HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation

Hanchen Ye[1], Xiaofan Zhang[1], Zhize Huang[2], Gengsheng Chen[2], Deming Chen[1]
[1]University of Illinois at Urbana-Champaign, [2]Fudan University
{hanchen8, xiaofan3, dchen}@illinois.edu, {18212020085, gschen}@fudan.edu.cn

ABSTRACT

To speedup Deep Neural Networks (DNN) accelerator design and enable effective implementation, we propose HybridDNN, a framework for building high-performance hybrid DNN accelerators and delivering FPGA-based hardware implementations. Novel techniques include a highly flexible and scalable architecture with a hybrid Spatial/Winograd convolution (CONV) Processing Engine (PE), a comprehensive design space exploration tool, and a complete design flow to fully support accelerator design and implementation. Experimental results show that the accelerators generated by HybridDNN can deliver 3375.7 and 83.3 GOPS on a high-end FPGA (VU9P) and an embedded FPGA (PYNQ-Z1), respectively, which achieve a 1.8x higher performance improvement compared to the state-of-art accelerator designs. This demonstrates that HybridDNN is flexible and scalable and can target both cloud and embedded hardware platforms with vastly different resource constraints.

1 INTRODUCTION

With deeper and more complicated layer connections, DNNs are becoming more compute- and memory-intensive, which require hardware accelerators to deliver high throughput, low end-to-end latency, and high energy efficiency. Recently, researchers have focused on building customized DNN accelerators by taking advantage of different hardware devices, including GPUs, FPGAs, and ASICs, to improve the speed and efficiency of DNN inference [1–8]. By considering the accelerator deployment for real-life AI applications, energy-hungry GPU-based accelerators are difficult to meet the energy/power constraints while the ASIC-based designs require a long time-to-market period. FPGAs, therefore, become promising candidates for DNN implementations with improved latency and energy consumption compared to GPUs while offering much more flexibility than ASICs because of their reconfigurable features [2–6].

Recent years, High-Level Synthesis (HLS) techniques have significantly improved the developing efficiency of FPGA-based hardware design by allowing to program FPGAs in high-level languages (e.g., C/C++)[9–11]. However, building a high-performance FPGA-based DNN accelerator is still non-trivial since it requires customized hardware implementation, iterative hardware/software verification to ensure functional correctness, and effective design space exploration for sophisticated accelerator configurations. To improve the efficiency of accelerator design, we have witnessed a growing interest in developing automation frameworks for building DNN accelerators from a higher level of abstraction, using DNN-specific algorithmic descriptions and pre-defined high-quality hardware templates for fast design and prototyping [5, 6, 12–15]. However, design difficulties still exist as recent development trends in cloud and embedded FPGAs present completely different challenges for satisfying diverse requirements of DNN applications. For example, latest-generation cloud FPGAs have widely utilized multiple dies to multiply the available resources for delivering higher throughput [16, 17]. However, the cross-die routing and distributed on-chip memory can easily cause timing violations, and lower the achievable performance once the accelerator designs fail to scale up/down to match the die size. On the other hand, embedded FPGAs are integrating heterogeneous components (e.g., CPUs, GPUs) to handle different parts of the targeted tasks efficiently. Without a highly flexible task partitioning strategy, it is impossible to fully utilize the on-chip resources and leverage all the advantages of the particular hardware. Meanwhile, many researchers are seeking for improvements from a software perspective by using fast CONV algorithms (e.g., Winograd and Fast Fourier Transform) [18–21]. Although these accelerators can achieve higher performance than conventional designs, they suffer from more stringent restrictions imposed by use cases and require more complicated design flows.

To address these challenges, we propose HybridDNN, an end-to-end framework of building and implementing high-performance DNN accelerators on FPGAs leveraging the Winograd algorithm. To summarize, the main contributions of this paper are as follows.

- We propose HybridDNN, which can generate highly optimized accelerators for the latest generation of cloud and embedded FPGAs. Described by HLS, the generated designs can be easily fine-tuned for better customization.
- We introduce a highly flexible and scalable DNN accelerator architecture with a hybrid-mode (Spatial and Winograd) CONV PE and a multi-dataflow (with Input Stationary (IS) and Weight Stationary (WS)) structure.
- We integrate a comprehensive set of tools in HybridDNN for performance estimation and design space exploration to guide the accelerator design with improved performance.

2 RELATED WORKS

There are intensive studies of designing and optimizing DNN accelerators on FPGAs. Authors in [2] present a dynamic data quantization scheme for DNN parameters to relax the required memory access bandwidth. To reduce DNN inference latency for both embedded and cloud platforms, DNNBuilder proposes a fine-grained, layer-based pipeline architecture along with optimal resource allocation targeting real-life DNN with high definition inputs [5]. More advanced optimizations are investigated in accelerator design to achieve better balance between DNN inference accuracy and efficiency, such as using extremely low precision DNN parameters [12], fast CONV algorithms [21], and hardware/software co-design [7, 22, 23]. The literature also focuses on developing systematic tools for building DNN accelerators. A framework is proposed in [24] to use systolic arrays for accelerating DNN inference and the framework proposed in [25] enables task partitioning with compute-intensive CONV layers implemented on an FPGA and fully-connected (FC) layers handled by a CPU. In addition,

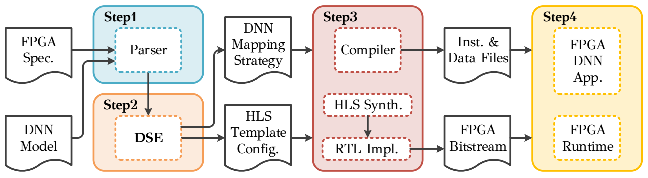


Figure 1: HybridDNN Design Flow

DeepBurning [13] introduces a design flow of using parameterized pre-defined RTL modules to construct accelerators and FP-DNN [15] uses a mixture of RTL and HLS for better flexibility.

