Overcoming Data Transfer Bottlenecks in FPGA-based DNN Accelerators via Layer Conscious Memory Management

Xuechao Wei[1,3], Yun Liang[1,*], Jason Cong[2,1,3,†]
[1]Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China
[2]Computer Science Department, University of California, Los Angeles, CA, USA
[3]Falcon Computing Solutions, Inc., Los Angeles, CA, USA
{xuechao.wei,ericlyun}@pku.edu.cn,cong@cs.ucla.edu

ABSTRACT

Deep Neural Networks (DNNs) are becoming more and more complex than before. Previous hardware accelerator designs neglect the layer diversity in terms of computation and communication behavior. On-chip memory resources are underutilized for the memory bounded layers, leading to suboptimal performance. In addition, the increasing complexity of DNN structures makes it difficult to do on-chip memory allocation. To address these issues, we propose a layer conscious memory management framework for FPGA-based DNN hardware accelerators. Our framework exploits the layer diversity and the disjoint lifespan information of memory buffers to efficiently utilize the on-chip memory to improve the performance of the layers bounded by memory and thus the entire performance of DNNs. It consists of four key techniques working coordinately with each other. We first devise a memory allocation algorithm to allocate on-chip buffers for the memory bound layers. In addition, buffer sharing between different layers is applied to improve on-chip memory utilization. Finally, buffer prefetching and splitting are used to further reduce latency. Experiments show that our techniques can achieve 1.36X performance improvement compared with previous designs.

1 INTRODUCTION

DNNs are compute-intensive learning models with growing applicability in a wide range of domains. FPGA is one of the promising hardware platforms for DNNs due to its high performance, energy efficiency and reconfigurability. Compared with other hardware platforms, off-chip memory bandwidth of FPGA is a limitation factor for high performance. Previous designs [1, 2, 19, 22] attempt to keep part or all the parameters or activations on chip to avoid off-chip memory data transfer by layer fusion. However, they are only applicable for simple neural networks with linear structures that do not need large memory volume for activations. There are also FPGA designs using very low bit precisions like 1- or 2-bit [14, 23]. But they are not applicable for all DNN scenarios. Recently, the data sizes, network topology and depth of DNNs are rapidly evolving. For example, the inception module introduced in the GoogLeNet [12], the residual block and dense block introduced in ResNet [6] and DenseNet [4], use non-linear structures with complex data dependency between layers. As a result, it is impossible to store all the parameters and activations in FPGA's on-chip memory for these latest models. Previous DNN accelerator designs [2, 8, 9, 11, 17, 20, 21] adopt a uniform memory management strategy for all the layers, which uses off-chip memory to store the entire data and uses on-chip memory for reuse within a tile.

We observe that different layers of DNNs often employ different shapes of tensors, leading to diverse computation and communication behaviors. The bandwidth requirements of some layers can exceed the device bandwidth limitation. As a result, the performance of these layers are bounded by the off-chip memory bandwidth. For example, more than half of the layers in Inception-v4 [13] are memory bounded. Moreover, within these memory bounded layers, over 60% of them even need 70 GB/s bandwidth, far exceeding DDR4 peak bandwidth which is 20 GB/s. But for the other layers, they are computation bounded and we can keep their data in off-chip memory without harming the performance. Based on this observation, our motivation is to devise a more elaborate on-chip memory allocation strategy in order to overcome the performance bottlenecks for the memory bounded layers. Prior work [5] tries to minimize the number of off-chip buffers on FPGAs. There also exist some prior works [10, 15] that use memory management techniques for optimizing the training process on GPUs. However, none of them explore on-chip memory management for DNN accelerators. For FPGA-based DNN accelerators, we are facing two challenges. On one hand, since different layers vary in terms of computation and communication behaviors, it is challenging to determine which data should be stored in on-chip memory to maximize the performance gain under on-chip memory constraints. On the other hand, the increasing complexity of DNNs poses great challenges for on-chip memory allocation. The traditional double buffer allocation for linear structures used by previous models like AlexNet and VGG is not enough for DNNs with complex graph topology.

In this paper, we propose a layer conscious memory management framework which jointly exploits the layer diversity in terms of computation to communication ratios and the disjoint lifespan information of memory buffers. It helps improve the performance of the memory bounded layers and thus the entire performance of DNNs by efficiently utilizing the on-chip memory. The core part of the framework is a memory allocation algorithm called DNNK. DNNK optimizes the performance of the whole DNN model by allocating the data of some layers in on-chip buffers while other layers accessing data from off-chip memory. Our framework also finds buffer reuse opportunities between tensors in order to improve on-chip memory utilization. For feature maps, a global liveness analysis is applied on the computation graph of a given DNN model to do feature buffer reuse. For weights, a buffer prefetching pass is firstly applied in order to hide the latency of weight loading. Then liveness analysis is again used for weight buffers to do buffer sharing between two layers that have disjoint lifespan. Finally, the framework uses a buffer splitting pass to separate two tensors with huge bandwidth requirement difference.

