PipeFuser: Building Flexible Pipeline Architecture for DNN Accelerators via Layer Fusion

Xilang Zhou[1†], Shuyang Li[1], Haodong Lu[2], Kun Wang[1*]
[1]State Key Laboratory of ASIC & System, Fudan University, Shanghai, China.
[2]Nanjing University of Posts and Telecommunications, Nanjing, China.
Email: †22112020068@m.fudan.edu.cn, *kun.wang@ieee.org

2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC) | 979-8-3503-9354-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ASP-DAC58780.2024.10473790

***Abstract*—In this paper, we propose a fused-pipeline architecture that leverages the layer fusion technique to harness the strengths of both non-pipeline and full-pipeline architectures while mitigating their disadvantages. In particular, we observe that the performance of the fused-pipeline accelerators is significantly influenced by the layer fusion strategies and intra-layer mapping schemes. To optimize and rapidly employ the fused-pipeline architecture, we present an end-to-end automation framework, named PipeFuser. At the core of PipeFuser is a genetic algorithm (GA)-based co-design engine, which is used to acquire near-optimal hardware configurations in the vast design space. Experimental results demonstrate that our fused-pipeline architecture achieves 2.3× to 3.3× higher performance over the non-pipeline design and 1.9× to 2.5× speedup compared to the full-pipeline architecture, with greater deployment flexibility.**

I. Introduction

DNNs have achieved transformative advancements across diverse domains and reshaped established computational research and application frameworks. However, the evolving depth and intricacy of DNN models pose significant challenges in hardware deployment, leading to a surge in research on DNN accelerators. Currently, state-of-the-art DNN accelerators mainly follow two archiectural paradigms.

To be specific, the first architectural paradigm features a non-pipeline, CPU-like architecture, thus also termed as DNN processors [5], [7], [16]–[18]. As shown in Fig. 1(a), this design employs a uniform processing unit to sequentially process each DNN layer. To support diverse DNN models without hardware modifications, it is necessary to adopt an instruction-centric control scheme. Nevertheless, it requires careful optimization of hardware resources to accommodate the diverse dimensions and access patterns of different DNN layers. In addition, frequent off-chip memory access presents pronounced limitations in both latency and power consumption [1].

The second paradigm typically adopts a full-pipeline architecture, as depicted in Fig. 1(b), where optimized hardware resources are finely tailored to each layer [13], [20]. This full-pipeline approach enables the swift transition of intermediate values to subsequent compute units upon generation, thereby reducing the number of off-chip memory access. However, the full-pipeline design also encounters some challenges regarding limited practicality: 1) With DNN layer counts increasing, the per-layer resources become scarce due to stringent area [4]; 2) An increase in model depth or input dimensions result in larger on-chip buffering capacity demand [21]; 3) The pipeline imbalance caused by DNN network irregularity often induces stall; 4) The centralized control mechanism inherent to full-pipeline architectures leads to a high timing overhead.

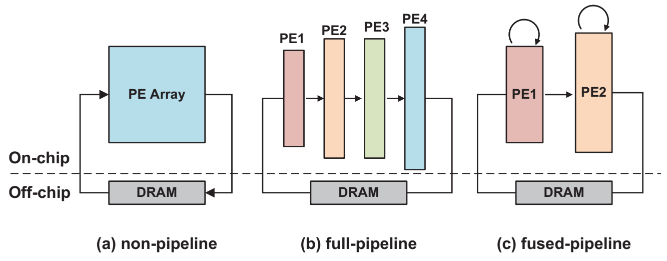

Fig. 1. Three distinct accelerator architectures: (a) A unified accelerator processing all DNN layers, (b) Dedicated accelerators tailored for each individual DNN layer, arranged in a full-pipeline order, (c) Pipelined accelerators where fused groups of layers are mapped onto specific accelerators.

