DUET: Boosting Deep Neural Network Efficiency on Dual-Module Architecture

Liu Liu
UC Santa Barbara
liu_liu@ucsb.edu

Zheng Qu
UC Santa Barbara
zhengqu@ucsb.edu

Lei Deng
UC Santa Barbara
leideng@ucsb.edu

Fengbin Tu
UC Santa Barbara
fengbintu@ucsb.edu

Shuangchen Li
UC Santa Barbara
shuangchenli@ece.ucsb.edu

Xing Hu
UC Santa Barbara
xinghu@ucsb.edu

Zhenyu Gu
Alibaba DAMO Academy
zhenyu.gu@alibaba-inc.com

Yufei Ding
UC Santa Barbara
yufeiding@cs.ucsb.edu

Yuan Xie
UC Santa Barbara
yuanxie@ece.ucsb.edu

***Abstract*—Deep Neural Networks (DNNs) have been driving the mainstream of Machine Learning applications. However, deploying DNNs on modern hardware with stringent latency requirements and energy constraints is challenging because of the compute-intensive and memory-intensive execution patterns of various DNN models. We propose an algorithm-architecture co-design to boost DNN execution efficiency. Leveraging the noise resilience of nonlinear activation functions in DNNs, we propose dual-module processing that uses approximate modules learned from original DNN layers to compute insensitive activations. Therefore, we can save expensive computations and data accesses of unnecessary sensitive activations. We then design an Executor-Speculator dual-module architecture with support for balance execution and memory access reduction. With acceptable model inference quality degradation, our accelerator design can achieve 2.24x speedup and 1.97x energy efficiency improvement for compute-bound Convolutional Neural Networks (CNNs) and memory-bound Recurrent Neural Networks (RNNs).**

***Index Terms*—Neural networks, accelerator architecture**

I. Introduction

Deep Neural Networks (DNNs) have led to important breakthroughs that expand the possibilities of applying artificial intelligence (AI) to many domains, including visual and auditory recognition [1], [21], [25], [44], natural language processing [14], [46], autonomous driving [9], medical analysis [16], and so forth. Although DNNs have been driving the mainstream of AI applications, it is challenging to efficiently deploy them on modern hardware due to their compute-intensive and data-intensive nature.

Computing activation is at the core of DNNs computations. However, not all the activations in DNNs need accurately computed results. We observe that frequently used activation functions in DNNs, such as ReLU in CNNs and sigmoid and tanh in RNNs, are noise-resilient in particular regions. As shown in Fig. 1, activations in the negative region of ReLU and the saturation regions of sigmoid and tanh are resilient to noises induced to pre-activated values. We refer to these regions where small changes in input become negligible after activation functions as being insensitive. In other words, these

This material is based upon work supported by the National Science Foundations (NSF) under Grant No. 1719160, 1725447, 1730309, and 1925717.

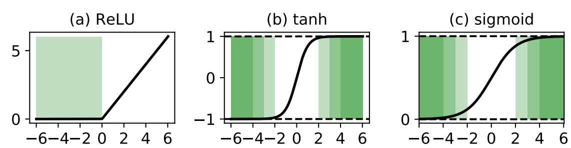

Fig. 1. Insensitive regions (shaded green) vs. sensitive regions (white) of (a) ReLU, (b) tanh, and(c) sigmoid nonlinear activation functions.

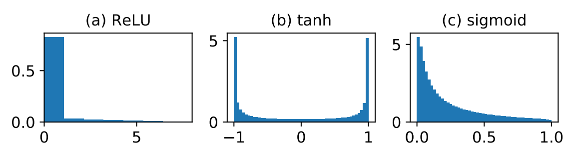

Fig. 2. Data distributions of (a) ReLU activations in pre-trained AlexNet on ImageNet, (b) tanh and (c) sigmoid activations in pre-trained LSTM on PTB.

activations in the insensitive regions can afford approximate results, while only the remaining activations in the sensitive regions need accurate computations. Moreover, as shown in Fig. 2, a large portion of activations are in the insensitive regions. This observation motivates us to use low-cost noisy computations on activations in the insensitive regions and only seek to compute expensive accurate computations on activations in the sensitive regions. While model compression techniques exploit static redundancy elimination by pruning [24], [33], [38] and quantizing model parameters [42], [47], [53], the redundancy of activations is dynamic and orthogonal to static model compression techniques. Exploiting dynamic activation redundancy opens up opportunities at computation skipping and memory access reduction.

To this end, we propose an algorithm-architecture co-design to exploit dynamic redundancy elimination and boosting DNN inference efficiency. At the algorithm level, we present the dual-module processing method: given a pre-trained DNN layer regarded as the accurate module, we propose to use a lightweight module with fewer parameters and lower precision, distilled offline from the accurate module, to approximate the

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00066

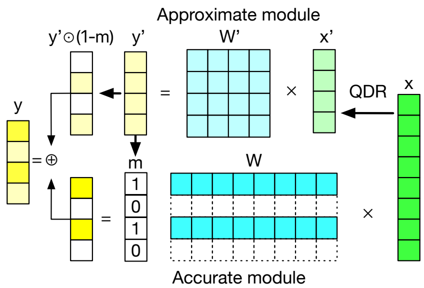

Fig. 3. Illustration of dual-module processing. The approximate module with quantized and dimension reduced (QDR) weights $W^{'}$ and inputs $x^{'}$ spends low-cost computations on intermediate results $y^{'}$ that are further used to generate the dynamic switching map m. With dynamic switching, the expensive computation and data access of executing the accurate module are saved. The final pre-activated vector y is a mixture of accurate and approximate results.

results in the insensitive regions. We illustrate the online dual-module processing of one layer in Fig. 3: firstly, we execute the approximate module with a quantization and dimension reduction step (QDR) on the input activations; secondly, we use a threshold-based neuron-wise dynamic switching method to precisely determine which activations can afford approximate computation, represented with 0s in the switching map m; then, we only run the accurate module to compute sensitive activations, represented with 1s in m; finally, we combine the outputs of the approximate module in the insensitive region and the outputs of the accurate module in the sensitive region.

At the hardware level, we propose *DUET*, a **DU**al-modul**E** archi**T**ecture. DUET has a dedicated Speculator running approximate modules and an Executor running accurate modules. Although dual-module processing could theoretically save a significant amount of operations, it poses challenges on accelerator architecture design when processing DNN layers. On the one hand, the Speculator could become the new bottleneck or increase critical path latency and degrade overall performance. On the other hand, imbalanced workloads caused by neuron-wise dynamic switching lead to computing resources underutilized in the Executor. The architecture design of DUET features fine-grained Speculator and Executor paralleling and balanced execution in the Executor. For memory-bound DNN layers, our design can save expensive memory accesses of accurate module computed by the Executor.

To summarize, our work presents a general algorithm-architecture co-design that achieves computation saving and memory access reduction on various types of DNN including CNN, LSTM, and GRU. We list our contributions as follows:

- We present the dual-module processing method and a learning algorithm to distill a lightweight approximate module from the original accurate module, i.e., the targeted DNN layer. We also propose a threshold-based neuron-wise dynamic switching method to choose and eliminate activation redundancy (see Section II).
- We design a specialized dual-module architecture to support general acceleration of DNN layers (see Section III). We facilitate the decoupled executor-speculator design with fine-grained pipeline to hide Speculator latency. Moreover, for compute-bound layers, we design an online **adaptive mapping** to address imbalance issue and improve Executor utilization. For memory-bound layers, our design enables dynamically skipping unnecessary memory accesses (see Section IV).
- We further conduct experiments and analysis to evidence the effectiveness of our solution on various models. Compared with normal DNN execution without dual-module design, *DUET* achieves 2.24x speedup and 1.97x energy reduction on average with balanced execution and reduced memory access. Besides, the lightweight Speculator only consumes 6.6% of total area and less than 7% of total energy consumption (see Section V).