*Corresponding author.
†J. Cong serves as the Chief Scientific Advisor of Falcon Computing Solutions Inc. and a distinguished visiting professor at Peking University, in addition to his primary affiliation at UCLA.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317875

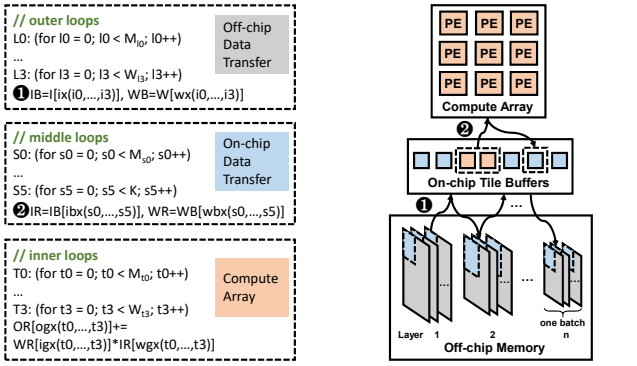
In summary, this paper makes the following contributions,

- **On-chip memory allocation for DNNs.** We design memory allocation algorithm to efficiently utilize the on-chip memory to improve the performance for memory bounded layers.
- **On-chip buffer sharing techniques.** We design buffer sharing techniques in order to improve on-chip buffer utilization. Furthermore, a buffer splitting technique helps improve performance in case the buffer sharing be too radical.
- **Weight buffer prefetching.** We design a weight buffer prefetching technique to reduce weight buffer loading time.

We evaluate our layer conscious memory management framework on three latest DNN models. Compared with the designs using uniform memory management, our techniques achieve up to 1.36X performance improvement on average.

2 BACKGROUND AND MOTIVATION

2.1 Memory Hierarchy of DNN Accelerator

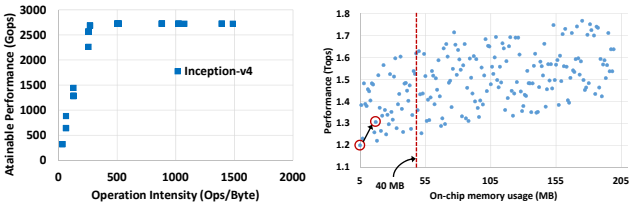


(a) Uniform memory hierarchy dataflow (b) State-of-the-art DNN architecture

Figure 1: Dataflow of previous DNN accelerators

DNNs contain various layers according to their operation types, of which the convolutional layers dominate both computation and storage of network running, especially for the recent models like GoogLeNet [12] and ResNet [6]. Thus the focus of DNN acceleration is still on the architecture design for convolutions. The basic operation of convolution involves a multiply-and-accumulation (MAC) operation on an input feature map tensor and a weight tensor, and returns an output feature map tensor. It contains 6 nested loops with upper bounds (M, C, H, W, K, K). M and C indicate the number of output and input feature maps. H and W are the height and width of a feature map. $K \times K$ is the filter size.

The convolutions have huge volume of computation and bandwidth requirement. Due to the limited on-chip resources, previous FPGA-based DNN accelerators [9, 17, 20, 21] adopt a two-level loop tiling dataflow by tiling each loop level twice and fully unroll the inner one. Then there are three sets of loops, as known as outer loops, middle loops and inner loops as shown in Fig. 1(a), signifying the off-chip data transfer, on-chip data transfer, and parallelism of computation, respectively. The loop orders, loop unrolling factors, loop tiling, as well as the number of loops in each set vary across different designs. But all these designs [9, 11, 17, 20, 21] adopt a uniform memory management methodology. As shown in Fig. 1(b), the input tensors are firstly partitioned into tiles, and then they are repeatedly loaded one after another from off-chip memory to on-chip memory and processed in sequence. The execution of each tile corresponds to an iteration of the *outer loops* $(L_0 - L_3)$. Once a tile is fetched from off-chip memory (**I** and **W**), it is stored in the input tile buffers (**IB** and **WB**) for data reuse (❶). The *middle loops* $(S_0 - S_5)$ represent the sequential processing of feeding data from the input tile buffers to the local buffers (**WR**, **IR** and **OR**) of compute array (❷). Parallel execution represented by the *inner loops* $(T_0 - T_3)$ is performed in the compute array in a pipeline manner. Finally, the output tile is stored back to off-chip memory. All the layers follow the same strategy.



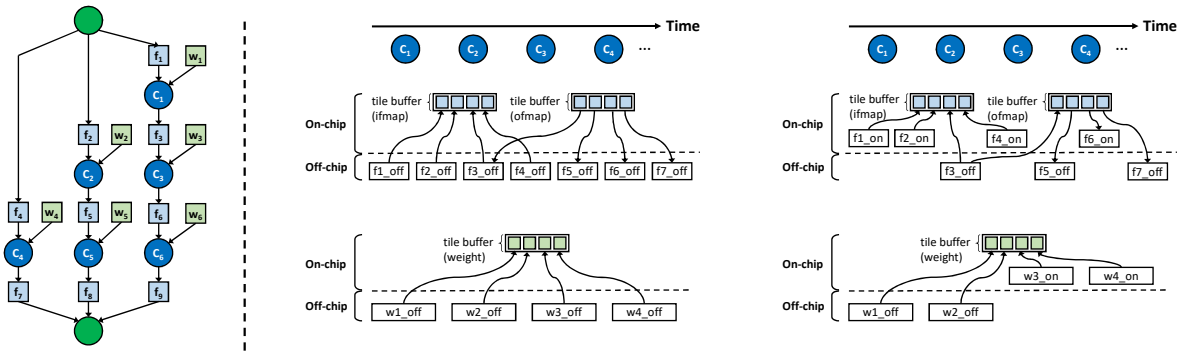
(a) Rooflines of VU9P FPGA (b) Design space of memory allocation

