STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators

Francisco Muñoz-Martínez
Universidad de Murcia
Murcia, Spain
francisco.munoz2@um.es

José L. Abellán
Universidad Católica de Murcia
Murcia, Spain
jlabellan@ucam.edu

Manuel E. Acacio
Universidad de Murcia
Murcia, Spain
meacacio@um.es

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

2021 IEEE International Symposium on Workload Characterization (IISWC) | 978-1-6654-4173-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/IISWC53511.2021.00028

***Abstract*—The design of specialized architectures for accelerating the inference procedure of Deep Neural Networks (DNNs) is a booming area of research nowadays. While first-generation rigid accelerator proposals used simple fixed dataflows tailored for dense DNNs, more recent architectures have argued for flexibility to efficiently support a wide variety of layer types, dimensions, and sparsity. As the complexity of these accelerators grows, the analytical models currently being used for design-space exploration are unable to capture execution-time subtleties, leading to inexact results in many cases as we demonstrate. This opens up a need for cycle-level simulation tools to allow for fast and accurate design-space exploration of DNN accelerators, 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-level microarchitectural simulation framework that can plug into any high-level DNN framework as an accelerator device and perform full-model evaluation (i.e. we are able to simulate real, complete, unmodified DNN models) of state-of-the-art rigid and flexible DNN accelerators, both with and without sparsity support. As a proof of concept, we use STONNE in three use cases: *i)* a direct comparison of three dominant inference accelerators using real DNN models; *ii)* back-end extensions and *iii)* front-end extensions of the simulator to showcase the capability of STONNE to rapidly and precisely evaluate data-dependent optimizations.**

I. INTRODUCTION

Deep Neural Networks (DNNs) constitute nowadays a promising breakthrough for a large number of artificial intelligence (AI) applications [1]. The fact that their inference phase must be primarily done *in-situ* has paved the way for the development of a plethora of accelerator architectures so as to maximize performance per watt while meeting latency and energy-efficiency demands ([2], [3], [4], [5], [6], [7], [8], [9] are a few examples). The key behind all of these recent architectures has been the capture of the different patterns of data reuse in what is known as a dataflow [10], [11] and the use of data-dependent optimizations to reduce computation and memory footprint [6].

First-generation *rigid* DNN inference accelerators ([2], [3], [12], [7]) focused their designs on fixed-size clusters of multipliers-and-accumulate units interconnected by means of a fixed tightly-integrated on-chip network fabric specifically tailored to efficiently support a particular dataflow. For example, the Google TPUv1 [2] is built by interconnecting 256×256 Multiply-Accumulate (MAC) units to a tightly-coupled 2D grid and supports a weight-stationary dataflow, while ShiDianNao [12] groups 8×8 MAC units supporting an output-stationary dataflow.

Unfortunately, as DNN models evolve at a rapid pace, these fixed designs fail to adapt well to the great diversity of layer types and dimensions in contemporary proposals. Table I shows the seven DNN models considered in this work. These models fall into three different application domains that mostly cover the diversity of machine learning models in the MLPerf benchmark suite [13], and represent different design tradeoffs for accuracy, memory requirements and

Domain	DNN Model	Sparsity	Dominant Layer Types
Image Classification	Mobilenets-V1 (M) [15]	75%	Factorized Convolution (FC)
			Linear (L)
	Squeezenet (S) [16]	70%	Squeeze Convolution (SC)
			Expand Convolution (EC)
	Alexnet (A) [17]	78%	Convolution (C)
			Linear (L)
	Resnets-50 (R) [18]	89%	Residual Function (RF)
			Convolution (C)
	VGG-16 (V) [19]	90%	Convolution (C)
			Linear (L)
Object Detection	SSD-Mobilenets (S-M) [20]	75%	Factorized Convolution (FC)
			Linear (L)
Language Processing	BERT (B) [21]	60%	Transformer (TR)
			Linear (L)

TABLE I: Contemporary DNN models explored in this work.[a]

[a]ImageNet [22], COCO [23] and squad-1.1 [24] dataset have been used to train the image classification, object detection and language processing models, respectively.