3 THE PROPOSED DESIGN FLOW

HybridDNN provides an end-to-end design framework which can generate high-performance instruction-based DNN accelerator designs and FPGA implementations in four steps (Figure 1). In **Step (1)**, the targeted FPGA specification and the pretrained DNN model are passed to HybridDNN parser to capture hardware resource availability and DNN structure. In **Step (2)**, HybridDNN launches the Design Space Exploration (DSE) engine for design guidelines by considering both hardware and software perspectives. Design choices related to configurations of hardware instances on FPGAs are considered as hardware perspectives (e.g., the parallel factors of PE, the tile size of Winograd CONV) while factors corresponding to runtime task mapping are considered as software perspectives (e.g., selections of PE operating mode, dataflow strategies, and task partitioning schemes). Detailed discussions of the DSE will be in Section 5. After running DSE, in **Step (3)**, the HLS template configurations are finalized and transformed into synthesizable C-level descriptions and ready for deployment on the targeted FPGA. The DNN mapping strategy is handled by HybridDNN compiler to generate executable instructions for running the generated accelerators which will be discussed in Section 4. In **Step (4)**, a light-weight runtime is deployed on the host CPU to manage the execution of the generated accelerator by enabling a 4-stage instruction pipeline and I/O data management.

4 ACCELERATOR DESIGN

HybridDNN generates a hybrid Spatial/Winograd DNN accelerator and provides solutions to two major challenges of using such a hybrid design: (1) HybridDNN efficiently reuses one PE for both conventional CONV (Spatial CONV) and Winograd CONV to minimize the overhead of FPGA computing resources and (2) HybridDNN provides a decent input/output data organization mechanism to support a flexible switch between the Spatial and Winograd modes. We will illustrate why these challenges are non-trivial and our detailed solutions in this section.

4.1 Architecture and Instruction Design

A folded accelerator architecture is generated by HybridDNN to maximize the support of different DNNs. As shown in Figure 3, the accelerator is constructed with four functional modules as: (1) a LOAD_INP and (2) a LOAD_WGT module for loading input feature maps and DNN parameters, respectively, from external memory to on-chip buffers; (3) a COMP module to handle computations; and (4) a SAVE module for sending intermediate results back to external memory. In addition, a controller (CTRL module) is designed for instruction fetch and decode. Solid lines and dash lines in Figure 3 represent the data path and instructions path, respectively.

To utilize these functional modules, we propose five different instructions as: LOAD_INP, LOAD_WGT, LOAD_BIAS, COMP,

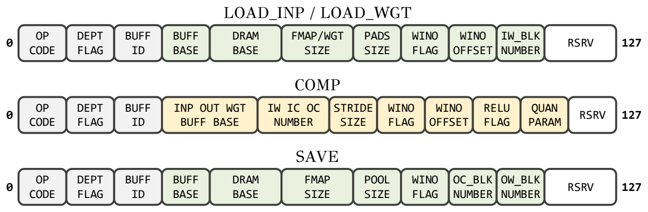


Figure 2: Customized Instruction Set

and SAVE (Figure 2). Each instruction is encoded using 128 bits and contains a WINO_FLAG domain for indicating the current CONV mode (whether Winograd or Spatial CONV). BUFF_BASE and DRAM_BASE domains in LOAD_INP, LOAD_WGT, and SAVE instructions allow the HybridDNN compiler to fully control the data access behavior of the accelerator and dynamically employ Input Stationary (IS) or Weight Stationary (WS) dataflow (with more details in Subsection 4.2.4) following the exploration of DSE.

In order to maximize performance, we employ two approaches to let all the functional modules work concurrently. We first allocate ping-pong buffers for input/output data from/to the external memory to overlap data access and computation. Then, we introduce handshake FIFOs between three pairs of data producer and consumer, such as "LOAD_INP and COMP", "LOAD_WGT and COMP", and "COMP and SAVE" (which are indicated as blue arrows in Figure 3), to prevent hazards. For each pair, the consumer will wait for the producer to emit a token through the handshake FIFO before reading and processing corresponding data. Meanwhile, the producer will wait for a token from the consumer as well, to avoid data pollution. With these two approaches, we can effectively hide the external memory access latency and improve overall performance.

