Reusing GEMM Hardware for Efficient Execution of Depthwise Separable Convolution on ASIC-based DNN Accelerators

Susmita Dey Manasi
University of Minnesota Twin Cities
Minneapolis, MN, USA

Suvadeep Banerjee
Intel Labs
Santa Clara, CA, USA

Abhijit Davare
Intel Labs
Hillsboro, OR, USA

Anton A. Sorokin
Intel Labs
Hillsboro, OR, USA

Steven M. Burns
Intel Labs
Hillsboro, OR, USA

Desmond A. Kirkpatrick
Intel Labs
Hillsboro, OR, USA

Sachin S. Sapatnekar
University of Minnesota Twin Cities
Minneapolis, MN, USA

ABSTRACT

Deep learning (DL) accelerators are optimized for standard convolution. However, lightweight convolutional neural networks (CNNs) use depthwise convolution (DwC) in key layers, and the structural difference between DwC and standard convolution leads to significant performance bottleneck in executing lightweight CNNs on such platforms. This work reuses the fast general matrix-vector multiplication (GEMM) core of DL accelerators by mapping DwC to channel-wise parallel matrix-vector multiplications. An analytical framework is developed to guide pre-RTL hardware choices, and new hardware modules and software support are developed for end-to-end evaluation of the solution. This GEMM-based DwC execution strategy offers substantial performance gains for lightweight CNNs: 7× speedup and 1.8× lower off-chip communication for MobileNet-v1 over a conventional DL accelerator, and 74× speedup over a CPU, and even 1.4× speedup over a power-hungry GPU.

CCS CONCEPTS

• **Hardware → Application specific integrated circuits**; • **Computing methodologies → Neural networks**.

KEYWORDS

depthwise convolution, lightweight CNN, deep learning accelerator

ACM Reference Format:
Susmita Dey Manasi, Suvadeep Banerjee, Abhijit Davare, Anton A. Sorokin, Steven M. Burns, Desmond A. Kirkpatrick, and Sachin S. Sapatnekar. 2023. Reusing GEMM Hardware for Efficient Execution of Depthwise Separable Convolution on ASIC-based DNN Accelerators. In *28th Asia and South Pacific Design Automation Conference (ASPDAC '23), January 16–19, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3566097.3567863

ASPDAC '23, January 16–19, 2023, Tokyo, Japan

ACM ISBN 978-1-4503-9783-4/23/01…$15.00
https://doi.org/10.1145/3566097.3567863

1 INTRODUCTION

Lightweight versions (e.g., MobileNets [1], Xception [2], EfficientNet [3], MNasNet [4], ShuffleNet [5]) of deep convolutional/deep neural networks (CNNs/DNNs) offer competitive accuracy in a wide range of vision applications, with much lower parameter counts and computational requirements than memory- and compute-intensive CNNs such as ResNet, GoogleNet, and VGGNet.

The depthwise convolution (DwC) layer is key to enabling these lightweight CNNs as it requires orders-of-magnitude lower parameter counts and multiply-accumulate (MAC) operations as compared to standard convolution (Conv2D) layers [1]. However, DwC offers limited scope for data reuse and parallelism and maps poorly to general ASIC-based DNN accelerators [6–9], which are optimized for Conv2D and do not directly support DwC operations. This poses a major bottleneck [5, 9, 10] in executing lightweight CNNs on such platforms. For example, Gemmini [9] executes DwC layers in a host CPU; VTA [7] replaces DwC with grouped convolutions; an enhanced version of VTA [11] executes DwC layers in a generic ALU core; in [12] numerous wasteful zeros are inserted to transform DwC into Conv2D. Accelerators for lightweight CNNs use separate compute engines for Conv2D and DwC [13, 14], resulting in high hardware resource requirements. While [15] reuses the same compute engine for Conv2D and DwC, it supports only a subset of Conv2D for lightweight CNNs. X-Layer [16] uses cross-layer fusion and heterogeneous dataflow for DwC, but requires frequent on-chip tensor formatting to handle inter-layer communication, leading to large hardware overheads. A customized network-on-chip (NoC) and dataflow specific to the Eyeriss v2 [17] architecture accelerates Conv2D and DwC, but does not generalize to mainstream accelerators with systolic or vector dot-product computation in a general matrix-vector multiplication (GEMM)-based array [6–9, 11].

Contributions: We develop a methodology that performs DwC and Conv2D (including fully connected (FC) layers) on the same hardware. The DwC computation is algorithmically manipulated as a channel-wise parallel matrix-vector multiplication and mapped to reuse the fast GEMM core (optimized for Conv2D) of a DNN engine. We develop hardware modules, and provide instruction-level support and testbenches, for end-to-end functional verification and evaluation of our GEMM-based strategy. Our solution is both *simple* and *practical*: through small hardware changes and instruction set

modifications that can easily be adapted in standard GEMM-based DL accelerators, we deliver large improvements in accelerator performance. Note that code-based modulation using front-end DNN development frameworks such as PyTorch or TensorFlow cannot achieve this solution for ASIC accelerator platforms.

We evaluate our DwC implementation in the GEMM core on a full hardware stack performing back-end synthesis, place, and route in an Intel 22FFL technology. The **features of our GEMM-based DwC execution method** are: *(i)* we *reuse existing hardware* resources; *(ii)* we incur *very small* supplementary hardware cost to integrate it in a GEMM core: we augment the GEMM core with image-to-column (Im2Col) hardware and multiplexers to enable the computation of DwC along with Conv2D; *(iii)* we *provide instruction-level support* for the new GEMM-based DwC operation; and *(iv)* we *substantially accelerate* the DwC computation, making it a small fraction of the network runtime (shown later in Fig. 9(a)).

Compared to an ALU-based DwC implementation, we demonstrate that our approach provides *end-to-end improvements* in MobileNet-v1 execution. For various GEMM array sizes, we achieve

- **5.10**× speedup, **1.65**× fewer off-chip accesses on a 32×32 array.
- **7.02**× speedup, **1.81**× fewer off-chip accesses on a 64×64 array.

We compare our GEMM-based strategy with CPU and GPU-based implementations of MobileNet. We obtain **73.9**× speed-up over an Intel(R) Xeon(R) Gold CPU. We also demonstrate **1.4**× speed-up over a high-performance NVIDIA Tesla V100S-PCI GPU, even though the GPU has ~16× more on-chip memory and is significantly more power-hungry than our custom accelerator solution. As compared to Eyeriss v2 [17], a heavily customized lightweight CNN accelerator, our approach achieves 1.95× performance improvement on a variant of MobileNet.

