Model-Platform Optimized Deep Neural Network Accelerator Generation through Mixed-Integer Geometric Programming

Yuhao Ding*,1, Jiajun Wu*,1, Yizhao Gao[1], Maolin Wang[2], Hayden Kwok-Hay So[1]
[1]*Department of Electrical and Electronic Engineering*, *University of Hong Kong*, Hong Kong
[2]*AI Chip Center for Emerging Smart Systems*, Hong Kong
{yhding, jjwu, yzgao}@eee.hku.hk, maolinwang@ust.hk, hso@eee.hku.hk

***Abstract*—Although there are distinct power-performance advantages in customizing an accelerator for a specific combination of FPGA platform and neural network model, developing such highly customized accelerators is a challenging task due to the massive design space spans from the range of network models to be accelerated, the target platform's compute capability, and its memory capacity and performance characteristics. To address this architectural customization problem, an automatic design space exploration (DSE) framework using a mixed-integer geometric programming (MIGP) approach is presented. Given the set of DNN models to be accelerated and a generic description of the target platform's compute and memory capabilities as input, the proposed framework automatically customizes an architectural template for the platform-model combination and produces the associated I/O schedule to maximize its end-to-end performance. By formulating DNN inference as a multi-level loop tiling problem, the proposed framework first customizes an accelerator template that consists of a parameterizable array architecture with SIMD execution cores and a customizable memory hierarchy using a MIGP to maximize the expected resource utilization. Subsequently, a second MIGP is used to schedule memory and compute operations as tiles to improve on-chip data reuse and memory bandwidth utilization. Experimental results from a wide range of neural network models and FPGA platform combinations show that the proposed scheme is able to produce accelerators with performance comparable to the state-of-the-art. The proposed DSE framework and the resulting hardware/software generator are available as an open-source package called AGNA with the hope that it may facilitate vendor-agnostic DNN accelerator development from the research community in the future.**

I. Introduction

Specializing an accelerator architecture for a particular deep neural network (DNN) to run on a particular platform has distinct power-performance advantages over the use of a generic design. However, manually optimizing an accelerator over the enormous design space spans from a large number of often changing DNN topologies, greatly varied platform capabilities, as well as sophisticated I/O and compute operation scheduling is not only time-consuming and error-prone, but the resulting performance is also difficult to predict early in the design process. Even small changes in the neural network model, such as altering the stride size, kernel size, or the number of channels used in a convolutional neural network (CNN) layer, can lead to significant performance degradation due to non-trivial hardware implementation issues such as mismatch in the number of available computing and on-chip memory resources, off-chip memory bandwidth, data layout or I/O schedule. As a result, development effort spent on optimizing one particular network on one particular target platform can hardly be reused when the network topology changes or when a different platform is targeted, which is very common in real-world industrial settings. To facilitate high productivity DNN accelerator development, it is therefore desirable to have automated design space exploration (DSE) tools that rapidly customize DNN accelerators for a target platform with good and reliable performance estimations.

In this paper, we introduce AGNA, an open-source, flexible, and extensible hardware generator for DNN acceleration that fills this gap. Given a DNN model and a generic specification of the compute and memory capabilities of the target platform, AGNA automatically produces a hardware accelerator optimized for the targeted model-platform combination. To achieve that, AGNA utilizes a highly customizable architecture template with an array of processing elements (PE) combined with a parametrizable memory hierarchy. Using this template, AGNA formulates the inference of each network layer as a multi-level loop tiling problem and uses a mixed integer geometric program (MIGP) to search for the optimal accelerator architecture based on a cross-layer analysis of the tiled loops under the performance and resource constraints of the target platform. At the hardware level, while the operations between the PE array execute asynchronously, the operations within the compute units of a PE are designed to execute with lock step in a single-instruction-multiple-data (SIMD) fashion to simplify hardware control logic. Finally, compute and I/O operations of each layer are scheduled to run on this architecture using a second MIGP to minimize overall execution latency, taking into account the hierarchical computing facilities, the amount of on-chip memory, as well as the performance of off-chip memory.

AGNA has been tested with a large combination of DNN models targeting a range of FPGA platforms with different compute and memory capabilities. Our results show that, on average, AGNA produced DNN accelerator designs and their corresponding schedules with $12.7\,\%$ overhead over the

*Equal contribution.

theoretical performance of the given platform, while achieving end-to-end inference performance comparable to the state-of-the-art. With that, the main contributions of AGNA are in the following aspects:

- We propose an automatic DNN inference accelerator generation framework that produces model-platform optimized hardware and software for FPGA systems;
- We propose two MIGP formulations and their relaxed solutions to achieve high-performance combined architectural search and operation scheduling in a unified framework;
- We contribute the design framework to the community as an open-source software to facilitate further enhancements[1].

In the next section, background and related work on optimizing DNN accelerators for FPGAs are described. The proposed design space exploration (DSE) methodology will be elaborated in Section III. Detailed implementation of the hardware template is discussed in Section IV. Experimental results are reported in Section V. We will conclude and layout future extensions to AGNA in Section VI.

II. Background & Related Work

A. DNN Accelerator on FPGA

A large body of works have repeatedly demonstrated the power and performance benefits when the design of an accelerator and the target neural network can be co-designed at the same time.

For instance, in the work of LUTNet [1], the K-LUT of FPGAs were used as building blocks to construct neural networks, which naturally lead to efficient hardware implementations. In [2], the authors proposed an ultra low-latency FPGA-based image analytical system using a quantized convolutional neural network that facilitate hardware pipeline operations. Other works such as [3], [4] also leveraged special hardware convolution operations like shift kernel or deformable kernel to achieve a better tradeoff between accuracy and hardware efficiency on FPGA. Furthermore, many works have explored using extreme low-bitwidth quantization to achieve better hardware performance [5], [6], [7].

However, given the enormous design space, it remains an unsolved challenge to optimize each model-platform combination to address the ever-changing real-world application requirements in speed and accuracy. As a result, in one line of work, techniques such as network architecture search (NAS) and hardware design space exploration (DSE) have been integrated together to achieve automatic algorithm-hardware codesign [8], [9], [10]. On the other hand, a number of works focus on the auto-generation of optimal hardware implementations for different networks and platforms. For instance, FP-DNN [11] designed an automated framework that maps the hardware implementation of a DNN from Tensorflow-description using predefined RTL-HLS templates of layers. [12] provided a framework that transforms C-program to CNN accelerator implementation in a systolic array architecture. SAMO [13] explored the automation of producing CNN accelerator targeting streaming architecture. In DNNBuilder [14], the authors proposed a fine-grained layer-based pipeline architecture with an automatic resource allocation algorithm to perform DSE in hardware implementation. Similar work also targeted Cloud FPGA [15] and ASICs [16]. Following this line of work, AGNA also aims to produce hardware accelerators that are customized for a given set of neural networks. It is unique in the way it searches for optimal architecture using MIGPs over a loop tiling formulation of a neural network and it shows that such formulation can be solved efficiently using relaxation-rounding algorithms.