II. Dual-Module Processing

The philosophy of dual-module processing is to use approximate modules for insensitive activations and only compute accurate results for sensitive activations. We propose to learn an approximate module from each DNN layer that is regarded as the accurate module. The learned approximate modules have low-volume parameters and low precision. We propose a threshold-based dynamic switching method to decide which activations are in the (in)sensitive regions.

We first explain the dual-module processing algorithm by taking a Feed-forward (FF) layer as an example and then extend it to CNNs and RNNs. For an FF layer with batch size of one, the operation is typically formulated as $z = \varphi(y), y = Wx + b$, where W is a weight matrix ($W \in \mathbb{R}^{n\times d}$), x is an input vector ($x \in \mathbb{R}^{d}$), b is a bias vector ($b \in \mathbb{R}^{n}$), y is a pre-activated output vector ($y \in \mathbb{R}^{n}$), z is an activated output vector ($z \in \mathbb{R}^{n}$), and φ is an activation function.

A. Learning Approximate Modules

The design goal of approximate modules are two-fold: firstly, much fewer computations and memory access than the original module; secondly, approximating the original modules accurately. Inspired by dimension reduction algorithms from the machine learning community [2], [37], we propose to project the input vector($x \in \mathbb{R}^{d}$) to a lower dimensional vector ($x^{'} \in \mathbb{R}^{k}$), and then construct weight matrix ($W^{'} \in \mathbb{R}^{n\times k}$) of the approximate module with significantly fewer parameters.

We use random projection as the dimension reduction method when constructing approximate modules. We constrain the elements in projection matrix ($P \in \mathbb{R}^{k\times d}$) to be ternary, i.e., $\sqrt{\frac{3}{k}} \cdot \{-1, 0, 1\}$, following the recommended probability distribution in random projection [2]. Each element in P has $1/6$ probability of being any non-zero value in the above set, and other $2/3$ probability of being zero. Thus, we can perform the dimension reduction, i.e., Ternary Random Projection, with additions and accumulations instead of MAC operations. Finally, the approximate module's weights can afford aggressive quantization as the approximate results are only used for unimportant activations.

Optimization Object. The goal of the approximate module is to make the estimated output results as accurate as possible. To train the approximate module such that it is a good approximation of the original module, we utilize the knowledge distribution method by taking the original module as the teacher network and the approximate module as the student network [36]. To this end, the optimization goal should be minimizing

$$\sum_{s} ||(Wx+b)-(W^{'}Px+b^{'})||_2^2 \tag{1}$$

where s is the mini-batch size of inputs.

Dynamic Switching. Given the results ($y^{'}$) from the approximate module and accurate results (y). The final output vector – a mixture of results from approximate and accurate modules – can be assembled by

$$y = y \odot m + y^{'} \odot (1-m) \tag{2}$$

where $m \in \{0,1\}^n$ is the switching map for determining which output activations belong to the insensitive region, and $\odot$ is point-wise multiplication. We have

$$\begin{cases} sigmoid/tanh: & if\ |y_i^{'}| > \theta_{th},\ m_i = 0;\ else\ m_i = 1 \\ ReLU: & if\ y_i^{'} < \theta_{th},\ m_i = 0;\ else\ m_i = 1 \end{cases} \tag{3}$$

where θ_{th} is the threshold can be obtained by tuning with the training or validation set to reach targeted saving.

B. General Applicability

Our method is generally applicable to various types of DNNs. We have illustrated dual-module processing on Feed-Forward (FF) layers with general matrix-vector multiplications (GEMVs). Here, we further discuss how to extend it to Convolutional (CONV) layers and RNNs using LSTM as an illustrative example.

Convolutional Neural Network. For a CONV layer, We can apply dual-module algorithm to CNN by first doing the *im2col* transformation on input tensor. Then, the input and output become matrices rather than vectors, but the overall algorithm is the same as FF layers.

Recurrent Neural Network. Different from FF layer, LSTM layer consists of a input-to-hidden matrix and a hidden-to-hidden matrix and takes current step embedding vector and previous step hidden vector as inputs. Accordingly, we revise the dual-module processing as discussed in FF layer to construct two low-dimensional and low-precision weight matrices in an approximate module. When training the weight matrices of approximate modules, we sum the loss of all time-steps in back-propagation.

III. DUET Architecture Design

In this section, we present a dedicated dual-module accelerator named DUET, to enable the proposed dual-module processing scheme with better performance and energy efficiency. Based on DUET's architecture, we further devise different dataflow and mapping strategies optimized for CNNs and RNNs, respectively.

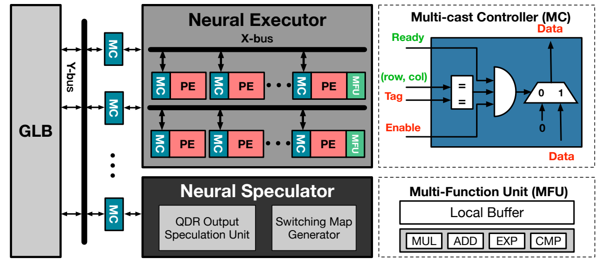

Fig. 4. Overall architecture with an Executor running accurate modules and a Speculator running approximate modules.

A. Overview

The top-level block diagram of DUET is shown in Fig. 4. Overall, the accelerator consists of three major components: an on-chip global buffer (GLB), a specialized 2D PE array called the Executor, and a decoupled Speculator to handle approximate module processing. In general, the Speculator and the Executor run in parallel. On the one hand, the Speculator uses outputs from the Executor to perform speculation, i.e., running approximate modules in the Speculator, generating approximate results and dynamic switching maps. On the other hand, Executor leverages the switching maps to reduce computations as well as memory access. The decoupled architecture of Executor and Speculator further enables a fine-grained pipeline design of the dataflow, which will be demonstrated in Section IV.

Control: DUET has two-level of control logic. The global control configures the whole system and is responsible for handling traffic between on-chip GLB and off-chip DRAM, traffic between GLB and Executor, and traffic between GLB and Speculator. The lower-level control consists of the control logic inside each PE and the Speculator, which runs independently.

Global Buffer: DUET has a 1MB GLB that can communicate with both the off-chip DRAM module and with the on-chip computation resources (Executor and Speculator) through the NoC. Besides the data needed for computation (input, weight, and output), GLB also stores the data required and generated by the Speculator. These data include the weights for the Speculator, the switching maps to be used to reduce computations for both CNNs and RNNs), the mapping configuration to balance PE workloads for CNNs (see Section IV-A) and finally the approximate speculation results (only for RNNs, see Section IV-B). GLB provides a total bandwidth of 512B/cycle to feed the Executor and the Speculator sufficiently. The parameters are chosen through design space exploration and are validated via a cycle-accurate simulator.

Network-on-Chip: As shown in Fig. 4, the NoC in DUET applies a similar design as appeared in Eyeriss [11], which has two dimensions: Y-bus and X-bus. The vertical Y-bus interacts 17 X-buses, with 16 for the Executor and one for the Speculator. Each PE in the Executor has a reconfigurable *(row, col)* ID, and different X-buses or PEs have the same ID if they are receiving the same data. During the data transmission, each data loaded from the GLB is given a specific *(row, col)* ID. In

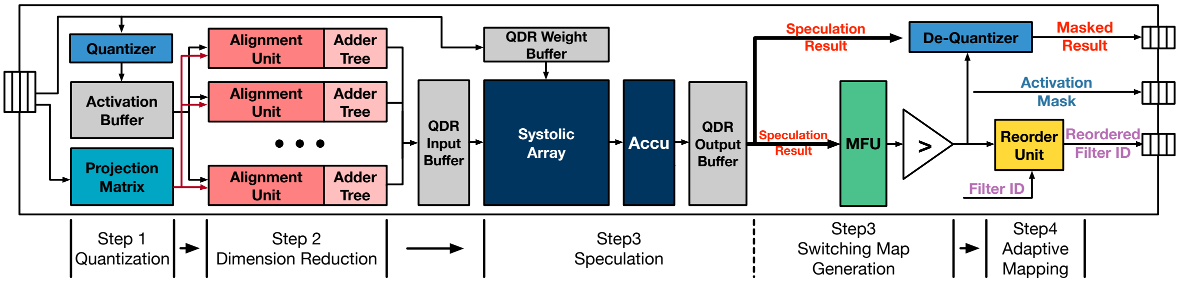

Fig. 5. Illustration of the Speculator that runs approximate modules. The Switching Map Generator computes the dynamic switching indices for computation skipping in the Executor while balance workloads.

order to correctly deliver the data to its destination, 17 Multi-cast Controllers (MC) as shown in Fig. 4 are used to compare the *row* ID with the *row* ID of each X-buses. Another 16 MCs will match the *col* ID of the data with the destination's *col* ID tag. The unmatched X-buses and PEs are deactivated to save energy.