Table 1: (left) Operations per layer type for MobileNet-v1, and (right) percentage of cycles spent in executing DwC and other layers with respect to total network cycles on two different hardware configurations.

Layer	#of MAC	Parameter	ASIC accelerator	DwC cycles	Conv2D + Other cycles
Conv2D	96.05%	74.61%	Hardware1	60.28%	39.72%
DwC	3.06%	1.06%	Hardware2	62.69%	37.31%
FC	0.18%	24.33%			

2 THE VALUE OF ACCELERATING DWC

We analyze the impact of DwC layers on the runtime of MobileNet-v1 [1], a representative lightweight CNN topology that is widely used in the machine learning (ML) community. In MobileNet, approximately every other layer is a DwC. Although the DwC layers together consist of just 3.06% of the total number of MAC operations and just 1.06% of the DNN parameters (see left part of Table 1), DwC computations constitute a significant performance bottleneck in the end-to-end runtime on general ASIC-based DNN accelerators [6–8, 11]. These platforms contain two key compute cores:
(i) a multiplier-rich **GEMM core** for Conv2D and FC layers,
(ii) a generic **ALU core** to execute activation, pooling, DwC, etc.
The right part of Table 1 shows the fraction of total network cycles spent in executing DwC layers on two different hardware configurations of a general DNN accelerator (see Fig. 2 for details) where DwC is executed in the ALU core by mapping the computation to *multiplication* and *addition* operations. The data is obtained using our analytical model from Section 5.1. It is clear that despite having a small number of MAC operations and low parameter counts, DwC layers present significant performance bottleneck (over 60% of the network runtime) and must be accelerated.

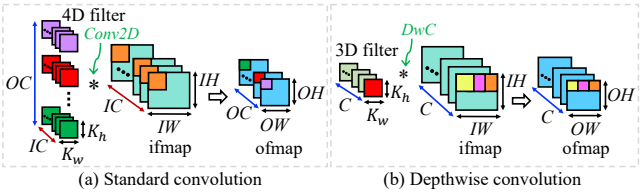

Figure 1: Standard convolution vs. depthwise convolution.

General ASIC-based DL accelerators can evolve with the growing variety of neural network topologies and have a mature compilation stack (i.e., TVM [18], XLA [19]) for end-to-end DNN execution, but are inefficient for DwC. Our economical and efficient DwC solution reuses the GEMM core of these accelerators, which is hardware-resource-rich and highly optimized for compute-intensive Conv2D.

3 PRELIMINARIES

Conv2D vs. DwC: Depthwise separable convolution (DwSC) is a key building block of lightweight CNNs and consists of two types of layers: *(i)* pointwise convolution, a special case of Conv2D where the kernel size is 1×1, and *(ii)* depthwise convolution (DwC).

Fig. 1 shows differences between DwC and Conv2D. In Conv2D (Fig. 1(a)), a ($K_h \times K_w \times IC \times OC$) 4D **filter** is convolved with an ($IH \times IW \times IC$) 3D input feature map (**ifmap**) to produce an ($OH \times OW \times OC$) output feature map (**ofmap**). This involves element-wise multiplication between each 3D filter channel and an ifmap region (i.e., the purple filter and orange ifmap region). The intermediate partial sums (**psums**) are accumulated across all IC channels to produce one ofmap channel element (purple ofmap box). The operation is repeated, using all OC 3D filters, to produce data in all ofmap channels. The full ofmap is built by repeating the process, sliding the filter through the ifmap with a convolution stride S. DNN accelerators execute Conv2D as a GEMM operation.

DwC (Fig. 1(b)) uses a ($K_h \times K_w \times C$) 3D filter, with an identical number, C, of filter/ifmap/ofmap channels. Element-wise MAC operations between each filter channel (red) and a same-sized sub-region of the ifmap channel (yellow) produce data in the ofmap channel (yellow). With a filter stride of S, the full ofmap is produced.

If $IC = OC = C$, DwC reduces the parameter counts and #of MAC operations by a factor of C over Conv2D [1]. Unlike Conv2D, DwC performs no summation across channels and there is only one 3D filter. It is challenging to efficiently execute DwC layers in DNN accelerators because (a) the DwC computation cannot be readily mapped to GEMM (b) there is limited available parallelism.
Target hardware platform: Our target hardware platform, represented in Fig. 2, is a general ASIC-based DNN inference accelerator, similar to TPU [6], VTA [7, 11], and GeneSys [8], that is primarily optimized for standard convolution. The **GEMM core** has a $J \times K$ **array** of processing elements (PEs). In every cycle, each column of the GEMM core performs element-wise MAC operations between two $1 \times J$ vectors and produces a single output. The K columns operate in parallel, producing K outputs in each clock cycle. A single $1 \times J$ ifmap vector is shared among all K columns (i.e., the ifmap vector is reused horizontally) while the PE array receives a

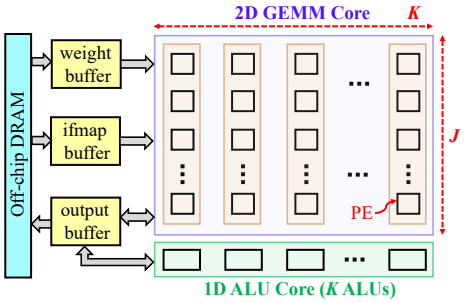

Figure 2: Block diagram of the hardware architecture

$J \times K$ submatrix of filter weights. The ifmap-weight matrix operation produces K psum/ofmap data in each cycle where psums are reduced vertically within each column. The GEMM core has access to three separate (filter, ifmap, psum/ofmap) on-chip buffers that communicate with off-chip DRAM. The **ALU core** is a $1 \times K$ **ALU array.** The K ALUs parallelly perform a single type of operation (×, +, min, max, shift, etc.). The ALU core has access to only the output buffer.

Our GEMM-based strategy for DwC is applicable to generic ASIC-based DNN accelerators. We demonstrate a full-stack evaluation on an open-source [11] VTA$^+$ engine, integrated with Apache TVM.

4 METHODOLOGY FOR GEMM-BASED DWC

4.1 Mapping Conv2D to GEMM

We illustrate how Conv2D is mapped to GEMM in Fig. 3(a). Let a_{ij}^k [o_{ij}^k] denote ifmap [ofmap/psum] element from the k^{th} channel. For the 4D filter, $w_{ij}^{k,fn}$ is the element from the k^{th} channel of the n^{th} 3D filter. The $1 \times J$ ifmap vector uses one element from each of J ifmap channels. The $J \times K$ weight matrix consists of data from K 3D filters; each matrix column uses data from J channels of one 3D filter. The ifmap-weight operation produces a $1 \times K$ vector of ofmap/psum (one element in each of K ofmap channels).

