A Low-Latency Power-Efficient Convolutional Neural Network Accelerator for Vision Processing Algorithms

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) are expanding their territory to many applications, including vision processing algorithms. This is because CNNs achieve higher accuracy compared to traditional signal processing algorithms. For *real-time* vision processing, however, their high demand for computational power and data movement limits their applicability to battery-powered devices. For such applications that require both real-time processing and power efficiency, hardware accelerators are inevitable in meeting the requirements. Recent CNN frameworks, such as SqueezeNet and GoogLeNet, necessitate a redesign of hardware accelerators, because their irregular architectures cannot be supported efficiently by traditional hardware accelerators. In this paper, we propose a novel hardware accelerator for advanced CNNs aimed at realizing real-time vision processing with high accuracy. The proposed design employs data-driven scheduling that enables support for irregular CNN architectures without run-time reconfiguration, and it offers high scalability through its modular design concept. Specifically, the design's on-chip memory management and on-chip communication fabric are tailored to CNNs. As a result, the new accelerator completes all layers of SqueezeNet and GoogLeNet in 14.30 ms and 27.12 ms at 2.47 W and 2.51 W, respectively, with 64 processing elements. The performance offered by the proposed accelerator is comparable to high-performance FPGA-based approaches (that achieve 1.06 to 262.9 ms at 25 to 58 W), albeit with significantly lower power consumption. If the hardware budget allows, these latencies can be further reduced to 6.71 ms and 11.70 ms, respectively, with 256 processing elements. In comparison, the latency reported by existing architectures executing large-scale deep CNNs ranges from 115.3 ms to 4309.5 ms.**

Keywords–Convolutional neural network; Hardware accelerator; On-chip memory optimization; On-chip communication

I. Introduction

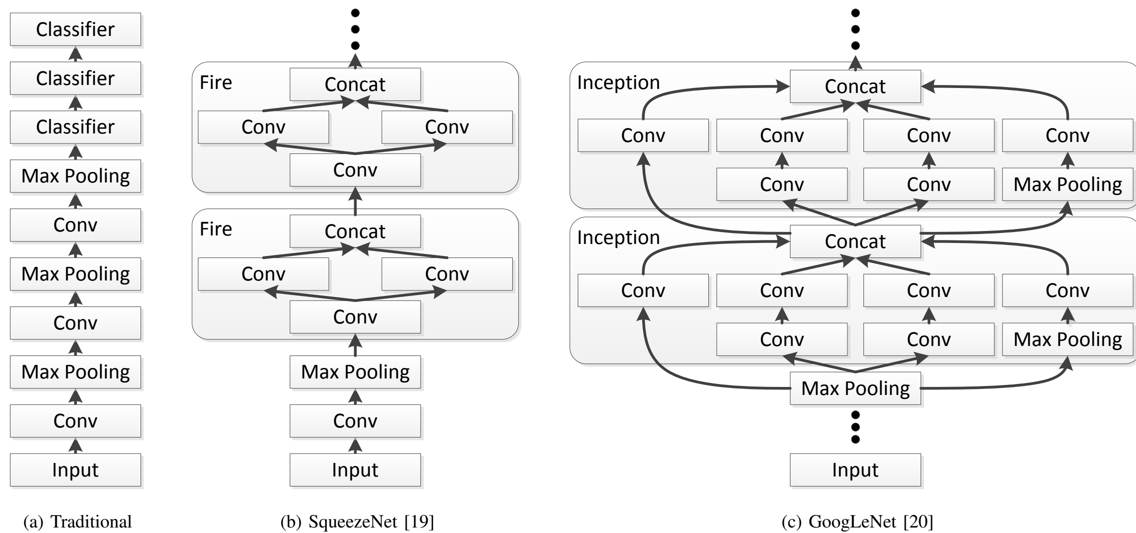
As unmanned vehicles and robotics keep evolving, there is a growing demand for power-efficient real-time vision processing. While deep Convolutional Neural Networks (CNN) offer high accuracy and are applicable to various vision processing algorithms, they are very challenging to employ for *real-time* vision processing, because of their high demand on computation and data movement [1]. It is well known that general-purpose processors cannot support CNN efficiently, because of their specific computational patterns [2]. Thus, various types of accelerators have been proposed based on Graphics Processing Units (GPU) [3], [4], Multiprocessor Systems-on-Chip (MPSoC) [5], [6], reconfigurable architectures [7]–[9], Field-Programmable Gate Arrays (FPGA) [2], [10], [11], analog circuits [12], in-memory computation [13], and dedicated hardware acceleration through Application Specific Integrated Circuits (ASIC) [14]–[17].

