A Collaborative Framework for FPGA-based CNN Design Modeling and Optimization

Jiandong Mu\textsuperscript{1,2}, Wei Zhang\textsuperscript{2}, Hao Liang\textsuperscript{3}, Sharad Sinha\textsuperscript{4}  
\textsuperscript{1,2}Hong Kong University of Science and Technology (HKUST), China  
\textsuperscript{3}Alibaba Group, China  
\textsuperscript{4}Indian Institute of Technology (IIT), Goa  
jmu@connect.ust.hk, wei.zhang@ust.hk, sharad@iitgoa.ac.in

Abstract—Convolutional neural network (CNN) has presented a great success in numerous areas and has sparked an increasing interest in accelerating CNN using hardware like FPGAs. However, efficient FPGA design for CNN applications requires a long development time and a strong background in hardware details. Consequently, an easy-to-use yet powerful auto CNN design optimization framework is required. In this work, we propose a collaborative framework to model and optimize the OpenCL based FPGA design for CNN applications according to the device resource limitation and the CNN specification. Our framework mainly consists of LoopTree, a novel data structure we propose to capture the structure of OpenCL based CNN design; a LoopTree based coarse-grained model, which will estimate the performance of the CNN design at the module level; and a source code based fine-grained model, which will estimate the CNN design performance in a cycle-accurate manner. Efficient designs can be achieved by collaborating the two models in a search and refined manner. A variety of OpenCL based designs have been implemented on board to verify our framework. The results show that our coarse-grained model and fine-grained model have an average estimation error of 10.2\% and 4.7\% which are much lower than prevalent operation statistics based estimation calculated by the predefined formula for specific loop schedules.

Keywords-CNN, FPGA, Modeling, Collaborative Framework.

I. INTRODUCTION

Deep neural network (DNN) has become one of the most popular algorithms for its high accuracy in many applications ranging from object detection and image classification to speech recognition. As one type of the most important DNNs, the acceleration of CNNs on platforms like multi-core CPU, GPU and FPGA is of immense interest these days. Among all of these, FPGA is one of the most promising platforms due to its short latency, high throughput and energy efficiency.

However, FPGA design has long been accused of its high requirements of a strong background in hardware design and detailed knowledge of target devices. This has greatly impeded FPGA from its broader application. Although high level synthesis (HLS) based on C, C++, OpenCL has been developed to ease this design process, it still remains a big challenge for new users to design efficient FPGA based implementations for CNN.

In addition, hardware design for CNN applications has a large design space. Design specifications such as tiling factors, memory layout and loop orders (in high level hardware design languages like HLS and OpenCL) can affect the design performance significantly. As a result, finding the optimal design from the huge designing pool becomes a non-trivial problem. Current approaches mainly focus on tuning the CNN design in a smaller design space with fixed loop orders and HLS pragmas (interchangeable with attributes and directive in this work), therefore, the optimal design might be missed.

Given the analysis above, we observe that there is a strong demand for an easy-to-use yet powerful framework to optimize FPGA designs for CNN. In this paper, we aim to solve this problem by employing a collaborative framework to model and optimize OpenCL based FPGA designs for CNN applications automatically. In our framework, the CNN designs are optimized in 3 steps. First, our model automatically forms a comprehensive CNN design space based on LoopTree. Hardware design specifications like loop orders, loop tiling, BRAM and DDR configurations and OpenCL attributes are taken into account. Then, we estimate the LoopTree execution time and resource usage by using a coarse-grained model. LoopTrees with the best resource usage and execution time tradeoff are selected as the candidate designs. Finally, an OpenCL source code based fine-grained model is applied to fine tune the selected designs so that the optimal design can be obtained.

Overall, we make the following contributions:

- A new data structure, LoopTree, is proposed to capture the structure of an OpenCL based FPGA design for CNN applications.
- A more comprehensive design space is created based on LoopTree. Factors including loop orders, loop tiling, BRAM and DDR configurations and OpenCL attributes are searched for more efficient designs.
- A coarse-grained model is proposed to estimate the performance of LoopTree by considering modules like computing unit, BRAM and DDR.
- An OpenCL source code based fine-grained model is proposed to estimate CNN design in a cycle-accurate manner. Four performance advisors are introduced to guide the tuning process.

The remaining paper is organized as follows. Section II presents the related work about CNN design space exploration and OpenCL source code profiling. Section III introduces the overview of our proposed framework and the design space generation. Section IV and section V introduce the coarse-grained model and fine-grained model, respectively. Section VI shows the experiment results of our framework, and Section VII concludes the paper.

