AutoWS: Automate Weights Streaming in Layer-wise Pipelined DNN Accelerators

Zhewen Yu, Christos-Savvas Bouganis
Imperial College London
London, UK
{zhewen.yu18, christos-savvas.bouganis}@imperial.ac.uk

***Abstract*—With the great success of Deep Neural Networks (DNN), the design of efficient hardware accelerators has triggered wide interest in the research community. Existing research explores two architectural strategies: sequential layer execution and layer-wise pipelining. While the former supports a wider range of models, the latter is favoured for its enhanced customization and efficiency. A challenge for the layer-wise pipelining architecture is its substantial demand for the on-chip memory for weights storage, impeding the deployment of large-scale networks on resource-constrained devices. This paper introduces AutoWS, a pioneering memory management methodology that exploits both on-chip and off-chip memory to optimize weight storage within a layer-wise pipelining architecture, taking advantage of its static schedule. Through a comprehensive investigation on both the hardware design and the Design Space Exploration, our methodology is fully automated and enables the deployment of large-scale DNN models on resource-constrained devices, which was not possible in existing works that target layer-wise pipelining architectures. AutoWS is open-source: https://github.com/Yu-Zhewen/AutoWS**

I. Introduction

Recently, there is a broad interest in designing efficient FPGA accelerators for Deep Neural Networks (DNNs), aiming to optimize the trade-off between resource usage and performance. Two primary architectural strategies have emerged: sequential layer execution and layer-wise pipelining. The former involves the execution of DNN layers onto a single Compute Engine (CE) with time-multiplexing [1]. In contrast, the latter strategy employs customized CE for each layer, interconnected in a chained manner for improved efficiency [2].

A crucial difference between these two strategies is their memory requirements. Sequential layer execution stores both weight and activation data off-chip, leading to intensive off-chip memory access. Techniques such as tiling and double buffering are commonly employed to mitigate data communication bottlenecks by maximizing data reuse and hiding latency. Existing research on layer-wise pipelining [2]–[4], on the other hand, restricts off-chip memory access to the inputs of the first CE and the outputs of the last CE, with intermediate activation data streamed through the CEs. Furthermore, all DNN parameters (weights) are preloaded into on-chip memory before inference begins. However, the substantial demand for the on-chip memory impedes the deployment of large-scale networks on resource-constrained devices.

To overcome this bottleneck, this paper introduces a novel memory management methodology for the layer-wise pipelining architecture, that exploits both on-chip and off-chip memory for weights storage. Our approach introduces memory fragmentation and in this work we address the challenges of deciding the memory fragmentation parameters and bandwidth allocation when dealing with multiple pipelined CEs, offering an automated solution to these critical issues. Furthermore, we bundle the above memory subsystem with a parameterised layer-wise pipelining architecture resulting the first such accelerator that supports partial weight streaming. Figure 1 illustrates the architectural distinctions between this work and existing solutions. Our contributions can be summarized as follows:

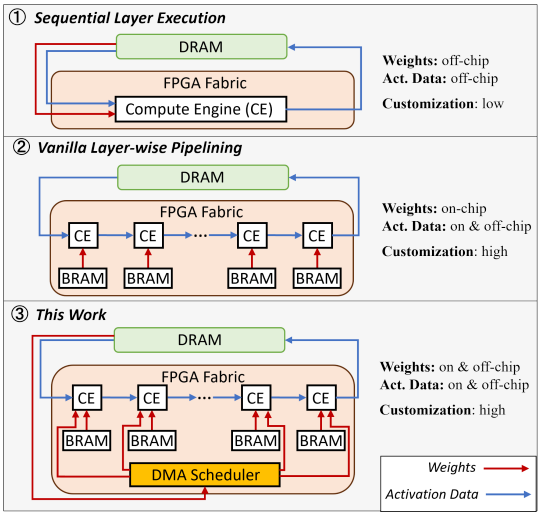


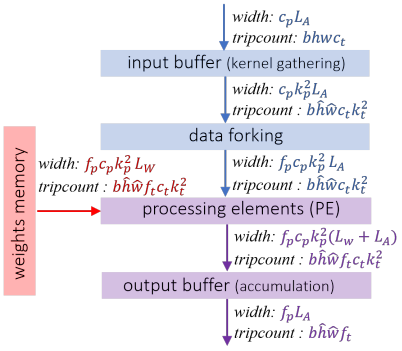
Fig. 1: Comparison of existing designs and our architecture.