B. Hardware Efficient Speculator Design

Fig. 5 shows the overall architecture of the Speculator. The design target of the Speculator is – with small area and energy consumption – to provide sufficient throughput for generating approximate results and switching maps that supply the Executor to reduce computations with negligible loss of model quality. Each index in the switching map is one-bit, indicating whether a specific neuron should use the approximate results or need to be updated later by the accurate results from the Executor. In other words, the Executor only needs to compute the output neurons with switching index equals to 1 in the switching map.

As discussed in Section II and illustrated in Fig. 5, our dual-module processing method introduces three auxiliary steps to generate switching maps before the layer execution: the quantization step, the dimension reduction step, and the speculation step. The first two steps together make the speculation computation to be both low-dimension and low precision (INT4). The approximate results will be further passed through the Multi-Function Unit (MFU) to perform non-linear activation (e.g., *ReLU*, *tanh*, and *sigmoid*) and generate final approximate results. Speculation output that falls in the sensitive region will be considered as important output and have its switching index set to one in the switching map. Finally, after generating switching maps, an optional analyzing step called adaptive mapping is added afterward to process the information. This step is essential for CNN execution to help balance the PEs' workloads (See Section IV-A). We demonstrate the process of the Speculator step by step below.

Pre-Step: Data preparing. At the beginning of running the Speculator, the ternary projection matrix **P** and quantized & dimension-reduced (*QDR*) weights are loaded into on-chip buffers, i.e., the Projection Matrix buffer and the QDR Weight Buffer as shown in Fig. 5.

Step 1: Quantization. We use 16-bit fixed-point data in the Executor's high-dimensional execution, where the fixed-point data are essentially INT16 with a scale in FP32. Therefore, the output of the Executor will also be the same format. In order to use these results as the input activation and multiply them with the low-precision QDR weights, we need to first quantize them from INT16 to INT4 values with the Quantizer and then stash them into the Activation Buffer. The conversion from 16-bit to 4-bit is realized by simply truncating the 12 least-significant bits (LSBs) and keeping the four most-significant bits (MSBs). Accordingly, the scaling factor also needs to be increased by 4096 (2^{12}) to maintain the same quantization range. The rounding error caused by 16b-to-4b quantization is inevitable, but such aggressive quantization applied in the Speculator has negligible impact on model quality as the values are only used in the insensitive regions.

Step 2: Dimension reduction. The dimension-reduction step multiplies the quantized input activation with the projection matrix **P** to further reduce the dimension. Since all the values in **P** come from the set $\{-1, 0, 1\}$, we can efficiently implement this step in hardware with addition and accumulation instead of using multipliers. As shown in Fig. 5, the Alignment Units first change the signs of input activations according to the element values of **P**. Then the Adder Trees perform the accumulation of sign-modified input activations. This carry-save-adder tree structure operating in pipeline provides high throughput for dimension reduction. The dimension-reduced inputs are buffered in *QDR* Input Buffer for the next step.

Step 3: Speculation computation & Switching map generation. As shown in Fig. 5, after quantization and dimension-reduction, the inputs and weights in low dimension and low precision are stored in QDR Input Buffer and QDR Weight Buffer. We then conduct the INT4 inner-product operations using a 16×32 Systolic Array. We use systolic in the Speculator rather than another 2D PE array to perform regular dense GEMM operation because such design achieves better energy efficiency with simple control. The size of the systolic array is searched based on design space exploration in Section V-F. The results from the Systolic Array will be accumulated with the partial sum and sent to the Multi-Function Unit (MFU) to calculate the final activated output. The MFU implements different activation functions, including *ReLU*, *tanh* and *sigmoid*. Equation (3) represents how to generate the switching map, and we further compare these approximate

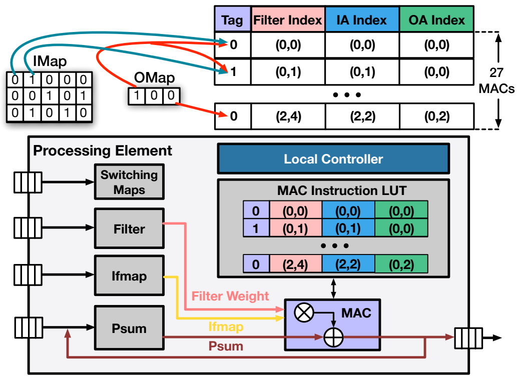

Fig. 6. Specialized Processing Element that handles dynamic switching to skip computations.

results with the predetermined thresholds from the fine-tuning phase. All the values that fall within the sensitive region will have their switching indices to be one.

Step 4: Adaptive Mapping. As mentioned above, with the switching maps and speculation results, the Executor only needs to compute a small portion of accurate activations. However, processing with sparsity information causes different PEs to have imbalanced workloads for CNNs [20], [45]. To tackle this problem, we further propose the **Adaptive Mapping** strategy to change the order of the computation between different output channels so that output tiles with similar workloads are computed together. Such reordering is very hardware efficient that can be handled directly inside the Speculator using a dedicated Reordering Unit. We will further demonstrate our approach in Section IV-A. As for RNNs, our designed dataflow guarantees that there is no workload imbalance between different PEs, and we can bypass the reordering operation. However, we do need to store the results for approximate activations; these 4-bit results are dequantized to 16-bit data with the Speculator's dequantizer and sent to GLB.

To sum up, the Speculator produces three types of information: firstly, the switching maps indicating which output can be skipped by the Executor; secondly, the reordered filter ID that the Executor follows when computing the output feature map; thirdly, the approximate activations required by RNN models. With decoupled Speculator and Executor design and the fine-grained pipeline between these two components, we can hide the latency of speculation as much as possible to increase the overall performance. The hardware efficient quantization and dimension reduction also greatly reduces the area and energy overhead to process approximate modules.

C. Specialized Executor Design

The Executor processes accurate modules using 16-bit fixed-point arithmetic. Overall, the Executor applies a typical spatial 2D PE architecture to explore different data reuse types. Furthermore, we add specialized hardware components inside each PE to leverage the dynamic switching maps to reduce computation and memory access. As shown in Fig. 6, each PE consists of dedicated buffers for different types of data, a 16-bit pipelined multiplier and adder, and a local multiply-accumulate micro-instruction lookup table (MAC Instruction LUT) which is the key of skipping unnecessary computations.

On the PE level, we regard processing DNNs as orchestrating and executing multiple MAC operations. Each MAC operation requires two loads for input and weight values, one load for partial sum, and one store for the updated partial sum. As illustrated in Fig. 6, the MAC operations are represented with micro-instructions (μinst) that stored in the MAC Instruction LUT. Each μinst contains the input activation (IA) index, weight (W) index, and output activation (OA) index indicating the address of the load/store operations. Besides, an extra tag-bit is added to enable computation saving. MAC operations with tag bit being zero will be directly skipped. We only use the indices to locate the relative positions of the values in the input/weight/output tile. As long as the tile's shape is kept the same, these indices do not change. For a given NN layer, the PEs are processing a static shape of input, weight, or output tile. Therefore, the μinst's indices only need to be generated once at the beginning of layer configuration, and remain unchanged and shared by all the PEs throughout the execution of the whole layer. The dynamic switching maps will be used to configure the tag bits for different tiles to enable computation skipping.