II. RELATED WORKS

FPGA based hardware optimization for CNNs has been a hot topic recently. In order to pursue a better performance, many works have been done on optimizing FPGA-based CNN designs via design space exploration. Existing approaches mainly...
focus on hardware parameters tuning and fail to explore all possible configurations which may lead to a suboptimal design. For example, in [11], an adder tree based parallel convolution engine (PCE) is proposed to compute convolutions efficiently. A set of tiling factors $T_K$, $T_m$, $T_n$, $T_r$, $T_c$, and $T_t$ are tuned to maximize the performance. However, the loop orders and memory configurations remain unexplored. In [2], only loop interchanges between the 4 outermost loops and attributes for the inner loops are considered in their DSE.

In addition, previous works estimate each design based on the predefined formula for a specific loop schedule, which may not be accurate enough to tell the priority of different configurations. For example, in [11][3] and [4], the number of the execution cycles is evaluated by counting the MAC operations according to the specific loop schedule. However, the contributions of other control logics and data transfers are not considered. This estimation method also requires the users to specify the loop schedule manually and it is not easy to pick a good design from a huge number of the possible loop schedules in the design space. In [2] and [5], BRAM and DDR access time have been taken into consideration. However, the loop initiation interval is assumed to be 1, which may not hold due to resource conflicts. Perfect loop flattening, BRAM partition and control logic are also not considered in their works. In this work, we try to consider all such overheads presented in the source code.

Performance analysis frameworks for OpenCL have been proposed in [6] and [7] to guide the OpenCL-based hardware design. However, we found the frameworks cannot be directly applied to CNN applications for several reasons. Firstly, their works are based on source code which may not be available for new users of CNN and OpenCL. Secondly, the proposed bandwidth models are based on access patterns and instructions statistics. However, the properties of the device are neglected. Finally, the proposed frameworks optimize the OpenCL applications through attributes tuning. Other optimizing methodologies in CNN-specific areas are not considered since their frameworks target general OpenCL applications. Therefore, their frameworks cannot be directly used to generate the efficient OpenCL designs for CNNs.

III. PROPOSED FRAMEWORK

The overall framework is shown in Fig. 1. In this framework, we aim to find efficient OpenCL based hardware designs for CNN according the board parameters provided by the users. To achieve this, a comprehensive design space of LoopTrees, which represent the configurations of OpenCL based designs

Figure 1: Collaborative framework for FPGA-based design generation and optimization

A. LoopTree: A Proposed Data Structure for CNN

We consider a convolutional layer with tiling and data buffering as illustrated in Fig. 2 but it could be any number of loops when using our proposed framework. We name the loop after corresponding iteration variable in this work for convenience. For example, $loop_j$ and $loop_i$, represent the loops in the filter dimension.

A new data structure, LoopTree, is built to represent this loop hierarchy. We employ a set of nodes to portray the loop schedule as illustrated in Fig. 3(a). Each node represents a loop. The name of the node is the same as the loop variable, and the color of the node stands for the operation inside the loop. The child node indicates a subloop of the current loop. We characterize the computation, DDR and BRAM’s loop schedules as branches of the LoopTree. In this example, 4 branches are shown in the figure to represent the input, weight, output buffer and MAC operation respectively. Please note that the LoopTree might have more than one root nodes since the memory and computing unit might have independent loop hierarchy.

Listing 1: Loops example for a typical convolutional layer with tiling and buffering.

```plaintext
1 for $m \leftarrow 0$ to $M$ step $= T_m$ do
2 for $i, j \leftarrow 0$ to $K$; $t_n \leftarrow 0$ to $T_n$; $n \leftarrow 0$ to $N$ do
3 \hspace{1em} \text{load}(W)
4 for $x \leftarrow 0$ to $X$ step $= T_x$; $y \leftarrow 0$ to $Y$ step $= T_y$ do
5 \hspace{2em} \text{load}(I)
6 \hspace{3em} for ($t_c, t_y \leftarrow 0$ to $T_c$; $t_n \leftarrow 0$ to $T_n$; $n \leftarrow 0$ to $T_n$ do
7 \hspace{4em} \text{O}[t_m][t_c][t_y] = W[t_m][t_c][t_y] \times I[t_m][t_c][t_n] + \text{S}[t_c][t_n] + j]
8 \hspace{2em} \text{store}(O)
```

Figure 2: Loops example for a typical convolutional layer with tiling and buffering.

Figure 3: (a) LoopTree for a typical convolutional layer as shown in Fig. 2 (b) Memory configurations with the same loop order but different branch positions. (c) LoopTree after pruning.

for CNNs, is automatically generated at the first step. This is followed by a design space pruning process since our design space is huge due to its comprehensiveness. Then, the coarse-grained model is employed to estimate the resource and latency of the LoopTree. After that, efficient candidates selected according to the user-specified objective function are forwarded to the fine-grained model to receive further optimization.