B. Geometric Programming

Constrained optimization is widely used in computer architecture and system research. By properly formulating a design problem into objectives and constraints functions, one can solve large-scale problem reliably and efficiently. Prior works have adopted approaches like polyhedral transformation [17], [18], [19], mixed-integer programming [20] in accelerator design and operation scheduling. In particular, geometric programming owns the advantages of modeling loop tiling/unrolling in a very generic and simple way. A geometric program is a non-linear optimization problem characterized as follows:

$$\begin{aligned} \min \quad & f_0(x) \\ \text{s.t.} \quad & f_i(x) \leq 1, \quad i = 1, ..., m \\ & h_i(x) = 1, \quad i = 1, ..., p \end{aligned} \tag{1}$$

where f_0,..., f_m are polynomials and $h_1, ..., h_p$ are monomials [21]. A monimial is a function $cx_1^{a_1}x_2^{a_2}\cdots x_n^{a_n}$ with $c > 0$, and a polynomials is a sum of monimial, $\sum_{k=1}^{K} c_k x_1^{a_{1k}} x_2^{a_{2k}} \cdot \cdot \cdot x_n^{a_{nk}}$, where $c_k > 0$. It can be transformed into a convex form and solved efficiently with guaranteed convergence to a global optimal solution [22].

Previous works have used geometric programming to optimize design transformation in HLS [23], data-reuse [24], [25], [26], and tiling size selection [27], [26]. In this work, we leverage geometric programming to capture spatial and temporal tiling of nested loops in DNN hardware generator and operations scheduling in a uniform manner and use an efficient relaxation-rounding algorithm to solve the proposed programs.

III. DSE Methodology

AGNA takes DNN models and FPGA platform specifications as input and automatically produces high-performance hardware accelerators. The DSE flow of AGNA is shown in Fig. 1. The DSE in AGNA is decoupled into *Architecture Search* and *Operation Scheduling* to reduce the design space. Each decoupled problem is formulated as a MIGP. Without

[1]Latest source code can be found at https://github.com/CASR-HKU/AGNA-FCCM2023, source code for artifact evaluation can be found at https://doi.org/10.25442/hku.22181803.

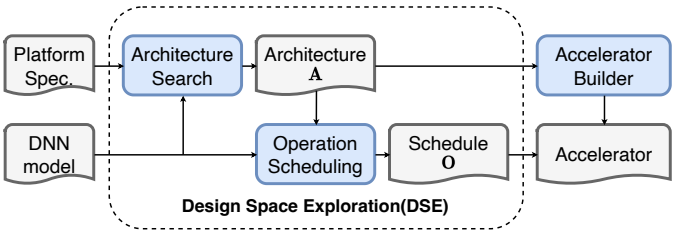


Fig. 1: Overview of AGNA design flow.

TABLE I: Notations of constants and variables in DSE.

Type	Notation	Description
Operation Constant	$\mathcal{D}$	The set of 6 enumerated loop dimensions: $\mathcal{D} = \{1, 2, 3, 4, 5, 6\}$.
	L_d	Loop bound of d-th loop of a layer.
	$\mathbf{L}$	The set of loop bounds in a layer: $\mathbf{L} = \{L_d \mid d \in \mathcal{D}\}$.
Platform Constant	N_{dsp}	Number of available DSP blocks.
	N_{mem}	Number of available memory blocks.
	α	DSP cost of one MACC operation.
	β	Memory cost of one buffer.
	$\gamma_{\text{rf}}, \gamma_{\text{buf}}$	Capacity of RF or buffer.
	w_{data}	Bitwidth of data.
	w_{bus}	Bitwidth of data bus on platform.
Design Variable	$\mathbf{A}$	The set of architecture parameters of PE. Design variable in *Architecture Search*.
	$\mathbf{O}$	The set of schedule parameters of a layer. Design variable in *Operation Scheduling*.

such decoupling, the design space can reach $\sim 10^3$ variables and $\sim 10^{1000}$ valid design points for ResNet-50.

In this section, we first introduce the abstraction methodology of AGNA in Section III-A. Then we formulate the MIGP of *Architecture Search* and *Operation Scheduling* in Section III-B and Section III-C, respectively. We also introduce a relaxation-rounding algorithm to solve MIGP in Section 1. Table I lists the notations of constants and variables used in DSE.

TABLE II: The columns under operation mapping show the semantics of L_d in different operations. 1 indicates the loop is skipped. The columns under memory mapping show the relation between L_d and the memory hierarchy. The checkmark indicates whether the memory is connected to the dimensions in the DSP array. Note that RF only exists between ABUF and DSP array.

	Operation mapping			Memory mapping			
	Conv	MatMul	FC	ABUF	WBUF	PBUF	RF
L_1	output channel	output row	output width		✓	✓	
L_2	input channel	input row	input width	✓	✓		✓
L_3	weight height	1	1		✓		✓
L_4	weight width	1	1		✓		✓
L_5	output height	1	1			✓	✓
L_6	output width	output column	1			✓	✓

```
for l1 in range(L_1):
  for l2 in range(L_2):
    for l3 in range(L_3):
      for l4 in range(L_4):
        for l5 in range(L_5):
          for l6 in range(L_6):
            output_act[l1,l5,l6] += weight[l1,l2,l3,l4] * \
            input_act[l2, l5*s+l3, l6*s+l4]
```

```
for o_t2 in range(O_T2):
  for o_s2 in range(O_S2):
    for o_p2 in range(O_P2):
      for o_q2 in range(O_Q2):
        for o_f2 in range(O_F2):
```

Fig. 2: The 6-dimension nested loop representation of a layer. s is the stride of input activation access. In *Operation Scheduling*, each L loop is further tiled into 5 sub-loops. The loops in the small green box show the example of tiling L_2 loop.

A. Abstraction Methodology

1) Operation Abstraction: The operation of a DNN layer can be represented in a 6-dimension nested loop as shown in Fig. 2. The loop bounds are denoted as $\mathbf{L}$. AGNA support convolution, depth-wise convolution, max pooling, average pooling, matrix-matrix multiplication, and fully connected layers by assigning $\mathbf{L}$ to different semantics. Detailed mapping of $\mathbf{L}$ is shown in Table II. Depth-wise convolution, max pooling, and average pooling layers are omitted since they have the same semantics as convolution layers and are only different in operations.

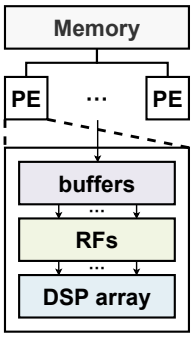
2) Architecture Abstraction: The target architecture of the accelerator in AGNA is built by an array of processing elements (PE) as shown in Fig. 3. PEs communicate with external memory through a multicast network. All PEs share the same control logic and operate in single instruction multiple data (SIMD) fashion. Each PE has one DSP array for computation and two levels of memory for storage.

