LCP: a Layer Clusters Paralleling mapping method for accelerating Inception and Residual networks on FPGA

Xinhan Lin, Shouyi Yin*, Fengbin Tu, Leibo Liu, Xiangyu Li, and Shaojun Wei
Institute of Microelectronics, Tsinghua University, Beijing 100084, China
*Corresponding author: yinsy@tsinghua.edu.cn

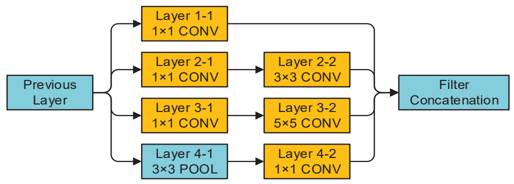
ABSTRACT

Deep convolutional neural networks (DCNNs) have been widely used in various AI applications. Inception and Residual are two promising structures adopted in many important modern DCNN models, including AlphaGo Zero's model. These structures allow considerably increasing the depth and width of the network to improve accuracy, without increasing the computational budget or the difficulty of convergence. Various accelerators for DCNNs have been proposed based on FPGA platform because it has advantages of high performance, good power efficiency, and fast development round, etc. However, previous FPGA mapping methods cannot fully adapt to the different data localities among layers and other characteristics of Inception and Residual, which leads to a underutilization of FPGA resources. We propose LCP, a Layer Clusters Paralleling mapping method to classify the layers into clusters based on their differences of parameters and data localities, and then accelerate them in different partitions of FPGA. We evaluate our mapping method by implementing Inception/Residual modules from GoogLeNet [8] and ResNet-50 [4] on Xilinx VC709 (Virtex 690T) FPGA. The results show that the proposed method fully utilizes resources and achieves up to 4.03× performance than the baseline and 2.00× performance than the state-of-the-art methods.

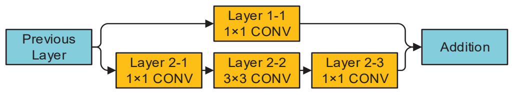
1 INTRODUCTION

Deep convolutional neural networks (DCNNs) have achieved surprising accuracy in many intelligent applications such as image classification, object detection, speech recognition, action recognition, scene understanding, and so on. Recent developments of DCNN models raise a new trend that networks are becoming deeper and wider for higher accuracy. However, broadening and deepening traditional DCNN structures usually increase the amount of computations and the difficulty of convergence. Inception [8] and Residual [4] are two state-of-the-art DCNN structures carefully crafted to provide high accuracy while avoiding the above problems. DeepMind's new Go software, AlphaGo Zero [7], using a Residual network, beat its predecessor AlphaGo Lee by winning 100 games to 0 after only 3-day training, which demonstrates the advancement of this structure. The typical structure of Inception and Residual modules are shown in Fig 1. The most salient features of these structures are: **1)** most layers are convolutional (CONV) layers, **2)** their parameters (e.g. the convolution kernel size) vary in a large range, which leads to large differences of the reuse time of inputs/outputs/weights, i.e. large differences of data localities, **3)** at the first layer, different convolution kernels are performed on the same input feature maps. The non-CONV layers such as pooling and ReLU are simple and easy to accelerate, thus they are omitted in this paper. Because Inception and Residual modules are two of the best DCNN structures up to now, the analysis and experiment in this paper are focused on such Inception/Residual modules, although our work applies to general DCNN structures.

DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196067



(a) Inception module: Inc3a in GoogLeNet [8]



(b) Residual module (block): Res2a in ResNet-50 [4]

Figure 1: Typical structure of Inception / Residual modules

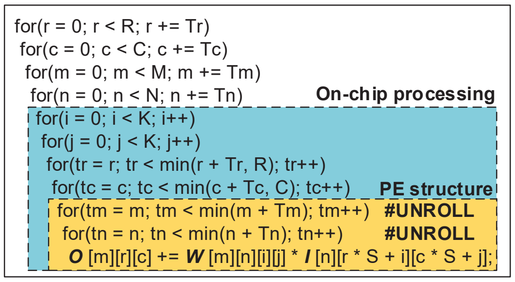
For accelerating DCNNs, FPGA-based accelerators has attracted more and more attention, due to their good performance, high energy efficiency and fast development, especially with the usage of high-level synthesis (HLS) tools. Due to the limited hardware resources of FPGA, usually mapping the entire network onto a single chip is difficult, so most DCNN to FPGA mapping methods [3, 5, 10] use a tiling-based and layer-by-layer style. Each layer is divided into tiles and then each tile is mapped onto the configured computing engine of FPGA. The layers are processed serially, and usually the computing engine stays unchanged when switching the processed layer, otherwise hardware designing would be very challenging [10]. We call such kinds of mapping methods **layer-level temporal mapping** in this paper. The main problem of such methods is that the computing engine shared by all layers forces all layers to use unified tiling parameters. If the layers have much different data localities, their resource requirements would not be simultaneously met, and cause underutilization of resource. For example, in Fig. 1(a), the weights in 3×3 and 5×5 kernels tend to be reused more times than those in 1×1 kernels, that means using less I/O bandwidth to support more computations. Therefore, when mapped to a same FPGA by temporal mapping, either the layers with 1×1 kernel waste computing resources, or the ones with larger kernels waste bandwidth.

