Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs

Xuechao Wei[1,3†] Cody Hao Yu[2,3†] Peng Zhang[3*]
Youxiang Chen[3] Yuxin Wang[3] Han Hu[3] Yun Liang[1] Jason Cong[1,2,3‡]
[1]Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China
[2]Computer Science Department, University of California, Los Angeles, CA, USA
[3]Falcon Computing Solutions, Inc, Los Angeles, CA, USA
{xuechao,ericlyun}@pku.edu.cn, {cody,pengzhang,youxiangchen,yuxinwang,hanhu,cong}@falcon-computing.com

ABSTRACT

Convolutional neural networks (CNNs) have been widely applied in many deep learning applications. In recent years, the FPGA implementation for CNNs has attracted much attention because of its high performance and energy efficiency. However, existing implementations have difficulty to fully leverage the computation power of the latest FPGAs. In this paper we implement CNN on an FPGA using a systolic array architecture, which can achieve high clock frequency under high resource utilization. We provide an analytical model for performance and resource utilization and develop an automatic design space exploration framework, as well as source-to-source code transformation from a C program to a CNN implementation using systolic array. The experimental results show that our framework is able to generate the accelerator for real-life CNN models, achieving up to 461 GFlops for floating point data type and 1.2 Tops for 8-16 bit fixed point.

1. INTRODUCTION

CNN is one of the key algorithms for the deep learning applications, ranging from image/video classification, recognition, and analysis to natural language understanding, advances in medicine, and more. The core computation in the algorithm can be summarized as a convolution operation on the multiple dimensional arrays. Although the algorithm requires computation power and communication bandwidth, it also offers significant potential for massive parallelization and extensive data reuse. Hence, FPGA implementations of CNN have seen an increased amount of interest from academia [1–10] due to the customizability of FPGAs.

Some existing CNN designs on FPGAs mainly focus on on-chip computation engine optimization by exploiting different parallel strategies [1–4]. The studies explore parallelism opportunities in input feature maps [2] and convolution kernels [1, 4]; while the work in [3] chooses to parallelize output feature maps. These implementations customize massively parallel processing elements (PEs) on FPGAs according to specific computation types; they achieve a high performance that exceeds modern CPUs, thanks to FPGA's abundant logic resources and reconfigurability. On the other hand, some designs take external memory communication into consideration to achieve high throughput at system-level [5–8, 11]. The study in [5] develops a memory-centric design method to maximize data reuse for memory bandwidth optimization. Meanwhile, to balance computation to communication ratio, the study in [6] leverages a roofline model to identify the optimal design option from a large design space, while the authors in [7, 8] propose analytical models to realize this goal. In addition, The authors in [11] quantitatively analyze different optimization objects, and then propose a specific dataflow architecture to minimize data movements and memory accesses. Although those implementations utilize FPGA resources well to achieve high throughput, the capacity of hardware resources in the FPGA increases continuously, which provides more than a thousand floating compute units in one FPGA chip—such as the Intel Arria 10 [12] and Xilinx Virtex UltraScale+ [13]. Once the existing customized designs of CNNs are applied to the latest device, the existing optimization approaches need to deal with the trade-off between high resource utilization and clock frequency, which leads to dramatic performance degradation.

To address such challenges, a suitable architecture for FPGAs plays an important role in developing a scalable CNN implementation. In particular, a systolic array architecture [14] is a specialized form of parallel computing with a deeply pipelined network of PEs. With the regular layout and local communication, the systolic array features low global data transfer and high clock frequency, which is suitable for large-scale parallel design on FPGAs. As a result, systolic array architecture has been widely used for FPGA accelerations, such as matrix multiplication [15] and bioinformatics [16]. As a result, researchers attempt to map CNN inference to systolic array architecture [10, 17] in recent years. Specifically, Caffeine [10] implements the massive parallelism for CNN inference on Xilinx Kintex Untrascale device. The design in [10] adopts a systolic-like architecture to mitigate the timing issue for the large design, but it still directly connects all PEs to the on-chip memory and results in not fully local interconnects. This is the reason that the design in [10] is outperformed by a later work [17] that adopts a complete systolic array architecture. The authors in [17] propose a 1-D systolic array design in OpenCL for AlexNet [18] CNN model with the help of an analytical model to realize the best design point and result in a high throughput design that outperforms all previous designs. However, this design only supports small models such as AlexNet as it assumes that all input feature maps reside in on-chip memory for computation. Moreover, applying the methodology in [17] to other CNN models is not straightforward due to the lack of an automated design space exploration approach. In this paper we investigate the challenges in systolic array implementations in CNNs, and propose an automated methodology to optimize the CNN design on systolic arrays. The major contributions can be summarized in the following.

- **A high-throughput CNN design using systolic array.** We first systematically investigate the benefits and challenges of the systolic design for CNN. Based on our modeling and design space exploration techniques, the end-to-end throughputs with our generated designs are able to achieve up to 461 GFlops and 1.2 Tops.
- **An analytical model and a design space exploration scheme.** We propose an analytical model to analyze performance and resource utilization of systolic designs. To deal with a tremendous design space, we develop an efficient design space exploration by considering hardware features, which reduces optimization runtime from hundreds of hours to less than one minute.
- **An end-to-end automation flow.** We implement a push-button

*These authors contribute equally to this work.
†This work is done while Xuechao Wei and Cody Hao Yu are interns at Falcon Computing Solutions Inc.
‡J. Cong serves as the Chief Scientific Advisor of Falcon Computing Solutions Inc. and a distinguished visiting professor at Peking University, in addition to his primary affiliation at UCLA.