To combinie the advantages and overcome the constraints inherent in non-pipelined and full-pipelined architectures, we develop an adaptable pipeline design termed ”fused-pipeline”, as illustrated in Fig. 1(c). In our design, the layers are grouped and fused to alleviate the resource constraints and improve the pipeline balance, eliminating the need to strictly partition each layer into individual pipeline stages. Furthermore, we employ a double-buffer optimization technique to further reduce the on-chip storage demand. To tackle the enlarged hardware design space, we develop an automated framework, named PipeFuser. PipeFuser incorporates a GA-based search engine for swift deployment of fused-pipeline accelerators. The main contributions of this paper are:

- We propose a novel accelerator architecture, termed fused-pipeline, for DNN accelerator deployment. In this architecture, the layer fusion technique is leveraged to achieve both flexibility and performance gain.
- We present an end-to-end automation framework, named PipeFuser, which can efficiently customize fused-pipeline accelerators from the direct high-level DNN model description.
- We introduce a GA-based co-design engine, which allows quick search to identify optimal layer fusion and mapping strategies for fused-pipeline DNN accelerators.
- We evaluate the performance of PipeFuser against other state-of-the-art DNN accelerators. Experimental results demonstrate that the fused-pipeline architecture outperforms the non-pipeline design with a 2.3-3.3× performance gain. Compared with the full-pipeline design, our solution achieves a 1.9-2.5× performance speedup while retaining flexibility. Additionally, the fused-pipeline architecture demonstrates superior DSP density than other architectures.

* Corresponding author

II. Background and Motivation

In this section, we first provide the layer fusion concept as well as its potential in realizing the fused-pipeline architecture. Then, we outline some design challenges posed by the fused-pipeline design.

A. Layer Fusion Technique

Layer fusion is a technique commonly used in non-pipeline accelerator optimization [10], which combines multiple layers into a single group. By employing this technique, the intermediate data transfers between on-chip and off-chip memory can be reduced, thereby decreasing latency and energy consumption from unnecessary data movement. Fig. 2 illustrates how three distinct layers are mapped onto a single accelerator during layer fusion. Within the fusion group, the intermediate output from layer 1(or 2) is directly fed into layer 2 (or 3). Since all the computations are performed on-chip, the external DDR memory access is eliminated. Additionally, the layer fusion technique has also been utilized in some pipeline accelerators to reduce the on-chip memory requirement [2], [11]. However, these implementations only fuse a limited number of layers without fully exploring the wide fusion space. In addition, these works typically overlook the effect of layer fusion on maitaining pipeline balance .

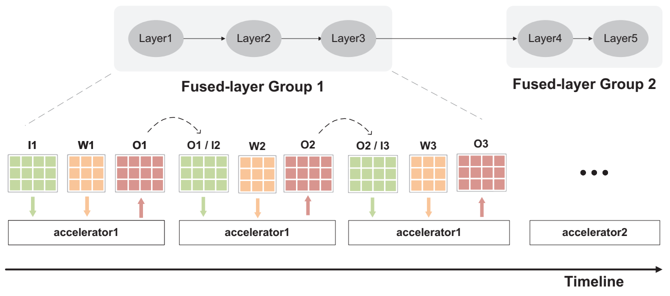

Fig. 2. In the fused-pipeline implementation, multiple layers are fused as a group and mapped to a generic accelerator.

B. More Efficient Pipeline

Fig. 3 shows the roofline model introduced in [14], which reveals the performance potential of a system by evaluating peak computational throughput against memory bandwidth. The compute-to-communication (CTC) ratio in the roofline model quantifies the number of multiply-accumulate (MAC) operations per byte transferred when an application runs on specific hardware. When the CTC ratio is insufficient, a program becomes memory-bound. As a result, the computational resource is underutilized as the performance is hindered by the diagonal roofline. The CTC ratio can be leveraged to reflect the strengths and weaknesses of DNN accelerator designs. We will use this concept to demonstrate the advantages of the fused-pipeline architecture.