We further use a simple example of CNN to demonstrate how the PE utilizes switching maps to skip unnecessary computations. As for RNNs, the case is even simpler. In the example shown in Fig. 6, every step the PE is processing a $3 \times 5 \times 1$ input tile, and a $3 \times 3 \times 1$ filter tile to generate a $1 \times 3 \times 1$ psum in the ofmap. Therefore, each PE is mapped with $3 \times 3 \times 3 = 27$ MAC operations for each step. To configure the tag-bit, we load the corresponding IMap and OMap from GLB and stored them in PE's local buffer. The OMap shows whether an output activation needs to be computed by the executor, and IMap shows whether the input activation is zero or not. We only mentioned output switching maps (OMap) in the previous content, but for CNNs, the ineffectual neurons are set to zero, making the OMap become the input sparsity maps (IMap) for the **next** layer. In this way, we pay the overhead of dynamic switching once, but the switching map is used twice for the current layer's OMap and the next layer's IMap. Besides, if a predicted effectual neuron turns out to be ineffectual after *ReLU*, we will update the switching index of that neuron from 1 to 0 and then send it to the GLB. With this **correction** step, when the OMap is loaded as IMap for next layer, it will have even higher sparsity to save more computations.

Based on the switching maps, we can easily decide whether a specific instruction can be skipped or not. For example, as shown in Fig. 6, the OMap shows that only the first element in the $1 \times 3 \times 1$ output tile needs to be computed. Therefore, all the MAC operations related to the other two output neurons can be discarded, leaving only nine necessary MAC operations. Moreover, since the IMap shows that 2/3 of the input activations are zero, we can further reduce six MAC operations. Finally, each MAC operation is augmented with a 1-bit tag using simple Boolean logic. PE's local control will locate the valid

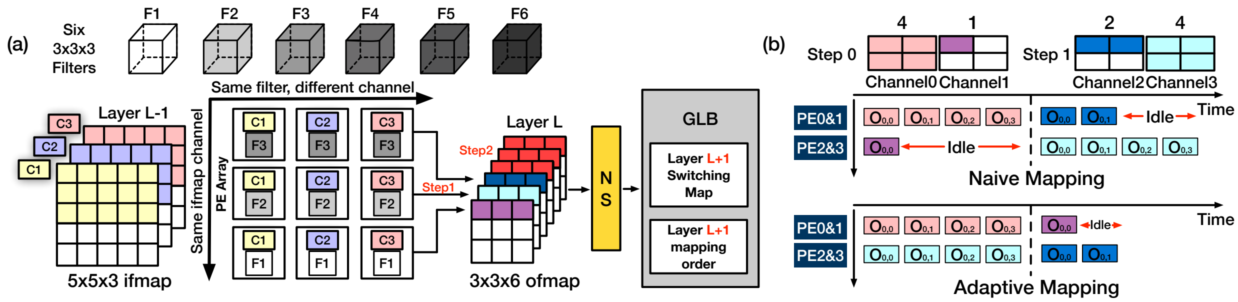

Fig. 7. Illustration of (a) CNN Dataflow and (b) Adaptive Mapping Strategy. Ifmap data are shared vertically in Executor, and each PE-row computes for a specific ofmap channel. The Speculator uses the Executor's output to perform output speculation for the next layer. After generating switching maps, the Speculator directly performs adaptive mapping to balance PE's workload before execution.

MAC operation in the instruction buffer to perform necessary computation.

IV. Dataflow and Mapping

DUET is able to support both CNNs and RNNs with a unified architecture. The efficient hardware design presented in Section III guarantees that different units can well handle their required tasks. However, to achieve the expected performance speedup and energy saving, we still need to have fine-grained dataflow design and hardware mappings. Essentially, we decouple the Speculator with the Executor so that they can run in parallel and let switching maps to be generated prior to run the Executor. Nevertheless, the data dependencies between the current layer's output and the next layer's run-ahead approximate module execution makes it challenging to orchestrate the execution dataflow of the Executor and the Speculator in pipeline. Therefore in this Section, we separately describe how DUET handles CNN models and RNN models while addressing different bottlenecks during the run-time.

A. Processing CNNs with Balanced Execution

DUET computes a CNN model layer by layer. The accelerator is configured once for each layer to sequentially process batches of ifmap. The configuration bits are generated offline based on the layer structure and hardware constraints. During runtime, they are loaded as a long scan chain to configure the accelerator to process a layer in a certain tiling shape and set up the mappings for the Executor and the Speculator.

a) Overall Pipeline: For illustrative purposes, suppose we have a CNN layer with a $5 \times 5 \times 3$ ifmap and six $3 \times 3 \times 3$ filters, as shown in Fig. 7(a). Therefore, the ofmap would be of size $3 \times 3 \times 6$, assuming no padding. In this example, the Executor consists of $3 \times 3 = 9$ PEs. Each line of PEs will together computes a specific channel of the ofmap. Within the same PE line, different PE loads different tiles of ifmap and filter that contribute to the same area in a specific ofmap channel. Thus, the output partial sum will be horizontally accumulated. Since different lines of PEs compute different output channels, input feature maps are shared vertically within the same column of PEs.

During the execution, the ofmap is computed step by step. Each step the PE array generates a $1 \times 3 \times 3$ output tile, each line generates a $1 \times 3 \times 1$ tile. Therefore, it takes 6 steps to finish computing the ofmap. After finishing a step, we first move along the channel dimension of the ofmap to compute the next tile. In this example, after the first tile is generated, we then compute the **red** tile as shown on the ofmap in Fig. 7(a). In this way, when these two tiles are sent to the Speculator as input activations to perform speculation, the speculation results using tile1 and tile2 can be further accumulated together. Otherwise, if tile1 and tile2 are different areas in the same channel, the speculation output of tile1 and tile2 will be also at different areas that cannot be accumulated. This will increase the memory footprint of the speculation.

Therefore, the Executor computes each layer's output tile by tile, while the Speculator uses computed tiles to perform sparsity speculation for the **next** layer. In this way, we can pipeline the speculation with execution, hiding the latency of speculation while lowering the memory overhead. In the example shown in Fig. 7, while the Executor is still computing layer **L**'s output, the Speculator is already using existing results to generate switching maps for layer **L+1**. Therefore, when the accelerator start to process layer **L+1**, it already has the OMap to be used to skip a considerable amount of unnecessary computations. Also, as mentioned above, the OMap of the previous layer will be updated by the Executor and serve as the IMap for the next layer to further reduce computations.

b) Adaptive Mapping for Balanced Execution: The key challenge of computation skipping in CNNs is the workload imbalance caused by irregular sparsity distribution, which further results in PE under utilization and performance degradation. Specifically, different PEs can have different numbers of necessary MAC operations in the same computation step. Therefore, PEs with less computations will finish earlier and have to wait for the other PEs.

Depending on the sparsity type, there have been several ways to balance the workloads. Prior work like [20], [41] focus on input and weight sparsity. Since the weight sparsity is static and generated offline during the training phase, they adopt offline sorting based on the weight density to reorder the computation sequence prior to the inference. This idea is not suitable for dynamic output switching (OS), as the switching map is generated dynamically during run-time. Online sorting will incur longer latency and energy consumption.

Other work like [4], [45] target on output sparsity. These work are based on a coupled executor/predictor design with *early termination* mechanism. Specifically, the prediction is indeed part of the execution process. If the prediction results indicate the output to be zero, the execution will be stopped here. Otherwise it will be completed. To tackle the workload imbalance caused by output sparsity, they mainly use two approaches. The first idea is to enable asynchronous PE execution, so that whenever a PE finishes its current computation, it can initiate a new computation step. However, this approach causes significantly design and energy overhead due to extra memory access and complicated NoC/buffer design. The second idea is to apply an empirical approach by expecting the computations to be evened out along a certain input dimension. For instance in *Predict* [45], the authors claim that the summation of the computations with the same coordinate across different ofmap channels is nearly the same. However, in order to achieve this balance, they need to increase the tile size of each computation step, which requires larger local buffer and memory footprint. We believe these approaches are sub-optimal and energy inefficient, and the reason is because their dataflow is based on single-module architecture which cannot generate the switching maps prior to the execution.

