AutoDNNchip: An Automated DNN Chip Predictor and Builder for Both FPGAs and ASICs

Pengfei Xu[1], Xiaofan Zhang[2], Cong Hao[2], Yang Zhao[1], Yongan Zhang[1], Yue Wang[1], Chaojian Li[1], Zetong Guan[1], Deming Chen[2], Yingyan Lin[1]
[1]Rice University, TX, USA, [2]University of Illinois at Urbana-Champaign, IL, USA
{eiclab, zy34, yz87, yw68, cl114, zg20, yingyan.lin}@rice.edu, {xiaofan3, congh, dchen}@illinois.edu

ABSTRACT

Recent breakthroughs in Deep Neural Networks (DNNs) have fueled a growing demand for domain-specific hardware accelerators (i.e., DNN chips). However, designing DNN chips is non-trivial because: (1) mainstream DNNs have millions of parameters and operations; (2) the design space is large due to the numerous design choices of dataflows, processing elements, memory hierarchy, etc.; and (3) an algorithm/hardware co-design is needed to allow the same DNN functionality to have a different decomposition, which would require different hardware IPs that correspond to dramatically different performance/energy/area tradeoffs. Therefore, DNN chips often take months to years to design and require a large team of cross-disciplinary experts. To enable fast and effective DNN chip design, we propose *AutoDNNchip* – a DNN chip generator that can automatically generate both FPGA- and ASIC-based DNN chip implementation (i.e., synthesizable RTL code with optimized algorithm-to-hardware mapping (i.e., dataflow)) given DNNs from machine learning frameworks (e.g., PyTorch) for a designated application and dataset without humans in the loop. Specifically, *AutoDNNchip* consists of two integrated enablers: (1) a *Chip Predictor*, built on top of a graph-based accelerator representation, which can accurately and efficiently predict a DNN accelerator's energy, throughput, latency, and area based on the DNN model parameters, hardware configuration, technology-based IPs, and platform constraints; and (2) a *Chip Builder*, which can automatically explore the design space of DNN chips (including IP selection, block configuration, resource balance, etc.), optimize chip design via the *Chip Predictor*, and then generate synthesizable RTL code with optimized dataflows to achieve the target design metrics. Experimental results show that our *Chip Predictor*'s predicted performance differs from real-measured ones by <10% when validated using 15 DNN models and 4 platforms (edge-FPGA/TPU/GPU and ASIC). Furthermore, both the FPGA- and ASIC-based DNN accelerators generated by our *AutoDNNchip* can achieve better (up to 3.86× improvement) performance than that of expert-crafted state-of-the-art accelerators, showing the effectiveness of *AutoDNNchip*. Our open-source code can be found at https://github.com/RICE-EIC/AutoDNNchip.git.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02…$15.00
https://doi.org/10.1145/3373087.3375306

ACM Reference Format:
Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, Yingyan Lin. 2020. *AutoDNNchip*: An Automated DNN Chip Predictor and Builder for Both FPGAs and ASICs. In *2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20), February 23-25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3373087.3375306

1 INTRODUCTION

We have seen the rapid adoption of Deep Neural Networks (DNNs) for solving real-life problems, such as image classification [1, 2], object detection [3], natural language processing [4], etc. Although DNNs enable high-quality inferences, they also require a large amount of computation and memory demand during deployment due to their inherently immense complexity [5–9]. Moreover, DNN-based applications often require not only high inference accuracy, but also aggressive hardware performance, including high throughput, low end-to-end latency, and limited energy consumption. Recently, we have seen intensive studies on DNN accelerators in hardware, which attempt to take advantage of different hardware design styles, such as GPUs, FPGAs, and ASICs, to improve the speed and efficiency of DNN inference and training [10**?** –21].

However, developing customized DNN accelerators presents significant challenges as it asks for cross-disciplinary knowledge in machine learning, micro-architecture, and physical chip design. Specifically, to build accelerators on FPGAs or ASICs, it is inevitable to include (1) customized architectures for running DNN workloads, (2) RTL programming for implementing accelerator prototypes, and (3) reiterative verifications for validating the functionality correctness. The whole task requires designers to have a deep understanding of both DNN algorithms and hardware design. In response to the intense demands and challenges of designing DNN accelerators, we have seen rapid development of high-level synthesis (HLS) design flow [22–25] and DNN design automation frameworks [16, 26–30] that improve the hardware design efficiency by allowing DNN accelerator design from high-level algorithmic descriptions and using pre-defined high-quality hardware IPs. Still, they either rely on hardware experts to trim down the large design space (e.g., use pre-defined/fixed architecture templates and explore other factors [16, 29]) or conduct merely limited design exploration and optimization, hindering the development of optimal DNN accelerators that can be deployed into various platforms.

To address the challenges above, we propose *AutoDNNchip*, an end-to-end automation tool for generating optimized FPGA- and ASIC-based accelerators from machine learning frameworks (e.g., Pytorch/Tensorflow) and providing fast and accurate performance estimations of hardware accelerators implemented on various targeted devices. The main contributions of this paper are as follows:

- ***One-for-all Design Space Description.*** We make use of a graph-based representation that can unify design factors in all of the three design abstraction levels (including IP, architecture, and hardware-mapping levels) of DNN accelerator design, allowing highly flexible architecture configuration, scalable architecture/IP/mapping co-optimization, and algorithm-adaptive accelerator design.
- ***Chip Predictor.*** Built on top of the above design space description, we propose a DNN *Chip Predictor*, a multi-grained performance estimation/simulation tool, which includes a coarse-grained, analytical-model based mode and a fine-grained, run-time-simulation based mode. Experiments using 15 DNN models and 4 platforms (edge-FPGA/TPU/GPU and ASIC) show that our *Chip Predictor*'s predicted error is within 10% of real-measured energy/latency/resource-consumption.
- ***Chip Builder.*** We further propose a DNN *Chip Builder*, which features a two-stage Design Space Exploration (DSE) methodology. Specifically, our *Chip Builder* realizes: (1) an architecture/IP design based on the *Chip Predictor*'s coarse-grained, analytical-model based prediction for a 1st-stage fast exploration and optimization, and (2) an IP/pipeline design based on the *Chip Predictor*'s fine-grained, run-time-simulation based prediction as a 2nd-stage IP-pipeline co-optimization. Experiments show that the *Chip Builder*'s 1st-stage DSE can efficiently rule out infeasible choices, while its 2nd-stage co-optimization can effectively boost the performance of remaining design candidates, e.g., 36.46% throughput improvement and 2.4× idle cycles reduction.
- ***AutoDNNchip.*** Integrating the aforementioned two enablers (i.e., *Chip Predictor* and *Chip Builder*), we develop *AutoDNNchip*, which can automatically generate optimized DNN accelerator implementation (i.e., synthesizable RTL implementation) given the user-defined DNN models from machine learning frameworks (e.g., Pytorch), application-driven specifications (e.g., energy and latency), and resource budget (e.g., size of the processing array and memories). Experiments demonstrate that the optimized FPGA- and ASIC-based DNN accelerators generated by *AutoDNNchip* outperform the recent award-winning design [31] by 11% and a state-of-the-art accelerator [32] by up to 3.86× .

As an automated DNN accelerator design tool, *AutoDNNchip* **is the first to highlight** all of the following features: (1) efficient and accurate performance prediction of DNN accelerators on 4 platforms, enabling fast optimal algorithm-to-accelerator mapping design and algorithm/accelerator co-design/co-optimization; (2) a design space description that unifies the descriptions of design factors from all of the three design abstraction levels in DNN accelerators into one directed graph, supporting arbitrary accelerator architectures (e.g., both homogeneous and heterogeneous IPs and their inter-connections), and (3) can automatically generate both FPGA- and ASIC-based DNN accelerator implementation that outperforms expert-crafted state-of-the-art designs for various applications.