A convolutional layer is employed as an example to illustrate our framework. Note that our framework can also be applied to other layers like pooling layer, fully connected layer, non-linear layer, fused layers etc.

A convolutional layer with tiling and data buffering as illustrated in Fig. 2 but it could be any number of loops when using our proposed framework. We name the loop after corresponding iteration variable in this work for convenience. For example, $loop_j$ and $loop_i$, represent the loops in the filter dimension.

A new data structure, LoopTree, is built to represent this loop hierarchy. We employ a set of nodes to portray the loop schedule as illustrated in Fig. 3(a). Each node represents a loop. The name of the node is the same as the loop variable, and the color of the node stands for the operation inside the loop. The child node indicates a subloop of the current loop. We characterize the computation, DDR and BRAM’s loop schedules as branches of the LoopTree. In this example, 4 branches are shown in the figure to represent the input, weight, output buffer and MAC operation respectively. Please note that the LoopTree might have more than one root nodes since the memory and computing unit might have independent loop hierarchy.
B. LoopTree Generation

The design space of the LoopTrees is automatically generated by exploring the following dimensions:

1) Loop Order: In light of the fact that different loop schedules have a significant impact on the CNN computing architecture, the loop ordering problem becomes nontrivial. In order to include all configurations, we try to explore all possible loop node permutations for computing unit and memory branches exhaustively.

Furthermore, LoopTree also varies as the tree branches at different nodes, although they share the same loop order, as shown in Fig. 3(b). The new LoopTree’s performance can be different from the original one since different branching nodes have distinct loop flattening results, BRAM size and BRAM partitions. This indicates another design flexibility and will further enlarge the design space in an exponential way. However, as we will show later, the design space can be dramatically reduced as our proposed pruning scheme is applied.

2) OpenCL Attributes: OpenCL has offered plenty of design attributes like loop unrolling, loop pipelining and array partitioning to ease the hardware designing process. In our framework, loop unrolling and loop pipelining have been considered for each loop. Array partitioning is applied to BRAM and is automatically selected to minimize the data access conflicts and thus maximize the design performance. Req’d work group size is set to (1, 1, 1) for our Xilinx FPGA since the best performance for the Xilinx board is achieved by one work-item per compute unit [8].

3) Loop Tiling: Loop tiling has been explored for CNN design optimization in many papers [2, 1, 4, 9, 10]. In this work, tiling in input channel dimension ($T_{in}$), input width and height dimension ($T_{lin}$) and output channel dimension ($T_{on}$) are considered.

C. Design Space Pruning

Due to the comprehensiveness of the framework, our design space considered is huge. As a consequence, a pruning scheme is applied to make the design space exploration practical. Several pruning rules have been proposed to shrink the design space.

1) Node Reduction: A major reason for the huge design space is the high loop hierarchy hence the design space can be narrowed down significantly by reducing the nodes in the LoopTree. This can be achieved by merging multiple nodes into a single node. For example, $loop_i$ and $loop_j$ stand for iterations through kernel width and height dimensions. These two loops can be merged into a single loop because they are likely to share the same OpenCL attributes and similar loop hierarchy according to the symmetry. The same reason also holds for loop pair $loop_{tx}$, $loop_{ty}$ and loop pair $loop_x$, $loop_y$.

2) Loop Pre-order: Not all the node orders of the LoopTree are valid for CNN applications. For example, $loop_n$ should be placed at a higher node in the LoopTree than its tiling counterpart $loop_{tn}$. Illegal orders are eliminated to save computing time and resources.

3) Loop Pipelining: For high level synthesis tools from companies like Xilinx, loops inside of the pipelined loop will be unrolled completely [11]. As a consequence, attributes exploration for loops inside the pipelined loop is not necessary in the DSE process. Unrolling factors are directly set to be equal to the loop trip counts.

4) Early Termination Rule: The design space generation process can be divided into several stages including specifying the tiling factors, LoopTree node orders, OpenCL attributes and so on. The latter stages will be automatically skipped if the previous stages have consumed more resources than the board owns.

The LoopTree for previous example after pruning is shown in Fig. 3(c). By applying the aforementioned pruning schemes, the design space of the LoopTree can be shrunk to a size computationally acceptable for current CPUs like Intel i7-7700K used in our experiments. Then the LoopTrees will be forwarded to the coarse-grained model which is presented in the following section.