In DUET, we solve the imbalance issue by the decoupled speculation/execution and hardware efficient dynamic **adaptive mapping**. The decoupled design ensures the switching maps to be generated prior to its execution, which also gives us extra time to perform online sorting using the proposed adaptive mapping. We further design a dedicated Reorder Unit in the hardware to reduce the mapping latency.

We use Fig. 7(b) to demonstrate our approach. Suppose in each step, we can generate two 2×2 switching maps corresponded to two ofmap channels. Besides, the Executor has 4 PEs that are organized in a 2×2 square. In the normal dense case, output channel 0 and 2 will be mapped to row 0 (PE0&1), and output channel 1 and 3 will be mapped to row 1 (PE2&3). Therefore, Channel 0 and 1 will be later processed at the same time, while channel 2 and 3 are computed together. However, due to output sparsity, the workload of these channel are unbalanced as shown in the figure. Thus, such naive mapping strategy will cause different row of PEs to be under-utilized within the same step. In this case, a better computation sequence would be to compute 0 and 3 first, followed by channel $1, 2$.

Therefore, our design achieves more balanced PE workloads by changing the order of computing different ofmap channels. Since the switching maps are generated in advance, we can examine the total workloads for different output channels within several tiles. Based on these numbers, we further group the output channels that have comparable operations together. After this, a new sequence of ofmap channel is generated. Later when the accelerator processes next layer, the Executor will load filter weights according to this new sequence to have balanced computation time.

Note that, adaptive mapping only changes the order of computing the ofmap channels. In other words, it only affects

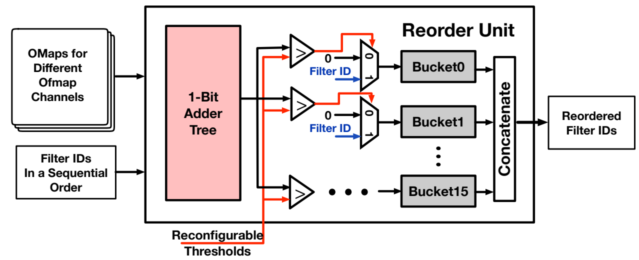

Fig. 8. Reorder Unit implements adaptive mapping for balanced execution. The output switching maps for different channels are summed up separately and divided into several groups. The IDs of different filters with similar workloads are stored together in a same bucket and sent out together.

the sequence of loading the filter data, while the ifmap access and data reuse pattern are not influenced. Also, the output are sequentially stored in the GLB according to their original sequence. For example, even though Channel **0,3** are computed together at first, and Channel **1,2** comes later, in the GLB they are still stored as Channel **0,1,2,3**. In this way we can keep the ID unchanged for the next layer when loading the ifmap data.

Our adaptive mapping only considers the imbalance issue caused by output sparsity so that different rows of PEs are balanced. Inside each row, there will still be imbalance within the PEs due to input sparsity. However, we observe with our experiments that, it is negligible compared with output imbalance. Besides, further enabling more complicated mapping and reordering strategy will considerably increase the sorting overhead.

c) Architecture Support for Adaptive Mapping: In our design, the adaptive mapping is done inside the Speculator's Reorder Unit. As tiles of switching maps are sequentially generated, we send them to the Reorder Unit in addition to writing them back to the GLB. The design of the Reorder Unit is shown in Fig. 8, it consists of multiple 1-bit adder trees that will sum up all the switching indices corresponded to a specific output channel. Each output channel will have a total count of estimated computations. Note that, this number does not represent the workloads for the whole channel, but for the tile that will be processed within one computation step. Then, we compare the sum with preset interval thresholds and write the channel IDs to the corresponding buffers, i.e., Buckets shown in Fig. 8.

Using the same example in Fig. 7(b), each of the four output channels will have a sum indicating its computation quantity. In this case, the sums are $4, 1, 2, 4$ for channel $0, 1, 2, 3$. Since there are two PE lines in the Executor, there will be two Buckets in the Reorder Unit. In this case, the channel that has more than two valid output elements will be stored in Bucket0, and others that contain less valid output will be stored in Bucket1. To do so, we only need to compare the four sums with a preset threshold 2, and channel ID $0, 3$ are grouped together, while ID $1, 2$ will be stored in the second Bucket. During execution, the Executor will load the filter data in the order of Bucket0, Bucket1, which gives us the optimal computation sequence as demonstrate above.

B. Reduceing Memory Accesses and Computations on RNNs

Finally, we describe how DUET handles RNN models. Compared with CNN models, processing RNNs is more memory-bound. For instance, given an input vector of length 1024, the weight matrix used for computing each gate in an RNN cell would be 1024×1024, which requires a 2MB of memory space. Furthermore, although the weights are shared between different input elements, the recurrent data dependency requires us to compute the previous hidden state before we can process the next one. Therefore, during the execution, we have to constantly and cyclically load each weight matrix from the off-chip DRAM. This motivates us to focus on reducing the off-chip memory access with the proposed mechanism, while keeping the dataflow simple enough to ease the control overhead and avoid workload imbalance.

a) Overall Dataflow: For illustrative purpose, we consider an LSTM network with two recurrent LSTM Cells L_1, L_2 and an input sequence with 3 elements x_1, x_2, x_3, as shown in Fig. 9(a). The inference is executed element by element and then layer by layer. To be more specific, as demonstrated by Fig. 9(b), for the first element x_1, the weights of L_1 are fetched from DRAM. Due to the limited on-chip memory capacity, each time we can only load part of the weight matrix corresponded to a specific gate. After the first hidden state of L_1 is computed, we use the results and x_2 to compute the second hidden state. To do so, we need to reload L_1's weight from DRAM. Finally, after all three input elements are passed through L_1, the same process is repeated for L_2.

Inside each layer, the computation is also handled in a sequential pattern. Using the same example in Fig. 9(b), suppose we are passing x_1 through L_1. We first compute the input gate i and then the forget gate f, followed by the update gate g to compute cell state C, and finally, we compute the output gate o to get the hidden state h. The computation is mainly matrix-vector multiplication and vector accumulation and activation functions.

b) Gate-level Dual-Module Pipeline: With the baseline dataflow, we further introduce how to utilize the Speculator to perform speculations for RNN models and hide the speculation latency. In DUET , we speculate the output of each gate before its execution. As illustrated by Fig. 9(b), we start with x_1 and L_1 and perform quantization and dimension reduction for the input gate i. Similar to CNN, this will give us a binary switching map indicating the important neurons that need to be updated with accurate results from the Executor. What's different is that, apart from the switching maps, we also store the approximated results for those *ineffectual* output neurons in the GLB. After the sparse high-precision computation for gate i is finished in the Executor, we add the two vectors together. As a result, in the final output vector, the approximate activations are generated by the Speculator, while the effectual activations are computed in the Executor. With the switching maps, for the weight matrix and bias vector, only the rows related to the accurate output activations need to be fetched from DRAM. Our approach saves memory accesses and computations.

During the Executor's execution of input gate i, the Speculator can start the speculation for the forget gate f, since we only need x_1 and h_0 as our input to perform the speculation. Similarly, the speculation of the other gates can also be hidden with the Executor's computation. Throughout the processing of each layer, only the speculation for input gate i cannot be hidden due to data dependencies.

c) Reducing Off-chip Memory Access and On-chip Computations with OMap: As shown by Fig. 9(c)(d), each PE line in the Executor is mapped with a specific row of the weight matrix to be multiplied with the input vector to generate a single output value. Therefore, if the switching map indicates that a specific output neuron is ineffectual, then we can completely skip the computations related to this neuron during the execution. More importantly, there is no need to load the corresponding row in the weight matrix from the DRAM. Saving weight data access is especially crucial as the profiling results show that off-chip memory access greatly influences the overall performance and energy consumption. With the proposed dual-module processing mechanism, we can directly reduce the amount of data to be loaded from DRAM to GLB and from GLB to Executor. The overall computation complexity and processing time are also reduced, further boosting the performance of executing RNN models.