Fig. 3(b) shows the translation of this mapping to the $J \times K$ PE array of the GEMM core, where *IC-k* and *OC-n* indicate the k^{th} ifmap/filter channel and n^{th} ofmap channel/3D filter, respectively. The ifmap vector is shared by the horizontal lines across all PE array columns. Each PE receives one element of the weight matrix where one column of the PE array operates on a single 3D filter. The GEMM operation is then performed in one cycle. Similar operations are repeated in the array until the full Conv2D layer is computed. The Conv2D computation as GEMM illustrated in Fig. 3 is applicable for Conv2D layer with any kernel size (i.e., $K_h, K_w \geq 1$).

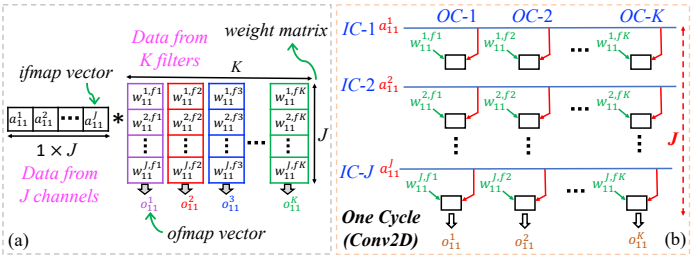

Figure 3: (a) Formulation of Conv2D as GEMM. (b) Illustration of mapping Conv2D in the $J \times K$ PE array of the GEMM core.

4.2 Mapping DwC to GEMM

Elementwise MAC operations in DwC are computed on a per-channel basis to compute ofmap channel elements using a single 3D filter. GEMM-based Conv2D cannot be readily extended for DwC because:

- DwC inherently offers no ifmap data reuse, and so it is not possible to share an ifmap vector across the array columns.
- No weight matrix can be formed: there is only one 3D filter.

We algorithmically map DwC to GEMM hardware, design supporting hardware modules, and devise a hardware mapping strategy.

Algorithmic formulation: We view the computation in each channel of a DwC layer as multiplication between a *matrix of ifmap data* and a *vector of filter data.* For a single channel, Fig. 4(a)–(b) shows a stride-1 DwC operation between a $K_h \times K_w$=3×3 filter channel and an $IH \times IW$=5×5 ifmap channel that produces an $OH \times OW$=3×3 ofmap channel. For DwC, a_{ij}^k, o_{ij}^k, and w_{ij}^k denote, respectively, individual ifmap, ofmap, and filter elements from the k^{th} channel.

In Fig. 4(b), a vector of length $(K_h \times K_w)$ is formed from the 2D filter kernel. The ifmap matrix is of dimension $(K_h \times K_w) \times (OH \times OW)$, where each matrix column comes from a 2D spatial window of ifmap data convolved with the filter kernel, e.g., the red window in the ifmap channel is vectorized to form the red column of the ifmap matrix. Other columns are shown by the color-codes on the ifmap channel and the ifmap matrix. The ifmap-filter matrix-vector computation produces an $(OH \times OW)$ ofmap vector using GEMM, which is applied to all channels to compute DwC channel-wise.

Mapping in the GEMM hardware: Fig. 4(c) shows the computation in a single PE array column, over multiple cycles, to translate the channel-wise GEMM mapping of DwC (Fig. 4(b)) to the GEMM core. PE array columns operate on separate channels in parallel. The complete mapping of DwC in the 2D PE array, per cycle, is shown in Fig. 4(d); *C-k* denotes the k^{th} channel. Each of the K columns of the GEMM core is equipped with an *Im2Col module* that sequences data to perform channel-wise parallel matrix-vector multiplication.

The PE array normally receives ifmap data directly from ifmap buffer in the natural array sequence for Conv2D GEMM; for DwC, data from the ifmap buffer must be rearranged in the Im2Col modules (e.g., in the first window, a_{31}^1 is needed after a_{23}^1 [Fig. 4(a)]). In each cycle, Im2Col supplies a column of the ifmap matrix while the filter vector is reused over multiple cycles in a weight-stationary dataflow.

Horizontal lines that share ifmap data across PE columns for Conv2D are disabled. Simple multiplexers are used to switch the GEMM core between Conv2D and DwC. The PE array uses the same datapaths for both modes to receive filter data from the weight buffer and to write ofmap/psum data to the output buffer.

Our DwC mapping strategy requires a PE array size where $J \geq (K_h \times K_w)$. Standard lightweight CNNs [1–3, 5] consist of DwC layers with 3×3 kernels. Therefore, the above constraint satisfies for all reasonable sized PE array ($J \geq 9$) sizes in DNN accelerators.

Since the maximum utilization for DwC operation per PE array column is bounded by $(K_h \times K_w)$, $J - (K_h \times K_w)$ PEs may remain unused in each PE array column during the computation of DwC. These are disabled by inducing zero valued weights: note that there

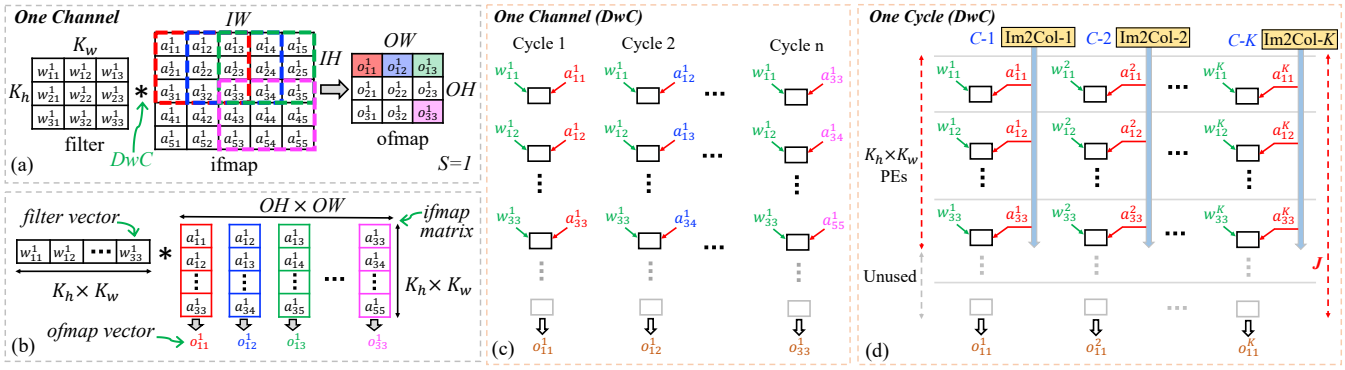

Figure 4: (a)–(b) DwC as matrix-vector multiplication. (c)-(d) Mapping of DwC computation in the 2D PE array of the GEMM core: (c) mapping in one column of the PE array over multiple cycles, (d) mapping in the $J \times K$ PE array over a single cycle.

are no such unused PEs for Conv2D. As we will show in Section 5.1, the cost of additional hardware to keep all PEs busy is not justified by the small gains in the overall network runtime.