Figure 2: Diversity of layer computation to communication ratios and difficulty in on-chip memory allocation

2.2 Memory Bottlenecks in DNNs

The topologies of DNNs have evolved into general computation graphs. As DNNs are going deeper and becoming more complex, the tensor shapes vary a lot across different layers, resulting in diverse computation and communication ratios [7]. We use the roofline model [18] to perform a layer by layer characterization in terms of both computation and data transfer. The platform we use is Xilinx VU9P FPGA. The peak performance is up to 2.7 Tops under 200 MHz frequency. There are four DDR banks and the theoretical bandwidth is up to 19.2 GB/s on each bank using DDR4. As input features, weights and output features access off-chip memory simultaneously according to the dataflow in Fig. 1, we assume each interface is assigned one third of the theoretical bandwidth which is 25.6 GB/s ($\frac{19.2 \times 4}{3}$). Then we plot the computational roof and bandwidth roof of input feature map in Fig. 2(a) for Inception-v4 using 8-bit data precision. The y-axis denotes the attainable performance (Tops), and the x-axis is the operation intensity (Ops/Byte), which represents operations per off-chip data transfer. Each point in the figure represents the (Tops, Ops/Byte) coordinate for a layer. Note that some points are overlapped. For Inception-v4, there exist 82 memory bound layers, accounting for 58% of the total layers. The same conclusion can be made for other latest models like ResNet, GoogLeNet, etc. In general, it is impractical to put all the activations and parameters in on-chip memory.

Our motivation is to put the data of some layers on-chip to reduce off-chip memory transfer. For a snippet of the block *inception_c1* in Inception-v4 as shown in Fig. 3(a), there are six convolution operations (C) connected by feature outputs (f). The other data source to convolution is weight (w). We firstly unfold the execution of the convolution operations with the uniform memory management method, as shown in Fig. 3(b). Each tensor is allocated a buffer in off-chip memory, and there are three tile buffers of input feature map (*ifmap*), output feature map (*ofmap*) and weight. Note that in current DNN computation graphs, f_1, f_2 and f_4 actually contain the same data. Here we differentiate them for inputs to different operations, and their final buffer allocation is determined by our techniques in Sec. 3. The performance of this model is 1.2 Tops using the design in [17] with 5 MB on-chip memory usage. We plot this point in Fig. 2(b). According to the profiling result in Fig. 2(a), we find that the performance of layers C_3 and C_5 is bounded by computation, while other layers are memory bound. Then we select the memory bounded layers and try to store their data into on-chip memory with limited size. As shown in Fig. 3(c), f_1, f_2, f_4 and f_6 are put in on-chip memory. Similarly, we could apply this for weights. After that, the performance is improved to 1.3 Tops. As layers could have different memory hierarchies during their execution, we call the design method in Fig. 3(c) as layer conscious memory management. Inception-v4 has 14 inception blocks in total.



(a) Computation graph (b) Uniform memory management (c) Layer conscious memory management

Figure 3: Memory footprints of uniform memory management and layer conscious memory management

For each block, we could choose where to store the data (on-chip or off-chip). Collectively, it constitutes a design space containing 16384 (2^{14}) points. We plot these points in Fig. 2(b). The x-axis is on-chip memory consumption (MB) while y-axis is performance (Tops). We can see that more on-chip memory doesn't necessarily mean higher performance. Even around the device limit (40 MB), there are lots of points that have not achieved the highest performance. That's because different layers have different memory bandwidth requirements. The tensor sizes also vary a lot across layers. This layer diversity renders the decision to put the data of which layer on chip difficult. Moreover, different tensors could share the same buffer once their lifespans do not overlap. The complexity of current DNN models imposes challenge for on-chip memory allocation.

3 MEMORY MANAGEMENT FRAMEWORK

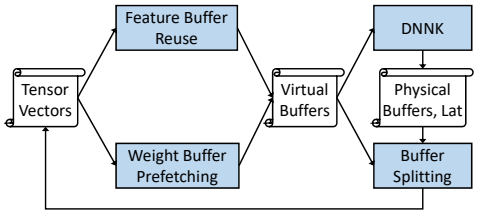


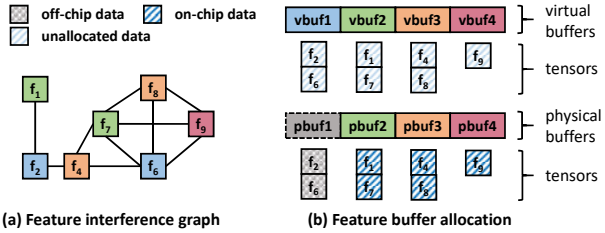
Figure 4: Layer conscious memory management framework

In this section we propose the Layer Conscious Memory Management framework (LCMM) for FPGA-based DNN accelerators. The framework can be integrated into prior design space exploration frameworks for FPGA-based DNN accelerators [11, 17, 20] which optimize the PE array and tile buffer structure of the DNN accelerators for further performance improvement. The flow of LCMM framework is shown in Fig. 4. Given a DNN model, after tensor vectors of feature map and weight are extracted from the computation graph. We firstly set PE array and sizes of tile buffers, and update tensor shapes according to the dataflow in Fig. 1. After that, the feature buffer reuse performs liveness analysis for the tensor vectors to find opportunities to accommodate two feature map tensors in the same buffer. The weight buffer prefetching performs a similar liveness analysis for the weight tensors. The dependency between weight tensors is represented by the prefetching dependence graph rather than the DNN computation graph.