computational complexity. From the data in this table, we can highlight two main sources of inefficiency in terms of performance and energy that are inherent to these first-generation DNN accelerators: *i) Wide range of DNN types:* DNN models are continuously evolving featuring different sizes and types of layers (e.g., the fourth column in Table I), hence leading to varying computing demands; *ii) Sparsity:* Modern DNN workloads exhibit different degrees of weight and input sparsity due to both network pruning and the use of the non-linear activation functions such as ReLU, respectively. Table I shows the significant state-of-the-art average weight sparsity ratio (from 60% to 90%) after applying an unstructured weight pruning approach similar to that described by Zhu et al. [14].

Exploiting this large diversity in computing demands makes rigid DNN accelerators, which are based on fixed on-chip topologies, highly ineffective, leading to poor scalability, under-utilization of the computing resources, and low energy efficiency [8], [9].

To overcome these limitations, recent proposals such as FlexFlow [5], MAERI [8] and SIGMA [9] advocate using *flexible* DNN accelerator fabrics, which can be reconfigured to efficiently map different dataflows and dot product partitions through the creation of dynamic-size clusters (i.e., a set of multipliers computing the same output) in the same hardware substrate. Of course, this flexibility comes at the cost of increased architectural complexity that urges for a more exhaustive design-space exploration for fine tuning before building the particular ASIC-based or FPGA-based DNN accelerator.

Additionally, other works are exploring *data-dependent* optimizations in DNN accelerators that try to reduce computation and memory footprint by exploiting hardware optimizations based on the input data. For example, SNAPEA [6] implements a data-dependent optimization that leverages the fact that there are no negative values

978-1-6654-4173-5/21/$31.00 ©2021 IEEE
DOI 10.1109/IISWC53511.2021.00028

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

in the input values of a Convolutional Neural Network (CNN). This approach statically re-orders at compile time the weights according to their signs, and periodically performs in hardware a single-bit sign check on the partial sum during the execution. Once the partial sum drops below zero, the rest of the computations are cut off, since the output value will inevitably be zero after applying the typical ReLU activation function in CNNs. In these cases, it is crucial to get access to the precise data values that will be used during the inference procedure.

Microarchitectural simulators have been extensively used during the design process of CPU and GPU architectures ([25], [26], [27], [28], [29], [30] are just a few examples), albeit as we explain in Section II, most recent efforts have focused on using analytical models to describe an accelerator design by means of simple yet insightful formulas. However, as we also demonstrate in Section II, contemporary analytical models, while very useful for exploring Pareto-optimal accelerator parameters [11], [31] lag far behind in timing accuracy when modeling more complex flexible architectures, and when running non-trivial computation (e.g., sparse computation or DNN layers that do not map well onto the accelerator substrate and lead to compute under-utilization) or data-dependent optimizations. In these cases, analytical models are not able to capture performance bottlenecks or unexpected behaviors that may occur during a real DNN full-model execution.

To the best of our knowledge, there is still no detailed, cycle-level, open-source microarchitectural simulator for extensive and accurate design-space exploration of DNN inference accelerators (further details are given in Section II and are summarized in Table II). To bridge this gap, in this work we present STONNE (which stands for *Simulation TOol of Neural Network Engines*), the first attempt to derive a cycle-level, highly-modular and highly-extensible simulator for DNN inference accelerator microarchitectural exploration[1]. STONNE builds on the observation that most current DNN accelerator architectures can be logically organized as three configurable on-chip network fabrics (distribution network, multiplier network, and reduction network) and the corresponding memory controller and buffers, and provides an easily expandable and configurable set of microarchitecture modules (for buffers, on-chip data delivery and memory controllers) that, conveniently selected and combined, can faithfully simulate both rigid DNN accelerators (e.g., the Google TPU [2]) and flexible DNN accelerators (e.g., MAERI [8] or FlexFlow [5]), including those exploiting sparsity (e.g., SIGMA [9]). Additionally, and unlike prior tools, STONNE is directly integrated with the widely used PyTorch DL framework [32] as an accelerator device, which enables cycle-level simulation of complete DNN models and precise evaluation of data-dependent optimizations used in a plethora of DNN accelerators (e.g. SNAPEA [6]).