Please note that this LoopTree representation is generic for loop based OpenCL designs. Convolutional designs other than the aforementioned example and other layers like pooling layer, fully connected layer, non-linear layer, fused layer can also be represented using LoopTree. The LoopTrees will be automatically generated considering the input, weight, output buffering by default. However, this can be handcrafted if users prefer feeding data in other ways. Other operations can also be manually taken into account by adding more branches.

IV. COARSE-GRAINED MODEL

Our coarse-grained model will evaluate the LoopTree execution time by evaluating the time for the computation unit, BRAM and DDR. In this section, we will demonstrate our coarse-grained model using the previous example in Fig. 3(c) for simplicity, although our model can handle more general cases.

A. Computing Unit

In CNN applications, computing time is dominated by the MAC operation which is indicated by the blue leaf node in the LoopTree in Fig. 3(c). Without losing generality, we assume the loop nodes for the MAC operation have trip counts of $C = \{C_m, C_{xy}, C_{tn}, C_{tx}, C_{ty}, C_{tm}, C_{tn}\}$ and unrolling factors of $U = \{U_m, U_{xy}, U_n, U_{tx}, U_{ty}, U_{tm}, U_{tn}\}$. A loop pipelining attribute is applied to the $loop_{ij}$ node. Then, we evaluate the time consumed by the MAC operation in the following steps.

1) II Estimation: Loop initiation interval (II), which refers to the number of clock cycles between the start times of consecutive loop iterations, is one of the most important indicators of the design throughput and latency. II is constrained by loop recurrent dependency and resource limitation and is therefore evaluated by:

$$II = \max \{RecII, ResII\}$$

where $RecII$ is the recurrence constrained II and $ResII$ is the resource constrained II. $RecII$ is closely dependent on the source code and is therefore impossible to be estimated according to the LoopTree. However, sophisticated architectures like systolic array [10] and adder tree [11] can resolve the dependencies and lead to an optimal $RecII$ of 1, which has an overwhelming advantage in throughput and latency over their $RecII > 1$ counterparts. As a result, we only focus on cases with $RecII = 1$ in our framework. The $ResII$ is dominated.
by conflicting access to the BRAM and DDR. The calculation is shown in later sections.

2) Perfect Loops: Nested loops with no logic specified between the loop statements are considered perfect loops [12]. Perfect loops can be flattened to improve the FPGA design throughput and reduce the execution time by covering the pipeline filling and emptying time. This is especially important for loops with high latency and low trip counts. In our coarse-grained model, we consider the loop node with more than one children nodes as an imperfect loop. The OpenCL loop unrolling attribute can also spoil the perfect loop by splitting one child node into several children nodes.

3) Latency Calculation: In light of the aforementioned pruning rules, the subloops of the pipelined loop are automatically completely unrolled. Consequently, the iteration latency for loop$_{ij}$ is $L_{mac}$, which stands for the latency of MAC operation. Then loop$_{ij}$ and loop$_{kxy}$ can be flattened into a single loop with a trip count of $C_{ij}C_{k}$ if $U_{kxy} = 1$. The resulting latency of the computing unit in Fig. 3(c) can be calculated in a unified expression:

$$L_{comp} = \left( \prod \frac{C_{r}}{U_{r}} - 1 \right) II + L_{mac} \prod s C_{s}$$

where $C_{r}, U_{r}$ are trip counts and unrolling factors for perfect loops nested to the pipelined loop, $C_{s}$ and $U_{s}$ are the trip counts and unrolling factors, respectively, for the other outer loops. $II$ is the initiation interval of the pipelined loop. The unrolling factors $U_{s}$ are eliminated because the Xilinx compiler does not schedule loops in parallel [11] which will diminish the unrolling effect.

In CNN applications, DSP usage is dominated by the MAC operation, thus can be estimated as

$$Res_{DSP} = \prod_{loops} U_{mac}^{loops}$$

where $\prod_{loop}$ denotes the product of all the loops that contain the MAC operation. This notation is also used in the following sections unless there is a specific interpretation. $Res_{DSP}$ serves as an important indicator in early termination rule. Design with DSP usage greater or less than the thresholds will be dropped in the pruning process.

B. BRAM Utilization and Timing Estimation

In this example, we consider input, weight and output buffers for timing and resources contribution although our framework can be easily extended to more buffers.

The BRAM size is determined by the loop nodes between the BRAM and the branch node, as we colored in Fig. 3. The address space formed by these loop variables indicates the least buffer size required to pass the data to other branches. In this example, the buffer size for 3 BRAMs are $K^{2}T_{m}N$, $T_{lin}^{2}T_{n}$ and $T^{2}T_{m}$ respectively.