2 BACKGROUND AND RELATED WORKS

FPGA- and ASIC-based DNN Accelerators. There has been intensive study in customized FPGA- and ASIC-based DNN accelerators. The accelerator in [10] uses loop tiling for accelerating convolutional layers on FPGAs. The DNNBuilder accelerator [16] applies

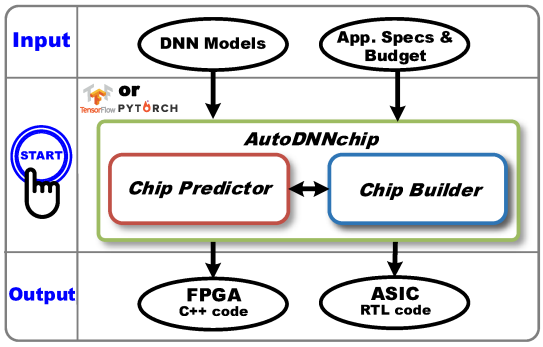


Figure 1: Overview of the proposed *AutoDNNchip* framework, which accepts user-defined DNN models/datasets and application-driven specifications to automatically generate optimized FPGA- or ASIC-based DNN accelerator designs.

an optimal resource allocation strategy, fine-grained layer-based pipeline, and column-based cache to deliver high-quality FPGA-based DNN accelerators. The work in [13] proposes a throughput-oriented accelerator with multiple levels (i.e., task, layer, loop, and operator levels) of parallelisms. The recent designs in [31, 33] introduce a hardware-efficient DNN and accelerator co-design strategy by considering both algorithm and hardware optimizations, using DNN building blocks (called Bundles) to capture hardware constraints. For ASIC-based DNN accelerators, efforts have been made in both industry and academia, where representative ones include TPU [17, 18], ShiDianNao [20], and Eyeriss [21], and different accelerators exploit different optimizations for various applications.

DNN Accelerator Performance Prediction. For designing FPGA-based DNN accelerators, current practice usually relies on roofline models [10] or customized analytical tools [13, 16] to estimate the achievable performance. For ASIC-based accelerators, recently published designs [21, 34, 35] introduce various performance prediction methods. Eyeriss [21] proposes an energy model for capturing the energy overhead of the customized memory and computation units and a delay model that simplifies the latency calculation. Similarly, MAESTRO [34] develops an energy estimation model that considers hardware design configurations and memory access behaviors, while Timeloop [35] adopts a loop-based description of targeted workloads and analyzes the data movement and memory access for latency estimation.

DNN Accelerator Generation. The tremendous need for developing FPGA-/ASIC-based DNN accelerators motivates the development of automated DNN accelerator generation. For example, DeepBurning [26] is a design automation tool for building FPGA-based DNN accelerators with customized design parameters using a pre-constructed RTL module library. DNNBuilder [16] and FP-DNN [28] propose end-to-end tools that can automatically generate optimized FPGA-based accelerators from high-level DNN symbolic descriptions in Caffe/Tensorflow frameworks. Caffeine [27] is another automation tool that provides guidelines for choosing FPGA hardware parameters, such as the number of processing elements (PEs), bit precision of variables, and parallel data factors. By using these automation tools, it is easier to bridge the gap between fast DNN construction in popular machine learning frameworks and slow implementation of targeted hardware accelerators.

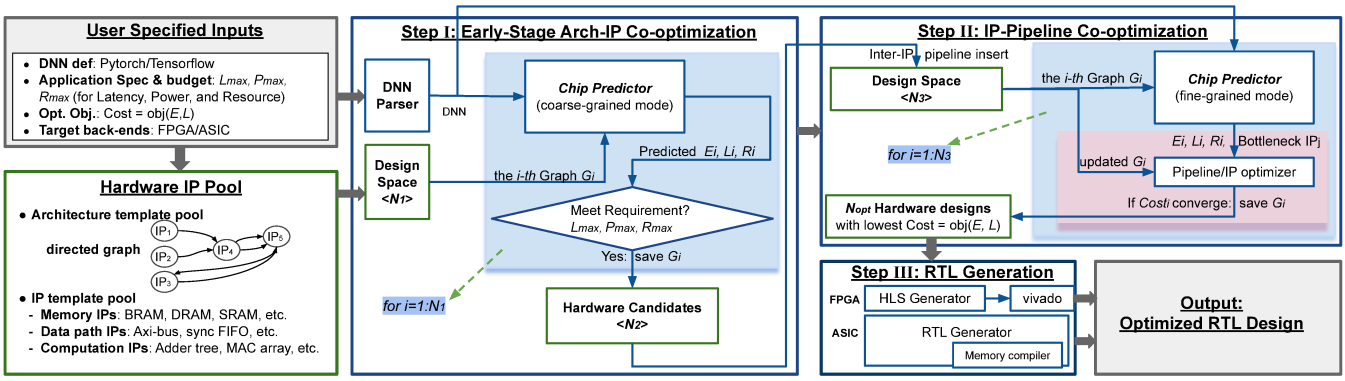


Figure 2: *AutoDNNchip*'s three-step design flow for the design space exploration, optimization, and DNN-to-RTL generation.

3 OVERVIEW OF *AUTODNNCHIP*

Fig. 1 shows an overview of the proposed *AutoDNNchip*, which can automatically generate optimized FPGA- or ASIC-based DNN accelerators as well as an optimal algorithm-to-hardware mapping (i.e., dataflow), according to three customized inputs: (1) the high-level DNN descriptions trained in desired datasets, (2) the application-driven specifications regarding DNN inference quality and hardware performance, and (3) the available resources of targeted platforms. The realization of *AutoDNNchip* is achieved by the proposed *One-for-all Design Space Description* (see **Section 4**), *Chip Predictor* (see **Section 5**), and *Chip Builder* (see **Section 6**).

One of the major challenges that *AutoDNNchip* needs to overcome is the lack of effective representations of DNN accelerators' large design space given the numerous design choices (e.g., dataflows, the number of pipeline stages, parallelism factors, memory hierarchy, etc.), as a precise and concise representation is a precondition of valid DNN accelerator design. To address this challenge, we propose a *One-for-all Design Space Description*, which is an object-oriented graph-based definition for DNN accelerator design that can unify the description of design factors from all of the **three** design abstraction levels into **one** directed graph. Furthermore, *AutoDNNchip* features another two key enablers, the *Chip Predictor* and the *Chip Builder*. Specifically, the proposed *Chip Predictor* can accurately and efficiently estimate the energy, throughput, latency, and area overhead of DNN accelerators based on the parameters that can characterize the algorithms, hardware architectures, and technology-based IPs. The proposed *Chip Builder* can automatically (1) explore the design space of DNN accelerators (including IP selection, block configuration, resource balance, etc.), (2) optimize chip designs via the *Chip Predictor*, and (3) generate synthesizable Verilog code to achieve target design metrics.

4 ONE-FOR-ALL DESIGN SPACE DESCRIPTION

Overview. It is well known that the design space of DNN accelerators can be very large. For effective and efficient design space exploration and optimization, it is critical that the design space can be precisely and concisely described, e.g., that the different design abstraction levels of optimization in DNN accelerator design, including architecture level, IP level, and hardware-mapping level, are considered. To this end, we adopt a *One-for-all Design Space Description* that unifies the design factors of the three levels into one directed graph. Table 1 lists the design factors which are sufficient for most cases and the last column shows the levels of design/optimization that may influence the corresponding factors. We can see that (1) most of the design factors are related to cross-level optimization which also reflects the fact that DNN accelerators have a large design space and (2) optimization at merely one level (or one hardware component) does not guarantee overall system performance. We thus adopt an object-oriented directed graph for the DNN accelerator design space description, an illustrative example of which is shown in Fig. 3. Specifically, a basic directed graph is first constructed using the PE array architecture, memory architecture and mapping/dataflow factors, where each node in the graph denotes a computation/data-path/memory IP and each directed edge denotes an inter-connection between nodes whose direction is determined by the corresponding data movement's direction. Proper attributes (e.g., those in Table 2) are then assigned to the nodes and edges of the directed graph in an object-oriented manner. In the following subsections, we will briefly describe four graph-based accelerator templates corresponding to four state-of-the-art DNN accelerators which are stored in the *Hardware IP Pool* (see Fig. 2 under the *User Specified Inputs*) of *AutoDNNchip* together with other templates to provide a sufficient number of design candidates, and then discuss the IP attributes for the nodes and edges.

Graph-based Accelerator Templates. Fig. 4 shows four graph-based accelerator template examples for describing DNN accelerators that can be translated into real hardware implementation by applying appropriate IP attributes. Specifically, Fig. 4 (a) shows a spatial architecture based on a single adder-tree based computation

Table 1: A summary of DNN accelerators' design factors.