On the basis of Inception and Residual networks' characteristics, we propose LCP, a Layer Clusters Paralleling Mapping method for accelerating Inception and Residual networks on FPGA. The key idea of it is classifying the layers into clusters based on their data localities and configurations, and then spatially mapping these layer clusters onto FPGA for parallel execution, thus the different hardware resource requirements of layers can be complementary. The main contributions are summarized as follows:

- The first work (to the best of our knowledge) to spatially map Inception/Residual modules onto FPGA based on their characteristics.
- A Layer Clusters Paralleling Mapping method system, including an effective and flexible distance-based algorithm for layer clustering, an efficient solving technique for the clusters paralleling problem, and cross-branch/layer optimizations for further improving resource utilization.
- A Caffe-to-FPGA framework puts our optimization methods together to maximize the throughput of FPGA for Inception/Residual networks acceleration.

2 BACKGROUND AND RELATED WORK

Since the limited resources on FPGA usually cannot entirely hold a whole CONV layer, most mapping methods use tiling techniques to divide the layer into tiles, and then execute the tiles one by one on a same hardware engine. Fig. 2 presents the pseudo code of the typical tiling-based DCNN accelerator widely used in previous work [3, 5, 10]. In this accelerator, loop R, C, M and N are tiled by Tr, Tc, Tm and Tn, and all the computations and data movements within a $Tr \times Tc \times Tm \times Tn$ tile are performed on-chip. The inner most two loops Tm, Tn are unrolled to exploit hardware parallelism.



```
for(r = 0; r < R; r += Tr)
 for(c = 0; c < C; c += Tc)
  for(m = 0; m < M; m += Tm)
   for(n = 0; n < N; n += Tn)                        On-chip processing
    for(i = 0; i < K; i++)
     for(j = 0; j < K; j++)
      for(tr = r; tr < min(r + Tr, R); tr++)
       for(tc = c; tc < min(c + Tc, C); tc++)         PE structure
        for(tm = m; tm < min(m + Tm); tm++)  #UNROLL
         for(tn = n; tn < min(n + Tn); tn++)  #UNROLL
          O [m][r][c] += W [m][n][i][j] * I [n][r * S + i][c * S + j];
```

Figure 2: Pseudo code of the baseline DCNN accelerator

The FPGA-based accelerating architecture corresponding to this pseudo code is shown in Fig. 3. It is composed of a computing core, weight/input/output buffers and a controller. The computing core performs convolutions, activate and pooling functions on to the input maps. It has Tm processing elements (PEs), each of which is a Tn-width vector dot-product unit connected with an accumulator in the end. This multiplier-adder tree structure corresponds to the unrolled inner most two loops Tm, Tn. The buffers are used to store the input, output and weight data of a tile. In previous temporal mapping methods [3, 5, 10], multiple layers are executed serially and shares a same computing core, so an on-chip controller is needed to adjust the iteration times and data addresses when the executed layer is switched. But the structure of computing core cannot be reconfigured during runtime, otherwise the design difficulty and hardware overheads would be large [10]. Therefore, in layer-level temporal mapping, all layers should use a same set of unroll factors (Tn, Tm), which is an important cause of resources underutilization. In our work, multiple accelerators in this pattern are used in association to accelerate layer clusters in parallel.

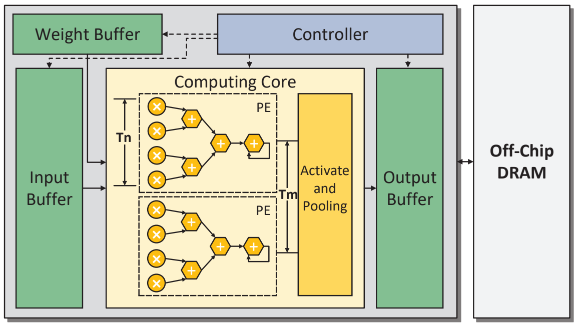


Figure 3: Architecture of the baseline DCNN accelerator

There are also several DCNN mapping methods beyond layer-level temporal mapping. [6] proposed a method that partitions the available FPGA resources into multiple processors, each of which is tailored for a different subset of the DCNN CONV layers, so that the resource utilization can be improved. However, for Inception and Residual modules, their work lacks of specific optimizations such as reusing the data among branches and layers, thus it tends to require more bandwidth, limit the computational parallelism, and finally lower the performance. [1] fuses the processing of multiple DCNN layers and enables caching of intermediate data between adjacent layers, so that the off-chip data transfer are reduced. However, this mechanism may cause severe limitations on tiling parameters and heavy burden on on-chip buffer. Moreover, their work lacks of optimizations for computing resources utilization.