The DSP array has 6 dimensions of DSPs, corresponding to the 6-dimension loop in $\mathbf{L}$. We denote the architecture of the DSP array as a vector $\mathbf{A} = \{A_d \mid d \in \mathcal{D}\}$. A_d is the number of parallel operations on dimension d. The total number of DSPs required in one PE is:

$$N_{\text{dsp}} = \alpha \prod_{d \in \mathcal{D}} A_d \tag{2}$$

Where α is the cost of DSPs per operation. For example, $\alpha = 0.5$ if two MACC operations are packed into one DSP block.

The memory in PE has two levels: buffer and register file (RF). We build ABUF, WBUF, and PBUF at the buffer level to store input activation, weight, and partial sum. Buffers directly communicate with external memory. Each buffer is connected to multiple RFs. RFs are smaller than buffers and only store a small copy of data from the upper-level buffer. We build RFs for ABUF due to the irregular access of input activation shown in Fig. 2. The access to the ABUF varies dynamically from stride and kernel size. With RF, the accelerator can flexibly support different data access and schedules while maintaining high computation efficiency and reducing memory footprint. RFs for WBUF and PBUF are eliminated since the access is static.




```
init_memory()
for vec_t in range(T):
  for vec_s in range(S):
    update_buffer()
    for vec_p in range(P):
      update_rf()
      for vec_q in range(Q):
        for vec_f in range(F):
            dsp_opertion()
```

Fig. 3: Abstract accelerator and pseudo-code of mapping from operation to accelerator. The background color of the pseudo-code corresponds to the scope of the hardware.

The required number of buffers or RFs in one PE can be computed by the product of its related dimensions. The related dimensions are shown in Table II. For example, the number of PBUF is given by $N_{\text{pbuf}} = A_1 A_5 A_6$. Similarly, we define N_{abuf}, N_{wbuf}, and N_{rf}. The total number of memory blocks required in one PE is:

$$N_{\text{mem}} = \beta \left(N_{\text{abuf}} + N_{\text{wbuf}} + N_{\text{pbuf}} \right) \tag{3}$$

Where β is the cost of memory blocks per buffer. As a vendor-agnostic tool, AGNA does not limit the memory type. For example, the memory blocks could be RAM18K in Xilinx devices or M20K in Intel devices. RF is built by LUTs thus is excluded from the above formulation.

Variable $\mathbf{A}$ is used to denote the architecture of PE. The total number of PEs built on the accelerator is denoted as N_{pe}.

3) Scheduling Methodology: There are 5 levels in the accelerator to be scheduled: T, S, P, Q, and F, as shown in Fig. 3. A loop is tiled to 5 levels and mapped to the levels in the accelerator. T is mapped to PE in the time domain where the loop body inside T is executed sequentially. S is mapped to PE in the space domain where the loop body is assigned to different PEs and executed in parallel. We refer to the loop body of S as a tile, the execution unit in PE. Inside the PE, a tile is further mapped to P, Q, and F. P and Q are assigned to the memory hierarchy of buffers and RFs respectively. All DSP operations in F are mapped to the DSP array and executed in parallel. Each level has 6 schedule parameters which are the tiling factors. A 5×6 matrix $\mathbf{O}$ is used to denote the schedule parameters of a layer. We denote the vector of schedule parameters at level l as $\mathbf{O}_l$, where $l \in \{T, S, P, Q, F\}$. A single schedule parameter at level l of d-th loop in $\mathbf{O}$ is denoted as $O_{l,d}$. e.g., $O_{T,4}$ denotes the schedule parameter at level T of 4-th loop.

B. Architecture Search

In this section, we discuss the formulation of architecture search. By modeling the resource utilization of the accelerator and architectural latency of each layer, a cross-layer MIGP is formulated. The architecture $\mathbf{A}$ will be produced in this stage.

1) Resource Utilization: The accelerator has a total number of N_{pe} PEs. PE utilizes most hardware resources including DSPs and memory blocks in the accelerator. With the resource utilization of single PE that is formulated in (2) and (3), the total number of DSPs used by the accelerator is formulated as $N_{\text{pe}} N_{\text{dsp}}$, and the total number of memory blocks used by the accelerator is formulated as $N_{\text{pe}} N_{\text{mem}}$.

2) Latency Model: We formulate the latency of i-th layer as the max of computation latency and communication latency:

$$\begin{aligned} comp^i_{\text{arch}} &= \frac{\prod_{d \in \mathcal{D}} \lceil L^i_d / A_d \rceil}{N_{\text{pe}}} \\ comm^i_{\text{arch}} &= \frac{\max\left(size^i_{\text{rd}}, size^i_{\text{wr}}\right) w_{\text{data}}}{w_{\text{bus}}} \\ lat^i_{\text{arch}} &= \max\left(comp^i_{\text{arch}}, comm^i_{\text{arch}}\right) \end{aligned} \tag{4}$$

$size^i_{\text{rd}}$ computes required size of input activation and weight in layer i. $size^i_{\text{wr}}$ computes required size of output activation in layer i. They are all formulated as posynomial of variable $\mathbf{A}$, e.g., $size^i_{\text{wr}} = L^i_1 L^i_5 L^i_6$. w_{data} is the bit width of data. w_{bus} is the bit width of the data bus between the accelerator and external memory.

3) Optimization Program of Architecture Search: With the resource utilization and latency model above, the architecture search problem for an m layers DNN model is formulated as:

$$\begin{aligned} \min_{\mathbf{A}} \quad & \sum_{i=1}^{m} lat^i_{\text{arch}} \\ \text{s.t.} \quad & N_{\text{pe}} N_{\text{dsp}} \le B_{\text{dsp}} \\ & N_{\text{pe}} N_{\text{mem}} \le B_{\text{mem}} \end{aligned} \tag{5}$$

The objective function is the sum of the architectural latency of each layer. Note that this formulation can be extended to multi-model optimization by formulating the geometric mean of architectural latency of multiple models. The extended objective function is still in the form of posynomial and thus the extended optimization program is still a geometric program.

C. Operation Scheduling

In this section, we discuss the formulation of operation scheduling. $\mathbf{A}$ becomes constant after the architecture search stage. The schedule of each layer is formulated as a MIGP individually. The operation schedule $\mathbf{O}$ will be produced by solving this MIGP.