Design factor	Description	Back-end	Opt. level
B_W, B_A, B_{Acc} [a]	Bit precision	F, A [b]	IP, Accuracy req.
$Freq.$	Clock frequency	F, A	Arch., IP
$Arch_{mem}$	Memory tech/hierarchy/volume	A	Arch., IP, Mapping
$Arch_{pe}$	PE array architecture	F, A	Arch., IP, Mapping
Bw	Port/Bus width for data transfer	A	Arch., IP
$Malloc$	Memory allocation	F, A	Arch., IP, Mapping
$Data\ Schedule$	DNN to accelerator mapping	F, A	Arch., IP, Mapping

[a] B_W, B_A, B_{Acc}: Bit precision for weights, activations, accumulations
[b] A: ASIC design; F: FPGA design

Table 2: A summary of attributes for the nodes and edges in the graph-based description

Compo.[a]	Hardware meaning	Attributes
Node	Memory IPs	Impl., Freq., Vol., Prec., Dt., StM., E, L[b]
	Computation IPs	Impl., Freq., Prec., StM., E, L
	Data Path IPs	Impl., Freq., Bw.[c] Prec., Dt., StM., E, L
Edge	IP inter-connections (IP dependency)	Start, End [d]

[a] Compo.: graph components including nodes and directed edges;
[b] Impl.: implementation, e.g., 14nm DRAM, 28nm SRAM, DSP48E, AXI-bus, sync FIFO, etc.; Freq.: clock frequency (MHz); Vol.: volume/capacity (bits); Prec.: bit precision; Dt.: data type including weights, input activations, and partial sums.; E/L: energy&latency overhead; StM.: the state machine storing all the states (including needed inputs and generated outputs) through the whole execution process;
[c] Bw.: port/bus width; [d] Start & End: the start and ending node for the directed edge.

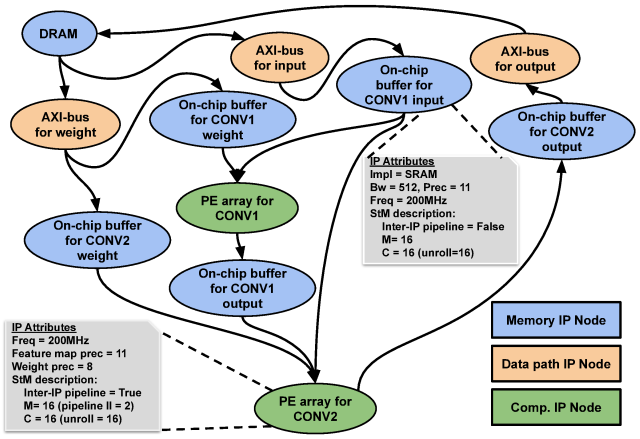


Figure 3: An illustrative example of the graph-based design space description for a heterogeneous architecture to accelerate residual block in ResNet [36], where M and C denote the output and input channel, respectively.

IP, which is a commonly-used architecture on FPGA-based accelerators; Fig. 4 (b) is a graph with 2 different computation IPs, including depth-wise convolutional (denoted as DW_CONV) and normal convolutional (denoted as CONV) ones commonly adopted in compact DNN models, and two BRAM IPs that handle the memory data arrangement for the computation IPs; Fig. 4 (c) is an architecture template for TPU [17] type DNN accelerators using a systolic array; and Fig. 4 (d) shows the graph-based representation for DNN accelerators with Eyeriss [21] type architectures, where the data path IPs (i.e., NoC IPs in Fig. 4 (d)) between PEs describe the local data reuse patterns of inputs, outputs, and weights.

IP Attributes. Table 2 summarizes the attributes for three types of node IP including memory (e.g., BRAM and off-Chip DRAM), data access (e.g., bus), and computation hardware that characterizes the corresponding design, as elaborated below: (1) The *Implementation or Impl.* attribute refers to the required hardware resource for implementing the IPs, e.g., DRAM and SRAM for implementing memory IPs, and AXI-bus and NoC for implementing data path IPs; (2) The *state machine or StM.* attribute is used to describe when the IPs will update their states between computation and loading/unloading data, where each state defines both the needed input address and generated output address. Fig. 5 shows that different pipeline designs can be captured by the IPs' state machine attribute:

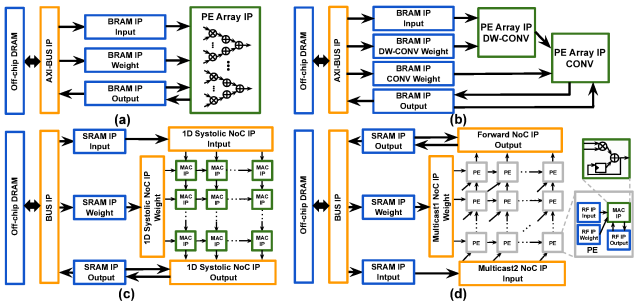


Figure 4: An illustration of 4 architecture templates in our *Hardware IP Pool* including 2 architectures for both state-of-the-art FPGA- and ASIC-based DNN accelerators.

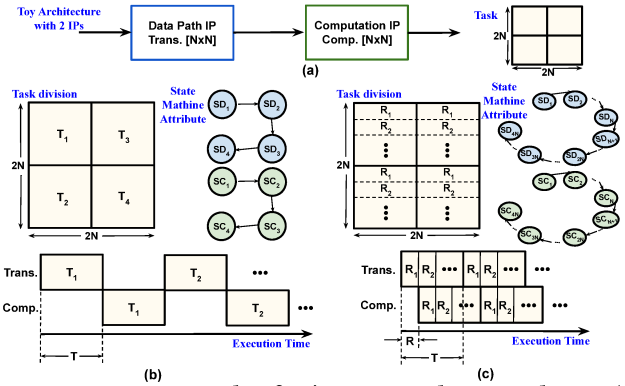


Figure 5: A toy example of IP's state machine attribute w/o and w/ considering inter-IP pipeline effects: (a) a simple architecture with 2 IPs, i.e., one data path IP and one computation IP; the task division, state machine, and the run-time process when (b) excluding the inter-IP pipeline and (c) considering the inter-IP pipeline, where SD and SC denote the state for data path IP and computation IP, respectively.

Fig. 5 (b) and (c) illustrate two kinds of designs (w/o and w/ inter-IP pipeline) and their corresponding state machine definition, respectively, where there are more states in Fig. 5 (c) for capturing the inter-IP pipeline between data transfer and computation IPs; (3) The *data precision or Prec.* attribute refers to the IPs' bit precision; (4) The *clock Frequency or Freq.* and *energy/latency or E/L* attributes capture the operating clock frequency and required energy/latency for IPs; and (5) The *port/bus width or Bw* and *memory volume or Vol.* attributes refers to the port/bus width of data path IPs and memory volume for memory IPs, respectively.

5 THE PROPOSED *CHIP PREDICTOR*

5.1 Overview

As shown in Fig. 6, the proposed *Chip Predictor* accepts DNN models (e.g., number of layers, layer structure, precision, etc.), hardware architectures (e.g., memory hierarchy, number of PEs, NoC design, etc.), hardware mapping, and IP design (e.g., unit energy/delay cost of a multiply-and-accumulate (MAC) operation and memory accesses to various memory hierarchies), and then outputs the estimated energy consumption, latency, and resource consumption when executing the DNN in the target accelerator defined by the given hardware architecture and hardware mapping. First, to capture the large search space and consider all the design abstraction levels (including architecture, IP, and hardware mapping levels), we construct a graph-based description that serves as one input of *Chip Predictor*. Second, to match the different tradeoff requirements

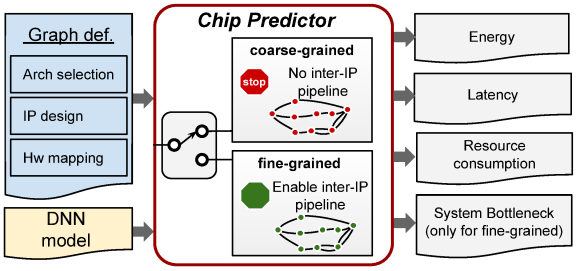


Figure 6: An overview of the proposed *Chip Predictor*.

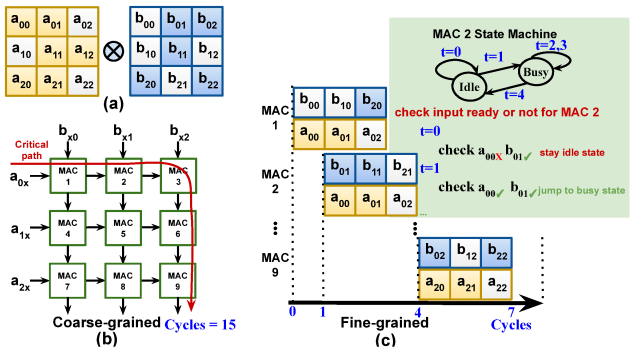


Figure 7: A toy example using a systolic array, illustrating: (a) the corresponding matrix-matrix multiplication, and (b) coarse-grained and (c) fine-grained latency estimation.

of the *Chip Builder*'s two-stage DSE, which aims for efficient and accurate design space exploration and optimization, our *Chip Predictor* adopts a mixed-granularity prediction: (1) a coarse-grained mode that can quickly provide IP performance estimation to enable the identification of critical paths when the inter-IP pipeline is not considered, in order to be used for the *Chip Builder*'s early-stage architecture and IP exploration and selection; and (2) a fine-grained mode that can perform accurate performance prediction by considering the pipeline dependency between IPs based on run-time simulations, in order to be used for the *Chip Builder*'s 2nd-stage DSE that targets IP-pipeline co-optimization.

