STONNE: A Detailed Architectural Simulator for Flexible Neural Network Accelerators

Francisco Muñoz-Martínez, Manuel E. Acacio
Universidad de Murcia
Murcia, SPAIN
{francisco.munoz2,meacacio}@um.es

José L. Abellán
Universidad Católica San Antonio
Murcia, SPAIN
jlabellan@ucam.edu

Tushar Krishna
Georgia Institute of Technology
Atlanta, Georgia, USA
tushar@ece.gatech.edu

***Abstract*—The design of specialized architectures for accelerating the inference procedure of Deep Neural Networks (DNNs) is a booming area of research nowadays. First-generation rigid proposals have been rapidly replaced by more advanced flexible accelerator architectures able to efficiently support a variety of layer types and dimensions. As the complexity of the designs grows, it is more and more appealing for researchers to have cycle-accurate simulation tools at their disposal to allow for fast and accurate design-space exploration, and rapid quantification of the efficacy of architectural enhancements during the early stages of a design. To this end, we present STONNE (Simulation TOol of Neural Network Engines), a cycle-accurate, highly-modular and highly-extensible simulation framework that enables end-to-end evaluation of flexible accelerator architectures running complete contemporary DNN models. We use STONNE to model the recently proposed MAERI architecture and show how it can closely approach the performance results of the publicly available BSV-coded MAERI implementation. Then, we conduct a comprehensive evaluation and demonstrate that the folding strategy implemented for MAERI results in very low compute unit utilizations (25% on average across 5 DNN models) which in the end translates into poor performance.**

***Index Terms*—Deep Neural Networks, Inference process, Simulation, Flexible accelerator architecture, Performance.**

I. Introduction

Contemporary Deep Neural Networks (DNNs) are organized as a large number of layers (e.g., convolution and fully-connected), each composed of a large set of neurons. Depending on the type of layer, each neuron performs a simple weighted addition of (or some of) the values obtained in the preceding layer. During the inference phase the already trained DNN model is used to make a prediction.

The difficulty in processing these workloads does not stem from the type of operations to be performed (simple Multiply-Accumulate operations or MACs with 8-bit operands might suffice [1]) but from the vast amount of MAC operations involved in the inference procedure of DNNs (e.g., 3.9 billions in ResNet-50). As a result, typical DNN layers are excessively large to be executed by an edge-computing accelerator in a single step. So, when processing a single layer, their neurons are grouped in smaller tiles that define the pattern in which the neurons' inputs, weights, and intermediate outputs (partial sums or psums) are delivered and reused within the accelerator's functional units. This pattern is called dataflow and determines the energy efficiency of an accelerator architecture when processing a certain DNN layer [1]. First-generation DNN inference accelerators focused their designs on fixed-size clusters of multipliers-and-accumulate units interconnected by means of a fixed on-chip network fabric specifically tailored to efficiently support a particular dataflow. Unfortunately, the inability of these designs to adapt well to the varying morphology among contemporary DNNs, and more importantly, among different layers within the same DNN (varying layer dimensions and types [2]), limits their potential advantages (low compute unit utilization and low reuse of data that is translated into low energy efficiency). To overcome this limitation, recent proposals such as FlexFlow [3], MAERI [4] and SIGMA [5] advocate using flexible DNN accelerator fabrics, which can be reconfigured to efficiently map different dataflows and partitions through the creation of dynamic-size clusters in the same hardware substrate. Of course, this flexibility comes at the cost of increased accelerator complexity that urges for a more exhaustive design-space exploration for fine tuning before building the particular ASIC-based or FPGA-based DNN accelerator prototype.

Traditionally, architectural simulators have become an integral part of the computer architecture research and design process, since they permit fast and accurate design-space exploration and rapid quantification of the efficacy of architectural enhancements in the early stages of a design, and have been extensively used during the design process of CPU and GPU architectures [6], [7]. However, and quite surprisingly, the same has not taken place until now for inference accelerator architectures. To the best of our knowledge, there is still no detailed open-source architectural simulator for extensive and accurate design-space exploration of next-generation flexible DNN inference accelerators. In this work we present STONNE (Simulation TOol of Neural Network Engines), a cycle-accurate, highly-modular and highly-extensible simulator aimed to bridge this gap.

Table I shows a qualitative comparison of STONNE with respect to contemporary publicly available simulators for DNN inference accelerators. As we can see, unlike STONNE, existing simulators were originally developed for first-generation DNN accelerators and do not give support for simulating flexible DNN architectures. This is at least in part because it is not possible, without significant "heavy lifting" to extend these simulation tools to support next-generation DNN accelerators,

TABLE I: State-of-the-art simulators for DNN architectures.