1) Loop Bound Constraints: To keep the effectiveness of computation, the schedule should cover all operations in the original loop $\mathbf{L}$. Therefore, the product of schedule parameters of all levels should be larger than the loop bound on the corresponding dimension. For each $d \in \mathcal{D}$:

$$L_d \le O_{T,d} O_{S,d} O_{P,d} O_{Q,d} O_{F,d} \tag{6}$$

2) Spatial Constraints: A valid schedule should ensure that each operation is mapped to a valid PE and DSP. In the target architecture, S and F levels are mapped to the accelerator in the space domain. S level is mapped to different PEs. PEs are connected via a multicast network and can be mapped to arbitrary dimensions at runtime. The schedule is valid when the product of $\mathbf{O}_S$ is no more than the total number of PEs. Level F is mapped to parallel DSPs. The connection of DSPs is hard-wired. Hence each scheduling parameter in level F should not exceed PE architecture on the corresponding dimension. The spatial constraints are:

$$\begin{aligned} &\prod_{d\in\mathcal{D}} O_{S,d} \le N_{\text{pe}} \\ &O_{F,d} \le A_d \quad \forall d \in \mathcal{D} \end{aligned} \tag{7}$$

3) Memory Utilization: Referring to the pseudo-code in Fig. 3, memory should store all required data for the inner loop. RFs are at the level Q and should start data for Q and F. Buffers are at the level P and should store data for P, Q, and F. The utilization of RFs and buffers can be formulated as:

$$\begin{aligned} U_{\text{rf}} &= \prod_{d\in\{2,5,6\}} O_{Q,d}O_{F,d} \\ U_{\text{abuf}} &= \prod_{d\in\{2,5,6\}} O_{P,d}O_{Q,d}O_{F,d} \\ U_{\text{wbuf}} &= \prod_{d\in\{1,2,3,4\}} O_{P,d}O_{Q,d}O_{F,d} \\ U_{\text{pbuf}} &= \prod_{d\in\{1,5,6\}} O_{P,d}O_{Q,d}O_{F,d} \end{aligned} \tag{8}$$

Note that U_{rf} and U_{abuf} listed here are in the case of $stride = 1$ for simplicity. Other stride sizes are also supported.

4) Computation Cycle: PE executes the tiles sequentially. Inside the execution of one tile, the PE iterates over level P and Q sequentially in the time domain. Level F is mapped to parallel DSPs. DSPs start to compute after the data in RFs are ready. We use double buffering on RFs to hide the latency of updating RFs. The computation latency of one layer is formulated as:

$$comp_{\text{schd}} = \max\left(U_{\text{rf}}, \prod_{d\in\mathcal{D}} O_{Q,d}\right) \prod_{d\in\mathcal{D}} O_{P,d}O_{T,d} \tag{9}$$

The result of $\max$ is the execution cycle of one P iteration. The first term in $\max$ is the update cycle of RFs. Here we update one data each cycle. The second term in $\max$ is the execution cycle of the DSP array, which is the product of $\mathbf{O}_Q$. Therefore, the execution cycle of one layer can be formulated as the product of one P iteration cycle and the total number of iterations which is the product of $\mathbf{O}_P$ and $\mathbf{O}_T$.

5) Communication Cycle: The communication cycle is determined by both the update cycle of buffers and the reuse pattern of buffers. We first formulate the update cycle of buffers. For ABUF:

$$updt_{\text{abuf}} = \frac{U_{\text{abuf}} w_{\text{data}}}{w_{\text{bus}}} \tag{10}$$

Algorithm 1 Relaxation and Rounding

Input: A m layers network with loop bound $\mathbf{L}^1, ..., \mathbf{L}^m$, rounding range R.
Output: Architecture $\mathbf{A}$.
1: Find the real-value solution $\widetilde{\mathbf{A}}$ for the relaxed program in (5) using standard geometric program solver
2: Add bounds to each variable: $\left\lfloor \widetilde{A}_d \times R^{-1} \right\rfloor \le A_d \le \left\lceil \widetilde{A}_d \times R \right\rceil, \forall d \in \mathcal{D}$
3: Add integer constraint to each variable: $A_d \in \mathbb{N}^+, \forall d \in \mathcal{D}$
4: Solve the new integer program using MIGP solver

Similarly, $updt_{\text{wbuf}}$ and $updt_{\text{pbuf}}$ can be formulated by substituting U_{abuf} in Eqn. 10 with U_{wbuf} and U_{pbuf}, respectively.

PEs are assigned with different tiles of computation. However, different tiles of computation may use the same tile of data. For example, a tile of output activation is accumulated by the partial sum from multiple different tiles. These tiles of partial sum correspond to different tiles of computation. Such kind of reuse can be categorized into temporal reuse and spatial reuse.

Temporal reuse happens when the tile of data in the buffer can be reused in the next tile of computation. The loop order in level T determines the temporal reuse pattern. The loop is ordered as $(L_1, L_5, L_6, L_2, L_3, L_4)$ from outermost to innermost for an output-stationary dataflow. In this case, data in PBUF is fully reused, while ABUF and WBUF need to be updated before each tile of computation.

Spatial reuse is conducted by the multicast network that connects all PEs and external memory. One data tile can be transferred to multiple PEs simultaneously through the multicast network.

The communication cycle of one layer under output-stationary dataflow is:

$$\begin{aligned} comm_{\text{schd}} =& updt_{\text{pbuf}} \prod_{d\in\{1,5,6\}} O_{T,d}O_{S,d} \\ &+ updt_{\text{wbuf}} \prod_{d\in\mathcal{D}} O_{T,d} \prod_{d\in\{1,2,3,4\}} O_{S,d} \\ &+ updt_{\text{abuf}} \prod_{d\in\mathcal{D}} O_{T,d} \prod_{d\in\{2,5,6\}} O_{S,d} \end{aligned} \tag{11}$$

6) Optimization Program of Operation Scheduling: With the analysis and modeling above, the operation scheduling problem is formulated as:

$$\begin{aligned} \min_{\mathbf{O}} \quad & \max\left(comp_{\text{schd}}, comm_{\text{schd}}\right) \\ \text{s.t.} \quad & U_{\text{rf}} \le \gamma_{\text{rf}} N_{\text{rf}} \\ & U_{\text{abuf}} \le \gamma_{\text{buf}} N_{\text{abuf}} \\ & U_{\text{wbuf}} \le \gamma_{\text{buf}} N_{\text{wbuf}} \\ & U_{\text{pbuf}} \le \gamma_{\text{buf}} N_{\text{pbuf}} \end{aligned} \tag{12}$$

Note that the constraints in (6) and (7) are also part of the operation scheduling problem.

D. Solving Algorithm

In the previous discussion, we formulate the architecture search problem and operation scheduling problem as two optimization programs (5) and (12). As mentioned before, directly solving the program in the enormous integer space is intractable. In AGNA, we use a relaxation-rounding algorithm to search for the solution efficiently. The detailed explanations are as follows.

1) Transformation: To cast our programs into standard GP formulation, we need to transform unsupported operations like $\max$ and ceiling. $\max$ can be eliminated by setting an intermediate variable and substituting the $\max$ operation in the original formula with the intermediate variable. The intermediate variable should be larger than any variables inside $\max$. The ceiling operations in (4) are also eliminated by setting an intermediate variable. The ceiling operations will be automatically recovered in integer scope. After that, all the previously discussed constraints and objectives can be reorganized as polynomial inequalities.

