Runtime Design Space Exploration and Mapping of DCNNs for the Ultra-Low-Power Orlando SoC

AHMET ERDEM and CRISTINA SILVANO, Politecnico di Milano
THOMAS BOESCH, ANDREA CARLO ORNSTEIN, SURINDER-PAL SINGH, and
GIUSEPPE DESOLI, STMicroelectronics

Recent trends in deep convolutional neural networks (DCNNs) impose hardware accelerators as a viable solution for computer vision and speech recognition. The Orlando SoC architecture from STMicroelectronics targets exactly this class of problems by integrating hardware-accelerated convolutional blocks together with DSPs and on-chip memory resources to enable energy-efficient designs of DCNNs. The main advantage of the Orlando platform is to have runtime configurable convolutional accelerators that can adapt to different DCNN workloads. This opens new challenges for mapping the computation to the accelerators and for managing the on-chip resources efficiently. In this work, we propose a runtime design space exploration and mapping methodology for runtime resource management in terms of on-chip memory, convolutional accelerators, and external bandwidth. Experimental results are reported in terms of power/performance scalability, Pareto analysis, mapping adaptivity, and accelerator utilization for the Orlando architecture mapping the VGG-16, Tiny-Yolo(v2), and MobileNet topologies.

CCS Concepts: • **Computer systems organization** → **Neural networks**; *Heterogeneous (hybrid) systems*; *System on a chip*; • **Hardware** → **On-chip resource management**;

Additional Key Words and Phrases: Ultra low-power embedded systems, hardware acceleration, convolutional neural networks, design space exploration

ACM Reference format:

Ahmet Erdem, Cristina Silvano, Thomas Boesch, Andrea Carlo Ornstein, Surinder-Pal Singh, and Giuseppe Desoli. 2020. Runtime Design Space Exploration and Mapping of DCNNs for the Ultra-Low-Power Orlando SoC. *ACM Trans. Archit. Code Optim.* 17, 2, Article 11 (May 2020), 25 pages.

<https://doi.org/10.1145/3379933>

1 INTRODUCTION

Research on machine learning has been more and more focused on convolutional neural networks (CNNs) as the dominant winning paradigm across a wide range of complex application domains such as image recognition and classification. To deploy CNN solutions in embedded mobile devices, hardware acceleration plays a key role in real-time operation modes with ultra-low power

Authors' addresses: A. Erdem and C. Silvano, Politecnico di Milano, Piazza Leonardo da Vinci, 32, Milan, 20133, Italy; emails: {ahmet.erdem, cristina.silvano}@polimi.it; T. Boesch and A. C. Ornstein, STMicroelectronics, 39, Chemin du Champ des Filles, Plan-Les-Ouates, Geneva, 1228, Switzerland; emails: {thomas.boesch, andrea.ornstein}@st.com; S.-P. Singh, STMicroelectronics, Plot No.1, Knowledge Park III, Greater Noida, 201308, India; email: surinder-pal.singh@st.com; G. Desoli, STMicroelectronics, Via Tolomeo, 1, Cornaredo, 20010, Italy; email: giuseppe.desoli@st.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2020/05-ART11

<https://doi.org/10.1145/3379933>

Fig. 1. CNN layer topologies.

consumption overcoming the limitations of fully programmable solutions. The main challenge of recent neural network (NN) accelerators is to sustain the maximum throughput for convolutional stages while fitting state-of-the-art CNN deep topologies with many layers and millions of parameters given the constraints imposed on the memory bandwidth, power consumption, and area costs.

In deep learning literature, recently many different deep convolutional neural network (DCNN) topologies have been introduced, such as VGG-16 [Simonyan and Zisserman 2014], GoogleNet [Szegedy et al. 2014], and ResNet [He et al. 2015]. Although these network topologies commonly have convolution operations and other types of layers, the application parameters of each network vary drastically. This is especially true for the convolutional layers because of the complex multidimensional memory access patterns. Having a different number of kernels or different feature map dimensions can shift the bottleneck from the memory perspective. Therefore, the memory access pattern behavior observed for the network is not uniform across all layers, leading to disparity between per-network and per-layer approaches. Even with aggressive compression techniques [Han et al. 2015], the memory footprints of model weights and feature activations are larger than a reasonable amount of on-chip memory.

To motivate our work, we analyzed three DCNN workloads: VGG-16, TINY-YOLO(v2), and MOBILENET [Howard et al. 2017], respectively used for image classification and object detection problems. They are suitable representatives of DCNN workloads in terms of computation and memory requirements. Figure 1(a) shows the VGG-16 network layer topology in terms of layers, parameters per layer, and intermediate feature map sizes. Among DCNNs, VGG-16 is particularly interesting for mapping to hardware accelerators because it not only consists of many 3×3 convolution kernels with a stride of 1 back-to-back, but also each time the number of its kernels increases, the feature map shrinks by half. This common pattern among DCNNs can also be observed on TINY-YOLO(v2) (Figure 1(b)), which is a simplified version of the popular object detection network Yolo version 2 [Redmon and Farhadi 2016] for a smaller memory footprint. In Figure 1, the batch normalization and nonlinear activation layers are not shown, as they are fused with convolutional layers and do not contribute significantly to the computational and memory complexity. The layer representation of MOBILENET is omitted in Figure 1, as it is composed of 27 layers [Howard et al. 2017].

These three DCNNs, selected as motivational examples, are also used in Section 6 as target workloads. Figure 2 shows the layer index of the networks on the x -axis and memory usage of the

Fig. 2. Memory requirements (KiB) by varying the convolutional layers for VGG-16, TINY-YOLO(v2), and MOBILENET.

layers in terms of kilobytes on the y-axis for input feature map and kernel filters. The features and filter weights are represented with 8-bit values. The first layers (1–7 in VGG-16 and 1–5 in TINY-YOLO(v2)) require less filter memory but use feature map memory more intensively; the opposite is true for the other layers. The one exception is the ninth layer of TINY-YOLO(v2): this is due to the 1×1 kernel and only 125 filters as the last output layer. We do not show the memory used by output feature maps because the information is already in the successive layer's input feature maps. The layers of MOBILENET are particularly diverse in terms of filter memory usage since half of the layers are depthwise convolutions, where the kernels are only bidimensional without a channel dimension (Figure 2(f)). From the results reported in Figure 2, we outline how different network topologies have different memory requirements and access patterns. Therefore, a single mapping solution might easily lead to suboptimal results for accessing the memory. This diversity among the layers creates challenging problems for mapping the network layers to hardware accelerators given that a single mapping solution for a convolutional layer does not fit for all configurations of the convolutional layers. The memory footprint and access patterns are significantly different in each layer of the topology (see Figure 2). Therefore, it is necessary to define more than one mapping policy to decide which portion of data would reside in the on-chip memory.

Orlando architecture (detailed in Section 2) fits very well for these target workloads, where the different convolutional layers can take advantage of specific memory access patterns. However, the variety of DCNN applications makes the design space exploration (DSE) challenging for exploiting the performance of the Orlando convolutional accelerators (CAs) and memory architecture. The way that on-chip resources are allocated per-layer significantly affects the performance and energy efficiency of the Orlando SoC. Given the huge size of the design space, exploration and cost-performance analysis are needed to select the best set of parameters for the target DCNN application. In this work, we propose a DSE based on a parametric model of the CAs. We consider application scenarios where different amounts of resources are allocated to a certain network. Furthermore, we consider a single input for a network (input batch size equal to 1), because the latency of the network during inference is more critical than the throughput. Hence, the multibatch processing is outside the scope of this work.