V. Evaluation

In this section, we evaluate the dual-module processing algorithm and the supporting accelerator architecture of our co-design for improved inference efficiency of DNNs. At the algorithm level, we present the trade-off between inference quality and efficiency in terms of FLOPs reduction and data access reduction. At the architecture level, we demonstrate the improved efficiency with DUET and compare DUET with state-of-the-art DNN accelerators with computation skipping.

A. Algorithm Evaluation

Benchmarks. We evaluate our method on a set of machine learning tasks. We choose AlexNet, ResNet18, and ResNet50 to perform image classification on the ImageNet dataset. We also evaluate RNN-based models on language modeling with LSTM and GRU using the PTB dataset and machine translation with GNMT using the WMT16 en-de dataset.

Quality and Efficiency trade-off. Our dual-module processing method provides a trade-off model inference quality with improved efficiency. With acceptable quality degradation, DUET can boost DNN execution efficiency that could be critical in latency and energy-constrained application scenarios. In Fig. 10 (a) and (b), we show the FLOPs reduction at different levels of accuracy loss in both top-1 and top-5. With 1% top-1 accuracy loss according to MLPerf, our method can reduce operations by 3.33x and 5.15x using AlexNet and ResNet18, respectively. Comparing with prior methods computation skipping in CNNs, namely SnaPEA [4], FBS [19],

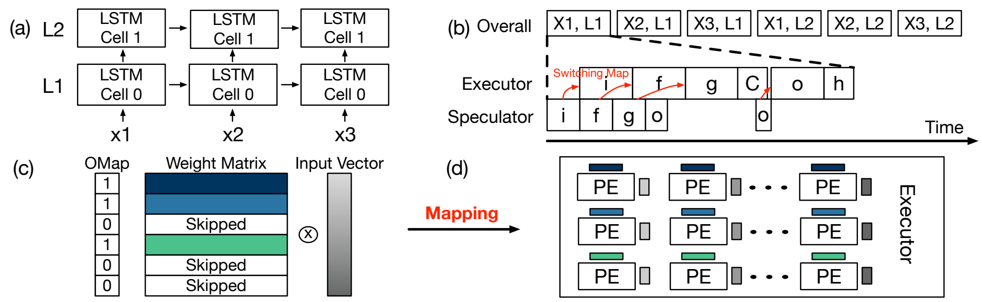

Fig. 9. Illustration of (a) sample LSTM network, (b) RNN Dataflow and (c)&(d) Executing GEMV with reduced computations and memory access. We apply a gate level speculation/execution pipeline to hide the speculation latency. Each PE-row in the Executor is mapped with a dot product between a row of weight matrix and the input vector. Each PE-row will finally generate a single output value. Therefore, if the switching index is zero, we can directly skip a complete dot-product operation.

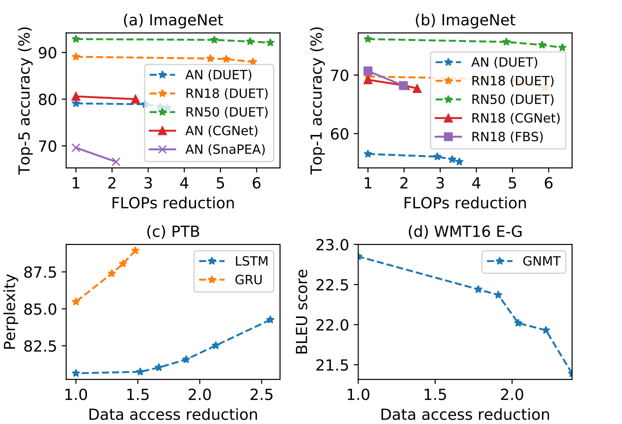

Fig. 10. Model inference quality vs. savings. Dual-module processing of DUET achieves better quality and saving trade-off and supports a wide range of DNN models.

and CGNet [27], our approach can achieve better quality and operation reduction trade-off.

As shown in Fig. 10(c), our method can reduce off-chip weight data access by 1.89x while achieving negligible quality degradation with one perplexity increase from baseline. Similarly, as shown in Fig. 10(d), serving online translation with GNMT is latency-sensitive because of the memory-bound nature of accessing weights of different LSTM layers at each time-step. With unnoticeable translation quality by users such as one BLEU score loss, our method can achieve 2.22x reduction on off-chip weight data access of the four-layer decoder in GNMT.

B. Architecture Evaluation Methodology

To evaluate our design, we develop a cycle-level simulator based on the architecture as in the state-of-the-art CNN accelerator [11]. Because dual-module processing is data-dependent, we adopt a hybrid simulation methodology: we integrate the cycle-level simulator with a deep learning framework, i.e., PyTorch. The input data to the simulator, including input activations, model parameters for Executor, and Speculator parameters, are all extracted from PyTorch. We use results from RTL synthesis by DesignCompiler under 45nm technology

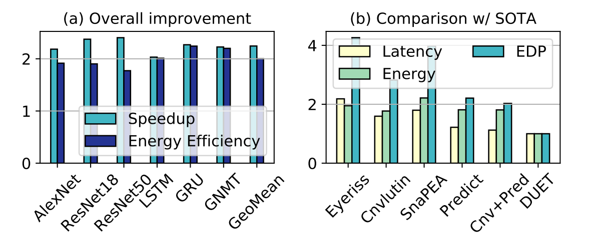

Fig. 11. (a) Overall performance speedup and energy efficiency; (b) Comparison with other accelerators.

for DUET control logic. We use CACTI and Micron Power Calculators for SRAM and DRAM estimation. Table I lists the area of major components in our design. The primary area consumption comes from the on-chip memory buffers, while the Executor accounts for 40.0% of the total chip area, and the Speculator only accounts for 6.6% of the area.

Comparison baselines. We first use single-module architecture with only the Executor as the baseline architecture. We also compare DUET's performance, energy consumption, and energy-delay-product (EDP) with state-of-the-art accelerator when running the same workload. Specifically, we extend and validate the simulator to support Eyeriss [11], Cnvlutin [5], SnaPEA [4], and Prediction [45]. These architectures span over dataflow optimization and computation skipping of sparse activations, which sufficiently supports the evaluation of DUET. We first illustrate the imbalanced execution caused by neuron sparsity and how our proposed adaptive mapping can mitigate it. Then we use other sparsity-oriented works to demonstrate the advantages of the proposed dual-module architecture design.

C. Performance speedup analysis

Overall speedup. We first show the overall speedup in Fig. 11(a). Compared with the single-module baseline design, DUET achieves 2.24x average speedup on typical CNN and RNN models. The performance improvements mainly come from three aspects: firstly, total computations are reduced with dual-module processing; secondly, hardware-efficient adaptive mapping ensures balanced execution and high PE utilization in the Executor; finally, advanced switching map generation

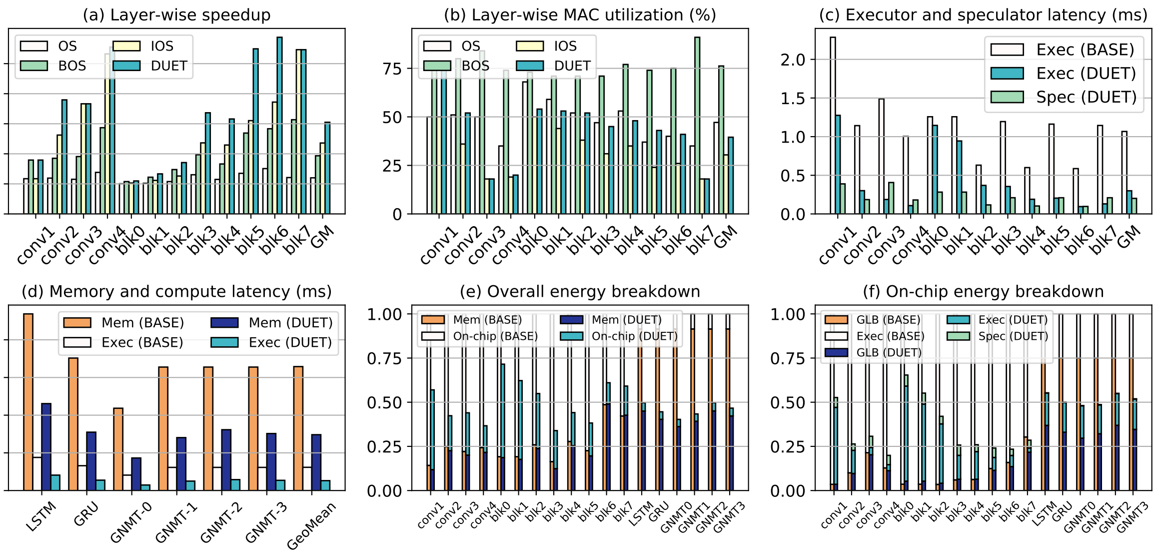

Fig. 12. Breakdown analysis for DUET. (a) Layer-wise speedup improvement applying different techniques in DUET. (b) Layer-wise MAC utilization improvement. (c) Latency comparison between the Executor, the Speculator, and the baseline single Executor design. (d) Memory and compute latency for RNN models. (e) Overall energy breakdown (w/ off-chip memory access). (f) On-chip energy breakdown (w/o off-chip memory access).

greatly reduced off-chip memory access for memory-bound workloads.

