feature map memory to perform its operation, and the result is stored in the feature map memory. To determine if accumulation is finished for one neuron, a counter is maintained for every neuron in the output feature map. The counter is stored in the feature map memory. The overhead of the memory will be discussed in Section V.

V. EVALUATION

A. Experimental Setup

We developed a cycle-level in-house simulator using SystemC [16]. The default simulation parameters are shown in Table II.

The proposed accelerator can take full advantage of the DRAM bandwidth, because the access pattern is always sequential. All feature maps are stored in the on-chip memory by adopting a sliding window technique, and the external DRAM is used only for weights. Since weights are prefetched in the order of layers, there is no need for random accesses to DRAM. Assuming the proposed accelerator runs at 1 GHz, then a 2 GB/s throughput is required to fetch one weight (16 bits) per cycle. According to the DDR4 standard, the maximum throughput can be up to 25.6 GB/s. Therefore, the DRAM throughput is high enough to easily supply one weight every cycle.

B. Performance Analysis

Table III shows the number of cycles required to execute *all layers* of SqueezeNet and GoogLeNet. Under the assumption that the proposed accelerator runs at 1 GHz (since ShiDianNao [9] also runs at 1 GHz), these results correspond to 14.30 ms and 27.12 ms for SqueezeNet and GoogLeNet, respectively.

Even though a direct comparison may not be meaningful due to fundamental differences in the design goals (low power vs. low latency) and benchmark (different CNNs), Eyeriss [13] is reported to execute the convolutional layers of AlexNet in 115.3 ms, and the convolutional layers of VGG-16 in 4309.5 ms. While a GPU executes all layers of these CNNs in 0.19 ms, FPGAs require 1.06 ms to 262.9 ms [1] [4]–[6]. The performance of the proposed accelerator is comparable to FPGA-based techniques. DaDianNao [14] offers even lower latency, but its power consumption is comparable to FPGA-based techniques. This is because it targets high-performance implementations supporting all the layers of large-scale CNNs and both the forward and backward processing steps.

TABLE III. NUMBER OF CYCLES REQUIRED TO EXECUTE ALL LAYERS OF THE CNN.

CNN	Number of cycles	Execution time*
SqeezeNet [11]	14,303,612	14.30 ms
GoogLeNet [12]	27,122,439	27.12 ms

* 1 GHz clock frequency is assumed.

TABLE IV. THE MAXIMUM SUPPORTED VALUES OF THE VARIOUS CNN CONFIGURATION PARAMETERS.

Parameter	Meaning	SqueezeNet	GoogLeNet
R	Rows	224	224
C	Columns	224	224
M	Input feature maps	1000	1000
N	Output feature maps	1000	1000
K	Filter size	7	7
S	Stride	2	2
O	Connections of a layer	2	4
T_n	Next layer	33	106
F_n^{start}	Start feature map	1000	1000
F_n^{end}	End feature map	1000	1000
F_n^{shift}	Feature map shift	1000	1000
Total number of layers		33	106
Total number of connections		40	204

It should also be noted that the proposed accelerator offers flexibility in that it can support SqueezeNet and GoogLeNet without run-time reconfiguration. Since SqueezeNet and GoogLeNet offer comparable accuracy with AlexNet and VGG-16, we believe they are good alternatives for power-efficient real-time vision processing.

On the other hand, ShiDianNao [9] reports 0.047 ms to execute all layers of ConvNN [17]. However, ConvNN is much smaller. For example, GoogLeNet requires 1502 million MAC operations, whereas ConvNN only needs 0.6 million. While it demonstrates an efficient implementation of small-scale CNNs, it is not proven with large-scale CNNs for high-accuracy vision processing algorithms.

C. Cost Analysis

To compute the *minimum* required memory size and the minimum required bit-width for the rings, it is essential to assess the *maximum* supported values of the parameters of the CNN configurations under investigation. These parameters are summarized in Table IV. The total number of layers used for the proposed accelerator is different from the number assumed in the original implementations of the CNN architectures. We slightly changed the architecture – in a mathematically equivalent manner – to better fit the underlying architecture of the accelerator. Specifically, instead of introducing an explicit concatenation layer, the output feature maps are directly connected to the next layer to reduce the memory requirement. Thus, if a pooling layer is followed by a concatenation layer, the pooling layer has to be split into the previous layers, because pooling layers are processed by the same PE where the output feature map is generated.

In the configuration memory, the basic parameters (R, C, M, N, K, S, and O) are stored for each layer and the connection parameters (T, F^{start}, F^{end}, and F^{shift}) are

TABLE V. THE MINIMUM REQUIRED MEMORY SIZES UNDER TWO DIFFERENT NUMBER REPRESENTATIONS.

Memory	SqueezeNet		GoogLeNet	
	16 bits	6 bits	16 bits	6 bits
Weight memory	1.289 MB	0.483 MB	4.119 MB	1.544 MB
Feature-map memory	9.132 MB	5.619 MB	3.333 MB	2.051 MB

stored for each connection. The total number of bits to required to store all of these is 2,793 and 12,106 for SqueezeNet and GoogLeNet, respectively. Since all PEs need to store them, the sum of the configuration memory size of all PEs is 0.021 MB and 0.092 MB for SqueezeNet and GoogLeNet, respectively, as shown in Table II.