5.2 The *Chip Predictor*'s Coarse-grained Mode

Overview. The *Chip Predictor*'s coarse-grained mode is analytical-model-based, i.e., using equations to formulate the accelerators' energy, critical path latency, and resource consumption given a DNN model and a graph-based hardware design description (see Fig. 6). Specifically, the energy and latency of IPs are first calculated using: (1) analytical equations as described below and (2) the attributes of each IP, where the unit energy/latency costs are obtained from single-IP RTL implementation or simulations; and the energy and latency consumption of the whole DNN accelerator are formulated by considering: (1) the total energy and latency of all IPs when executing the DNN model and (2) the energy and latency overhead of the CPU and on-chip controller.

Analytical-model-based Intra-IP Modeling. If we define the energy, latency and resource utilization as E, L, and R, respectively, and use ip_{comp}, ip_{dp}, ip_{mem} to denote the computation IP, data path IP and the memory IP, respectively, the energy and latency of the computation IPs can be formulated by:

$$E_{ip_{comp}} = e_1 + (\#states) \times (e_2 + e_{mac} \times U) \tag{1}$$

$$L_{ip_{comp}} = l_1 + (\#states) \times l_{mac} \tag{2}$$

where $(\#states)$ denotes the total number of the states in the IP state machine; U denotes the unrolling factor (PE parallelism) for the computation IP; e_{mac} and l_{mac} denote the unit energy and latency costs for a MAC operation, respectively; e_1 and l_1 are the energy and latency overhead for warming up, i.e., configure the data path and pre-load data; and e_2 denotes the energy overhead of run-time control of CPU or on-chip logic units. Meanwhile, the energy and latency of the data path IPs can be formulated by:

$$E_{ip_{dp}} = e_3 + (\#states) \times (e_4 + V \times e_{bit}) \tag{3}$$

$$L_{ip_{dp}} = l_2 + (\#states) \times (l_3 + \frac{V}{P_w} \times l_{bit}) \tag{4}$$

where V denotes the total data volume (bits) needed to be transferred when the IPs are called; P_w denotes the port width for the corresponding data path; e_{bit} and l_{bit} denote the unit energy and latency costs for each bit of data access, respectively; e_3 and l_2 are the energy and latency overhead for warming up, respectively; and e_4 and l_3 denote the energy and latency overhead of the run-time control of CPU or on-chip logic units, respectively.

Analytical-model-based Inter-IP Modeling. For the system performance, including energy, latency, and resource consumption of a convolutional layer or a DNN building block (e.g., the Bundle in [31, 33]), the resource and energy consumption are obtained by summing up that of all the IPs in the graph, and the total latency can be calculated by summing up the latency of all the IPs on the critical path of the graph, i.e.,

$$R_{mem} = \sum_{ip_{mem} \in G} Vol_{ip_{mem}} \tag{5}$$

$$R_{mul} = \sum_{ip_{comp} \in G} U_{ip_{comp}} + R_{mul_{dec}} \tag{6}$$

$$E = \sum_{ip \in G} E_{ip} \tag{7}$$

$$L = \max_{path \in G} \sum_{ip \in path} L_{ip} \tag{8}$$

where G denotes the whole graph; R_{mem} denotes the total memory volume consumption for one type of memory; R_{mul} denotes the total number of multipliers used in both the computation IPs and when decoding the memory address, with the latter denoted as $R_{mul_{dec}}$. Regarding the latency estimation, the inter-IP pipeline effects are excluded in the coarse-grained mode and it can be captured in the fine-grained mode of *Chip Predictor* (see Section 5.3). As a toy example, Fig. 7 (b) and (c) illustrate the latency estimation when operating a matrix-matrix multiplication in a systolic array using both the coarse-grained mode and fine-grained mode, where the resulting estimated latency results are 15 and 7 cycles, respectively.

5.3 The *Chip Predictor*'s Fine-grained Mode

Overview. In the fine-grained mode of the *Chip Predictor*, we adopt: (1) *Algorithm* 1 to perform run-time simulations based on inter-IP pipeline to obtain the corresponding inter-IP latency; and (2) the *Chip Predictor*'s coarse-grained mode to get the IPs' energy and latency for estimating the intra-IP performance.

Implementation. The run-time simulation algorithm is described in *Algorithm* 1, where each IP (denoted as *ip*) has (1) its neighbour IPs on the graph defined as *ip.prev* and *ip.next*, respectively, and *ip* will use the data from *ip.prev* as its inputs and pass its outputs to *ip.next*; and (2) a state machine to store different states (including its needed inputs and generated outputs) through the whole execution process. For each clock cycle in the simulation, *ip* can jump to the next state when (1) it has finished generating all the outputs in its current state (i.e., *ip* is in an idle status) and (2) *ip.prev* has generated all the inputs *ip* needed for the next state. If *ip* is in an idle status but its needed inputs are not ready from *ip.prev*, it will continue to wait on the idle status, resulting in an increase of the idle cycles associated with this IP; If *ip* is in a busy status, it will generate its outputs and jump to an idle status when it finishes generating all the outputs in this state.

Algorithm 1 Run-time sim. in the fine-grained *Chip Predictor*

Input: One accelerator design described by graph G;
For each $edge$ in G
  $ip_{start} \longleftarrow edge'$s starting node;
  $ip_{end} \longleftarrow edge'$s ending node;
  Add ip_{start} to $ip_{end}.prev$;
  Add ip_{end} to $ip_{start}.next$;
Initialize energy and latency: $E = 0$, $cycles = 0$;
While not all inference outputs are stored back
  $cycles \longleftarrow cycles + 1$;
  For each ip in G
    If (ip is $idle$) & (all needed inputs $\in$ outputs of $ip.prev$)
      $ip \longleftarrow busy$;
      ip jumps to the next state;
    If (ip is $idle$) & (not all needed inputs $\in$ outputs of $ip.prev$)
      $ip.idle\_cycles \longleftarrow ip.idle\_cycles + 1$;
    If (ip is $busy$) & (not all outputs for ip is ready)
      Update the ready outputs for ip;
    If (ip is $busy$) & (all outputs for ip is ready)
      $ip \longleftarrow idle$;
      $E \longleftarrow E + E_{ip}$;
$L \longleftarrow \frac{cycles}{global\ clk\ freq}$;
$ip_{bottleneck} \longleftarrow ip$ with minimum idle cycles.

For better understanding, Fig. 7 uses a toy example to show that the *Chip Predictor*'s fine-grained mode (see Fig. 7 (c)) can more accurately estimate the required latency than its coarse-grained mode. In this 3×3 systolic array with the local-data-forwarding and computation operations being pipelined, we assume each MAC unit takes 3 cycles to do the computation and 1 cycle to forward the data to its nearby MAC units. In the coarse-grained mode case, we add the intra-IP latency in the graph's critical path to estimate the overall latency (see Fig. 7 (b)), resulting in an estimated latency of 15 cycles. In the fine-grained mode case (see Fig. 7 (c)), we define the state machine for each MAC unit and adopt *Algorithm* 1 to keep track of when each MAC unit jumps to the next state. In this particular example, MAC 2 will wait at cycle 0 since its required input data a00 is not ready, and it will jump to next state to start computing at cycle 1 when all its required inputs are ready. We can see that the fine-grained mode's estimated latency (7 cycles, the same as the ground truth) is more accurate for modeling the overlapped computation and data transferring in this example. In practical designs, the overall latency is not determined by merely one stage, so the *Chip Builder* will launch the *Chip Predictor* to simulate the whole graph iteratively in order to generate an optimal design for the whole accelerator system.

6 THE PROPOSED *CHIP BUILDER*

Fig. 2 elaborates the design flow of *AutoDNNchip* that leverages the *Chip Builder*'s two-stage DSE engine. To effectively explore the design space (e.g., the design factors in Table 1), *AutoDNNchip* involves three major steps as shown in Fig. 2: (1) the 1st-stage DSE: an early stage architecture and IP configuration exploration to efficiently rule out infeasible designs using the *Chip Predictor*'s coarse-grained mode; (2) the 2nd-stage DSE: an inter-IP pipeline exploration and IP optimization to effectively boost the performance of the remaining design candidates resulting from the 1st-stage DSE; and (3) a design validation through RTL generation and execution.

