TGPA: Tile-Grained Pipeline Architecture for Low Latency CNN Inference

Xuechao Wei[1,3,*], Yun Liang[1], Xiuhong Li[1], Cody Hao Yu[2,3,*], Peng Zhang[3] and Jason Cong[1,2,3,†]

[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,lixiuhong}@pku.edu.cn,{hyu,cong}@cs.ucla.edu,pengzhang@falcon-computing.com

ABSTRACT

FPGAs are more and more widely used as reconfigurable hardware accelerators for applications leveraging convolutional neural networks (CNNs) in recent years. Previous designs normally adopt a uniform accelerator architecture that processes all layers of a given CNN model one after another. This homogeneous design methodology usually has dynamic resource underutilization issue due to the tensor shape diversity of different layers. As a result, designs equipped with heterogeneous accelerators specific for different layers were proposed to resolve this issue. However, existing heterogeneous designs sacrifice latency for throughput by concurrent execution of multiple input images on different accelerators. In this paper, we propose an architecture named Tile-Grained Pipeline Architecture (TGPA) for low latency CNN inference. TGPA adopts a heterogeneous design which supports pipelining execution of multiple tiles within a single input image on multiple heterogeneous accelerators. The accelerators are partitioned onto different FPGA dies to guarantee high frequency. A partition strategy is designd to maximize on-chip resource utilization. Experiment results show that TGPA designs for different CNN models achieve up to 40% performance improvement than homogeneous designs, and 3X latency reduction over state-of-the-art designs.

1 INTRODUCTION

CNNs (Convolutional Neural Networks) are compute-intensive learning models with growing applicability in a wide range of domains in recent years. With the trend of CNN development for higher accuracy, CNN models are becoming much deeper and more complex in terms of the number of layers and data dependencies [5, 7, 10, 20]. Considering the diversity of CNN models and high inference accuracy using low bit-width data types, FPGAs have become a particularly attractive accelerator option due to their high performance, energy efficiency and high reconfigurability when compared to GPUs and ASICs. By integrating multiple dies, the latest FPGAs incorporate much more resources including high volume of DSPs, on-chip memory and off-chip bandwidth than previous generations, which provides more design spaces and chances to further boost CNN performance.

Previous CNN accelerator designs [2, 15, 23, 27, 29] adopt a homogeneous design methodology that uses a uniform accelerator architecture and executes all layers of a model on the same hardware sequentially. All resources contribute to the execution of each individual layer of a given model. So the optimization target of homogeneous designs is the **latency** of a single input image or a small batch. However, different layers in a CNN model have different input data shapes in terms of the number of input or output channels, feature map row and column, and kernel sizes. As a result, the homogeneous design may cause dynamic resource inefficiency for some layers. In order to solve this problem, a heterogeneous design methodology is proposed in several previous works [12, 19, 21, 28]. A heterogeneous design incorporates multiple convolutional layer accelerators on a single FPGA. Each of them is optimized specifically for one or a set of layers that are executed on it. In contrary to the homogeneous design, a heterogeneous design mainly aims to optimize **throughput** rather than latency by concurrently executing multiple input images in pipelining on different accelerators. However, from the perspective of a single image processing, different layers still execute in sequential order on different accelerators with less resource consumption compared with a homogeneous design, resulting in long latency. Thus the existing heterogeneous designs cannot meet the latency requirement well despite they have ideal resource efficiency for the services where demand for low response time is much tighter than throughput [9]. Moreover, the latest FPGAs contain more on-chip resources than previous generations by integrating multiple dies. But crossing-die paths would have long delay and then cause frequency degradation [25]. In addition, each die has its own resource volume. As a result, when designing a heterogeneous architecture for a multi-die FPGA, crossing-die timing critical paths should be limited as much as possible. Placement of multiple accelerators on multiple dies must be considered in order to achieve resource utilization balancing.

In this paper we propose a heterogeneous design named tile-grained pipeline architecture (TGPA) for low latency inference of CNN models. TGPA benefits from three features in terms of latency reduction. First, TGPA has high efficiency of both arithmetic units and on-chip memory in terms of layer specific structures for different accelerators. Second, to save off-chip memory data transferring, TGPA has on-chip buffers external to accelerators in order to support pipelining execution of multiple accelerators at the granularity of tile. Apart from the external buffers, within each accelerator, internal buffers are tightly coupled with computation units for data reuse within a tile. Third, TGPA has higher frequency than homogeneous designs by delicate placement of multiple accelerators that constrains timing critical paths within dies. With the decoupled buffer design, timing critical datapaths between computation units and data communication with local buffers could be constrained within a single die. In order to generate a TGPA design with minimal latency for a given CNN model, we design analytical models and an algorithm to make the following two decisions. It not

*Work done during internship at Falcon Computing Solutions, Inc.

†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.

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00

https://doi.org/10.1145/3240765.3240856

only determines the number of accelerators, the structure for each accelerator and to which die each accelerator should be assigned. But also it determines the amounts of on-chip memory portions for internal buffers and external buffers and interfaces for each accelerator. Finally, we implement an automation flow to perform TGPA generation from high-level model descriptions to FPGA bitstreams so that underlying hardware considerations are unnecessary for end users. In summary, our contributions are three-folds as follows.

- **A low latency CNN accelerator design.** We propose a heterogeneous accelerator architecture for low latency CNN inference. It has a higher resource efficiency than previous designs, and is easier to achieve higher operating frequency.
- **An algorithm to partition CNN model graphs and map accelerators on multiple FPGA dies.** The algorithm is efficient to determine the number of accelerators and their assignment on multiple dies, as well as on-chip memory allocation for each accelerator.
- **An end-to-end automation flow.** We also design a push-button end-to-end automation flow to generate the accelerator from high level CNN model descriptions.

Experiment results show that the TGPA is able to achieve up to 90% of the resource efficiency for different CNN models, and the end-to-end performance with our generated designs are able to achieve up to 1.5 Tops with 210 MHz frequency on Amazon EC2 F1 instances.