We see the following contributions in this work:

- We demonstrate the value of cycle-level simulation for accurate design-space exploration of DNN accelerators (Section II). Particularly, we show that a state-of-the-art analytical model can underestimate the number of clock cycles for certain DNN layers by more than 400%.
- We present (Sections III and IV) and validate (Section V) STONNE[2], the first simulator, to the best of our knowledge, that is connected as an accelerator device with a contemporary DL framework (PyTorch [32]), and enables cycle-level microarchitectural simulation of inference accelerators (with both dense and sparse computation support) running complete DNN models.
- We demonstrate the usefulness, versatility and capability of STONNE via three diverse use cases. In the first one, we perform a direct comparison between TPU, MAERI and SIGMA type inference architectures running the seven DNN models shown in Table I. In the other two use cases, we demonstrate the capacity of STONNE to be extended and to model data-dependent optimizations. Particularly, the second use case considers the modification of the back-end of the simulator by implementing SNAPEA [6], whereas the third use case analyzes the potential of static filter scheduling in DNN sparse accelerators, which entails modifications to STONNE's front-end.

[1]Support of training procedures in STONNE is part of our ongoing work.

[2]The STONNE Simulator can be found here: https://github.com/stonne-simulator/stonne.

	Cycle Level	Architecture Type	Sparsity Support	FullModel Eval	DataDep Opt
MAERI BSV	✓	Flexible	✗	✗	✗
SIGMA RTL	✓	Flexible	✓	✗	✗
SCALE-Sim	✗	Rigid	✗	✗	✗
MAESTRO TimeLoop	✗	Both	✗	✗	✗
DNNSim	✗	Rigid	✓	✗	✗
SMAUG	✓	Rigid	✗	✓	✗
STONNE	✓	Both	✓	✓	✓

TABLE II: State-of-the-art Simulators for DNN Accelerators.

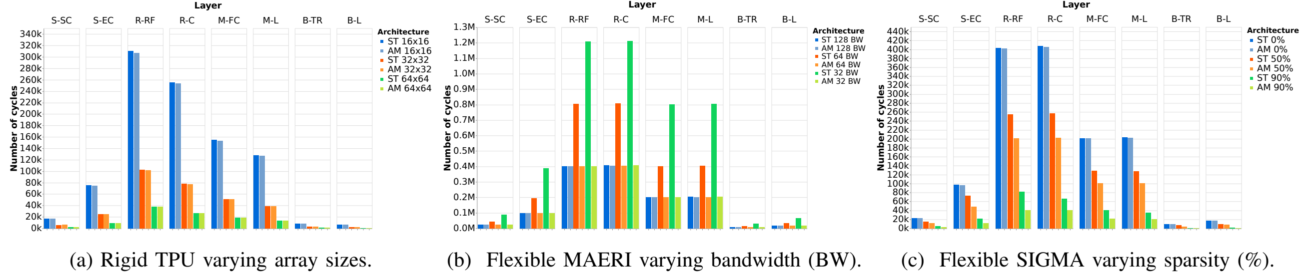
II. Motivation and Related Work

Table II shows a qualitative comparison of STONNE with respect to contemporary publicly available tools for design-space exploration of DNN inference accelerators. For the comparison, we consider five desirable features that a DNN inference simulator should meet: 1) *cycle-level simulation*; 2) *support for both rigid and flexible DNN accelerator architectures*; 3) *support for sparse executions*; 4) *ability to perform complete evaluations of deep learning models*; and 5) *ability to implement and evaluate data-dependent optimizations*.