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 [18], which won the 2012 ImageNet contest) usually require a huge number of weights (up to 100s of MB [13]) and weights are not reused. The weights should be stored in an external memory (e.g., DRAM), and the performance is bounded by the memory access time [13].

This challenge is being addressed by recent CNN architectures. Two representative examples are SqueezeNet [19] and GoogLeNet [20]. 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 [19] and GoogLeNet [20] architectures are not as regular as the traditional CNN architecture of Figure 1(a). AlexNet [18] and VGG-16 [21] are often used to evaluate prior work. Nevertheless, if the goal is to achieve high-accuracy vision processing, we believe SqueezeNet and GoogLeNet are good substitutes, because they offer comparable accuracy and are better suited to hardware acceleration due to their use of fewer weights.

To realize real-time vision processing, all layers of the CNN should run on the accelerator seamlessly. For example, Eyeriss [14], [22] 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 [15] 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 [23] offers low latency for all

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



(a) Traditional (b) SqueezeNet [19] (c) GoogLeNet [20]

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 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. The accelerator has been enhanced from its original design [1] by employing on-chip memory optimization techniques, such as a sliding window and prefetching. 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 [24], [25]). 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 [15], [22]), but it is significantly lower than FPGA-based approaches (that consume 25 to 58 W [10], [24], [25]) and DaDianNao [23] (that consumes 15.97 W).

The rest of this paper is organized as follows: Section II discusses related work. After presenting the functional requirements and the architecture of the proposed accelerator in Section III, the details of the employed data-driven scheduling are explained in Section IV. In Section V, other salient features of the accelerator are described. Section VI provides experimental results, and Section VII 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. The design proposed in this paper relies on a fully digital ASIC implementation, using existing standard CMOS technology. We chose this approach, because it is practical (especially as compared to in-memory computation [13] and 3-D memory [26]), and we can potentially integrate a large number of PEs in a power-efficient manner (compared to FPGA implementations [2], [10]), as also acknowledged by [27]. The proposed accelerator can work with approximation [4], [28], compression [29], [30], and it can exploit the presence of zero weights [31]–[33].

There is a trade-off between latency and power consumption among these accelerators. The GPU approach achieves 0.19 ms latency at 227 W [34], while FPGAs offer a range of 1.06 ms to 262.9 ms at 25 W to 58 W [10], [24], [25]. These values are measured under AlexNet [18] or VGG-16 [21]. 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 [15], [27], [35], [36]. For example, Eyeriss [14] executes the convolutional layers of AlexNet [18] in 115.3 ms at 0.278 W [22].

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.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Compared to Eyeriss [14], the proposed accelerator offers significantly lower latency through its modular design (that allows for higher clock frequencies), weight prefetching (optimized memory access patterns to DRAM), and by using larger on-chip memory. Additionally, the data-driven scheduling enables seamless execution of all layers without reconfiguration. ShiDianNao [15] also supports seamless execution of all layers, by storing all weights and feature maps in on-chip memory. However, the ShiDianNao [15] architecture was evaluated only with small-scale CNNs whose weights and feature map sizes fit into on-chip memory. Furthermore, both Eyeriss [14] and ShiDianNao [15] 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. The pseudo codes of the three layers are shown in Figure 2.

```
for(row=0; row<R; row++)
  for(col=0; col<C; col++)
   for(ofm=0; ofm<M; ofm++)
    for(ifm=0; ifm<N; ifm++)
     for(i=0; i<K; i++)
      for(j=0; j<K; j++) {
       y = S*row+i;
       x = S*col+j;
       feature_map[layer][ofm][row][col] +=
        weights[ofm][ifm][i][j] *
        feature_map[prev_layer][ifm][y][x];
      }
```

(a) Convolutional layer

```
for(row=0; row<R; row++)
  for(col=0; col<C; col++)
   for(ofm=0; ofm<M; ofm++)
    for(i=0; i<K; i++)
     for(j=0; j<K; j++) {
      y = S*row+i;
      x = S*col+j;
      if(feature_map[layer][ofm][row][col] <
       feature_map[prev_layer][ofm][y][x])
        feature_map[layer][ofm][row][col] =
        feature_map[prev_layer][ofm][y][x];
     }
```

(b) Max pooling layer

```
for(row=0; row<R; row++)
  for(col=0; col<C; col++)
   for(ofm=0; ofm<M; ofm++) {
    for(i=0; i<K; i++)
     for(j=0; j<K; j++) {
      y = S*row+i;
      x = S*col+j;
      feature_map[layer][ofm][row][col] +=
       feature_map[prev_layer][ofm][y][x];
     }
    feature_map[layer][ofm][row][col] /= (K*K);
   }
```

(c) Average pooling layer

Figure 2. Pseudo codes of the 3 layers supported by the proposed accelerator. [R: Number of rows of the output feature map; C: Number of columns of the output feature map; M: Number of output feature maps; N: Number of input feature maps; K: Filter size; S: Stride. All of the R, C, M, N, K, and S are of the current layer.]