As one of the most important indicators in FPGA design, enormous efforts have been made to make $II$ equal to 1 so that the throughput can be maximized and the latency can be minimized. Inspired by this fact, we partition the BRAM automatically to avoid the data access conflicts and to minimize

$$P_{bram,var} = \min \left\{ \prod_{loops} U_{bram,var}^{loops}, P_{max} \right\}$$

where $bram$ indicates different BRAMs, which are input, weight, output respectively in this example. $U_{bram,var}$ is the unrolling factor of loop with iteration variable $var$. $P_{bram,var}$ is the partition factor of $bram$ in dimension $var$. $P_{max}$ is the port number limitation constrained by users. The partition type is automatically selected as cyclic partition suits the continuous memory address and block partition suits the jumping memory address. Complete partition will be applied when the partition factor in dimension $var$ equals its BRAM size.

However, $II > 1$ still may happen due to the limited BRAM ports. We model the $II$ caused by BRAM as the following equation:

$$II_{bram} = \max_{var} \left\{ \frac{\prod_{loops} U_{bram,var}^{loops}}{P_{bram,var}} \right\}$$

Then the BRAM data buffering time can be calculated in a similar way as the computing unit we mentioned previously.

$$L_{bram} = \left( \prod \frac{C_{r}}{U_{r}} - 1 \right) II + L_{bram} \prod s C_{s}$$

where $C_{r}, U_{r}, C_{s}, U_{s}$ has the same meaning as their counterparts in the previous section. $II$ is the initiation interval of the pipelined loop of the corresponding memory branch and is constrained by the larger one of $II_{bram}$ and $II_{ddr}$, which we will introduce later.

C. DDR Timing Estimation

DDR access is sensitive to burst transactions and bit-widths [13]. Therefore, in our model, we test the bandwidth vs burst transactions and bit-widths on an ADM-PCIe-7V3 board using SDAccel 2016.2 from the Xilinx company. The result is shown in Fig. 5. We observed that a minimum 64 burst transactions and a bit-width of int16 (512bits) are required to access DDR memory at peak speed.

DDR access also puts a restriction in the loop II due to the limitation of the physical DDR port number. For DDR memory
with $B$ banks, the resulting II can be estimated by

$$II_{ddr} = \frac{1}{B} \prod_{\text{loops}} U_{ddr}^{\text{loops}} \times N_{ddr}$$

where $U_{\text{loops}, ddr}$ is the unrolling factors of the DDR access and $N_{ddr}$ is the number of DDR accesses in the code.

The execution time caused by $ddr$ is estimated by the bandwidth model together with the LoopTree:

$$t_{ddr} = \prod_{\text{loops}} C_{\text{loops}} \frac{\text{Bytes}_{ddr}}{BW_{\text{bits,burstlen}}}$$

where $\text{Bytes}_{ddr}$ indicates the data amount in bytes per transaction. The $BW_{\text{bits,burstlen}}$ is the bandwidth with respect to corresponding bit-widths and burst length. The bit-widths are determined by the transaction data type and the burst length can be obtained from loop node variable and trip count specifications. $\prod_{\text{loops}} C_{\text{loops}} \text{Bytes}_{ddr}$ is used instead of the global memory size in order to include cases with repeated global memory accesses.

D. Putting It All Together

We estimate the LoopTree performance by taking all aforementioned facts into consideration. The $II$ of each branch is given by the maximum $II$ of the corresponding computing unit, BRAM, and DDR. The latency of each branch is then estimated using previous formula considering the facts that the computing units, buffers, and memory accesses might be covered if they are pipelined in the same branch. Finally, the throughput of the LoopTree is obtained according to the total time, since different LoopTrees are not pipelined in our framework.

After the LoopTrees in the design pool are evaluated using the coarse-grained model, the candidate designs are selected according to the user-specified object function to meet the requirements in different scenarios. 2 commonly used selecting rules are implemented in our framework: (1) minimum running time. (2) minimum time-area product. We use the weighted area mentioned in [14] as the area indicator.

V. FINE-GRAINED MODEL

By employing the design space generation and the coarse-grained model we can select a batch of candidate LoopTrees for the CNN application. OpenCL code can be produced based on code templates and libraries since our LoopTree has a comprehensive and detailed description of the target design. Then our fine-grained model fine tunes the OpenCL source code to generate an optimal design.

A. Source Code Based Estimation

The final execution time can be underestimated by the LoopTree based coarse-grained model since the control logic is omitted in the estimation. Further optimization can be achieved by getting more accurate profiling information, thus a cycle accurate, LLVM intermediate representation (IR) based fine-grained model is proposed in our framework.