Analytical Modeling. SCALE-Sim [33], MAESTRO [11], TimeLoop [31] and DNNSim [34] have recently been proposed as frameworks that enable the analysis of different dataflows in DNN architectures. These tools are very powerful for fast exploration of high-level architectural details, as they are based on analytical models that calculate the degree of data reuse and computations using simple equations. These types of simulators work accurately when it comes to rigid architectures as they are simple enough to be represented by a set of formulas. However, when the complexity of the accelerator grows and/or the computation does not follow regular patterns, these models fail to faithful capture the exact behavior of the architecture.

Figure 1 shows this fact quantitatively. First, SCALE-Sim only models simple rigid architectures (e.g., TPU-like systolic arrays) and do not have support to handle sparsity. Figure 1a shows the number of cycles obtained with this analytical model and with the cycle-level execution model implemented in STONNE after running eight different representative layers (Squeeze, Expand, Factorized and Regular Convolutions – *SC, EC, FC, C*; Linear – *L*; and Transformers – *TR*) extracted from Squeezenet (S), Resnets-50 (R), Mobilenets (M) and BERT (B). We have configured both models to simulate an Output-Stationary systolic array varying the size of the array of processing elements (PEs) from 16×16 to 64×64. As we can see, we obtain for the three configurations almost the same number of cycles for both alternatives, demonstrating that analytical models are valuable tools when it comes to rigid DNN accelerator architectures.

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.



(a) Rigid TPU varying array sizes. (b) Flexible MAERI varying bandwidth (BW). (c) Flexible SIGMA varying sparsity (%).

Fig. 1: Runtime for 8 DNN layers run on a simulated DNN inference accelerator using STONNE (ST) and an analytical model (AM). We use the following notation when plotting the results: `X-Y`, where `X` is the DNN model and `Y` is the layer type.

Contrarily, we have observed that analytical models fail to faithfully capture microarchitectural details of flexible DNN accelerator architectures, and therefore, are not appropriate to identify many of the bottlenecks or unexpected behaviors that may occur during a real DNN full-model execution. To demonstrate this claim, we perform a set of experiments for a 128-multiplier flexible dense accelerator simulating MAERI [8], using the detailed analytical model provided by the authors of the MAERI paper [8]. Figure 1b plots the number of cycles reported by both STONNE and the analytical model for different global buffer bandwidth (i.e., number of elements that the global buffer can deliver per cycle to the processing elements) configurations. In both cases we modify the parameter that controls the bandwidth to consider 128 (full bandwidth), 64 and 32 elements/cycle. We use the same layers as before. As we can see, the analytical model perfectly matches the performance obtained with STONNE when there is full bandwidth (average difference of 1.03%), as this ideal case can be easily represented by a set of mathematical formulas. However, as the bandwidth decreases, STONNE begins to report a much higher number of cycles. This is due to the ability of a cycle-level simulator like STONNE to faithfully capture the stalls produced in the architectural pipeline and that arise as a result of the increasing number of conflicts in the MAERI's distribution and reduction networks. The difference between the results reported by STONNE and the analytical model for 32 elements/cycle increases up to 400% (see M-FC in Figure 1b), alerting about the important limitations of the analytical models.

Furthermore, we also observe that an analytical model is not capable of accurately representing DNN *sparse* executions. Figure 1c shows the same executions as before, but this time we have configured STONNE to model a sparse accelerator like SIGMA [9]. Again, we compare the results against the analytical model provided by the authors of SIGMA [9]. This time, we assume full bandwidth and variable sparsity ratio of the matrices between 0% and 90%. In this case, we also observe a perfect match between both STONNE and the analytical model when the sparsity ratio is 0%, but this similarity begins to diverge as the sparsity ratio increases (diverging up to 92% for a sparsity ratio of 90%). The reason for this difference is that the actual distribution of zeros in the matrices, which affects the cluster sizes, and in the end, the performance obtained by the architecture, cannot be modeled analytically. Instead, cycle-level, full-model evaluations with real weight values are needed to capture it.