To support a traditional/generic CNN, only one type of layer connection is enough, which is shown in Figure 3(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 3(b), and all feature maps can be sent to multiple layers, as shown in Figure 3(c). Finally, output feature maps of different layers can be concatenated as input feature maps of a layer, as shown in Figure 3(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.

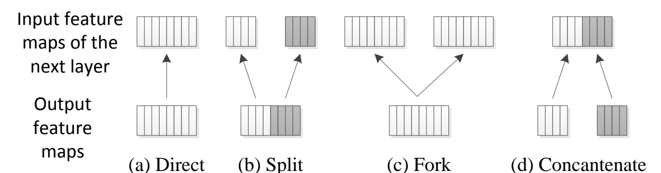


Figure 3. The 4 different types of layer connections supported by the proposed CNN accelerator that can be used to implement various CNN architectures.

B. Overall 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 4 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 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. The details of the communication architecture will be explained in Section V-B.

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 (Section IV-A). If the

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

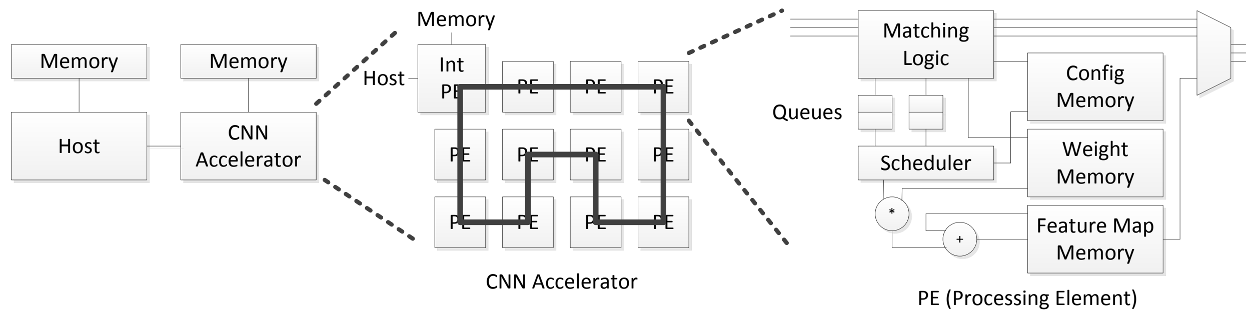


Figure 4. The architecture of the proposed accelerator and a high-level overview of one processing element. The pseudo codes of the 'Matching Logic' and the 'Scheduler' modules are presented, respectively, in Figure 6 and Figure 8.

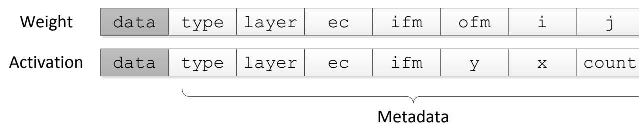


Figure 5. Examples of message formats, including the pertinent metadata. [ec: Escape channel; ifm: Input feature map number; ofm: Output feature map number.]

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 5 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. To make the notation consistent with the pseudo code in Figure 2, 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 proposal.

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.

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

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.

The rest of this section focuses on how data-driven scheduling is implemented.

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. However, this mapping strategy may incur load imbalance, because it is inherently coarse-grained. The issue of load imbalance will be discussed in Section VI.

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

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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

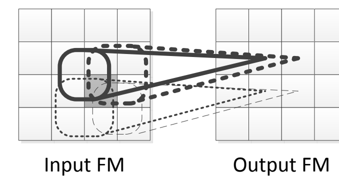


Figure 7. Illustration of how an activation is used for multiple filters.

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 5. The pseudo code in Figure 6 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, 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 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 7 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 8 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 6. As shown in Figure 5, 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

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
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 8. 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.]

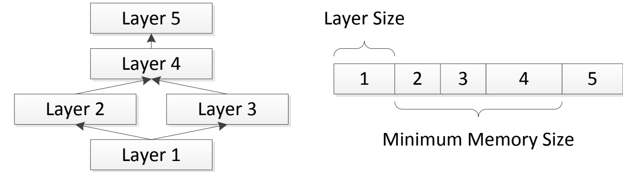


Figure 9. The minimum memory size requirement if parallel layers exist.

```
max = 0;
for(l=0; l<L; l++) {
   total_size=0;
   for all destination layers of layer l
     last = find the last destination layer;
   for(n=l; n<=last; n++)
     total_size += layer size of layer n;
   if(max < total_size)
     max = total_size;
   }
return max;
```

Figure 10. Pseudo code for determining the minimum memory size for each PE. The same pseudo code is used to determine the minimum size of the weight memory and the feature map memory. [L: Total number of layers.]