Step I. Early Stage Architecture and IP Configuration Exploration. As shown in the middle part of Fig. 2, this step considers the following exploration. First, the DNN model from a mainstream machine learing framework is applied to the DNN parser to extract the DNN layer information, e.g., layer types (CONV, Pooling, ReLU, Reorg [31], etc.), feature map inter-connections (Concat, Add, etc.), and layer shapes (shape of weight and feature map tensors). Second, according to the given DNN model, performance requirements (e.g., latency and throughput) and hardware budgets (e.g., resource and power budget of FPGA or ASIC), a design space of size N_1 is generated by fetching commonly-used or promising hardware architecture templates and hardware IP templates from the *Hardware IP pool*. For example, when the given resource budgets are tight, a folded hardware architecture will be chosen instead of a flattened one; whereas flattened structures which facilitate IP pipelines are preferred when there are sufficient budgets. Third, an architecture and IP configuration optimization is then performed to rule out most of the infeasible choices and trim down the design space to N_2 ($N_2 < N_1$) promising candidates, e.g, more efficient with a lower latency. This fast early exploration makes use of the analytical nature of the *Chip Predictor*'s coarse-grained mode.

Algorithm 2 IP-pipeline co-optimizationusing the *Chip Builder*

Input: Design space D_G with N_2 graphs;
For each G in D_G
  For each $edge$ in G
    $ip_{start} \longleftarrow edge'$s starting node;
    $ip_{end} \longleftarrow edge'$s ending node;
    Add ip_{start} to $ip_{end}.prev$;
    Add ip_{end} to $ip_{start}.next$;
  While simulated (using Algorithm 1) latency L_G does not converge
    $ip \longleftarrow$ simulated bottleneck IP (i.e., $ip_{bottleneck}$ from Algorithm 1);
    If inter-IP pipeline is adopted for ip and $ip.next$
      allocate more resource to ip;
    Else
      adopt inter-IP pipeline between ip and $ip.next$;
      update the state machine of ip;
      update the state machine of $ip.next$;
Select top N_{opt} candidates in D_G

Step II. Inter-IP Pipeline Exploration and IP Optimization. This step accepts the resulting N_2 designs and performs further exploration and IP optimization using *Algorithm* 2. First, inter-IP pipelines are inserted into different locations of the corresponding computation graphs, resulting in a new design space of size N_3, i.e., N_3 new graphs with different inter-IP pipeline designs. Second, for each of these graphs, the bottleneck IPs will be recorded during *Algorithm* 1's run-time simulations and then optimized via deeper inter-IP pipeline design or re-allocating more resource until convergence based on the *Chip Predictor*'s fine-grained mode's predicted performance, as shown in *Algorithm* 2. Third, the top N_{opt} design candidates will be chosen according to the *Chip Predictor*'s predicted energy consumption or/and latency, and then passed to the next step for validation through RTL generation and execution.

Step III. Design Validation through RTL Generation and Execution. In this step, we generate RTL code for the top N_{opt} optimized designs through an automated code generation procedure: (1) For the FPGA back-end, the generated files include the testbench for a board-level implementation, the binary file for the quantized-and-reordered weights, and the C-code for the HLS IP implementation. We use Vivado [22] to actually generate the bitstream and meanwhile eliminate the designs that fail in place and route (PnR) to guarantee that *AutoDNNchip*'s generated designs are valid; (2) For the ASIC back-end, the generated files include the RTL testbench for the DNN model, the quantized-and-reordered weights, the synthesizable RTL code, and the memory specifications. The RTL code could be further passed to an EDA tool like

Design Compiler and IC Compiler to generate gate-level/layout netlist, during which Memory Compilers could take the memory specifications to generate the memory design. After this step, all the output designs are fully validated with correct functionality.

7 EXPERIMENT RESULTS

In this section, we evaluate the proposed *AutoDNNChip* on 20 DNN models across 4 platforms (3 edge devices including edge-FPGA/TPU/GPU and 2 ASIC-based accelerators).

7.1 Validation of the *Chip Predictor*

Methodology and Setup. Table 3 summarizes the details of our validation experiments for the *Chip Predictor*, including the platforms, performance metrics, DNN models, methods to obtain the unit parameters, employed precision for the weights and activations, and frequency of the corresponding computation core. **Methodology.** In order to conduct a solid validation, we validate the *Chip Predictor* by comparing its predicted performance with actual device-measured ones on 3 edge devices (Ultra96 FPGA [37], edge TPU [18], Jetson TX2 [38]) and paper-reported ones of 2 published ASIC-based accelerators (Eyeriss [21] and ShiDianNao [20]), when adopting the same experiment settings (e.g., clock frequency, DNN model and dataset, bit precision, architecture design, and dataflow, etc). **Benchmark DNN Models and Datasets.** For the 3 edge devices, we consider 15 representative compact/light-weight DNN models (see Table 4 and Table 5, where the models in Table 5 use the ImageNet dataset [39] and the models in Table 4 use the dataset in the System Design Contest of the DAC 2019 conference [40]); for the 2 published DNN accelerators, we use the same benchmark models and datasets as the original papers. **Unit Parameters.** The unit energy/latency parameters are obtained through either real-device measurement or synthesized RTL implementation as mentioned in Section 5. For the 3 edge devices, we measure the unit energy and latency by running the basic IP operations (such as the memory accesses and the MAC computation) over multiple sets of experiments under different settings and average the energy and latency values to get unit parameters. Specially, for memory accesses, we change the clock frequency, memory volume, port width, bit precision, and burst read length; for the MAC operations, the clock frequency, total number of MACs and parallelism of MACs are changed. For the ASIC-based accelerators, the unit parameters are obtained either from the paper [21] or gate-level simulations of the synthesized RTL implementation on the same CMOS technology.

Table 3: Experiment settings for the *Chip Predictor*'s cross-platform/model/design/dataset validation.

Arch/Device	Metrics[a]	DNNs[b]	Unit Param.[c]	Precision <W,A>[d]	Freq.[e] (MHz)
Ultra96 [37]	*E, L, R*	Compact	Measured	<11, 9>	220
Edge TPU [18]	*E, L*	Compact	Measured	<8, 8>	500
Jetson TX2 [38]	*E, L*	Compact	Measured	<32, 32>	1300
Eyeriss [21]	*E, L, R*	AlexNet	Reported	<16, 16>	250
ShiDianNao [20]	*E*	Small	Synthesized	<16, 16>	1000

[a] Metrics – *E*: energy, *L*: latency, *R*: resource;
[b] DNN benchmarks – Compact: see the 15 compact DNN models in Table 4 and Table 5; AlexNet [41]; and Small: DNNs used in [20] (< 5 convolutional/fully-connected layers);
[c] Methods to obtain the unit parameters;
[d] Bit precision for different types of data, i.e., <weight precision, activation precision>;
[e] Clock frequency for the computation core.

Table 4: The 10 model variants of the SkyNet backbone [31].

DNN	SK	SK1	SK2	SK3	SK4	SK5	SK6	SK7	SK8	SK9
Size (MB)	1.75	1.79	2.11	1.18	1.77	3.21	3.79	3.05	0.96	1.95
Layer #	14	14	14	14	17	14	16	14	14	17
Bypass	✓	✓	✓	✓	✓	-	-	-	-	-

Table 5: The 5 model variants of MobileNetV2 [42] using different channel scaling factors and input resolutions.

DNN	V-Model 1	V-Model 2	V-Model 3	V-Model 4	V-Model 5
Resolution	128	128	224	224	224
Channel scaling	0.5	1.0	0.5	1.0	1.4

Validation of the Predicted Energy Consumption. We compare the *Chip Predictor*'s predicted energy with the measured ones from 3 edge devices, including Ultra96 FPGA [37] (edge FPGA), edge TPU [18], and Jetson TX2 (edge GPU) [38]) under the same settings (see Table 3).

Fig. 8 summarizes the validation results, and shows that **the maximum prediction error of our *Chip Predictor* is 9.17% for all 15 DNN models across 3 platforms**. Specifically, the prediction error ranges from 0.89% to 8.13%, 2.12% to 7.67%, and 2.72% to 9.17%, for the cases using the edge GPU, the edge FPGA, and the edge TPU, respectively, and the corresponding average prediction error is 5.40%, 5.20%, and 6.05%, respectively. We notice the energy consumption of the SkyNet and SK1-SK4 models are relatively large using the edge TPU. The reason is that these models contain unsupported operations (e.g., short-cut paths and feature map reorganization [31]) that need to be handled by the embedded CPU instead of the optimized tensor unit with higher efficiency.