4.2 Hybrid Spatial and Winograd PE

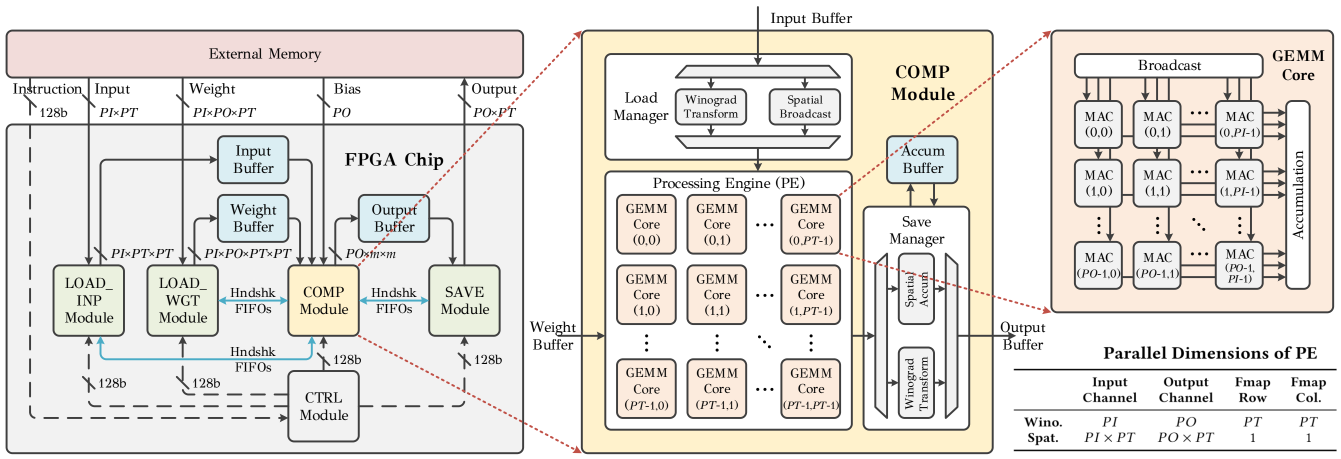
The COMP module can dynamically reuse one PE to process either Spatial or Winograd CONV. As shown in Figure 3, the proposed COMP module allocates most of the computation resources to a PE with dynamic parallel factors according to different CONV modes. It also contains an accumulating buffer and a couple of load and save managers which can be reconfigured to satisfy different data access and computation patterns of Spatial or Winograd CONV.

4.2.1 ***Winograd CONV.*** Assuming a convolutional layer with a 3-dim input feature D (size $H \times W$ with C channels) and a 4-dim kernel G (size $R \times S$ with K output and C input channels), an $F(m \times m, r \times r)$ Winograd algorithm [18] can generate output as Eq. 1. In this equation, Y and d represent output and input tiles, while g represents kernels. A, G, and B are constant transforming matrices and $\odot$ represents Element-Wise Matrix Multiplication (EWMM). Input feature D is partitioned into multiple input tiles, d, with size $(m+r-1) \times (m+r-1)$ while adjacent tiles share an $(r-1)$ overlap. In Eq. 1, the output tile size is $m \times m$ and the kernel size is $r \times r$.

$$Y = A^T \left[\left[GgG^T \right] \odot \left[B^T dB \right] \right] A \tag{1}$$

The advantage of Winograd CONV comes from the lower computation complexity. For example, an $F(4 \times 4, 3 \times 3)$ Winograd algorithm requires 36 multiplications for one output tile, while the Spatial CONV needs 144 multiplications. The reduction of multiplications is 4 times in this case. Although Winograd CONV introduces extra additions due to the transformation in Eq. 1, the cost of implementing addition is much lower than multiplication in hardware. These extra additions will not cause obvious performance slowdown.

To further improve the efficiency of implementing Winograd CONV in hardware, we transform Eq. 1 and express it in the form



	Input Channel	Output Channel	Fmap Row	Fmap Col.
Wino.	PI	PO	PT	PT
Spat.	$PI \times PT$	$PO \times PT$	1	1

Figure 3: Hardware Architecture Design Generated by HybridDNN

of General Matrix Multiplication (GEMM) shown in Eq. 2.

$$Y_{k,\tilde{x},\tilde{y}} = A^T \left[\sum_{c=1}^{C} U_{k,c} \odot V_{c,i,\tilde{x},\tilde{y}} \right] A \qquad (2)$$

where U and V represent GgG^T and $B^T dB$, respectively; and the pair $(\tilde{x}, \tilde{y})$ indicates the coordinate of the input tile. Since we can split all the EWMMs in Eq. 2 into $(m+r-1) \times (m+r-1)$ independent GEMMs, both Winograd and Spatial CONV can be represented in the form of GEMM. With the uniform representation, we can instantiate one engine but support two CONV modes.

4.2.2 *Processing Engine (PE)*

We define PI, PO, and PT as three dimensions of parallel factors in a PE. Figure 3 shows that a PE contains a $PT \times PT$ array of GEMM cores, and each GEMM core is a $PI \times PO$ broadcast array. The parallel dimensions of Spatial and Winograd CONV are shown in Figure 3, where PI and PO can be scaled to meet the resources restrictions of different FPGA platforms. PT is equal to $(m+r-1)$ which indicates the input tile size of Winograd CONV algorithm.

For each GEMM core, we unroll along input and output channel dimensions, which means we will broadcast PI channels of the input feature maps and collect PO channels of the output feature maps during the computation. In this fashion, one GEMM core is able to compute one General Matrix-Vector Multiplication (GEMV) in each clock cycle. By configuring the PE in Spatial mode, all the GEMM cores are merged into one large broadcast array; while in Winograd mode, each GEMM core is responsible for calculating one element of the EWMM operation shown in Eq. 2.