2) Relaxation: The proposed programs will then be relaxed and reorganized as standard GP in (1) to allow the GP solver to produce a solution in one shot. All the original integer variables are allowed to be positive real values, i.e. $x > 0, \forall x \in \mathbf{A}, \mathbf{O}$.

3) Rounding: After we obtain the relaxed solution, it will be rounded into integers. Since the simple rounding results can neither be guaranteed to be feasible nor optimal, we search for optimal results within a rounding range. In other words, for each variable in the relaxed solution, a certain number of the nearest integers will be selected to be the candidate for the final solution. This is done by setting integer upper and lower bounds on variables. In AGNA, we use a geometrical range factor R that multiplies or divides the original relaxed solution to get the bounds. By adding these additional constraints of the rounding range, the final program will be fed into an integer programming solver. Choosing a larger range factor leads to a larger search space, which takes longer to solve. This can be a tradeoff between efficiencies and qualities of the solution.

Alg.1 shows an example of using the relaxation-rounding algorithm to solve the architecture search problem in (5).

IV. Hardware Template

Given the architecture produced by the DSE framework, AGNA customize a hardware template to produce the accelerator. This section introduces the detailed implementation of the hardware template and the optimizations of the template.

A. Overall Architecture

Fig. 4 shows the overall architecture of the template. Besides the PE array as mentioned in the III-A, the template also consists of a top controller (AGNA controller) for reading instructions from external memory and redistributing to other modules, and some special computation modules for batch normalization, quantization, residual add, and activation. The instructions are stored in the external memory and transferred to the accelerator through the memory interface. To improve

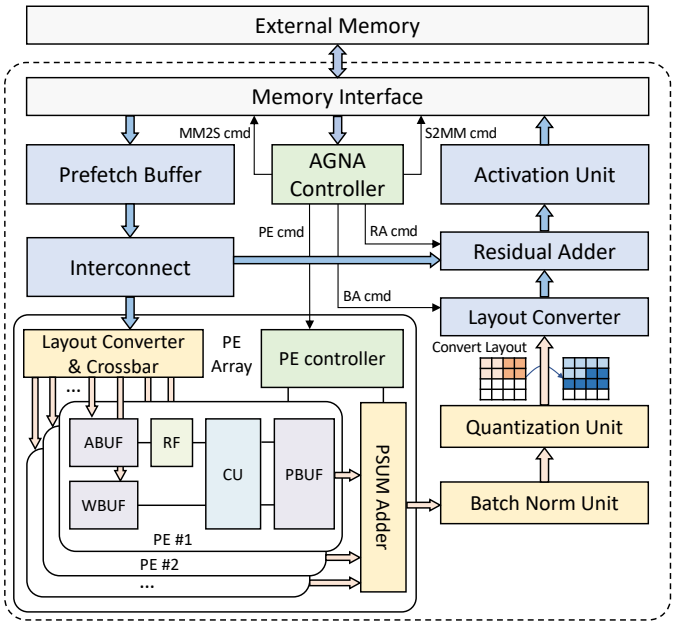


Fig. 4: The architecture of proposed hardware template.

the execution efficiency, the instructions only include layer parameters $\mathbf{L}$ and schedule parameters $\mathbf{O}$. The AGNA controller fetches the instructions and generates commands to the different modules for control. When proceeding to the next layer, the controller will fetch another batch of instructions and regenerate new commands. Besides, we set up a prefetch buffer to hide the memory latency from the external memory to the acclerator. The interconnect module is responsible for dispatching input activations and weights to PEs or bypassing to residual adder. After the computations are finished by PE array, PSUM adders will add up the partial sums. The output results will go through all the special modules to obtain output activation data. Finally, the interface will write the valid results back to the external memory.

B. Processing Element (PE)

As mentioned in the III-A, each PE consists of buffers, RFs, and a DSP array. All the PEs share the same PE controller

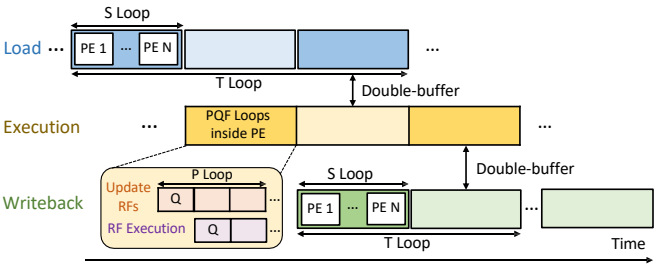


Fig. 5: PE array processing diagram for T, S, P, Q, and F loops. The hardware can achieve a high-throughput pipeline by using double buffering.

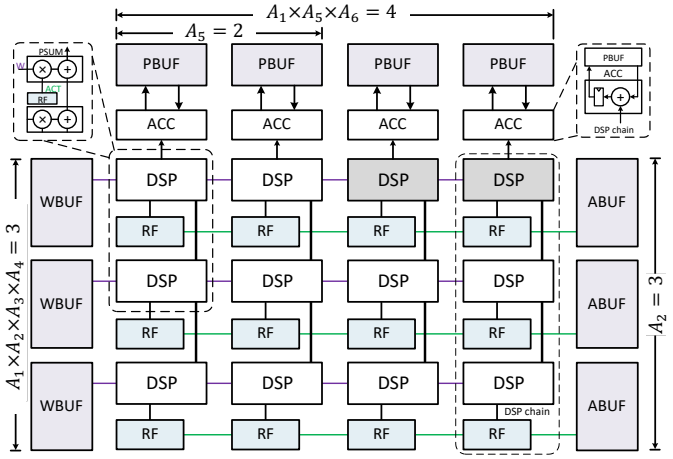


Fig. 6: An example PE architecture with $\mathbf{A} = \{1, 3, 1, 1, 2, 2\}$

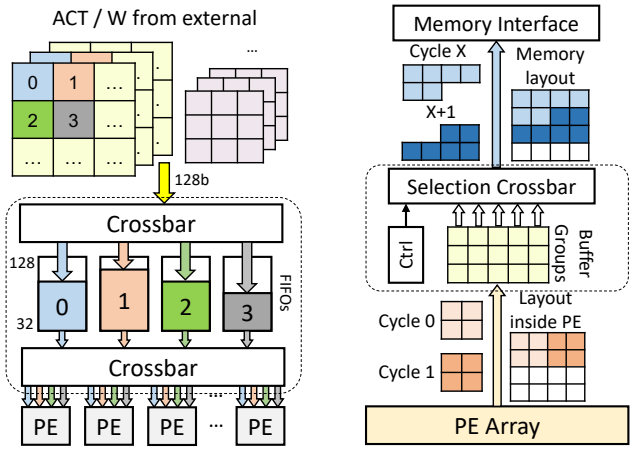


Fig. 7: An example of layout converter on the read (left) and the writeback (right) sides.

as a SIMD-like execution to reduce the hardware overhead introduced by the control logic.