In Fig. 9 and Table 6, we validate the proposed *Chip Predictor* by comparing it to 2 state-of-the-art ASIC-based accelerators: Eyeriss [21] and ShiDianNao [20]. For Eyeriss [21], we first compare the predicted energy breakdown of the first and fifth convolutional layers of AlexNet, of which the maximum error is 5.15% and 1.64%, respectively, as shown in Fig. 9 (a). Since the memory accesses dominate the energy consumption [21], we further compare the number of DRAM and SRAM accesses. In Fig. 9 (b), we present the error between the predicted and Eyeriss's paper-reported results. The relatively large error of SRAM accesses in the first convolutional layer is caused by the unsupported large stride number (4 in this case) since our predictor only considers the commonly used strides of 1 and 2 for simplicity. Note our *Predictor* can be straightforwardly

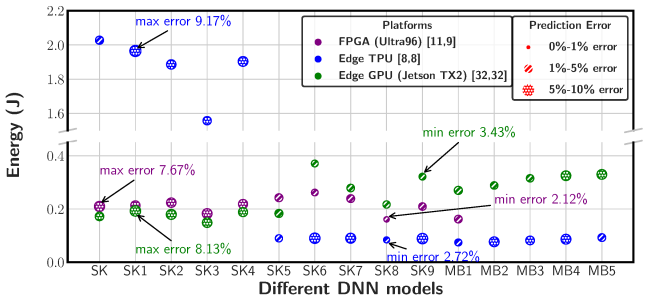
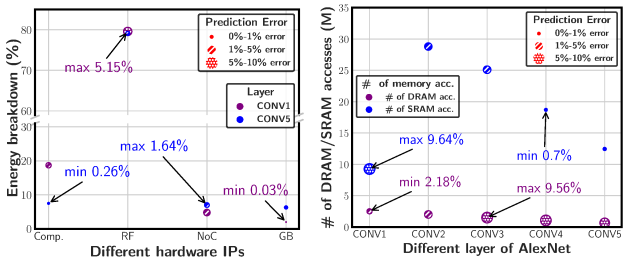


Figure 8: The energy prediction error of the *Chip Predictor* when using the 15 DNN models in Table 4 and Table 5 running on 3 edge devices including an edge FPGA [37], Edge TPU [18], and edge GPU [38].



(a) (b)

Figure 9: The *Chip Predictor*'s energy prediction error considering the Eyeriss architecture [43]: (a) The energy breakdown for AlexNet's 1st and 5th convolutional layers and (b) the # of DRAM and SRAM accesses of convolutional layers.

extended to include other stride values. Note that the prediction errors of DRAM accesses in the last three layers are relatively large, because the input data are compressed to save DRAM accesses in [21] and we lack their information regarding the input data sparsity. The validation results over ShiDianNao [20] are listed in Table 6. By showing the average energy over 10 DNN benchmarks of the 4 IPs in [20], we verify the maximum prediction error is 9.59%, where the error is mainly due to the difference between our adopted commercial CMOS IP library and the one used in [20].

Table 6: The energy prediction error of the *Chip Predictor* when using the architecture of ShiDianNao [20]: The energy breakdown over 10 benchmarks.

IP	Computation	Input SRAM	Output SRAM	Weight SRAM
Predicted (%)	89.2	7.4	1.7	1.6
Paper-reported (%)	89.0	8.0	1.6	1.5
Prediction error	**0.35%**	**-7.19%**	**9.59%**	**7.87%**

Validation of the Predicted Latency. The latency prediction of the *Chip Predictor* is validated over the measured results of the same 15 DNN models and 3 edge devices and shown in Fig. 10. he edge GPU, the Ultra96 FPGA board, and the edge TPU, respectively, and the corresponding average prediction error is 4.85%, 3.73%, and 6.57%, respectively. **The maximum latency prediction error of our *Chip Predictor* is 9.75%.** Specifically, the prediction errors range from 0.89% to 9.75%, 1.78% to 5.98%, and 2.92% to 9.44%, when using the edge GPU, the edge FPGA, and the edge TPU, respectively. The corresponding average prediction error is 4.85%, 3.73%, and 6.57%, respectively. Similar to the case in Fig. 8, the latency of the SkyNet and SK1-SK4 models when using the edge TPU are relatively large because of the unsupported operations.

Table 7 summarizes the latency prediction when running the 5 convolutional layers of AlexNet on Eyeriss [21] with the largest error peaking at 4.12%. The predicted latency generated by the proposed *Chip Predictor* is smaller than the paper-reported results, as *Chip Predictor* does not consider the special scenario when the accelerator needs to access memory multiple times for one single wordline of data. Such a scenario only happens when one wordline of data is physically stored in multiple wordlines of the memory. The *Chip Predictor* can be extended to include such a case by configuring corresponding memory data arrangements.

Validation of the Predicted Resource Consumption. Table 8 summarizes the *Chip Predictor*'s predicted resource consumption

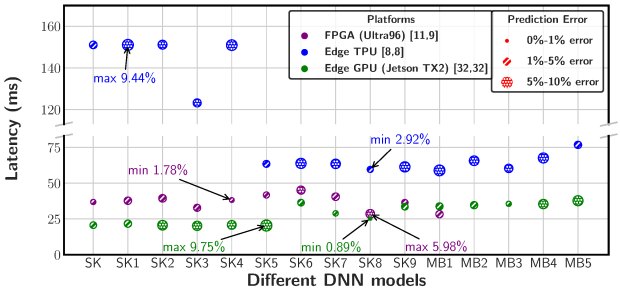


Figure 10: The latency prediction error of the *Chip Predictor* when using 15 DNN models on 3 edge devices including Ultra96 FPGA board, Edge TPU, and Jetson TX2 (edge GPU).

Table 7: The latency prediction error of the *Chip Predictor* when using the architecture of Eyeriss [21]: The latency when processing the 5 convolutional layers of AlexNet.

AlexNet Layer	CONV1	CONV2	CONV3	CONV4	CONV5
Predicted latency (ms)	16.04	37.58	21.09	15.59	9.79
Paper-reported latency (ms)	16.5	39.2	21.8	16	10
Prediction error	**-2.88%**	**-4.12%**	**-3.24%**	**-2.56%**	**-2.14%**

based on the experiments using Ultra96 FPGA [37]. Specifically, the predicted resource consumption for the 2 critical on-chip resources of FPGAs, DSP48E and BRAM18K, is validated against those obtained from the post-implementation utilization reports, and has a corresponding prediction error of smaller than 4.2% and 3.2%, respectively. Note that the DSP48Es are the embedded multipliers in the FPGA and the BRAM18K is the main on-chip memory resource in the FPGA. In addition, the 6 cases, i.e., Bg. 1-6 in Table 8, correspond to 6 designs under varied resource budgets. For the validation over ASIC-based DNN accelerators, we consider the MAC utilization as the validation metric. Estimated MAC utilization among the 5 convolutional layers of AlexNet is the same as the paper-reported ones in Eyeriss [21], because the MAC utilization is only determined by the parallelism level of the PE array in Eyeriss.

Table 8: The resource consumption prediction error of the *Chip Predictor*'s on the Ultra96 FPGA board when considering 6 different designs given 6 different resource budgets.

Resource type	Val.	Bg. 1	Bg. 2	Bg. 3	Bg. 4	Bg. 5	Bg. 6
DSP48E	Predicted	35	69	141	213	285	331
	Measured	36	72	144	216	288	360
	Error	**-3.2%**	**-4.2%**	**-2.4%**	**-1.2%**	**-1.0%**	**-0.8%**
BRAM18K	Predicted	65	87	175	265	354	446
	Measured	64	86	173	259	346	432
	Error	**+1.0%**	**+0.8%**	**+1.2%**	**+2.2%**	**+2.4%**	**+3.2%**

7.2 Evaluation for the *Chip Builder* and *AutoDNNchip*

In this subsection, we evaluate the performance of the proposed *Chip Builder*, which makes use of the time-efficient and accurate *Chip Predictor* to perform an effective two-stage DSE efficiently, and *AutoDNNchip*. Specifically, we study the performance of the resulting DNN accelerators generated and optimized by the *Chip Builder* and *AutoDNNchip*. First, we show experiment results to visualize the *Chip Builder*'s two-stage DSE process; Second, we study the performance improvement resulted from the *Chip Builder*'s second-stage IP-pipeline co-optimization in terms of bottleneck

blocks' latency and idle cycle reduction; Finally, we validate the effectiveness of the *AutoDNNchip* by comparing the performance of its generated FPGA- and ASIC-based accelerators (i.e., the corresponding RTL implementation) with that of state-of-the-art designs under the same conditions.

Evaluation Setup. In this set of experiments, we consider the application-driven specifications and constraints summarized in Table 9, where the throughput requirement and power budget are set to meet real-time applications of visual recognition (e.g., image classification and object detection [3]) on edge devices. For the FPGA-based accelerator design, we use a state-of-the-art edge device Ultra96 FPGA board [37] whose resource budget is fixed; for the ASIC-based accelerator design, we evaluate our generated designs through RTL simulations. Regarding the design space exploration, we use *Algorithm* 1 to perform the accelerator design optimization, considering the architecture/dataflow search space and the design factors in Table 1 for the IP/dataflow design.