	End-to-End evaluation	Easy to Extend	Flexible architecture
MAGNet [8] DNNBuilder [9]	✗	✗	✗
MAERI BSV [10]	✗	✗	✓
TVM [11]	✓	✗	✗
SCALE-Sim [12] MAESTRO [13]	✗	✓	✗
SMAUG [14]	✓	✓	✗
STONNE	✓	✓	✓

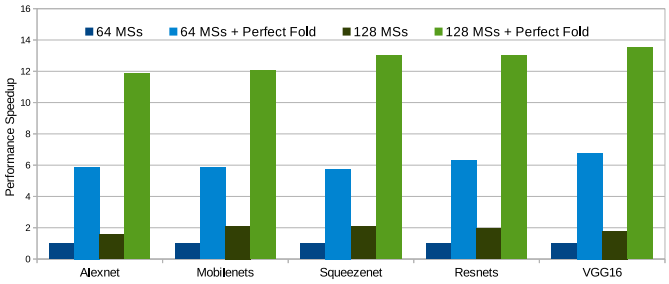

Fig. 1: Performance results for simple design-space exploration in a simulated MAERI accelerator: speedup obtained by doubling compute resources (128 multipliers) and speedup that would be obtained for an ideal implementation of folding (`+ Perfect Fold`).

since they were tailored to specifically simulate a certain type of rigid architecture (e.g., a systolic array as in Google TPU [15]). Among all the alternatives, only the MAERI BSV model and SMAUG claim to model early flexible architectures. However, none of them really allows neither for efficient design-space exploration nor for rapid quantification of the impact that modifications in the architecture of a flexible inference accelerator would have on both performance and energy consumption.

To motivate how STONNE can be used for research in the design and implementation of flexible next-generation DNN accelerators, we consider the case of the state-of-the-art MAERI architecture (further details in Section III). In particular, we examine how performance is affected by both the number of multipliers in the MAERI architecture (64 and 128 multipliers) and the strategy currently being used to handle folding[1]. The latter is done through the execution of five complete DNN models: AlexNet [16], MobileNets [17], Squeezenet [18], Resnet-50 [19] and VGG-16 [20]. To configure the MAERI model, we use the best tile configuration for every layer of every DNN by using mRNA [21], the search exploration tool to configure the MAERI architecture. Additionally, we assume perfect bandwidth (no contention) between memory and the MAERI's processing elements (a fabric of multipliers with a tree-based reduction tree of adders to efficiently map MAC operations onto the hardware substrate).

[1]Folding is utilized when a neuron needs more multiplication operations than the number of multiplier units available in hardware. Then, the neuron is "folded" to be processed in several sequential steps and partial results should be accumulated and taken at inter-steps boundaries.

As we can see in Figure 1, when the number of multipliers in MAERI doubles (`128 MSs`), the performance improvement almost achieves ideal scaling (an average of 1.88×). This significant speedup is obtained by doubling the amount of hardware resources for computing MAC operations, which might not be a feasible design decision to increase performance in highly-constrained low-end edge-computing devices. A much more cost-effective solution to increase performance in MAERI is optimizing the implementation of folding. Ideal folding with both 64 and 128 multipliers (`64/128 MSs + Perfect Fold`) reports impressive performance benefits. In particular, average speedups of 6.1× and 12.6× with respect to the baseline (`64 MSs`). In Section IV we will dig into the details and explain the reasons behind this performance bottleneck.

We see the following contributions in this work:

- We present for the first time STONNE, a cycle-accurate architectural simulator for flexible DNN inference accelerators that features high modularity, high configurability and end-to-end evaluation.
- We model, configure, implement and validate a MAERI architecture in STONNE by using end-to-end evaluation with state-of-the-art DNN models. We validate our implementation against a publicly available real BlueSpec SystemVerilog (BSV)-based prototype [10]. We obtain an average difference of just 15% in terms of total executed cycles and we identify where this difference comes from.
- We conduct a comprehensive characterization to illustrate the key benefits of STONNE. In particular, we demonstrate the significant performance bottleneck of the BSV-based prototype of MAERI when implementing folding. By using the statistics reported by STONNE, we identify that the main root of such a performance bottleneck lies in the current design of the MAERI's reduction network (RN).

The rest of the work is organized as follows. First, Section II explains the organization of STONNE. After that, Section III describes the family of flexible accelerator architectures that STONNE simulates. Section IV demonstrates the accuracy and potential of the tool and evaluates a performance bottleneck found in a typical flexible DNN architecture. Finally, Section V presents the main conclusions of this work and outlines the ongoing work.

II. STONNE FRAMEWORK

STONNE is a highly-configurable cycle-accurate next-generation DNN accelerator simulator. The simulator has been developed in C++ and allows for end-to-end evaluation as it is connected with Caffe DL framework. Current version of the simulator can fully execute any relevant DNN model and, as the GRASP and SOLID programming principles of object-oriented design have been followed to build the simulator, STONNE is highly extensible and can be easily modified to support any particularity of any DNN model (e.g., different type of layers). In addition, it can be easily extended to