After the OpenCL source code, Fig. 6(a), is imported into the fine-grained model, an LLVM [15] front end is employed to parse the OpenCL code into the LLVM IR. Then the control flow graph (CFG), which is an alternative representation of the original OpenCL code, is constructed in a recursive way, as described in [6][7]. The CFG consists of a set of basic blocks and arrows between them indicating dependencies between the connected basic blocks. This is illustrated in Fig. 6(b).

The latency of the OpenCL application is estimated by evaluating the latency of the CFG. First of all, the latency of each basic block is estimated by scheduling the instructions in an ASAP policy, as illustrated in Fig. 6(c). Then the latency of the CFG is evaluated based on the basic blocks, dependencies and corresponding OpenCL attributes, which has been well investigated in [6][7]. In addition to the CFG latency evaluation, our fine grained model also supports the onboard measurements based global memory model mentioned in the previous section. This differentiates our model from the pattern matching method in [11] and statistical analysis in [6]. Although the bandwidth library varies from board to board, our bandwidth model provides an accurate DDR timing prediction and beneficial guidance for CNN memory layout design.

In contrast to the LoopTree based coarse-grained model, our fine-grained model estimates the code performance at the LLVM instruction level. The contribution of each logic element is considered so that the fine-grained model has a higher accuracy than the coarse-grained model. Consequently, more optimizations can be conducted based on the fine-grained model.

B. Performance Advisor

Recall that one of the key purposes of this work is to assist OpenCL design for CNN applications. Therefore, we propose 4 performance advisors to quantify the design performance and identify the design bottleneck.
Table I: Convolutional layer benchmark configurations

<table>
<thead>
<tr>
<th>config</th>
<th>memory</th>
<th>computing unit</th>
<th>kernels</th>
<th>parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>non-coalesce</td>
<td>systolic</td>
<td>single</td>
<td>256</td>
</tr>
<tr>
<td>4-6</td>
<td>coalesce</td>
<td>systolic</td>
<td>single</td>
<td>256</td>
</tr>
<tr>
<td>7-9</td>
<td>coalesce</td>
<td>systolic</td>
<td>multiple</td>
<td>256</td>
</tr>
<tr>
<td>10-12</td>
<td>coalesce</td>
<td>systolic</td>
<td>multiple</td>
<td>512</td>
</tr>
<tr>
<td>13-15</td>
<td>coalesce</td>
<td>systolic</td>
<td>multiple</td>
<td>1024</td>
</tr>
<tr>
<td>15-18</td>
<td>coalesce</td>
<td>adder tree</td>
<td>multiple</td>
<td>256</td>
</tr>
<tr>
<td>19-20</td>
<td>coalesce</td>
<td>adder tree</td>
<td>multiple</td>
<td>1024</td>
</tr>
</tbody>
</table>

1) $P_{bw}$ stands for the potential from increasing the bandwidth usage. We estimate $P_{bw}$ by calculating the average bandwidth utilization:

$$P_{bw} = 1 - \frac{\sum_{ddr} Bytes_{ddr} \times BW_{ddr}}{\sum_{ddr} Bytes_{ddr} \times PBW_{ddr}}$$

where $Bytes_{ddr}$ is the data size transferred from $ddr$, which is one of all global memories. $BW_{ddr}$ is the bandwidth for $ddr$ and $PBW_{ddr}$ is the peak bandwidth for $ddr$. A higher $P_{bw}(0 < P_{bw} < 1)$ indicates the design can benefit more from a longer global memory coalesce length, more transaction bit-widths or a DDR with a higher bandwidth.

2) $P_{rep}$ represents the potential from reducing the repeated data accesses to the global memory, which is caused by limited onchip buffer size or impropor memory layout. In our model, $P_{rep}$ is calculated by global memory size over total transferred data size:

$$P_{rep} = 1 - \frac{\sum_{ddr} MemSize_{ddr}}{\sum_{ddr} MemSize_{ddr} \times C_{loops,ddr} \times Bytes_{ddr}}$$

where $MemSize_{ddr}$ is the global memory size for $ddr$ and $C_{loops,ddr}$ is the trip counts for loops that contains $ddr$. $P_{rep}$ also varies from 0 to 1. A close to 1 $P_{rep}$ indicates a highly repeated DDR accessing pattern; therefore, an appropriate memory layout should be considered to reduce the DDR transactions.

3) $P_{tile}$ serves as an indicator of the overhead of the edge effect. This overhead will shrink as tile size increases:

$$P_{tile} = \left(1 - \frac{T_i}{(T_i \times stride + 2 \times padding)^2}\right) \times \frac{t_{bram, in}}{t_{exe}}$$

where $t_{bram, in}$ is the time consumed by input data buffering.