Im2Col module: The data in each column of the ifmap matrix in Fig. 4(b) are not consecutive elements in the ifmap channel. Since the ifmap data (ofmap from the previous layer) of a CNN is generated and stored at runtime, it is not possible to change data layout to obtain the desired sequencing. The Im2Col module creates the ifmap matrix from a 2D ifmap channel at runtime, producing vectorized data from a 2D spatial window of an ifmap channel (i.e., one column of the ifmap matrix) every cycle.

We have developed a line buffer and window buffer-based [20] Im2Col module. Fig. 5 shows our Im2Col hardware module for stride-one DwC. The module consists of $(K_h - 1)$ line buffers that store $(K_h - 1)$ rows of the operating ifmap channel at a time and right shift each data element every cycle. We design each line buffer using a dual-port SRAM that can read/write data independently in each cycle. The SRAM read/write addresses are generated to index the SRAM cells sequentially and create a shift register functionality.

A single Im2Col module receives a single ifmap element every cycle from the ifmap buffer. After initial stall cycles when the GEMM core waits for Im2Col buffer lines to be filled, the module produces one column of the ifmap matrix per cycle. The interface between the ifmap buffer, and all K Im2Col modules together, transfers 1D ifmap data (i.e., $1 \times K$) per cycle, similar to the data rate between the ifmap buffer and PE array for Conv2D.

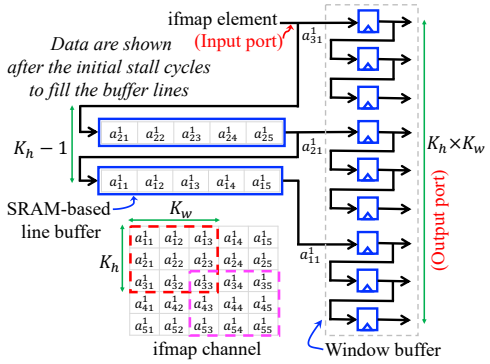

Figure 5: Im2Col hardware, creating vectorized data from a $K_h \times K_w$ ifmap window per cycle (dotted rectangles show example windows).

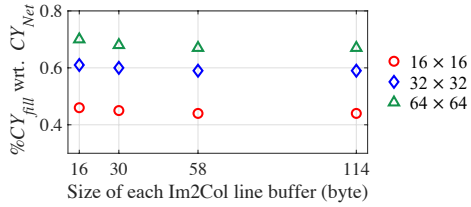

Figure 6: Percentage of stall cycles to fill the Im2Col buffer lines wrt. total computation cycles of MobileNet-v1.

5 END-TO-END IMPLEMENTATION FLOW

We first develop an analytical performance model that guides our hardware design choices. We alter the baseline VTA$^+$ hardware engine and its instruction set to provide instruction-level support for GEMM-based DwC. Finally, we outline our efforts on RTL hardware implementation and end-to-end functional verification.

5.1 Analytical Model-Guided Design Decisions

In this section, we perform a cost/benefit analysis for optimizations that superficially appear promising for our GEMM-based strategy.

This process is guided by our analytical framework for the performance of a lightweight CNN. We model the execution of *(i)* Conv2D and FC in the GEMM core, *(ii)* our GEMM-based or traditional ALU-based execution of DwC *(iii)* ReLU, bias addition, shift, min, and pooling in the ALU core. The framework takes the layer shape and the hardware configuration (i.e., GEMM/ALU dimensions, on-chip buffer sizes/bandwidths, Im2Col line buffer size, data bitwidths) as inputs and estimates the number of computation cycles. For this first-order estimate, our analytical prediction model does not consider stall cycles due to data communication with the off-chip DRAM. However, we model the stall cycles while the GEMM core waits for the appropriate ifmap data from the Im2Col modules. These assumptions are adequate to capture trends and guide the following pre-RTL design decisions to identify key hardware design trade-offs for our Im2Col-augmented GEMM core:

(1) Should Im2Col-induced stalls be hidden? Fig. 6 shows the percentage of stall cycles to fill the Im2Col buffer lines (CY_{fill}) with respect to the total compute cycles (CY_{Net}) for MobileNet-v1. For various size of the Im2Col buffer lines for three different hardware configurations (i.e., $J \times K$ = 16×16, 32×32, 64×64), the number of stall cycles to fill the Im2Col line buffers is a small (<0.8%) fraction

Table 2: Comparison of network performance on three different engines for MobileNet-v1 to analyze $S2$ Im2Col hardware.

Engine specifications*1		E1	E2	E3
ifmap data rate per GEMM column (bits/cycle)	stride 1	8	8	4
	stride 2	32	8	4

Hardware configurations*2			Gain of E1 wrt. E2	Gain of E1 wrt. E3
$J \times K$	ifmap buffer	output buffer		
16×16	32 kB	64 kB	2.09%	7.24%
32×32	64 kB	128 kB	2.80%	9.54%
64×64	128 kB	256 kB	3.05%	11.10%

*1 *bitwidth of ifmap = 8 bit;* *2 *Size of each Im2Col line buffer = 58 byte*

of the total network cycles. Therefore, *we do not add hardware* (e.g., double-buffering Im2Col lines) to hide these stall cycles.

(2) Should dedicated stride-two Im2Col hardware be built? DwC layers in lightweight CNNs typically use stride-one ($S1$) or stride-two ($S2$). The design of an Im2Col module for $S2$ DwC is similar to the $S1$ hardware in Fig. 5, with modified connectivity among the SRAM lines and the shift registers in the window buffer. Between two consecutive windows in an ifmap channel along the height/width dimension, there is 2× less overlap of ifmap elements for $S2$ as compared to $S1$: the modified connectivity supports this lower overlap in the ifmap data. To avoid hardware overheads of implementing $S2$, we may reuse the $S1$ Im2Col hardware to compute $S2$ DwC, at the cost of stall cycles to handle invalid window buffer outputs (i.e., not a valid column of the ifmap matrix).

