Power-Driven DNN Dataflow Optimization on FPGA

(Invited Paper)

Qi Sun[1], Tinghuan Chen[1], Jin Miao[2], and Bei Yu[1]

[1]CSE Department, The Chinese University of Hong Kong, NT, Hong Kong SAR
[2]Cadence Design Systems, CA, USA
{qsun,thchen,byu}@cse.cuhk.edu.hk, jmiao@cadence.com

***Abstract*—Deep neural networks (DNNs) have been proven to achieve unprecedented success on modern artificial intelligence (AI) tasks, which have also greatly motivated the rapid developments of novel DNN models and hardware accelerators. Many challenges still remain towards the design of power efficient DNN accelerator due to the intrinsically intensive data computation and transmission in DNN algorithms. However, most existing efforts in the domain have taken latency as the sole optimization objective, which may often result in sub-optimality in power consumption. In this paper, we propose a framework to optimize the power efficiency of DNN dataflow on FPGA while maximally minimizing the impact on latency. We first propose power and latency models that are built upon different dataflow configurations. Then a power-driven dataflow formulation is proposed, which enables a hierarchical exploration strategy on the dataflow configurations, leading to efficient power consumption at limited latency loss. Experimental results have demonstrated the effectiveness of our proposed models and exploration strategies, where power improvement has shown up to 31% with latency degradation of no worse than 6.5%.**

***Index Terms*—FPGA, Deep Neural Network, Dataflow Optimization, Power.**

I. Introduction

Deep Neural Networks (DNNs) have been proven to achieve unprecedented success on a wide range of AI relevant applications, such as image classification, object detection, and design for manufacturing [1]–[3]. FPGA has been chosen as a promising hardware platform to deploy DNN model due to the reconfigurability [4], [5]. When deploying a DNN application onto an FPGA board, in an end-to-end automation flow, two pivotal steps are involved: architectural design and dataflow optimization. For architectural design, a typical DNN accelerator architecture is shown in Fig. 1. The processing unit (PU) is responsible for the computations, which is composed of several processing engine (PE) arrays. A multiplier-accumulator (MAC) can be implemented by various processing engines (PEs), such as arithmetic multiplication array or systolic array [6]–[8]. For dataflow optimization, an analytical model, per the given architecture, is often developed to capture and measure various design configurations and their attainable performances by metrics like latency, power consumption, data transfer size, and on-chip resource consumption, etc. [6], [8]–[13]. The optimal design configuration is iterated and achieved through above explorations [6], [8], [11], [12].

For fast prototyping, the deployment of DNN onto FPGA is facilitated by high-level synthesis (HLS) tools which emancipate designers from the complicated hardware description languages (like Verilog or VHDL) crafting. In addition, HLS makes it possible for efficient DNN design modeling and configurations [14], unlocking rapid design space explorations for high-performance designs [15].

Previous literatures on FPGA based DNN accelerators have primarily focused solely on the latency reduction [8], [10], [11], [13]. In latency-driven designs, hardware resources are activated without restrictions for best performances. Apparently, such latency-driven designs often end up with sub-optimality in power consumption and are excluded from applications demanding low-power.

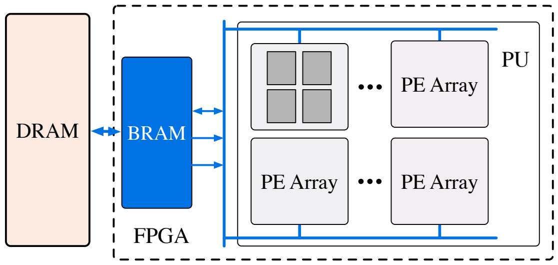


Fig. 1 DNN FPGA accelerator architecture.

In the traditional ASIC or FPGA hardware design domain, power-driven design methodologies have been well studied. Chen *et al.* proposed a low-power HLS methodology for general application FPGA designs without significant loss of performance [16]. Besides, optimization on linear algebra core was presented to achieve high-performance while reducing power consumption [17]. Nevertheless, the power efficiency explorations and optimizations on DNN deployed onto FPGA have yet been sufficiently studied.

In this paper, we propose a framework to optimize the power efficiency of DNN dataflow on FPGA while minimizing the impact of latency. We first propose power and latency models that are built upon different dataflow configurations. Then a power-driven dataflow formulation is proposed, which enables a hierarchical exploration strategy on the dataflow configurations, leading to globally optimal power efficiency with insignificant latency loss.

We also deploy two DNN models on FPGA to verify the effectiveness of our proposed framework. In this architecture, neighboring layers are fused to reduce the volume of data transferred between on-chip buffer and off-chip memory. Systolic array is adopted as the computation core to balance data access and computation. Overall experimental results have demonstrated the power improvement of up to 31% with latency degradation of no worse than 6.5%.

The rest of our paper is organized as follows. Section II provides preliminaries including concrete analysis from DNN dataflow to systolic array and fused layer. Section III introduces our power formulation. Section IV presents a dataflow configuration estimation and exploration process using our proposed power-driven formulation. Section V demonstrates the experiments and results. Finally, we conclude this paper and provide further discussions in Section VI.