4) $P_{overhead}$ stands for the overhead of non-MAC operations like control logic and memory accesses. This parameter is estimated by predicted throughput over theoretical peak throughput:

$$P_{overhead} = 1 - \frac{\#ops}{t_{exe} \times \#P \times F_{req}}$$

$\#ops$ is the number of operations, which is $N \times M \times image\_size^2 \times K^2$ for the convolutional layer in this example. $\#P$ is the parallelism of the design. A high $P_{overhead}$ indicates non-MAC logic, like memory accesses and control logic, have consumed too much time or the computing loop II is larger than 1. The $P_{overhead}$ can be released by applying multiple kernels to hide the non-MAC operation latency, using a more appropriate computing architecture like systolic array to make the computing II smaller, or increasing the number of computing units.

Table II: Pooling layer benchmark configurations

<table>
<thead>
<tr>
<th>config</th>
<th>memory layout</th>
<th>pooling type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-9</td>
<td>non-coalesce</td>
<td>max pooling</td>
</tr>
<tr>
<td>10-18</td>
<td>coalesce</td>
<td>max pooling</td>
</tr>
<tr>
<td>19-27</td>
<td>non-coalesce</td>
<td>ave pooling</td>
</tr>
<tr>
<td>28-36</td>
<td>coalesce</td>
<td>ave pooling</td>
</tr>
</tbody>
</table>

VI. EXPERIMENTAL EVALUATION OF PROPOSED FRAMEWORK

Our experiments are conducted on an ADM-PCIE-7V3 board from Alpha Data company with a Virtex 7 (XC7VX690T) FPGA from Xilinx company and two 8 GB ECC-SODIMMs for memory speeds up to 1333 MT/s. The alphadata board is connected to host via a PCI-e 3.0 X8 interface. The host binary file is running on an Intel(R) Core(TM) i7-7700K CPU@4.2GHz. The SDAccel 2016.2 tool flow from Xilinx is used to implement the OpenCL design and program the FPGA.

A. Experiment Benchmark

To authenticate our framework, we test our framework in a layer by layer manner. 20 convolutional layer and 36 pooling layer OpenCL based designs are implemented on board to verify the accuracy of our framework. The configurations of the benchmark are shown in Table I and II. Different design features including memory coalescing, computing architecture and multi-kernel are considered in our benchmarks. The memory layout in [13] is implemented as coalescing designs in configuration 4-20. Three non-coalescing designs of different tile sizes are provided in configuration 1-3 for comparison. The systolic array architecture proposed in [10] and adder tree structure used in [11] and [9] are implemented as two commonly used computing units in configuration 1-15 and 16-20, respectively. Multi-kernel based designs are implemented in configuration 7-20. Six single kernel based designs are also provided for comparison in configuration 1-6.

Non-linear layers like ReLU are quite simple and can be fused into other layers. They have limited contribution to the final time consumption. Fully connected layer shares a similar architecture with convolutional layer. As a result, the activation layer and fully connected layer are not necessarily needed since the provided benchmarks are already sufficient to verify the model accuracy.

B. Evaluation of Coarse-grained Model

Fig. 7(a,c) plots the onboard running throughput, coarse-grained and fine-grained model predicted throughput and the MAC operation statistics based throughput estimation for the convolutional layer and pooling layer, respectively. Fig. 7(b,d) plots the estimation error for latency estimation. The experiments show that operation statistics based method will overestimate the throughput significantly. Our coarse-grained model can estimate the performance with an average error of 10.2% which is much more precise than the MAC operation analysis.

We also observe that the coarse-grained model will slightly overestimate the throughput because the coarse-grained model mainly takes the data buffering, DDR access and computing unit into account. The time consumed by control logics may be omitted. However, we will see that this is fixed in our fine-grained model which is based on source code.
C. Evaluation of Fine-grained Model

The coarse-grained model has an overwhelming advantage in running speed. However, the coarse-grained model may fail to tell the difference between detailed hardware architectures since it does not depend on the source code. On the contrary, our fine-grained model can measure minor differences between different architectures. Fig. 7 shows that our fine-grained model can estimate the throughput with an average error of 4.7%. Fig. 8(a) plots the performance of the systolic array architecture and adder tree architecture. As is shown in the figure, our fine-grained model is able to tell the difference between the two structures although they have a similar performance. The performance comparison between the max pooling layer and average pooling layer is shown in Fig. 8(b). The average pooling layer has a slightly higher execution time than the max pooling layer because the average pooling has $DIV$ and accumulation operation that takes time in hardware execution. Note that we can implement $DIV$ using a hash table to improve the pooling layer performance. We did not do this for the reason that we show that our fine-grained model is able to tell the existence of high latency operations.

D. Evaluation of Design Advisors

Four design advisors are proposed to guide the fine-grained model tuning. Two case studies are shown in Fig. 8(c,d).