1) PE controller: The execution of PE array is controlled by the PE controller. Fig. 5 shows how the PE array execute the T, S, P, Q, and F loops. Blue blocks refer to the load stage in which the PE array fetches input activation and weight from the prefetch buffer, and yellow blocks belong to the PE execution stage where all the PEs execute P, Q, and F loops simultaneously. The writeback stage is shown in green blocks, indicating that the PE array is writing data back to the external memory. As illustrated in Section III, T and S loops are mapped to the PE array temporally and spatially. Before starting each layer, the PEs are assigned to an S index for the S loop in the load and writeback stages, and they share the same execution controller to process P (update RFs from ABUF), Q (RF execution loop), and F (parallel DSP operations) loops inside PE. We use double buffering on all the buffers and RFs to overlap the load, writeback and execution stages.

2) DSP array: Fig. 6 shows the PE design for based on an example of architecture $\mathbf{A} = \{1, 3, 1, 1, 2, 2\}$. The RFs with the same A_2 (i.e. same row in Fig. 6) are connected to the same ABUF as input. Similarly, WBUFs are shared in the A_5 and A_6 dimensions, while PBUFs are shared in the A_2, A_3, and A_4 dimensions. Because all the computations in A_2, A_3, and A_4 dimensions will be accumulated to the same partial sums, the DSPs are separated into different DSP chains to improve the timing performance. If the DSP slice supports at least 24-bit input MACC mode (e.g. DSP48E2 in Xilinx UltraScale FPGA), we can take advantage by combining two INT8 MACC while sharing the same kernel weights [28]. For instance, the two adjacent DSPs with the light gray color in Fig. 6 can be replaced with one, and the total number of DSPs is reduced to half.

C. Layout Converter

As a hardware template, it should support different architecture and data bus width combinations. At the load stage, the data bus width may not match the buffer write width. At writeback stage, since the output activation will be used as input activation in next layer, the layout of the writeback data should be the same as the input activation. The mismatched layout would cause low burst length when the data are loaded in next layer which will result in low bandwidth utilization.

To address the bandwidth and layout mismatch problem, we design a layout converter to eliminate the mismatch. Fig. 7 presents an example of the proposed converter in the case of 128-bit AXI bus and BRAMs with a 32-bit bandwidth. Combining the BRAMs to write 128 bits in parallel is simple but not flexible because the architecture varies for different DNN models in our general framework. Alternatively, we use four 128-in-32-out FIFOs to match the bandwidth. If all these FIFOs containing different tiles are not empty most of the time, the output side (to PE array) can approximately achieve the 128-bit bandwidth. For the writeback side, we should keep all the activations in memory with the same data layout and also match the bandwidth between the memory interface and PE array (6 versus 4 in the example of Fig. 7). Therefore, another layout converter is applied to match the bandwidth difference between PBUF fetching and memory writing and convert the data layout of output activation.

V. Evaluation

A. Experiment Setup

a) Target Models: The DNN models we test are: AlexNet [29], MobileNet-v2 [30], ResNet-50 [31], VGG16 [32], and YOLO-v2 [33]. To differentiate the input size of the same model, we add the input size as postfix to the model. e.g., AlexNet_227 is the AlexNet with input size of 227×227. We target 8-bit and 16-bit fixed-point data widths for all models. All results are obtained with batch size set to 1.

b) Target Platforms: We target three hardware platforms from embedded devices to cloud accelerator cards: Ultra96, ZCU102, and U200. Both Ultra96 and ZCU102 are Zynq devices, here we set the target data bus width of them as

TABLE III: Schedule performance on different model-platform combinations.

Model	Data width	Ultra96			ZCU102			U200		
		Theoretical Latency†	Schedule Latency†	Norm. Perf.‡	Theoretical Latency†	Schedule Latency†	Norm. Perf.‡	Theoretical Latency†	Schedule Latency†	Norm. Perf.‡
AlexNet_227	8	5204	6059	1.164	4043	4154	1.027	1028	1103	1.073
	16	10407	11559	1.111	8085	8315	1.028	2056	2234	1.087
MobileNet-v2_192	8	859	979	1.140	852	882	1.035	213	222	1.042
	16	1717	1851	1.078	1704	1778	1.043	426	599	1.406
MobileNet-v2_224	8	1096	1305	1.191	1080	1158	1.072	270	288	1.067
	16	2191	2529	1.154	2159	2568	1.189	540	573	1.061
ResNet-50_224	8	5762	7743	1.344	3315	3353	1.011	855	910	1.064
	16	11524	14361	1.246	6630	6942	1.047	1709	1736	1.016
VGG16_224	8	29615	36115	1.219	11691	12318	1.054	3405	3956	1.162
	16	59230	78433	1.324	23381	24646	1.054	6810	7365	1.081
YOLO-v2_448	8	24909	34963	1.404	6435	6576	1.022	1854	2200	1.187
	16	49818	77016	1.546	12870	13161	1.023	3708	3877	1.046

† The unit of latency is 10^3 cycle. ‡ Norm. Perf. refers to schedule latency that is normalized by theoretical latency. The lower the better.

128-bit, which is the maximum AXI port width from PL to DDR on Zynq devices. U200 has a direct connection to on-board DDR4 memory. Here we target 512-bit data bus width on U200, which is the maximum AXI port width of one Xilinx Memory IP. We pack two MACC operations in one DSP for 8-bit data width to further improve throughput.

c) GP Solver: As discussed in Section III-D, we solve the optimization programs in two stages. The relaxed program in the first stage is solved by GPKit [34]. The rounded integer program in the second stage is solved by SCIPOpt [35].

B. Search Time

We compare the search time of proposed relaxation-rounding algorithm with other methods in Table IV. We evaluate the operation scheduling of conv1 in ResNet-50 with 3 different approaches: Method 1 is the proposed relaxation-rounding algorithm which searches in rounded space with MIGP solver. Method 2 searches directly in original space with MIGP solver. Method 3 searches in rounded space exhaustively by iterating all valid designs in the search space. The rounding range R is fixed to 3 for all variables in this section. The effect of R will be discussed in detail later.

By comparing method 2 to method 1, we observe that the proposed rounding scheme significantly decreases the search space. With the rounded search space, search time is saved by $500\times$. Comparing method 3 to method 1, the rounded space is too large for exhaustive search to get the solution. We run the exhaustive search with 32 threads and each thread can evaluate $\sim$ 4000 designs per second. The estimated search time still reaches thousands of days for just one layer. On the other hand, the relaxation-rounding algorithm helps the solver to search more efficiently.