Layer-wise breakdown for different techniques. To further give more insights to the evaluation results, we provide a layer-wise breakdown for the CONV layers of AlexNet and ResNet18, and we show the effectiveness of our schemes in four stages. The comparison baseline is to only use the Executor with the same mapping and basic dataflow. As shown in Fig. 12(a), skipping computations given output switching map (OS) without adaptive mapping can only obtain 1.20x speedup on average due to imbalanced execution. Enhanced by adaptive mapping, i.e., Balanced Output Switching (BOS), the performance speedup can achieve 1.93x. Moreover, with integrated input and output switching maps (IOS), the performance can be boosted to 2.36x as more computations can be skipped. Finally, the integrated input and output switching maps (DUET) design with adaptive mapping can achieve a 3.05x average speedup.

We use the layer-wise MAC utilization of CONV layers, from AlexNet and VGG16, in Fig. 12(b) as the metrics to evaluate execution efficiency. For example, CONV5 in AlexNet has 65.5% computation sparsity when using OS, which could have 2.9x speedup, yet only achieves 1.36x. The gap between actual speedup and theoretical computation reduction indicates the severe imbalanced execution caused by coupled OS speculation. The average MAC utilization of OS only is less than 50%, again, due to imbalanced execution. While integrating the input sparsity with output sparsity (IOS) could have more computation reduction than using OS only, PEs are more under-utilized – on average, 30% in Fig. 12(b). We observe higher speedups on CONV layers with more channels. DUET can have more balanced execution with output switching and adaptive mapping. Compared with imbalanced OS, the average utilization of balanced OS can be improved from 47% to 76%; the average performance speedup is increased from 1.20x to 1.93x. Similarly, IOS boosted with our adaptive mapping (*DUET*), the average MAC utilization and the speedup of CONV layers increase from 30% to 39% and from 2.36 to 3.05x, respectively.

As Executor and Speculator compute for important neurons and unimportant neurons to deliver final activated results, balancing the processing time of Executor and Speculator is critical to achieving better performance rather than having the Speculator become the new bottleneck. The latency results of Executor and Speculator are shown in Fig. 12(c). Compared with baseline Executor without computation skipping enabled by the dynamic switching from Speculator, DUET can reduce Executor average latency from 1.06 ms to 0.29 ms with dynamic switching and adaptive mapping. On average, Speculator latency is 0.20 ms that can be hidden with the latency of Executor with pipelined processing.

For memory-bound RNN layers, we focus on the latency of off-chip memory access and on-chip computations. As shown in Fig. 12(d), BASE processing is severely bounded by accessing weight data from off-chip memory. Enabled by dynamic switching in DUET , the off-chip weight data accessing latency is reduced to 0.30 ms from 0.65 ms.

D. Energy efficiency analysis

Overall energy savings. As shown in Fig. 11(a), DUET can achieve 1.95x energy saving on average using Executor-Speculator dual-module processing compared with baseline single-module processing. The energy saving is achieved by cutting on-chip computations and buffer access as well as cutting off-chip data access. Specifically, for Executor, the computations and local buffer access are greatly reduced by

TABLE I
AREA BREAKDOWN.

Units	Parameters	Area (mm^2)
Global Buffer	1MB	5.655
Executor	256 (16-bit)	4.236
Speculator		0.699
Align. & Adder Trees	4096 (4-bit)	0.378
Systolic Array	16x32 (4-bit)	0.318
Activation Buffer	8KB	0.023
Projection Buffer	32KB	0.090
QDR Input Buffer	4KB	0.011
QDR Weight Buffer	128KB	0.352
QDR Output Buffer	4KB	0.011
Multi-Func. Unit	16 (4-bit)	0.034
Reorder Unit	16 Buckets (32B)	0.014

taking advantage of the prepared switching maps. For the Speculator, although we need to load QDR weights and store the computed approximate data, we make sure the computations are low-dimension and low-precision, which keeps the cost of these extra memory accesses as low as possible. Besides, the approximated results for the insensitive activations are reused to save energy consumption further.

Energy Breakdown. To further interpret the energy efficiency of DUET , we show the layer-wise energy breakdown in Fig. 12(e) and (f). For compute-bound layers such as CONV layers, the energy saving benefits are mostly from the reduction of MAC computations and local buffer accesses in the Executor. For memory-bound layers such as RNNs, reducing weight data accesses from off-chip memory enabled by DUET helps energy efficiency. These results support the above analysis and demonstrated our initial optimization targets. We show the on-chip energy breakdown in Fig. 12(f). The Speculator's energy consumption only consumes a small portion, ranging from 3.5% to 6.3% for CONV layers and less than 1% for RNNs compared with the total on-chip energy of baseline.

E. Comparison with SOTA CNN Accelerators

A special case of dua-module processing is ReLU-based output sparsity prediction typically appeared in CNNs. While the focus of DUET is supporting computation and memory accesses reduction in general DNN models, we compare with state-of-the-art CNN accelerators, i.e., *Eyeriss* [11], *Cnvlutin* [5], *SnaPEA* [4], and *Predict* [45], with computation skipping to demonstrate the benefits of our architecture design choices. We scale all designs to have the same number of MACs and similar on-chip memory size, and we normalize all results to DUET , as shown in Fig. 11(b). DUET achieves better results in terms of latency, energy, and energy-delay-product (EDP).

Performance comparison. Performance-wise speaking, Eyeriss equals a dense baseline as it only supports power-gating to save energy but computation skipping to improve performance; thus, it has the worst latency among others. Equipped with either input sparsity detection or output sparsity prediction mechanisms, Cnvlutin, SnaPEA, and Predict can reduce processing latency from computation skipping, the performance improvements are limited by only single-source computation skipping from either input or output. The workload

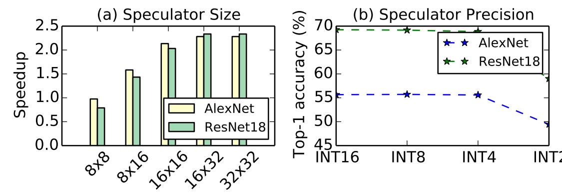

Fig. 13. Design space exploration of (a) Speculator size and (b) Speculator Precision.

imbalance caused by irregular sparse activations as in Cnvlutin and SnaPEA compromises the performance.

Energy efficiency comparison. *Cnvlutin*, *SnaPEA*, and *Prediction* use only one level of on-chip buffer and have no local data reuse, thus those designs consume 1.77x, 2.21x, and 2.21x more energy than DUET, as shown in Fig. 11(b). Even though those three designs support computation skipping, the energy consumption is at the same level as Eyeriss. Since buffer accessing is the major source of on-chip energy consumption [11], accelerators without data reuse at the local buffer level would inevitably consume much more energy to access global buffer. DUET uses the same two-level on-chip memory hierarchy as in Eyeriss with local data reuse to improve energy efficiency.