To quantitatively determine the penalty of this choice, we analyze three different engines (E1, E2, and E3). E1 (with extra hardware) uses the faster dedicated Im2Col module for $S2$ operation where a valid window is produced every cycle, while E2 and E3 use the $S1$ Im2Col hardware to execute DwC with both strides. $S2$ DwC operation in E2 [E3] is approximately 4× [8×] slower than E1.

However, the impact on total network cycles (Table 2) of E1 as compared to E2 and E3 for MobileNet-v1 is more muted: faster execution of $S2$ DwC layers at the expense of more hardware cost provides small (only 2-11%) performance gain. Achieving this small gain requires substantial hardware costs (MUXes, a second clock at half the frequency, and 4× higher memory bandwidth). Therefore, we use the $S1$ Im2Col hardware to execute DwC with both strides.

Table 3: Percentage of DwC cycles with respect to total network computation cycles for MobileNet-v1 on five different hardware configurations with increasing PE array size.

PE array size	16×16	32×32	64×64	128×128	256×256
DwC cycles	5.6%	7.5%	9.1%	11.4%	14.4%

(3) Should extra hardware be built to increase PE array utilization for DwC? As stated earlier, during the computation of our GEMM mapped DwC, some PEs may remain unused in the PE array. For example, for a 32×32 array, only 28% of the PEs are utilized during the execution of a DwC layer with a kernel size of 3×3. Under our GEMM-executed DwC where a subset of the PE array is unused, Table 3 shows the percentage of cycles for DwC computations with respect to the total network computation cycles for various PE array sizes for MobileNet-v1. DwC constitutes a small fraction (5–14%) of the network cycles across all array sizes. Even high-end DNN accelerators (e.g., TPUv2 and TPUv3 [19]) do not go above 128×128 due to high bandwidth requirements and low utilization. Thus, for realistic array sizes, the small potential performance gain by increasing utilization does not justify the high hardware overhead.

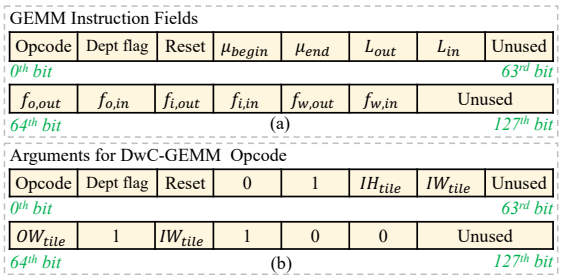

Figure 7: (a) The GEMM instruction fields (128 bit) [7]. (b) Instruction fields for the new DwC-GEMM opcode.

5.2 Defining a New GEMM Opcode

Due to the differences in mapping Conv2D and DwC to GEMM, a GEMM instruction defined to walk through the tensors and describe the data access patterns for Conv2D cannot be directly used for DwC. For the Im2Col-based DwC GEMM operation, we define a new GEMM opcode called DwC-GEMM, by modifying a generic GEMM instruction for Conv2D. We demonstrate this instruction-level support using the existing ISA of VTA$^+$. The ISA adopts two-level nested loops to define a deep learning operator (i.e., Conv2D) where the access indices of the tensor operands (i.e., weight, ifmap, and ofmap) in their respective buffers are computed via an affine function [7].

For the GEMM-based DwC, we select a new data layout for the filter data in memory. In the memory, the filter tensor of DwC layer is pre-stored using our data layout where K_w and K_h dimensions are kept in the innermost loops to enable consecutive access of the filter weight vectors required to perform parallel matrix-vector multiplication in the GEMM core (Fig. 4(d)). For ifmap/ofmap data, we use the same layout for Conv2D and DwC, ensuring data layout compatibility between two consecutive Conv2D and DwC layers.

Fig. 7(a) shows the fields of the VTA$^+$ GEMM instruction. L_{out} and L_{in} are the fields to contain the extent of the two nested loops (i.e., outer loop and inner loop, respectively). The fields $f_{\alpha,out}$ and $f_{\alpha,in}$ contain the scaling factors associated with the outer and inner loop, respectively, that are used in an affine function to compute the access indices for the data type α in its respective buffer. Here $\alpha \in \{o, i, w\}$ indicating ofmap, ifmap, and weight data, respectively. For a tensor operation, μ_{begin} and μ_{end} fields are used to unroll additional computation loops not covered by these two nested loops.

Instruction fields in our new DwC-GEMM opcode resemble those of GEMM. The opcode supports stride-1 and stride-2 DwC. Fig. 7(b) shows the arguments defined for the DwC-GEMM instruction fields. A subset of instruction fields are populated using tiled dimensions of ifmap and ofmap. These arguments are used by the opcode loops that describe the functionality of DwC-GEMM to compute appropriate indices to access the weight, ifmap, and ofmap data in their respective buffers. The Im2Col hardware produces invalid windows during the initial cycles when the Im2Col line buffers are filled, during the transition between ifmap rows, and between cycles during the computation of $S2$ DwC. The opcode definition skips the ofmap write to the output buffer to handle these invalid windows. Our new DwC-GEMM opcode uses instruction fields from the GEMM instruction, and is a fully compatible extension of the existing GEMM instruction. We verified the functionality of DwC-GEMM through C++ based behavioral hardware simulation.

5.3 Hardware Design and Verification

Hardware development: The required hardware modules for our GEMM-based execution of DwC were implemented using Chisel HDL, and parameterizable Chisel testbenches were developed to verify the functionality of each individual hardware unit.

The hardware has an array of Im2Col modules that parallelly operates on multiple 2D channels of a 3D ifmap tensor. The line buffers in the Im2Col modules are implemented using dual-port SRAMs. Due to the variation of data dimensions across layers within a network, the tile sizes of the ifmap tensor can vary from layer to layer, and the Im2Col hardware unit is designed to operate on different ifmap tile sizes across different layers of a network. An index generator module decodes the new DwC-GEMM opcode and generates appropriate indices to access the ifmap, weight, and ofmap buffers. The index generator module is equipped with appropriate valid signals to handle the invalid windows from the Im2Col modules.

The Im2Col and index generator modules were integrated with the GEMM core of VTA$^+$, with system-level integration of these hardware modules through additional control logic and pipeline stages to enable the evaluation on the full hardware stack of VTA$^+$.