TABLE IV: Comparison of search time with other methods on the same operation scheduling task. The number after the search space gives the total number of valid designs of the search space. Relaxation-rounding algorithm can significantly reduce the search space. The MIGP formulation can help the solver to search effectively compare to exhaustive search.

#	Search space	Search method	Time (sec)
1	Rounded (7.36×10^{12})	MIGP	**0.33**
2	Original (2.28×10^{40})	MIGP	164.69
3	Rounded (7.36×10^{12})	Exhaustive search	5.75×10^7

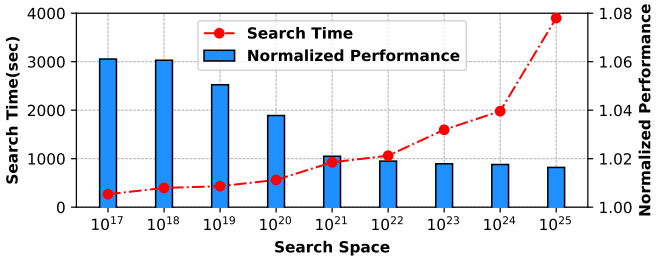


Fig. 8: Effect of search space. Normalized performance is the execution latency of produced schedule that is normalized by the theoretical minimal latency on target platform. Both search time and normalized performance are the lower the better.

C. Search Space

The MIGP problem is solved efficiently with the relaxation-rounding algorithm. However, the adjustable search space affects not only the search time but also the quality of produced schedules. To show the effect of search space on search time and produced schedules. We evaluate the operation scheduling of all layers in ResNet-50 with different search spaces. The search space is adjusted by the rounding range R that is mentioned in Alg.1. The results are shown in Fig. 8.

As the search space increases, more feasible schedules will be evaluated and thus may produce better schedules. However, the search time also increases. As search space increases $10^8\times$ from the smallest one to the largest one in the figure, the search time increases $14.6\times$. However, it only improves the performance of the whole model by 4%. By breaking down the performance of each layer, we observe that some layers already have good enough performance even when the search space is small. Only a few layers get performance improvement from the increased search space. Manually adjusting search space of each layer for infinite model-platform combinations is unpractical. Instead, we monitor the progress of the solver and adjust the search space adaptively.

TABLE V: Hardware performance of AGNA and comparison with other DNN Accelerators.

	Model	Bus width	Data width	Platform	Frequency (MHz)	DSP utilization	BRAM utilization	Performance
DNNBuilder [14]	AlexNet	512‡	16	ZC706	200	808/900	303/545	170.0 fps
	VGG16	-‡	16	KU115	235	4318/5520	1578/2160	65.0 fps
DNNExplorer [36]	VGG16-conv†	-‡	16	KU115	200	4444/5520	1648/2160	55.4 fps
FlexCNN [37]	VGG16-conv†	512‡	16	U250	241	37.98%§	45.93%§	19.89 ms
	VGG16-conv†	512‡	8	U250	198	8.7%§	58.43%§	13.18 ms
Vitis AI [38]	MobileNet-v2	2×128	8	Ultra96	287	326/360	126/216	10.17 ms
	ResNet50	2×128	8	Ultra96	287	326/360	126/216	30.80 ms
	MobileNet-v2	2×128	8	ZCU102	281	2130/2520	765/912	3.73 ms
	ResNet50	2×128	8	ZCU102	281	2130/2520	765/912	11.45 ms
	VGG16	2×128	8	ZCU102	281	2130/2520	765/912	49.63 ms
AGNA (ours)	MobileNet-v2¶	128	8	Ultra96	214	354/360	153/216	8.42 ms
	ResNet50	128	8	Ultra96	214	358/360	153/216	68.97 ms
	VGG16	128	8	Ultra96	214	354/360	136/216	176.91 ms
	MobileNet-v2¶	128	8	ZCU102	214	2365/2520	632/912	6.98 ms
	ResNet50	128	8	ZCU102	214	2429/2520	509/912	21.71 ms
	VGG16	128	8	ZCU102	214	2303/2520	737/912	79.04 ms

† Fully connected layers are not implemented. ‡ Referenced from their source code. § Only percentage is reported.
¶ MobileNet-v2_224 is used for fair comparison.

D. Schedule Performance

Table III lists the schedule performance of all target model-platform combinations. The theoretical latency is measured by $\max(comm_{\text{theo}}, comp_{\text{theo}})$. $comm_{\text{theo}}$ is the minimal required cycle to finish all data transfer between accelerator and external memory. $comm_{\text{theo}}$ is bounded by the data bus width. $comp_{\text{theo}}$ is the minial required cycle for accelerator to finish all computations. $comp_{\text{theo}}$ is bounded by the total number of DSP on target platform. The schedule latency is measured based on the produced schedule $\mathbf{O}$ on the target architecture $\mathbf{A}$. Theoretical latency indicates the theoretical minimal latency that the platform can achieve. By normalizing schedule latency with theoretical latency, we can evaluate how close the produced accelerator is to the theoretical model. This value also indicates the DSP utilization and bandwidth efficiency of produced accelerators. Given that the average normalized performance among all combinations is 1.127, the proposed DSE framework is effective in a wide range of model-platform combinations.

Though most of the normalized performance are close to 1, we still observe that Ultra96 underperforms in general compared to other platforms. This is caused by the limited on-chip memory resources in Ultra96. In this case, loops are tiled into smaller chunks to ensure that data can fit into buffer capacity. When loop tiles are small, data is less reused, which needs more data exchange and results in a more memory-bound schedule. Thus the performance on Ultra96 is even worse in large models like YOLO-v2.

E. Hardware Evaluation

By customizing the accelerator template with parameters solved by the proposed DSE, AGNA can produce accelerator that is optimized for target model-platform combination. We evaluate the end-to-end latency of produced accelerators and show the performance in Table V. The produced accelerators achieve 96.3% DSP utilization and 68.4% BRAM utilization on average without timing violation.

We compare our work with other FPGA DNN accelerators in Table V. DNNBuilder and DNNExplorer design dedicated modules for part of the layers and a generic module for the remaining layers. They achieve high throughput by optimizing the bandwidth allocation of pipelined modules. Such layer pipeline design is not suitable for very deep networks. On the other hand, our DSE framework has no limitation on the depth of the target network. FlexCNN proposes a versatile systolic array and supports various convolution operations. They use exhaustive search to determine the parameters of systolic array and greedy algorithm to schedule each layer. Exhaustive search is realizable in FlexCNN since the design space of systolic array has only 3 variables. In terms of resource utilization, FlexCNN underutilizes the on-chip DSP, especially in the case of 8-bit data width where the computations are implemented on LUT by HLS. Instead, our framework is able to produce accelerators with high DSP utilization without compromising frequency. We also compare our work with Vitis AI, a Xilinx AI accelerator library for non-commercial usage only. Vitis AI provides DPU as the core component for acceleration. DPU is highly optimized for Xilinx devices and has the highest frequency among other works. DPU also uses two data buses for loading data resulting in $2\times$ bandwidth. We already outperform DPU on MobileNet-v2 on Ultra96, even without such optimizations in our current hardware template. With the optimization of hardware template in the future, AGNA will produce comparable performance.