Energy-delay-product comparison. To better compare different architectures, we show the comparison on energy-delay-produce (EDP) in Fig. 11(b). The EDP of *SnaPEA* and *Predict* are 3.98x and 2.21x more than DUET , respectively. While other designs with computation skipping from both input and output activations, i.e., *Predict+Cnvlutin*, can achieve comparable performance than DUET , our design demonstrates 1.81x and 2.03x better in energy efficiency and EDP, respectively.

F. Design Space Exploration

Speculator Size. Here we investigate the impact on performance when choosing different sizes of the Speculator while fixing the size of the Executor. Specifically, we modify the systolic array size and scale other components in the Speculator accordingly. The two benchmarks we use are AlexNet and ResNet18. As shown in Fig. 13(a), when the Speculator is small, e.g. 8x8, 8x16, the performance improvements are sub-optimal. This is because the Speculator cannot provide sufficient throughput to support the Executor, processing of Speculator becomes the performance bottleneck. Besides, when increasing the size of Speculator to 32x32, the performance merely improves, meaning the latency of the Speculator is already hidden by the Executor. We need to increase the size of Executor if we want to have more speedups. Therefore, we choose the systolic array size to be 16x32.

Speculator Precision. We also study how compute precision affects the approximation quality of the Speculator, which helps us decide the trade-off between hardware consumption and model accuracy. With the same benchmarks, we show in Fig. 13 that INT4 is a preferred precision with negligible accuracy loss indicating good approximation quality. With 4-bit data representation and computation, we are able to achieve area and energy-efficient approximation, as demonstrated above.

VI. Related work

Academia and industry have proposed various architectures for the acceleration of DNNs [8], [10], [11], [12], [15], [17], [18], [28], [29], [31], [35], [43], [49], and here we focus on algorithm-architecture co-design to save computations and data accesses.

While our work focuses on dynamic redundancy elimination leveraging noise-resilience in DNN activations, an orthogonal line of research is statically pruning weights to decrease memory footprint and data access. Fine-grained weight sparsity was integrated into DNN accelerators through compressed storage and computation skipping of zero weights [20], [22], [23], [26], [41], [51]. Coarse-grained weight sparsity was further proposed to mitigate the indexing overhead and irregular access [13], [32], [48], [54]. While we do not use pruned models in our evaluation, dual-module processing can be combined with other model compression techniques by taking compressed layers as accurate modules.

Other studies propose dynamic redundancy elimination base on certain criteria. One scenario is leveraging *ReLU*-induced activation sparsity as either input sparsity detection [3], [5], [20], [23], [30], [39], [50], [54], [55] or output sparsity prediction [4], [7], [34], [45]. Exploiting *ReLU*-induced activation sparsity is only a special case of our dual-module processing. The approximate results are useful in the insensitive regions instead of only for prediction and then discarded. Besides neuron-wise computation skipping, channel-wise feature map suppressing and gating can reduce computations leveraging the multi-channel feature of CONV layers [19], [27], [48]. However, those studies are limited to saving computations of CONV layers while our design can also save memory access of FC and RNN layers. Computation skipping in RNNs is also proposed leveraging the particular cell structure and the temporal input similarity [6], [40], [52]. However, those methods depend on certain applications and lack of evaluation on NLP tasks such as machine translation.

VII. Conclusion

In this paper, we present an algorithm-architecture co-design to boost the execution efficiency of DNNs. Firstly, our dual-module algorithm uses lightweight approximate modules to compute insensitive activations and seeks to accurate modules to compute sensitive activations with skipped computations and data accesses. Secondly, our DUET design with specialized and decoupled Executor and Speculator supports balanced execution and memory accesses reduction. Compared with standard single-module processing, DUET can achieve 2.24x performance speedup and 1.97x energy efficiency improvement.

References

[1] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[2] D. Achlioptas, "Database-friendly random projections," in *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2001, pp. 274–281.

[3] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu, and T. Delbruck, "Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE transactions on neural networks and learning systems*, no. 99, pp. 1–13, 2018.

[4] V. Aklaghi, A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and R. Gupta, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks." ISCA, 2018.

[5] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 1–13.

[6] V. Campos, B. Jou, X. G. i Nieto, J. Torres, and S.-F. Chang, "Skip RNN: Learning to skip state updates in recurrent neural networks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=HkwVAXyCW

[7] S. Cao, L. Ma, W. Xiao, C. Zhang, Y. Liu, L. Zhang, L. Nie, and Z. Yang, "Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 216–11 225.

[8] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 247–257, 2010.

[9] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Computer Vision (ICCV), 2015 IEEE International Conference on*. IEEE, 2015, pp. 2722–2730.

[10] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[11] 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.

[12] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[13] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 189–202.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *Proceedings of NAACL-HLT*, 2019.

[15] 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, no. 3. ACM, 2015, pp. 92–104.

[16] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, p. 115, 2017.

[17] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops 2011)*. IEEE, 2011, pp. 109–116.

[18] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 751–764, 2017.

[19] X. Gao, Y. Zhao, ukasz Dudziak, R. Mullins, and C. zhong Xu, "Dynamic channel pruning: Feature boosting and suppression," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=BJxh2j0qYm

[20] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York,

NY, USA: ACM, 2019, pp. 151–165. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358291

[21] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

[22] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 75–84.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 243–254.

[24] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[25] 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*, 2016, pp. 770–778.

[26] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 674–687.

[27] W. Hua, Y. Zhou, C. De Sa, Z. Zhang, and G. E. Suh, "Boosting the performance of cnn accelerators with dynamic fine-grained channel gating," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 139–150. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358283

[28] 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[29] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[30] D. Kim, J. Ahn, and S. Yoo, "A novel zero weight/activation-aware hardware architecture of convolutional neural network," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 1462–1467.

[31] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 380–392.

[32] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," *arXiv preprint arXiv:1811.04770*, 2018.

[33] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems*, 2017, pp. 2181–2191.

[34] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, "Predictivenet: an energy-efficient convolutional neural network via zero prediction," in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 1–4.

[35] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 369–381.

[36] L. Liu, L. Deng, Z. Chen, Y. Wang, S. Li, J. Zhang, Y. Yang, Z. Gu, Y. Ding, and Y. Xie, "Boosting deep neural network efficiency with dual-module inference," in *International Conference on Machine Learning (ICML)*, 2020.

[37] L. Liu, L. Deng, X. Hu, M. Zhu, G. Li, Y. Ding, and Y. Xie, "Dynamic sparse graph for efficient deep learning," in *Seventh International Conference on Learning Representations (ICLR)*, 2019.

[38] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5058–5066.

[39] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: a déjà vu-free differential deep neural network accelerator," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 134–147.

[40] D. Neil, J. H. Lee, T. Delbruck, and S.-C. Liu, "Delta networks for optimized recurrent network computation," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 2584–2593.

[41] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 27–40.

[42] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5456–5464.

[43] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 267–278.

[44] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint*, 2017.

[45] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, "Prediction based execution on deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 752–763.

[46] Y. Wang, M. Huang, X. Zhu, and L. Zhao, "Attention-based lstm for aspect-level sentiment classification," in *Proceedings of the 2016 conference on empirical methods in natural language processing*, 2016, pp. 606–615.

[47] Y. Xu, Y. Wang, A. Zhou, W. Lin, and H. Xiong, "Deep neural network compression with single and multiple level quantization," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[48] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 548–560.

[49] 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 the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[50] J. Zhang, C. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "Snap: A 1.67 21.55tops/w sparse neural acceleration processor for unstructured sparse deep neural network inference in 16nm cmos," in *2019 Symposium on VLSI Circuits*, 2019, pp. C306–C307.

[51] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 20.

[52] X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu, "Towards memory friendly long-short term memory networks (lstms) on mobile gpus," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 162–174.

[53] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," in *International Conference on Machine Learning*, 2019, pp. 7543–7552.

[54] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 15–28.

[55] J. Zhu, J. Jiang, X. Chen, and C.-Y. Tsui, "Sparsenn: An energy-efficient neural network accelerator exploiting input and output sparsity," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 241–244.