DAC '17, June 18-22, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062207

automation flow to perform CNN design generation from high-level C code to FPGA. No hardware-related, low-level considerations are necessary for end users.

2. CNN SYSTOLIC ARRAY

In this section we first introduce CNN's background and the opportunity for parallelism. We then present the systolic array architecture that we design for CNN implementation. Finally, we present three main challenges to map a CNN onto a systolic array architecture.

2.1 Background of CNN on FPGA

A convolutional neural network is a typical deep learning neural network that is adopted in applications like image and video processing. In recent years CNN has evolved quickly, especially with a boost from the visual recognition challenge (ImageNet [18]). CNN models, such as AlexNet, VGG, and GoogleNet [18–20], consist of several to hundreds of cascaded layers. Although these networks vary significantly in terms of topology and complexity, the basic computations in each layer are common—such as convolutional, fully connected, pooling, sigmod and ReLU [21]. Among these computation blocks, convolutional and fully connected layers contribute over 90% of the computational complexity. Despite the fact that we have implemented entire AlexNet and VGG16 models on FPGAs, and fully connected layers can be converted into convolutional layers [10], in the remainder of this paper we focus on the systolic array architecture synthesis and optimization for convolutional layers.

Code 1: C Code of a Convolutional Layer

```
L1: for(o = 0; o < O; o++) // Output feature #
L2: for(i = 0; i < I; i++) // Input feature #
L3: for(c = 0; c < C; c++) // Feature column
L4: for(r = 0; r < R; r++) // Feature row
L5: for(p = 0; p < K; p++) // Kernel weight
L6: for(q = 0; q < K; q++) // Kernel height
  OUT[o][r][c] += W[o][i][p][q] *
                  IN[i][r+p][c+q];
```

A simplified code of the convolutional layer can be summarized in Code 1. Although the computation pattern is as simple as multiplication and accumulation, the calculation requires a huge volume of both computation power and data transfer bandwidth.

On the other hand, the algorithm also provides considerable potential for both the massive parallelism and intensive data reuse. In the original six-level nested loop, three (*L1*, *L4*, *L3*) are parallelizable because they do not have data dependency; the remaining loops (*L2*, *L5*, *L6*) have dependency carried for the accumulation of array *out*. However, these loops are still parallelizable by leveraging the associative law of the addition operations.

The state-of-the-art FPGAs claim to have a peak performance of over 1TFlops with thousands of floating point DSP blocks, but fully exploiting the computing power from many of distributed DSP blocks is not trivial efficient FPGA designs, which require full pipelining and massive parallelization on all the DSP blocks. Traditional methods apply loop unrolling on multiple for-loops to enable massive parallelization [6] and perform memory partitioning to feed the multiple data into different DSP blocks [5]. Each copy of the computation unit in the array is called a process element (PE). PEs are connected to memory blocks directly. While they are able to achieve the massive parallelization with full pipelining in the architecture design, the implementation of the design may have difficulty in making the timing closure for a high clock frequency, or even passing the place and route. The underlying reasons for the timing issue are three-fold. First, large-scale data reuse between DSP blocks introduces large fan-out. Second, these direct interconnects become long wires when the DSP blocks are distributed among the whole FPGA chip. Third, large multiplexes are required for the output data collection.

2.2 Systolic Array Architecture for CNN

We present a novel 2-D systolic array architecture for CNN on FPGA in Fig. 1. As shown in this figure, each PE shifts the data of *W* and *IN* horizontally and vertically to the neighboring PEs at each cycle. This 2-D topology matches the 2-D structure in the FPGA layout so that it can achieve timing constraints easily because of low routing complexity. In addition, there is a SIMD vector accumulation inside each PE. The parallelization factor of the SIMD factor is usually power of two due to the dedicated inter-DSP accumulation interconnect in modern FPGAs.

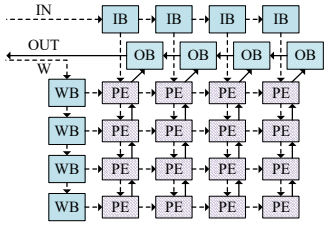


Figure 1: Systolic Array Architecture for CNN

This architecture is able to tackle the timing issue for massive parallelization within CNN. Its key features can be summarized as 1) local interconnect and 2) shifting data transfer. As shown in Fig. 1, the global and large fan-out interconnect is split into local interconnects between neighboring PEs. In addition, the input/output data are shifted into/from the PE array and between the neighboring PEs so that the multiplexes are eliminated. With the local, short, peer-to-peer interconnects, systolic array architecture can achieve high frequency even in the case of massive parallelization with over a thousand PEs.

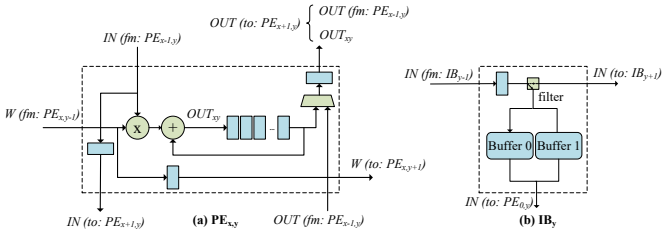


Figure 2: Structure of PE and Input Feature Map Buffer (IB)

The architectures of a PE with index (x, y) and buffer structure to store input feature maps (IB) are presented in Fig. 2. For the $PE_{x,y}$ in Fig. 2 (a), it passes input data down to $PE_{x+1,y}$ and passes weight data right to $PE_{x,y+1}$ at each cycle. The PE also accumulates the product of the input and weight data ($OUT_{x,y}$) that passed from its neighboring PEs. In addition, the output data are also shifted out across the PE array, so the output of $PE_{x,y}$ is either the result produced by itself ($OUT_{x,y}$) or the shifted result from its neighboring PEs.