3 MOTIVATION

Because the data locality of different layers in an Inception/Residual network vary greatly, layer-level temporal mapping usually causes underutilization of DSPs and memory bandwidth in FPGA. We take Inc3a for example to illustrate this. Inc3a is an typical Inception module in GoogLeNet [8]. The parameters of CONV layers in Inc3a are listed in Table 1. We can see that the number of operations (NOP), the number of Data ($NData$, including inputs, outputs and weights) and the theoretical maximum number of supported operations per datum (OPD_{max}, indicating the data localities) vary greatly among different layers. When using a typical temporal mapping method [10] to implement Inc3a onto a Xilinx Virtex-7 690T FPGA with 200MHz clock frequency and 9GB/s I/O bandwidth limit, the DSPs and bandwidth are underutilized, as shown in Fig. 4.

Table 1: Inc3a parameters

Layer	R	C	N	M	K	S	NOP	$NData$	OPD_{max}
1-1	28	28	192	64	1	1	19267584	212992	90.46
2-1	28	28	192	96	1	1	28901376	244224	118.34
2-2	28	28	96	128	3	1	173408256	297344	583.19
3-1	28	28	192	16	1	1	4816896	166144	28.99
3-2	28	28	16	32	5	1	20070400	54272	369.81
4-2	28	28	192	32	1	1	9633792	181760	53.00

There are 3 major causes for this. **First**, the OPD_{max} of different layers are quite distinct. The target FPGA in this example can support about 128 OPD if fully utilized, and the layer with the OPD_{max} most closed to this value (e.g. Layer 2-1) tends to achieve best overall resource utilization. If a layer's OPD_{max} is much larger than

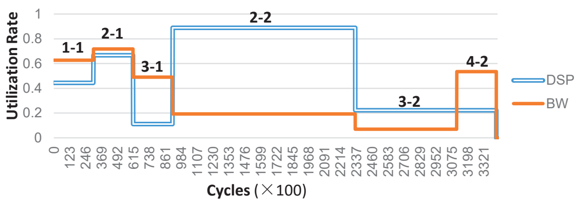


Figure 4: The utilization rate of DSPs and I/O bandwidth over cycles (Network: Inc3a 32bit, FPGA: Xilinx Virtex-7 VX690T, 200MHz, 9GB/s, Mapping method: [10])

platform's OPD (e.g. Layer 2-2), it tends to be compute-intensive and the utilization rate of DSP would far exceed that of bandwidth. On the contrary, a layer with much smaller OPD_{max} (e.g. Layer 3-1) tends to be data-intensive and would waste a lot of DSPs. **Second**, temporal mapping forced unified unroll factors are not fit for all layers. In this example, $(Tn = 5, Tm = 128)$ are used to tile all the layers. However, the size of some layers are too small (e.g. Layer 3-2), or not divisible by unroll factors, resulting in many idle cycles of PEs. **Last but not least**, previous mapping methods do not take the special characteristics of Inception/Residual networks into account, such as the data reusability among branches and contiguous layers, and thus loses the underlying optimization opportunities.

We develop several strategies to address these problems. **First**, the layers can be classified into several clusters and the layers in a same cluster have similar resource utilization tendency. Different clusters will be executed on different partitions of FPGA in parallel (**spatial mapping**), so that their resource requirements will be complementary, and different unroll factors can be used to fit for each cluster. **Second**, the first layers (with the same kernel size) of different branches can be merged, so that the size of layers are enlarged and the potential computational parallelism and data locality are increased. **Third**, some inter-layer data transfer can be performed on-chip, thus the bandwidth resources can be saved.

4 LCP: LAYER CLUSTERS PARALLELING MAPPING

4.1 Layer Clustering

We develop a distance-based algorithm to classify layers into clusters. The distance represents the difference of resource utilization tendencies between two layers, and the smaller it is, the greater chance these two layers are classified into a same cluster. To exactly define the distance, we first have to define the feature vector $\mathbf{L_i}$ for each layer i. $\mathbf{L_i}$ is composed of seven components, the first one of which is the theoretical maximum number of supported operations per datum (OPD_{max}), and the rest are the layer's CONV parameters N, M, R, C, K, S. Because of their large variation ranges, OPD_{max} is normalized by the FPGA's OPD (deduced by the ratio of computing power to bandwidth), while other components are normalized by their average values. That is to say, $\mathbf{L_i} = \{l_{i1}, l_{i2}, l_{i3}, l_{i4}, l_{i5}, l_{i6}, l_{i7}\}$, where $l_{i1} = OPD_{max,i}/OPD_{FPGA}$, $l_{i2} = N_i/\bar{N}$, $l_{i3} = M_i/\bar{M}$, $l_{i4} = R_i/\bar{R}$, $l_{i5} = C_i/\bar{C}$, $l_{i6} = K_i/\bar{K}$, $l_{i7} = S_i/\bar{S}$. Then, the distance between two feature vectors is: $D(\mathbf{L_i}, \mathbf{L_j}) = \sqrt{\sum_{x=1}^{7}\{\alpha_x \cdot (l_{ix} - l_{jx})^2\}}$, where α_x is an empirical weight for x-th component. Since OPD_{max} directly reflects the data locality, N and M decide the upper limit of computational parallelism, their weights are much larger then others, e.g. $\alpha = (0.6, 0.15, 0.15, 0.025, ..., 0.025)$.