- A scalable CE template featuring tunable unroll factors, and a flexible memory structure supporting static and dynamic weights loading, tunable in per-layer basis.
- A Design Space Exploration (DSE) process, which balances the processing rates of CEs and optimizes the allocation of off-chip bandwidth in an iterative and greedy way, based on our performance and resources modeling.
- A deterministic DMA scheduler that links off-chip memory to multiple CEs in a time-multiplexed manner, allowing the efficient exploitation of off-chip bandwidth with two clock domains.

II. Related Works

Early research on DNN acceleration focuses on sequential layer execution which only exploits intra-layer parallelization. Examples include DnnWeaver [5], Angel-Eye [6] and

979-8-3503-4859-0/DATE24/© 2024 EDAA



Symbols	Definitions
b	batch size
c	input channel number
h, w	input height/width
k	kernel size
$\hat{h}, \hat{w}$	output height/width
f	filter number
L_W	weights bitwidth
L_A	activations bitwidth

x_p and x_t refer to the parallelism (unroll factor) and the tripcount of the loop iterates over any given symbol x, where $x = x_p x_t$

Fig. 2: Dataflow of the compute engine.

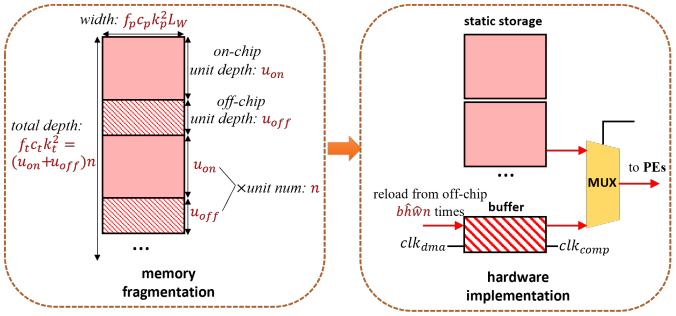


Fig. 3: Fragmentation of the weights memory. The memory structure is split into the static regions that stay on-chip all the time and the dynamic regions which are reloaded from off-chip.

Snowflake [7]. Subsequently, this line of research evolved into the use of the systolic array design, with investigations into efficient weight and activation data reuse strategies [8], [9]. Accelerators adopting the sequential layer execution architecture are often designed to be general-purpose. For instance, in Vitis AI [1], a single DPU IP configuration, leveraging a dedicated instruction set, can accelerate a wide range of different DNNs.

In contrast, layer-wise pipelining follows a distinct design methodology where the accelerator design is customized to each specific DNN workload, leading to improved performance. Previous work in layer-wise pipelining has predominantly focused on the efficient utilization of on-chip memory resources. Notable examples include fpgaConvNet [3], which relies on synthesis tools to determine the suitable resource type for weight storage (e.g., BRAMs or LUTRAMs), and hls4ml [4], which provides users with control over this design aspect. DNNExplorer [10] incorporates this design choice into its Design Space Exploration (DSE) process. Furthermore, the authors of FINN [2] observed that BRAMs might not be fully utilized due to parallel computation falling short of the fabric's provision. They addressed this issue by optimizing BRAM utilization through overclocking.

It is important to note that existing research on the layer-wise pipelining architecture has primarily concentrated on the optimization of on-chip memory only. In this paper, we demonstrate a novel memory management scheme that exploits both on-chip and off-chip memory, with the whole process automated.

III. Compute Engines

Our architecture is depicted as ③ in Figure 1. In this section, we focus on the internal structure of the proposed Compute Engine (CE), including its computational dataflow and memory structure, all of which can be tailored on a per-layer basis.