System-level testbench: We developed a Python-based testbench that can run a full DwC layer using our GEMM-based execution strategy. The testbench tiles and schedules the DwC computation onto the accelerator hardware. The weight tensor is packed using our selected data layout to pre-store it in the off-chip DRAM. The software stack of VTA$^+$ is used to perform JIT (just-in-time) compilation of the instructions and produce the accelerator binaries. The testbench performs system-level functional verification of the end-to-end hardware stack against a numpy-based golden reference.

6 RESULTS

6.1 Evaluation of GEMM-based DwC

We evaluate our GEMM-based execution of DwC on VTA$^+$ [11], a general DNN accelerator platform used by ML developers across industry. We compare the GEMM-based DwC enabled VTA$^+$ against the baseline VTA$^+$ where DwC layers are executed in the ALU core. We also compare our approach against a modern Intel CPU, a high-performance NVIDIA GPU, and Eyeriss v2 [17], a lightweight CNN accelerator. CPU/GPU specifications are listed in Table 5.

As a representative lightweight CNN we use MobileNet-v1 that has both DwC and Conv2D layers. We use a cycle-accurate simulator of VTA$^+$ [21] to obtain the end-to-end performance metrics for a network (i.e., cycle counts and off-chip DRAM counts). While executing a workload, the simulator extracts performance metrics from the signal traces of the RTL implementation of the hardware. We report end-to-end cycle counts from the simulator that *include both computation cycles* and *stall cycles* due to off-chip DRAM accesses.

Comparison vs. conventional DwC: We use following notations:

(i) GEMM-executed DwC (**GED**), run in the GEMM core (our work).
(ii) ALU-executed DwC (**AED**), run in the ALU core (baseline).

In both cases, Conv2D and FC are executed in the GEMM core while operations that follow each layer such as ReLU, shift and min for ofmap bitwidth conversion, and bias addition are executed in the ALU core. For all evaluations, the hardware uses 8-bit ifmap/weight and 32-bit psum/ofmap data. GED Im2Col modules use 58 bytes per line buffer. Other specifications (i.e., array dimensions, SRAM sizes, and off-chip bandwidth) of GED and AED are provided throughout this section as each configuration is introduced.

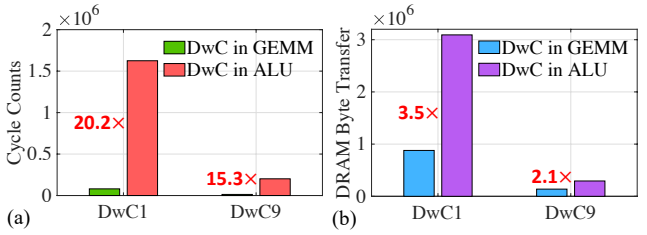

Figure 8: GED vs. AED performance for DwC1 and DwC9 layers of MobileNet-v1 on (a) cycle counts, (b) DRAM accesses.

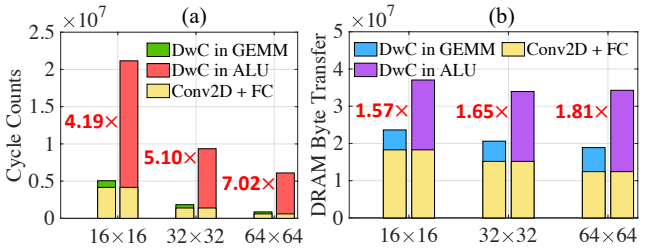

Figure 9: GED vs. AED overall performance for MobileNet-v1 on (a) end-to-end network runtime, (b) DRAM accesses.

Performance comparison: The cycle-accurate simulator shows that DwC in the GEMM core is very fast, and takes even less time than the cheap ALU operations (ReLU, shift, min, etc.) that follow each DwC tile. Fig. 8 shows the performance gain of GED vs. AED for the first (DwC1, a wide and thin layer) and last (DwC9, a narrow and fat layer) depthwise convolution layers of MobileNet on a $J \times K$ = 32×32 hardware configuration (H1). H1 uses 32kB, 32kB, and 128kB for the ifmap, weight, and output buffer sizes, respectively, and 256 bits/cycle off-chip bandwidth. GED offers 20.2× [15.3×] speed-up and 3.5× [2.1×] lower off-chip communication over AED for DwC1 [DwC9]. Similar benefits are seen for other DwC layers.

Fig. 9 shows the gain in the end-to-end network performance for MobileNet adopting GED as compared to AED. The evaluations are shown for H1 and for two more hardware configurations, H2 and H3. H2 uses $J \times K$ = 16×16, 16kB, 32kB, and 64kB for the ifmap, weight, and output buffer sizes, respectively, and 64 bits/cycle off-chip bandwidth. H3 uses $J \times K$ = 64×64, 64kB (ifmap), 64kB (weight), and 256kB (output) of buffer sizes with 512 bits/cycle off-chip bandwidth. Since VTA$^+$ executes the first Conv2D layer that takes the RGB input image in the host CPU and offloads the rest of the network to the accelerator, our evaluation in Fig. 9 excludes the first Conv2D layer. GED offers substantial speed-up in the network runtime as compared to AED, with speedups of 5.10× 4.19×, and 7.02× on H1, H2, and H3, respectively. GED is also more efficient than AED in terms of total off-chip DRAM access count, 1.65× better on H1, 1.57× better on H2, and 1.81× better on H3.

Area comparison: To determine the area cost of the supplementary hardware modules in GED, we synthesize, place and route (SP&R) both GED and AED using Intel 22FFL process technology. Table 4 summarizes the post-SP&R results for two hardware configurations, normalized to AED, both running at 1GHz clock. The area cost of the supplementary hardware modules to enable our GEMM-based execution of DwC is a small (4–6%) fraction of the accelerator area.

Table 4: Post-SP&R results for GED and AED (Intel 22FFL).