The minimum size of the weight and feature-map memories varies for different PEs, depending on the feature map assignment. For regularity, we used the same memory size across all PEs. The proposed accelerator does not depend on the type of number representation. All analysis results shown so far is based on 16-bit fixed-point representation, which is the most popular setup in previous efforts. If, instead, we adopt 6-bit representation [18], the memory size can be further reduced. Table V shows both cases.

Obviously, the memory size required for the proposed accelerator is significantly larger than that of existing accelerators. This is because the design goal of the proposed accelerator is to minimize latency as much as possible at a reasonable hardware cost. Considering the fact that recent Intel processors employ 8 MB of L3 cache and multiple 256 KB L2 and 32 KB L1 caches and DaDianNao [14] has a 36 MB embedded on-chip DRAM, we believe that 10 MB of on-chip memory is affordable for a stand-alone hardware-based CNN accelerator.

D. Power Estimation

It is estimated that the power consumption of the proposed accelerator is similar to ShiDianNao [9], which consumes 320.10 mW (except for the memory power, which will be discussed shortly), assuming an operating frequency of 1 GHz. Both designs run at the same clock frequency, employ the same number of PEs (64), and use the same types of functional units (multipliers and adders). The overhead of the control logic would obviously be different, but according to the analysis in Eyeriss [13], the power consumption of the control logic corresponds to only 9.5% to 10.0% of the total power budget. In general, the biggest consumer of power is the on-chip memory. Since the proposed accelerator employs a significantly larger memory, it consumes more power than ShiDianNao, which has a 288 KB on-chip memory. By using the per-access energy model of CACTI [19] and the number of memory accesses obtained through simulation, the power consumption of both the on-chip memory and DRAM can be estimated. Including the power consumption of the other components reported by ShiDianNao, the total power consumption (including DRAM accesses) of the proposed accelerator is estimated as 2.47 W and 2.51 W for SqueezeNet and GoogLeNet, respectively. Despite the fact that these numbers are based solely on estimation, it is clear that the power consumption of the proposed accelerator is significantly lower than FPGA-based approaches (that consume 8 to 18.61 W) and DaDianNao's 15.97 W [14].

VI. CONCLUSIONS

This paper proposes a novel hardware-based accelerator for deep CNNs used to realize *power-efficient real-time* vision processing. This attribute is enabled by *modular design*, optimized memory access patterns due to *weight prefetching*, and larger on-chip memory. More importantly, the new accelerator can execute *all layers* of SqueezeNet and GoogLeNet in 14.30 ms and 27.12 ms, respectively, which are comparable to high-performance FPGA-based approaches, but with significantly lower power consumption at 2.47 W and 2.51 W, respectively. The use of *data-driven scheduling* can seamlessly support advanced CNN architectures without any reconfiguration.

REFERENCES

[1] nVIDIA, "Tesla m4 gpu accelerator," 2016.

[2] C. Wang et al., "Cnn-based object detection solutions for embedded heterogeneous multicore socs," in 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Jan 2017, pp. 105–110.

[3] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Sept 2010, pp. 273–283.

[4] J. Qiu et al., "Going deeper with embedded fpga platform for convolutional neural network," in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 26–35.

[5] X. Wei et al., "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in ACM/EDAC/IEEE Design Automation Conference (DAC), June 2017, pp. 1–6.

[6] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™deep learning accelerator on arria 10," in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. New York, NY, USA: ACM, 2017, pp. 55–64.

[7] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: scalable and efficient neural network acceleration with 3d memory," in Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 751–764.

[8] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in Proceedings of the 43rd International Symposium on Computer Architecture, 2016, pp. 367–379.

[9] Z. Du et al., "ShiDianNao: shifting vision processing closer to the sensor," in 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), June 2015, pp. 92–104.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems 25, 2012, pp. 1097–1105.

[11] F. N. Iandola et al., "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," CoRR, vol. abs/1602.07360, 2016.

[12] C. Szegedy et al., "Going deeper with convolutions," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2015, pp. 1–9.

[13] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," IEEE Journal of Solid-State Circuits, vol. 52, no. 1, Jan 2017, pp. 127–138.

[14] Y. Chen et al., "Dadiannao: A machine-learning supercomputer," in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Dec 2014, pp. 609–622.

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

[16] Accellera, "Systemc 2.3.3," November 2018.

[17] M. Delakis and C. Garcia, "Text detection with convolutional neural networks," in International Conference on Computer Vision Theory and Applications, 2008, pp. 290–294.

[18] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," CoRR, vol. abs/1603.01025, 2016.

[19] S. J. E. Wilton and N. P. Jouppi, "Cacti: an enhanced cache access and cycle time model," IEEE Journal of Solid-State Circuits, vol. 31, no. 5, May 1996, pp. 677–688.

 ISBN: 978-1-61208-748-1