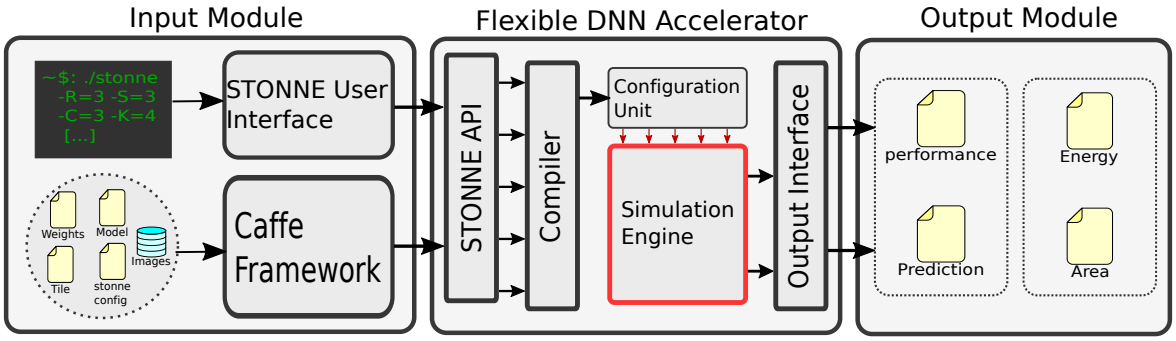

Fig. 2: High-level diagram of the STONNE framework.

model different architectures of flexible DNN accelerators, tile configuration mappings and dataflows.

A. STONNE Organization

Figure 2 illustrates a high-level diagram of STONNE with the three major components involved in the end-to-end simulation flow. First, the **Input Module** determines the layer to be run, creates an instance of the simulator and loads the parameters of the layer and the initial inputs and weights onto the architecture. Once the architecture has been configured, the **Flexible DNN Accelerator** module carries out the detailed cycle-by-cycle simulation of the layer, collecting statistics during the process. Once the simulator finishes, the **Output Module** takes in the values of the counters collected by the simulator and produces different files with the statistics of the execution.

Next, we further describe the details of each module:

(1) Flexible DNN Accelerator: This constitutes the principal block of STONNE (see the central block in Figure 2), and it is mainly composed of the modeled architecture of the flexible DNN accelerator (`Simulation Engine`) to simulate, whose details are further explained in Section III. The accelerator is interfaced by means of the `STONNE API` that allows users to create an instance of the simulator according to a hardware configuration file, load the layer and tile parameters, and load the weights and inputs onto the memory of the simulator. Once all these parameters have been defined, the `Compiler` generates all the control signals that configure the architecture through the `Configuration Unit`. Then, the simulator starts the execution and the results and statistics being collected are reported through an output interface.

(2) Input Module: Due to the flexibility that the `STONNE API` provides, the simulator can be fed easily using any of the well-known DL frameworks already available. In this work, we have modified the Caffe DL framework (see left block in Figure 2) to connect it to the simulator so that it is able to run an instance of the `Simulation Engine` (e.g., MAERI) transparently to the user. This way, a Caffe user just needs to select the typical *.caffemodel* file with the weights, choose the inputs[2] (e.g., a set of images) and briefly modify each layer block defined in the *.prototxt* DNN model file to specify the layers to be simulated, the path of the hardware configuration file with the parameters of the architecture to simulate (e.g., the number of multipliers) and the tile configuration for every layer. After Caffe is launched with those defined parameters, it takes the control and creates an instance of STONNE. Then, Caffe drives a layer-by-layer execution, sending the configuration parameters for every layer, copying the weights and the inputs of that layer onto the simulator memory, and copying back the results after the simulator finishes and produces the statistics file. This process is repeated for every layer until the end of the execution, producing the final prediction for each input (thus performing the whole inference process).

Furthermore, since Caffe requires a more complicated installation and use, apart from this mode of execution, we have also enabled the `STONNE User Interface` that facilitates the execution of STONNE. Through this mode, the user is presented with a prompt to load any layer and tile parameters onto a selected instance of the simulator and run it with random weights and input values. This mode allows for faster executions and hence facilitates faster prototyping and debugging.

(3) Output module: Once a simulation for a certain layer has been completed, this module is used for reporting simulation statistics such as performance, compute unit utilization, number of accesses to SRAM, wires and FIFOs, etc. Besides, this output module also reports the amount of energy consumed and the on-chip area required by the simulated architecture. Currently, we are extending the simulator to provide such area and energy numbers. Moreover, since the STONNE simulator is a back-end compute platform of Caffe, it also outputs the result of the prediction when running a certain DNN model for certain input data.

B. Flexible DNN Accelerator Architecture

As previously commented, STONNE emerges as the first cycle-accurate simulation tool that enables exploration of the design space of flexible accelerator architectures. In this section, we explain the general flexible DNN inference accelerator architecture that is implemented as baseline in STONNE and whose high-level diagram is shown in Figure 3.

[2]Throughout this work, we use STONNE to characterize the inference process of several contemporary DNN models aimed to image classification.