Moreover, for the input buffer (IB) shown in Fig. 2 (b), double buffering is enabled for the pipelining. All the input feature map data are shifted across the IB chain as a pipeline while each IB selectively stores the data that belongs to the corresponding column of PEs. The similar structure is applied to the buffers for weight (WBs) and output feature map (OBs) as well so that the entire systolic array architecture can be fully pipelined.

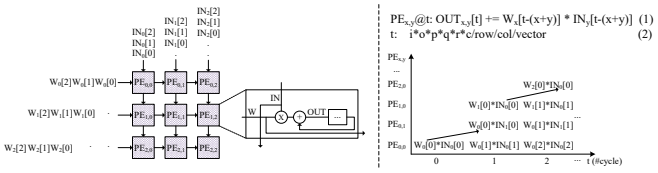


Figure 3: An Example of Systolic Array Cycle-Level Scheduling

There are several considerations related to systolic array execution. First, the data required for the computation of the PEs have to be transferred from the boundary PE and across multiple PEs. Since only the boundary PE connects to input data, data reuse between each row and column of PEs is required. More importantly, a systolic array runs in a regular and synchronized way such that fine-grained pipelining is performed between every neighboring

PE. Therefore, a suitable scheduling of the PE executions is essential for systolic array design, especially the synchronization of the data on each PE from different directions. Fig. 3 shows one possible scheduling of PE execution after performing the feasible mapping that determines which loop corresponds to which systolic array dimension (see Section 3.2 for detailed discussions). We use $PE_{x,y}@t$ to denote the mapping of cycle number t and PE indexes (x, y) onto data access indexes. It means that at cycle t, the product of W and IN accumulates on OUT, as shown by the equation (1) in Fig. 3. Cycle number t is computed by dividing the total loop number by $row \times col \times vector$ as shown by equation (2) in Fig. 3, where row and col are the number of PEs in row and column, and $vector$ is the number of SIMD computation within each PE. As shown in the right part of Fig. 3, $PE_{0,0}$ gets weight data W from buffer WB and input feature map data IN from buffer IB at the first cycle. $PE_{0,0}$ performs the multiplication of the two inputs and accumulates the result OUT in the register within the PE along with previous partial accumulated results. Meanwhile, the other PEs are stalled because no data is being received from at least one of its inputs. At cycle 1, the data W from $PE_{0,0}$ is passed to $PE_{0,1}$ and the data IN is passed to $PE_{1,0}$. As a result, both $PE_{0,1}$ and $PE_{1,0}$ have the required data to perform an execution. At the same cycle, $PE_{0,0}$ is able to perform the execution with new data coming from the input buffers as well. As can be seen for the 3×3 systolic array example shown in Fig. 3, all PEs are active after five cycles. Thus, they can synchronously read data from their neighboring PEs, perform computation and pass data to the next PEs simultaneously in each cycle. After the in-PE computation has been finished, finally, OUT in the shift register is shifted across vertical PEs to the corresponded OB.

2.3 Challenges in CNN Systolic Array

Although the systolic array architecture is able to significantly benefit designs on the FPGA, mapping a CNN computation onto a systolic array structure is not straightforward. We summarize the mapping process in three steps and describe their challenges along with examples using the configuration of AlexNet [18] layer 5, $(I, O, R, C, P, Q) = (192, 128, 13, 13, 3, 3)$, as follows:

1. Find a feasible mapping. We need to first find a feasible mapping in the systolic array to guarantee that the proper data is available at specific locations in the PE array at every cycle. Specifically, we attempt to select three loops in Code 1 to represent the 3 dimensional parallelism of the 2-D systolic array: PE row, PE column and the SIMD vector inside a PE. As mentioned in the previous section, systolic array requires data reuse in both directions, so the corresponding loops need to carry the data reuse of two different arrays (*W* and *IN*), while the third loop needs to carry the accumulation of the output (*OUT*). Failing to satisfy this rule will cause a non-feasible mapping. For example, mapping loop *L3* and *L4* into a PE row and column is not feasible because data reuse does not happen on array *W* which does not relate to either loop *L3* or *L4*.

2. Select a PE array shape. Next, we select the PE array shape by determining the size of each dimension, which impacts the performance in terms of 1) the required DSP number, 2) the clock frequency, and 3) the DSP efficiency. The DSP efficiency is defined as the effective computation ratio performed by DSPs:

$$Eff = \frac{effective\ operation\ no.}{total\ operation\ no.} \quad (1)$$

We use an example in Table 1 to illustrate the impact of systolic array shape. Both configurations map loop $(L1, L3, L2)$ to systolic arrays $(row, column, vector)$ but with different shapes. As can be seen, assuming the clock frequency for both configurations are the same (280 MHz), *sys2* has a higher DSP utilization but a relatively lower DSP efficiency compared with *sys1*. This is because *sys2*'s shape $(16, 10, 8)$ does not match the mapped trip counts of the mapped loops $(128, 13, 192)$.