VI. Conclusion

In this paper, we have presented AGNA, an open-source deep neural network (DNN) accelerator hardware and software generator for FPGA platforms. AGNA relies on the two proposed MIGP formulations and their relaxed solutions to perform DSE in customizing a generic accelerator template for the given network model-platform combination. Through extensive experiments using many combinations of DNN models and platforms, we have demonstrated AGNA's capability

to produce DNN accelerators with performance comparable to the state-of-the-art in research and vendor-provided frameworks. Importantly, although the accelerators produced currently may not be the fastest in all model-platform combinations, AGNA is vendor-agnostic and is designed to be easily extensible, making it suitable for real-world deployments and serving as a basis for future research. In the future, we plan to improve AGNA with advanced scheduling capability to work with multi-bank memories, as well as to improve its performance through low-level hardware optimizations. We also plan to explore novel network model quantization and pruning techniques by leveraging the processing architecture and scheduling capabilities of AGNA.

ACKNOWLEDGMENT

This work was supported in part by the Research Grants Council (RGC) of Hong Kong under the Research Impact Fund project R7003-21. This work was also supported by AI Chip Center for Emerging Smart Systems (ACCESS), sponsored by InnoHK funding, Hong Kong SAR.

REFERENCES

[1] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Rethinking Inference in FPGA Soft Logic," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 26–34.

[2] M. Wang, K. C. M. Lee, B. M. F. Chung, S. V. Bogaraju, H.-C. Ng, J. S. J. Wong, H. C. Shum, K. K. Tsia, and H. K.-H. So, "Low-Latency In Situ Image Analytics With FPGA-Based Quantized Convolutional Neural Network," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 7, pp. 2853–2866, 2022.

[3] Q. Huang, D. Wang, Z. Dong, Y. Gao, Y. Cai, T. Li, B. Wu, K. Keutzer, and J. Wawrzynek, "CoDeNet: Efficient Deployment of Input-Adaptive Object Detection on Embedded FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 206–216. [Online]. Available: https://doi.org/10.1145/3431920.3439295

[4] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek, and K. Keutzer, "Synetgy: Algorithm-Hardware Co-Design for ConvNet Accelerators on Embedded FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 23–32. [Online]. Available: https://doi.org/10.1145/3289602.3293902

[5] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231217315655

[6] Y. Zhang, J. Pan, X. Liu, H. Chen, D. Chen, and Z. Zhang, *FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations*. New York, NY, USA: Association for Computing Machinery, 2021, p. 171–182. [Online]. Available: https://doi.org/10.1145/3431920.3439296

[7] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 15–24. [Online]. Available: https://doi.org/10.1145/3020078.3021741

[8] Z. Dong, Y. Gao, Q. Huang, J. Wawrzynek, H. K. So, and K. Keutzer, "HAO: Hardware-aware Neural Architecture Optimization for Efficient Inference," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 50–59.

[9] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317829

[10] M. S. Abdelfattah, L. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane, "Best of Both Worlds: AutoML Codesign of a CNN and Its Hardware Accelerator," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020.

[11] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 152–159.

[12] 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 *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3061639.3062207

[13] A. Montgomerie-Corcoran, Z. Yu, and C.-S. Bouganis, "SAMO: Optimised Mapping of Convolutional Neural Networks to Streaming Architectures," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 418–424.

[14] 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 *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2018, pp. 1–8.

[15] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 73–82. [Online]. Available: https://doi.org/10.1145/3289602.3293915

[16] P. Xu, X. Zhang, C. Hao, Y. Zhao, Y. Zhang, Y. Wang, C. Li, Z. Guan, D. Chen, and Y. Lin, "AutoDNNchip: An Automated DNN Chip Predictor and Builder for Both FPGAs and ASICs," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 40–50. [Online]. Available: https://doi.org/10.1145/3373087.3375306

[17] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving High Level Synthesis Optimization Opportunity through Polyhedral Transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 9–18. [Online]. Available: https://doi.org/10.1145/2435264.2435271

[18] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop Transformations: Convexity, Pruning and Optimization," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 549–562. [Online]. Available: https://doi.org/10.1145/1926385.1926449

[19] J. Wang, L. Guo, and J. Cong, *AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA*. New York, NY, USA: Association for Computing Machinery, 2021, p. 93–104. [Online]. Available: https://doi.org/10.1145/3431920.3439292

[20] Q. Huang, A. Kalaiah, M. Kang, J. Demmel, G. Dinh, J. Wawrzynek, T. Norell, and Y. S. Shao, "CoSA: Scheduling by Constrained Optimization for Spatial Accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 554–566.

[21] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, "A tutorial on geometric programming," *Optimization and engineering*, vol. 8, no. 1, pp. 67–127, 2007.

[22] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[23] Q. Liu, T. Todman, W. Luk, and G. A. Constantinides, "Optimizing Hardware Design by Composing Utility-Directed Transformations," *IEEE Transactions on Computers*, vol. 61, no. 12, pp. 1800–1812, 2012.

[24] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Combining Data Reuse With Data-Level Parallelization for FPGA-Targeted Hardware Compilation: A Geometric Programming Framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 3, pp. 305–315, 2009.

[25] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. Cheung, "Compiling C-like Languages to FPGA Hardware: Some Novel Approaches Targeting Data Memory Organization," *The Computer Journal*, vol. 54, no. 1, pp. 1–10, 2011.

[26] J. Liu, J. Wickerson, and G. A. Constantinides, "Tile size selection for optimized memory reuse in high-level synthesis," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

[27] J. Shan, M. R. Casu, J. Cortadella, L. Lavagno, and M. T. Lazarescu, "Exact and Heuristic Allocation of MuIti-kernel Applications to Multi-FPGA Platforms," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[28] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep Learning with INT8 Optimization on Xilinx Devices White Paper (WP485)," *Xilinx White Paper*, 2016.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[30] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[32] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[33] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[34] E. Burnell, N. B. Damen, and W. Hoburg, "GPkit: A Human-Centered Approach to Convex Optimization in Engineering Design," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020.

[35] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano, "PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite," in *Mathematical Software – ICMS 2016*. Springer International Publishing, 2016, pp. 301–307.

[36] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-Based DNN Accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400302.3415609

[37] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo, and J. Cong, "FlexCNN: An End-to-End Framework for Composing CNN Accelerators on FPGA," *ACM Trans. Reconfigurable Technol. Syst.*, dec 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3570928

[38] "Vitis-AI/models/AI-Model-Zoo," https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo, accessed: 2021-11-20.