One of the main motivations for our work is the huge design space of different runtime resource allocation and convolutional layer mappings, given the platform constraints and workload description. To address this problem, we propose to apply an automatic DSE approach to cover

more points much faster than applying manual exploration. Our work aims to enable a Pareto analysis with more points by reducing the configuration selection and evaluation time. In contrast, the manual exploration approach relies on the knowledge and intuition of some field experts to select the best configuration points. This is a time-consuming process; in addition, it is more inclined to provide suboptimal and less-accurate solutions. Since manual exploration is not a viable solution, automatic DSE has been used successfully in the past for manycore SoC (like in the *MULTICUBE* framework [Silvano et al. 2011]).

To summarize, our work makes the following key contributions:

- We propose an automatic DSE approach for mapping a given DCNN topology to the Orlando SoC to analyze the impact of different runtime mapping and resource allocation parameters to reveal insights on power/performance trade-offs of the CA and memory subsystem of the Orlando architecture.
- We propose to exploit the results of the DSE at runtime for mapping convolutional layers to the Orlando processing and memory resources. The goal is to optimize the runtime usage of on-chip resources such as CAs and scratch-pad memories, as well as the memory bandwidth toward the off-chip memory.
- We extend the analytical accelerator utilization model proposed in Erdem et al. [2018] by considering the topology of each layer rather than using an approximated per-network model.

2 ORLANDO ARCHITECTURE

STMicroelectronics introduced in Desoli et al. [2017] the Orlando ultra-low-power CNN SoC manufactured with 28-nm FD-SOI technology for smart embedded mobile systems and IoT devices. The Orlando SoC demonstrator integrates an ARM Cortex microcontroller, various peripherals, eight digital signal processor (DSP) clusters, a reconfigurable dataflow accelerator fabric connecting streaming DMAs, and eight CAs. The chip includes four 1-MB SRAM banks, a dedicated bus port, and fine-grained power gating to sustain the maximum throughput for convolutional stages (Figure 3). The chip reaches 1.175 GHz at 1.1 V with a theoretical peak CA performance of 676 GOPS. It is operational at 200 MHz with 0.575 V with 41-mW average power consumption on AlexNet [Krizhevsky et al. 2012] with eight chained CAs with a peak efficiency of 2.9 TOPS/W [Desoli et al. 2017]. Based on Desoli et al. [2017], the Orlando architecture demonstrated significant energy efficiency compared to recent neural network accelerators [Chen et al. 2014; Chen et al. 2015; Chen et al. 2017; Sim et al. 2016] implementing AlexNet.

Multiple CAs can be grouped or chained together to handle various sizes of feature maps and multiple kernels in parallel by using the interconnection given by the programmable stream switch adapting to different NN topologies and feature and kernel tensor geometries. Subtensors can be processed with local buffer resources available in each accelerator [Desoli et al. 2017]. The configurable batch size (indicated as slice size) and variable number of parallel kernels enable optimal trade-offs for the available input and output bandwidth sharing across different units and computing logic resources. Feature and kernel data batches can be processed either sequentially with multiple CAs in a virtual processing chain or iteratively with intermediate results being stored in on-chip memory and fetched in the subsequent batch processing round. Kernel sets (shown in Figure 4(a) as *KER* from 0 to *Q*) are partitioned in batches processed sequentially, and intermediate results can be stored in the on-chip memory.

Various kernel sizes (up to 12×12), subtensor batch sizes (up to 16), and parallel kernels (up to 4) can be handled by a single CA instance, but any kernel size can be accommodated with the

Fig. 3. Orlando SoC top-level block diagram [Desoli et al. 2017].

Fig. 4. Convolution layer mapping and configuration of multiple accelerators.

accumulator input. The CA includes a line buffer to fetch up to 12 feature map data words in parallel with a single memory access.

Efficient mapping of a logical CNN task graph to the architectural computing and memory resources requires that the execution of convolutional layers be partitioned by slicing the kernel and input activation tensors. Each subtensor is mapped to a different CA, and the partial results can either be sent to the memory or directly streamed into another accelerator's input for processing a different slice of the same kernel subtensor for direct accumulation (as shown in Figure 4(a)). The shape of the subtensors is constrained by a relatively large number of parameters, such as the available local storage for each CA (line buffers and kernel buffers), the total on-chip memory storage that can be used for input and output activation maps, and the size of the kernels for a given layer compared to the maximum kernel size supported by the accelerators [Desoli et al. 2018].

Common deep learning frameworks (TensorFlow, PyTorch, etc.) are high-level frameworks that use optimized primitives when deployed on processor architectures and enable the user to connect different layers with scripting languages such as Python. However, to execute DCNN workloads on specialized hardware or SoC platforms, a full system software stack is needed to transform a high-level to a low-level representation and then generate the binary executable to be deployed

Fig. 5. The overall DCNN system stack for the Orlando SoC.

on the platform. The high-level representation usually includes graph structures where branching and merging of tensor operations are present, and the tools working on graph-level optimizations determine the order of operations by scheduling the branches. Figure 5 provides an overall picture of such a DCNN system stack, and it highlights where our work stands. In particular, the runtime resource allocation and runtime convolutional layer mapping are the layers that our DSE targets.

3 PROPOSED EXPLORATION METHODOLOGY

The exploration methodology is composed of two phases. The first phase of the exploration is applied per network, whereas the second phase is applied for each layer of the network, as shown in Figure 6. The per-network runtime DSE is dedicated to resource allocation and dynamic voltage frequency scaling (DVFS) configuration selection for the whole network. The per-layer phase includes a series of explorations of the layer mapping to the CAs for each convolutional layer of the network topology. The reason we have a two-phase exploration is that the per-network exploration selects parameters that will affect and constraint the second per-layer phase. The on-chip memory size (MEM_{SIZE}) and number of accelerators (N_{CA}) are especially critical in terms of restricting the parameter space to the meaningful subset. Although we are not employing different exploration

Fig. 6. Overview of the runtime DSE and mapping flow.

methodologies to the phases, it is possible to have different per-phase exploration methodologies depending on the complexity of the models.

An overview of the proposed exploration and mapping flow is shown in Figure 6. The framework takes the CNN topology parameters and the available resources on the Orlando SoC. The topology parameters mainly include the convolutional layer parameters such as kernel size, number of kernels, number of feature map channels, feature map dimensions, and kernel strides. Orlando resource constraints are on-chip memory, number of CAs, and external memory bandwidth available for the whole platform. The runtime resource allocation selects a suitable allocation conforming to the Orlando resource constraints. Then the model configuration is used by the DVFS configuration selection, where a frequency-voltage operating point is appointed to the network. All of the parameters selected until this point are attributed to the whole network as part of the annotated Orlando model. This model is then used in runtime layer mapping to explore the mapping parameter space. This mapping process is applied for each layer separately because the workload characteristics of each layer can be different. The output is a set of mapped convolutional layers for the selected annotated Orlando model, then the overall flow for the network is repeated for all of the different resource allocation and DVFS selection. At the end, Pareto analysis is used to achieve the Pareto set of the mapped CNNs to Orlando in terms of power consumption versus execution time trade-off.

4 DESIGN SPACE PARAMETERS

Given the target CNN, the design space of each layer is composed of the possible mappings for the different parameters of the Orlando architecture (Table 1). Each configuration point in the design space constrains resource allocation configuration. The exploration of different mappings is feasible because the Orlando architecture offers runtime parameters related to how CAs will compute a convolutional layer.