A. Dataflow and Parallelization

As depicted in Figure 2, the dataflow involves interconnected building blocks, facilitated by FIFOs with handshake interfaces:

- **Input buffer**: exists in convolution and pooling operations. A k×k 2D window slides over the spatial dimensions h, w of activation data, implemented using shift registers to maximize data reuse.
- **Data forking**: exists only in convolution operations and duplicates the incoming activation data for f copies, corresponding to f different filters.
- **Weights memory**: stores the weights in convolution and fully connected operations. More details on its implementation are provided in Section III-B and Figure 3.
- **Processing elements (PEs)**: an array of parallel processing elements that handle elementwise operations such as multiplication, addition, and ReLU activations. In cases where weights memory is not involved, this array may consume multiple activation data streams.
- **Output buffer**: utilized in convolution, this buffer accumulates incoming activation data streams across the 2D window and channel dimensions.

B. Weights Storage

For easy illustration, we discuss the weight storage for convolutional layers, as any fully connected layer can be generalized to the case that k, h, w are equal to one. In existing layer-wise pipelined designs [2], [3], the required on-chip memory depth and width for a convolutional layer should be

$$M^{dep} = f_t c_t k_t^2, \quad M^{wid} = f_p c_p k_p^2 L_W \tag{1}$$

respectively, to prevent any computation stalls within the PEs. The symbols used here are defined in Figure 2.

One novelty of the proposed work is in the introduction of memory weight fragmentation. Under this scheme, the original weight memory structure is fragmented into static and dynamic regions (Figure 3), where the weights under the static regions are stored as in the conventional approaches, where the dynamic regions are sharing the same physical memory structure in a time-multiplexed manner.

Specifically, there are n fragments stored in the on-chip memory, each with a depth of u_{on}; and n fragments in the off-chip memory, each with a depth of u_{off}. The memory width remains the same as before. Therefore, the total depth of on-chip and off-chip memory can be represented as:

$$M_{on}^{dep} = u_{on} n, \; M_{off}^{dep} = u_{off} n, \; M^{dep} = M_{on}^{dep} + M_{off}^{dep} \tag{2}$$

The shared off-chip buffer is implemented with dual-port Block RAMs (BRAMs), supporting different clocks and port widths

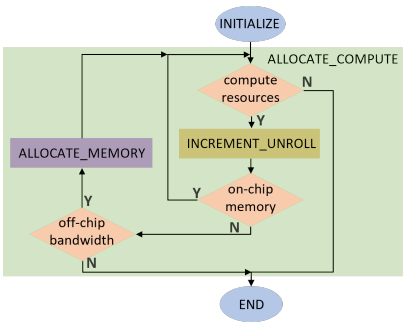


Fig. 4: A high-level visualization of Algorithm 1

on the read and write sides, and allowing for independent control and data-transfer rates. As a result, the proposed architecture incorporates two distinct clock domains:

- clk_{comp}: controls the execution of various computations within the CEs and the reading of the shared buffer.
- clk_{dma}: manages the process of loading weights from the off-chip memory and writing them into the shared buffer.

During run-time, the PE array iteratively reads weights between the static on-chip storage and the off-chip buffer, controlled by address counters and additional "Read-After-Write" checking. This iterative process repeats r times.

$$r = b\hat{h}\hat{w}n, \tag{3}$$

as convolution weights are reused on b, $\hat{h}$ and $\hat{w}$ dimensions (Figure 2).

C. Resource and Performance Modelling

In the proposed CE template, k_p, c_p and f_p control the parallelism of computations, while n, u_{on} and u_{off} dictate the memory storage structure (Figure 2, Equation 2). These variables, along with other user-defined parameters, including the computation clock frequency clk_{comp}, type of operation O, weights bitwidth L_W and activations bitwidth L_A, define the configuration of the CE. The combination of these variables and parameters allows us to estimate the area a, the off-chip bandwidth β, and the throughput θ of a single CE.

$$\mathcal{V} = \{k_p, c_p, f_p, n, u_{on}, u_{off} | clk_{comp}, O, L_W, L_A\} \Rightarrow a(\mathcal{V}), \beta(\mathcal{V}), \theta(\mathcal{V}) \tag{4}$$

The area estimation a is calculated based on regression models developed by randomly sampling values of the tunable variables and collecting post-synthesis results. The throughput estimation θ, is based on analytical models, leveraging the predictable parallelism behavior that can be analyzed in a cycle-accurate manner. Our approach to estimating area and throughput follows the same methodology as the previous work [2], [3]. In addition, specific to this work, The average off-chip bandwidth required by an individual CE is estimated by:

$$\beta(\mathcal{V}) = M^{wid} \cdot clk_{comp} \cdot \frac{u_{off}}{u_{on}+u_{off}} \tag{5}$$

Here, the product of the first two terms represents the number of bits required per second. The final scaling term accounts for the dual-port buffer's ability to load weights from off-chip memory, irrespective of whether the PEs are reading from on-chip storage or the off-chip buffer itself.