II. Preliminaries

A typical DNN model contains several operational layers, including convolution, pooling and ReLU, where convolutional layers dominate both storage and computation in current popular DNN structures [1], [18]. The limited on-chip memory resources would therefore cause frequent data transfer between on-chip buffer and

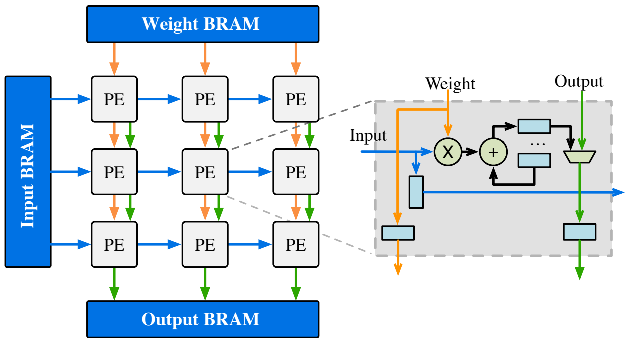


Fig. 2 Systolic array architecture and processing engine structure. Data are fed into array from the top and the left.

off-chip memory. In addition, pipelining and parallelism, depending on the size of on-chip buffer and the processing unit (PU) allocated by on-chip logic resources (LUTs) and DSPs, are often applied on convolutional layers. Convolutional layers therefore have been the focal of success in design and optimization of hardware accelerators.

Recall in Fig. 1, we show a typical DNN FPGA accelerator architecture. The accelerator is composed of off-chip memory (usually by DRAM) and an FPGA accelerator chip. On the aspect of hierarchical memory, it can be decomposed into three parts: off-chip DRAM, on-chip global buffer (BRAM) and local buffer (register). Accessing unit data at different levels implies different energy consumption [19]. We load data from off-chip DRAMs to global buffers. Then the data are passed from global buffers to local buffers (in PE array) and do the computations. Afterwards, outputs are stored to DRAMs.

Systolic array is used as the processing engine array, as shown in Fig. 2, which becomes more and more popular [8], [20], [21] in DNN FPGA and ASIC designs recently. Each PE in systolic array connects to its neighbors while the boundary PEs also connects to the global buffers (BRAM). In each cycle, the input feature and kernel weight are passed to its neighbors which are then stored in local buffers in PE simultaneously. The boundary PEs receive data passed down from the global buffers. The input feature and kernel weight are multiplied first, then passed through accumulation operation with partial sum. Note that for computations in each PE, the input features are fed from the left to right, whereas the weights are fed from the top to bottom. By contrast, results of all PEs flow from top to bottom, into global buffers. Obviously, the size of systolic array also impacts the power consumption when passing data and conducting computations.

The design of DNN Dataflow refers to the selection procedure of configurations on data storage and data access pattern on FPGA. Typical techniques include loop tiling, unrolling, data reuse and layer fusion. We detail each of them as below.

Loop tiling partitions a loop into smaller blocks. For example, Fig. 3(a) is a 6-level for-loops of a convolutional layer. The outermost loop with step size 1 is decomposed into two sub-loops and the size of inner sub-loop is OC, as shown in Fig. 3(b). The decomposed sub-loops are responsible for transferring data between off-chip memory and on-chip global buffer. The sub-loop boundary is called the loop tiling factor, which reflects the size of global buffers. Data loaded in the outer loop flow into the local buffers (in computation engines) in the inner loops.

To facilitate parallelism, we unroll the convolutional group function as shown in Fig. 3(c). Originally, only a single convolution function is called for IC times. During computations, different data segments flow into the same PU sequentially. Once we set the unrolling factor as IC, the original loop will be replaced with IC parallel convolutional functions. In this case, these data will be passed into IC different PUs in parallel. Loop unrolling also relies on loop tiling factors because we always unroll the sub-loops after tiling the loop. Traditional works usually unroll and parallel as many tasks as possible. This kind of choices imply challenges on power since lots of on-chip logics are instantiated and activated. In order to improve power performance, the unrolling and tiling factors need to be carefully configured.

Data reuse strategy is also beneficial. Fig. 3(d) is an example of the output data reuse strategy. The output data are loaded from DRAM to BRAM once and reused in the inner loop several times. The blue line in Fig. 3(d) shows the reuse situation. Similarly, if we reuse inputs, we only need to load inputs once. Different strategies may affect the total length of dataflow, hence the communication power consumption.

A naive DNN accelerator design runs layers one by one on FPGA. Due to the sequential nature, the output features of the previous layer have to be first stored in off-chip memory before starting the next layer computation. The drawback is obvious that we have to transfer the same data between FPGA and DRAM repeatedly. But for layer fusion, once we get parts of the output data of the first layer, they will be passed to compute engines of the second layer as inputs and start the computations immediately. We can fuse convolution layers with the neighboring pooling layers or ReLU layers [21], [22]. A more effective strategy is to fuse several neighboring convolution layers together with pooling and ReLU layers, which further reduces the size of data transfer [23]. Some extra data processing is necessary to handle the overlapping data resulted from layer fusion, at the cost of more memory consumption. Using layer fusion limits the choice of data reusing strategy. If we reuse input features of the second layer, only when we finish all the computations of one block in the first layer, can we start the computations of the corresponding block in the second layer. In this process, computations in the second layer are blocked, which is not helpful for the system pipelining. Typically, as shown in Fig. 4(b), we reuse outputs in the first layer, which means we must reuse inputs in the second layer.

III. Power Minimization

In this section, we discuss the details of our proposed power optimization framework. As energy is defined as the integral of power over the latency of an underline task, after introducing parameters and notations, we will first derive energy and latency models respectively. We will further show the overall power minimization formulation to be explored and optimized.