Table 1 includes the selected design space for resource allocation and energy efficiency parameters. These parameters act as a constraining factor for some of the convolutional layer mapping parameters. The following sections describe the categories of parameters and the role of each parameter.

4.1 Resource Allocation Parameters

These parameters are the number of CAs, on-chip memory (MEM_{SIZE}), and external memory bandwidth (EXT_{BW}) available to a DCNN workload. They are used to simulate on-chip resource

Table 1. Design Space Parameters

Resource Allocation		Energy Efficiency		Conv. Layer Mapping	
MEM_{SIZE}	128, 256, 512, 1024, 2048, 4096	$DVFS_{CONFIG}$	Vdd/MHz	N_{CA_chain}	$N_{CA}/N_{CA_parallel}$
N_{CA}	1, 2, 4, 6, 8	1	0.575/200	$N_{CA_parallel}$	N_{CA}/N_{CA_chain}
EXT_{BW}	200, 400, 800, 1,000, 2,000, 3,000	2	0.6/266	N_{PAR_KERNEL}	1, 2, 4
		3	0.7/450	$BATCH_{SIZE}$	1-16
		4	0.825/650	$STRIPE_{SEQ}$	Depth or Vertical
		5	1.0/950	$N_{STRIPES}$	See Equation (2) and Section 5.3
		6	1.1/1,175		

sharing in the context of multiple simultaneous workloads running. Therefore, we limit the availability of each resource to achieve Orlando model configuration. The range of values for MEM_{SIZE} is 128 KB to 4 MB, which is the total amount of on-chip memory. The values for N_{CA} range from 1 to 8 with even values, because with odd numbers of accelerators, it is hard to take advantage of chaining and parallel accelerators. The EXT_{BW} parameter has a range from 200 MB/s to 3 GB/s.

4.2 Energy Efficiency Parameters

In principle, DVFS is a runtime feature of the platform. In our exploration, it is considered fixed for all layers of a given CNN. This is due to the latency required to adjust to the new DVFS configuration that is usually significantly greater than a single inference latency of a CNN topology. This parameter allows us to run the inference of the networks at different energy-efficiency points and allows the system to adapt to the needs of the application. The available values of DVFS configuration are listed in Table 1.

4.3 Convolutional Layer Mapping Parameters

The convolutional layer mapping parameters are also shown in Table 1. These are mainly related to the configurable parameters of the stream switch and CAs, shown in the coprocessor subsystem in Figure 3. The stream switch enables different numbers and combinations of CAs to run either in parallel or in chain with the help of the DMA engines available on the platform. These different modes affect the execution performance and data reuse patterns, hence directly influencing the external bandwidth usage of the system. The adjustable runtime parameters of the CAs basically allow the CAs to adapt the characteristics of the specific workloads of a convolutional layer. The N_{CA_chain} and $N_{CA_parallel}$ parameters (also indicated as M and N in Figure 4(b) (Parallel/Chained Execution)) are related to how a set of (total of N_{CA}) accelerators are configured in the stream switch, where N_{CA_chain} is the number of chained accelerators and $N_{CA_parallel}$ is the number of parallel accelerator chains. Since the total number of CAs available to the given workload is limited, this constraint on N_{CA_chain} and $N_{CA_parallel}$ is expressed by Equation (1), where L is the set of convolutional layers for a given network.

$$N_{CA_chain_i} \times N_{CA_parallel_i} \leq N_{CA} \forall i \in L \quad (1)$$

The parameters N_{PAR_KERNEL} and $BATCH_{SIZE}$ are the number of parallel kernels and the number of feature channels (a.k.a. feature batches) that have to be processed by a single CA, respectively. The physical limitation on these parameters comes from the design of the CAs (see Section 2). The last two parameters, $STRIPE_{SEQ}$ and $N_{STRIPES}$, relate to how the data is partitioned in the situation that data needed for a single layer cannot fit into the on-chip memory resources allocated to the network. Therefore, the feature map is partitioned into $N_{STRIPES}$, and only one

Fig. 7. Feature map striping.

stripe and filters related to those stripes are copied to the memory, and all convolution operations are performed in a sliding window manner on that chunk of data that resides in the on-chip memory. The $STRIPE_{SEQ}$ parameter determines the way the stripes are formed. It can be either depth-first or vertical-first as represented in Figure 7. In the depth-first case, the feature data is partitioned from the horizontal dimension so that CAs can process the input feature data in the depth-first order (Figure 7(b)). In the vertical-first case, they are partitioned from the channel dimensions, and CAs process the data in the vertical stripe first before moving on to the next stripe (Figure 7(c)). The two different ways of striping exist because different layers might have different memory access patterns as shown in Figure 2.

These parameters implicitly allow different kinds of data reuse available naturally in the convolutional layer. The depth-first approach allows more data reuse of output feature maps by accumulating in smaller feature map regions. In contrast, the vertical-first approach takes advantage of reuse of kernel weights, as each sliding window instance shares kernel parameters. There is also the possibility of including the horizontal-first approach. It would have had similar trade-offs to the vertical-first approach; however, the CAs in the Orlando architecture can take advantage of larger width feature maps. For that reason, we did not consider the horizontal-first approach in our methodology.

To simulate the situations in which the data fits into the allocated on-chip memory, we prune the configuration points with total memory requirement (Equation (2)) that exceeds the MEM_{SIZE} . The details of how the memory footprint is calculated are explained in Section 5.3.

$$MEM_{SIZE} \geq MEM_i = MEM_{input_i} + MEM_{output_i} + MEM_{kernel_i} \forall i \in L \quad (2)$$

Considering the design of experiments selected in Table 1, the cardinality of the design space is composed of 1,080 configuration points. Since the second exploration phase depends partially on the configuration point selected in the first phase, it is much harder to analytically formulate or indicate the cardinality of a combined design space.

5 ACCELERATOR MODEL

The scope of the model is to have a fast DSE and mapping for large design spaces. Therefore, we introduce an analytical model for the accelerators that calculates the execution time per layer and the power consumption per network.

Besides the parameters listed in Table 1, the accelerator model (detailed in Section 5) also exposes the number of bits used to represent the kernel filter weights, input feature map, and output feature map. These three bit-width parameters affect the memory footprint and data transfers. The reason they are not included as precision parameters is that they are kept as constant precision for

Table 2. Data Bit-Width

Type	Symbol	Bit-Width Set
Kernel filter weights	γ_k	8
Input feature map	γ_i	8
Output feature map	γ_o	16

Orlando CAs' working principle that consumes 8-bit input features and kernels to produce 16-bit output features. The preconfigured bit-widths are reported in Table 2 with the symbols used in the equations.

The performance model (Section 5.1), the utilization model (Section 5.2), and the memory usage model (Section 5.3) are considered per layer within the network; therefore, all equations described in those sections are actually indexed per layer. However, for the sake of simplicity, we decided to omit the per-layer indices except for the resource allocation and energy efficiency parameters listed in Table 1. Although the resources allocated and DVFS are selected for the network, mapping parameters change significantly per layer even under the same resource allocation scheme.

5.1 Performance Model

To calculate the execution time for each layer in the network, we compute the number of clock cycles necessary for the accelerators to finish computation of the layer and the time required for the data transfers between the on-chip memory and the external memory. Let us assume that the data needed for the layers like filter weights or the previous feature map reside in the external memory.