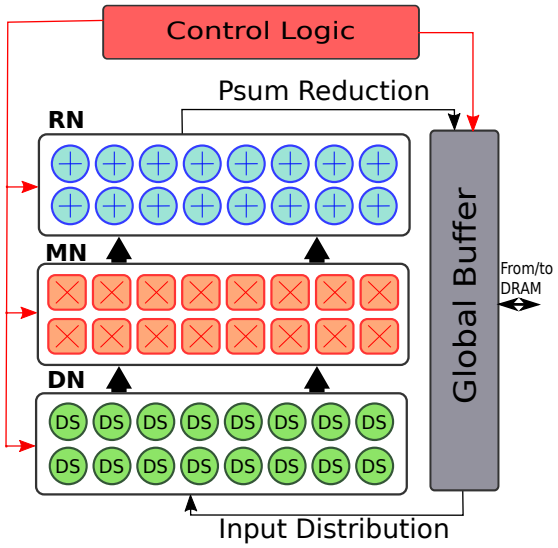

Fig. 3: Overview of the general flexible DNN inference accelerator architecture that is implemented as baseline in STONNE.

STONNE is equipped with all the major basic components of any recently proposed next-generation DNN accelerator [4], [5]: i) A set of **Multipliers and adders** to carry out the multiply-and-accumulate operations required by each of the NN's neurons. ii) A **Memory hierarchy** composed of local storage, some on-chip/on-package global storage, and some connectivity to off-chip DRAM memory. Local storage is made up of buffers or registers that are typically private to certain groups of multipliers and adders, and are used to temporarily hold both input data (weights, activations and psums) and output results. On-chip/on-package global storage is typically shared by all multipliers and adders in the accelerator, and can be composed of either a single memory element (i.e., the Global Buffer, GB) or a hierarchy of different levels of memory (e.g., a cache hierarchy). These memory elements can be software- or hardware-managed. Off-chip DRAM memory can be private to the DNN accelerator or shared with a host compute platform such as a CPU. And finally, iii) **Control logic** that is used to reconfigure the connectivity among the compute and/or memory elements within the previous two components, thereby the architecture of the DNN inference accelerator can be made flexible and adaptable to efficiently map any compute/memory partitions and dataflows.

All the on-chip components are interconnected by using a three-tier network fabric composed of a Distribution Network (DN), a Multiplier Network (MN), and a Reduce Network (RN). These networks must accomplish certain requirements in order to provide the flexibility that the simulator promises. First, to compute all the MAC operations of a certain DNN layer, the DN distributes the required weights, activations or partial sums from the GB towards the MN. To enable all types of dataflows, the DN must provide support for unicast, multicast and broadcast data delivering. After the distribution, the multipliers at the MN carry out the multiplication operations generating the operands of the partial sums to be accumulated, and finally the RN network is equipped with adders that implement the required accumulations. Again, to enable dataflow flexibility, the RN must be capable of reducing any number of multiplier clusters of any size simultaneously.

III. MAERI Architecture

In this work, we have created an instance of the previous general flexible DNN inference accelerator architecture that corresponds to the MAERI architecture [4]. This instance will be utilized for description and evaluation in the rest of the sections of this work. Note that, as our simulator is highly-modular, highly-configurable and highly-extensible by design, we can easily modify any of the above components to model any other type of accelerator architecture.

An overview of the BSV-coded implementation of the MAERI architecture [10] that we have faithfully modeled in STONNE is shown in Figure 4. At a high-level, there are two tree-based topologies that implement both the DN and RN networks, and 1D-mesh point-to-point network for the MN network. The Global Buffer is called Prefetch Buffer (PB) in MAERI. The PB needs arbitration at the write ports. The figure also illustrates an example of mapping three Virtual Neurons (VNs) on five different multipliers each. A VN is the most basic primitive in MAERI and is, in essence, a configurable cluster of multipliers used to execute the multiply operations in a certain output neuron. In principle, any VN could be set up using any number of multipliers.

A. Flexible Network Fabric

Next, we detail the implementation of the three networks (DN, MN and RN) in the MAERI architecture.

Distribution Network (DN): As it is shown in Figure 4, the DN in MAERI is a binary-tree-based network topology that is replicated as many times as the number of read ports available in the PB. In the figure, the number of read ports is four so there are four sub-trees. Our simulator allows to configure the number of ports and sub-trees according to the user requirements. Each of these sub-trees is in charge of delivering the weights, inputs and psums from the PB to a different group of multipliers sited in the MN (explained below) through multicast, unicast, or broadcast operations. Each node of the DN is just a bufferless low-cost Distribution Switch (DS) that selects whether to send the input to one or both outputs using a bit vector that is set by the input source. Due to the simplicity of the DSs, the DN can provide single-cycle traversals from the PB to the MN for every piece of data.