3. Determine the data reuse strategy. After we identify the systolic array mapping and shape, we determine the data reuse strategy by choosing proper tiling sizes to achieve extensive data reuse. In other words, it requires exploiting the data reuse carried on multiple for-loops with long reuse distance, which in turn leads to the large reuse buffers. However, there are more than ten thousand design options in the trade-off between the on-chip memory utilization and off-chip bandwidth saving, including selection of the arrays to be reused, the loops that carry the data reuse, and tiling sizes for the selected loops carrying data reuse.

Table 1: Impact of Systolic Array Shape to Performance

	ROW	COL.	VEC.	DSP Util.	DSP Eff.	Peak Thrpt.
sys1	11 (L1)	13 (L3)	8 (L2)	71.5%	96.97%	621 GFlops
sys2	16 (L1)	10 (L3)	8 (L2)	80.0%	60.00%	466 GFlops

Taking *sys1* in Table 1 again as an example, since the systolic array design we used is fully pipelined, the theoretical peak throughput is $96.97\% \times 2 \times 11 \times 13 \times 8 \times 280 \simeq 621$ GFlops. This can be achieved by choosing proper tiling sizes for each loop (e.g., $Tile(I, O, R, C, P, Q) = (4, 4, 13, 1, 3, 3)$) to balance data reuse and memory bandwidth. However, if we use inappropriate tiling sizes such as $Tile(I, O, R, C, P, Q) = (2, 2, 2, 2, 2, 2)$, then we require around 67 GB/s memory bandwidth to achieve the peak throughput (the analytical model is described in Section 3). In fact, we only get 162 GFlops on Intel's Arria 10 with 19 GB/s bandwidth for this low QoR configuration.

3. ANALYTICAL MODELS

All these design challenges and their interplay need to be considered in a unified way with high-level modeling. In this section, we formulate the overall optimization problem as maximizing the system throughput under the systolic mapping feasibility condition and resource constraints.

3.1 Architecture Abstraction

Before we can perform the detailed modeling, an abstraction of the architecture is necessary. A loop tiling representation is proposed in Fig. 4 for this purpose, which establishes the link between the architecture and high-level program code. The tiled loops in the intermediate representation contains all the architecture considerations in the systolic array, such as PE array mapping, PE array shape, data reuse strategy, etc. This representation itself is a sequential program, which enables us to perform the modeling in a general way using program analysis techniques and tools such as polyhedral model.

```
// outer loops
L0: for (Lo = 0; Lo < l0; Lo++)
...
Ln: for (Ln = 0; Ln < ln; Ln++)
// middle loops
S0: for (So = 0; So < s0; So++)
...
Sn: for (Sn = 0; Sn < sn; Sn++)
// inner loops
T0: for (T0 = 0; T0 < t0; T0++)
T1: for (T1 = 0; T1 < t1; T1++)
T2: for (T2 = 0; T2 < t2; T2++)
  // Orignal loop body
```

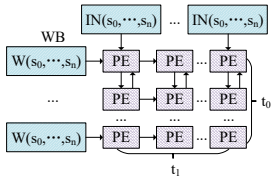


Figure 4: Loop Tiling Representation for Systolic Array Mapping

The program in Fig. 4 is transformed from the original code in Code 1 by loop tiling. The semantic of the program is preserved by the transformation if we ignore the precision error of reordering the floating point accumulation. Then the tiled loops are associated with the architecture consideration as the following. The overall computation is performed block by block sequentially, where each block is corresponding to an iteration of the ***outer loops*** (L_0-L_n). Since the blocks are calculated independently, the outer loops do not impact the throughput.

Once a data block is fetched from off-chip memory, it is stored in the input buffers (IB and WB) for date reuse. The ***middle loops*** (S_0-S_n) represent the sequential processing of feeding data from input buffers to the PE array. The bounds of the middle loops $\vec{s} = (s_0, ..., s_n)$ determine the sizes of the reuse buffer. The data accessed by the PE is represented by the array access address which is indexed by the iterator of middle loops.

Parallel execution is performed in the PE array in the fine-grained pipeline. The ***inner loops*** (T_0-T_2) represent the parallelism in the PE array where each iteration of the inner loops is corresponding to a parallel DSP unit in the array. The shape of the systolic array

is determined by the bounds of inner loops ($\vec{t} = (t_0, t_1, t_2)$); while the systolic array mapping feasibility is determined by the relation between the inner loop iterators and the array access addresses in the loop body.

In addition, when we perform the loop tiling, the original loop bounds may not be divisible by the tiling sizes ($\vec{s}$, $\vec{t}$) we selected. This leads to a waste of computation that affects the DSP efficiency.

3.2 Feasible Mapping to Systolic Array

As demonstrated in Section 2, the architecture of the systolic array is determined by the three inner loops that are selected to map to PE row, PE column and SIMD vector inside the PE. There are many alternatives for this loop-to-architecture mapping, but not every one of them can finally have a feasible mapping in the systolic fashion. The condition of the feasible systolic mapping can be summarized as: *each of the three array variables (W, IN, and OUT) has to have fine-grained data reuse carried out at least one of the three inner loops.*

We formulate the condition for feasible mapping by introducing the binary variables k_l to indicate loop-to-architecture mapping ($k_l = 1$ if the loop l is selected as one of the inner loops; otherwise $k_l = 0$):

$$\sum k_l = 3, \forall r, \sum k_l \times c_{rl} > 0 \tag{2}$$

where c_{rl} indicates data reuse of array r on loop l: $c_{rl} = 1$ if loop l carries the fine-grained data reuse of array r; otherwise $c_{rl} = 0$.