2 BACKGROUND AND MOTIVATION

2.1 State-of-the-art CNN Accelerator Designs

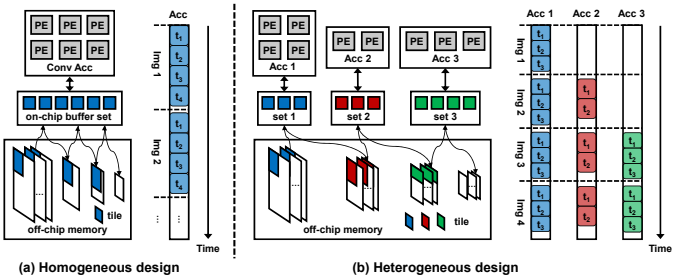


Figure 1: State-of-the-art CNN accelerator designs

A typical CNN model consists of a number of consecutive layers that work in a feedforward process for inference. Modern CNN models are becoming more and more complex in terms of network topology and layer shape diversity. The state-of-the-art CNN accelerator designs on FPGA could be divided into two categories. The first design methodology uses one uniform accelerator architecture to process all layers of a model in order. We name this design as *homogeneous design* which has been used in many prior works [2, 15, 23, 27, 29]. As shown in Fig. 1(a), the homogeneous design takes one input image at a time for processing by all layers. After the inference of the current image is finished, the next image is fetched for processing, and so on. For a single layer's inference, due to the limited FPGA resources compared with the huge volume of computation and bandwidth requirement of CNN models, a homogeneous design firstly divides an input feature map into tiles. Then it repeatedly loads the tiles one after another from off-chip memory to on-chip memory, and then processes the tiles in sequence by a jointly optimized accelerator design for all convolutional layers. All resources on the FPGA board are used to process each layer. Hence the homogeneous designs are helpful for latency reduction. As different layers have various dimensions of input/output feature maps, kernel sizes and feature map height and width, the same tile size of homogeneous design will lead to dynamic resource inefficiency for some layers. Even though, currently dynamic reconfiguring FPGA for different layers' processing is not considered. Because the reconfiguration overhead is not negligible compared with the total inference latency. We make statistics on the DSP efficiency for the first 50 layers from three latest CNN models. The results are shown in Fig. 2. We could see obvious resource efficiency diversities from the three models, resulting in average 44%, 81% and 59% efficiency.

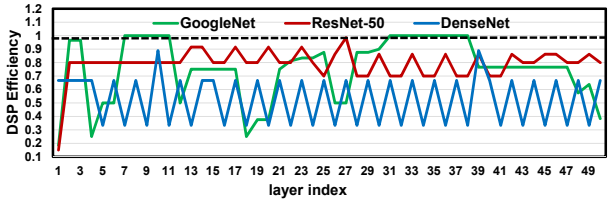


Figure 2: Arithmetic unit utilization

An effective way to alleviate the dynamic resource inefficiency problem is to design multiple accelerators, each of which is specifically optimized for each individual convolution layer, in order to maximize resource efficiency on a single FPGA board. This design methodology is referred to as *heterogeneous design* which is used in [12, 19, 21]. In a heterogeneous design, the accelerators have different resource efficiency from each other in terms of the number and size of process elements, on-chip buffer size, and so on. As shown in Fig. 1(b), due to the data dependency between consecutive layers, one intuitive way to maximize resource efficiency of all accelerators is to concurrently execute different input images in pipelining on different accelerators designed for different layers. Although the concurrent execution of multiple accelerators could maximize resource efficiency and improve performance compared to the homogeneous designs, this improvement mainly contributes to throughput improvement of multiple input images at the cost of latency of single input image. Because during the execution of each layer, the resources are only from one accelerator rather than the whole FPGA board.

2.2 State-of-the-art FPGA Architecture

The state-of-the-art FPGAs claim to have a peak performance of over 1 TFlops with thousands of Digital Signal Processing (DSP) blocks. Limited by the circuitry overhead, the new generations of FPGAs are manufactured by integrating multiple dies mounted on a passive silicon interposer, rather than simply increasing transistor density and chip size [25]. Apart from abundant logic and DSP resources as well as traditional on-chip memory resources such as LUTRAM, Block RAM etc., these FPGAs contain a new memory block named UltraRAM that enables up to hundreds of megabits total on-chip storage, equating to a 4X increase in on-chip compared with last generation of Xilinx FPGAs. In addition, these latest FPGAs are usually equipped with multiple DDR banks. The FPGA vendor synthesis tool chain allows monolithic designs that consume resources on multiple dies, automatically performing crossing-die placement and routing [25]. But there still exist signals that routes are unable to propagate, such as carry chains, DSP cascades, Block RAM address cascades. Therefore, the number of timing critical paths that must cross dies should be limited as much as possible in order to satisfy timing closure.

3 TILE-GRAINED PIPELINE ARCHITECTURE

In this section, we present an accelerator architecture for low latency CNN inference. It has the following two benefits.

Heterogeneity. Our architecture adopts the heterogeneous design. It has the potential to improve on-chip resource efficiency by layer

specific designing. Moreover, the concurrent execution of the multiple accelerators comes from pipelining of different tiles within a single input image rather than that of different input images. Thus the latency of the single image inference could be reduced.

Frequency. The architecture adopts systolic array to do convolution operations because 2-D topology of systolic array matches the 2-D structure in the FPGA layout well. On the other hand, to avoid the internal timing critical datapaths between PEs and data accesses to internal buffers to be placed across multiple dies, we constrain each accelerator within a single FPGA die. These two strategies make the architecture satisfy the timing closure more easily.

3.1 Architecture Design Overview

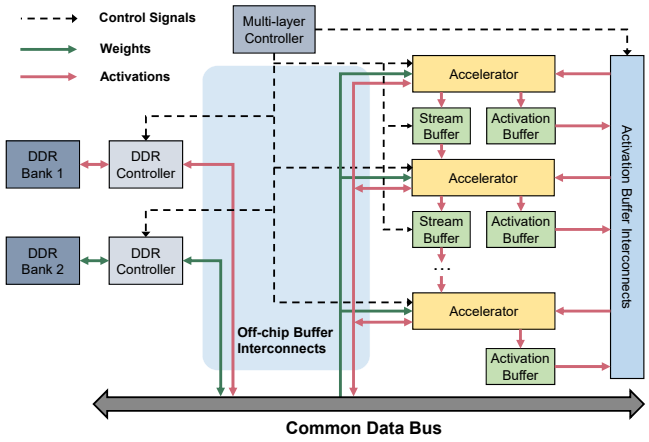


Figure 3: Architecture overview

The architecture overview is shown in Fig. 3. There are multiple convolutional layer accelerators that are designed specifically for layers executing on each of them. Each accelerator processes convolutions and other operations attached with each convolution such as relu, pooling, etc. Accelerators are connected by stream buffers to enable pipelining execution of consecutive layers in a given CNN model. Unlike existing heterogeneous designs that take image as pipeline granularity, our architecture allows the current layer to begin its computation once enough input data, referred as a tile, has been buffered. Different accelerators process tiles from different layers in parallel. We name this execution model as tile-grained pipelining, and the architecture is named tile-grained pipeline architecture (TGPA).