Multiplier Network (MN): This network is conformed by a set of Multiplier Switches (MSs) that can be configured to act as either forwarders or multipliers. The forwarding configuration is used to forward psums from the PB to the RN so that folding can be supported, whereas the multiplier configuration mode is utilized to compute a multiplication

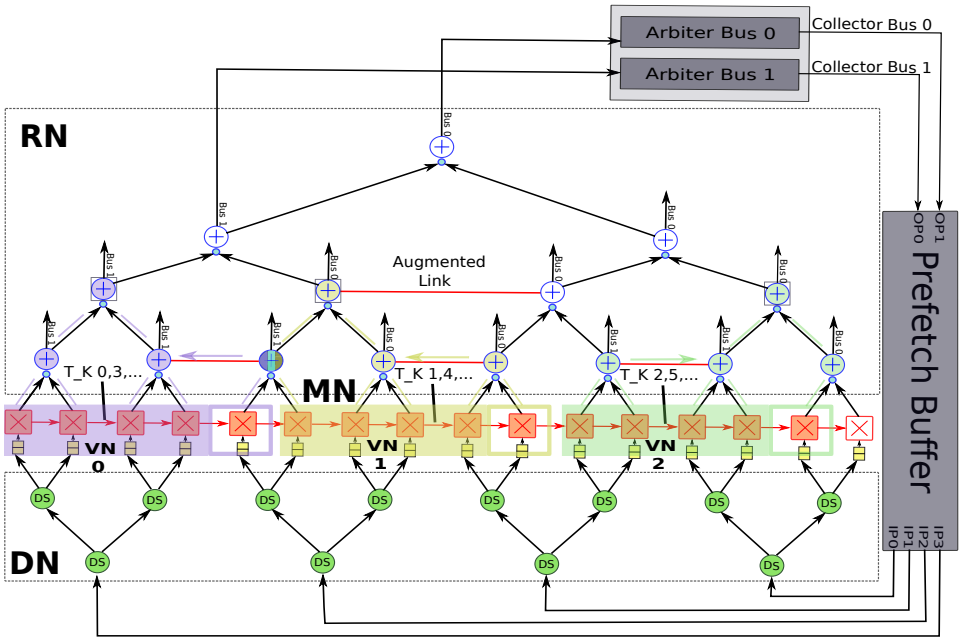

Fig. 4: Overview of the MAERI architecture. There are three Virtual Neurons (VN 0, 1 and 2) mapped on five different multipliers each. The fifth multiplier in each VN is the one that would be used to enable folding.

between a weight and an input value. In case folding is needed (further details in Section III-B), the architecture would need to allocate one extra MS for each VN to perform the forwarding of the psums calculated in the previous iterations of the same VN.

Reduce Network (RN): The RN is implemented as a tree-based topology augmented with one adder unit per node, a well-known layout for efficiently executing reduction operations. The tree structure is augmented with links between the nodes of the same level (horizontal links) that do not share the same parent. That is why the RN in MAERI is called *Augmented Reduction Tree* (ART). More specifically, each node is a configurable Adder Switch (AS) that can be statically configured as either *2:1 ADD*, *3:1 ADD*, *1:1 ADD plus 1:1 forward*, or *2:2 forward*. This configurable capability within each RN node along with the augmented links are key aspects to enable high flexibility in MAERI, as they permit flexible support of multiple and non-blocking virtual reduction trees over a single physical tree hardware substrate.

As it is shown in Figure 4, each AS is connected to a different collector bus (there are two CBs in the figure). Each CB is used to write partial or final outputs in the PB. Note that, the number of CBs must be equal to the number of write ports in the PB. Since there are usually more ASs than collector buses, write requests often produce conflicts that will be managed by means of a bus arbiter module, which is shown at the top of the figure. Clearly, the higher the number of CBs and write ports the higher performance benefits that can be obtained due to lower network contention. However, this comes at the cost of higher energy consumption and on-chip area requirement. In STONNE, we can easily study the impact of the number of output ports, CBs and their corresponding arbiters at design time according to what is needed.

B. Supported DNN mappings in MAERI

MAERI can be configured at execution time to run any number of VNs of arbitrary size. Basically, the DNN tile mapping taxonomy is similar to the one mentioned in [21]. First, a layer is defined with 7 parameters as *Layer(R, S, C, K, N, X', Y')* where R and S are the number of rows and columns in a filter respectively, C is the number of channels, K is the number of filters, N is the batch size, and X' and Y' are the number of rows and columns in the output respectively. Additionally, we have added in STONNE a new parameter (G) that allows MAERI to support factorized convolutions.

This way, we define a tile as *Tile(T_R, T_S, T_C, T_G, T_K, T_N, T_X', T_Y')*, where $T\_R \times T\_S \times T\_C$ parameters are a subset of the filter dimensions, and therefore, what defines the size of the VN. Similarly, $T\_G \times T\_K \times T\_N \times T\_X' \times T\_Y'$ parameters represent the subset of number of groups, filters per group, input fmaps, and output dimensions, respectively, thus defining the number of VNs that are mapped onto the architecture. Note that, if the size of a VN cannot hold the entire filter size (i.e., $(T\_R/R \times T\_S/S \times T\_C/C) > 1$), the architecture will have to enable what is called *folding* as it will be necessary to iterate over the same VN, storing partial sums in some temporal storage (the prefetch buffer in the case of MAERI) and sending them back to the VN to be reduced with the calculated in the subsequent iteration.

An example of tile mapping is depicted in Figure 4, where we have a Tile(T_R=2, T_S=2, T_C=1, T_G=1, T_K=3, T_N=1, T_X'=1, T_Y'=1) mapped into a MAERI instance and folding is enabled. Notice that with this tile shape, the

TABLE II: Set of layers executed to validate the MAERI architecture simulated with STONNE.