Algorithm 1 Greedy DSE

Require: $\mathcal{D}$, A, B ▷ target DNN, area constraint, bandwidth constraint
procedure INITIALIZE($\mathcal{D}$)
  for $l \in \mathcal{D}$ **do**
    $k_{p\_l}, c_{p\_l}, f_{p\_l} \leftarrow 1$ ▷ minimize compute resources
    $M^{dep}_{off\_l} \leftarrow 0$ ▷ all weights on-chip

procedure DELTA_BANDWIDTH($\mathcal{D}, l$)
  $\mathcal{D}' \leftarrow \mathcal{D}; l' \leftarrow l$; INCREMENT_OFFCHIP($l'$)
  return $\Delta B \leftarrow \sum_{l' \in \mathcal{D}'} s_{l'}\beta_{l'}$ - $\sum_{l \in \mathcal{D}} s_l \beta_l$ ▷ bandwidth difference
procedure WRITE_BURST_BALANCE($\mathcal{D}, l$)
  $r_{max} = max(r_{l'}$, for $l' \in \mathcal{D}$ and $l'! = l)$ ▷ Equation 10
  return $r_{max}/(b\hat{h}_l\hat{w}_l)$
procedure INCREMENT_OFFCHIP($\mathcal{D}, l$)
  $M^{dep}_{off\_l} \leftarrow M^{dep}_{off\_l} + \mu$; ▷ μ, hyperparameter
  $n_l \leftarrow$ WRITE_BURST_BALANCE($\mathcal{D}, l$)
procedure ALLOCATE_MEMORY($\mathcal{D}$, A, B)
  while $\sum_{l \in \mathcal{D}} a^{mem}_l > A^{mem}$ **do** ▷ on-chip mem limit
    $l \leftarrow$ SORT_BY($l \in \mathcal{D}$, DELTA_BANDWIDTH($\mathcal{D}, l$))[0]
    $\mathcal{D}' \leftarrow \mathcal{D}; l' \leftarrow l$; INCREMENT_OFFCHIP($\mathcal{D}', l'$)
    if $\beta'_{io} + \sum_{l' \in \mathcal{D}'} s_{l'}\beta_{l'} > B$ **then return** False ▷ bandwidth limit
    $\mathcal{D} \leftarrow \mathcal{D}'$
  return True

procedure INCREMENT_UNROLL(l)
  for $v_l \in \{k_l^2, f_l, c_l\}$ **do**
    if $v_{p\_l} < v_l$ **then**
      $v_{p\_l} \leftarrow v_{p\_l} + \phi$; **return** TRUE ▷ ϕ, hyperparameter
  return FALSE
procedure ALLOCATE_COMPUTE($\mathcal{D}$, A, B)
  while $\sum_{l \in \mathcal{D}} a_l \leq A$ **do**
    $l \leftarrow$ SORT_BY($l \in \mathcal{D}$, θ_l)[0] ▷ slowest layer
    $\mathcal{D}' \leftarrow \mathcal{D}; l' \leftarrow l$; $S_1 \leftarrow$ INCREMENT_UNROLL(l')
    $S_2 \leftarrow$ ALLOCATE_BANDWIDTH($\mathcal{D}'$, A^{mem}, B)
    if $\sum_{l' \in \mathcal{D}'} a_{l'} > A'$ or $!S_1$ or $!S_2$ **then** break ▷ area limit
    $\mathcal{D} \leftarrow \mathcal{D}'$

INITIALIZE($\mathcal{D}$); ALLOCATE_UNROLL($\mathcal{D}$, A, B);
return $\min_{l \in \mathcal{D}} \theta_l$