Algorithm 1 Layer Clustering

```
Require: $*L, *\alpha, NC_{EXP}, \theta_E, \theta_S, \theta_M, I_{max}$
Ensure: $NC, **Cluster$
 1: $NC = NC_{EXP}; I = 0;$
 2: $*Center$ = init_cluster_centers($L$, $NC_{EXP}$);
 3: while ($I++ < I_{max}$)
 4:     $*Center_{bak}$ = copy($Center$);
 5:     // Layer Classification:
 6:     for ($i = 1; i \le NC; i++$)
 7:         $NL[i] = 0; Cluster[i] = \emptyset;$
 8:     for ($i = 1; i \le NL_{total}; i++$)
 9:         if ($D(L[i], Center[j]) \le D(L[i], Center[k])$, forall $1 \le j, k \le NC$)
10:             $Cluster[j][NL[j]++] = L[i]$; // insert $L[i]$ into $Cluster[j]$
11:     // Cluster Elimination:
12:     for ($i = 1; i \le NC; i++$)
13:         if ($NL[i] < \theta_E$)
14:             remove $Cluster[i]$; remove $Center[i]$;
15:             $NC--$; break;
16:     // Center Updating:
17:     for ($i = 1; i \le NC; i++$)
18:         $Center[i]$ = calculate_mean($Cluster[i]$);
19:     // Cluster Splitting:
20:     if ($NC \le NC_{EXP}/2$)
21:         for ($i = 1; i \le NC; i++$)
22:             $*V$ = calculate_varience($Cluster[i]$); //$V[x]$ corresponds to $x$-th component of layer feature vector
23:             $WV_{max}$ = max($\alpha[x] * V[x]$); //maximum weighted varience
24:             if ($WV_{max} > \theta_S$)
25:                 insert two new centers: $Center[i] \pm (0, \ldots, 0, WV_{max}/\alpha[x], 0, \ldots, 0)$;
26:                 remove $Center[i]$; $NC++$; break;
27:     // Cluster Merging:
28:     foreach ($1 \le i, j \le NC, i \ne j$)
29:         if ($D(Center[i], Center[j]) \le \theta_M$)
30:             insert a new center: $(NL[i] * Center[i] + NL[j] * Center[j])/(NL[i] + NL[j])$;
31:             remove $Center[i]$; $NC--$; break;
32:     // Termination:
33:     if (is_unchanged($Center$, $Center_{bak}$))
34:         break;
35: do layer classification based on final cluster centers;
```

Algorithm 1 shows the detail process of our algorithm, which is inspired by ISODATA [2], a well-known clustering algorithm for grouping similar objects. First, our algorithm sets initial cluster centers for excepted number of clusters. Then each layer is classified into the most closed center's cluster. After that, it iteratively adjusts the clusters and updates the cluster centers. If any cluster contains too few layers, it would be eliminated. If any cluster has a too large inner scatter (deduced by the variance of the layers in it), it would be split into two. If two clusters are too closed, they would be merged into one. The thresholds of above processes are predetermined. These processes are repeated until all cluster centers stay unchanged during an whole iteration, or the maximal number of iterations is reached.

4.2 Layer Clusters Paralleling

The layer clusters, each of which contains one or more layer(s) with similar features, are intended to be executed in parallel on different partitions of FPGA. Now the problem is how to search the joint solution (including all clusters' tiling parameters) with maximum performance under the given hardware resource constraints. A cluster's design space includes millions to hundreds of millions of solutions, which only needs several seconds to a few minutes to enumerate (on a PC with Intel i7-6700K CPU and 16GB RAM). However, for two or more clusters, the joint design space is the Cartesian product of all their respective design spaces, which leads to a completely unacceptable enumeration time. To address this