Given that the presence of DMA engines (Figure 3) in the Orlando architecture enable asynchronous memory transfers, we assume that memory transfers and CA executions overlap (Equation (3)).

$$T_{total} = \max(T_{compute}, T_{mem_access}) \quad (3)$$

$$T_{compute} = \frac{CA_{work}}{CA_{utilization}} \times T_{clk} \times R_{CA} \quad (4)$$

$$CA_{work} = IFM_W \times IFM_H \times BATCH_{SIZE} \quad (5)$$

Total accelerator execution time is calculated by using Equation (4). The T_{clk} is the clock period given by the DVFS configuration of the Orlando platform, and CA_{work} is the number of cycles of work for a CA to finish a batch of the feature map. The cycle count depends on IFM_W and IFM_H , which respectively represent the width and height of the 3D input feature map (IFM) from a specific layer in the CNN topology. How many cycles are spent actually toward the layer computation is represented by the $CA_{utilization}$ coefficient. The $CA_{utilization}$ coefficient in the equation is calculated per layer according to Section 5.2, and it takes values ranging from 0% to 100% (in the ideal case). The term R_{CA} (Equation (6)) represents how many times (a.k.a. rounds) the CAs need to run to compute a single set of feature activations. The term of R_{CA} is affected not only by the convolutional layer parameters IFM_D and $N_{KERNELS}$ in a layer but also by the runtime parameters as described in Equations (7) and (8).

$$R_{CA} = R_{batch} \times R_{feature} \quad (6)$$

Equation (7) describes how many times an accelerator chain needs to run to consume all input feature channels (IFM_D). The whole chain of CAs is configured with the same $BATCH_{SIZE}$ parameter; therefore, in one full execution of spatial dimensions (IFM_W and IFM_H) of the feature

map, a chain of CAs can process $BATCH_SIZE \times N_{CA_chain}$ of channels. Similarly, the number of rounds needed to run for using all of the kernels is shown in Equation (8). The total number of kernels finished can also be calculated using $N_{PAR_KERNEL} \times N_{CA_parallel}$ since parallel CAs are loaded with different sets of kernel filters and N_{PAR_KERNEL} defines the number of kernels loaded in each CA.

$$R_{batch} = \left\lceil \frac{IFM_D}{BATCH_SIZE \times N_{CA_chain}} \right\rceil \quad (7)$$

$$R_{feature} = \left\lceil \frac{N_{KERNELS}}{N_{PAR_KERNEL} \times N_{CA_parallel}} \right\rceil \quad (8)$$

As shown in Equation (9), T_{mem_access} depends on the bandwidth requirements of features, kernels, and output values for the layers and the external bandwidth of the system. The way memory access and the footprint are calculated is detailed in Section 5.3.

$$T_{mem_access} = \frac{BW_{feature} + BW_{kernels} + BW_{outputs}}{EXT_{BW}} \quad (9)$$

The term T_{clk} used in the models comes from the clock frequency given by the selected DVFS configuration.

5.2 Utilization Model

The utilization model is introduced to estimate the wasted cycles per layer for CAs. The main reason there are cycles not utilized is that CAs need at least $(K_H - 1)$ lines of feature maps and K_W amount of pixels to start producing an output. The delay in units of cycles is expressed by Equations (11) and (12).

$$CA_{utilization} = \frac{CA_{work}}{CA_{work} + (D_{fstartup} + D_{lstartup})} \quad (10)$$

$$D_{lstartup} = IFM_H \times K_W \times BATCH_SIZE \quad (11)$$

$$D_{fstartup} = IFM_W \times (K_H - 1) \times BATCH_SIZE \quad (12)$$

The main factors that play roles in underutilization are the feature map and filter kernel dimensions. Therefore, the characteristics of a layer and its workload are essential to attain accurate utilization that leads to a more precise model. The delay values $D_{lstartup}$ and $D_{fstartup}$ are wasted cycles due to the line start-up and frame start-up. By using these wasted cycles along with cycles that actually produce an output, $CA_{utilization}$ is calculated (Equation (10)).

Figure 8 shows the progression of a CA through one frame of a feature map. The red lines signify data already loaded into the CA buffers, whereas yellow lines indicate data that need to be loaded that stall the CAs. The blue rectangle shows the kernel window. When a dashed line is shown, it means that it is stalled and waiting for data. Although Figure 8 shows 2D data, there is also a batch of feature map channels that is characterized by $BATCH_SIZE$ in Equations (11) and (12). In Figure 8(a), at the beginning of the frame $K_H - 1$ lines must be loaded (yellow lines). This frame start-up cost is also formulated in Equation (12). In Figure 8(b), after the (red) lines are loaded, the CA waits for K_W pixels for the one line (shown by a partial yellow line). After it is loaded, the CA starts to execute in a sliding window fashion, whereas the DMA engines load the necessary data for the next pixels of the feature map, as shown in Figure 8(c). When the CA engine reaches the end of a line (Figure 8(d)), it needs to move on to the next line. However, it requires K_W amount of pixels of the next line to produce an output and to begin a sliding window (Figure 8(e)).

Besides the aforementioned underutilization, there is also the time to configure the whole acceleration framework to get the dataflow. However, we decided not to include this cost in the model

Fig. 8. CA execution progression with data stalls.

because it corresponds to 20 to 40 register file accesses per layer, and this is negligible with respect to the time required to run a layer.

5.3 Memory Usage Model

The memory usage model estimates the memory footprint of the input feature map, weight filters, and output feature map. In the case of input feature maps, due to the striping, the size of stripes are considered. In this way, it is possible to utilize the on-chip memory by taking advantage of spatial locality. The striping is explained in Section 4.3.

To calculate the memory occupied by an input feature stripe, Equations (13) and (14) are used. In the case of no striping (# of stripes == 1), Equation (13) is used when the whole feature map is located in the on-chip memory. Otherwise, Equation (14) is used to account for the overlapping regions of the kernel windows. However, to keep the equation simpler, the kernel stride factor is assumed to be equal to 1. This leads to $(K_H - 1 = 2)$ for both VGG-16 and Tiny-Yolo(v2).

$$MEM_{stripe} = IFM_W * IFM_H * BATCH_{SIZE} * N_{CA_chain} * \gamma_i \quad (13)$$

or

$$MEM_{stripe_i} = IFM_{W_i} * \left(\frac{IFM_{H_i}}{N_{STRIPES_i}} + K_{H_i} - 1 \right) * BATCH_{SIZE_i} * N_{CA_chain} * \gamma_i \quad (14)$$

The model introduced in this work considers memory transfers and the CA executions overlapped by taking advantage of DMA engines and double buffering. Due to the double buffering conditions, when there is a requirement to have more accelerator batch runs (indicated as R_{batch}), the execution model requires two stripes. Thus, one buffer can be used for data transfers, whereas

accelerators work on the other feature stripe (Equation (15)).

$$MEM_{input} = \begin{cases} 2 \times MEM_{stripe}, & \text{if } R_{batch} > 1 \\ MEM_{stripe}, & \text{otherwise} \end{cases} \quad (15)$$

Similarly, the memory footprint of kernel filters is also calculated as shown in Equation (16). MEM_{kernel} is mostly affected by the convolutional mapping parameters, as they describe how the CAs will consume and process the data. Again, due to the usage of double buffering, two different kernel buffers are assumed to exist simultaneously.

$$MEM_{kernel} = 2 \times N_{CA_parallel} \times N_{PAR_KERNEL} \times BATCH_SIZE \times N_{CA_chain} \times \gamma_k \quad (16)$$