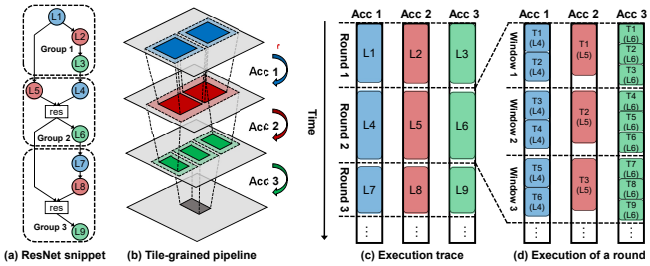


Figure 4: Tile-grained pipelining paradigm

We illustrate how tile-grained pipelining works in Fig. 4 taking a model snippet from ResNet [7] as an example. As shown in Fig. 4(a), the snippet contains 9 convolutional layers indexed in topological order. We suppose TGPA has 3 accelerators, and then 3 consecutive convolutional layers execute concurrently in tile-grained pipelining. As shown in Fig. 4(b), each accelerator takes a number of tiles from its previous accelerator as input, and produces an output tile for the next accelerator. The tile size each accelerator processes may be different from each other. For example, in Fig. 4(b), the second accelerator takes 2 tiles from the first accelerator to form one of its input tiles, and the third accelerator splits an output tile from the second accelerator into three tiles for its processing. In general, we partition a CNN model graph into groups as shown in Fig. 4(a). Each group contains the same number of consecutive convolutional layers equal to the number of accelerators. As a result, layers with indexes 1, 4, 7, $\cdots$ are mapped on the first accelerator, and layers 2, 5, 10, $\cdots$ are mapped on the second accelerator, and so on. Layers from different groups execute sequentially and the execution of a group is called a *round* as shown in Fig. 4(c). When each round ends, there is a synchronization to wait until all output data are accumulated from the last layer in the current group. Then the next round begins its execution. The pipelining execution of a round is exemplified in Fig. 4(d) for round 2 of Fig. 4(c). Within a round, for the least number of tiles in a stream buffer that could enable the launch of an accelerator, we name the execution time of these tiles a tile *window*. The accelerator with the largest tile window would dominate the latency of this round. In Fig. 4(d), the latency of round 2 is determined by the window of the third accelerator. Some of the state-of-the-art CNN networks have complex branches and data dependency between layers. To handle this scenario, apart from stream buffers, we also equip the accelerators with activation buffers to store complete output feature maps for future rounds. Moreover, considering the limited capacity of on-chip memory, we choose to spill some of the activation buffers into off-chip memory once the on-chip memory limitation is exceeded. The activation buffer interconnects and the off-chip buffer interconnects compose the I/Os for all the cross-round activation read and write. The multi-layer controller controls the buffer accesses for each accelerator according to the data dependency of layers mapped on it. In addition, we distribute the off-chip memory accesses for weights and activations onto two DDR banks to improve bandwidth utilization.

3.2 Accelerator Design

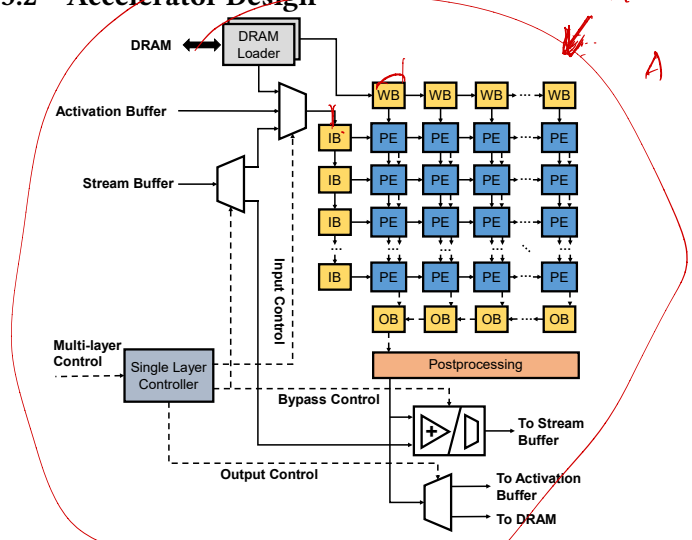


Figure 5: Accelerator design

Fig. 5 shows the accelerator design for processing a single convolutional layer and its auxiliary layers. We adopt a systolic array to perform convolution operations. Systolic array [11] has been proved to be a suitable architecture for CNN acceleration [2, 9, 23].

The systolic array architecture is a specialized form of parallel computing with a deeply pipelined network of PEs. With the regular layout and local communication, the systolic array features low global data transfer and high clock frequency, which is suitable for large-scale parallel design, especially on FPGAs. In addition, there are internal on-chip buffers to store weights, input and output feature maps, in order to keep data reuse within a tile. The postprocessing module is used to perform relu, pooing and zero padding, etc. The single layer controller determines the input and output for the computation part according to the control signal transmitted from the multi-layer controller. It is also responsible for the bypass control which is used to handle the "false" layer dependency in a round that happens when two layers are in topological order but have no data dependency. For example, $L4$ and $L5$ in the second round in Fig. 4 have "false" layer dependency. In this case, to keep the pipeline work correctly, the input control multiplex chooses its input from activation buffer or DRAM. The bypass control multiplex firstly chooses to output the data from the previous stream buffer directly to the next stream buffer, then it subsequently outputs results from PE array. If the current layer needs a residual addition operation, the multiplex will be replaced by an adder to perform the addition before sending the results to the next stream buffer.

3.3 Stream Buffer

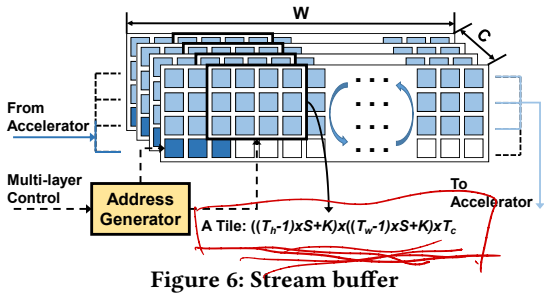


Figure 6: Stream buffer

The stream buffers are used to connect multiple accelerators to support tile-grained pipelining. There are three challenges to enable pipelining at the granularity of tiles: 1) One output tile needs the padding data from its neighboring tiles to form an input tile until it could be sent to the next accelerator. 2) Each tile will be reused by some weights tiles. The space in the buffer for a tile thereby should not be flushed until the reuse on it finishes. 3) Different accelerators may require different tile sizes. Therefore, we adopt the line buffer structure for our stream buffer design. Its design diagram is shown in Fig. 6. Suppose the current convolutional layer has C input feature maps with row and column sizes by $H \times W$. It convolves with M kernels with C channels and the kernel size is $K \times K$. The shifting stride of the kernels is S. We could see that the line buffer is a circular buffer consisting of $K+S$ lines. The size of each line is $W \times C$. At the beginning, the first K rows of input feature maps are loaded to input buffer line 1 to K from the output of the previous accelerator. After that, tiles with padding data in the K rows slide through these lines to be sent to the accelerator and fill the internal buffers. Here the first challenge is solved. Meanwhile, the next S input feature map rows are being transferred to the line buffer. Once the reuse for the tiles and transfer are both done, the next round begins. The second challenge for the concern of reuse is also solved. If we approach to the end of the line buffer, we will restart from the first line. Fig. 6 illustrates this process with $K = 3$ and $S = 1$, and the input tile size is $((T_h-1)\times S+K)\times((T_w-1)\times S+K)\times T_c$ with tiling factors for H, W and C are $T_h = 1$, $T_w = 2$ and $T_c = C$. To handle with the third challenge, we must perform tile reorganization to deal with the tile size mismatching between neighboring accelerators. The address generator in Fig. 6 generates correct addresses for the received data and put them in the line buffers according to the multi-layer control signals.