The feature buffer reuse and weight buffer prefetching will be illustrated in detail in Sec. 3.1 and Sec. 3.2. The buffers returned by the two buffer reuse processes are called virtual buffers as they have not been allocated physical on-chip memory. Then the on-chip memory allocation algorithm called *DNN knapsack* (DNNK) is applied to the virtual buffers. Based on the modelling of performance improvement through on-chip buffer allocation, DNNK uses a dynamic programming algorithm to allocate physical buffers for the virtual buffers to derive the minimal latency. DNNK will be introduced in Sec. 3.3. Finally, buffer splitting in Sec. 3.4 is performed if there exist tensors that have not been allocated.

3.1 Feature Buffer Reuse

We observe that the lifespans of different feature tensors could be different, which provides the opportunity to share the same buffer to save storage if the lifespans of two tensors do not overlap. For example, as shown in Fig. 3(a), f_2 and f_6 could share the same buffer because the data consumed by C_2 will not be used again and thereafter the buffer it occupies could be reused by C_3.



(a) Feature interference graph (b) Feature buffer allocation

Figure 5: Feature buffer reuse by liveness analysis

We thereby apply accurate liveness analysis for the whole computation graph to determine the number of on-chip buffers for feature tensors, and an interference graph is built as shown in Fig. 5(a). Note that the computation bounded tensors such as f_3 and f_5 are not included in the interference graph. Then we run a revised graph coloring algorithm [5] to determine the minimal on-chip memory size to accommodate the feature tensors. The problem is solved optimally by left-edge algorithm in polynomial time. Different from the solution in [5], our target is minimizing total size of buffers rather than the number of buffers. The coloring result shows that 4 buffers can be allocated for the 6 feature tensors. The mapping of tensor data onto the allocated buffers is shown in Fig. 5(b). If several tensors could share the same buffer, the size of the buffer is determined by the tensor with the largest size. For example, tensors f_2 and f_6 are mapped on $vbuf1$. Supposing the sizes of f_2 and f_6 are 0.2 MB and 0.1 MB respectively, the size of $vbuf1$ is 0.2 MB. Then after on-chip memory allocation, we can see that the virtual buffers $vbuf2$, $vbuf3$ and $vbuf4$ are allocated with physical on-chip buffers, while $vbuf1$ is spilled to off-chip memory. We will discuss the on-chip memory allocation process in details in Sec. 3.3.

3.2 Weight Buffer Prefetching

Different from features, the lifespans of weights span across the whole execution of the computation graph, hence storage for weight tensors is permanent. On the other hand, the on-chip buffers for weight tensors can be prefetched before they are used by the corresponding nodes. Then weights could be reused for multiple instances of inference. Next, we design a prefetching technique for weight tensor data in order to hide the prefetching overhead.

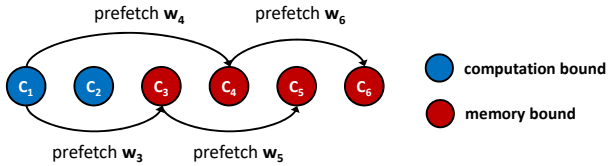


Figure 6: Weight buffer prefetching

For each memory bound node C_k, we compute the time T for loading the weight tensor from off-chip memory to on-chip memory. From C_k we do a backtrace to locate a node $C_{k'}$ to guarantee the elapsed time between $C_{k'}$ and C_k is greater or equal to T. Then we create an edge which ends at C_k and begins with $C_{k'}$. If the weight loading for C_k could begin earlier than the execution of $C_{k'}$, the weight tensor will be on-chip when C_k begins to execute and the loading time could be hidden by the execution of the nodes before C_k. The edge is named as prefetching edge for C_k. After all the prefetching edges are created for the memory bound nodes, we build a graph called prefetching dependence graph (PDG) which expresses the order of prefetching operations for weight tensors. The PDG for the computation graph of Fig. 3(a) is shown in Fig. 6. We can see that if two prefetching edges do not overlap, their end nodes could share the same buffer for weight tensors. Thus for weight tensors of the memory bound nodes, we build a weight interference graph similar to the interference graph for feature tensors in Fig. 5(a). After that, as shown in Fig. 3(c), buffer allocation could also be applied on the weight interference graph to save on-chip buffer sizes for weight tensors.