Last, the output of CAs, also known as output feature maps, might consume on-chip memory in the form of partial accumulation buffers. This concept is considered in Equation (18). Hence, if there are multiple batch rounds, it means that the system does not need accumulation buffers and streams out the data directly to the external memory for the next layer. Additionally, when the striping is depth-first, CAs will process input channels first and the accumulator buffers hold just a subset of the output feature map size ($OFM_W \times OFM_H$). This relation is shown in Equation (17).

$$Out_{size} = \begin{cases} OFM_W \times OFM_H \times N_{CA_parallel} \times N_{PAR_KERNEL}, & \text{if } STRIPE_{SEQ} = \text{Vertical} \\ \frac{OFM_W \times OFM_H}{N_{STRIPES}} \times N_{CA_parallel} \times N_{PAR_KERNEL}, & \text{if } STRIPE_{SEQ} = \text{Depth} \end{cases} \quad (17)$$

$$MEM_{output} = \begin{cases} 0, & \text{if } R_{batch} = 1 \\ Out_{size} \times \gamma_o, & \text{otherwise} \end{cases} \quad (18)$$

Equations (19), (20), and (21) describe the relationship between the memory footprint and the bandwidth for input feature maps, kernel parameters, and output feature maps, respectively. The input feature and kernel parameter bandwidth usage are related to how many times the configured CAs need to run to complete one layer and the output feature bandwidth equals to the total size of the output feature map in bytes since the outputs will be transferred only once to the external memory.

$$BW_{feature} = MEM_{stripe} \times R_{CA} \times N_{STRIPES} \quad (19)$$

$$BW_{kernels} = MEM_{kernel} \times R_{CA} \quad (20)$$

$$BW_{output} = OFC_W \times OFC_H \times OFC_D \times \gamma_o \quad (21)$$

These bandwidth usages are critical in terms of determining the total time required for accessing the external memory. In an ideal mapping, it should be equal to or less than the computation cycles of the CAs; therefore, the transfer times are overlapped with computation via DMA (Equations (9) and (3)).

5.4 Power Model

We obtained a quick but accurate enough estimate of the power consumption of the CA system based on our internal tests and measurements of hardware efficiency, and we considered a network-level model based on the aggregated data obtained from the performance model. To estimate the average power consumption, we introduce Equation (23), where the $GOPS_i$ term is based on the layer configuration parameters of layer i (using Equation (22)), and it is normalized to its execution time. Then it is divided by the $GOPS/W$ metric, which is the hardware efficiency value that comes directly from the $DVFS_{CONFIG}$ selected.

$$GOPS_i = 2 * \frac{OFM_{W_i} \times OFM_{H_i} \times OFM_{D_i} \times IFM_{D_i} \times K_{H_i} \times K_{W_i}}{10^9} \quad (22)$$

In Equation (22), the terms OFM_{W_i} , OFM_{H_i} , and OFM_{D_i} represent the width, height, and number of the output feature map for each layer i , respectively.

$$P = \frac{\sum_i GOPs_i / T_{total_i}}{GOPs/W} \quad (23)$$

After the power consumption for the network, the efficiency is calculated using Equation (24) as a unit of fps/W .

$$Efficiency = \frac{1 / \sum_i T_{total_i}}{P} \quad (24)$$

5.5 Depthwise and Pointwise Convolution Mapping

The Orlando CAs, as described in Section 2, are natively capable of handling 3×3 convolution kernel masks. However, some recent DCNN topologies (e.g., Howard et al. [2017]) include depthwise convolution and pointwise convolutional layers. Although these two types of layers are not natively supported by Orlando, we have optimized mapping of depthwise and pointwise convolutions on Orlando.

Depthwise convolutions differ from regular convolutional layers in that they have kernel masks with only one channel. In this way, each kernel corresponds to one channel of the input feature map. To map these layers, we consider the $BATCH_{SIZE}$ parameter equal to the N_{PAR_KERNEL} parameter, and we do not use CA chaining due to the lack of accumulation along the feature channel dimension. Furthermore, because kernel masks contain separate channel dimensions, we have $R_{batch} == 1$. In this way, it is possible to process multiple kernels with corresponding input feature channels utilizing line buffers and MAC clusters in CAs.

Pointwise convolutions are convolutions with kernel masks with a single dimension, and hence the feature map reuse due to the sliding window is not present. This creates problems for CA utilization, as the MAC clusters are designed to take advantage of the sliding window. Nevertheless, to provide acceleration support for such convolutional layers, each one of the MAC operations in the channel dimensions is mapped to different MAC clusters in a CA and utilizes a cluster for a dot product. Since each cluster is composed of 3 MAC units and only one of them is used for computation purposes, the maximum achievable utilization is only one third of a 3×3 convolutional kernel. Because CAs contain 12 MAC clusters, the $BATCH_{SIZE}$ is limited to 12 to enable the mapping described previously.

6 EXPERIMENTAL SETUP

The proposed runtime DSE and mapping methodology is targeted to the Orlando SoC. As workloads, we used MOBILENET [Howard et al. 2017], VGG-16 [Simonyan and Zisserman 2014], TINY-YOLO(v2) [Redmon and Farhadi 2016], and PICO-YOLO, a reduced version of TINY-YOLO(v2) customized to fit the memory subsystem of the Orlando SoC (see Section 7.1). The layer parameter details for TINY-YOLO(v2) and PICO-YOLO workloads are presented in Table 3 along with the GOPs required for each layer. The parameters values are reported in Howard et al. [2017] for MOBILENET and in Simonyan and Zisserman [2014] for VGG-16. The total complexity of the convolutional layers are 30.69, 6.93, 1.14, 1.08 GOPs per inference for VGG-16, TINY-YOLO(v2), MOBILENET, and PICO-YOLO, respectively.

7 EXPERIMENTAL RESULTS

This section begins characterizing our model by comparing to a board implementation of an in-house developed DCNN called *Pico-Yolo*. Then, to evaluate the benefits in terms of flexibility and efficiency of the proposed runtime exploration and mapping for Orlando CAs, we explored the

Table 3. Layer Parameters of Tiny-Yolo(v2) and Pico-Yolo Network Topologies

Layer Name	Tiny-Yolov(v2)				Pico-Yolo			
	Channels	Features	Kernels	GOPs	Channels	Features	Kernels	GOPs
Conv1	3	416	16	0.15	3	240	16	0.05
Conv2	16	208	32	0.40	16	120	32	0.13
Conv3	32	104	64	0.40	32	60	64	0.13
Conv4	64	52	128	0.40	64	30	128	0.13
Conv5	128	26	256	0.40	128	15	256	0.13
Conv6	256	13	512	0.40	256	15	256	0.27
Conv7	512	13	1,024	1.59	256	14	128	0.12
Conv8	1,024	13	1,024	3.19	128	14	256	0.12
Conv9 ¹	1,024	13	125	0.04	256	14	60	0.01

(a) 200MHz

(b) 800MHz

Fig. 9. Model characterization results.

performance and power consumption trade-offs for VGG-16, TINY-YOLOV(v2), and MOBILENET workloads for different resource allocations. Finally, for each choice of resource allocation, we explored all mapping possibilities on the CA units. The experimental results have been organized in terms of scalability, adaptivity, CA's utilization, and Pareto analysis. It is outside the scope of this work to compare power and performance of the Orlando SoC with respect to other state-of-the-art hardware accelerators.