IV. Macro-architecture

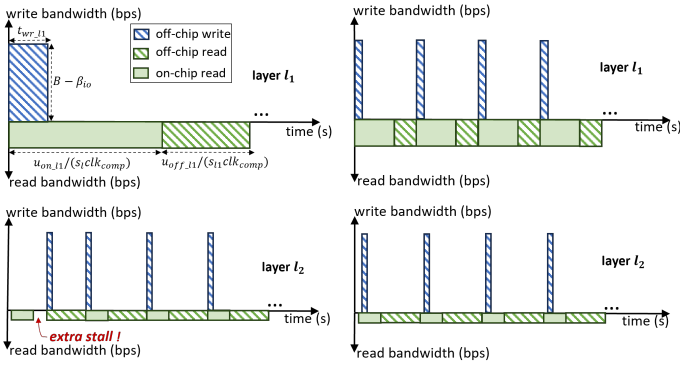
Let us denote the DNN model as $\mathcal{D}$, where each layer, denoted as $l \in \mathcal{D}$, is mapped to a CE on the hardware. To distinguish layer-specific values, we introduce the subscript $_l$ into previously defined symbols.

A. Design Space Exploration

Due to the layer-wise pipelined architecture, CEs are interconnected using FIFOs to accommodate variations in processing rates and data port width. Consequently, the overall throughput of the pipeline is determined by the slowest CE, leading to a resource-constrained optimization problem:

$$\max(\min_{l \in \mathcal{D}} \theta_l) \quad s.t. \;\; \beta_{io} + \sum_{l \in \mathcal{D}} s_l \beta_l \leq B, \;\; \sum_{l \in \mathcal{D}} a_l \leq A \tag{6}$$

B and A denote the device constraint on off-chip bandwidth and area respectively. β_{io} accounts for the bandwidth cost for the first PE to read input samples and the last PE to write prediction results, as illustrated in Figure 1.



(a) imbalanced burst numbers (b) balanced burst numbers

Fig. 5: Two-layer example of write/read scheduling

s_l is defined as the "slow-down" factor, which quantifies the throughput ratio between the slowest CE and the current CE. It accounts for situations where the processing rates of different CEs are not perfectly matched. In such cases, the required off-chip bandwidth can be scaled down proportionally, without impacting the overall pipeline throughput.

$$s_l = \frac{\min_{l \in \mathcal{D}} \theta_l}{\theta_l} \tag{7}$$

The objective of Equation 6 is to maximize the accelerator's throughput by identifying the optimal combination of $\mathcal{V}$ (Equation 4) for all CEs. However, conducting an exhaustive search can be time-consuming. To address this, our DSE method (Algorithm 1) employs a greedy approach, which iteratively optimizes the computation and memory (Figure 4).

- **greedy compute allocation**: In this step of the optimization, our heuristic is to incrementally promote the throughput of the slowest CE. We achieve this by iteratively increasing the unroll factors in dimensions such as k_l^2, f_l, c_l by a user-defined step size ϕ. After each adjustment, we re-evaluate the CE's throughput and repeat the process. When the allocated on-chip memory area exceeds the limit A_{mem}, we transition to the next phase, which focuses on off-chip bandwidth allocation.
- **greedy memory allocation**: Initially, we begin with a design where all weights reside on-chip. In each iteration, we select a layer and evict one memory block to off-chip memory. This block has a depth of μ and a width of M_l^{wid} words. The choice of the layer is to minimize the marginal impact on bandwidth due to this eviction. Based on the total memory depth evicted to off-chip storage, denoted as $M_{off\_l}^{dep}$, the algorithm calculates the optimal number of memory fragments n_l using a "write burst balancing" strategy (explained in the following section). This calculation guides the determination of u_{on_l} and u_{off_l} as per Equation 2.

In this DSE algorithm, two user-defined hyperparameters, ϕ and μ, control the step sizes of the exploration. The choice of these hyperparameters affects the trade-off between exploration time and solution optimality. A larger step size accelerates exploration, but may lead to sub-optimal solutions.

TABLE I: Characteristics of evaluated models. The accuracy of some quantized models is higher than the floating point versions due to extra fine-tuning.

Network	ImageNet Accuracy				Params	MACs
	W4A4 [11]	W4A5 [12]	W8A8 [1]	F32		
mobilenetv2	65.6	65.7	67.7	71.9	3.5M	0.3G
resnet18	70.3	70.5	70.0	69.8	11.7M	1.8G
resnet50	-	77.3	76.0	76.1	25.6M	4.1G

B. DMA Connection and Scheduling

At the microarchitecture level, we employ a demultiplexer to manage the routing between the DMA port and multiple CEs. The demultiplexer is controlled by a configuration sequence that outlines the order and the duration of serving each individual CE.