Name	R	S	C	G	K	N	X	Y
TINY	3	3	6	1	6	1	5	5
LATE_SYNTHETIC	3	3	20	1	20	1	5	5
EARLY_SYNTHETIC	3	3	6	1	6	1	20	20

number of mapped VNs is 3 (T_K) and the VN size is 4 ($T\_R \times T\_S$). As folding is needed, each VN would require one extra multiplier that would act as a forwarder (see the fifth multiplier allocated for each VN).

IV. Validation and Evaluation

A. Validation

To validate the MAERI architecture that we have simulated with STONNE against a real hardware implementation, we use the original BSV MAERI design [10]. For the validation process, we run STONNE using its direct user interface (the `STONNE User Interface` in Figure 2). Recall that this execution mode allows for easy configuration of the Simulation Engine (to model a MAERI architecture in this case), DNN layer configuration and memory/compute partition tiles.

Since the BSV MAERI version does not have the large flexibility of our cycle-accurate architectural simulator, which can model almost any combination of the parameters of the flexible accelerator (e.g., number of MSs, number of trees in DN, number of input/output ports in the Prefetch Buffer), we are heavily constrained in the number of validation experiments we can carry out. This way, we have configured both STONNE and BSV versions with 32 MSs and 4 DN/RN bw parameters. In other words, 4 input/output ports in the Prefetch Buffer, 4 trees in the DN (8 MSs per tree) and 4 Collector Buses. In addition, the BSV version can only execute the three different types of layers listed in Table II, with the tile shape *Tile(T_R=3, T_S=3, T_C=1, T_G=1, T_K=1, T_N=1, T_X'=3, T_Y'=1)*.

For functional validation of the STONNE-based MAERI simulator, we carry out an exhaustive head-to-head comparison between the STONNE and the BSV versions in terms of the output values produced by every single executed DNN layer in both platforms. After validation of all possible supported DNN layers with the above tile configuration, we can confirm that our implementation of the MAERI architecture is correct.

To evaluate the accuracy of timing simulation, Figure 5 shows a comparison of the total number of executed cycles reported by the BSV MAERI and STONNE versions after running the three types of layers supported by the BSV version (TINY, LATE_SYNTHETIC and EARLY_SYNTHETIC). As we can see, our implementation of MAERI in STONNE shows an average difference in terms of total executed cycles of just 15% as compared to the real BSV MAERI version (from 11% to 19%). These results demonstrate that STONNE closely mimic the characteristics of the hardware version. Additionally, after an in-depth analysis of the BSV MAERI code, we have found out that the small performance difference is mainly due to the testbench module used to run the BSV

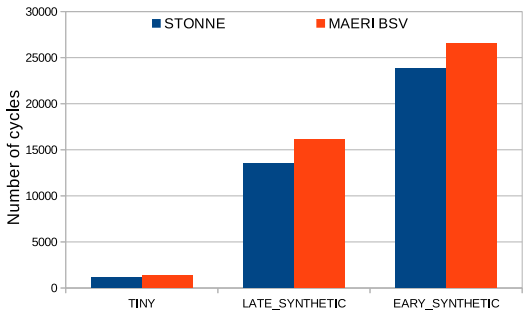

Fig. 5: Comparison in terms of number cycles between BSV MAERI and STONNE for the layers described in Table II.

MAERI implementation. In particular, we discovered that the testbench does not fully leverage all the available input ports in parallel to feed the architecture. That is the reason why our simulator achieves slightly better performance numbers (lower amount of clock cycles) for the two largest layers.

Finally, we have also validated the feasibility of the STONNE framework for conducting end-to-end evaluations (see Table I). To do so, we have run five DNN models (AlexNet, MobileNets, Squeezenet, Resnet-50 and VGG-16) with a test set of 50 images from ImageNet, and for every image we have compared the score digits (output of the last fully-connected layer of each DNN) and predicted label reported by Caffe DL when running on a real back-end (CPU), with those obtained for the executions on the simulated MAERI architecture. We observed exactly the same results for all the images in both cases.

B. Performance analysis of MAERI with real DNNs

The results previously shown demonstrate the correctness of the MAERI architecture being simulated by STONNE. Now, we show some of the key benefits of STONNE in providing detailed performance analysis when running real DNN models through its execution mode that enables end-to-end evaluation, i.e., the execution of real DNN models driven by a DL framework (Caffe in the current version of our simulation framework). For the performance analysis, we simulate a baseline MAERI implementation with 64 MSs. On top of this simulated architecture we execute the five DNN models already mentioned (AlexNet, MobileNets, Squeezenet, Resnet-50 and VGG-16). For every layer, we obtain the optimal tile configuration reported by the mRNA tool and reconfigure the MAERI architecture accordingly before execution. Due to the large number of tile configurations needed for running all the layers in all the DNN models, we omit all the different tile configurations utilized for the sake of brevity. Note that, the BSV MAERI implementation cannot be used to carry out this design-space exploration as it only supports a particular tile configuration as commented in Section IV-A.