A. Notations

Denote the number of layers in a DNN model as Z. For a convolutional layer i, the number of input channels is N_i, number of output channels is M_i, kernel size is K_i, kernel stride is S_i, feature height is H_i, and feature width is W_i. Here we assume the output feature size is equal to the input feature size, for the expression simplicity.

Generally, we summarize the parameters in TABLE I. Some have been used in Fig. 3(b) and Fig. 3(c). We can decompose the input convolutional channels into blocks, with IC_i channels in each block. Output channels are decomposed into blocks and each has OC_i output channels. Features in each channel are also

```
for to in range(0, M):
  for row in range(0, H):
    for col in range(0, W):
      for ti in range(0, N):
        for k1 in range(0, K):
          for k2 in range(0, K):
            $OUT_{to,row,col} += WT_{to,ti,k1,k2} \times IN_{ti,row+k1,col+k2}$
```

(a)

```
for to in range(0, M, OC):
  for to1 in range(0, OC):
    for row in range(0, H):
      for col in range(0, W):
        for ti in range(0, N):
          for k1 in range(0, K):
            for k2 in range(0, K):
              $OUT_{to1+to,row,col} += WT_{to1+to,ti,k1,k2} \times IN_{ti,row+k1,col+k2}$
```

(b)

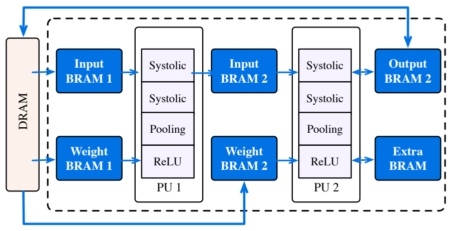
```
conv_group(BIN[IC],BWT[OC][IC],BOUT[OC]):
  for L1 in range(0, OC):
    for L2 in range(0, IC):
      convolution($BIN_{L2}$,$BWT_{L1,L2}$,$BOUT_{L1}$)

conv_group(BIN[IC],BWT[OC][IC],BOUT[OC]):
  for L1 in range(0, OC):
    convolution($BIN_0$,$BWT_{L1,0}$,$BOUT_{L1}$)
    convolution($BIN_1$,$BWT_{L1,1}$,$BOUT_{L1}$)
    ...
    convolution($BIN_{IC-1}$,$BWT_{L1,IC-1}$,$BOUT_{L1}$)
```

(c)

```
for to in range(0, M, OC):
  for row in range(0, H, PH):
    for col in range(0, W, PW):
      load_output(BOUT[OC][PH][PW],DOUT)
      for ti in range(0, N, IC):
        load_input(BIN[IC][PH][PW],DIN)
        load_weight(BWT[OC][IC][K][K],DWT)
        conv_group(BIN[IC],BWT[OC][IC],
                   BOUT[OC])
      store_output(BOUT[OC][PH][PW],DOUT)
```

(d)

Fig. 3 Pseudo-code of dataflow optimization. (a) A 6-loops convolutional layer. (b) Loop Tiling. (c) Loop Unrolling. (d) Output Data Reuse.



```
for to1 in range(0, $M_1$, $OC_1$):
  for row1 in range(0, $H_1$, $PH_1$):
    for col1 in range(0, $W_1$, $PW_1$):
      for ti1 in range(0, $N_1$, $IC_1$):
        load_input($BIN_1$[$IC_1$][$PH_1$][$PW_1$],$DIN_1$)
        load_weight($BWT_1$[$OC_1$][$IC_1$][$K_1$][$K_1$],$DWT_1$)
        systolic_group_1($BIN_1$[$IC_1$],$BWT_1$[$OC_1$][$IC_1$],
                         $BOUT_1$[$OC_1$],$U_1$)
      prepare_for_layer2($BOUT_1$[$OC_1$][$PH_1$][$PW_1$],
                         extra_buffer)
      for to2 in range(0, $M_2$, $OC_2$):
        load_output($BOUT_2$[$OC_2$][$PH_2$][$PW_2$],$DOUT_2$)
        load_weight($BWT_2$[$OC_2$][$OC_1$][$K_2$][$K_2$],$DWT_2$)
        systolic_group_2($BOUT_1$[$OC_1$],$BWT_2$[$OC_2$][$OC_1$],
                         $BOUT_2$[$OC_2$],$U_2$)
        store_output($BOUT_2$[$OC_2$][$PH_2$][$PW_2$],$DOUT_2$)
```

(b)

Fig. 4 Fusing two convolutional layers: (a) hardware architecture using systolic array; (b) pseudo-codes.

TABLE I List of Parameters

Name	Definition
OC_i	# of output channels per feature block in fused layer i
IC_i	# of input channels per feature block in fused layer i
PH_i	Feature height of feature block in fused layer i
PW_i	Feature width of feature block in fused layer i
Th_i	row # of PEs in one systolic array in fused layer i
Tw_i	column # of PEs in one systolic array in fused layer i
U_i	# of instantiated systolic arrays in fused layer i

decomposed into smaller blocks, with height PH_i and width PW_i. The corresponding block stride is PS_i, which can be determined according to the block size and kernel stride. The pseudo-code of fusing two convolutional layers using systolic array is in Fig. 4(b). The output channels of the first layer are the input channels of the second layer. Therefore, we have $M_1 = N_2$ and $OC_1 = IC_2$. Also, the number of input blocks of the second layer is equal to the number of blocks in the first layer.