3.3 DNN Knapsack Allocation

After all tensors are allocated with virtual buffers, we apply on-chip memory allocation for the virtual buffers in order to minimize latency. We use latency as our performance metric, because FPGA is a low latency hardware, especially for DNN applications. The information of each virtual buffer, including buffer size and the tensors sharing the buffer, is recorded in a virtual buffer table as shown in Fig. 7(a). For a computation graph as shown in Fig. 3(a), supposing the number of nodes in the graph is N, the latency $lat_i (0 \le i < N)$ for processing node C_i is equal to the minimum of computation and data transferring according to the dataflow in Fig. 1, as computation and data transferring are executed in parallel with the help of double buffering. Thus lat_i is defined in Eq. 1.

$$lat_i = \max\{lat_i^c, \{x_i^d lat_i^d | d \in \{if, wt, of\}\}\} \tag{1}$$

where lat_i^c is the latency for processing layer i by the computation units, and d denotes a tensor data type which could be if, wt or of, representing the data source is input features, weights or output features respectively. The latency information of a given operation is recorded in the operation latency table as shown in Fig. 7(c). The tensor data are categorized according to the node index in the computation graph, and their data sources. For example, t_i^d denotes a tensor for data source d of node i. Thus lat_i^d denotes the latency for transferring a tile of t_i^d. x_i^d is a binary variable representing whether t_i^d is put on-chip or off-chip: if $x_i^d = 1$, t_i^d is put on-chip; otherwise, it is put in off-chip memory. For t_i^d, we define the metric L_i^d to demonstrate its contribution to latency reduction if it is put in on-chip memory. L_i^d is computed in Equation 2.

$$L_i^d = x_i^d (lat_i^d - \max\{lat_i^{d'} | lat_i^{d'} < lat_i^d\}) \tag{2}$$

The latency reduction of tensor d is equal to the difference between the latency of tensor d and the maximal latency values that are less than lat_i^d. The latency reduction for each tensor could be found in the tensor metric table as shown in Fig. 7(b).

In order to formulate the on-chip memory consumption of all the tensors, we define two binary variables o_i^{dk} and y_k. If tensor t_i^d is assigned virtual buffer $vbuf_k$, $o_i^{dk} = 1$; otherwise, $o_i^{dk} = 0$. y_k indicates whether $vbuf_k$ is allocated with on-chip or off-chip memory. y_k is computed as:

$$y_k = \begin{cases} 1, \text{ if } \sum_i^N \sum_d x_i^d o_i^{dk} \neq 0; \\ 0, \text{ otherwise .} \end{cases} \tag{3}$$

Then the latency reduction for the virtual buffer $vbuf_k$ is computed in Eq. 4.

$$Lb_k = \sum_i^N \sum_d o_i^{dk} (x_i^d L_i^d - x_i^p \sum_{d' < p} (x_i^{d'} L_i^{d'})) \tag{4}$$

In Lb_k, there is a tensor p which we call it the *pivot* tensor. If a pivot tensor is allocated in off-chip memory, the tensors that have been put in on-chip buffers will have no effect on latency reduction if their latency values are less than p. The subtrahend means the sum of latency reduction of these tensors. We call it pivot compensation for latency. For example, considering the 3 tensors of C_4 in Fig. 3(a), the latency reduction for f_7, w_4 and f_4 are 0.01, 0.01 and 0.05 respectively. If we have put f_7 in on-chip memory and w_4 in off-chip memory, then the pivot is w_4. If next f_4 is allocated an on-chip buffer, the latency reduction it brings should be 0.04 (0.05 − 0.01) because the 0.01 has been counted when f_7 is put in on-chip memory. The problem of maximizing latency reduction is then formulated as:

$$\begin{aligned} \text{maximize} \quad & \sum_k^{|\overrightarrow{vbuf}|} y_k Lb_k \\ \text{subject to} \quad & \sum_k^{|\overrightarrow{vbuf}|} y_k S_k \le R_{sram} \end{aligned} \tag{5}$$

where R_{sram} is the total on-chip memory size.

Buf. ID	S (MB)	Start ID	End ID
$vbuf_1$	1.01	1	2
$vbuf_2$	1.18	3	4
...	...	...	...
$vbuf_n$	0.72	n_s	n_e

(a) Virtual buffer table

Tensor ID	L (ms)	IF	OP
...	...	...	...
3	0.05	if	C_1
4	0.01	of	C_2
...	...	...	...
12	0.01	wt	C_4

(b) Tensor metric table

OP	lat^c	lat^{if}	lat^{wt}	lat^{of}
...	...	...	...	...
C_3	0.16	0.15	0.41	0.08
C_4	0.03	0.10	0.05	0.04
C_5	0.16	0.15	0.41	0.08
...	...	...	...	...

(c) Operation latency table

Figure 7: DNN computation graph metric tables

The classic **0-1 knapsack** problem could be reduced to the maximizing latency reduction problem. Meanwhile, we must consider the pivot compensation in Eq. 4 and dynamically update pivot of an operation according to the buffer allocation status of its tensors. Based on the dynamic programming flow for the **0-1 knapsack** problem, we design an algorithm called DNN Knapsack (DNNK) that applies the pivot compensation. The algorithm is shown in Alg. 1. The DNNK algorithm contains two nested loops as shown by line 1 and line 4 in Alg. 1. The first loop iterates through all the unallocated buffers $\overrightarrow{vbuf}$. The second loop sweeps all possible on-chip memory sizes. The pivot compensation is performed (lines 8–11) according to Eq. 5. If a buffer is put in off-chip memory, pivot will be updated as illustrated by lines 14–16.