7.1 Model Characterization

In this section, we compare our analytical model with a board implementation of an in-house DCNN, namely PICO-YOLO. Since the CA runs are orchestrated by one of the DSPs embedded in the Orlando SoC, their overhead is included in the performance measurements. To remove this overhead from the measurements, the relationship between the rounds of CA runs (R_{CA}) and the error of our model with respect to the board implementation is estimated with linear regression for both 200 MHz and 800 MHz (Figure 9) corresponding to the values 1 and 3 of $DVFS_{CONFIG}$ reported in Table 1. The accuracy of the linear regression results is summarized in Table 4. The p -value in Table 4 shows that the slopes in Figure 9(a) and (b) reject the null hypothesis; therefore, we can statistically assume that the slope shows the processor overhead per round.

¹9th convolutional layer is a pointwise convolution hence the kernel window is 1×1 mask. This is one of the reasons for low GOPs for the layer.

Table 4. Linear Regression on Accuracy Error Results

Frequency	R-Value	p-Value	Overhead per Round
200 MHz	0.992	$1.06e^{-5}$	13.86 μ sec
800 MHz	0.991	$1.12e^{-5}$	3.47 μ sec

To evaluate the accuracy of our model, expressed as the absolute percentage error for each layer, we used Equation (25), where the terms E_i , P_i , and B_i express the error, predicted execution time, and board execution time, respectively.

$$E_i = \frac{|P_i - B_i|}{B_i}, \forall i \in L \quad (25)$$

Moreover, we use the weighted mean on the layer accuracy error, where the weights are the number of operations per layer. The predicted execution time is given by our model adjusted by the processor overhead estimated by linear regression according to the overhead per round expressed in Table 4. Overall, our model introduces an error of 15.63% and 15.56% for the 200 MHz and 800 MHz cases, respectively. This error is due to the internal memory bank and bus congestion because the board implementation does not include data striping to multiple memory banks.

7.2 Scalability

In the first set of experiments, to evaluate the scalability of the DCNN workloads mapped to the Orlando architecture, we estimated the performance and power consumption by using the proposed models on a number of different configurations by varying the number of CAs. We assumed a total on-chip memory allocated to the DCNN workload of 512 KB. Scalability results are reported in Figure 10 and Figure 11, in terms of power and performance by varying the DVFS configurations (as reported in Table 1) and the number of CAs (1, 2, 4, and 8). We report the normalized power/performance results for two different values of the external memory bandwidth. Power consumption values (bars) and performance values (lines) are normalized with respect to a single CA assuming the 0.575 Vdd/200 MHz configuration. For each configuration of Vdd/frequency, we analyzed the scalability in terms of power and performance by varying the CAs.

In the case of 800 MB/s external bandwidth, the performance scales up to the DVFS configuration 3 (corresponding to 0.7 Vdd/450 MHz) across different workloads. However, when the bandwidth increases to 1.6 GB/s, the results start to show some differences between the workloads and number of CAs allocated. With 1.6 GB/s external bandwidth, MOBILENET workload scales up to the configuration 5 (1.0 Vdd/950 MHz), whereas VGG-16 scales differently in the ranges between 1 to 2 CAs and 4 to 8 CAs. Considering 1 to 2 CAs, the performance is still saturated at configuration 5; however, when using 4 to 8 CAs, performance is limited by the external bandwidth at DVFS configuration 4 (0.875 Vdd/650 MHz). Similar behavior is observed by using PICO-YOLO and TINY-YOLO(v2): the 4 to 8 CAs allocation saturates at DVFS configuration 4 (0.875 Vdd/650 MHz) and 3 (0.7 Vdd/450 MHz), respectively.

7.3 Mapping Adaptivity

In this section, we evaluated the benefits due to the adaptivity of the proposed methodology by comparing our *adaptive mapping* to the *one-fits-all* mapping applied to the different convolutional layers. Evaluation results are reported in Figure 12 for VGG-16, TINY-YOLO(v2), and MOBILENET. For each network, the one-fits-all mapping has been calculated by selecting the best mapping based on memory usage, as reported in Figure 2, then using this mapping done at design time for all of

Fig. 10. PICO-YOLO (left) and TINY-YOLO(v2) (right) normalized power and performance scaling by varying DVFS configurations (refer to Table 1) and number of CAs for two values of external memory bandwidth.

the other convolutional layers. The adaptive mapping is obtained by choosing the best mappings resulting from the DSE for each layer as proposed in Section 3.

For the three selected networks, Figure 12 reports the throughput expressed as GOPs per second on the y-axis by varying the convolutional layers on the x-axis. We assume eight CAs, 512 KB of on-chip memory, 800 MB/s external bandwidth as resource allocation, and 0.6 Vdd/266 MHz DVFS configuration. In Figure 12(a), the sixth convolutional layer is selected for one-fits-all mapping, because it represents the middle ground between the feature memory and filter memory usage presented in Figure 2(a) and (d), respectively. Similarly, in Figure 12(b), the fourth layer is selected for the one-fits-all mapping according to the results shown in Figure 2(b) and (c). In Figure 12(c), the fifth convolutional layer is selected as one-fits-all being one of the most computational heavy layers in terms of the number of operations.

For the three selected networks, the throughput of the adaptive mapping outperforms or matches the one-fits-all mapping for all of the convolutional layers. (The cases where the throughput is equal correspond to when the adaptive mapping used is identical to the one-fits-all). We can notice that the throughput of the first layer is very small for both adaptive and one-fits-all mapping, because the channel size for the first layer is very small (since the input is a three-channel RGB image), and it reduces the amount of data reuse generating utilization problems with eight CAs. For MOBILENET (Figure 12(c)), there is an alternating trend for the layers, because the utilization of the hardware is significantly different between depthwise convolutions and pointwise convolutions. The second reason is that the depthwise layers with a stride factor of 2 (layers 4, 8, 12, and 24) reduce the data reuse within the sliding windows.

Fig. 11. MOBILENET (left) and VGG-16 (right) normalized power and performance scaling by varying DVFS configurations (refer to Table 1) and number of CAs for two values of external memory bandwidth.

7.4 CA Utilization

CA utilization is explored in terms of low-memory (128 KB) and high-memory (512 KB) options for the selected workloads. For each option, we analyzed the behavior with 400 MB/s, 800 MB/s, and 1.6 GB/s external bandwidth. The results include utilization of accelerators per each layer of the networks and aggregated with a weighted average for the whole network. The y-axis of Figures 13 and 15 represents the percentage CA utilization (useful computation cycles) in the range from 25% to 100%, whereas the y-axis of Figure 14 is in the range from 70% to 100%. The difference between VGG-16 and the other networks is that VGG-16 does not contain any pointwise convolutional layer, so the minimum utilization achieved on one of its layers is larger than 70%. For space reasons, we reported the results for 800 MB/s only; however, the behavior obtained for 400 MB/s and 1.6 GB/s is quite similar. The number of allocated CAs is reported as $\#ca$ for each memory option and workload in Figures 13, 14, and 15. Since the results are after exploration of mapping parameters per each layer, the utilization values are the best values under the assumption of allocated resources and DVFS parameters. The results summarize how much of the $T_{compute}$ mentioned in Section 5.1 is utilized for actual meaningful computation. Due to the stalls during line start-up and frame start-up detailed in Section 5.2, some cycles are spent before either the pipelines of CAs are filled or waiting for data to be ready in the CA's buffers. This utilization concept is mostly independent of the data stalls due to limited external bandwidth. However, with increased bandwidth, our exploration methodology tries to include faster compute configurations; thus, it prefers higher-utilization mappings. In the VGG-16 workload, the low memory allocation mostly affected the first layers of the network. The main reason for this behavior is that with low memory, it is not

Fig. 12. Throughput comparison of adaptive mapping vs. one-fits-all mapping by varying the convolutional layers for VGG-16, TINY-YOLO(v2), and MOBILENET.