4.2.3 *Load and Save Manager*

Given the reusable PE, the next step is to pass sufficient and organized data (DNN input feature maps and parameters) to perform efficient computations. A novel reconfigurable load and save manager is proposed to handle data supply for diverse data access and computation patterns of Winograd and Spatial CONV. For Spatial mode, the load manager directly loads input feature maps and broadcast them to the PE, while the save manager sums up the results from each row of GEMM cores and pass the partial sum to the accumulating buffer. For Winograd mode, the load manager performs an online input transform from input tile d to $B^T dB$, and passes the transformed input to GEMM cores in the PE. The save manager also transforms the output tile with constant matrix A and pass the transformed results to the accumulating buffer. Regarding DNN parameters for Winograd, we perform an offline transformation from pretrained DNN models.

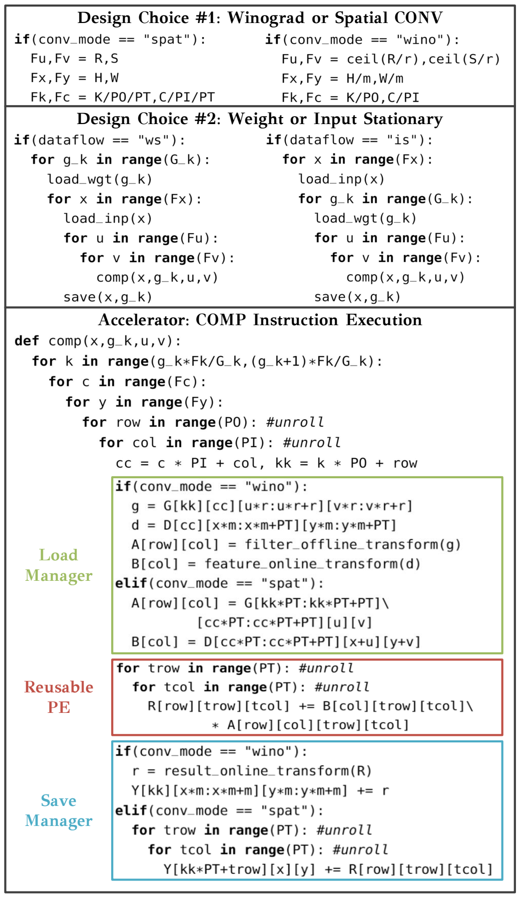
4.2.4 *CONV Operation Partition*

Due to the limited memory resources on FPGA, feature maps and weights are not always accommodated by on-chip memory. To solve this problem, we use an adaptive partition strategy to ensure a flexible support for different FPGA platforms. We partition input and output feature maps into H (Spatial CONV) or H/m (Winograd CONV) groups along the dimension of feature map height H. We partition the DNN weights into G_K groups along the dimension of output channels K. Under this strategy, one CONV operation is partitioned into $H \times G_K$ and $H/m \times G_K$ groups for Spatial and Winograd CONV, respectively. We also provide two types of dataflow for processing the CONV operation as IS and WS. For IS dataflow, the accelerator first loads one group of input feature maps, followed by G_K groups of weights serially. It then completes all calculations related to the available input feature maps. For WS dataflow, accelerator keeps weights on chip, and for each group of weights, all groups of input feature maps need to be loaded to carry out the computation.

4.2.5 *Computing Mechanism*

Figure 4 shows the pseudo-code of Spatial and Winograd CONV execution in our accelerator. Two design choices are first decided as: (1) CONV mode (Winograd or Spatial CONV) which determines the computing pattern of CONV operations; and (2) dataflow strategy (IS or WS) which determines the loops and instructions order and data dependency. The design choices here are often empirical which are mainly determined by computation/memory demands of DNN layers and available resources of the targeted FPGA. In general, the Winograd CONV requires higher memory access bandwidth than the Spatial one, while IS prefers larger feature maps compared to WS. In HybridDNN, we propose a novel DSE (Section 5) to leverage these design choices and guarantee optimized performance. With these information encoded in the instructions generated by HybridDNN compiler, our accelerator can flexibly execute CONV operations in four different design combinations but largely reuse the same PE without redundant structures.