We have also performed simulations with 128 MSs but have not included the results since we have observed the same trends as for 64.

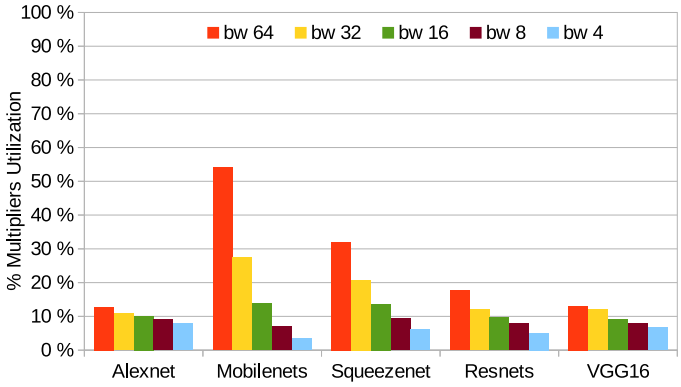

Fig. 6: Percentage of utilization in a MAERI-based architecture simulated using STONNE and using 64 multipliers.

One of the key aspects that reflects the efficiency in processing a particular DNN layer through a DNN inference accelerator is the resulting utilization of its compute resources, which depends on both the number of resources assigned to the configured tiles (theoretical utilization) and how these resources are actually leveraged.

This way, guided by this observation, we first conduct a comprehensive analysis of the resources that are mapped when processing the five DNN models in the simulated MAERI architecture with the baseline 64-MS configuration. Table III shows the number of MSs that are mapped depending on the theoretical VN size (first column). Since our experiments reveal that folding is necessary for 98% of the computation time, we assume that an extra MS is always needed to be able to map all the tiles (see the discussion in Section III-B) and therefore the real VN size is one unit larger (second column). We always map as many neurons as possible (third column). The average frequency of every VN mapping configuration across the five DNN models is also shown in the fourth column. Note that, ordering the table data by frequency helps us understand the most common VN mapping configurations and the amount of idle resources left in each case.

As we can see in the fifth column of the table, we are far from reaching the near-100% theoretical MS utilization that these flexible accelerator architectures for inference at the edge should approach. Specifically, the utilization of the MSs for the most frequent VN mapping configurations is very low. For example, a VN with a theoretical size of 36 MSs (the most typical VN configuration), and thus requiring a total of 37 MSs (the extra MS needed with this folding implementation), would leave 27 MSs unused, which results in a theoretical utilization of just 58%.

On the other hand, as it is explained above, this theoretical utilization is just the number of MSs that are mapped according to the tiles that have been used. However, the effective utilization depends not only on this, but also, on the capacity of leveraging all these mapped resources. Figure 6 shows, for each one of the 5 DNN models considered in this work, the percentage of effective utilization for the 64 MSs as

TABLE III: Theoretical MS utilization rate when mapping realistic VN sizes onto a 64-wide MAERI configuration.

Theoretical VN Size	Real VN Size	# of VNs	Frequency	Theoretical MS Utilization
36	37	1	83%	58%
32	33	1	10%	52%
50	51	1	1%	78%
49	50	1	1%	76%

TABLE IV: Configuration tiles used to run AlexNet DNN.

Name	T_R	T_S	T_C	T_G	T_K	T_N	T_X'	T_Y'
CONV1	11	11	1	1	1	1	1	1
CONV2	5	5	2	1	2	1	1	1
CONV3	3	3	4	1	3	1	1	1
CONV4	3	3	6	1	2	1	1	1
CONV5	3	3	6	1	2	1	1	1
FC6	1	12	1	1	8	1	1	1
FC7	1	16	1	1	4	1	1	1
FC8	1	8	1	1	10	1	1	1

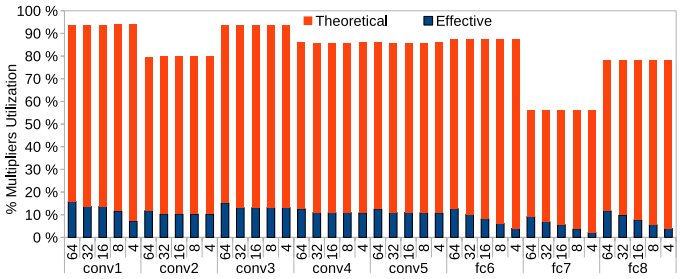

Fig. 7: Percentage of multipliers (MS) utilization of the baseline MAERI architecture simulated in STONNE with 64 MSs after execution of AlexNet. We compare the reported utilization (Effective) with the maximum achievable (Theoretical).

the input/output bandwidth is varied from 64 elements/cycle (maximum possible bandwidth) to 4 (see *bw N* in the figure, where *N* equals 64, 32, 18, 8 and 4).