DwC in GEMM vs ALU	$J \times K$	Total on-chip buffer size	Normalized wrt. AED		
			Total area	Macro area	Hardware overhead for GED
AED	16×16	24 kB	1	0.67	6%
GED			1.06	0.70	
AED	32×32	96 kB	1	0.62	4%
GED			1.04	0.65	

Energy comparison: We evaluate the energy consumption of GED as compared to AED. We use the breakdown of various performance metrics provided by the simulator and post-SP&R power-performance characteristics of the hardware to obtain the energy metric. While executing a network, the performance breakdown obtained from the simulator are: #of active GEMM cycles, #of active ALU cycles, #of stall cycles, #of accesses for each of the on-chip buffers, and #of off-chip DRAM access. We combine data from the backend run (effective clock frequency, dynamic and leakage power of GEMM and ALU cores, and energy/access for each on-chip buffer) with simulator data to obtain the energy consumption of GED and AED. Energy numbers for GED (normalized to AED energy) are shown in Fig. 10, for MobileNet on H1 and H2. For both, GED consumes much lower energy than AED: 48.8% lower on H1 and 35.6% lower on H2. Although GED consists of additional hardware modules (i.e., Im2Col, multiplexer) to enable the execution of DwC in the GEMM core, they introduce low overhead and speed up GED substantially over AED, which leads to the reduction in overall energy consumption of GED.

Comparison with CPU and GPU: We evaluate the performance of GED by comparing it against a modern Intel CPU and a high-performance NVIDIA GPU. Table 5 summarizes the specifications for each hardware platform. We use ONNX Runtime, an industry-standard DL acceleration framework, to execute MobileNet-v1 on CPU and GPU. To measure the execution time on CPU and GPU, we take the average of 10,000 inferences while ignoring the first 100 runs for warm-up. Table 5 shows the runtime per inference across the three hardware platforms. As demonstrated, our approach achieves 73.86× speed-up over CPU and 1.41× speed-up over GPU. It is important to note that the GPU platform we use is a highly advanced GPU built for ML acceleration. It is equipped with large on-chip memory (~16× higher than GED) and therefore, this is not entirely a fair comparison point for GED. Moreover, GPU is significantly more power-hungry than ASIC-based accelerators. Nevertheless, GED outperforms even this powerful GPU.

Comparison with Eyeriss v2: We compare the performance of GED with Eyeriss v2 under the same area budget. The configurations of the two accelerators with similar on-chip SRAMs are:

- **GED:** 64×64 PE array with 4096 MACs, 236kB on-chip SRAMs.
- **Eyeriss v2:** 192 PEs with 384 MACs, 246kB on-chip SRAMs.

The area of Eyeriss v2 in a 65nm node is obtained from [17]. The area is scaled using the factor $s = (\eta_2/\eta_1)^2$ to match with the area

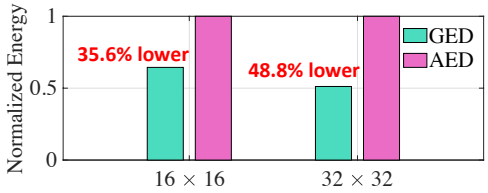

Figure 10: GED vs. AED: Normalized energy (MobileNet-v1).

Table 5: GED vs. {CPU, GPU} Performance for MobileNet-v1.

Hardware Platform	Runtime (ms)	Speed-up of GED
CPU: Intel(R) Xeon(R) Gold 6132, @2.60GHz, Memory: 768GB DDR4	64.26	73.86×
GPU: NVIDIA Tesla V100S-PCI, Memory: 32GB HBM2	1.23	1.41×
GED: 64×64 PE array, 396kB SRAMs, @1GHz, Off-chip bandwidth: 512 bits/cycle	0.87	1.00×

of GED where η_1 and η_2 represent the feature sizes from respective technology nodes. Eyeriss v2 reports performance for MobileNet-v1 with width multiplier of 0.5 and resolution multiplier of 0.57, therefore, we also use the same benchmark for GED. As compared to Eyeriss v2, GED shows 1.95× speed-up for this variant of MobileNet.

Eyeriss v2 uses a heavily customized NoC and dataflow to accelerate DNN workloads that require substantial area for control logic and register storage within each PE. In contrast, the computation cores of GED do not require local control or register storage within each PE. GED performs vector operations by directly accessing on-chip buffers while the compute engines operate under centralized control. As a result, the GED architecture can fit more MAC units than Eyeriss v2 under the same area budget, leading to more parallelism during the computation, which directly contributes to its higher performance.

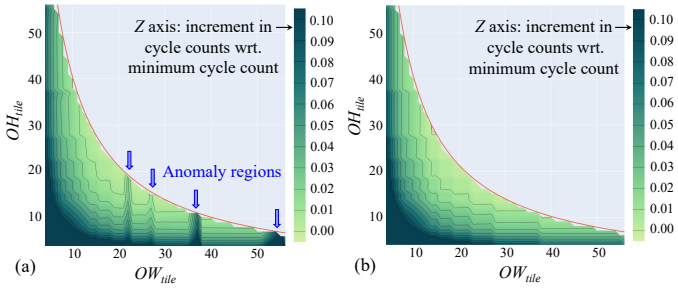

Figure 11: Tiling space exploration for DwC1 while executed on GED: (a) skewed tiling allowed, (b) balanced tiling only.

6.2 Design Space Exploration

Tiling space exploration: Due to limited on-chip storage, CNN layers are executed by slicing the data tensors into tiles. The sizing of these tiles in each layer impacts the accelerator performance. To optimize tile sizes for our new GEMM-based DwC operation, we performed tiling space exploration to develop an efficient tiling algorithm for DwC. Fig. 11 illustrates the relative increment in cycle count with respect to minimum cycle count as we vary tile sizes for the height/width dimensions of ifmap/ofmap. The tiling space is shown for a representative layer, DwC1, while executed on H1 (defined in Section 6.1). The Z axis of the plot is indicated by the color heatmap: lighter [darker] green indicates smaller [larger] cycle count for the layer. The gray regions are illegal due to the size constraints from the on-chip buffers (red curve).

Fig. 11(a) allows any values for the height and width tiles (i.e., OH_{tile} and OW_{tile}) as long as they do not violate the buffer size constraints. As indicated by the blue arrows, there are anomaly regions where the cycle counts are up to 10% worse than the minimum cycle count. An example point in such an anomaly region is OW_{tile} = 55. Since for DwC1, OW = 112, at this point the width of ofmap tensor gets split into [55, 55, 2]. Since the last tile is very

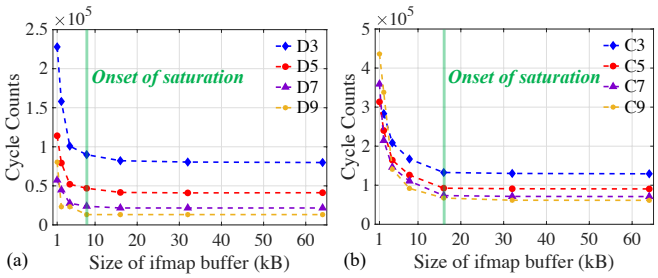

Figure 12: Cycle counts vs. buffer sizes for (a) DwC layers, (b) Conv2D layers. (output buffer size = 4 × ifmap buffer size; cycle count of C3 is very large at ifmap buffer = 1kB and omitted for better visibility).

narrow (i.e., 2), and while the computation in GEMM and load/store operation can normally overlap, in this case there is not enough computation for the GEMM to do. Therefore, the load/store operations do not get properly overlapped with GEMM computations resulting in additional stall cycles. Similar trends are seen for other DwC layers as well. In order to avoid such skewed tilings we impose constraints to generate a more balanced set of sizing for the tiles. For example, instead of [55, 55, 2] the tiling generation algorithm picks the most balanced one with three tiles, which is [38, 37, 37]. Fig. 11(b) shows the tiling space for the balanced tiling scenario where the space does not contain any anomaly regions.