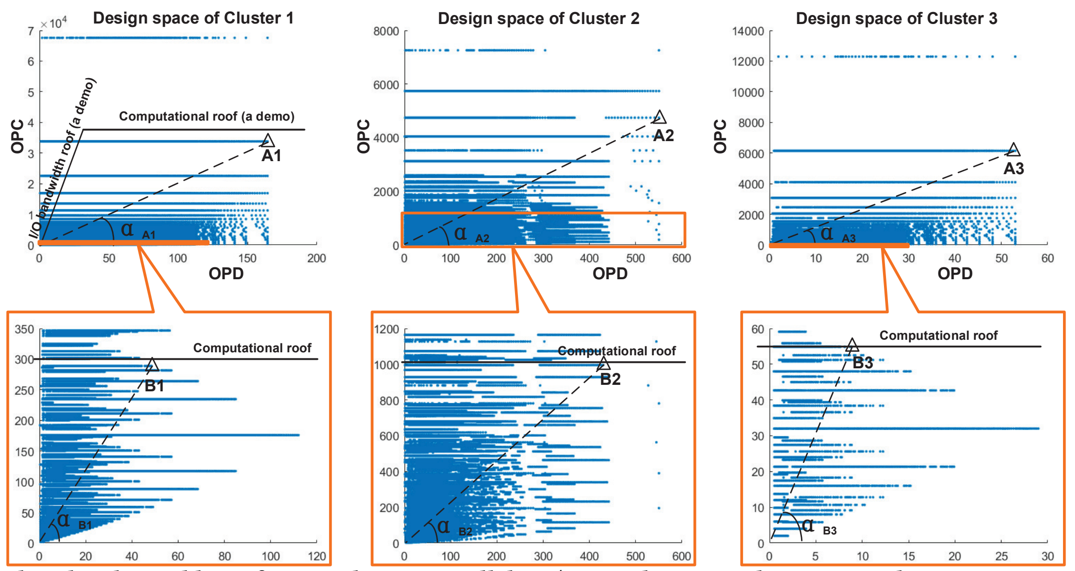


Figure 5: Jointly solve the problem of Layer Clusters Paralleling (Network: Inc3a 32bit, FPGA: Xilinx V7 690T, 200MHz, 9GB/s)

problem, we base on roofline model [9, 10] and develop an efficient technique for simultaneous searching multiple design spaces.

Still take Inc3a for example, we merge its first layers of the former three branch (1-1,2-1,3-1) into a new layer (the related technique will be introduced in Section 4.3). According to the distances, the merged layer is classified into Cluster 1, Layer 2-2 and Layer 3-2 are classified into Cluster 2, and Layer 4-2 are classified into Cluster 3. Then, each cluster's design space in roofline model format is generated, as shown in Fig. 5. The vertical axis represents the numbers of operations per cycle (**OPC**), corresponding to the performance in original roofline model, and the horizontal axis represents the numbers of operations per data (**OPD**), corresponding to the Computation to Communication (CTC) ratio in original roofline model. For example, in the design space of Cluster 1, Point A1 represents a possible design. Its vertical coordinate represents the OPC it can reach, and also implies the computing resources it will consume. Its horizontal coordinate represents the OPD it can reach, and $\tan\alpha_{A1} = OPC/OPD$ represents the I/O data per cycle (**DPC**), implies the bandwidth it will consume. The target FPGA for mapping onto will impose a roof-shaped constraints according to its available resources. Therefore, the uppermost (best performance) and then the rightmost (least bandwidth requirement) design point under the roofline is the possible optimal solution.

Now consider the case of multiple clusters paralleling, which cannot be simply constrained by the original roofline. The computation and bandwidth constraints for simultaneously implementing n design points onto the target FPGA are: **1)** $\sum_{i=1}^{n} OPC_{Ai} \leq OPC_{FPGA}$, **2)** $\sum_{i=1}^{n} \tan\alpha_{Ai} \leq DPC_{FPGA}$, where OPC_{FPGA} and DPC_{FPGA} are deduced from the FPGA's DSP and bandwidth resources. For example, the target FPGA is a Xilinx Virtex-7 690T FPGA with 200MHz clock frequency and 9GB/s I/O bandwidth limit. It has 3600 DSPs, and can support up to 1440 OPC (32bit). Since the overall performance follows the "Cannikin Law", i.e., it is depends on the worst performance of all parallel clusters, our resource allocation strategy is to distribute the computing power to clusters according to the numbers of operations ($NOPs$) they have, so that all the clusters can be finished at approximately the same time. Therefore, we redefine the computational roof of Cluster i as: $OPC_{i,roof} = OPC_{FPGA} \cdot NOP_i / NOP_{total}$. In this example, $OPC_{1,roof} = 297.93$, $OPC_{2,roof} = 1087.90$, $OPC_{3,roof} = 54.17$. Then we find out the uppermost and then the rightmost point under the roofline for each cluster as current candidate solution, as ($B1$, $B2$, $B3$) in Fig. 5. If the candidate solution cannot meet the bandwidth constraint , or cannot be successfully synthesized and implemented, following adjustments are iteratively performed until the design is successfully implemented: **1)** Turning down the computational roofs. It reduces the DSP requirement, and also tends to reduce the bandwidth requirement, at the cost of performance. The ratio between computational roofs is kept as $OPC_{i,roof}/OPC_{j,roof} = NOP_i/NOP_j$, so that the execution time of all clusters are balanced and a huge number of meaningless solutions can be pruned. **2)** Moving the design point left. It tends to reduce the on-chip buffer requirement, at the cost of bandwidth.