In addition, all the possible fine-grained data reuses in a program can be analyzed in advance. Since the fine-grained data reuse for array r on loop l requires the data access on array r in different loop l iterations to be the same, we use polyhedral model [22] to represent this condition:

$$\forall \vec{i} \in \mathcal{D}, F_r(...i_{l-1}, i_l, i_{l+1}, ...) = F_r(i_0, ..., i_{l-1}, i_l + 1, i_{l+1}, ...) \tag{3}$$

where $\vec{i}$ is an iterator vector which lists loop iterators from the outermost loop to the inner loop in the loop nest; $\mathcal{D}$ is an iteration domain which defines the range of the loop iterators; F_r is an access function which maps the loop iterators into the access indexes of array r.

3.3 Resource Utilization Modeling

Since the computation is mainly floating-point multiplication and accumulation, the DSP and on-chip block RAM (BRAM) are the two critical resource types. The DSP utilization is simply determined by the product of the inner loop bounds $\vec{t}$ and the DSP usage per PE that depends on the data type width and the FPGA platform:

$$D(\vec{t}) = DSP\_per\_PE \times \prod t_l \tag{4}$$

On the other hand, the modeling of BRAM utilization needs to consider the data reuse in the input and output buffers. Due to the double buffering mechanism for hiding data transfer overhead, the buffer size is equal to two times the data block size of the array. The block size can be modeled as the total amount of data that is accessed in the middle and inner loops in Fig. 4.

$$DA_r(\vec{s}, \vec{t}) = \left| \left\{ \vec{a} | \vec{a} = F_r(\vec{i}) \wedge \vec{i} \in \mathcal{D}_{\vec{s},\vec{t}} \right\} \right| \tag{5}$$

where $\vec{s}$ and $\vec{t}$ are again the bounds of middle and inner loops, respectively; $\mathcal{D}_{\vec{s},\vec{t}}$ is the iteration domain of the middle and inner loops; $\vec{a}$ is the access index vector of a multi-dimensional array. Counting an integer set with linear constraints can be solved by the polyhedral library [23], but it has high computational complexity. By leveraging the feature of CNN algorithm, we can simplify the model by counting the range of each dimension of the array access index, so that the total size is the product of range size of each dimension. CNN only have two kinds of index patterns in the program: the one consists of only one iterator, such as *out[o][r][c]* and *w[i][o][p][q]*; and the other is the sum of two iterators, such as *r+p* in the access *in[i][r+p][c+q]*. For the first case, the range for the dimension is the bound of the corresponding middle and inner loops. For the second case, the range can be calculated as the sum of the bound of the two iterators, e.g. if the bound of r is t_0 and bound of p is 3, the range size of $r + p$ is $(t_0 + 3) - 1$.

To simplify the address generation complexity of multiple dimensional arrays, the Intel OpenCL design flow tool will allocate the actual memory size as the rounding up power of two value. Finally, the total BRAM utilization is formulated as follows:

$$B(\vec{s}, \vec{t}) = \sum_r (c_b + 2^{\lceil \log_2 DA_r(\vec{s},\vec{t}) \rceil}) + (c_p \times \prod(\vec{t})) \tag{6}$$

where c_b is a constant BRAM cost for the IBs and OBs, c_p is the BRAM cost for each PE, and $\vec{t}$ is the bound vector of inner loops.

3.4 Performance Modeling

In the CNN systolic design, both computation and data transfer may be the performance bottleneck for different design options. The adoption of double buffering in the input and output enables us to model the throughput in a decoupled way, so the overall throughput T is dominated by the lower one of computation throughput (PT) and external memory transfer throughput (MT).

$$T(\vec{s}, \vec{t}) = min(PT(\vec{s}, \vec{t}), MT(\vec{s}, \vec{t})) \tag{7}$$

Since the systolic array is executed in the fully pipelined way, each PE will complete two floating point operations (multiplication and accumulation) in each cycle. However, the quantization effect (described in Section 2.3) may lead to wasted computation on the incomplete data blocks on the boundaries of the original loops. By defining the clock frequency as F, the computational throughput is modeled as the number of effective floating operations in the original code performed every second:

$$PT(\vec{s}, \vec{t}) = Eff(\vec{s}, \vec{t}) \times \prod \vec{t} \times 2 \times F \tag{8}$$

where $Eff(\vec{s}, \vec{t})$ is the DSP efficiency defined in Eq. 1.

In addition, external memory transfer throughput (MT) is defined as the number of effective floating point operations performed in each block divided by the time it takes to transfer the data required by these operations. Due to the hardware feature, the memory bandwidth limitation is not only for overall memory access BW_{total}, but for each memory access port BW_{port} (array in, w, and out). The transferred data amount and bandwidth determines the data transfer time, so MT can be modeled as follows:

$$MT(\vec{s}, \vec{t}) = min(MT_t(\vec{s}, \vec{t}), MT_r(\vec{s}, \vec{t})), r \in R \tag{9}$$

$$\begin{aligned} MT_t(\vec{s}, \vec{t}) &= \frac{Eff(\vec{s}, \vec{t}) \times 2 \times \prod(\vec{s} \times \vec{t})}{\sum DA_r(\vec{s}, \vec{t}) / BW_{total}} \\ MT_r(\vec{s}, \vec{t}) &= \frac{Eff(\vec{s}, \vec{t}) \times 2 \times \prod(\vec{s} \times \vec{t})}{DA_r(\vec{s}, \vec{t}) / BW_{port}} \end{aligned} \tag{10}$$

3.5 Putting It All Together

Finally, the overall optimization problem can be formulated as the combination of the following two subproblems.