In the **top-left** region of Figure 5, we present the write and read scheduling for layer l_1. Writing operates in burst mode to fill the off-chip memory buffer, fully utilizing the available device bandwidth, $B-\beta_{io}$, after deducting the input and output transmissions. Therefore, we can calculate the duration of this write burst as:

$$t_{wr\_l1} = \frac{M_{l1}^{wid} \cdot u_{off\_l1}}{(B-\beta_{io})} \tag{8}$$

The interval between burst writes is the sum of time spent on reading the static on-chip storage and off-chip buffer:

$$t_{rd\_l1} = \frac{u_{on\_l1} + u_{off\_l1}}{s_{l1} clk_{comp}} \tag{9}$$

And this pattern will repeat for r_{l1} times (Equation 3).

Moving to the **bottom-left** of the figure, which displays the scheduling of the layer l_2 instead. Here, we assume layer l_2 is connected after l_1 in the pipeline. Consequently, the first read of layer l_2 begins slightly later than that of layer l_1 due to the pipeline depth between these two layers. In addition, r_{l2} is four times r_{l1}, so as the number of burst writes. However, this mismatch in the number of bursts introduces additional stalls in l_2 when the DMA is occupied with writing the substantial weight chunk for l_1. These stalls occur r_{l1} times in total, affecting overall performance.

In contrast, looking at the **top right** and the **bottom right** of Figure 5, where r_{l1} is set equal to r_{l2}, those stalls are diminished. Therefore, we employ the "write burst balancing" strategy in Algorithm 1, that enforces the following condition:

$$\forall l_1, l_2 \in D,\ r_{l1} = r_{l2} \tag{10}$$

This condition ensures that the number of bursts for different layers is equal, avoiding the stalls and optimizing overall performance.

V. Evaluation

A. Experiment Setup

We target the AMD Xilinx FPGA devices and use Vivado 2019.1 for hardware synthesis. Regarding the DNNs, we deploy

TABLE II: Latency (ms) results across different networks and devices. * denotes W4A4, † denotes W4A5 , ◇ denotes W8A8

mobilenetv2			
Architecture	Device		
	Zedboard	ZC706	ZCU102
layer-sequential	8.3* [11]	7.3* [11]	5.3† [12]
vanilla layer-pipelined	✗	9.2*	2.3†
this work	325.9*	4.8*	2.3†

resnet18			
Architecture	Device		
	ZC706	ZCU102	U50
layer-sequential	40.4* [11]	13.7† [12]	3.0◇ [1]
vanilla layer-pipelined	✗	✗	1.3◇
this work	27.0*	7.0†	1.3◇

resnet50			
Architecture	Device		
	ZCU102	U50	U250
layer-sequential	21.1† [12]	6.0◇ [1]	5.6◇ [1]
vanilla layer-pipelined	✗	15.0◇	1.8◇
this work	578.7†	3.4◇	1.8◇

the quantized models provided by existing research [1], [11], [12], and their accuracy, number of Multiply–ACcumulate (MAC) operation, and parameters are summarized in Table I.

We extend fpgaConvNet [3], which is an open-source toolflow, that generates layer-wise pipelined accelerators. We build our own weights fragmentation (Figure 3), DSE method (Algorithm 1), and DMA scheduling (Figure 5) upon that toolflow. In the rest of the paper, we refer to the original fpgaConvNet, which did not exploit off-chip weights storage, as the "vanilla layer-pipelined" approach.

As our design methodology is fully automated, it can also be easily integrated into other layer-wise pipelined toolflows, such as FINN [2] and hls4ml [4], in the future.