4 DESIGN OPTIMIZATION

There is a large design space to explore in order to find the TGPA design with minimal latency. It consists of four aspects which interact with each other. First, it is difficult to determine the number of accelerators. Although designing a specific accelerator for each convolutional layer will have the maximal resource efficiency, there are several factors limiting the number of accelerators, including on-chip resources, off-chip memory bandwidth and some logics that keep duplicate as the number of accelerators increases. Second, how to partition resources among accelerators, including DSPs for PE array and internal buffers for data reuse within a tile, and determine the shape of each systolic array is also challenging. Third, determining the memory interface type for each accelerator, and the on-chip memory size for each external buffer is also not trivial. Finally, we must guarantee the maximization and balancing of resource efficiency during the partitioning and mapping of the accelerators onto multiple dies. In this section, we first model the latency of TGPA. Then we design a heuristic algorithm to find the architecture with minimal possible latency.

4.1 TGPA Latency Modelling

The TGPA pipeline in Fig. 3 consists of accelerators and stream buffers, as well as the the on-/off-chip buffer interconnets for intermediate activations accessing and weights accessing. The latency of the stream buffers will be hidden by that of accelerators during the execution of each round due to their line buffer structures. Hence we only model the performance of accelerators in terms of computation and communication. According to the execution model illustrated in Sec. 3, the latency of a TGPA design for executing a CNN model is determined by the number of rounds and the latency for each round. As all the accelerators execute in parallel in a round, the latency of a round is determined by the accelerator with the largest latency for processing all tiles within the current round. We suppose the number of accelerators in the TGPA design is N, and the number of rounds is R. The expression of the latency $\mathcal{L}$ is shown in Eq. 1.

$$\mathcal{L} = \sum_{r=1}^{R} \max_{n=1}^{N} \mathcal{L}_{acc}(r,n),\ \mathcal{L}_{acc}(r,n) = \frac{\mathcal{N}^{acc}_{act\_ops}(r,n)}{\mathcal{T}_{acc}(r,n)}$$
$$\mathcal{N}^{acc}_{act\_ops}(r,n) = \prod_{d \in D_{(r,n)}} \lceil B_{(r,n)}(d)/T_d \rceil \times T_d \tag{1}$$

where $\mathcal{L}_{acc}$ and $\mathcal{T}_{acc}$ are the latency and throughput for an accelerator n in round r. $\mathcal{N}^{acc}_{act_ops}$ is the number of actual operations for processing a given layer. It is obtained by the upper bounds and tiling factors of all dimensions. $B(d)$ returns the upper bound for dimension d. $\mathcal{T}_{acc}$ is computed in Eq. 2.

$$\mathcal{T}_{acc} = \min(\mathcal{T}^{comp}_{acc}, \mathcal{T}^{comm}_{acc})$$
$$\mathcal{T}^{comp}_{acc} = 2 \times \prod_{d} P_d \times F,\ \mathcal{T}^{comm}_{acc} = \min_{p}\{\mathcal{T}^{p}_{comm}\} \tag{2}$$

where $\mathcal{T}^{comp}_{acc}$ and $\mathcal{T}^{comm}_{acc}$ are computation throughput and communication throughput for processing a tile. $\mathcal{T}^{comp}_{acc}$ equals to the number of multiply and accumulations the accelerator could do multiplying the frequency F. P_d denotes the parallelism factor in dimension d, and $\prod_d P_d$ equals to the number of DSPs. $\mathcal{T}^{p}_{comm}$ is the communication throughput at array port p. For a given port, a tile

could be loaded from on-chip buffers (stream buffer or activation buffer), or off-chip buffers as illustrated in Fig. 3.

$$\mathcal{T}^p_{comm} = \begin{cases} \frac{\mathcal{N}^{tile}_{ops} \times F}{\mathcal{S}^p_{tile}/PF}, \text{if } IF(p) = 1 \\ \frac{\mathcal{N}^{tile}_{ops}}{\mathcal{S}^p_{tile}/\mathcal{B}^p_{acc}}, \text{otherwise} \end{cases} \quad \mathcal{N}^{tile}_{ops} = 2 \times \prod_{d \in D} T_d \tag{3}$$

In Eq. 3, $IF(p)$ denotes the memory type for port p. If $IF(p) = 1$, p loads data from on-chip buffer, and PF is the partition factor for the buffer. Otherwise, it loads data from off-chip memory. $\mathcal{B}^p_{acc}$ is the bandwidth obtained by the accelerator. In a single round, the obtained bandwidth by a given accelerator is determined by the percentage of its off-chip memory accesses in all of the simultaneous access requests by all accelerators. The equation to compute $\mathcal{B}^p_{acc}$ is shown in Eq. 4.

$$\mathcal{B}^p_{acc} = \frac{\mathcal{N}^p_{tile} \times \mathcal{S}^p_{tile}}{\sum_p \sum_{n=1}^{N} IF(p) \times \mathcal{N}^p_{tile} \times \mathcal{S}^p_{tile}} \times \mathcal{B}_{bank} \tag{4}$$

where $\mathcal{N}^p_{tile}$ denotes the number of tiles array p will load in a given round, and $\mathcal{S}^p_{tile}$ is the tile size. Both of them could be computed by polyhedral model by parsing the systolic array dataflow defined in [23].

4.2 Optimization

We divide the design space into two parts and design an algorithm involving communication optimization and computation optimization individually. As communication and computation interact with each other, we make the following three heuristic pruning strategies in order to reduce and separate the design spaces while keeping the optimal results as much as possible.

1. Do not apply tiling on feature map height H and kernel size K. Our systolic array implementation has three levels of parallelism, and H has the same size with W for all layers. So we choose to map W on systolic array and keep the stream buffers be the same structure for all accelerators. For K, they are usually small, say, mostly 3 and 1, especially for the latest models.

2. Adopt the parallelism factors as the partition factors for internal buffers. It could guarantee that performance not be bounded by internal buffer access.

3. Keep the accelerators be put in the same ascending order with dies. It helps reduce the crossing-die paths to the greatest extent because any two accelerators will have only one path through a stream buffer. That is, given two accelerators with indexes p and q, supposing the indexes of dies which the two accelerators are put on are i_p and i_q, we define that $i_q \geq i_p$ if $q > p$. With the three pruning strategies, we list the buffer information in Tab. 1 to be referred by our optimization process. Each item in the table quantifies the buffer type, size and number of each buffer, as well as the partition factor.