As we can see in Figure 6, MS utilization for all the five DNN models is particularly low in almost all cases: average utilization[3] of 25%, 16%, 11%, 8% and 5% for an input/output bandwidth of 64, 32, 16, 8 and 4 elements/cycle, respectively. Obviously, the lower the bandwidth at the PB's input and output interfaces, the higher the under-utilization of the 64 multipliers, which will be idle-waiting whereas contention arises at the DN and RN networks due to the limited number of input and output ports in the PB, respectively. However, even when considering the best hardware configuration (where the number of input and output ports equals the number of multipliers, i.e. bandwidth of 64 elements/cycle), multiplier utilization results are also far from a near-60%-utilization that the theoretical calculations previously discussed promised. In fact, utilization rate of multipliers is extremely low in both Alexnet and VGG-16, not even surpassing 10%.

We noticed that this performance issue becomes more evident when analyzing the percentage of multiplier utilization

[3]Note that we count the multipliers that are being used as forwarders as effective utilization.

for single layers. Figure 7 shows these results for every layer in the case of AlexNet (the DNN model that reports the lowest utilization). Table IV shows the different tile configurations employed to run every one of the layers in AlexNet. In the figure, we overlap the effective multiplier utilization rate (see blue bars) with the maximum theoretical multiplier utilization that could be achievable according to the tile configurations. Clearly, multiplier utilization is extremely low compared with the theoretical value that could be achieved. Even in the configurations with no bandwidth restrictions (`bw 64`), the maximum utilization achieved is 15% (CONV1 and CONV3) with some layers experimenting an utilization below 9% (FC7). This significant difference between the theoretical and effective utilization is, as explained in Section III, due to the dependency between the MN and the RN introduced by the psum when folding is used. This, impedes to iterate over the same output neuron in a pipeline manner, hurting the utilization of the mapped resources and significantly degrading performance as was shown in Figure 1. This demonstrates the importance of properly supporting folding in a flexible accelerator architecture, as well as the need of a much more efficient implementation that allows for significant increase in the utilization rate of the compute resources.

V. Conclusions

In this work we have presented STONNE, a cycle-accurate, highly-modular and highly-extensible simulation framework that enables end-to-end evaluation of flexible accelerator architectures running complete contemporary DNN models. We have used STONNE to faithfully simulate a MAERI-based architecture with an average difference of only 15% in total executed cycles. In addition, we demonstrate that the folding strategy implemented by the accelerator architecture is extremely inefficient, as it lowers compute unit utilization to an average of 25% across 5 DNN models, which results into a maximum performance degradation of up to 610%.

As part of our ongoing work, we are currently exploring the design of a novel reduction network capable of providing more efficient support to folding by avoiding the need to re-distribute the partial sums again once they reach the Prefetch Buffer. Additionally, we are extending STONNE to also report results of on-chip area and energy consumption based on the recently proposed Accelergy [22] framework.

Acknowledgments

The authors wish to thank Hyoukjun Kwon for for his clarifications on certain technical aspects related to MAERI. This work has been supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant "RTI2018-098156-B-C53". Francisco Muñoz-Martínez is supported by fellowship 20749/FPI/18 from Fundación Séneca, Agencia Regional de Ciencia y Tecnología de la Región de Murcia.

References

[1] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *arXiv preprint arXiv: 1703.09039v2 (2017)*, Aug. 2017.

[2] Y.-H. Chen, J. S. Emer, T. Krishna, 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, Jan. 2017.

[3] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 553–564, May 2017.

[4] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 461–475, Mar. 2018.

[5] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Mar. 2020.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, 2011.

[7] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, p. 197–209.

[8] A. Venkatesan, Y. S. Shao, M. Wang, and J. C. et al., "Magnet: A modular accelerator generator for neural networks," *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2019.

[9] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. mei Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2018.

[10] "Maeri code v1," https://github.com/hyoukjun/MAERI.

[11] T. Chen, T. Moreau, Z. Jiang, and L. Z. et al., "TVM: An automated end-to-end optimizing compiler for deep learning," *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 579–594, Oct. 2019.

[12] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv: 1811.02883v1 (2019)*, Feb. 2019.

[13] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn-dataflows: A data-centric approach," *arXiv preprint arXiv: 1805.02566 (2019)*, May 2019.

[14] S. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, "SMAUG: End-to-end full-stack simulationinfrastructure for deep learning workloads," *arXiv preprint arXiv: 1912.04481v2 (2019)*, Dec. 2019.

[15] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *44th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2017, pp. 1–12.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *International Conf. on Neural Information Processing Systems (NIPS)*, pp. 1106–1114, Dec. 2012.

[17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv: 1704.04861 (2017)*, Apr. 2017.

[18] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5mb model size," *arXiv preprint arXiv: 1611.10012 (2016)*, Nov. 2016.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv: 1512.03385v1 (2015)*, Dec. 2015.

[20] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv: 1409.1556v6 (2016)*, Apr. 2016.

[21] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, "mRNA: Enabling efficient mapping space exploration for a reconfigurable neural accelerator," *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 282–292, Apr. 2019.

[22] Y. N. Wu, J. S. Emer, and V. Sze, "Accelergy: An architecture-level energy estimation methodology for accelerator designs," *International Conference On Computer Aided Design (ICCAD)*, Nov. 2019.