Data used in systolic array should be in the matrix format while they are often stored as tensors in DRAM. The process of transforming data from the tensors to matrices is conducted on board to avoid unnecessarily repeated DRAM access and extra power load. Denote the height and width of systolic array for fused layer i as Th_i and width Tw_i, the number of instantiated systolic arrays in a processing unit as U_i. For any feature tensors, they should be flattened as matrices, with height $(\frac{PH_i - K_i}{PS_i} + 1) \times (\frac{PW_i - K_i}{PS_i} + 1)$. For weights, they are flattened as matrices with width OC_i. The depth of the transformed matrix is D_i, as shown in Equation (1) [24].

$$D_i = K_i^2 \times IC_i. \tag{1}$$

B. Modeling the Energy

Without loss of generality, the total energy can be composed of two parts, data transfer and computation.

As mentioned before, there are three-levels of hierarchical memory in DNN accelerator system. We measure the data transfer energy by calculating the data size transferring between all these levels. The energy of accessing one unit data at each level is regarded as a constant. Compared with on-chip buffers, accessing data from DRAM consumes much more energy. In Fig. 4(b), the *load* and *store* functions read and store data between DRAM and FPGA. As shown in Fig. 4(a), both two layers need to load weights from off-chip memory to on-chip buffer. The first layer needs to load input features from DRAM. Meanwhile, we transfer output features, or partial sums of the second layer between off-chip memory and on-chip buffer several times since we reuse input features here. Other sources of energy consumptions include extra buffers used to prepare data and handle data overlaps for the second fused layer, as well as passing data between global buffer and systolic array, etc.

For simplicity, we group the data transfer energy in systolic array as part of the computation energy due to the computations within systolic array. The data transfer energy of layer i can be simplified as in Equation (2). Here $\{\alpha_1, \alpha_2, \dots, \alpha_7\}$ are model-specific constants that can be predetermined once the DNN model is given. For example, if reusing outputs, α_1 is the multiplication of feature size and channel number, and $(\alpha_1 \ / \ OC_i)$ refers to the input data size we load. Similarly, $(\alpha_7 \ / \ IC_i)$ refers to how many output data we load, if we reuse inputs. $(\alpha_2 \ / \ PH_i \times PW_i)$ stands for weights energy. In addition, we have four terms $(\alpha_5 \ / \ PH_i)$, $(\alpha_6 \ / \ PW_i)$, $\alpha_3 PH_i$ and $\alpha_4 PW_i$ for fused data preparations.

$$\begin{aligned} ED_i = & \frac{\alpha_1}{OC_i} + \frac{\alpha_2}{PH_i \times PW_i} + \alpha_3 PH_i + \alpha_4 PW_i \\ & + \frac{\alpha_5}{PH_i} + \frac{\alpha_6}{PW_i} + \frac{\alpha_7}{IC_i}. \end{aligned} \tag{2}$$

As to the computation energy, naively, we may assume the total computation energy is a constant. This is because from the perspective of DNN model description, the total number of computations (accumulation and multiplication) are definite. However, for hardware deployments, the computation energy is tightly proportional to the sizes of computation engines. As mentioned earlier, the size of computation engine (or systolic array) is roughly a product of the height of input feature block times the width of weight block. Input feature block with size $(IC_i \times PH_i \times PW_i)$ is decomposed into $(\lceil(\frac{PH_i-K_i}{PS_i}+1)\times(\frac{PW_i-K_i}{PS_i}+1)\ /\ Th_i\rceil)$ sub-matrices. Each has size $(D_i \times Th_i)$, where D_i is the depth as shown in Equation (1). Similarly, for weights with size $(OC_i \times K_i^2 \times IC_i)$, they are decomposed into $(\lceil OC_i\ /\ Tw_i\rceil)$ sub-matrices. Each has size $(D_i \times Tw_i)$. The ceiling operations here are to tackle with the boundary situations.

In each computation cycle, all the $(Th_i \times Tw_i)$ PEs are in active status and consume energy. For each pair of inputs and weights sub-matrices, $(D_i + Th_i + Tw_i - 2)$ cycles are needed to finish the computations. Energy consumed to finish the computation of one feature block and one corresponding weight block using systolic array is denoted as eb, in Equation (3), where ec is the energy consumed by each PE in each cycle, including inside data transfer energy.

$$\begin{aligned} eb\ &= \left\lceil \frac{(\frac{PH_i-K_i}{PS_i}+1)\times(\frac{PW_i-K_i}{PS_i}+1)}{Th_i} \right\rceil \times \left\lceil \frac{OC_i}{Tw_i} \right\rceil \times \\ &(Th_i \times Tw_i) \times (D_i + Th_i + Tw_i - 2) \times ec. \end{aligned} \tag{3}$$

The total energy consumed by layer i is the summation of all blocks, as in Equation (4).

$$\begin{aligned} EC_i =& \lceil N_i/IC_i\rceil \times \lceil H_i/PH_i\rceil \times \lceil W_i/PW_i\rceil \\ &\times \lceil M_i/OC_i\rceil \times eb. \end{aligned} \tag{4}$$

Here we assume that the energy consumption have nothing to do with the number of systolic arrays (parallelism), *i.e.* U_i. The reason is that no matter how many systolic arrays we have instantiated, we only call it $(\lceil(\frac{PH_i-K_i}{PS_i}+1)\times(\frac{PW_i-K_i}{PS_i}+1)\ /\ Th_i\rceil \times \lceil OC_i\ /\ Tw_i\rceil)$ times in computations. By contrast, the size of each individual systolic array is proportional to energy consumption.