Cycle-level Simulation. Among all the alternatives, only the MAERI BSV [35], SIGMA RTL [36] implementations and SMAUG [37] claim to model flexible accelerator architectures with cycle-level precision. However, none of them really allows for efficient design-space exploration and rapid prototyping. MAERI BSV and SIGMA RTL are just two limited hardware implementations in Bluespec HDL and Verilog, respectively, written to demonstrate the effectiveness of these two flexible architectures. Hence, they are not prototypes adapted to be extended or to carry out the inference procedure of a complete DNN model or perform design-space exploration. Although SMAUG is aimed to efficiently support full-model simulation of flexible architectures, actually this flexibility only means that it is able to execute any layer with any tile configuration mapping. However, the architectures currently being supported are a systolic array and the NVIDIA Deep Learning Accelerator (NVDLA), which cannot be considered flexible accelerators. Besides, since SMAUG is a trace-based simulator, it is unable to run whole modern DNN models, and therefore, has to resort to a sampling approach, which impedes its use for real full-model evaluations or examine data-dependent architectural optimizations. STONNE addresses all the above shortcomings.

III. STONNE Framework

STONNE is a cycle-level microarchitectural simulator for DNN inference accelerators. STONNE is open-sourced under the terms of the MIT license. To allow for full-model evaluations, STONNE is connected with a Deep Learning (DL) framework (PyTorch [32] and Caffe [38] DL frameworks in the current version). Therefore, STONNE can fully execute any dense and sparse DNN model supported by the DL framework that uses as its front-end. The simulator has been written entirely in C++, following the well-known GRASP and SOLID programming principles of object-oriented design [39]. This has simplified its development and makes it easier the implementation of any kind of DNN inference accelerator microarchitecture, tile configuration mapping and dataflow.

Figure 2(a) shows a high-level view of STONNE with its three major modules for full-model simulation flows.

Simulation platform: This constitutes the principal block (see the central block in Figure 2(a)), since it includes the implementation of the simulated DNN accelerators (i.e., `Simulation Engine`) whose different internal microarchitecture modules allow to compose and cycle-by-cycle simulate both rigid and flexible DNN accelerators. These modules are further described in Section IV. The composition of each accelerator (i.e., the selection of the microarchitecture modules) is defined by the user through a hardware configuration file given by the input module. These modules are configured through the `Configuration Unit` at runtime according to a set of signals generated by the `Mapper` based on the configured microarchitectural modules and the DNN layer type and shape to be executed.

The simulation platform is interfaced by means of a set of coarse-grained instructions called the `STONNE API` (Table III). This API

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

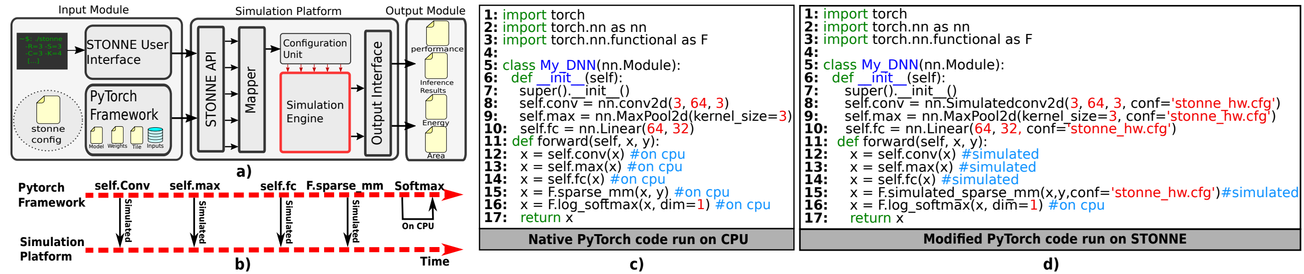


Fig. 2: a) STONNE framework. b) DNN simulation example. c) Native PyTorch code for CPU. d) Modified PyTorch code for STONNE.

Instruction	Description
CreateInstance	Creates an instance of STONNE.
ConfigureCONV	Configures the accelerator to run a convolution operation.
ConfigureLinear	Configures the accelerator to run a fully-connected layer.
ConfigureDMM	Configures the accelerator to run a matrix multiplication.
ConfigureSpMM	Configures the accelerator to run a sparse matrix multiplication.
ConfigureMaxPool	Configures the accelerator to run a max pooling layer.
ConfigureData	Configure weights, inputs and outputs addresses from the CPU to the accelerator memory.
RunOperation	Launches the simulation according to the current configuration of the architecture.