Figure 4. Once the FSM finishes all the scheduled operations, it pops the next activation from the queue. A functional unit 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 VI.

V. Accelerator Design

This section presents the salient features of the proposed CNN accelerator that support data-driven scheduling.

A. Memory Optimization

Since on-chip memory is usually much smaller compared to the size of the weights and feature maps, a sliding window technique is adopted to manage on-chip memories. This technique is used for both the weight and feature map memories. Once all computations in a layer are complete, the window of memory slides so that the completed layer is freed from the memory, and the next layer is allocated. The PEs are not synchronized for the sliding process. Each PE decides to slide the memory independently from others.

In the case of the weight memory, a prefetching technique is used. All weights are stored in the external memory and they are prefetched while computation is progressing. Weights are stored in the order of layers and are always accessed sequentially. Thus, weights can be prefetched by taking full advantage of the throughput. Initially, the weight memory is allocated to the first couple of layers (the exact number depends on the weight size and memory size). As an example, let us suppose 2 layers are initially allocated. Before starting the computation, the weight memory is filled with weights of the allocated layers. Once the first layer completes, it is freed, and the third layer is allocated. While the PE is working on the second layer (using the weights of the second layer in the memory), the weights of the third layer are prefetched.

A memory needs to be large enough to store at least two layers: one for the source layer, and the other for the destination layer of the activation transfer. The destination layer of the activation transfer may not be the right next layer. As illustrated in Figure 9, there may be parallel layers in the CNN architecture. The proposed accelerator processes one layer at a time. In the example of Figure 9, the activations from layer 2 should be sent to layer 4. Thus, the weight and feature map memories need to retain memory space for layers 2 through 4. The algorithm to determine the minimum size of memory is shown in Figure 10.

The layer size depends on the type of memory. In case of the weight memory, the size of a layer is equal to the *filter size* $\times$ *number of assigned feature map indices* $\times$ *size of one weight*. Recall that the feature map index is a combination of input and output feature map numbers. The size of one weight depends on the number representation. We may use a 16-bit fixed-point number, or a 6-bit log value [28]. The proposed accelerator is not tied to any particular number representation. The feature map memory has two parts. One part is for output feature maps and the other is for counters. The layer size of output feature maps is equal to the *number of neurons of one output feature map* $\times$ *number of assigned output feature maps* $\times$ *size of one neuron*. The size of one neuron also depends on the number representation. Similarly, the layer size of the counters is equal to the *number of neurons of one output feature map* $\times$ *number of assigned output feature maps* $\times$ *size of one counter*. The size of one counter is log of *the filter size* $\times$ *the number of input feature maps*.

B. Communication Architecture

The proposed accelerator supports three types of communication patterns.

- **Broadcasting:** Since activations to convolutional layers are used by many PEs (often all PEs), they are broadcast. However, activations to pooling layers are not injected into the network, because they are processed by the same PE.
- **Single-source unicasting:** Weights are always sent from the interface PE. Since weights are not shared by PEs, one weight is sent to one PE (unicasting). Other initialization messages are also always sent from the interface PE.
- **Peer-to-peer unicasting:** The majority of traffic falls under the previous two types. If there are not enough output feature maps compared to the number of PEs, multiple PEs are assigned to the same output feature map, and partial sums need to be sent to a designated PE.

The traffic patterns are not required to preserve the message delivery order, i.e., the order of message arrival can be different

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

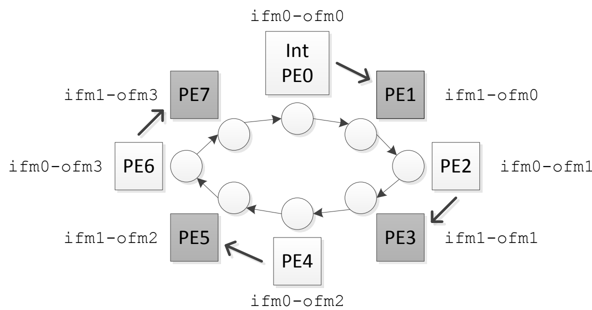


Figure 11. A logical view of the communication architecture of the proposed CNN accelerator. An example is given to illustrate how to minimize the hop distance under peer-to-peer unicasting.