The overall model energy consumption E_{total} is the summation of computation and data transfer energy of all layers, as shown in Equation (5).

$$E_{total}\ =\ \sum_{i=1}^{Z}(ED_i + EC_i). \tag{5}$$

C. Modeling the Latency

Intuitively, the latency consists of computation latency and data transfer latency. The characteristic of using systolic array is that, in each PE, we pass data to neighboring PEs per cycle and finish the computation simultaneously. Therefore, part of data transfer latency has overlapped with the computation latency. Total data transfer latency is determined by the size of data we transfer. Although some transfer overlaps could occur per engineering implementations, we can still sum up all the data to be transferred to estimate the latency. The on-chip data transfer latency should be divided by the degree of system parallelism since the data are passed into the unrolled computation engines simultaneously.

Except for the latency regarding to systolic arrays, the data transfer latency in layer i can be formulated as in Equation (6), where $\{\beta_1, \beta_2, \ldots, \beta_7\}$ are model-specific constants. The explanations to these terms are similar to Equation (2). For example, $(\beta_1\ /\ OC_i)$ is for input data transfer latency.

Here we highlight the latency of passing data into systolic arrays. For input feature block with size $(IC_i \times PH_i \times PW_i)$, and weight block with size $(OC_i \times IC_i \times K_i^2)$, we have computation latency in Equation (7).

$$\begin{aligned} LD_i =& \frac{\beta_1}{OC_i} + \frac{\beta_2}{PH_i \times PW_i} + \beta_3 PH_i + \beta_4 PW_i \\ &+ \frac{\beta_5}{PH_i} + \frac{\beta_6}{PW_i} + \frac{\beta_7}{IC_i}. \end{aligned} \tag{6}$$

$$\begin{aligned} lc\ &= \left\lceil \frac{(\frac{PH_i-K_i}{PS_i}+1)\times(\frac{PW_i-K_i}{PS_i}+1)}{Th_i} \right\rceil \times \left\lceil \frac{OC_i}{Tw_i} \right\rceil \\ &\times (D_i + Th_i + Tw_i - 2), \end{aligned} \tag{7}$$

where D_i is the depth of transformed matrix as in Equation (1). The total latency consumed by layer i is the summation of all blocks, as in Equation (8).

$$\begin{aligned} LC_i\ =& \lceil\lceil N_i\ /\ IC_i\rceil \times \lceil H_i\ /\ PH_i\rceil\ /\ U_i\rceil \times \\ & \lceil W_i\ /\ PW_i\rceil \times \lceil M_i\ /\ OC_i\rceil \times lc. \end{aligned} \tag{8}$$

Since we have U_i instantiated parallel systolic arrays in total for fused layer i, the overall latency should be divided by U_i. The ceiling operations here are to tackle with the boundary situations, too.

The total model latency L_{total} can be formulated as Equation (9), which is summation of latencies of all layers.

$$L_{total} = \sum_{i=1}^{Z}(LD_i + LC_i). \tag{9}$$

D. Power Minimization Formulation

The overall optimization objective is to minimize the ratio of energy consumption over system latency, constrained by resources, as shown in Formula (10). This formulation is also constrained by latency upper bound L_{upper} since we cannot allow latency to be infinitely large. $Buffer_{total}$ and DSP_{total} represent the total buffers and DSPs available on FPGA. $Buffer_{used}$ and DSP_{used} denote the memory and computation resources used.

$$\begin{aligned} \min\ & \frac{E_{total}}{L_{total}}, \\ \text{s.t.}\ & Buffer_{used} \le Buffer_{total}, \\ & DSP_{used} \le DSP_{total}, \\ & L_{total} \le L_{upper}. \end{aligned} \tag{10}$$

Equation (9) and Equation (5) are both the summation of all layers. However, for a given DNN model, the resources used on chip only need to handle one fused layer group, i.e., we run the fused groups one by one sequentially.

Let $Buffer^i_{global}$ represent the size of global buffers used by fused layer i, which includes the following terms. Denote the unit size of buffers used to store these data as C_{GLB}, which is determined by data precision. The size of input features buffers needed by one block in layer i is $IC_i \times PH_i \times PW_i \times C_{GLB}$. Similarly, the sizes of output features and weights buffers are $OC_i \times PH_{i+1} \times PW_{i+1} \times C_{GLB}$ and $OC_i \times IC_i \times K_i^2 \times C_{GLB}$ respectively. For the second fused layer, as shown in Fig. 4(a), extra buffers are needed to store the overlapping features. The horizontal overlapping data

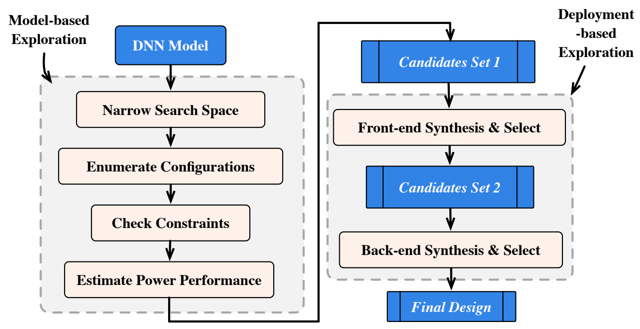


Fig. 5 The proposed exploration flow, where the input is DNN model, the output is final dataflow configuration.

need buffers with size $IC_{i+1} \times W_{i+1} \times PS_{i+1} \times C_{GLB}$, and the vertical data have size $IC_{i+1} \times PH_{i+1} \times PS_{i+1} \times C_{GLB}$.