In the roofline model, both fused-pipeline and full-pipeline transfer intermediate data through on-chip memory without additional off-chip access. Consequently, their performance is computation-bound. Given identical constraints, however, fused-pipeline and full-pipeline occupy different positions, where fused-pipeline achieves a higher performance. Through resource sharing between layers, fused-pipeline releases more

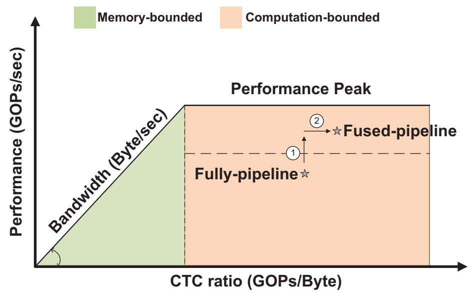

Fig. 3. An performance comparison of full-pipeline and fused-pipeline architectures using the roofline model.

hardware resources than full-pipeline. This allows an accelerator to incorporate more processing elements (PEs) for performance improvement, as indicated by direction 1. Meanwhile, preserving memory resources diminishes bandwidth restrictions, which enhances performance towards direction 2. To conclude, fused-pipeline architecture stands out to be more resource-efficient and performance-optimized than its counterparts.

C. Design Challenges

Challenge 1: How to formulate an efficient fusion strategy. A suitable fusion strategy is crucial in achieving an optimized accelerator design. Determining the number of fusion groups and selecting the layers to be fused are critical decisions. An ideal fusion strategy can balance the pipeline to maximize throughput, while maintaining high resource utilization and CTC ratio. However, identifying the optimal fusion strategy is challenging, due to a large number of fusion combinations, even for modestly-sized networks.

Challenge 2: How to explore the expansive mapping space. It is also essential to define the customization parameters for accelerators. This task encompasses reasonable resource allocation similar to optimization techniques in conventional DNN accelerator designs. Given that different layers in a fused group are handled by a generic accelerator, optimized parameters become pivotal for enhancing resource utilization. When combined with the co-optimization from Challenge 1, parameter optimization results in a broad exploration space, which makes exhaustive search infeasible.

III. Framework Overview

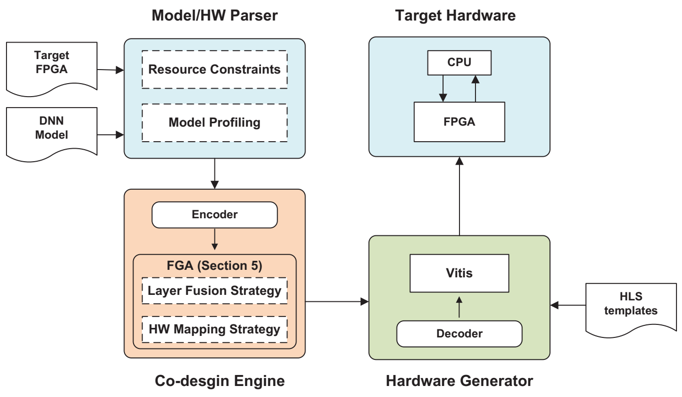

Fig. 4. An overview of PipeFuser Framework.

To enable rapid deployment of the fused-pipeline accelerators, we propose an automation framework named PipeFuser.

PipeFuser comprises three main components: Model/HW Parser, Co-design Engine, and Hardware Generator.

Model/HW Parser. PipeFuser accepts the DNN definition files and trained parameters as inputs. During this stage, detailed layer information, such as layer type, configuration, computation and memory demands and quantization scheme, is extracted. The target FPGA platform's specifications including available DSPs, BRAMs, and external memory bandwidth, are also taken into account.

Co-design Engine. Central to PipeFuser, the co-design engine focuses on determining the optimal layer fusion and hardware mapping scheme, ensuring performance targets are met within hardware constraints. The engine is based on GA, and the parsed model information is encoded into a genome representation. In-depth discussion of the co-design engine can be found in Section V.