4.3 Cross-branch/layer Optimizations

Based on the special characteristics of Inception/Residual networks, we take some additional cross-branch/layer optimizations to enlarge the computational parallelism and data reusability, so that the final resource utilization and overall performance can be further improved. Due to the space limitation, we only introduce two:

1) First layers merging. An Inception/Residual module contains more than one branches, the first CONV layers of which share the same input map. If these layers also have the same kernel size K and stride S, their kernels can be naturally stacked in M direction, so that they are merged into one. Suppose the n original layers has M_1, M_2, ..., M_n output channels respectively, then the merged layer has $M_1 + M_2 + ... + M_n$ output channels. Since there is a one-to-one correspondence between kernel sets (one kernel set = $K \times K \times N$ weights) and output channels, the merging causes no interference among output channels. The merged layer's output can be easily divide into original outputs with M_1, M_2, ..., M_n channels respectively. First layers merging simultaneously enlarges the computational parallelism and data locality. It has almost no side effects, so we tend to perform it before layer clustering at most cases.

2) On-chip inter-layer data transfer. There is a direct production and consumption relationship between successive layers. However, in basic CONV tiling methods [10], tiling is performed within a layer, thus the inter-layer data always have to take circuitous routes, leave from computing array, pass through on-chip buffers and off-chip memory, and come back to computing array. This aggravates the burden on bandwidth and on-chip storage. Work [1] proposes a fused-layer tiling method, the tiling window of which is cross-layer and pyramid-shaped, thus it enables direct data transmission between successive layers and minimizes data movement. However, this mechanism would limit the Tn factors of all layers except the last one to be equal to N, otherwise the output sums are uncompleted and cannot be consumed by next layer. More specifically, fusing k successive layers brings two extra constraints: **i)** $Tn_{i+1} = Tm_i, i = 1, 2, ..., k-1$, **ii)** $Tn_i = N_i, i = 1, 2, ..., k-1$. That would greatly narrow the design space, and require huge on-chip buffer. Based on this, we make some improvement and design an advanced on-chip inter-layer data transmission mechanism. We find that if the $(k-1)$-th layer runs on output reuse mode (each output stays on-chip until all N corresponding products are accumulated) and the k-th layer runs on input reuse mode (each input stays on-chip until all M corresponding weights have multiplied it), the $(k-1)$-th layer could be tiled without constraint ii). Then, considering the synchronization relationship between the $(k-1)$-th layer and the k-th layer, the constraint ii) can be rewritten as: $Tn_i = N_i, i = 1, 2, ..., k-2$, and $\frac{N_{k-1}}{Tn_{k-1}} \cdot K_{k-1}^2 = \frac{M_k}{Tm_k} \cdot K_k^2$. Since k usually equals 2 or other small value [1], our effort considerable loosens the constraint ii), reduces the side effects and extends the application range. This optimization is usually used in the cases with plenty computing resources and insufficient bandwidth.

4.4 Workflow of the Caffe-to-FPGA Mapping Framework

Puts all proposed methods together, we design a mapping framework to maximize the throughput of FPGA for Inception/Residual networks acceleration. It loads the script description of networks in Caffe and produces the FPGA implementation, as shown in Fig.6. First, it takes Caffe prototxt file and the target FPGA's specification as inputs. The first layers of different branches are merged if possible. Second, it performs a distance-based algorithm to divide the layers into several clusters. Then, it optimizes inter-layer data transfer if necessary. After that, it generates the design space for each cluster, and jointly search the global optimum solution for all clusters at once. With the solution containing the tiling parameters of each clusters, it generates a HLS-based C/C++ code, which can be synthesized and implemented by Vivado tools. If the implementation fails, we iteratively adjust the solution until it succeeds.

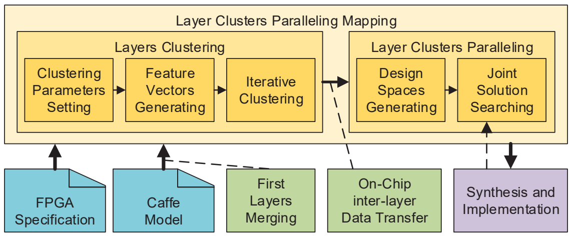

Layer Clusters Paralleling Mapping
Layers Clustering
Clustering Parameters Setting
Feature Vectors Generating
Iterative Clustering
Layer Clusters Paralleling
Design Spaces Generating
Joint Solution Searching
FPGA Specification
Caffe Model
First Layers Merging
On-Chip inter-layer Data Transfer
Synthesis and Implementation

Figure 6: Mapping framework

5 EVALUATION

5.1 Experimental Setup

We evaluate the proposed mapping method with several typical Inception/Residual modules from GoogLeNet [8] and ResNet-50 [4]. Table 2 describes the parameters of these benchmarks. Three state-of-the-art DCNN mapping methods [1, 6, 10] are selected for comparison. Work [10] (**Base**) is a typical tiling-based layer-level temporal mapping and can serve as a baseline. The rest two methods are beyond traditional layer-level temporal mapping. Work [1] (**FL**) fuses adjacent layers to reduce the data movement, and work [6] (**MCLP**) partitions the target FPGA into multiple processors, each of which runs a different subset of the layers. The target chip in the evaluation is Xilinx VC709 (Virtex 690T) FPGA board working on 200 MHz and 9GB/s I/O bandwidth limit.