Problem 1: Given a nested loop L that functions as CNN, finding a set S that contains all feasible systolic array configurations:

$$\mathcal{S}_L = \left\{ (\vec{k}, \vec{t}) \mid \sum \vec{k} = 3, \prod \vec{t} \leq D_{total}, \forall r, \sum k_l \times c_{rl} = 1 \right\} \tag{11}$$

where $\vec{k}$ is the mapping vector mentioned in Section 3.2, $\vec{t}$ is the bounds of the inner loops and D_{total} is total DSP numbers.

Problem 2: Given a systolic array configuration $(\vec{k}, \vec{t})$, finding the optimal bounds of the middle loops $\vec{s}$ so that the overall design throughput is maximized:

$$maximizing\ T(\vec{s}, \vec{t}), \text{s.t. } B(\vec{s}, \vec{t}) < B_{total}, D(\vec{t}) < D_{total}$$

where T, B, and D have been defined in Eq. 7, Eq. 6, and Eq. 4, respectively.

The complex calculation of $B(\vec{s}, \vec{t})$ and $H(\vec{s}, \vec{t})$ makes the problem neither linear nor convex, which in turn leads to the difficulty in analytical solving. On the other hand, the entire design space of the two problems is tremendously large, which makes brute-force search impractical. In fact, our implementation spends roughly 311 hours on traversing every design option for one of the convolutional layers from the AlexNet [18] CNN model on Intel's Xeon E5-2667 CPU with 3.2GHz frequency. In the next section, we will show that the size of design space can be reduced significantly when taking practical hardware architecture into consideration.

4. DESIGN SPACE EXPLORATION

Under the performance and resource modeling, our design space exploration identifies a valid design option with the highest throughput. However, the working frequency for a design is hard to model. As a result, we develop a two-phase process in Fig. 5 which first filters the design space into a small set of candidates using the proposed analytical model in Section 3 with a given clock frequency, and then goes through the hardware generation flow for the selected designs to obtain the one that has the best on-board performance.

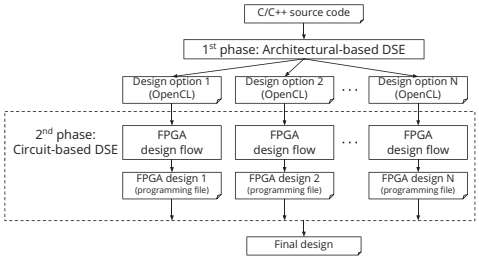


Figure 5: Two-Phase Design Space Exploration

In the architectural-based phase, we reduce the design space by considering resource utilization and on-chip BRAM features. Due to the scalability of the systolic PE array architecture we adopted, the clock frequency will not drop significantly with low DSP utilization, so we can prune the design options with low DSP utilization by adding the following constraint into Problem 1.

$$D(\vec{t}) \geq c_s \times D_{total} \tag{12}$$

where c_s is a constant to set a lower bound of DSP utilization defined by a user. The value of c_s determines the design space of the rest of the process. For example, by applying Eq. 12 with $c_s = 80\%$, the number of available systolic PE array mappings is reduced from 160K to 64K for one of the convolutional layers from AlexNet [18].

In addition, we also reduce the design space of data reuse strategies in terms of value of $\vec{s}$ by leveraging the fact that BRAM sizes in the implementation are always rounded up to the power of two. In details, we prune the design space by only exploring the candidates of $\vec{s}$ whose values are the power of two. The pruned design space of data reuse strategies can still cover the optimal solution in the original design space because 1) our throughput object function is a monotonic non-decreasing function of $\vec{s}$, and 2) BRAM utilization is the same for the options of $\vec{s}$ whose values have the same rounding up the the power of two. By applying the pruning on the data reuse strategies, the design space reduces exponentially so that we are able to perform an exhaustive search to find the best strategy and result in an additional 17.5× saving on the average search time for AlexNet convolutional layers.

Consequently, the first phase of our design space exploration process takes less than 30 seconds to identify a set of high throughput design options instead of hundreds of hours. In the second phase, designs in the set are then synthesized using an Intel SDK for OpenCL Application [24] to realize the clock frequency. We use the actual frequency to refine the performance estimation for deciding the best systolic array design.

5. IMPLEMENTATION AND EXPERIMENT

5.1 End-to-end Automation Flow

We implement a push-button design flow framework to generate an executable system on FPGAs from a user-written intuitive CNN program in Fig. 6. A user only needs to specify the nested loop that functions as a CNN layer using a pragma, as shown in the left side of Fig. 6. Our automation flow shown in the right side of Fig. 6 first analyzes the user program using the ROSE compiler infrastructure [25] to obtain necessary information such as iteration domains and data access patterns. Subsequently, we perform design space exploration to identify multiple valid design options with the highest estimated throughput. The design options are parameterized to instantiate template files, including OpenCL systolic array implementation (kernel), as well as the C/C++ software program (host).

```
#pragma ACCEL systolic auto
for(o = 0; o < O; o++)
for(i = 0; i < I; i++)
for(c = 0; c < C; c++)
for(r = 0; r < R; r++)
for(p = 0; p < K; p++)
for(q = 0; q < K; q++)
  out[o][r][c] +=
       w[o][i][p][q] *
        in[i][r+p][c+q];
```

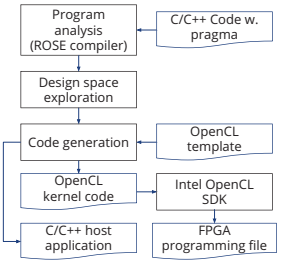


Figure 6: Programming Model and Execution Flow

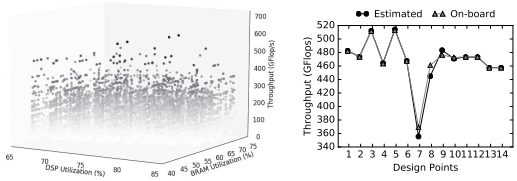
Finally, the instantiated OpenCL kernel is synthesized by the Intel FPGA SDK for OpenCL [24] for the physical implementation.