B. Overall Results

Table II provides a comparative analysis of our methodology, the "vanilla layer-pipelined" approach, and other "layer-sequential" architectures. We define device size relative to model parameters; for example, ZCU102 is "large" for MobileNetV2 but "small" for ResNet50 due to its larger parameter size. Key observations include:

- "Vanilla layer-pipelined" excels on "large" FPGA devices with ample on-chip memory. For example, mapping MobileNetV2 to ZCU102 achieves 2.3ms latency, less than half of "layer-sequential" (5.3ms). Similar trends apply to ResNet18 on U50 and ResNet50 on U250.
- Our methodology maintains latency on these "large" devices, as our greedy DSE automatically determines that there is no need to store weights off-chip, and the designs become primarily compute-bound.
- When on-chip memory resources become bottleneck, the "vanilla layer-pipelined" approach may become inferior to "layer-sequential" or, in some cases, may not fit the device at all (marked as "✗" in the table).
- The advantage of our proposed methodology becomes evident on these "smaller" devices. For example when mapping ResNet50 to U50, our approach reduces the latency from 15.0ms (in the "vanilla layer-pipelined" approach) to 3.4ms. It also surpasses the "layer-sequential" approach, which requires 6.0ms.
- In some cases, such as MobileNetV2 on Zedboard and ResNet50 on ZCU102, "layer-sequential" achieves the lowest latency. This is because the off-chip bandwidth on these devices is limited compared to the number of parameters in those models. This bandwidth constraint restricts the full application of our proposed methodology. Additionally, the overhead of implementing per-layer FIFOs and buffers becomes significant in these cases.

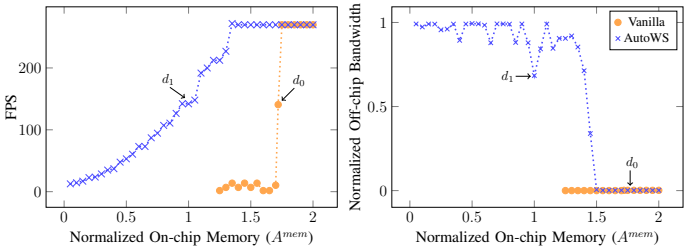


Fig. 6: resnet18-ZCU102, memory and performance trade-off. Normalization is against the max resource available on device

TABLE III: resnet18-ZCU102, memory resource breakdown

Design Point	Off-chip BW (Gbps)			BRAM Usage (MB)				DSP	FPS
	act	wt	total (util.)	act_fifo	wt_buff	wt_mem	total(util.)		
Vanilla (d_0)	0.1	0.0	0.1 (0%)	0.4	0.0	8.3	8.7 (172%)	1113	141
AutoWS (d_1)	0.1	105.0	105.1 (68%)	0.4	0.1	4.6	5.1 (99%)	1180	142

In summary, the results in Table II demonstrate that "layer-pipelined" approaches have a clear advantage over "layer-sequential" approaches on "large" devices with ample on-chip memory resources. Our work extends this advantage to more resource-constrained devices by considering the access requirements of the weights and leveraging off-chip memory bandwidth.

C. Case Study: resnet18-ZCU102

In this section, we provide a case study that focuses on mapping ResNet18 to ZCU102, offering detailed insights into our implementation. Firstly, we conducted a parameter sweep, as depicted in Figure 6, systematically adjusting the budget of on-chip memory (A^{mem}), while keeping the budgets of compute resource (LUT, DSP) and off-chip bandwidth fixed. All resource numbers are normalized to the specifications of a single ZCU102 device.

Based on the value of A^{mem}, Figure 6 on the left can be split into three regions:

- $[0, 1.25)$: Here, the "vanilla" approach cannot fit, resulting in no provided design points. However, AutoWS exhibits steadily improved throughput as on-chip memory resources increase.
- $[1.25, 1.75)$: The "vanilla" approach is feasible but lags behind AutoWS in throughput, suggesting the bottleneck changes from the memory capacity to the bandwidth.

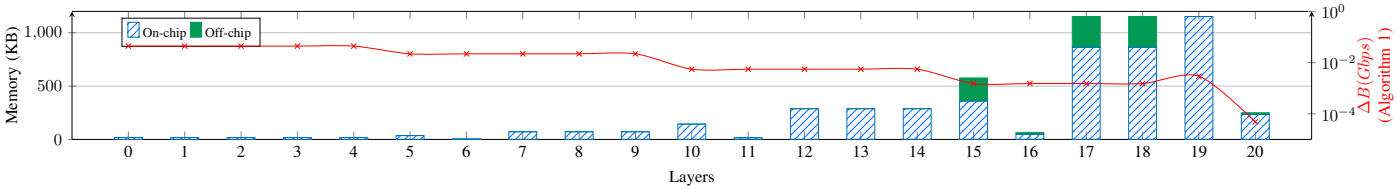


Fig. 7: resnet18-ZCU102, per-layer on-chip and off-chip allocation for the design point d_1

- $[1.75, 2)$: In this range, both the "vanilla" approach and AutoWS converge to the same set of design points as the accelerator becomes compute-bound.