TABLE III: STONNE API Instruction Set.

is the manner in which the input module (i.e., the DL framework) can interact with the simulated accelerator, configuring its simulation engine according to the user configuration file, loading layer and tile parameters, and configuring the weights and the inputs addresses in the main memory. The STONNE API can be easily extended to support new instructions.

Input Module: Due to the flexibility that the `STONNE API` provides, the simulator can be fed easily using a standard DL framework. To this end, we have modified the PyTorch DL framework[3] (see the left block in Figure 2(a)) to connect it to the simulator and to make it able to run an instance of the `Simulation Engine` transparently to the user. This way, a PyTorch user just needs to select the typical *.pb* file with the weights, choose the inputs (e.g., a set of images or sentences) and briefly modify each DNN model to include the path of the hardware configuration file with the parameters of the accelerator to simulate, and the tile configuration for every layer. Furthermore, since PyTorch requires a more complicated installation and use, apart from this mode of execution, we have also enabled the `STONNE User Interface`. This is basically a tool inside STONNE in which the user is presented with a prompt and a set of well-defined commands to load any layer and tile parameters onto a selected instance of the simulator, and run it with random weights and input values. This allows for faster executions, facilitating rapid prototyping and debugging.

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, and activity counts of different components such as wires, FIFOs or SRAM usage (i.e., number of accesses). In particular, STONNE reports two different output files: First, a general file in *json* format that includes a summary of the statistics and facilitates their processing through user-created scripts; Second, a *counter file* written in a customized format which contains the activity counts for each component of the architecture (e.g., multiplier, wire, adder, etc). From these activity counts, the output module is able to report the amount of energy consumed by the simulated architecture. To do so, STONNE includes a script that given the counter file and a table-based energy model similar to Accelergy [40], computes the total consumed energy taking into account the cycle-level activity stats for each module and the corresponding energy costs. Similarly, the area numbers are obtained by employing a table-based model, calculating the final area based on the architectural parameters and the area cost of each one. Obviously, these statistics depend, for example, on the particular data format (e.g., FP16 or INT8) utilized to represent the parameters of the DNN model. So, STONNE includes different energy and area tables that can be used. To derive these tables, we ran synthesis using Synopsys Design-Compiler, and place-and-route using Cadence Innovus on each module of the simulated accelerator (further details in Section V).

Walk-through example: Figures 2(b-d) clarify the interaction between the Input Module (i.e., PyTorch) and the Simulation Platform with a walk-through example illustrating the execution of a simple DNN model composed of 5 typical DNN operations: *Conv2d*, *MaxPool*, *Linear*, *sparse_mm* and *log_softmax*. First, Figure 2b graphically shows this interaction over time (x-axis) when running these operations. As we can see, the execution is driven layer-by-layer by the DL framework (PyTorch in this case) that offloads compute-intensive layers (e.g., a convolution layer) to the simulated accelerator, and runs natively in the real CPU those layers that are not compute-intensive enough for acceleration (e.g., a SoftMax layer).

More specifically, for each intensive computational operation such as a convolution (*nn.Conv2d)*, the DL framework configures the corresponding memory addresses onto the simulator (using the *ConfigureData* instruction) and configures the layer to be run (*ConfigureCONV* instruction). Then, the simulator takes control and runs the operation on a cycle basis on the simulated accelerator. Once the simulator finishes, it reports the statistics through the Output Module, notifies back the DL framework and returns it control to continue with the next operations (*nn.MaxPool*, *nn.Linear* and *F.sparse_mm*). As shown, those operations that are not worth for acceleration (e.g., softmax operation) are executed directly by the DL framework (as it would be done in a real scenario), so correctness of the entire execution is ensured.