Hardware Generator. The configuration results from the co-design are used to generate fused-pipeline accelerators using our High Level Synthesis (HLS)-based operator library. Here, we configure computational and memory resources to each sub-accelerator and lay out the execution details for layer fusion. We then set up the connectivity parameters for the accelerator. The generated synthesizable C-level descriptions are passed to Vitis for deployment to the target FPGA.

IV. Architecture

A. Overview

The fused-pipeline DNN accelerator generated by PipeFuser adopts a fine-grained pipelined structure. As shown in Fig. 5, every pipeline stage corresponds to a group of fused layers. Each accelerator fetches data from its predecessor via double-buffers. Computation is deferred until the preceding accelerator finishes its layer group. There is no resource overlap between the groups, promoting throughput and efficient resource utilization. The accelerator comprises three main components: computation, memory, and control units, as will be detailed below.

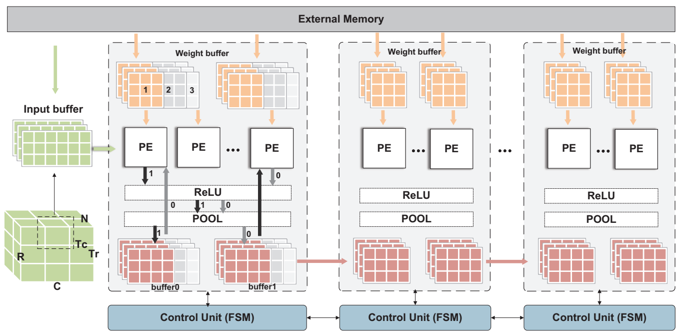

Fig. 5. Fused-pipeline accelerator architecture.

B. Microarchitecture

Computation. We follow the design principles described in the state-of-the-art HLS convolution layer accelerator design in [19]. The convolution operation is performed by PEs. Each PE contains a tree-based MAC unit taking input vectors sized T_n. The accelerator configures T_m PEs in parallel, facilitating concurrent computations. Here, T_n and T_m correspond to the degree of parallelism in processing input and output channels. The products in each PE are passed to a parallel adder tree of depth $\log_2(d)$ to calculate partial sums. The parallelism in the input channel is doubled since we use 8bit data precision and only require half a DSP unit. Pooling and activation operations are basic vector functions that do not consume DSP resources, functioning as independent modules. Therefore, for a single group accelerator, arbitrary computational operations in the model can be accomplished.

Memory. We use double-buffer to optimize memory access. The usage of double-buffer not only reduces data waiting time but also bolsters the layer-fusion computational mode. As illustrated in Fig 5, when buffer0 is used as input, the computational result is saved in buffer1. Conversely, executing the subsequent layer treats buffer1 as the input, with the result stored in buffer0. The size of the double-buffer corresponds to the maximum output data size within the fusion group. The layer fusion computation alleviates the on-chip storage demands because storage is allocated solely for the fusion group instead of for each layer. Consequently, the storage size does not grow linearly with the number of model layers. Meanwhile, off-chip DDR memory holds all the input data and model parameters. Each DDR access fetches input data sized $T_r \times T_c$, where T_r and T_c represent the height and width of the tiles extracted from the output feature maps.

Control Unit. The control module operates under a *finite state machine (FSM)* that oversees computational count and data access positioning. Each fusion group has a separate FSM to control the execution sequence. Unlike the full-pipeline design with centralized control, the fused-pipeline employs distributed control, thus reducing control complexity and improving timing performance.

C. Computation Flow

Fig. 6 depicts the computational flow of the architecture. We divide the input into multiple tiles to reduce on-chip memory overhead, and assume that the data moving overhead is eliminated by double-buffers. Each accelerator processes one tile of every layer in a fusion group at a time. As illustrated, accelerator 1 sequentially executes the first tile for layers 1, 2, and 3. Upon completing a tile, accelerator 1 passes the output to accelerator 2, which commences computations for the second fusion group, while accelerator 1 shifts to compute the next tile of its assigned fusion group. This sequential execution avoids resource sharing between accelerators, expediting group processing and bolstering throughput.