3.4 Buffer Splitting

Because of buffer reuse between tensors, if a virtual buffer is spilled to off-chip memory, all the tensors sharing this virtual buffer will not be put on chip. For example, in Fig. 5, both f_6 and f_2 will be in off-chip memory if $vbuf1$ is spilled. It may cause the spilling of some tensors that have smaller buffer size requirement but larger latency reduction cost, which is referred as *misspilling*. To alleviate the side effect of misspilling, we apply a buffer splitting pass to tentatively separate the tensors sharing one virtual buffer into multiple virtual buffers. It provides the opportunity for some of the tensors to be reallocated on-chip buffers. The idea of buffer

Algorithm 1: DNNK memory allocation algorithm

input : Unallocated buffer list $\overrightarrow{vbuf}$
Operation latency table T_{lat} and tensor metric table T_{metric}
output: Allocated buffer list $\overrightarrow{pbuf}$

for $i \leftarrow 1$ *to* $|\overrightarrow{vbuf}|$ **do**
  $tensor\_buf \leftarrow \overrightarrow{vbuf}(i)$
  $tensor\_buf.L \leftarrow \sum_{t \in tensor\_buf.tensor\_list} t.L$
  for $j \leftarrow 0$ *to* R_{sram} **do**
    if $j \geq tensor\_buf.S$ **then**
      $L_0 \leftarrow L(i-1, j)$
      $L_1 \leftarrow L(i-1, j - tensor\_buf.S) + tensor\_buf.L$
      `// pivot compensation`
      for $t \in tensor\_buf.tensor\_list$ **do**
        $op \leftarrow t.get\_op(T_{metric})$
        $p \leftarrow op.get\_pivot(T_{lat})$
        $L_1 \leftarrow L_1 - \sum_{d<p} (pbuf\_table(op.get\_idx(d), j)t.L)$
      if $L_0 > L_1$ **then**
        $L(i, j) \leftarrow L_0, pbuf\_table(i, j) \leftarrow 0$
        `// pivot update`
        for $t \in tensor\_buf.tensor\_list$ **do**
          $op \leftarrow t.get\_op(T_{metric})$
          $op.update\_pivot(T_{lat})$
      else
        $L(i, j) \leftarrow L_1, pbuf\_table(i, j) \leftarrow 1$
    else
      $L(i, j) \leftarrow L(i-1, j), pbuf\_table(i, j) \leftarrow 0$
$\overrightarrow{pbuf} \leftarrow backtrace(pbuf\_table)$

splitting is to add a *false* lifespan overlap edge between two tensors which actually have no lifespan overlap, if the variance of sizes and latency reductions between the two tensors exceeds a threshold. The false overlapping edge forces the two tensors that should have been assigned the same buffer to have different colors. If we add a false lifespan overlap between f_2 and f_6 and allocate two virtual buffers to f_2 and f_6 respectively, f_6 will still have the chance to have a buffer allocation in the DNNK process if f_2 doesn't. In each iteration, the rationale of adding a false lifespan overlap edge is to greedily find the virtual buffer with the largest size. Then we add the edge between the tensor which has the same size as the virtual buffer and its neighbor.

4 EXPERIMENT EVALUATION

The LCMM framework can be combined with any of the prior FPGA-based accelerators designs. In this work, we demonstrate the benefit of LCMM by combining it with [17], a representative CNN accelerator using systolic array architecture. The systolic array with uniform memory management (UMM) in [17] will be used as our baseline. We use three latest DNN models as our benchmark suite, which includes ResNet-152 (**RN**), GoogleNet (**GN**) and Inception-v4 (**IN**). All the designs are implemented by Vivado HLS and synthesized by Xilinx SDAccel 2018.2 flow, and are evaluated on the Xilinx VU9P FPGA. We use various precisions, 8- and 16-bit fixed point, and 32-bit floating point data types for evaluation. Our evaluation methodology consists of two parts. We first evaluate LCMM by comparing it to baseline designs. Detailed analysis in terms of performance improvement and resource utilization is given. We also compare our results with the state-of-the-art designs.

4.1 Effectiveness of LCMM

We list the detailed comparison results of designs using UMM and designs using LCMM in Tab. 1. We can see that LCMM outperforms UMM for all benchmarks. The average performance speedup is 1.36X. Taking the 8-bit cases as example, we get 1.42X, 1.23X and 1.17X performance speedup for **RN**, **GN** and **IN** respectively. The speedup comes from two aspects. The first is LCMM improves the efficiency of the memory bound layers through on-chip buffer allocation. Tab. 2 lists the detailed on-chip memory consumption. We can see that LCMM has much higher on-chip memory utilization than UMM due to the usage of tensor buffers instead of tile buffers only. In Tab. 2, we use a metric named Percentage of On-chip