Visualizing the *Chip Builder*'s Two-stage DSE Process. For demonstrating the effectiveness of the *Chip Builder*'s two-stage DSE engine, here we visualize the DSE process, when using *AutoDNNchip* to design an FPGA-based accelerator for achieving competitive performance as the award winning state-of-the-art design in [31] given the same target performance specification/constraint, FPGA board, DNN model, and dataset. The FPGA measured energy consumption of both the resulting design from the *AutoDNNchip* and the reported one of [31] are marked in purple in Fig. 11. It demonstrates that: (1) the DSE engine of the *Chip Builder* can effectively trim down the design choices and generate optimized designs with better performance compared to the state-of-the-art design published in [31]. **Without humans in the loop, the *AutoDNNchip* can indeed automatically generate DNN accelerators that achieve optimized performance**; (2) most of the design choices can be efficiently ruled out by the 1st stage of the DSE engine, i.e., the early stage exploration based on the *Chip Predictor*'s coarse-grained analytical performance estimation; and (3) the 2nd stage IP-pipeline co-optimization of the *Chip Builder* can effectively boost (up to 36.46% improvement and an average of 28.92% improvement) the performance, i.e., throughput of the DNN accelerators here, as compared to that of the designs resulted from the 1st stage DSE. The final generated design candidates in the HLS code format will be passed to Vivado [22] for implementation. Then, we eliminate the designs that fail in the PnR step as shown in Fig. 11 and find an optimal design from the remaining ones. As a reference point, the 1st stage DSE takes about 0.65 ms for each design point and only 0.8 hour for exploring a total of 4.6 million design points when running on an Intel Core i5 CPU with a single thread, thanks to the analytical nature of the *Chip Predictor*.

Table 9: The considered application-driven specifications (i.e., throughput requirement) and constraints (i.e., power and resource budget) when evaluating the *Chip Builder*'s generated FPGA- and ASIC-based DNN accelerators.

Target Back-end	Application	Opt. Obj.	Th./P. Req.	Res. Budget
Ultra96 FPGA	Object Detection	E, L	20FPS 10W	DSP=360, FF=141120 LUT=70560, BRAM=432
ASIC	Vision Tasks [20]	E, L	15FPS 600mW	On-chip SRAM=128KB # of MAC units=64

Evaluation of the *Chip Builder*'s 2nd-stage Optimization. Fig. 12 summarizes the evaluation experiments for the *Chip Builder*'s 2nd-stage Optimization process. As described in Section 6, this stage targets an IP-pipeline co-optimization and thus can lead to more balanced pipeline and more efficient resource allocation. From Fig. 12, we can see that the *Chip Builder*'s 2nd-stage optimization can achieve up to 2.4× idle cycles reduction, when optimizing the design of SkyNet's 6 blocks [31] on the Ultra96 edge FPGA board [37].

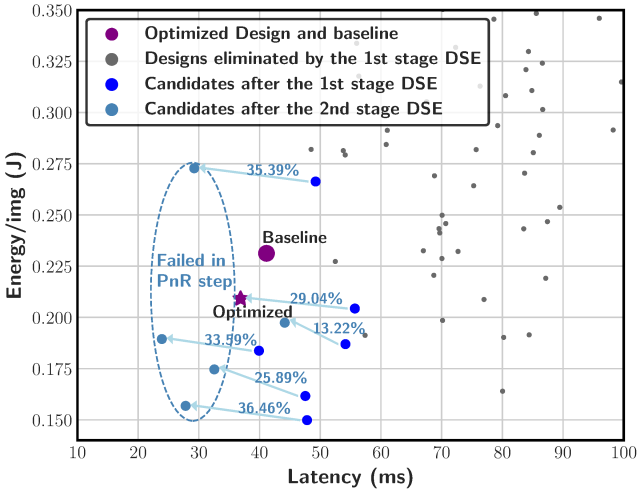


Figure 11: Visualizing the energy consumption per image and processing latency of the resulting designs from the *Chip Builder*'s 1st and 2nd stage optimization, when using *AutoDNNchip* to design an FPGA-based accelerator for meeting the performance of a state-of-the-art design [31] given the same performance specification/constraint, FPGA board, DNN model, and dataset (see Table 9).

Evaluation of *AutoDNNchip*'s Generated FPGA-based Accelerators. Fig. 11 shows that the DNN accelerator which is generated by *AutoDNNchip* can apparently outperform the recent award-winning design [31]. We further conduct another set of experiments to compare the performance of *AutoDNNchip*'s generated DNN accelerators on the Ultra96 FPGA board with that of a mobile CPU (Pixel2 XL [32]), when both designs (1) adopt the settings in Table 3,

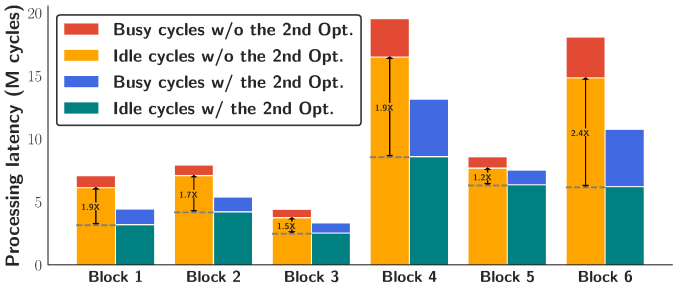


Figure 12: The busy and idle cycles of the bottleneck IPs in SkyNet's 6 different blocks, before and after conducting the *Chip Builder*'s 2nd-stage IP-pipeline co-optimization, when using the *AutoDNNchip* to generate designs for the Ultra96 FPGA board with the same target performance as [31].

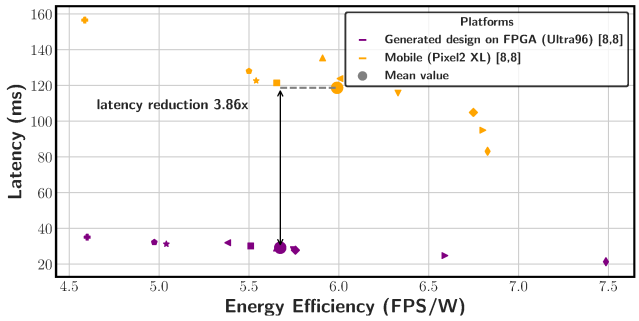


Figure 13: Processing latency and energy efficiency on Ultra96 FPGA compared with a mobile device (Pixel2 XL) on 10 compact DNN models

using the same bit precision and the 10 DNN models in Table 4, and (2) try to minimize the latency for considering time-critical applications. Note that the DNN mapping to the mobile CPU is optimized using Tensorflow Lite [44]. Fig. 13 illustrates the corresponding latency vs. energy efficiency, where the results under the same DNN models are marked with makers of the same shape. We can see that *AutoDNNchip* generated accelerators consistently achieve smaller latency than the baselines under the same DNN model and settings while having similar (<15% difference) energy efficiency. Specifically, *AutoDNNchip* generated accelerators achieve an average latency reduction of 3.86× while having slightly better (10%) or worse (differs <15%) energy efficiency, demonstrating the effectiveness of *AutoDNNchip* in generating optimized FPGA-based accelerators.

Evaluation of *AutoDNNchip*'s Generated ASIC-based Accelerators. Fig.14 illustrates that *AutoDNNchip* indeed can generate ASIC-based accelerator that leads to an optimal tradeoff between latency and energy consumption by visualizing the latency vs. energy consumption of the generated accelerators, where dots with different colors correspond to designs based on different hardware templates. Furthermore, we evaluate the performance of *AutoDNNchip* generated ASIC-based accelerators by comparing their energy consumption with that of a state-of-the-art ASIC-based accelerator [20] based on 5 shallow neural networks, which are used in [20] for performance evaluation, given both having the same throughput constraint as shown in Table 9. Fig. 15 shows the comparison, where all the energy consumption in Fig.15 are obtained from RTL implementation and simulation. We can see that *AutoDNNchip* generated ASIC-based accelerators consistently outperform [20] in all the 5 networks with energy consumption improvement ranging from 7.9% to 58.3%, demonstrating the effectiveness of *AutoDNNchip* in generating optimized ASIC-based accelerators.

For the aforementioned set of experiments, We first use the application-driven performance and constraints (see Table 9) to perform design space exploration and then validate the generated designs using RTL simulations adopting the same clock frequency (1 GHz) and technology (65nm) as our baseline [20]. Specifically, the DSE process optimizes the accelerators' energy-delay-product, and considering different: (1) hardware templates with three different architectures [17, 20, 21] (denoted as template 1/2/3 in Fig. 14), (2) memory size and # of PEs within the resource constraint, (3) memory allocation (i.e., input/weight/output buffer), and (4) memory accesses and reuse patterns.