Fig. 13. TINY-YOLO(v2) topology: percentage of CA utilization cycles for each layer and for the network.

possible to put more kernel filters on-chip than what is necessary to take advantage of more than 1 to 2 CAs (Figure 14). In the TINY-YOLO(v2) workload, the ninth layer is quite different compared to the other layers, and it reaches below 30% utilization because the ninth layer is a pointwise convolutional layer (Figure 13). However, in terms of number of operations, the contribution of this layer is quite insignificant (see Table 3). The MOBILENET workload differs from the previous two DCNN workloads significantly since it is mainly composed of pointwise and depthwise convolutions. Be-

Fig. 14. VGG-16 topology: percentage of CA utilization cycles for each layer and for the network.

Fig. 15. MOBILENET topology: percentage of CA utilization cycles for each layer and for the network.

cause of this, we can see alternating utilization patterns, where pointwise layers (3, 5, 7, ..., 27) are below 30% utilization and depthwise convolutions with stride 2 (4, 8, 12, 24) are significantly lower than other depthwise convolutions (Figure 15). The geometry of the convolutional layers plays an important role, since the channel dimension (depth) of the input feature maps are shallow in the first layers of the network; therefore, it is also hard to utilize chained accelerator configurations.

Nevertheless, the utilization for the first four layers is higher than 85%. The explanation comes from the height and width of the input feature map since first lines of a frame and beginning of each line cause wasted cycles for the CAs, larger feature maps are much more efficiently processed due to the amortized cost of CA stalls. This effect is also visible when we observe layers 11, 12, and 13, which have lower utilization due to smaller 14×14 feature dimensions. A similar situation is given also for the TINY-YOLO(v2) workload (Figure 13). The last three layers (layers 6–8) have the lowest utilization of close to 70% (except the ninth layer). Again, the reason is the dimension of the feature map being 13×13 . However, layer 2 achieves the highest utilization numbers under multiple CAs, which is explained by the larger feature map and having enough input channels to fill the accelerators completely. With higher memory allocations, the utilization drawback in the first layer is mitigated. Even if we are not able to show the results for all combinations of the resource allocation, with the choice of the best convolutional layer mapping, it is possible to achieve more than the 70% utilization for all resource allocations for the layers that the Orlando CA natively supports. This confirms the importance of adaptive mapping techniques to take advantage of the accelerator.

7.5 Pareto Analysis

To obtain the most useful configurations from the design space, we applied DSE combined with Pareto front analysis to our design space. This is motivated by the huge design space that results from the combination of resource allocation, energy efficiency, and convolutional layer mapping parameters, which in our VGG-16 use case mapped to the Orlando architecture reaches 10 million configuration points, and for the TINY-YOLO(v2) case, it reaches 5.3 million configuration points. In the cases of PICO-YOLO and MOBILENET, the number of configurations are 6.7 and 5.6 million, respectively. After applying heuristic search and Pareto filtering techniques, the Pareto front in terms of normalized power consumption with respect to normalized execution time has been derived and represented in Figure 16 for our workloads. By using Pareto filtering, we reduced the number of design choices to 117 and 161 for PICO-YOLO and TINY-YOLO(v2), respectively. Similarly, for MOBILENET and VGG-16, the number of design choices after Pareto filtering are reduced to 160 and 140 design points, respectively. This corresponds to a reduction of five orders of magnitude in the number of design points. Each point in the Pareto front represents a good trade-off from the point of view of power/performance. Depending on the desired execution conditions such as ultra-low-power or high performance, a corresponding point from the Pareto front can be selected after this analysis, without being concerned with suboptimal solutions. Analyzing the Pareto front, we can observe that there are 193 unique combinations of mapping parameters (these unique runtime configurations are represented by different colors in Figure 16). This analysis confirms that to reach a target power/performance trade-off, the DSE must include mapping parameters as well.

8 RELATED WORKS

Mapping of computations to actual processing elements (PEs) can be achieved with flexible mapping techniques such as SmartShuttle [Li et al. 2018], which is an adaptive layer partitioning and mapping technique proposed using an analytical model of data requirements of convolutional layers. SmartShuttle focuses on reducing DRAM to on-chip memory transfers, and FlexFlow [Lu et al. 2017] proposes an accelerator architecture that can take advantage of parallelism in the convolutional layers by dynamically mixing different parallelization strategies, and by optimizing the dataflow from on-chip memory to PEs. In contrast to FlexFlow, Scale-Sim [Samajdar et al. 2018] is a systolic array accelerator simulator used to study the different configurations of architectures with different DCNN workloads to assess the effect of different dataflow types introduced by Eyeriss [Chen et al. 2017]. Similarly, TETRIS [Gao et al. 2017] uses the dataflow patterns described in Eyeriss. However, the authors rely on HMC memory stack instead of large on-chip SRAM as a

Fig. 16. Power/performance Pareto fronts.

system design approach. TETRIS work in Gao et al. [2017] describes how they partitioned feature data and schedule to different vaults (DRAM). TETRIS uses partition schemes similar to that we proposed in Section 4.3. However, what TETRIS mentions as “output-feature-map partitioning” is in our case the combination of two mapping parameters: the number of parallel CAs and parallel kernels per CA, thus giving us more degrees of freedom in the partitioning phase. TETRIS tries to minimize the on-chip SRAM by using a sophisticated 3D stacked memory; however, TANGRAM [Gao et al. 2019] tries to scale the accelerator using distributed SRAMs per 2D tile that are quite similar to Eyeriss. The authors introduce buffer sharing between the tiles to improve intralayer parallelism and alternate loop ordering for consecutive layers to orchestrate dataflow between layers. Timeloop [Parashar et al. 2019] is another systematic approach to map DNNs to accelerators based on the connectivity of PEs and on/off-chip storage elements. Although they describe computation mapping techniques similar to the sliding window, the pipeline start-up inefficiencies are not taken into account compared to our work, where we included them in the proposed utilization model described in Section 5.2. mRNA [Zhao et al. 2019] is a mapping space explorer for the DCNN accelerator named MAERI [Kwon et al. 2018] that tries to model analytically the accelerator-specific heuristics, such as utilization and latency, to prune a huge mapping space before generating configurations.

Since flexibility of accelerators is very important, FPGAs (being naturally adaptive) are gaining attention in this domain along with ASIC accelerators. In Zhang et al. [2015], a tiling methodology takes advantage of data reuse to adapt to the network’s convolutional layers to fully utilize the DSP resources of the FPGA. DCNNs can be quite large in terms of data and computation compared to FPGA resources. To address this problem, Venieris and Bouganis [2016] developed a method-

ology called *fpgaConvNet* to take advantage of reconfigurability in FPGAs by using synchronous dataflow. DnnWeaver [Sharma et al. 2016] proposes a macro-op ISA for deep neural networks and hand-optimized architectural components to execute these macro-ops. The authors use performance estimators to drive a DSE to come up with suitable FPGA configurations for a given DCNN. Similarly, Guan et al. [2017] try to map the DCNN topologies to FPGA resources by using RTL-HLS hybrid templates. The authors propose a mapping methodology that considers the software side of the system where the FPGA is connected to a host system through some means of communication, such as PCI Express. Another promising approach to accelerate DCNNs is to use systolic arrays of PEs. This approach has been explored by Wei et al. [2017], who designed a systolic accelerator and proposed a two-phase DSE based on an analytical model.