Table 1: Detailed results

Benchmark	Design	Performance			% Utilization			Speedup
		Latency (ms)	Tops	Freq. (MHz)	DSP	CLBs	SRAM	
RN 8-bit	UMM	18.806	1.227	190	83	32	14	1.42
	LCMM	13.258	1.747	180	83	41	86	
RN 16-bit	UMM	22.253	1.126	190	83	44	18	1.46
	LCMM	15.243	1.644	180	83	58	85	
RN 32-bit	UMM	125.720	0.184	180	83	80	22	1.45
	LCMM	86.754	0.266	160	83	87	80	
GN 8-bit	UMM	5.589	0.936	190	83	22	10	1.23
	LCMM	4.650	1.148	180	83	29	88	
GN 16-bit	UMM	6.366	0.668	190	83	42	15	1.29
	LCMM	4.929	0.863	180	83	52	83	
GN 32-bit	UMM	24.454	0.213	160	83	61	22	1.25
	LCMM	19.439	0.269	160	83	70	83	
IN 8-bit	UMM	7.110	1.293	190	75	34	12	1.17
	LCMM	6.030	1.528	180	75	46	89	
IN 16-bit	UMM	9.595	0.968	190	75	50	16	1.36
	LCMM	6.972	1.319	180	75	57	88	
IN 32-bit	UMM	37.515	0.213	170	75	72	21	1.33
	LCMM	28.255	0.325	160	75	86	81	

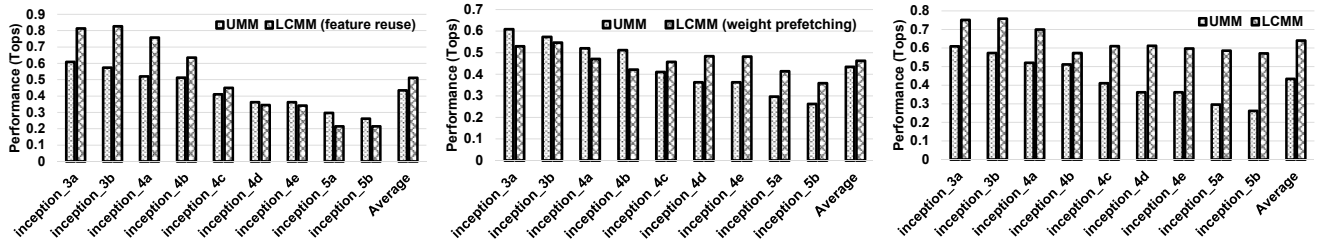
Table 2: On-chip memory utilization

Design	% On-chip Memory Utilization								
	RN			GN			IN		
	BRAM	URAM	POL	BRAM	URAM	POL	BRAM	URAM	POL
UMM 8-bit	8	15	94%	8	10	83%	8	13	78%
LCMM 8-bit	34	87		26	84		26	88	
UMM 16-bit	8	21	94%	8	17	82%	8	18	79%
LCMM 16-bit	30	82		22	86		21	88	
UMM 32-bit	12	25	84%	10	25	61%	10	24	66%
LCMM 32-bit	27	82		28	80		22	80	

Layers (POL) to denote the percentage of layers that benefit from LCMM in the total memory bounded layers. Taking ResNet-152 for example, 14 buffers are allocated to store the 94% of memory bounded tensors, and there are 9 of them consuming 32 URAM blocks. Other buffers consume 64, 96, 128 and 288 URAM blocks. The second reason is from the improvement of computation efficiency. The usage of tensor buffers in LCMM designs help to improve the data reuse. The sizes of tile buffers of LCMM designs is thereby smaller than UMM. Thus, the reduced tile sizes lead to reduction of actual operations [17]. We also find that the improvement of ResNet-152 is higher than GoogLeNet and Inception-v4. That's because the network structure of ResNet is much simpler. Hence the number of required buffer for feature map tensors is less than other two networks. Especially for the 8-bit precision ResNet-152, the allocated buffers could cover all tensors except 3 weight tensors with size 320 memory blocks.

Furthermore, the performance improvement also increases when data precision increases from 8-bit to 16-bit. Then it drops after the precision becomes 32-bit. The reason for the increase is that the bandwidth requirements of operations also increase for the designs use 16-bit precision compared with 8-bit. The sizes of most allocated buffers at 16-bit precision keep the same as 8-bit precision, though. On the contrary, when the bitwidth increases to 32-bit, the sizes of buffers will also increase. On the other hand, it needs 5 DSPs to perform a floating point multiply-and-accumulation (MAC) operation on Xilinx FPGAs. For a fixed point MAC, it needs only 1 DSP. Therefore bandwidth requirement decreases compared with the fixed point data types.

We further use 16-bit implementation of GoogLeNet for more detailed analysis. The results are shown in Fig. 8. We firstly apply feature map reuse only with all weights being stored in off-chip memory. In Fig. 8(a), we can see that from *inception_3a* to *inception_4b*, obvious performance improvement are obtained from feature buffer reuse. There are two reasons for this improvement. Firstly, from *inception_3a* to *inception_4c*, the sizes of feature maps are sufficiently large. Thus the feature map reuse helps mitigate the off-chip memory bandwidth for the layers with small filter sizes like 3 or 1. Afterwards, the bandwidth requirement for the weight increases as the feature map dimensions decrease. Therefore the bandwidth requirements for weights increase and the performance is bounded by data transferring for weights as the feature map sizes



(a) Feature buffer reuse (b) Weight buffer prefetching (c) Overall

Figure 8: Detailed analysis on GoogLeNet

decrease to as small as 14 or 7 when the network goes deeper. On the other hand, if we only use the weight buffer prefetching technique, as shown in Fig. 8(b), from *inception_3a* to *inception_4b*, the performance is bounded by data transferring for input feature maps. After that, thanks to weight buffer prefetching, weights data transferring is no longer the performance bottleneck. Finally, the integration of feature buffer reuse and weight buffer prefetching guarantees the performance improvement across the whole network, which is shown in Fig. 8(c).