Table 1: On-chip buffer information

Buffer		Size	Number	Factor
INT	IB	$(T_h + R - 1) \times (\lceil T_w / P_w \rceil + S - 1) \times T_c$	P_w	P_c
	WB	$\lceil T_m / P_m \rceil \times T_c \times R \times S$	P_m	P_c
	OB	$\lceil T_m / P_m \rceil \times T_h \times T_w$	P_m	1
EXT	Str.	$(W + S - 1) \times C$	$T_h + R$	PF
	Act.	$(H + R - 1) \times (W + S - 1) \times C$	1	PF

The algorithm flow is shown in Fig. 7. The input of the algorithm is a directed acyclic graph (DAG) $G(V, E)$ representing a CNN model. The vertices set V represents convolutional layers (merged with the postprocessing layers), and E represents data dependency between layers. We first set a tentative number of accelerators n and an external buffer size constraint Q_{ext} to explore. Then the algorithm goes through two steps to obtain a solution in the current iteration. The first step allocates on-chip memory for external buffers and determines the memory interfaces for all accelerators. After that,

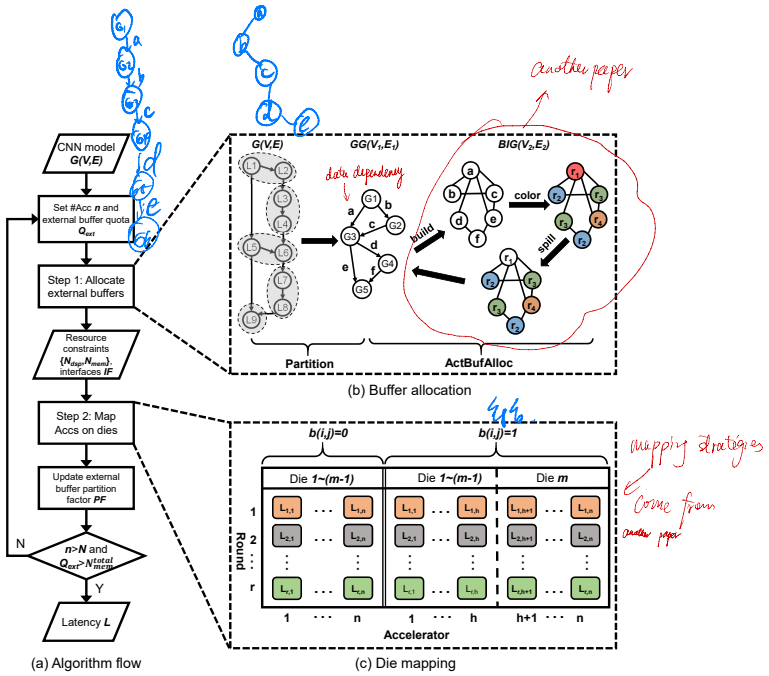


Figure 7: Algorithm flow

Algorithm 1: TGPA optimization algorithm

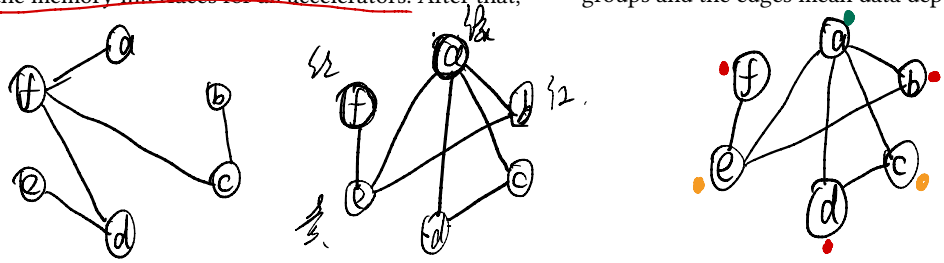
input : A CNN model $G(V, E)$
output: The minimal latency $\mathcal{L}$

1 **for** $e \in E$ **do**
2 $\quad e.size \leftarrow Size_{act}(e.v)$ // $e = \{v, u\}$ // init by its start vertex
3 $Q_{ext} \leftarrow 0$
4 $PF \leftarrow seed$
5 **for** $n \leftarrow 1$ **to** $|V|$ **do**
6 $\quad$ **while** $Q_{ext} < \mathcal{N}^{total}_{mem}$ **do**
7 $\quad\quad$ ***Step 1***: External buffer allocation for $G(V, E)$
8 $\quad\quad GG(V_1, E_1) \leftarrow Partition(G(V, E))$
9 $\quad\quad S_{str} \leftarrow \sum_{v \in V_1} Size_{str}(v)$
10 $\quad\quad Q_{act} \leftarrow Q_{ext} - \mathcal{N}^B_{mem}(S_{str}, 1 + \max\{B_l(r) | l \in L_n\}, PF)$
11 $\quad\quad \{S_{act}, IF\} \leftarrow ActBufAlloc(GG(V_1, E_1), Q_{act})$
12 $\quad\quad$ **for** $i \leftarrow 1$ **to** m **do**
13 $\quad\quad\quad \mathcal{N}_{mem}[i] \leftarrow \mathcal{N}_{mem}[i] + \frac{(Q_{ext} - S_{act})}{m}$
14 $\quad\quad$ ***Step 2***: Map n accelerators onto m dies
15 $\quad\quad \mathcal{L}_n \leftarrow \sum_r \mathcal{L}(r, n, m, \mathcal{N}_{dsp}[m], \mathcal{N}_{mem}[m])$
16 $\quad\quad$ **if** $\mathcal{L}_n < \mathcal{L}$ **then**
17 $\quad\quad\quad \mathcal{L} \leftarrow \mathcal{L}_n$
18 $\quad\quad Q_{ext} \leftarrow Q_{ext} + \Delta Q$
19 $\quad\quad PF \leftarrow (\sum_{i=1}^{n} P_w(i) \times P_c(i))/n$

with the remaining on-chip memories and DSPs, the second step partitions and maps the accelerators onto different dies. Latency $\mathcal{L}$ and partition factor of external buffers PF will be updated before the next trial.

$$\mathcal{L}(r, i, j, dj, bj) = \begin{cases} \mathcal{L}(r, i, j-1, \mathcal{N}_{dsp}[j-1], \mathcal{N}_{mem}[j-1]), \text{ if } d(i, j) = 0; \\ \min_{(x,y)}\{\max\{\mathcal{L}(r, i-1, j, dj-x, bj-y), \mathcal{L}_{acc}(r, i)\}\}, \\ \text{otherwise.} \end{cases} \tag{5}$$