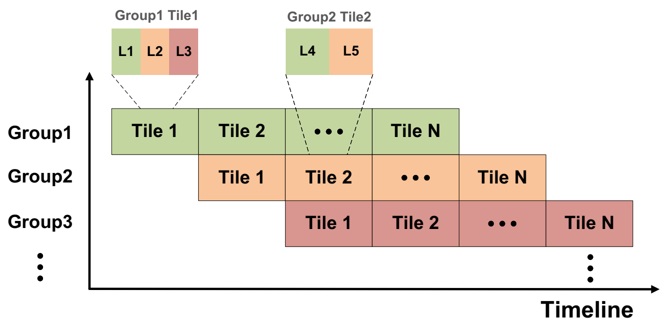

Fig. 6. Computation flow of the fused-pipeline architecture.

V. Co-design Engine

Based on the fused-pipeline architecture outlined in sections III and IV, we propose a GA-based co-design engine that swiftly pinpoints the optimal fusion strategy and hardware mapping choice amidst an expansive search space. The GA is one of the most widely adopted algorithms for solving combinatorial optimization problems due to its simplicity and lightness [9].

A. Encoding

To optimize our fused-pipeline architecture, we design a co-design algorithm to explore fusion and mapping space, as detailed in Algorithm 1. As shown in Fig. 7, our initial step involves encoding the template parameters into a genome with dimensions $(5, N)$. Here, N represents the total number of fused layer groups, matching the accelerator count, n_{acc}. Each group possesses five customizable parameters, namely T_r, T_c, T_m, T_n, and G, respectively. Among them, G signifies the layers fused within a given group, T_r and T_c represent the dimensions of the tile size, and T_m and T_n denote the parallelism dimensions. Since n_{acc} affects the dimension of the solution, it is not suitable to be computed by the GA. Therefore, we traverse n_{acc} in a certain range to explore the optimal number of accelerators.

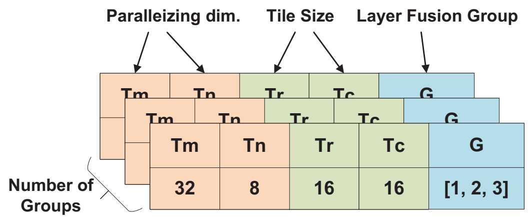

Fig. 7. Encoding design space parameters into genome form.

B. Algorithm Flow

Initialization. The initial population X of T individuals is generated by randomly initializing each variable within its bounds. Each group is described by five parameters: T_r, T_c, T_m, T_n and G. Note that, T_r and T_c are global parameters across different layers. Due to interlayer dependencies, only one set of T_r and T_c is required, and others are backward calculated. For G, layers are divided based on n_{acc} to determine fused groups.

Crossover. Crossover swaps parameters between two sets of solutions with a certain probability to leverage beneficial genes, thus forming a children's subset referred to as "offspring". Two crossovers are involved. The first is the crossover of global parameters T_r, T_c by directly swapping them between two sets of solutions. The second is to perform a crossover for T_m, T_n. For T_m and T_n, swaps occur only within the same fusion group due to their association with the layer fusion scheme. After randomly selecting genomes for crossover, a validation check ensures that resulting offspring adhere to device constraints.