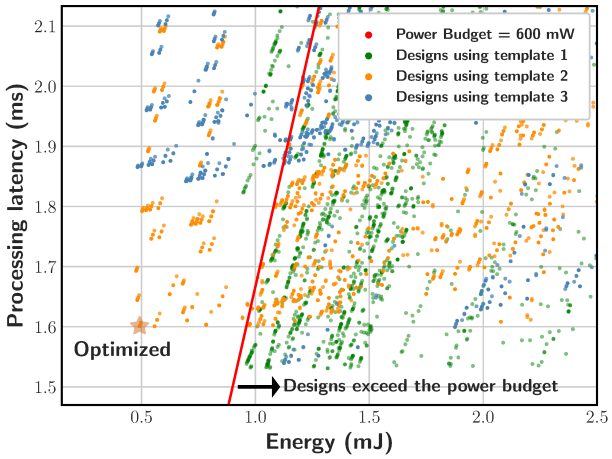


Figure 14: Visualizing the latency vs. energy consumption per image of the ASIC-based accelerators in the design space pool, when using *AutoDNNchip* to design an ASIC-based accelerator for meeting the performance of a state-of-the-art ASIC-based accelerator [20], with both having the same performance constraints, DNN model, and dataset (see Table 9).

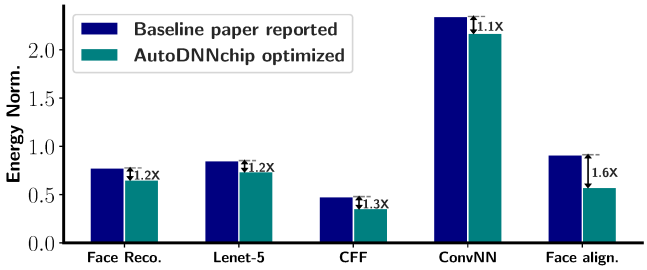


Figure 15: Comparing the normalized energy consumption between the *AutoDNNchip* generated ASIC-based accelerators and [20], when accelerating 5 shallow neural networks under the same throughput requirement.

8 CONCLUSIONS

To close the gap between the growing demand for DNN accelerators with various specifications and the time consuming and challenging DNN accelerator design, we develop *AutoDNNchip* which can automatically generate both FPGA- and ASIC-based DNN accelerators. Experiments using over 20 DNN models and 4 platforms show that DNN accelerators generated by *AutoDNNchip* outperform state-of-the-art designs by up to 3.86×. Specifically, *AutoDNNchip* is made possible by the proposed *one-for-all design space description*, *Chip Predictor*, and *Chip Builder*. Experiments based on 15 DNN models and 4 platforms demonstrate that the *Chip Predictor*'s prediction error is smaller than 10% compared with real-measured ones, and the *Chip Builder* can effectively and efficiently perform design space exploration and optimization.

ACKNOWLEDGMENTS

This work is supported in part by the NSF RTML grant 1937592 and NSF 1801865, the IBM-Illinois Center for Cognitive Computing System Research (C3SR), and XMotors.ai.

REFERENCES

[1] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR*, vol. abs/1409.1556, 2014.

[2] Y. Wang, Z. Jiang, X. Chen, P. Xu, Y. Zhao, Y. Lin, and Z. Wang, "E2-Train: Training State-of-the-art CNNs with Over 80% Energy Savings," in *Advances in Neural Information Processing Systems*, pp. 5139–5151, 2019.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, pp. 91–99, 2015.

[4] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig, "The Microsoft 2016 conversational speech recognition system," in *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, pp. 5255–5259, IEEE, 2017.

[5] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 389–400, ACM, 2018.

[6] Y. Wang, T. Nguyen, Y. Zhao, Z. Wang, Y. Lin, and R. Baraniuk, "Energynet: Energy-efficient dynamic inference," in *Advances in Neural Information Processing Systems (Workshop)*, 2018.

[7] J. Shen, Y. Fu, Y. Wang, P. Xu, Z. Wang, and Y. Lin, "Fractional Skipping: Towards Finer-Grained Dynamic Inference," in *The Thirty-Forth AAAI Conference on Artificial Intelligence*, 2020.

[8] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-Means: Re-Training and Parameter Sharing with Harder Cluster Assignments for Compressing Deep Convolutions," in *Thirty-fifth International Conference on Machine Learning*, 2018.

[9] Y. Wang, J. Shen, T.-K. Hu, P. Xu, T. Nguyen, R. Baraniuk, Z. Wang, and Y. Lin, "Dual dynamic inference: Enabling more efficient, adaptive and controllable deep inference," *IEEE Journal of Selected Topics in Signal Processing*, 2019.

[10] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[11] Y. Lin, S. Zhang, and N. Shanbhag, "Variation-Tolerant Architectures for Convolutional Neural Networks in the Near Threshold Voltage Regime," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, pp. 17–22, Oct 2016.

[12] S. Liu, A. Papakonstantinou, H. Wang, and D. Chen, "Real-time object tracking system on FPGAs," in *2011 Symposium on Application Accelerators in High-Performance Computing*, pp. 1–7, IEEE, 2011.

[13] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, "Throughput-optimized FPGA accelerator for deep convolutional neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 3, p. 17, 2017.

[14] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, IEEE, 2017.

[15] C. Zhuge, X. Liu, X. Zhang, S. Gummadi, J. Xiong, and D. Chen, "Face recognition with hybrid efficient convolution algorithms on FPGAs," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pp. 123–128, ACM, 2018.

[16] 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 *Proceedings of the International Conference on Computer-Aided Design*, p. 56, ACM, 2018.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, IEEE, 2017.

[18] Google Inc., "Edge TPU." https://coral.withgoogle.com/docs/edgetpu/faq/, accessed 2019-09-01.

[19] Y. Lin and J. R. Cavallaro, "Energy-efficient convolutional neural networks via statistical error compensated near threshold computing," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2018.

[20] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[21] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43th Annual International Symposium on*, pp. 367–379, IEEE Press, 2016.

[22] Xinlinx, "Vivado High-Level Synthesis." https://https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, accessed 2019-09-16.

[23] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xpilot: A platform-based behavioral synthesis system," *SRC TechCon*, vol. 5, 2005.

[24] D. Chen, J. Cong, Y. Fan, and L. Wan, "Lopass: A low-power architectural synthesis system for FPGAs with interconnect estimation and optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 4, pp. 564–577, 2009.

[25] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *2011 International Conference on Field-Programmable Technology*, pp. 1–8, IEEE, 2011.

[26] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family," in *Proceedings of the 53rd Annual Design Automation Conference*, p. 110, ACM, 2016.

[27] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[28] 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)*, pp. 152–159, IEEE, 2017.

[29] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, *et al.*, "MAGNet: A Modular Accelerator Generator for Neural Networks," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2019.

[30] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[31] X. Zhang, H. Lu, C. Hao, J. Li, B. Cheng, Y. Li, K. Rupnow, J. Xiong, T. Huang, H. Shi, *et al.*, "SkyNet: a Hardware-Efficient Method for Object Detection and Tracking on Embedded Systems," *arXiv preprint arXiv:1909.09709*, 2019.

[32] Google Inc., "Pixel Phone 2 XL." https://store.google.com/product/pixel_3?srp=/product/pixel_2/, accessed 2019-09-01.

[33] 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 Design Automation Conference*, p. 206, ACM, 2019.

[34] H. Kwon, M. Pellauer, and T. Krishna, "MAESTRO: an open-source infrastructure for modeling dataflows within deep learning accelerators," *arXiv preprint arXiv:1805.02566*, 2018.

[35] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315, IEEE, 2019.

[36] 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*, pp. 770–778, 2016.

[37] Xilinx Inc., "Avnet Ultra96." https://www.xilinx.com/products/boards-and-kits/1-vad4rl.html, accessed 2019-09-01.

[38] NVIDIA Inc., "NVIDIA Jetson TX2." https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/, accessed 2019-09-01.

[39] J. Deng, W. Dong, R. Socher, L. jia Li, K. Li, and L. Fei-fei, "Imagenet: A large-scale hierarchical image database," in *In CVPR*, 2009.

[40] J. Hu, J. Goeders, P. Brisk, Y. Wang, G. Luo, and B. Yu, "2019 DAC system design contest on low power object detection," *When Accuracy meets Power: 2019 DAC System Design Contest on Low Power Object Detection*, 2019.

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

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

[43] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[44] Google Inc., "Tensorflow Lite." https://www.tensorflow.org/lite, accessed 2019-09-01.