The local buffers $Buffer^i_{local}$ of fused layer i are needed in each PE to store the intermediate data. Therefore, local buffer size is equal to $U_i \times Th_i \times Tw_i \times C_{PE}$, where C_{PE} is a constant representing size of local buffers per PE.

The number of DSPs used in each PE in systolic array can be assumed as a constant, denoted as C_{DSP}, which is also determined by data precision. The total number of DSPs used by one layer is equal to $U_i \times Th_i \times Tw_i \times C_{DSP}$.

For a fused layer group with L layers, we have $Buffer_{used}$ and DSP_{used} in Equations (11) and (12).

$$Buffer_{used} = \sum_{i=1}^{L} \left(Buffer^i_{local} + Buffer^i_{global} \right), \tag{11}$$

$$DSP_{used} = \sum_{i=1}^{L} \left(U_i \times Th_i \times Tw_i \times C_{DSP} \right). \tag{12}$$

We use the proposed model as the metric of measuring power performance of dataflow configurations. Any configurations violating the constraints will be excluded.

IV. Dataflow Exploration

In this section, we discuss how to explore power optimal dataflow configurations based on our proposed formulation. The Formula (10) is non-convex and non-linear. Besides, all the parameters should be integers according to their hardware meanings. It is nearly impossible to enumerate all configurations in the search space and run synthesis for them all. We therefore propose hierarchical exploration which can be decomposed into two steps: **model-based exploration** and **deployment-based exploration**. The overall solution exploration flow is in Fig. 5.

In **model-based exploration**, we estimate the power performance of dataflow configurations using our power model. For a given DNN model, firstly, we narrow the search space by adding more practical and empirical constraints. This step is reasonable since some specific domain knowledge cannot be expressed explicitly in the objective function. For example, the on-chip buffer size is usually the power of two. Therefore, the search space for $\{OC_i, PH_i, PW_i, IC_i\}$ is narrowed at a large scale. Further, to help solve the latency constraints in Equation (10), we can constrain the parameters in $\{U_i, Th_i, Tw_i\}$ to be greater than an acceptable lower bound. Layer fusion strategy is highly related to the DNN model structures. The fused layers should share similar structures due to hardware regularity especially in FPGA. For example, for VGG16, we split the model into several groups at pooling layers.

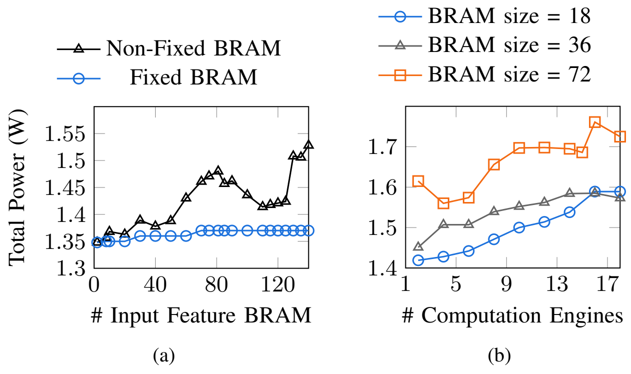


Fig. 6 (a) If the number of BRAMs is not fixed, as the number increases, more power is consumed. By fixing the BRAM but read in same amounts of data, the power consumption is stable; (b) An example of deploying 2×2 convolutions on PYNQ-Z1. As the number of computation engines increases, the power increases. With more BRAMs used, the power performance also decreases.

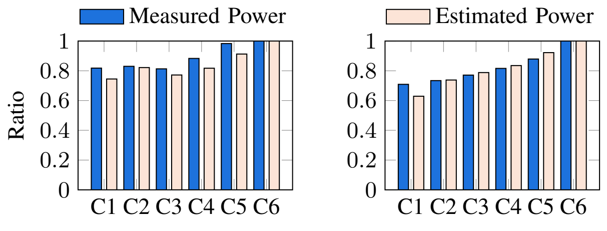
Then all layers in one group are fused, including convolutional layers, ReLU layers, and pooling layers. For AlexNet, convolutional layers sharing 3×3 kernels are fused. For layers with 11×11 and 5×5 kernels, they are not fused since the different kernel sizes may lead to unstructured hardware designs which consume lots of logic resources to do the logical controls. For the layers which are not fused, we can follow the empirical ideas as proposed in [25]. After narrowing the exploration space with such, all possible legal configurations can be enumerated. Then we check the constraints and discard the configurations violating any constraint. A set of candidates with lower power values out of the remaining configurations will survive.

In **deployment-based exploration**, we further verify the configurations selected in previous step, removing those incompatible to FPGA hardware platform. Thus an even smaller set of candidates are generated after front-end synthesis. Eventually, those candidates passing all above pruning procedures are fed into the next step to run the back-end synthesis to get the actual power-performance metrics. The final optimal configurations is the one with the best back-end power performance.