From the tiling space of Fig. 11(b), it is also evident that the middle region of the plot, where $OH_{tile} \approx OW_{tile}$, provides smaller cycle counts than other regions: since the weight data is reused across both the height and width dimensions of ifmap, these dimensions have equal importance for tiling. Based on these observations, we choose an integer tiling scheme that maximizes $OH_{tile} \times OW_{tile}$ under the constraints that the values are nearly equal and balanced. This produces tile sizes for which the cycle counts are very close to the minimum cycle counts, e.g., for DwC1 on H1, minimum cycle count over the tiling space is 80351 while the cycle count at the tiling generated by our approach is 80356.

Requirement of buffer sizes: We analyze the on-chip buffer size requirement of DwC and compare them with the buffer size requirement of Conv2D since both types of layers are executed in the GEMM core in GED. We execute individual representative DwC and Conv2D layers from MobileNet on GED with a 32×32 PE array and profile the cycle counts as a function of buffer sizes. The weight buffer requirement for DwC is very small (≤ 1kB to store a filter tile for this configuration), and increasing the weight buffer size does not impact DwC cycle counts. Across Conv2D, cycle counts saturate after 32kB making 32kB an optimal choice. Thus, a weight buffer optimized for Conv2D also meets the optimal size requirement for DwC.

Fig. 12 shows the cycle counts of four representative DwC and Conv2D layers (Di and Ci denote ith DwC and Conv2D layer, respectively) as a function of the ifmap buffer size. The output buffer size is set to be 4× the ifmap buffer size while 32kB is used for the weight buffer. As can be seen, the cycle counts of the DwC layers start to saturate at 8kB which is at the left from the point (i.e., 16kB) where the saturation starts for Conv2D. Therefore, ifmap and output buffer sizes that are optimally chosen for Conv2D meet the optimal buffer size requirements for DwC as well. These demonstrate that the hardware resources in the GEMM core that are optimized for Conv2D also work well for our GEMM-based execution of DwC.

7 CONCLUSION

A new methodology is proposed to execute DwC on general ASIC-based DNN accelerators, reusing the fast GEMM core with minor hardware augmentations. All hardware modules have been developed and integrated into the VTA$^+$ DNN accelerator hardware stack, including instruction-level support and system-level testbenches. Our GEMM-based DwC demonstrates large performance gains over ALU-based, CPU-based, and GPU-based implementations.

ACKNOWLEDGMENTS

This work is supported in part by AFRL under the DARPA RTML program under award FA8650-20-2-7009 and internship at Intel Strategic CAD Labs. The U. S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL, DARPA, or the U. S. government. The authors would like to acknowledge the contribution of Zhiang Wang from UCSD.

REFERENCES

[1] A. G. Howard, *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861*, 2017.
[2] F. Chollet, "Xception: Deep Learning With Depthwise Separable Convolutions," in *Proc. CVPR*, July 2017.
[3] M. Tan and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *Proc. Int. Conf. on Machine Learning*, vol. 97, pp. 6105–6114, June 2019.
[4] M. Tan, *et al.*, "MnasNet: Platform-Aware Neural Architecture Search for Mobile," in *Proc. CVPR*, June 2019.
[5] X. Zhang, *et al.*, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *Proc. CVPR*, June 2018.
[6] N. P. Jouppi *et al.*, "In-datacenter Performance Analysis of a Tensor Processing Unit," in *Proc. ISCA*, pp. 1–12, Jun. 2017.
[7] T. Moreau, *et al.*, "A Hardware-Software Blueprint for Flexible Deep Learning Specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.
[8] H. Esmaeilzadeh, *et al.*, "VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis," in *Proc. ICCAD*, 2021.
[9] H. Genc, *et al.*, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," *arXiv:1911.09925*, Nov. 2019.
[10] D. Zhang, *et al.*, "A full-stack accelerator search technique for vision applications," *arXiv:2105.12842*, May 2021.
[11] S. Banerjee, *et al.*, "A Highly Configurable Hardware/Software Stack for DNN Inference Acceleration," *arXiv preprint arXiv:2111.15024*, Nov. 2021.
[12] B. Liu, *et al.*, "An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution," *Electronics*, vol. 8, Mar. 2019.
[13] M. Baharani, *et al.*, "DeepDive: An Integrative Algorithm/Architecture Co-Design for Deep Separable Convolutional Neural Networks," in *Proc. GLSVLSI*, pp. 247–252, June 2021.
[14] D. Wu, *et al.*, "A High-Performance CNN Processor Based on FPGA for MobileNets," in *Proc. FPL*, pp. 136–143, 2019.
[15] L. Bai, *et al.*, "A CNN Accelerator on FPGA Using Depthwise Separable Convolution," *IEEE T. Circuits-II*, vol. 65, pp. 1415–1419, Aug. 2018.
[16] N. Vedula, *et al.*, "X-Layer: Building Composable Pipelined Dataflows for Low-Rank Convolutions," in *Proc. PACT*, pp. 103–115, 2021.
[17] Y.-H. Chen, *et al.*, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *IEEE JETCAS*, vol. 9, pp. 292–308, Apr. 2019.
[18] T. Chen, *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proc. OSDI*, pp. 578–594, Oct. 2018.
[19] N. P. Jouppi, *et al.*, "A Domain-Specific Supercomputer for Training Deep Neural Networks," *Communications of the ACM*, vol. 63, pp. 67–78, July 2020.
[20] M. Gurel, *A Comparative Study between RTL and HLS for Image Processing Applications with FPGAs.* University of California, San Diego, 2016.
[21] "VTA Hardware Design Stack." https://github.com/pasqoc/incubator-tvm-vta.