5.2 Experimental Setup

We evaluate our model and systolic array architecture design in Intel's Arria 10 GT 1150 board which contains 1518 hardened floating point DSPs. The underlying OpenCL implementation of the systolic array design is synthesized using the Intel SDK 16.0 for OpenCL application [24]. We adopt two widely used real-life CNN models, AlexNet [18] and VGG16 [19], for evaluation. We use 32-bit floating points and fixed points to evaluate our framework. For fixed point evaluation, we use 8-bit data type for weights and 16-bit for pixels, by which the top-1 and top-5 ImageNet classification accuracy degradation could be less than 2% [11]. For each model, we generate the design with the optimal performance for all layers according to our two-phase DSE process.

5.3 Results and Analysis

In this experiment, we use a unified systolic array design configuration for all the convolutional layers in each CNN model instead of making an optimal design for each layer, because it has big performance overhead to reprogram the FPGA for different layers. For example, Fig. 7 (a) depicts all valid design options of AlexNet convolutional layers using floating point precision with a given clock frequency (280 MHz) reported by our framework. The density for each design point represents the throughput where darker means higher. As can be seen, high throughput design options may cost moderate BRAM blocks and DSPs due to lower design overhead. This motivates the first phase of our design space exploration. In addition, since the frequency is a given constant value, Fig. 7 (a) is not able to reflect the impact of different clock frequencies.



(a) Pruned Design Space (b) Model Accuracy Analysis

Figure 7: Design Space and Analytical Model Analysis for AlexNet

To deal with the impact of frequency variant, we use the top 14 design options from Fig. 7 (a) and perform P&R at the same time to realize the actual clock frequency. Fig. 7 (b) shows a comparison of on-board results against the analytical model of all 14 designs sorted by estimated throughput. As can be seen, our design space exploration identifies 6 designs with the highest estimated throughput. It means that those designs have the same, minimum computation overhead but adopt different data reuse strategies. This difference results in different clock frequencies at the P&R stage of the design flow, and it is hard to be predicted in advance. According to Fig. 7 (b), our model perfectly matches the on-board results ($< 2\%$ error on average) by using the real working frequency. This illustrates the accuracy of our analytical model.

Table 3 shows the working frequency, resource utilization, and the systolic array design configuration we used for each CNN model as an order of PE row, column and vector [1]. We can see that the de-

[1] We only show the by-layer throughputs for floating point precision due to the similar throughput trend and page limit.

Table 2: Comparison to State-of-the-art Implementations

	[9]	[10]	[10]	[11]	[17]	[26]		Ours		
FPGA	Altera Stratix-V	Xilinx VC709	Xilinx KU060	Arria10 GX 1150	Arria10 GX 1150	Arria10 GX 1150		Arria10 GT 1150		
Frequency (MHz)	120	150	200	150	303	370	385	239.62	221.65	231.85
CNN	VGG	VGG	VGG	VGG	AlexNet	VGG	VGG	AlexNet	VGG	VGG
Precision	fixed 8-16 bit	fixed 16 bit	fixed 16 bit	fixed 8-16 bit	float 16 bit	float 32 bit	fixed 16 bit	float 32 bit	float 32 bit	fixed 8-16 bit
Logic Utilization	153K (25%)	300K (81%)	100K (31%)	161K (38%)	246K (58%)	N/A	N/A	350K (82%)	354K (83%)	313K (73%)
DSP Utilization	727 (37%)	2833 (78%)	1058 (38%)	1518 (100%)	1476 (97%)	1320 (87%)	2756 (91%)	1290 (85%)	1340 (88%)	1500 (49%)
BRAM Utilization	1500 (58%)	1248 (42%)	782 (36%)	1900 (70%)	2487 (92%)	1250 (46%)	1450 (54%)	2360 (86%)	2455 (90%)	1668 (61%)
Latency/Image (ms)	262.9	65.13	101.15	47.97	1.06	35.5	17.18	4.05	54.12	26.85
Throughput	117.8	354	266	645.25	1382	866	1790	360.4	460.5	1171.3

Table 3: Frequency and Resource Utilization

Model	PE shape	Freq. (MHz)	LUT	DSP	BRAM	FF
AlexNet	(11,14,8)	270.8	57%	81%	45%	40%
VGG	(8,19,8)	252.6	59%	81%	47%	40%

Table 4: Throughput for Convolutional Layers of AlexNet

Layer	**1**	**2**	**3**	**4**	**5**	**Avg.**
Thrpt.	123.5	225.0	541.7	541.6	600.0	406.1
DSP Eff.	18.51	33.70	81.03	81.03	90.00	40.32

Table 5: Throughput for Convolutional Layers of VGG16

Layer	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Thrpt.	223.86	450.11	600.27	601.69	601.57	602.44	602.44
DSP Eff.	36.36	72.73	96.97	96.97	96.97	96.97	96.97
Layer	**8**	**9**	**10**	**11**	**12**	**13**	**Avg.**
Thrpt.	602.42	602.83	602.83	602.49	602.49	602.49	561.38
DSP Eff.	96.97	96.97	96.97	96.97	96.97	96.97	89.11

signs generated by our framework have high resource utilization and suitable shapes that match most of the layers in CNN models.