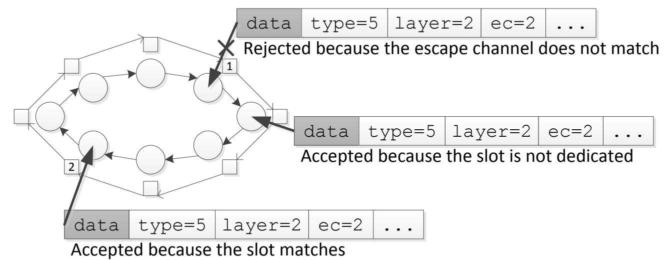


Figure 12. Employing *escape channels* ('ec') to avoid protocol-level deadlocks. Escape channels are implemented as slots.

from the order of departure for the same source-destination pair. Considering these patterns, we chose a uni-directional 1-D ring as a communication fabric, because it costs less than other packet-based Networks-on-Chip (NoC) [37]. The logical view of the communication architecture is shown in Figure 11. The topology is similar to that of Chain-NN [38], but we employ a ring instead of a systolic chain.

In the ring architecture, each PE interfaces with a ring stop. A message in the ring is ejected if it is destined for the local PE. Otherwise, it is forwarded to the next hop. A new message can be injected from the PE only if there is no message in the ring stop.

The ranges of feature map indices are assigned in such a way as to minimize the hop distance of peer-to-peer unicasting. As mentioned in Section IV-A, 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. If feature map indices are assigned in the same order as the topological order in the ring, the hop distance can be minimized. An example is given in Figure 11. In this example, `ofm0` is assigned to PE0 and PE1. Since PE1 processes the last input feature map, `ifm1`, it is designated to accumulate the partial sums and generate the final output. Since PE1 is located after PE0 in the ring topology, the partial sum sent from PE0 to PE1 takes only one hop.

To avoid protocol-level deadlocks, the concept of an *escape channel* is adopted. The ring itself does not cause network-level deadlocks, but because of the cyclic dependency caused by the upper-level protocol, deadlocks may occur. If the weight and feature map memories were infinite, there would be no chance of deadlocks, because the cyclic dependency would be broken by the memory. However, because the memory employs sliding, a later layer can only start when a previous layer completes, which forms a dependency from a later layer to a previous layer.

For example, let us suppose a CNN with 3 sequential layers and a PE have a feature map memory that is only enough to store two layers. At a certain moment, in PE0, layer 1 has completed, and activations are being transferred from layer 1 to layer 2. Layer 1 can be freed only after the transfer is complete, and layer 3 can then be allocated. In the proposed accelerator, PEs are not globally synchronized. Thus, another PE, say PE1, may have completed layer 2 and proceeded to layer 3. PE1 starts sending activations from layer 2 to layer 3. PE0 is also supposed to accept these activations for layer 3, but PE0 cannot, because the memory is not available. These activations can be removed from the ring only if all the assigned PEs accept them. Thus, until PE0 accepts these activations, they remain in the ring. The ring architecture employed by the proposed accelerator allows injection only if there is room in the ring. If the ring becomes full with these activations (from layer 2 to layer 3), PE0 may indefinitely not be able to receive activations from layer 1 to layer 2. This degenerate situation forms a cyclic dependency and causes deadlock.

To address this issue, an escape channel is introduced, and the escape channel is implemented as slots. As illustrated in Figure 12, slots are assigned and they are rotated in the ring. Each escape channel has one dedicated slot, and other slots can be used by all escape channels. Escape channels are assigned to layers. A different layer has a different escape channel. When an activation is generated, its escape channel is identified by the layer and put into the metadata, as shown in Figure 5. When an activation is to be injected, it can be injected only if its layer's escape channel matches with the slot. In this way, we can avoid the protocol-level deadlocks at a minimal cost.

The number of escape channels cannot be more than the number of ring stops (i.e., the number of PEs). In case of a deep CNN (e.g., GoogLeNet), the number of layers may be more than 100. A convolutional layer may need two escape channels, because it may need to send partial sums. Therefore, the number of escape channels should be optimized.

Since not all layers are active at a given time, we can reuse escape channels for the layers whose lifetime does not overlap. In fact, because of the data dependency between layers, no more than 3 layers can be processed at the same time. For example, let us suppose layer 1 is connected to layer 3 and layer 3 is connected to layer 5. Because there might be parallel layers, a layer may not be connected to the right next layer. The PEs are finished with layer 3 only if all activations are received from layer 1. Since the PEs are not globally synchronized, one of them may finish earlier than others and start layer 5. At this moment layers 1, 3, and 5 are active. However, because of data dependencies, no PE can process the layer after layer 5 until layer 3 completes. Since there are parallel layers between layers 1 and 5, the minimum number of escape channels is 10 in this case (2 of each layer times 5 layers), if all of the 5 layers are convolutional layers. Pooling layers do not need escape channels, because their activations are not injected into the network.

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

TABLE III. Number of cycles required to execute all layers of the CNN.