Mutation. Crossover alone does not yield offspring that outperform their parents. We also introduce mutation to enhance offspring performance and prevent early convergence. Through the mutation operator, we modify one or more genes to increase population variability. Our GA features three

Algorithm 1: Explore Fusion and Mapping Spaces

```
Data: the probability of the crossover operation $P_c$,
      the probability of the mutation operation $P_m$,
      the number of the solutions within the
      population $T$, the number of generation $G$, the
      total layer of the net $N_{layer}$
Result: optimal number of accelerator $n_{acc}$, optimal
        solution $S$
1  while $n_{acc} <= N_{layer}$ do
2      initialize population X of T individuals;
3      while $g <= G$ do
4          evaluate fitness of each individual;
5          select parents for next generation;
6          if $random(0,1) < P_c$ then
7              create offspring through crossover;
8              exclude ineligible offspring;
9          end
10         if $random(0,1) < P_m$ then
11             create offspring through mutation;
12             exclude ineligible offspring;
13         end
14     end
15 end
```

mutation types: Mutation-Parallel-Dim, Mutation-Tile Size, and Mutation-Group. For Mutation-Parallel-Dim, we mutate the parallelism dimension by randomly generating a set of T_m and T_n. For Mutation-Tile Size, we randomly select paired genes and assign them a new random tile size. For Mutation-Group, we slightly alter the layer fusion results to prevent non-convergence of the outcomes. A validity check is also required for mutated offspring.

Evaluation and Selection. Evaluation and selection is a crucial process of the GA. This process iteratively refines the solution population until an optimal solution is discovered. Existing literature presents a variety of selection schemes for genetic algorithms. After crossover and mutation, we evaluate the population using tournament selection. Each individual in the population is randomly paired with another. Individuals are compared pairwise. Offspring with high fitness levels are selected for the next round, while others are removed from further consideration.

VI. Experimental Results

A. Experimental Setup

Platform. To assess the performance of fused-pipeline accelerators generated by PipeFuser, we utilize the Xilinx Alveo U200 FPGA hosted on an Intel Xeon Server CPU (Gold 5218R@2.10GHz). The designs are synthesized using the Vitis 2022.1 unified platform.

Hardware Baseline. To evaluate the fused-pipeline architecture, we compare our design with several state-of-the-art FPGA-based accelerators encompassing different architectures: (1) non-pipeline: HybridDNN [15], which mixes the winograd algorithm and FlexCNN [3] based systolic array; (2) full-pipeline: DNNbuilder [21] and TGPA [13]; (3) hybrid architecture: DNNExplorer [20] that adopts a full-pipeline for

TABLE I
Performance Comparison with Previous Works

Design	HybridDNN [15]	FlexCNN [3]	DNNBuilder [21]	TGPA [13]	DNNExplorer [20]	DeepBurning-SEG [4]	Ours
Device	VU9P	U250	KU115	VU9P	KU115	KU115	U200
Model	VGG16	VGG16	VGG16	VGG19	VGG16	VGG16	VGG16
Precision	12bit	16bit (8bit)	16bit (8bit)	16bit	16bit	8bit	8bit
Freq.(MHz)	167	241	235	210	200	235	250
DSPs	5136 (75.9%)	4736 (37.98%)	4318 (78%)	4096 (60%)	4444 (80.5%)	5128 (92.9%)	5026 (73.5%)
BRAM	NA	2692 (45.93%)	1578 (81%)	1690 (78%)	1648 (76.3%)	1486 (76.9%)	1310 (81.8%)
Throughput (GOPS)	3376	1543 (2329)	2011 (4022)	1510	1702	4778	**7684**
DSP Density (GOPS / DSPs)	0.66	0.33	0.93	0.37	0.38	0.93	**1.52**

TABLE II
Scalability Evaluation

Model	VGG11	VGG13	VGG16	VGG19	ResNet50	ResNet152
Group.number	3	3	4	5	4	6
Freq.(MHz)	250	250	250	235	220	220
DSPs	3982	4390	5026	5621	3560	4496
BRAM	982	1073	1310	1582	1244	1543
Throughput (GOPS)	3560	5231	7684	8431	2106	5415

initial layers and non-pipeline for the latter ones. (4) segment-grained pipeline: Deepburning-SEG [4], featuring dynamic scheduling of Processing Units to perform computation.

DNN Workloads. For DNN workloads, we consider two widely adopted DNN models: VGGNet [12] and ResNet [8]. We perform DNN inference using 8-bit [6] fixed-point data representation. We assume the fused-pipeline execute a single CNN inference task with streaming input data (e.g., video surveillance system).