The performance of the two designs is shown in Table 4 and Table 5, respectively. We can see that most of the layers of the two CNN models achieve near-peak performance. However, the throughput and DSP efficiency of AlexNet's layer 1 are much lower than other layers. For two reasons. First, layer 1 has only 3 large input feature maps which make the shape of layer 1 quite different from other layers so that a common design for all layers including layer 1 is hard to find. As a result, we folded layer 1 to have more small feature maps to make its configuration more consistent with others. Second, the kernel size (11) of layer 1 is much larger than others (5 and 3). In order to obtain one design for all layers, our framework chose the data reuse strategy that benefit other layers more. Although the selected data reuse strategy is able to let other layers achieve high throughput, it causes the throughput of layer 1 to be bounded by memory bandwidth. In addition, the layer 1 of VGG16 has a lower performance than other layers as well. This is because the layer 1 image row number (16) is inconsistent with other layers, and lead to low DSP utilization of PEs' parallelism and pipelining. However, VGG16 still has a better overall performance than AlexNet since it has a more regular network shape that shows better scalability for its uniform hardware design.

We finally compare the end-to-end results for both models with state-of-the-art CNN designs in Table 2. We use latency for processing one image and throughput for performance comparison. As can be seen, our performance outperforms all previous work except for [17] and [26]. The work in [17] improves DSP utilization by adopting Winograd transformation [27] that is planned to be included in our design in the future. According to the throughput improvement reported by [17], the throughput of our designs can be potentially improved by 2× if applied Winograd transformation. On the other hand, the authors in [26] implement the accelerator kernel using System Verilog and wrap the kernel to an OpenCL IP. Consequently, low-level design optimizations such as register-level optimization (e.g. limiting the maximum fan-out number for registers) can be applied to guarantee a higher frequency. Since the design of [26] includes such CNN model dependent optimization, it is hard to be adapted for other models.

6. CONCLUSIONS AND FUTURE WORK

In this paper we propose a systolic array architecture for high throughput CNN on FPGAs. We solve the challenges of PE mapping, shape selection and data reuse strategy by accurate modeling techniques and effective design space exploration strategies. We also implement a framework to automate CNN to the systolic array mapping process. Evaluation results show that our design achieves up to 1171 Gops on Intel's Arria 10 device.

There are several directions that we can investigate further in the future. In particular, existing work [17, 28, 29] has demonstrated that applying Winograd [27] and fast Fourier transformations to convolutional computation can significantly improve resource efficiency. We believe that these transformations could also benefit our architecture and further improve the throughput.

7. ACKNOWLEDGMENT

The author would like to thank Intel for providing the Arria 10 board and reference design of systolic matrix multiplication.

8. REFERENCES

[1] S. Cadambi *et al.*, "A Programmable Parallel Accelerator for Learning and Classification," in *PACT*, 2010.
[2] M. Sankaradas *et al.*, "A Massively Parallel Coprocessor for Convolutional Neural Networks," in *ASAP*, 2009.
[3] S. Chakradhar *et al.*, "A Dynamically Configurable Coprocessor for Convolutional Neural Networks," *ISCA*, 2010.
[4] C. Farabet *et al.*, "CNP: An FPGA-based processor for Convolutional Networks," in *FPL*, 2009.
[5] M. Peemen *et al.*, "Memory-centric accelerator design for Convolutional Neural Networks," in *ICCD*, 2013.
[6] C. Zhang *et al.*, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015.
[7] N. Suda *et al.*, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *FPGA*, 2016.
[8] S. I. Venieris *et al.*, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *FCCM*, 2016.
[9] J. Qiu *et al.*, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *FPGA*, 2016.
[10] C. Zhang *et al.*, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in *ICCAD*, 2016.
[11] Y. Ma *et al.*, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *FPGA*, 2017.
[12] Intel Arria 10.
[13] Xilinx Ultrascale Architecture.
[14] H. T. Kung *et al.*, *Algorithms for VLSI Processor Arrays*, 1979.
[15] J. Wang *et al.*, "Customizable and High Performance Matrix Multiplication Kernel on FPGA," in *FPGA*, 2015.
[16] A. C. Jacob *et al.*, "Design of Throughput-Optimized Arrays from Recurrence Abstractions," in *ASAP*, 2010.
[17] U. Aydonat *et al.*, "An OpenCL Deep Learning Accelerator on Arria 10," in *FPGA*, 2017.
[18] A. Krizhevsky *et al.*, "ImageNet Classification with Deep Convolutional Neural Networks," in *NIPS*, 2012.
[19] K. Simonyan *et al.*, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv*, 2014.
[20] C. Szegedy *et al.*, "Going Deeper with Convolutions," *arXiv*, 2014.
[21] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *MM*, 2014.
[22] D. L. Kuck, *Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.
[23] S. Verdoolaege, "Isl: An Integer Set Library for the Polyhedral Model," in *ICMS*, 2010.
[24] Intel SDK for OpenCL Applications.
[25] ROSE Compiler Infrastructure.
[26] J. Zhang *et al.*, "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network," in *FPGA*, 2017.
[27] S. Winograd, *Arithmetic Complexity of Computations*, 1980.
[28] C. Zhang *et al.*, "Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System," in *FPGA*, 2017.
[29] L. Lu *et al.*, "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs," in *FCCM*, 2017.