Table 2: Benchmark parameters

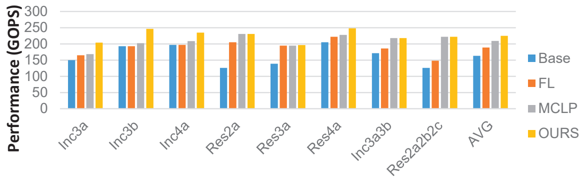
Benchmark	R	C	N	M	K	S	OPD_{max}
Inc3a	28	28	16 ~ 192	16 ~ 128	1 ~ 5	1	28.99 ~ 583.19
Inc3b	28	28	32 ~ 256	32 ~ 192	1 ~ 5	1	54.90 ~ 712.28
Inc4a	14	14	16 ~ 480	16 ~ 208	1 ~ 5	1	28.70 ~ 287.47
Res2a	56	56	64	64 ~ 256	1 ~ 3	1	63.35 ~ 510.55
Res2b	56	56	64 ~ 256	64 ~ 256	1 ~ 3	1	100.76 ~ 510.55
Res2c	56	56	64 ~ 256	64 ~ 256	1 ~ 3	1	100.76 ~ 510.55
Res3a	28	28	128 ~ 256	128 ~ 512	1 ~ 3	1 ~ 2	56.62 ~ 636.93
Res4a	14	14	256 ~ 512	256 ~ 1024	1 ~ 3	1 ~ 2	92.66 ~ 327.71

5.2 Performance Evaluation

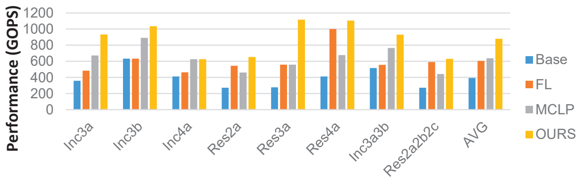
To comprehensively evaluate the performance of the four methods, we implement not only single module but also several consecutive modules (Inc3a3b, Res2a2b2c), and each of them has two bit width versions (32/16). The target FPGA can support at most 128/320 OPD for 32/16 bit operations. For majority of the benchmark layers, the OPD_{max} exceeds 128. Therefore, 32 bit benchmarks tend to be compute-intensive, while 16 bit benchmarks tend to have more balanced resources utilization. The performance evaluation results are shown in Fig. 7. Generally speaking, all the four methods achieve higher performance on 16 bit benchmarks, and the latter three methods all bring performance improvement over baseline. MCLP achieves higher performance than FL on more benchmarks, which demonstrates its advantage on resource allocation. FL's performance never surpasses MCLP's on 32 bit benchmarks, because the benefit of optimization on data movement is limited in compute-intensive cases. Our method comprehensively uses multiple optimizations aimed at inception/residual modules. So it has better effectiveness and applicability than the other three methods. On 32 bit and 16 bit benchmarks, our method achieves 196.27 ~ 246.41 GOPS and 625.92 ~ 1116.54 GOPS, respectively. The average improvements of our method over baseline on 32/16 bit benchmarks are 37.76% and 122.75% respectively, which are considerably higher than those of FL and MCLP. The largest improvement is as high as 303.0%, achieved on Res3a 16bit.

5.3 Resources Utilization Analysis

We take Res3a 16bit, on which our method achieves greatest performance improvement compared to the baseline, as example, to introduce why our optimizations work. Table 3 shows the overview of resource utilization in the mapping of Res3a 16bit on 690T FPGA by the baseline method [10] and our method. We can see that our accelerator has almost fully utilized FPGA's hardware resource. For

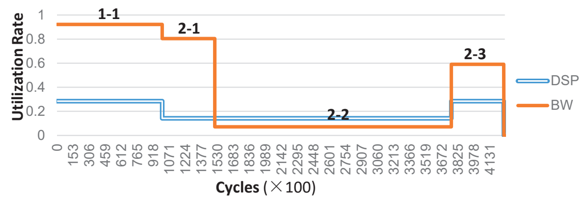


(a) Performance evaluation on 32bit benchmarks

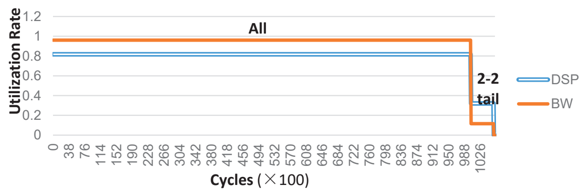


(b) Performance evaluation on 16bit benchmarks

Figure 7: Performance evaluation (VX690T, 200MHz, 9GB/s)