Whereas in our work we are focusing on the modeling of convolutional hardware accelerators and DSE, other previous works [XLA Team 2017; Chen et al. 2018; Rotem et al. 2018] proposed mapping and schedule DCNN operations on processor architectures. TensorFlow XLA [XLA Team 2017] is one such framework introducing just-in-time and ahead-of-time compilation of TensorFlow computation graphs. In this way, the framework aims to merge operations and generates compiled kernels specialized to the underlying CPU or GPU. The authors of Glow [Rotem et al. 2018] introduce an intermediate representation by extending the well-known LLVM IR to implement compiler techniques specific to DCNN workloads. Along with the low-level IR, Glow uses a high-level representation that is a node-based dataflow graph. It comes with a set of optimizations for memory layout driven by buffer lifetime analysis. However, Glow is heavily focused on the training phase and the processor architecture. On the contrary, TVM [Chen et al. 2018] proposes an intermediate representation (based on Halide) that allows merging of multiple operations with convolutions, scheduling, and code generations. TVM also explores the scheduling space by using an ML-based cost model since the target of the work is not device specific. However, in our work, we can apply exhaustive search due to the accurate analytical hardware model.

9 CONCLUSION

In this article, we describe an efficient way of exploring on-chip resource allocation for accelerated DCNN workloads on the Orlando flexible and scalable architecture. The article also addresses the problem of the exploration of the large design parameter space associated with the mapping of DCNNs to the Orlando computing and memory resources. The mapping is considered to take advantage of the flexibility of the Orlando platform to adapt to the workload of each convolutional layer in a DCNN. Exploration results have been analyzed in terms of power/performance scalability and trade-offs for the VGG-16, Tiny-Yolo(v2), and MobileNet CNN topologies.

This work focuses on the Orlando architecture; however, in principle, the striping method is generalizable to other architectures, as it is mainly related to workload memory partitioning. Moreover, we believe that the formulation of the utilization model can also be easily extended to similar architectures, as there is only one architecture-dependent coefficient related to the number of cycles of work for a CA to finish a batch of the feature map.

REFERENCES

- T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. 2015. A high-throughput neural network accelerator. *IEEE Micro* 35, 3 (May 2015), 24–32. DOI: <https://doi.org/10.1109/MM.2015.41>
- Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-end optimization stack for deep learning. arxiv:1802.04799.

- Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, et al. 2014. DaDianNao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, Los Alamitos, CA, 609–622. DOI: <https://doi.org/10.1109/MICRO.2014.58>
- Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (Jan. 2017), 127–138. DOI: <https://doi.org/10.1109/JSSC.2016.2616357>
- Giuseppe Desoli, Thomas Boesch, Surinder Pal-Singh, and Nitin Chawla. 2018. A new scalable architecture to accelerate deep convolutional neural networks for low power IoT applications. In *Proceedings of Embedded World 2018*.
- Giuseppe Desoli, Nitin Chawla, Thomas Boesch, Surinder-Pal Singh, Elio Guidetti, Fabio De Ambroggi, Tommaso Majo, et al. 2017. 14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems. In *Proceedings of the 2017 IEEE International Solid-State Circuits Conference (ISSCC'17)*. IEEE, Los Alamitos, CA, 238–239. DOI: <https://doi.org/10.1109/ISSCC.2017.7870349>
- Ahmet Erdem, Cristina Silvano, Thomas Boesch, Andrea C. Ornstein, Surinder-Pal Singh, and Giuseppe Desoli. 2018. Design space exploration for orlando ultra low-power convolutional neural network SoC. In *Proceedings of the 29th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'18)*. IEEE, Los Alamitos, CA, 1–7. DOI: <https://doi.org/10.1109/ASAP.2018.8445096>
- Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and efficient neural network acceleration with 3D memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 751–764. DOI: <https://doi.org/10.1145/3037697.3037702>
- Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized coarse-grained dataflow for scalable NN accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, NY, 807–820. DOI: <https://doi.org/10.1145/3297858.3304014>
- Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Proceedings of the 25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. IEEE, Los Alamitos, CA, 152–159. DOI: <https://doi.org/10.1109/FCCM.2017.25>
- Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. arxiv:1510.00149.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. arxiv:1512.03385.
- Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. arxiv: 1704.04861.
- Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS'12)*. 1106–1114. <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>.
- Hyounjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, 461–475. DOI: <https://doi.org/10.1145/3173162.3173176>
- Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, and Xiaowei Li. 2018. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *Proceedings of the 2018 Design, Automation, and Test in Europe Conference and Exhibition (DATE'18)*. IEEE, Los Alamitos, CA, 343–348. DOI: <https://doi.org/10.23919/DATE.2018.8342033>
- Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, Los Alamitos, CA, 553–564. DOI: <https://doi.org/10.1109/HPCA.2017.29>
- Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Bruce Khailany, Stephen W. Keckler, and Joel S. Emer. 2019. Timeloop: A systematic approach to DNN accelerator evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*. IEEE, Los Alamitos, CA, 304–315. DOI: <https://doi.org/10.1109/ISPASS.2019.00042>
- Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, faster, stronger. arxiv:1612.08242.
- Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. arxiv:1805.00907.

- Ananda Samajdar, Yuhao Zhu, Paul N. Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN accelerator. arxiv:1811.02883.
- Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, Los Alamitos, CA, 1–12.
- C. Silvano, W. Fornaciari, G. Palermo, V. Zaccaria, F. Castro, M. Martinez, S. Bocchio, et al. 2011. MULTICUBE: Multi-objective design space exploration of multi-core architectures. In *VLSI 2010 Annual Symposium*, N. Voros, A. Mukherjee, N. Sklavos, K. Masselos, and M. Huebner (Eds.). Springer Netherlands, Dordrecht, 47–63.
- J. Sim, J. S. Park, M. Kim, D. Bae, Y. Choi, and L. S. Kim. 2016. 14.6 A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoT systems. In *Proceedings of the 2016 IEEE International Solid-State Circuits Conference (ISSCC'16)*. IEEE, Los Alamitos, CA, 264–265. DOI : <https://doi.org/10.1109/ISSCC.2016.7418008>
- Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arxiv:1409.1556.
- Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going deeper with convolutions. arxiv:1409.4842.
- S. I. Venieris and C. Bouganis. 2016. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *Proceedings of the 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. IEEE, Los Alamitos, CA, 40–47. DOI : <https://doi.org/10.1109/FCCM.2016.22>
- Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and J. Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. ACM, New York, NY, 1–6. DOI : <https://doi.org/10.1145/3061639.3062207>
- XLA Team et al. 2019. XLA: Domain-specific compiler for linear algebra that optimizes TensorFlow computations.
- Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. ACM, New York, NY, 161–170. DOI : <https://doi.org/10.1145/2684746.2689060>
- Zhongyuan Zhao, Hyoukjun Kwon, Sachit Kuhar, Weiguang Sheng, Zhigang Mao, and Tushar Krishna. 2019. mRNA: Enabling efficient mapping space exploration for a reconfiguration neural accelerator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*. IEEE, Los Alamitos, CA, 282–292. DOI : <https://doi.org/10.1109/ISPASS.2019.00040>

Received March 2019; revised November 2019; accepted January 2020