To enhance the generality of Winograd algorithm, we also introduce a kernel decomposition method in Figure 4 to support larger kernel size ($> r \times r$) but using a $F(m \times m, r \times r)$ Winograd algorithm. Suppose we target a CONV layer with a $R \times S$ kernel ($R > r$, $S > r$), the kernel will be decomposed into $\lceil \frac{R}{r} \rceil \times \lceil \frac{S}{r} \rceil$ kernels with size $r \times r$, where zero padding will be applied if necessary. By accumulating the partial results of CONV with $r \times r$ kernels, we can output the same results of Wingrad CONV with larger kernel size.



```
Design Choice #1: Winograd or Spatial CONV
if(conv_mode == "spat"):             if(conv_mode == "wino"):
  Fu,Fv = R,S                          Fu,Fv = ceil(R/r),ceil(S/r)
  Fx,Fy = H,W                          Fx,Fy = H/m,W/m
  Fk,Fc = K/PO/PT,C/PI/PT              Fk,Fc = K/PO,C/PI
Design Choice #2: Weight or Input Stationary
if(dataflow == "ws"):                if(dataflow == "is"):
  for g_k in range(G_k):               for x in range(Fx):
    load_wgt(g_k)                        load_inp(x)
    for x in range(Fx):                  for g_k in range(G_k):
      load_inp(x)                          load_wgt(g_k)
      for u in range(Fu):                  for u in range(Fu):
        for v in range(Fv):                  for v in range(Fv):
          comp(x,g_k,u,v)                      comp(x,g_k,u,v)
      save(x,g_k)                          save(x,g_k)
Accelerator: COMP Instruction Execution
def comp(x,g_k,u,v):
  for k in range(g_k*Fk/G_k,(g_k+1)*Fk/G_k):
    for c in range(Fc):
      for y in range(Fy):
        for row in range(PO): #unroll
          for col in range(PI): #unroll
            cc = c * PI + col, kk = k * PO + row
Load        if(conv_mode == "wino"):
Manager       g = G[kk][cc][u*r:u*r+r][v*r:v*r+r]
              d = D[cc][x*m:x*m+PT][y*m:y*m+PT]
              A[row][col] = filter_offline_transform(g)
              B[col] = feature_online_transform(d)
            elif(conv_mode == "spat"):
              A[row][col] = G[kk*PT:kk*PT+PT]\
                        [cc*PT:cc*PT+PT][u][v]
              B[col] = D[cc*PT:cc*PT+PT][x+u][y+v]
Reusable    for trow in range(PT): #unroll
PE            for tcol in range(PT): #unroll
                R[row][trow][tcol] += B[col][trow][tcol]\
                          * A[row][col][trow][tcol]
Save        if(conv_mode == "wino"):
Manager       r = result_online_transform(R)
              Y[kk][x*m:x*m+m][y*m:y*m+m] += r
            elif(conv_mode == "spat"):
              for trow in range(PT): #unroll
                for tcol in range(PT): #unroll
                  Y[kk*PT+trow][x][y] += R[row][trow][tcol]
```

Figure 4: Pseudo-Code of the Execution of CONV

4.3 Memory Management

The memory access patterns are different for Spatial or Winograd CONV due to different parallel factors. It causes problems when successive layers are not implemented in the same CONV mode so that data reordering is inevitable between these layers. To solve this problem, HybridDNN proposes a novel memory structure and an efficient data reordering mechanism. The on-chip buffers are partitioned with factors shown in Table 1 to enable the parallel access to the data. The data layouts in on-chip buffers and external memory are shown in Figure 5 when $PT = 4$. We consider a GEMV operation of a vector with PI channels of input feature maps as basic operation. So each element shown in Figure 5 represents a vector of PI or PO channels for input or output feature maps, respectively.

Given this proposed data layout, we design a reconfigurable feature for the SAVE module to support all four possible data layout transforms (WINO-to-WINO, WINO-to-SPAT, SPAT-to-SPAT, and SPAT-to-WINO), while the LOAD module supports two transforms (WINO-to-WINO and SPAT-to-SPAT) as shown in Figure 5. For example, we assume the first (left) and the second (right) layer are implemented by Winograd and Spatial CONV, respectively. For the first layer, the SAVE module will work at WINO-to-SPAT mode and pass the output feature maps to the external memory following the blue arrows. Then, for the second layer, the LOAD module will load the inputs from the external memory following red arrows. With this mechanism, the required data reordering is offloaded to the SAVE module, which ensures proper data layouts for different CONV modes chosen by the successive layer.

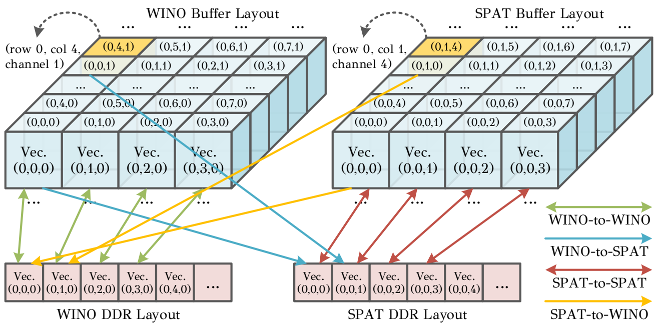


Figure 5: Feature Maps Data Layout

Table 1: Partition Factors of On-Chip Buffers

	In Buffer	Weight Buffer	Out Buffer
In Channel	$PI(PI \times PT)$	$PI(PI \times PT)$	–
Out Channel	–	$PO(PO \times PT)$	$PO(PO \times PT)$
Fmap Row	$PT(1)$	–	$m(1)$
Fmap Col.	$PT(1)$	–	$m(1)$
Weight Row	–	$PT(1)$	–
Weight Col.	–	$PT(1)$	–

*Partition factors inside of the brackets belong to Spatial mode, while factors outside of the brackets belong to Winograd mode.

5 PERFORMANCE ESTIMATION AND DESIGN SPACE EXPLORATION

With the scalable and flexible accelerator design paradigms, HybridDNN can explore a large design space in both hardware and software perspectives. In this section, we first develop highly-accuracy analytical models for resource utilization and latency estimation, and then design a DSE tool to effectively explore design space and find out the optimal configurations for building DNN accelerators.