B. Comparison with Previous Implementations

Throughput. For a fair comparison, we double the throughput of 16-bit designs before comparing them with our accelerator. Table I shows that fused-pipeline architecture outperforms HybridDNN and FlexCNN by 1.6-2.3×. Compared to DNNBuilder and TGPA, we observe a 1.35-1.8× improvement. Although theoretical predictions might favor full-pipeline designs, real-world DNN characteristics and hardware limitations can compromise performance. In full-pipeline accelerators, the FC layer consumes major bandwidth, resulting in an unbalanced pipeline. In contrast, the fused-pipeline accelerator can fuse the convolutional and FC layers to allocate more bandwidth resources and ensure pipeline balance. Moreover, our fused-pipeline uses distributed control for better timing and achieves performance gains over full-pipeline. Compared with Deepburning-SEG, which utilizes out-of-order processing unit execution, our approach prioritizes sequential execution that rapidly frees hardware resources, thus ensuring enhanced throughput.

DSP Density. DSP density, defined as the computation-to-resource use ratio, is a key indicator of hardware efficiency in FPGA designs. The metric indicates whether a hardware design suffers from resource underutilization. Given equal throughput, an accelerator with less DSP usage offers better computational density. Our fused-pipeline architecture employs double buffering to reduce communication latency and minimizes DSP data wait times. In addition, our strategic layer fusion scheme ensures the optimal DSP utilization. Consequently, our fused-pipeline achieves the highest DSP density compared to alternative architectures.

C. Without Layer Fusion

To further demonstrate that layer fusion improves pipeline balance, we compare it against a conventional layer-by-layer full-pipeline accelerator using the same template [19]. We still optimize the hardware parameters of the full-pipeline accelerator using the GA. Specifically, for the full-pipeline design, the number of pipeline stages is equal to the number of model layers. This allows us to demonstrate the effectiveness of the fused-pipeline approach. Note that, the tile size remains consistent across both accelerators. As Fig. 8 demonstrates, using VGG16 as a benchmark, our accelerator outperforms the full-pipeline in throughput. Furthermore, our design is more BRAM-efficient for the same input dimensions, owing to the use of shared intermediate memory that reduces BRAM consumption.

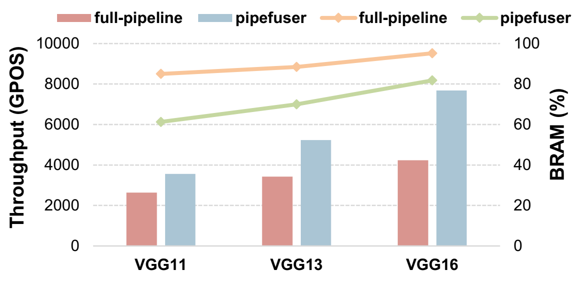

Fig. 8. Performance comparison of full-pipeline and fused-pipeline designs using the same template.

D. Scalability Evaluation

With Increasing Layer Depth. Compared to other accelerator designs, PipeFuser remains scalable with rising DNN model layer counts. We test PipeFuser on models with various layer depths and find that it consistently generates optimized accelerators adapting to the model's layer depth while ensuring efficient resource allocation. The scalability of PipeFuser is gauged through two strategies: 1) Comparing accelerator generated by PipeFuser against those without layer fusion. 2) Modulating the layer count, deploying configurations including VGG13, VGG16, and VGG19. This demonstrates PipeFuser's capacity for architecture optimization across an array of models. Table II shows the results of these scalability evaluations.

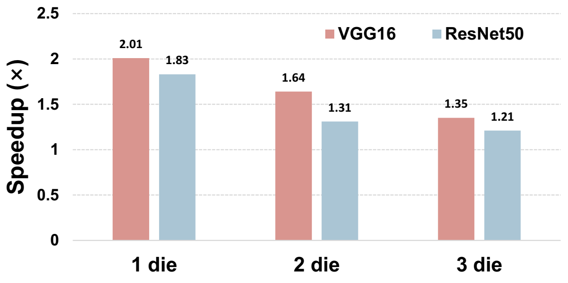

Fig. 9. Performance evaluation of PipeFuser with limited resources.