The algorithm details are shown in Alg. 1. Each vertex has an attribute named *size* denoting the buffer size it requires to store its padded output feature maps. Each edge also has the *size* attribute denoting the size of data flowing from the start vertex to the end vertex. At the beginning, the size of each edge is initialized by that of its start vertex. The partition factor PF is initialized by a random *seed* value. The ***Step 1*** allocates on-chip memory for stream buffers and activation buffers under the constraint of Q_{ext}. It firstly partitions the input CNN model represented by a graph $G(V, E)$ into groups according to the number of accelerators n. Each group contains n layers that are mapped onto the n accelerators in a round as shown in Fig. 7(b). The groups are formed into a new graph called group graph $GG(V_1, E_1)$ where the vertices are groups and the edges mean data dependency between groups. In



line 10, We accumulate the stream buffer sizes across all groups and compute the required total on-chip memory size for stream buffers via $\mathcal{N}_{mem}^{B}$ which is defined in [4] to compute on-chip memory blocks. Its three platform independent parameters are the number of buffers, buffer size and partition factor for each buffer. As we don't apply tiling on H, so T_h is 1. $B_l(r)$ returns the kernel width for layer l, and L_n contains all layers executing on accelerator n. After that activation buffers are allocated for GG in procedure $ActBufAlloc$. It is a revised version of the classic register allocation algorithm proposed in [3]. The minimum size of buffer usage S is also returned. The memory interfaces for all vertices are recorded in IF. We omit its implementation details and only strength our revisions. This procedure consists of 3 steps as known as **Build**, **Color** and **Spill** as shown in Fig. 7(b). We firstly **Build** a buffer interference graph BIG via dataflow analysis and live range analysis on GG. The **Color** phase assigns each vertex a color that is different from all its neighbors. Each color has a weight value denoting the on-chip buffer size. It must be large enough to accommodate the largest required buffer size of the vertices with the same color. Once a color is selected for a vertex, its weight will be updated if its current value is smaller than the size of the vertex. Each time we finish coloring a vertex, we will examine if the total size of current colors exceeds Q_{act}. If Q_{act} has been exceeded, we go to **Spill** and select a vertex from the spilling list with the maximal cost to spill it into off-chip memory.

In ***Step 2***, we find a minimum $\mathcal{L}$ in Eq. 1 given the number of accelerators and dies, as well as resource constraints on each die. As $\mathcal{L}$ is the summation of latencies of all the rounds and each round executes on the same set of accelerators, therefore the minimal latency for each round could guarantee the minimum of $\mathcal{L}$. Under the third pruning strategy, we use a dynamic programming algorithm to compute the minimal latency of a given round. The suboptimal structure $\mathcal{L}(r, i, j, dj, bj)$ means the minimum latency of round r when assigning i accelerators on j dies with **remaining** available DSP, BRAM resources on the j^{th} die being dj and bj. The i^{th} accelerator could only be put on the, (1) j^{th} die or, (2) some die before the j^{th} die. According to the pruning strategy, in the second case, there will be no accelerators being put on the j^{th} die. Hence we only consider the resource limitations on the j^{th} die in the suboptimal structure instead of those on all j dies. The state transition function is shown in Equation 5, where $b(i, j)$ is a binary variable. $b(i, j) = 1$ indicates that we put the i^{th} accelerator on the j^{th} die; otherwise, we don't put the i^{th} accelerator on the j^{th} die. The algorithm process is illustrated in Fig. 7(c). When $b(i, j) = 0$, for a given round, the solution is equal to that when the number of dies is $(j - 1)$. When $b(i, j) = 1$, we enumerate a resource constraint pair (x', y') for DSP and on-chip memory and leverage the design space exploration designed in [23] to find a design with the highest performance under (x', y') for the i^{th} systolic array with actual resource consumption (x, y). In this case, the latency is determined by the maximum of latency of the i^{th} systolic array and that of when we put $(i - 1)$ systolic arrays on j dies supposing the remaining DSP and on-chip memory resources on the j^{th} die are dj and bj. Thus $\mathcal{L}(r, i, j, dj, bj)$ is the minimum among all the (x', y') pairs. At the end of each iteration, Q_{ext} will be updated by adding ΔQ which is taken from each die one at a time in a round-robin way. PF is updated by the average partition factor of internal input buffers across all systolic arrays as well. The time complexity of the ***Step 1*** is $O(|V|^2)$. We regard the time for obtaining $\mathcal{L}_{acc}$ by DSE as a constant C, because it traverses the upper bounds of convolutions which are constant. Thus the time complexity of the ***Step 2*** is $O(\frac{|V|}{n} * n * m * \mathcal{N}_{dsp} * \mathcal{N}_{mem}^{total}) = O(|V| * m * \mathcal{N}_{dsp} * \mathcal{N}_{mem}^{total})$. So the time complexity of Alg. 1 is $O(|V|^2 * \frac{\mathcal{N}_{mem}^{total}}{\Delta Q}(m * \mathcal{N}_{dsp} * \mathcal{N}_{mem}^{total} + |V|))$.

5 EVALUATION

5.1 Methodology

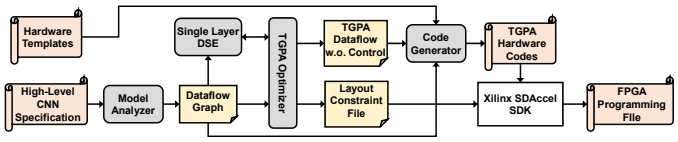


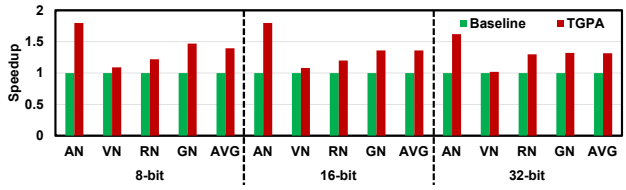
Figure 8: Automation flow

We implement an automation flow to generate an executable system on FPGAs from a high-level CNN specification. The main components of the flow is shown in Fig. 8. We leverage the PyTorch [18] framework to parse the high-level CNN descriptions into computation graphs that are stored as dataflow graphs. The nodes of a dataflow graph are layers and edges are data dependencies between layers. With the dataflow graph as input, the TGPA optimizer determines the number of systolic arrays and assignment of them on multiple dies, as well as on-chip memory allocation for each systolic array by Alg. 1. Structure of each systolic array is obtained from our design space exploration process. The output of the TGPA optimizer is the TGPA pipeline without multi-/single-layer control logics. After that, the structures of all accelerators are parameterized to instantiate template files, including Vivado HLS systolic array implementations, ReLU, pooling and stream buffers (kernel). The control logics are also inserted directed by the dataflow graph. Finally, the instantiated kernel is synthesized by the Xilinx SDAccel SDK for the physical implementation under multi-die assignment constraints. The systolic array assignment information is used to define the constraint file.