5.1 Resources Utilization Modeling

The resource utilization of the generated accelerator is determined by PI, PO, PT, and $DATA\_WIDTH$. Since HybridDNN supports $F(2 \times 2, 3 \times 3)$ and $F(4 \times 4, 3 \times 3)$ Winograd algorithms, PT should be equal to 4 or 6. It is possible to support larger PT in Winograd algorithm, but it also introduces a large amount of extra additions which eliminates the advantage of using Winograd mode [21]. The data bitwidth is represented by $DATA\_WIDTH$. Regarding DSP utilization, we use Eq. 3 by considering three factors as: (1) the size of PE; (2) the data bitwidth of the multiplier inputs; and (3) the number of multipliers for address generation. α is the correction term related to quantization strategies and β is the number of DSPs for address generation, which is an FPGA-independent constant.

$$N_{DSP} = PI \times PO \times PT^2 + \alpha \times PO \times m^2 + PO + \beta \tag{3}$$

For memory resource, we assume all the on-chip memory are implemented using BRAM. The utilization of BRAM mainly comes from on-chip buffers as shown in Eq. 4, where $BRAM\_WIDTH$ is the data bitwidth of one BRAM instance on the targeted FPGA.

$$N_{BRAM} = \frac{DATA\_WIDTH}{BRAM\_WIDTH} \times \Big(PI \times PT^2 + PI \times PO \times PT^2 + (1+\alpha) \times PO \times m^2\Big) \tag{4}$$

We also estimate the LUT utilization as:

$$N_{LUT} = \gamma \times \left(PI \times PO \times PT^2\right) \times \left(1 + \delta \times m^2\right) \tag{5}$$

where γ is the number of LUTs per MAC unit and δ is the correction term related to the impact of m. In our design, α, β, γ, and δ can be pre-defined through profiling.

5.2 Latency Modeling

The latency of executing a CONV layer using the proposed COMP module with working frequency $FREQ$ can be calculated by Eq. 6 (Spatial) and Eq. 7 (Winograd).

$$T_{CP}^{spat} = \frac{K \times C \times R \times S \times H \times W}{FREQ \times PI \times PO \times PT^2} \tag{6}$$

$$T_{CP}^{wino} = \frac{K \times C \times \lceil \frac{R}{r} \rceil \times \lceil \frac{S}{r} \rceil \times PT^2 \times H \times W}{FREQ \times PI \times PO \times PT^2 \times m^2} \tag{7}$$

The latency of using Winograd mode is lower than Spatial mode when running the same CONV layer due to lower computation complexity. Assuming the external memory bandwidth is BW, we use Eq. 8 (Spatial) and Eq. 9 (Winograd) to indicate the latency of loading data in LOAD_WGT module.

$$T_{LDW}^{spat} = \frac{K \times C \times R \times S}{min(BW, FREQ \times PI \times PO \times PT)} \tag{8}$$

$$T_{LDW}^{wino} = \frac{K \times C \times \lceil \frac{R}{r} \rceil \times \lceil \frac{S}{r} \rceil \times PT^2}{min(BW, FREQ \times PI \times PO \times PT)} \tag{9}$$

Noted that Winograd mode asks more data from memory compared to Spatial mode, so the latency of loading data in Winograd mode is much longer. For example, assuming $m = 4$ and $r = 3$ with 5×5 kernel, the loading latency of Winograd mode is $\frac{2 \times 2 \times 6^2}{5 \times 5} = 5.76\times$ compared to Spatial mode. We also calculate the latency of LOAD_INP (Eq. 10) and SAVE module (Eq. 11):

$$T_{LDI} = \frac{C \times H \times W}{min(BW, FREQ \times PI \times PT)} \tag{10}$$

$$T_{SV} = \frac{K \times H \times W}{min(BW, FREQ \times PO \times PT)} \tag{11}$$

Assuming all the functional modules work concurrently, the overall latency is determined by the one with the longest latency. However, it is more complicated in reality because there exist data dependencies between these modules. Taking this and the computing mechanism described in subsection 4.2 into consideration, the memory access latency that cannot be hidden is separately calculated as $T_{penalty}$. So, the overall latency can be modeled as:

$$T^{spat-is} = max\left(T_{LDI}, H \times T_{LDW}^{spat}, T_{CP}^{spat}, T_{SV}\right) + T_{penalty}^{spat-is} \tag{12}$$

$$T^{spat-ws} = max\left(G_K \times T_{LDI}, T_{LDW}^{spat}, T_{CP}^{spat}, T_{SV}\right) + T_{penalty}^{spat-ws} \tag{13}$$

$$T^{wino-is} = max\left(T_{LDI}, \frac{H}{m} \times T_{LDW}^{wino}, T_{CP}^{wino}, T_{SV}\right) + T_{penalty}^{wino-is} \tag{14}$$

$$T^{wino-ws} = max\left(G_K \times T_{LDI}, T_{LDW}^{wino}, T_{CP}^{wino}, T_{SV}\right) + T_{penalty}^{wino-ws} \tag{15}$$

5.3 Architecture-Aware DNN Mapping

With the accurate estimations of resource overhead and latency, the design space exploration becomes an optimization problem targeting the lowest overall latency of processing a specific DNN model. We use Table 2 to describe this optimization problem when targeting a DNN with L CONV or FC layers. We assume the latency of the l-th layer is T_l. Also, we introduce a hardware parameter NI to represent the number of accelerator instances on one FPGA. To solve the optimization problem, we propose a novel DSE algorithm to search for the optimal design choice in 3 steps. In **Step(1)**, given the limited choices of PT, for each PT, we take turns to increase the value of PI, PO, and NI until any one of the resource constraints is no longer satisfied. We then collect the possible combinations regarding HW parameters listed in Table 2. In **Step(2)**, we consider SW parameters (CONV modes and dataflows) on top of the possible combinations from the first step, and evaluate the layer latency using Eq. 12-15. Finally, in **Step(3)**, we traverse all candidate choices from the second step and select the best one. Assuming Step(1) provides N different hardware instance candidates, the computation complexity of Step(2) and Step(3) should be $O(N \times L)$ and $O(N)$, respectively.