CNN	Number of cycles	Execution time*
SqeezeNet [19]	14,303,612	14.30 ms
GoogLeNet [20]	27,122,439	27.12 ms

* 1 GHz clock frequency is assumed.

VI. Evaluation

A. Experimental Setup

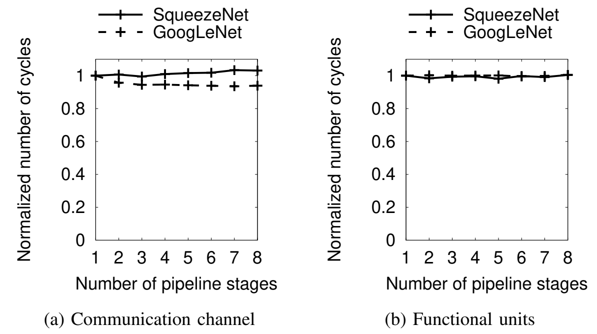
We developed a cycle-level in-house simulator using SystemC [39]. Since it is an architecture-level simulator, detailed analysis of the hardware cost is not available. However, we will discuss pipelining, which is related to clock speed, and the on-chip memory size, which has the most significant contribution to the hardware cost of the proposed accelerator. 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 [15] 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 [22] 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 [10], [24], [25], [34]. The performance of the proposed accelerator is comparable to FPGA-based techniques. DaDianNao [23] 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.



(a) Communication channel (b) Functional units

Figure 13. The number of pipeline stages does not have significant impact on the number of cycles required to complete the execution of the CNN. Thus, the performance of the proposed accelerator can potentially be enhanced by employing even higher clock frequencies.

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 [15] reports 0.047 ms to execute all layers of ConvNN [40]. 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. Another previous work [35] is reported to execute a particular CNN in 20.55 ms, but said CNN is also small (20.81 million operations).

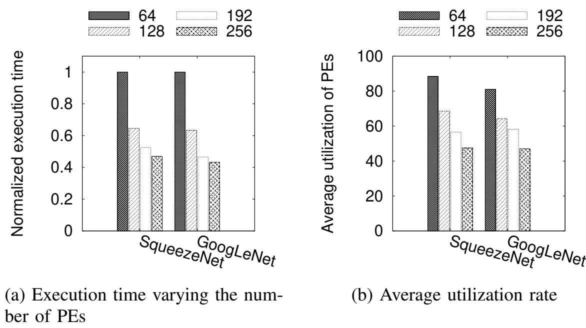
C. Scalability Analysis

If the budget allows, it is possible to further enhance the performance of the proposed accelerator by increasing the number of pipeline stages and the number of PEs. Figure 13 shows normalized number of cycles for SqueezeNet and GoogLeNet when the number of pipeline stages of the communication channel and the functional unit changes. In case of SqeezeNet, when the number of stages in the communication channel increases, there is a slight increase in the number of cycles. However, the increase is only 2.41% when the number of pipeline stages increases from 1 to 8.

In the case of pipelining the functional units, the pipeline may stall because of data hazards. However, all operations scheduled by an accepted activation are independent, because the operations are for different neurons. Data hazards happen only if there are overlapped neurons in the scheduled operations triggered by different activations. This probability is very low for convolutional layers. Even though the probability is relatively high for pooling layers, most of the cycles in the CNN are spent on convolutional layers. Therefore, the data hazards do not have significant impact on the number of cycles.

Figure 14 shows the normalized execution time and utilization rate when the number of PEs increases, up to 256. The execution time keeps decreasing and reaches 6.71 ms and 11.70 ms for SqueezeNet and GoogLeNet, respectively, when the number of PEs is 256. However, the utilization rate also decreases from 88.35% to 47.47% (SqueezeNet), and from

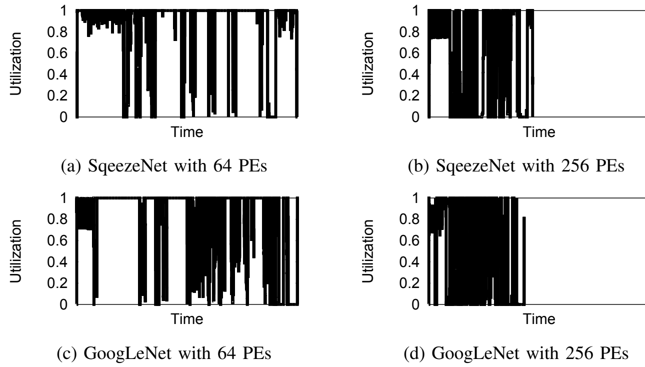
2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



(a) Execution time varying the number of PEs

(b) Average utilization rate

Figure 14. The execution time keeps decreasing with increasing number of PEs, but the utilization drops gradually.