V. Experimental Results

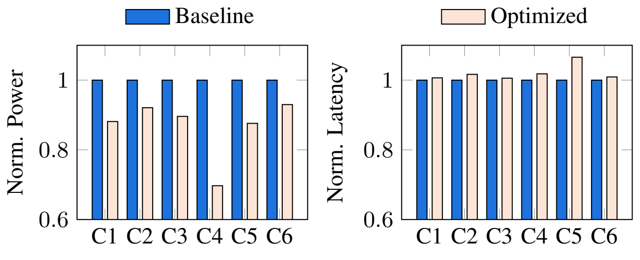
All experiments are conducted on Xilinx Zynq UltraScale+ MPSoC ZCU102 or PYNQ Z1 [26]. The design and deployment flow are via Xilinx Vivado HLS 2018.2. Some fundamental convolution examples, together with AlexNet and VGG16 are evaluated. The L_{upper} is set as 108% of the baseline latency for AlexNet and VGG16.

Firstly, the influences of instantiating different numbers of BRAMs (global buffers) and computation engines are shown in Fig. 6. As in Fig. 6(a), we only load and store data between DRAM and FPGA, with different buffer sizes and data sizes. The x-axis represents the number of BRAMs. If we enlarge the BRAM capacity, *i.e.* instantiate more BRAMs, the system power will increase significantly. If the BRAM size is fixed and we load in the same data with the non-fixed version, data transfer wouldn't have negative impacts on power performance. Another example in Fig. 6(b) shows the situation of finishing a 2×2 convolution task at various parallel levels. The convolution task is decomposed into several smaller parallel sub-tasks. One parallel sub-task is assigned with one computation engine. The difficulties of partitioning the



(a) (b)

Fig. 7 Model fidelity analysis: (a) AlexNet; (b) VGG16.



(a) (b)

Fig. 8 The results of a set of configurations on: (a) power; (b) latency.

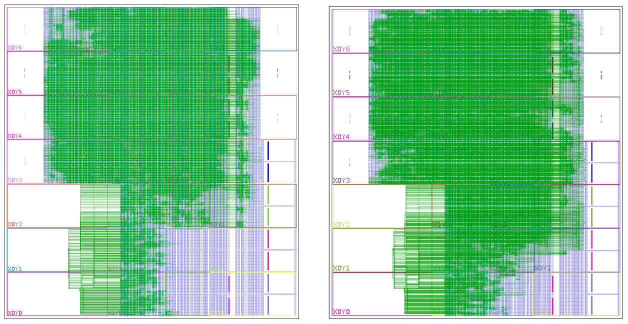
TABLE II Power Performance and Utilization

DNN Model	Power (W)	DSP	BRAM	FF	LUT
AlexNet	4.042	1195	192	82255	126326
VGG16	5.333	1179	618	61034	134706

origin problem into sub-tasks lead to some fluctuations. It is remarkable that the power consumption trend increases as the number of computation engines increases. Considering the situations of using same number of computation engines with different BRAM sizes, obviously, more BRAMs consume more power. Inspired by this experiment, it is reasonable to instantiate less BRAMs to save power without loss of parallelism.

Secondly, two well known DNN models, AlexNet [1] and VGG16 [18] are applied to evaluate the power fidelity of our design flow. We use the 16-bit fixed-point for feature maps and 8-bit fixed-point for weights. A set of initial designs are selected randomly to show the model fidelity, shown in Fig. 7(a) and Fig. 7(b). The x-axis represents the index. To show the order fidelity, all the results are normalized to 1. The power performance estimated by our model have similar orders compared with the back-end synthesis power results. This experiment has shown a clear proof on high fidelity of our power model.

Crucially, a reasonable trade-off between power and latency is also often desired, *i.e.* improvements on power should come with only minimal or acceptable latency degradation. The initial designs in Fig. 7(a) and Fig. 7(b) are used here as the baselines. Some parameters of these designs are fixed and others are free to vary in the search space, to get the corresponding model-optimized configurations. As shown in Fig. 8(a) and Fig. 8(b), the baseline designs are represented as 1 and the optimized configurations are represented with ratios to the baselines. Most designs can achieve more than 10% power benefits with the best even up to 31%. As to the latency loss, most are less than 2% while only one is about 6.5%. It therefore, with the help of the proposed model, becomes intriguing and effective to trade small latency for much better power efficiency.



(a) (b)

Fig. 9 The schematics of our designs: (a) AlexNet; (b) VGG16.

Finally, we show the placement and routing results of VGG16 and AlexNet in Fig. 8 on ZCU102 to further consolidate our designs. As shown in Fig. 9, not all hardware resources are used in our optimized design and routed wires are not very congested. As a result, power consumption can be reduced. The corresponding performance and utilization reports are in TABLE II. We consume similar number of DSPs for computations in two models. The model structure of VGG16 is more friendly to layer fusion, since the model can be divided directly at pooling layers. Each group has two or three convolutional layers and more BRAMs are consumed here. Compared with VGG16, the model structure of AlexNet is more complex therefore the layer fusion is limited. That explains why less BRAMs are consumed for AlexNet.

VI. Conclusion and Discussions

In this paper, we have proposed a power-driven dataflow optimization flow, which can estimate the dataflow configuration power and guide the design efficiently. The results show that better power performance can be obtained at a low cost of latency. There are still several critical challenges existing in dataflow optimization of DNN accelerator, and we expand a little bit in the rest of the section.