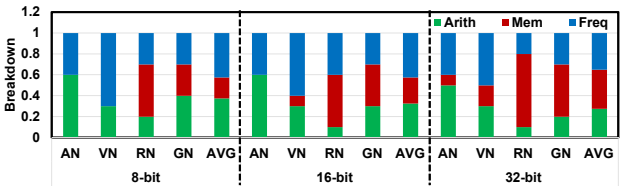
We use four latest CNN models as our benchmark suite. It includes AlexNet (**AN**), VGG-19 (**VN**), ResNet-152 (**RN**) and GoogleNet (**GN**). We evaluate the designs generated by our automation flow and algorithm on AWS F1 EC2 instances. There are two kinds of instances on F1 equipped with FPGAs. Currently we use the instance size including one Xilinx VU9P FPGA. The FPGA has 3 dies containing 2585 K logic elements and 6800 DSPs. Moreover, it has about 40 megabytes on-chip memory resources including 75.9 Mb Block RAMs and 270 Mb UltraRAMs. The underlying TGPA designs are implemented by Vivado HLS and synthesized by Xilinx SDAccel 2017.1 flow. We use various precisions, 8- and 16-bit fixed point, and 32-bit floating point data types to evaluate how different precisions affect resource utilization. Our evaluation methodology consists of three parts. We firstly verify the effectiveness of the TGPA designs by comparing them with baseline designs. The baseline designs are homogeneous designs as shown in Fig. 1(a) using the systolic array implementations as shown in Fig. 5 without input/output and bypass controls. Then we analyze the scalability of the TGPA designs as the number of dies increases. In the effectiveness and scalability experiments, the batch size we use is 1 and the fully connected layers are not included during the execution. We finally compare the TGPA designs for end-to-end evaluation with the state-of-the-art CNN designs. In this evaluation, we use a small batch size 4 to avoid underutilization of PE array when executing fully connected layers.

5.2 Performance Improvement over Baseline

Fig. 9(a) shows the performance improvement of TGPA designs over the baseline designs. We can see that on average, the performance improvements of TGPA architecture are 56%, 48% and 32% for 8-bit,



(a) Overall performance improvement over baseline designs



(b) Performance improvement breakdown

Figure 9: Performance improvement

Table 2: Detailed design information

Model	Precision	Baseline			TGPA					
		Arith.	Mem. (INT)	Freq. (MHz)	Acc#	Mapping	Arith.	Mem. INT	Mem. EXT	Freq. (MHz)
AN	8-bit	40%	14%	160	5	(1,1,3)	80%	9%	6%	200
	16-bit	40%	20%	160	5	(1,1,3)	80%	16%	10%	200
	32-bit	56%	18%	160	5	(1,1,3)	84%	24%	22%	200
VN	8-bit	77%	13%	160	4	(1,1,2)	89%	10%	6%	210
	16-bit	77%	19%	160	4	(1,1,2)	89%	16%	12%	210
	32-bit	84%	19%	160	4	(1,1,2)	92%	22%	20%	210
RN	8-bit	68%	8%	140	10	(3,3,4)	84%	6%	14%	200
	16-bit	68%	14%	140	10	(3,3,4)	84%	14%	30%	200
	32-bit	80%	18%	140	6	(2,2,2)	88%	19%	40%	200
GN	8-bit	56%	8%	140	11	(3,4,4)	84%	6%	19%	200
	16-bit	56%	14%	140	11	(3,4,4)	84%	15%	30%	200
	32-bit	70%	18%	140	8	(2,3,3)	88%	20%	48%	200

16-bit and 32-bit precisions respectively. The improvements are mainly from three aspects. First, arithmetic utilization is improved by heterogeneity of TGPA. Second, the external buffers of TGPA reduce off-chip memory data transferring for activations. Third, frequency improvement is obtained by placement of accelerators onto multiple dies. The performance improvement breakdown for them is shown in Fig. 9(b). The design details including the number of accelerators and their mapping results on multiple dies, arithmetic utilization and buffer allocation results are all shown in Tab. 2. For the baseline designs, the number of accelerators is 1 and we only allocate internal buffers for them. For the mapping results of TGPA designs, we list the number of accelerators that are mapped on each of the 3 dies.

We can see that AlexNet benefits most from arithmetic utilization improvement (about 40% improvement). That is because the number of input feature maps of its first layer is only 3 while for other layers, it is at least 96. This differentiation is exaggerated by the fact that AlexNet has only 5 convolutional layers. The specific design for its first layer effectively mitigates this differentiation. VGG-19 has a relatively regular network structure for different layers, so we only have 12% of performance improvement from four systolic arrays. Both AlexNet and VGG-19 barely have improvement from buffer allocation because all five layers are no longer bounded by off-chip memory bandwidth with less than 80% of Block RAMs for internal buffers. Although ResNet-152 has residual building block structure, its layer shapes are regular except for kernel sizes of which about two-thirds are 1x1 kernels. The regular layer shapes cause limited performance improvement (8%) from heterogeneous design. However, at least the layers with 1x1 kernel sizes are memory bound for input feature maps because data reuse for these layers are limited compared with those with 3x3 or 7x7 kernel sizes. As a result, ResNet-152 benefits most from buffer allocation. GoogleNet could both benefit from heterogeneous design and buffer allocation. For one thing, the concatenation operations in these two models increase irregularity. For the other, there is also a large proportion of 1x1 kernels in them.

5.3 Scalability

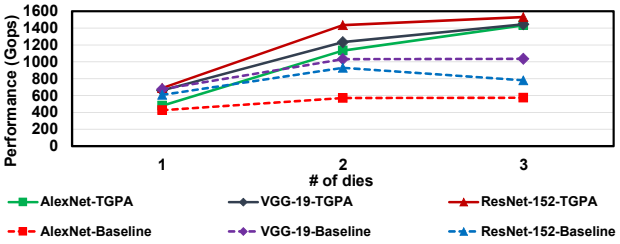


Figure 10: Scalability of TGPA

In addition to higher DSP efficiency and better crossing-die placement adaptability, the TGPA architecture also has better scalability than the homogeneous systolic array design on a multi-die FPGA. We evaluate three CNN models, AlexNet, VGG-19 and ResNet-152, as case studies for the scalability analysis. We still use the systolic array design as our baseline homogeneous design. Then three optimal designs are generated under the resource limitations in terms of one, two and three dies respectively. Compared to the one die designs, we get 1.46X and 1.39X performance improvement for the two dies and three dies designs on average as shown in Fig. 10. We also generate three TGPA designs of which the accelerators are mapped on one, two and three dies. For the TGPA designs that are mapped on two and three dies, we get average 2.01X and 2.24X performance improvement over the design that executes on one die. The TGPA designs have better scalability as the number of dies increases. On one hand, more dies provide more resources and flexibility. On the other hand, the crossing-die timing issue makes the homogeneous design not scalable.