Fig. 8(c) plots the advisors guided design for the convolutional layer. We first notice that $P_{tile}$, which indicate the potential of the tiling size, is small. The performance increases 30.2% as we increase the tiling size from 8 to 28. $P_{tile}$ drops in this process as the potential of the tiling size has been exploited. Then we can increase the performance by increasing the DDR bandwidth usage since $P_{bw}$ is high. This is achieved by applying a coalescing memory layout. Finally, the control logic overhead indicated by $P_{overhead}$ can be hidden by using multiple kernels and deploying more DSPs, and a total speed-up of 7.199 can be obtained.

In Fig. 8(d), the hardware design for the pooling layer can be optimized in a similar way. Firstly, we observe that the potential of the tiling size and data replication is not much while the potential of the bandwidth usage and control logic overhead is high. This inspired us to speed up the design by increasing parallelism to reduce the logic overhead, and using coalescing memory layout to fully utilize the bandwidth. Finally, a total speed-up of 13.032 is achieved.

E. Proposed Hardware Design

In this section, we generated the design for VGG16-SVD based on our Xilinx Virtex-7 690T board. In comparison with RTL design, OpenCL design is less efficient in resource usage and routing. The minimum time-area product selecting rule turns out to be a better choice to get a good tradeoff between the routing ability and the performance. The proposed hardware is paralleled in the $Loop_{tileij}$ (input width and height) and $Loop_{pem}$ (output channel) dimensions. Our framework suggests that paralleling in these two dimensions can significantly reduce the BRAM port usage and ease the routing process. Furthermore, according to the design space exploration, the loop trip count for $Loop_{pem}$ is set as 32, and the trip count for $Loop_{tileij}$ is set as 64. A total of 2060 DSPs are utilized to process the data efficiently. Memory layout in [13] is applied in the design to maximize the DDR bandwidth. Our coarse-grained model also suggests that a fused implementation of convolutional, ReLU and pooling layer can significantly reduce the memory transfer time. Finally, an overall performance of 274.24 GOPS is achieved.

We have listed a few designs based on OpenCL as is shown in Table III. In particular, to make a fair comparison, we compare our design with [18], in which the same board and same tool flow are used. It turns out that our design can achieve better performances than theirs even when the differences in data type are taken into account. We are also aware of the performance gap between our design and the work in [19], and we think the gap is resulted from two reasons. One reason is that Arril 10 and Altera OpenCL SDK they used have significant advantages
Table III: Performance Comparison with other OpenCL-based hardware implementation

<table>
<thead>
<tr>
<th>Tool flow</th>
<th>FPGA</th>
<th>DSP</th>
<th>CNN</th>
<th>Precision</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA2016 [16]</td>
<td>Altera OpenCL SDK</td>
<td>Stratix-V GSD8</td>
<td>120MHz</td>
<td>VGG16</td>
<td>fixed(8-16b)</td>
</tr>
<tr>
<td>PipeCNN [17]</td>
<td>Altera OpenCL SDK</td>
<td>Stratix-V GXA7</td>
<td>181MHz</td>
<td>AlexNet</td>
<td>float</td>
</tr>
<tr>
<td>FP716 [18]</td>
<td>Xilinx SDAccel</td>
<td>Vortex-7 690T</td>
<td>200MHz</td>
<td>float</td>
<td>50 GOPS</td>
</tr>
<tr>
<td>FPGA2017 [19]</td>
<td>Altera OpenCL SDK</td>
<td>Stratix-V GXA7</td>
<td>385MHz</td>
<td>VGG16</td>
<td>fixed16</td>
</tr>
<tr>
<td>Our work</td>
<td>Xilinx SDAccel</td>
<td>Vortex-7 690T</td>
<td>160MHz</td>
<td>VGG16</td>
<td>fixed16</td>
</tr>
</tbody>
</table>

in the frequency and the DSP efficiency. The other reason is that we optimize the time-area product in the process of design for a better routability, and therefore, the resulted design may not be optimal for throughput.

VII. CONCLUSION

In this work, we propose a collaborative framework to optimize the OpenCL based CNN design. A new data structure, LoopTree, is proposed to capture the main features of OpenCL based hardware design. Firstly, LoopTrees are automatically generated to form a comprehensive design space. Then a coarse-grained model is employed to evaluate the LoopTree and to find candidate designs. Finally, a fine-grained model is used to tune the candidate designs under the guidance of performance advisors. The experiments confirm the accuracy of our framework and show that our framework can achieve efficient OpenCL based FPGA designs for CNN applications.

REFERENCES

[12] Xilinx Vivado design suite user guide.