Table 2: DSE Optimization Problem

HW Parameters	PI, PO, PT, NI
SW Parameters	$\{mode_1, mode_2, \dots mode_L\}$, $\{dataflow_1, dataflow_2, \dots dataflow_L\}$
Constraints	$PI \geq PO \geq 1, PT \in \{4, 6\}$, $N_{LUT} < LUT, N_{DSP} < DSP, N_{BRAM} < BRAM$, $mode_l \in \{"spat", "wino"\}, dataflow_l \in \{"is", "ws"\}$
Objective	$\sum_{l=1}^{L} T_l$

Table 3: Resource Utilization of VU9P and PYNQ-Z1

	LUTs	DSPs	18Kb BRAMs
VU9P	706353 (59.8%)	5163 (75.5%)	3169 (73.4%)
PYNQ-Z1	37034 (69.61%)	220 (100%)	277 (98.93%)

6 EXPERIMENTAL RESULTS

For demonstration, HybridDNN targets a cloud FPGA (Semptian NSA.241 with Xilinx VU9P) and an embedded FPGA (Xilinx PYNQ-Z1) for generating DNN accelerators. For the cloud design, input images are sent through PCIe to the on-board DDR4, while the output results are collected by the host CPU. For the embedded design using PYNQ-Z1, the inputs are copied from SD card to main memory and processed by the programmable logic (PL). Results are then sent back to the processor system (PS).

6.1 VGG16 Case Study

We implement accelerators for running VGG16 on VU9P and PYNQ-Z1. Since VU9P has three dies (which shares the multi-dies feature in latest cloud FPGAs), HybridDNN generates six accelerator instances (with configuration: $PI = 4$, $PO = 4$, and $PT = 6$) to match the number of dies as two instances can fit in one die. For the design on PYNQ-Z1, HybridDNN generates one accelerator instance with the configuration of $PI = 4$, $PO = 4$, and $PT = 4$. Since using Winograd algorithm can be beneficial to process the CONV layers in VGG16, the proposed DSE select Winograd CONV for both designs, and we present the resource utilization in Table 3. Compared to the conventional architecture which only supports Spatial CONV, the overhead of adding Winograd supported hybrid structure (including the Winograd transformation and the reconfiguration features of the functional modules) costs only 26.4% extra LUTs but no extra DSPs on a VU9P FPGA. The main reason is that, in HybridDNN, Spatial and Winograd CONV can reuse the same PE to avoid wasting resources. This feature also helps to exploit available FPGA resources better, as most of the FPGA-based DNN accelerators mainly rely on DSP resources while leaving a considerable amount of LUTs unused.

6.2 Evaluation and Comparison

In Table 4, we compare the performance of HybridDNN to previously published works on VGG16 model. Results show that our design for VU9P achieves a 1.8x higher performance (GOPS) and 2.0x higher energy efficiency compared to the state-of-art DNN

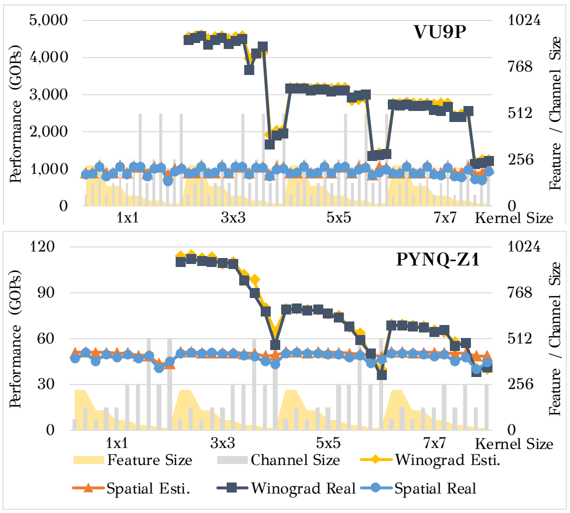


Figure 6: Performance of VU9P and PYNQ-Z1

accelerator implemented using the same FPGA. To evaluate the flexibility of HybridDNN, we extend our test cases and evaluate 60 and 40 CONV layers (with different feature map size, channel number, and kernel size) using the generated accelerators targeting VU9P and PYNQ-Z1, respectively. Results in Figure 6 indicate the performance of Spatial mode is stable and close to the peak achievable performance, while the performance of Winograd mode presents certain patterns that fluctuates across different CONV layers. By handling the same CONV layer, Winograd mode spends less computation time than Spatial mode, which equivalently causes higher demands of memory access bandwidth (as the same amount of DNN parameters needs to be loaded from DRAM within a smaller time-slot). When a memory-bound is encountered, the performance of Winograd mode drops. In the VGG16 case study, the DSE selects all CONV layers of VGG16 to be implemented in Winograd mode due to the sufficient memory bandwidth. However, in other scenarios (e.g., IoT applications) where the available memory bandwidth is limited by the embedded devices, Spatial CONV may outperform Winograd. The flexible support for both Spatial and Winograd CONV allows the proposed HybridDNN framework to deliver the optimal solutions and fit into a wide range of different scenarios. We also compare the estimated results from our proposed analytical models to the HybridDNN generated hardware implementation results, and only 4.27% and 4.03% errors are found for accelerators running on VU9P and PYNQ-Z1, respectively. The accurate estimations guarantee valid design space explorations in HybridDNN.