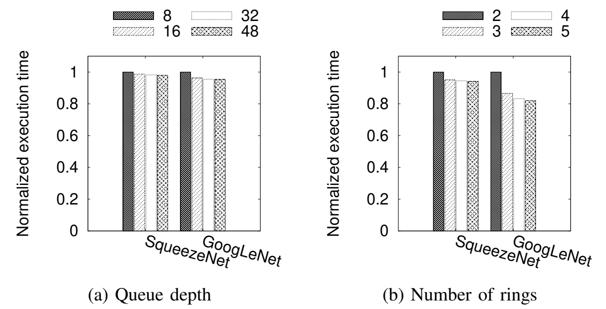
(a) SqeezeNet with 64 PEs

(b) SqeezeNet with 256 PEs

(c) GoogLeNet with 64 PEs

(d) GoogLeNet with 256 PEs

Figure 15. A utilization drop is observed in-between layers. This is attributed to load imbalance among the PEs.



(a) Queue depth

(b) Number of rings

Figure 16. Sensitivity analysis pertaining to the depth of the various queues and the number of employed interconnection rings. The chosen queue depth is 16 and the number of employed rings is 3, which are parameters shown to provide a good balance between performance and cost.

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

81.03% to 47.10% (GoogLeNet). We found that this is not due to lack of scalability, but due to load imbalance.

Figure 15 shows the utilization rate over time for 64 and 256 PEs. As shown in this Figure, the utilization rate often hits maximum (100%) even when 256 PEs are used, which means the proposed accelerator is scalable in terms of the number of PEs. Comparing (a) versus (b), and (c) versus (d), we can observe a utilization drop, which is more frequent with 256 PEs than 64 PEs. The utilization drop is observed in-between layers. Though no global synchronization is assumed, the PEs cannot proceed to the next layer until other PEs finish their computation, because of data dependencies. Since our mapping strategy is coarse-grained, the workload may not be evenly distributed. If the number of PEs increases, the size of the assigned workload decreases, which makes the load imbalance relatively more significant. We will address this issue by fine-grained load-balancing in our future work.

D. Sensitivity Analysis

We determined the queue depth and the number of rings based on the sensitivity analysis shown in Figure 16. Specifically, our experiments indicate that a queue depth of 16 strikes a good balance between performance and cost. Similarly, 3 communication rings are seen as a cost-effective tradeoff. Recall that one of the rings is dedicated to prefetching weights.

E. 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 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 minimum memory size is computed as explained in Section V. 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

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

popular setup in previous efforts. If, instead, we adopt 6-bit representation [28], the memory size can be further reduced. Table V shows both cases. Furthermore, since compression and pruning techniques [29]–[32] are also applicable to our accelerator, those techniques will be adopted in our future work.

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 [23] 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.

The longest message that the ring should carry is the activation, which is accompanied by the type of the message, the escape channel number, the layer number, the input feature map number, the position (`y` and `x`), and the counter, as shown in Figure 5. For future extensions, we assume 6 bits are used for the message type (i.e., 64 types of messages can be supported). The number of escape channels is computed as described in Section V, and is shown in Table II. The number of bits required to specify the layer, input feature map, and position can be calculated from Table IV. Since the maximum value of the counter is the number of output feature maps, 10 bits are assigned to this field. In total, 68 bits and 71 bits are required for one ring for SqueezeNet and GoogLeNet, respectively. For the ring employed for weight prefetching, the metadata includes the type of the message, escape channel number, layer number, input feature map number, output feature map number, and position (`i` and `j`), as shown in Figure 5. The total number of bits required for the weight ring is 58 and 61 for SqueezeNet and GoogLeNet, respectively.

F. Power Estimation

It is estimated that the power consumption of the proposed accelerator is similar to ShiDianNao [15], 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 [22], 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 [41] 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 25 to 58 W) and DaDianNao's 15.97 W [23].

VII. Conclusions and Future Work

This paper proposes a novel hardware-based accelerator for deep CNNs used to realize *power-efficient real-time* vision processing. The new design achieves significantly lower execution latencies than existing power-efficient ASIC-based accelerators, primarily due to its inherent ability to operate at higher clock frequencies. 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. We expect that the proposed accelerator will expedite the widespread adoption of CNNs for power-efficient real-time vision processing, which is especially useful in the domains of unmanned vehicles, autonomous robotics, and surveillance cameras.

The data-driven scheduling scheme introduced in this work was applied only to CNNs. Nevertheless, we believe that it could also be used in other types of neural networks, and, specifically, in more recent networks, such as Recurrent Neural Networks (RNN) [42], Faster R-CNN [43], You Only Look Once (YOLO) [44], and Single Shot Detector (SSD) [45]. Moreover, if the load imbalance when the number of PEs grows beyond 256 is adequately addressed, we expect that the latency can be reduced even further.