With Limited Resources. To further demonstrate the flexibility of the fused-pipeline, we limit the available resources. For test purpose, we use the U200 board, a multi-die FPGA platform that can be viewed as an integration of three smaller FPGAs. We evaluate both the fused-pipeline and full-pipeline under diverse die numbers using the same hardware template. As shown in Fig. 9, under the constraint of a single die, the fused-pipeline's layer fusion strategy frees more resources, yielding larger performance gains. As available resources increase, full-pipeline can dedicate more computational units to each layer, thereby mitigating the latency of bottleneck layers. Consequently, the speedup advantage decreases. In conclusion, the fused-pipeline architecture demonstrates superior flexibility compared to the full-pipeline, adeptly catering to the deployment specifications of DNN accelerator under various resource constraints.

VII. Conclusion

To address the limitations of existing DNN accelerators based on non-pipeline or full-pipeline architectures, we develop a novel fused-pipeline architecture that leverages layer fusion to enhance flexibility and performance. We also create an automated end-to-end design framework, named PipeFuser, to efficiently construct DNN accelerators. Experimental results validate the advantages of our proposed approach. Compared to non-pipeline designs, the fused-pipeline architecture achieves 2.3-3.3× higher performance. Additionally, our architecture provides 1.9-2.5× speedup over full-pipeline implementations while offering greater flexibility to accommodate diverse DNN workloads and hardware constraints. The fused-pipeline also attains higher DSP density than alternative architectures.

Acknowledgments

This work was financially supported in part by National Key Research and Development Program of China under Grant 2021YFA1003602, and in part by Shanghai Pujiang Program under Grant 22PJD003.

References

[1] T. Alonso and K. Vissers, "Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 2, pp. 1–34, 2021.

[2] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[3] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo, and J. Cong, "Flexcnn: An end-to-end framework for composing cnn accelerators on fpga," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 16, no. 2, pp. 1–32, 2023.

[4] X. Cai, Y. Wang, X. Ma, Y. Han, and L. Zhang, "Deepburning-seg: Generating dnn accelerators of segment-grained pipeline architecture," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1396–1413.

[5] R. Chen, H. Zhang, S. Li, E. Tang, J. Yu, and K. Wang, "Graph-opu: A highly integrated fpga-based overlay processor for graph neural networks," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE Computer Society, 2023, pp. 228–234.

[6] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep learning with int8 optimization on xilinx devices," *White Paper*, 2016.

[7] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 769–774.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[9] S.-C. Kao and T. Krishna, "Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[10] L. Mei, K. Goetschalckx, A. Symons, and M. Verhelst, "Defines: Enabling fast exploration of the depth-first scheduling space for dnn accelerators through analytical modeling," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 570–583.

[11] J. Shen, Y. Huang, M. Wen, and C. Zhang, "Toward an efficient deep pipelined template-based architecture for accelerating the entire 2-d and 3-d cnns on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 7, pp. 1442–1455, 2019.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[13] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, "Tgpa: Tile-grained pipeline architecture for low latency cnn inference," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 2018, pp. 1–8.

[14] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[15] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "Hybriddnn: A framework for high-performance hybrid dnn accelerator design and implementation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[16] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "Opu: An fpga-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 35–47, 2019.

[17] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 122–132.

[18] Y. Yu, T. Zhao, M. Wang, K. Wang, and L. He, "Uni-opu: An fpga-based uniform accelerator for convolutional and transposed convolutional networks," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 28, no. 7, pp. 1545–1556, 2020.

[19] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[20] X. Zhang 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.

[21] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: An automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. ACM, 2018, pp. 1–8.