Table III offers a detailed resource breakdown for two specific design points, namely d_0 and d_1, selected from Figure 6. These design points correspond to the "vanilla" approach and AutoWS, respectively. For d_0, its off-chip bandwidth is notably underutilized, as the "vanilla" approach did not account for weight transfers (wt). Conversely, AutoWS (d_1) effectively utilizes this resource.

Regarding BRAM usage, it is calculated as the product between the number of BRAMs and the maximum capacity per BRAM. The usage falls into three categories:

- **act_fifo**: FIFOs and buffers connecting PEs and storing intermediate activation data.
- **wt_buff**: buffers for loading off-chip weights.
- **wt_mem**: static on-chip weight storage (Figure 3).

The costs of **act_fifo** and **wt_buff** are relatively minor (below 10%) compared to **wt_mem**. A comparison between design points d_0 and d_1, with similar throughput, reveals that AutoWS saves BRAM utilization by 70%.

Furthermore, Figure 7 shows the layer-wise memory allocation of our DSE algorithm. In this case, 5 out of 21 layers have part of the weights stored off-chip (layers 15 to 18 and 20). The selection of these layers prioritizes minimal bandwidth impact with smaller ΔB, as outlined in Algorithm 1. We visualize this criterion as the red curve in Figure 7, with the corresponding values obtained at the end of DSE. The savings in BRAM in Table III are actually larger than the size of the off-chip weights in Figure 7, as they are two different metrics. Some BRAMs were not fully filled in d_0 and the corresponding weights are now moved off the chip in d_1.

D. Object Detection

Furthermore, we evaluate the effectiveness of our proposed methodology in the context of the COCO object detection task, using the quantized to 8 bits YOLOv5n model, and targeting the ZCU102. AutoWS (8.7ms) achieves a 36% latency reduction compared to Vitis AI (13.7ms) [1], and a 9% reduction compared to the "vanilla layer-pipelined" (9.5ms).

VI. Conclusion

In this paper, we introduce AutoWS, a novel memory management methodology capable of partially reloading weights from off-chip memory and efficiently delivering them to multiple pipelined CEs. Our hardware design, which is template-based and adaptable through a greedy DSE process, builds upon the advantages of "vanilla layer-pipelined" approaches over "layer-sequential" architectures. Moreover, we extend these benefits to resource-constrained devices. Future work would explore software-hardware co-design, such as weight encoding and pruning methods, to further enhance performance.

Acknowledgement

For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

References

[1] V. Kathail, "Xilinx vitis unified software platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 173–174.

[2] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotofana, and M. Blott, "Memory-efficient dataflow inference for deep cnns on fpga," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 48–55.

[3] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A toolflow for mapping diverse convolutional neural networks on embedded fpgas," *arXiv preprint arXiv:1711.08740*, 2017.

[4] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. Di Guglielmo, P. Harris, J. Krupa *et al.*, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," *arXiv preprint arXiv:2103.05579*, 2021.

[5] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[6] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 37, no. 1, pp. 35–47, 2017.

[7] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: A model agnostic accelerator for deep convolutional neural networks," *arXiv preprint arXiv:1708.02579*, 2017.

[8] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[9] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[10] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnexplorer: a framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[11] S.-E. Chang, Y. Li, M. Sun, R. Shi, H. K.-H. So, X. Qian, Y. Wang, and X. Lin, "Mix and match: A novel fpga-centric deep neural network quantization framework," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 208–220.

[12] M. Sun, Z. Li, A. Lu, Y. Li, S.-E. Chang, X. Ma, X. Lin, and Z. Fang, "Film-qnn: Efficient fpga acceleration of deep neural networks with intra-layer, mixed-precision quantization," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 134–145.