Design Space Exploration. In Fig. 5, likes most existing efforts [6], [8], [11], [13], although the tremendously large design space can be effectively reduced by domain knowledge, resource constraints and empirical rules, and hence combinations of design parameters can be almost exhaustively enumerated to determine the optimal solution with the help of simulations [8], [11], the quality of design space exploration still heavily depends on the configuration performance measurement model and the time of simulation. Recently, machine-learning-based design space exploration methodologies have been proposed to achieve good design parameters of circuits without much more simulations [27]–[29], where a set of representative design configurations are selected for simulation so that the predicted performance curve can be accurately fitted with less simulations. The optimal design configuration can be therefore well determined by the trained model. We believe the machine-learning-based design space exploration will be a good option to fast and effectively design DNN accelerators.

Timing Closure. Some low-power DNN FPGA accelerators often result in high device utilization, which potentially imposes difficulty on timing closure [30]. even though systolic array with the local communication and regular layout have been used as the computation core for the best performance. Recently, many arts were proposed to generate better quality of result by tuning design parameters in logic synthesis and physical synthesis [31]–[33]. In addition, FPGA placement and routing methods were customized for different scenarios [34]–[36]. However, performance benefits by either tuning these back-end design parameters or customizing FPGA placement and routing methods are more limited. Note that compared to RTL language, HLS is easier to revise. If revising HLS code, tuning back-end design parameters and customizing FPGA placement/routing methods are considered in a uniform framework, it would bring more benefits for timing closure.

VII. Acknowledgment

This work is partially supported by Tencent Technology, Cadence Design Systems, and The Research Grants Council of Hong Kong SAR (No. CUHK24209017). We also acknowledge the fruitful discussions in experiment with Cong Hao from UIUC, Jianli Chen from Fudan University, Meng Zhang and Zhenyu Zhu from Southeast University.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Conference on Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[2] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3431–3440.

[3] R. Chen, W. Zhong, H. Yang, H. Geng, X. Zeng, and B. Yu, "Faster region-based hotspot detection," in *ACM/IEEE Design Automation Conference (DAC)*, 2019, p. 146.

[4] Q. Zhang, M. Zhang, T. Chen, Z. Sun, Y. Ma, and B. Yu, "Recent advances in convolutional neural network acceleration," *Neurocomputing*, vol. 323, pp. 37–51, 2019.

[5] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[6] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

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

[8] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *ACM/IEEE Design Automation Conference (DAC)*, 2017, pp. 29:1–29:6.

[9] L. Lu and Y. Liang, "SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs," in *ACM/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[10] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 56:1–56:8.

[11] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 45–54.

[12] ——, "Performance modeling for CNN inference accelerators on FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

[13] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of FPGA-based deep convolutional neural networks," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2016, pp. 575–580.

[14] J. Mu, W. Zhang, H. Liang, and S. Sinha, "A collaborative framework for FPGA-based CNN design modeling and optimization," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 139–1397.

[15] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 430–437.

[16] D. Chen, J. Cong, and Y. Fan, "Low-power high-level synthesis for FPGA architectures," in *IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2003, pp. 134–139.

[17] A. Pedram, A. Gerstlauer, and R. A. Van De Geijn, "A high-performance, low-power linear algebra core," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2011, pp. 35–42.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations (ICLR)*, 2015.

[19] Y.-H. Chen, J. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, 2017.

[20] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *IEEE/ACM International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[21] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A customizable matrix multiplication framework for the Intel HARPv2 Xeon+FPGA platform: A deep learning case study," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 107–116.

[22] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 578–594.

[23] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 22:1–22:12.

[24] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," in *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2018, pp. 1354–1367.

[25] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2018, pp. 343–348.

[26] "Xilinx Inc." http://www.xilinx.com.

[27] W. Lyu, F. Yang, C. Yan, D. Zhou, and X. Zeng, "Batch bayesian optimization via multi-objective acquisition ensemble for automated analog circuit design," in *International Conference on Machine Learning (ICML)*, 2018, pp. 3312–3320.

[28] Y. Ma, S. Roy, J. Miao, J. Chen, and B. Yu, "Cross-layer optimization for high speed adders: A pareto driven machine learning approach," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

[29] S. Zhang, W. Lyu, F. Yang, C. Yan, D. Zhou, X. Zeng, and X. Hu, "An efficient multi-fidelity bayesian optimization approach for analog circuit synthesis," in *ACM/IEEE Design Automation Conference (DAC)*, 2019, p. 64.

[30] "UltraFast Design Methodology Guide for the Vivado Design Suite," https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug949-vivado-design-methodology.pdf.

[31] Q. Yanghua, H. Ng, and N. Kapre, "Boosting convergence of timing closure using feature selection in a learning-driven approach," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.

[32] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang, "Lamda: Learning-assisted multi-stage autotuning for FPGA design closure," 2019, pp. 74–77.

[33] Q. Yanghua, C. Adaikkala Raj, H. Ng, K. Teo, and N. Kapre, "Case for design-specific machine learning in timing closure of FPGA designs," in *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016, pp. 169–172.

[34] P. Maidee, C. Ababei, and K. Bazargan, "Timing-driven partitioning-based placement for island style FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 24, no. 3, pp. 395–406, 2005.

[35] C.-W. Pui, G. Chen, W.-K. Chow, K.-C. Lam, J. Kuang, P. Tu, H. Zhang, E. F. Y. Young, and B. Yu, "RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 67:1–67:8.

[36] C.-W. Pui, G. Chen, Y. Ma, E. F. Young, and B. Yu, "Clock-aware ultrascale FPGA placement with machine learning routability prediction," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 929–936.