5.4 Comparison with State-of-the-art Designs

Finally we compare our end-to-end results with state-of-the-art designs. In this experiment we still use AlexNet, VGG and ResNet-152 for comparison (GoogleNet's on-board results are still not available in any published papers we could find). We use designs with the best results on different FPGA platforms we could find for this comparison. We use three metrics for performance comparison. Latency is the time for processing a small batch size (4). It is used to validate the effectiveness of our designs. Throughput is for performance comparison's purpose. It (Gops) is not used to measure the rate for processing multiple images. We also compare the performance density which reflects arithmetic utilization. It is defined as the number of arithmetic operations that one DSP slice executes in one cycle. The comparison results are shown in Table 3. For AlexNet and ResNet-152, the latency results are better than all other works. For VGG, our latency is longer than [29] due to 45% frequency gap. That is because the authors in [29] tune the frequency of their design at RTL level while TGPA is totally implemented by HLS. In addition, our designs have better performance densities than others except the results in [2, 6, 16] which exceed 2. The work in [2] adopts Winograd transformation for arithmetic optimization, which has been widely used on different acceleration platforms [8, 13, 14]. The works in [6, 16] also resort to logic resources to implement part of multiplications for the fixed point design. We implement all of our multiply and accumulation operations by DSPs. Finally, the throughputs of our designs outperform most of the state-of-the-art results. First of all, our designs have higher DSP consumption to support more parallelism. Second, the frequencies are competitive

Table 3: Comparison to state-of-the-art designs

Design	[2]	[23]	TGPA	[26]	[29]	TGPA	[6]	[16]	TGPA
CNN Model	AlexNet			VGG			ResNet-152		
FPGA	Arria10 GX 1150	Arria10 GT 1150	Xilinx VU9P	Xilinx KU060	Arria10 GX 1150	Xilinx VU9P	Stratix-V GSMD5	Arria10 GX 1150	Xilinx VU9P
Frequency (MHz)	303	240	200	200	385	210	150	150	200
Precision	float 16 bit	float 32 bit	fixed 16 bit	fixed 16 bit	fixed 16 bit	fixed 16 bit	fixed 16 bit	fixed 16 bit	fixed 16 bit
Logic Utilization	246K (58%)	350K (82%)	468K (40%)	100K (31%)	N/A	493K (42%)	42K (25%)	141K (33%)	506K (43%)
DSP Utilization	1518 (100%)	1320 (87%)	4480 (66%)	1058 (38%)	2756 (91%)	4096 (60%)	1036 (65%)	1046 (69%)	4096 (60%)
BRAM Utilization	2487 (92%)	2360 (86%)	3364 (78%)	782 (36%)	1668 (61%)	3380 (78%)	919 (46%)	2536 (93%)	2960 (68%)
URAM Utilization	N/A	N/A	117 (13%)	N/A	N/A	140 (15.6%)	N/A	N/A	351 (39%)
Throughput (Gops)	1382	360	1432	266	1790	1510	364	315	1463
Latency/Image (ms)	0.98	4.05	1.03	101.15	17.18	22.35	62.14	71.71	17.34
Performance Density (ops/DSPslice/cycle)	3.09	1.17	1.60	1.26	1.68	1.84	2.35	2.01	1.78

because we use systolic array and there are few crossing-die timing critical paths. Third, usage of UltraRAM greatly reduces off-chip memory footprints.

6 RELATED WORKS

Design and synthesis of hardware accelerators on FPGA for CNNs has been one of the most active research topics in computer architecture and design automation. The prior works can be broadly divided into two categories: homogeneous and heterogeneous designs.

Homogeneous Designs. The works of [15, 27] adopt a parallel PE array with centralized local buffer design to exploit optimal parallelism factors. The design in [29] uses a distributed local buffer design to reduce the complexity of computation to communication interconnects. The works of [2, 17, 23] map convolutional and fully connected layers onto systolic arrays to improve frequency through global data transfer elimination and low fan-in/fan-out. All of the designs above are homogeneous designs that have benefit on latency reduction of a single input image. But they have low resource efficiency due to irregularity of CNN models. In addition, deploying a homogeneous design on a multi-die FPGA will encounter timing issues due to the long delay of too many crossing-dies paths.

Heterogeneous Designs. Heterogeneous design has been adopted in several previous works to improve the resource efficiency for the entire model. The authors in [12, 19] place multiple convolutional layer accelerators on a single FPGA in order to maximize the overall resource utilization by optimizing each accelerator for one or some specific layers that are executed on this accelerator. All of them aim to improve throughput of multiple input images rather than reduce latency of one image execution. The kernel fusion technique [1, 24] adopts pipelining strategy between multiple layers to save off-chip memory accesses. There are no discussions in those works on how to solve the resource inefficiency issue. The work in [28] exploits the layer irregularity on a FPGA cluster containing multiple FPGAs. The authors in [22] also adopt layer specific accelerator to improve arithmetic efficiency via reconfiguration.

As far as we know, our work is the first to reduce CNN inference latency leveraging the high resource efficiency of heterogeneous design.

7 CONCLUSIONS

In this work, we propose a new CNN accelerator architecture named tile-grained pipeline architecture (TGPA) for low latency inference. TGPA uses the heterogeneous design methodology to resolve the resource inefficiency existing in the homogeneous designs, which helps to reduce latency when a single input image executes in tile-grained pipelining on the multiple accelerators. By adopting systolic array for single accelerator design and placing accelerators onto different FPGA dies to avoid crossing-dies timing critical paths, TGPA achieves higher frequency than homogeneous designs. Experiment results show that the TGPA designs generated by our algorithm achieve up to 40% performance improvement than homogeneous designs and 3X latency reduction than previous designs.

8 ACKNOWLEDGEMENTS

This work is supported by Beijing Natural Science Foundation (No. L172004). We thank all the anonymous reviewers for their feedback.

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] G. Chaitin. 2004. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Not.* (2004).
[4] J. Cong, P. Wei, C. H. Yu, and P. Zhou. 2017. Bandwidth Optimization Through On-Chip Memory Restructuring for HLS. In *DAC*.
[5] H. Gao, Z. Liu, K. Q. Weinberger, and L. van der Maaten. 2017. Deep Residual Learning for Image Recognition. In *CVPR*.
[6] 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*.
[7] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.
[8] Intel. 2016. "Not so fast, FFT": Winograd. https://ai.intel.com/winograd/. (2016).
[9] N. P. Jouppi, C. Young, N. Patil, and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.
[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*.
[11] H. T. Kung and C. E. Leiserson. 1979. *Algorithms for VLSI Processor Arrays*.
[12] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang. 2016. A High Performance FPGA-based Accelerator for Large-scale Convolutional Neural Networks. In *FPL*.
[13] Liqiang Lu and Yun Liang. 2018. SpWA: An Efficient Sparse Winograd Convolutional Neural Networks Accelerator on FPGAs. In *DAC*.
[14] L. Lu, Y. Liang, Q. Xiao, and S. Yan. 2017. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. In *FCCM*.
[15] 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*.
[16] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J. s. Seo. 2017. End-to-end scalable FPGA accelerator for deep residual networks. In *ISCAS*.
[17] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss, and E. Chung. 2015. Toward Accelerating Deep Learning at Scale Using Specialized Hardware in the Datacenter. In *Hot Chips*.
[18] PyTorch. 2018. https://pytorch.org. (2018).
[19] Y. Shen, M. Ferdman, and P. Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *ISCA*.
[20] 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. In *CVPR*.
[21] 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*.
[22] S. I. Venieris and C. S. Bouganis. 2016. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *FCCM*.
[23] 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*.
[24] 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*.
[25] Xilinx. 2018. Large FPGA Methodology Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/ug872_largefpga.pdf. (2018).
[26] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong. 2016. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In *ICCAD*.
[27] 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*.
[28] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong. 2016. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster. In *ISLPED*.
[29] J. Zhang and J. Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *FPGA*.