References

[1] J. Lee and C. Nicopoulos, "A convolutional neural network accelerator for power-efficient real-time vision processing," in Proceedings of the The Twelfth International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS), 2019, pp. 25–30.

[2] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '15, 2015, pp. 161–170.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 6, June 2017, pp. 1137–1149.

[4] M. Imani, M. Masich, D. Peroni, P. Wang, and T. Rosing, "Canna: Neural network acceleration using configurable approximation on gpgpu," in 2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC), Jan. 2018, pp. 682–689.

[5] C. Wang, Y. Wang, Y. Han, L. Song, Z. Quan, J. Li, and X. Li, "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.

[6] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in 2015 Design, Automation Test in Europe Conference Exhibition, March 2015, pp. 683–688.

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[7] 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.

[8] S. M. A. H. Jafri, T. N. Gia, S. Dytckov, M. Daneshtalab, A. Hemani, J. Plosila, and H. Tenhunen, "Neurocgra: A cgra with support for neural networks," in 2014 International Conference on High Performance Computing Simulation (HPCS), July 2014, pp. 506–511.

[9] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima, "A cgra-based approach for accelerating convolutional neural networks," in 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, Sept. 2015, pp. 73–80.

[10] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "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, 2016, pp. 26–35.

[11] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on fpga," in 2017 IEEE International Symposium on Circuits and Systems (ISCAS), May 2017, pp. 1–4.

[12] C. Mead and M. Ismail, Analog VLSI Implementation of Neural Systems. Springer, 2012.

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

[14] 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, ser. ISCA '16, 2016, pp. 367–379.

[15] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "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.

[16] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in 2013 IEEE 31st International Conference on Computer Design (ICCD), Oct. 2013, pp. 13–19.

[17] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '14, 2014, pp. 269–284.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1097–1105.

[19] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$1mb model size," CoRR, vol. abs/1602.07360, 2016.

[20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2015.

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

[22] 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.

[23] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Dec. 2014, pp. 609–622.

[24] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), June 2017, pp. 1–6.

[25] 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, ser. FPGA '17, 2017, pp. 55–64.

[26] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in Proceedings of the 43rd International Symposium on Computer Architecture, 2016, pp. 380–392.

[27] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in Proceedings of 2010 IEEE International Symposium on Circuits and Systems, May 2010, pp. 257–260.

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

[29] Y. Wang, H. Li, and X. Li, "Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices," in Proceedings of the 35th International Conference on Computer-Aided Design, ser. ICCAD '16, 2016, pp. 13:1–13:6.

[30] J. Zhu, Z. Qian, and C. Y. Tsui, "Bhnn: A memory-efficient accelerator for compressing deep neural networks with blocked hashing techniques," in 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Jan. 2017, pp. 690–695.

[31] D. Kim, J. Ahn, and S. Yoo, "A novel zero weight/activation-aware hardware architecture of convolutional neural network," in Design, Automation Test in Europe Conference Exhibition (DATE), 2017, March 2017, pp. 1462–1467.

[32] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in International Conference on Learning Representations, 2016.

[33] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), June 2016, pp. 1–13.

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

[35] A. Dundar, J. Jin, B. Martini, and E. Culurciello, "Embedded streaming deep neural networks accelerator with applications," IEEE Transactions on Neural Networks and Learning Systems, vol. 28, no. 7, July 2017, pp. 1572–1583.

[36] A. Dundar, J. Jin, V. Gokhale, B. Martini, and E. Culurciello, "Memory access optimized routing scheme for deep networks on a mobile coprocessor," in 2014 IEEE High Performance Extreme Computing Conference (HPEC), Sept. 2014, pp. 1–6.

[37] J. Lee, C. Nicopoulos, H. G. LEE, and J. Kim, "TornadoNoC: a lightweight and scalable on-chip network architecture for the many-core era," ACM Trans. Archit. Code Optim., vol. 10, no. 4, 2013, pp. 56:1–56:30.

[38] S. Wang, D. Zhou, X. Han, and T. Yoshimura, "Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks," in Design, Automation Test in Europe Conference Exhibition (DATE), 2017, March 2017, pp. 1032–1037.

[39] SystemC, 2012. [Online]. Available: www.accellera.org

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

[41] 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.

[42] A. Graves, A. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 6645–6649.

[43] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," CoRR, vol. abs/1506.01497, 2015. [Online]. Available: http://arxiv.org/abs/1506.01497

[44] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," CoRR, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[45] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: single shot multibox detector," CoRR, vol. abs/1512.02325, 2015. [Online]. Available: http://arxiv.org/abs/1512.02325

2020, © Copyright by authors, Published under agreement with IARIA - www.iaria.org