7 CONCLUSIONS

In this paper, we presented HybridDNN, a framework for building DNN accelerators on FPGAs with high-performance and energy efficiency. We proposed a highly scalable DNN accelerator architecture for efficient deployment onto cloud/embedded FPGAs and a flexible PE structure with hybrid Spatial/Winograd CONV support for diverse CONV layers. We designed a comprehensive analytical tool for fast and accurate performance estimation (with 4.27% and 4.03% error rate for accelerator designs in VU9P and PYNQ-Z1, respectively), and a DSE engine to provide the best configurations regarding CONV modes (Spatial/Winograd), dataflows (IS/WS), and parallel factors. With the above novel technologies, HybridDNN delivered accelerators with the highest performance peaking at 3375.7 (VU9P) and 83.3 (PYNQ-Z1) GOPS, and the best energy efficiency (73.5 GOPS/W) compared to previously published results.

Table 4: Comparison with Previous Works

	[26]	[4]	[6]	Ours	
Device	Xilinx VU9P	Arria10 GX1150	Xilinx VU9P	Xilinx VU9P	PYNQ Z1
Model	VGG16	VGG16	VGG16	VGG16	VGG16
Precision	16-bit	16-bit	16-bit	12-bit*	12-bit*
Freq.(MHz)	210	385	214	167	100
DSPs	4096	2756	5349	5163	220
CNN Perf.(GOPS)	1510	1790	1828.6	**3375.7**	83.3
Power(W)	NA	37.5	49.3	45.9	2.6
DSP Effi. (GOPS/DSP)	0.37	0.65	0.34	**0.65**	0.38
Energy Effi. (GOPS/W)	NA	47.78	37.1	**73.5**	32.0

*DNN parameters are quantized to 8-bit; input feature maps are set to 12-bit in PE due to the Winograd matrix transformation

ACKNOWLEDGMENTS

This work is supported in part by the IBM-Illinois Center for Cognitive Computing Systems Research (C3SR) and XMotors.ai.

REFERENCES

[1] Tianshi Chen et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, pages 269–284, 2014.
[2] Jiantao Qiu et al. Going deeper with embedded FPGA platform for convolutional neural network. In *Proc. of FPGA*, 2016.
[3] Xiaofan Zhang et al. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *Proc. of FPL*, 2017.
[4] Jialiang Zhang and Jing Li. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proc. of FPGA*, 2017.
[5] Xiaofan Zhang et al. DNNBuilder: an automated tool for building high-performance dnn hardware accelerators for FPGAs. In *Proc. of ICCAD*, 2018.
[6] Yao Chen et al. Cloud-DNN: An open framework for mapping dnn models to cloud FPGAs. In *Proc. of FPGA*, 2019.
[7] Xiaofan Zhang et al. SkyNet: a hardware-efficient method for object detection and tracking on embedded systems. In *Proc. of MLSys*, 2020.
[8] Pengfei Xu et al. AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs. In *Proc. of FPGA*, 2020.
[9] Deming Chen et al. Lopass: A low-power architectural synthesis system for fpgas with interconnect estimation and optimization. *IEEE TVLSI*, 18(4):564–577, 2009.
[10] Deming Chen et al. xpilot: A platform-based behavioral synthesis system. *SRC TechCon*, 2005.
[11] Kyle Rupnow et al. High level synthesis of stereo matching: Productivity, performance, and software constraints. In *Proc. of FPT*, 2011.
[12] Junsong Wang et al. Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. In *Proc. of FPL*, 2018.
[13] Ying Wang et al. DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family. In *Proc. of DAC*, 2016.
[14] Chen Zhang et al. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE TCAD*, 2018.
[15] Yijin Guan et al. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Proc. of FCCM*, 2017.
[16] Manish Deo. Enabling next-generation platforms using Intel's 3d system-in-package technology, 2017. Accessed: 2019-11-23.
[17] Xilinx. Ultrascale architecture configuration, 2019. Accessed: 2019-11-23.
[18] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proc. of CVPR*, 2016.
[19] Liqiang Lu et al. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *Proc. of FCCM*, 2017.
[20] Roberto DiCecco et al. Caffeinated FPGAs: FPGA framework for convolutional neural networks. In *Proc. of FPT*, 2016.
[21] Chuanhao Zhuge et al. Face recognition with hybrid efficient convolution algorithms on FPGAs. In *Proc. of GLSVLSI*, 2018.
[22] Weiwen Jiang et al. Accuracy vs. efficiency: Achieving both through FPGA-implementation aware neural architecture search. In *Proc. of DAC*, 2019.
[23] Cong Hao et al. FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge. In *Proc. of DAC*, 2019.
[24] Xuechao Wei et al. Automated systolic array architecture synthesis for high throughput cnn inference on FPGAs. In *Proc. of DAC*, 2017.
[25] Hanqing Zeng et al. A framework for generating high throughput cnn implementations on FPGAs. In *Proc. of FPGA*, 2018.
[26] Xuechao Wei et al. TGPA: tile-grained pipeline architecture for low latency CNN inference. In *Proc. of ICCAD*, 2018.