(a) Resources utilization rate of the baseline method [10]



(b) Resources utilization rate of our method

Figure 8: The utilization rate of DSPs and I/O bandwidth over cycles (Net: Res3a 16bit, FPGA: VX690T, 200MHz, 9GB/s)

better analyzing the detail resources utilization of different methods, Fig.8 shows how the resource utilization rates change over cycles in the case of implementing Res3a 32bit on 690T FPGA. From Fig. 8(a) , we can see the utilization curves of the baseline method [10] fluctuates dramatically and stay lower than 20% in long time ranges, thus a lot of resources are waste. This is due to the inherent drawbacks of layer-level temporal mapping. First, the four layers' OPD_{max} are 157.3, 56.6, 636.9, 181.1, respectively, while the platform can support at most 320 OPD for 16bit operations. The first two layers' OPD_{max} are smaller than the platform's, so they are data-intensive and hungry for bandwidth, and waste many DSPs. Second, because of bandwidth constraint, the first two layers cannot use large unroll factors, and these limited unroll factors should also be shared by the rest two layers under temporal mapping, which hinders them to utilizes more resources. In contrast, the utilization rates of our method are higher and more stable. At most cycles, the DSP utilization and bandwidth utilization keep at 81.8% and 96.1%, respectively, as shown in Fig. 8(b). This is owing to the spatial mapping mechanism and aforementioned specific optimizations. First, spatial mapping allows the layer clusters to run in parallel, so that their resource requirements can be complementary. In this example, three clusters use 1280, 1152 and 512 DSPs, respectively. And the bandwidth usages are 5.74, 1.04 and 1.86 GB/s, respectively. Second, each cluster run on a unique partition of FPGA, so that the layers have more freedom to adopt suitable unroll factors, which help to balance the clusters' execution time. The cluster of Layer 2-2 spends a little more time because of the limit of tiling granularity.All the clusters simultaneously start to run and end at similar times, so that the waste of hardware resources is reduced. Third, first layer merging enlarges the layer's potential parallelism and data locality, and on-chip inter-layer data transmission reduce the bandwidth requirement under the same tiling parameters. They together make larger unroll factors feasible, which help to utilize more DSP resources. For the foregoing reasons, our method leads to much better resource utilization and performance.

Table 3: Resources usage (Net: Res3a 16bit, FPGA: VX690T, 200MHz, 9GB/s)

	DSP	BW(GB/s)	BRAM	FF	LUT
[10]	1024 (28%)	8.30 (92%)	520 (17%)	283518 (32%)	261999 (60%)
OURS	2944 (81%)	8.65 (96%)	1654 (56%)	395903 (45%)	387651 (89%)

6 CONCLUSION

In this work, we propose LCP, a Layer Clusters Paralleling mapping method to classify the layers into clusters based on their differences of configurations and data localities, and then accelerates them in different partitions of FPGA. Additional cross-branch/layer optimizations are combined into the mapping framework to further improve resources utilization. We realize implementations on Xilinx VC709 board which outperform all previous work.

7 ACKNOWLEDGMENT

This work was supported in part by the NSFC under Grant 61774094, the China National High Technologies Research Program (No.2015A A016601) and the China Major S&T Project (No.2013ZX01033001-001-003).

REFERENCES

[1] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Ieee/acm International Symposium on Microarchitecture.* 1–12.
[2] G. H. Ball and D. J. Hall. 1965. *ISODATA, a novel method of data analysis and pattern classification.* Stanford Research Institute, Menlo Park, California.
[3] Zhang Chen, Fang Zhenman, Zhou Peipei, Pan Peichen, and Cong Jason. 2016. Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. In *ICCAD.*
[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. *computer vision and pattern recognition* (2016), 770–778.
[5] Atul Rahman, Jongeun Lee, and Kiyoung Choi. 2016. Efficient FPGA acceleration of Convolutional Neural Networks using logical-3D compute array. In *Design, Automation & Test in Europe Conference & Exhibition.* 1393–1398.
[6] Yongming Shen, Michael Ferdman, and Peter A Milder. 2017. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. *international symposium on computer architecture* (2017), 535–547.
[7] D Silver, J Schrittwieser, K Simonyan, I Antonoglou, A. Huang, A Guez, T Hubert, L Baker, M. Lai, and A Bolton. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354.
[8] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Computer Vision and Pattern Recognition.* 1–9.
[9] Samuel Williams, Andrew Waterman, and David Patterson. 2009. *Roofline: an insightful visual performance model for multicore architectures.* ACM. 65–76 pages.
[10] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Acm/sigda International Symposium on Field-Programmable Gate Arrays.* 161–170.