4.2 Comparison with State-of-the-art Designs

Table 3: Comparison with State-of-the-art Designs

Design	[3]	Ours	[16]	Ours
DNN Model	ResNet-50		ResNet-152	
FPGA	Xilinx VU9P	Xilinx VU9P	Xilinx VU9P	Xilinx VU9P
Frequency (MHz)	214	180	200	180
DSP Util.	5489 (80%)	5632 (83%)	4096 (60%)	5632 (83%)
BRAM Util. (MB)	7.20 (76%)	3.98 (42%)	6.45 (68%)	2.84 (30%)
URAM Util. (MB)	27.68 (82%)	27.00 (80%)	19.56 (39%)	27.68 (82%)
Logic Util.	728K (62%)	692K (59%)	506K (43%)	776K (66%)
Throughput (Tops)	1.235	1.672	1.463	1.644
Latency/Image (ms)	8.12	6.46	17.34	15.24
Perf. Density (ops/DSPslice/cycle)	1.05	1.65	1.78	1.62

We compare our 16-bit fixed point results with the best two end-to-end results [3, 16] on ResNet. Both performance and resource utilization results are shown in Tab. 3. For performance comparison, our designs have 1.35X and 1.12X speedup in terms of throughput (Tops) over the two designs respectively. The design in [3] consumes more on-chip memory than ours because it tries to keep all intermediate feature maps between different accelerators on-chip. On the contrary, our design only stores the results of memory bound layers in on-chip memory as shown in Tab. 2. The 80% URAM utilization overcomes the off-chip memory bottlenecks for 94% of memory bounded layers, which cannot be achieved without LCMM. Moreover, after the off-chip memory bottleneck is overcome, we could use smaller tile size to improve computation efficiency, leading to less BRAM consumption. Our design has 12% performance improvement over [16] at the cost of 37% more DSP utilization, 20% more on-chip memory consumption, as well as 50% more logic usage. Instead of storing the whole feature maps in on-chip memory, the design in [16] streams a tile of feature map data between accelerators to avoid off-chip memory footprints. In addition, the heterogeneous design in [16] has higher performance density. Fortunately, LCMM is orthogonal to the heterogeneous design methodology which could be integrated into our designs in the future to further improve performance density.

5 CONCLUSIONS

In this work, we propose the layer conscious memory management framework for general DNN accelerator designs on FPGA. LCMM optimizes the performance of DNN accelerators through on-chip memory allocation by exploiting the layer diversity and memory lifespan information. It performs liveness analysis on the feature tensors and prefetching time span analysis on weight tensors to implement reuse on both feature and weight buffers, improving buffer utilization. Experiments show that our techniques can achieve 1.36X performance improvement compared with previous design.

6 ACKNOWLEDGEMENTS

This work is supported by Beijing Natural Science Foundation (No. L172004), Municipal Science and Technology Program under Grant Z181100008918015.

REFERENCES

[1] M. Alwani, H. Chen, M. Ferdman, and P. Milder. 2016. Fused-layer CNN accelerators. In *MICRO*.
[2] U. Aydonat, S. O'Connell, D. Capalija, A. Ling, and G. Chiu. 2017. An OpenCL Deep Learning Accelerator on Arria 10. In *FPGA*.
[3] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen. 2019. Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs. In *FPGA*.
[4] H. Gao, Z. Liu, K. Q. Weinberger, and L. van der Maaten. 2017. Densely Connected Convolutional Networks. In *CVPR*.
[5] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *FCCM*.
[6] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.
[7] X. Lin, S. Yin, F. Tu, L. Liu, X. Li, and S. Wei. 2018. LCP: A Layer Clusters Paralleling Mapping Method for Accelerating Inception and Residual Networks on FPGA. In *DAC*.
[8] L. Lu, Y. Liang, Q. Xiao, and S. Yan. 2017. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. In *FCCM*.
[9] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *FPGA*.
[10] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *MICRO*.
[11] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. Seo, and Y. Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *FPGA*.
[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 2014. Going Deeper with Convolutions. *arXiv* (2014).
[13] C. Szegedy, S. Loffe, V. Vanhoucke, and A. Alemi. 2016. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. *arXiv* (2016).
[14] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *FPGA*.
[15] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *PPoPP*.
[16] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong. 2018. TGPA: Tile-Grained Pipeline Architecture for Low Latency CNN Inference. In *ICCAD*.
[17] X. Wei, Cody H. Yu, P. Zhang, Y Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *DAC*.
[18] S. Williams, A. Waterman, and D. Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communication of ACM* (2009).
[19] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y. Tai. 2017. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *DAC*.
[20] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*.
[21] J. Zhang and J. Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *FPGA*.
[22] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. Hwu, and D. Chen. 2018. DNNBuilder: An Automated Tool for Building High-performance DNN Hardware Accelerators for FPGAs. In *ICCAD*.
[23] R. Zhao, W. Song, W. Zhang, T. Xing, J. Lin, M. Srivastava, R. Gupta, and Z. Zhang. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *FPGA*.