Figure 2c shows the native PyTorch code that would be used to run these operations on a CPU (or GPU), while Figure 2d shows the required modifications to off-load the aforementioned intensive computational operations onto STONNE (lines 8, 9, 10 and 15). As we can see, each operation instance to be off-loaded is replaced by a similar operation that adds the prefix *Simulated* to its name. This

[3] Other DL frameworks, such as Tensorflow, can be easily integrated with STONNE using the same STONNE API philosophy.

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

allows PyTorch to distinguish when the operation has to be run on STONNE rather than natively on the CPU (or GPU). Furthermore, the arguments of the operations have to be extended to include the hardware configuration file (i.e., *stonne_hw.cfg*) that will be used by the simulation platform to create the instance of the simulated accelerator. As it may be appreciated, those lines that do not change will run normally on PyTorch, maintaining the correctness of the execution.

Note that mapping of non matrix multiplication layers on STONNE, such as Pooling (e.g., *nn.MaxPool*) and batch normalization, is not a problem, as they can be easily supported in flexible accelerator architectures without additional specific SIMD modules (as required in some other architectures) [8]. Even crossing layers (i.e., kernel fusion) operations could be mapped onto the processing units of a flexible architecture, although this latter feature has not yet been incorporated into STONNE. As we illustrate with use cases 2 and 3 in Section VI, STONNE can be easily extended to incorporate other hardware and software optimizations.

IV. STONNE Simulation Engine

STONNE builds on the observation that most current DNN accelerator architectures can be logically organized as three configurable network fabrics (distribution network, multiplier network, and reduction network) and the corresponding memory controller and buffers [41], and provides an easily expandable and configurable set of microarchitecture modules (for buffers, on-chip data delivery and memory controllers) that, conveniently selected and combined, can model both rigid and flexible DNN accelerators (see Figure 3a).

A. On-Chip Networks

All the on-chip components are interconnected by using a general three-tier network composed of a Distribution Network (DN), a Multiplier Network (MN), and a Reduce Network (RN), inspired by the taxonomy of on-chip communication flows within DNN accelerators [8]. These networks can be configured to support any topology to model state-of-the-art accelerators such as the TPU, Eyeriss-v2, ShDianNao, SCNN, MAERI and SIGMA, among others.

*1) **Distribution Networks (DNs)**:* In order to deliver the data from the Global Buffer (GB) to the MN, we implement the next DNs:

- **Tree Network (TN)**: A TN (illustrated in Figure 3b(e)) is a binary-tree-based network topology inspired by the MAERI distribution network that is replicated as many times as the number of read ports available in the GB, and that provides single-cycle unicast, multicast and broadcast data delivery from the GB to the multipliers [8].
- **Benes Network (BN)**: A BN (illustrated in Figure 3b(f)) is an N-input N-output non-blocking topology with $2 \times log(N) + 1$ levels, each with N tiny 2×2 switches. This DN is implemented in SIGMA [9] and ensures efficient single-cycle unicast, multicast and broadcast data delivery from the GB to the MN.
- **Point to Point Network (PoPN)**: Unlike the two DNs described above, the PoPN (illustrated in Figure 3b(g)) provides only unicast data delivery from one source point (typically the GB) to a destination (typically a multiplier). This is the basic component to build an interconnect for a systolic array such as the TPU.

*2) **Multiplier Networks (MNs)**:* These networks are made up of 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 GB to the RN so that folding[4] can be supported, whereas the multiplier configuration mode is utilized to compute a multiplication between a weight and an input value. Currently, we support two MN topologies:

- **Linear Multiplier Network (LMN)** (Figure 3b(h)): This RN is capable of leveraging the spatio-temporal data reuse (e.g., when processing the sliding window operation of convolution DNN layers) by using forwarding links between each pair of multipliers. This reduces the bandwidth pressure on the memory and on the DN by reusing data across different multipliers. The LMN is utilized in several DNN accelerators (such as MAERI and TPU).
- **Disabled Multiplier Network (DMN)** (Figure 3b(i)): Removes completely the forwarding links between the multipliers and is aimed at performing basic GEMMs. This MN is presented in DNN accelerators such as SIGMA [9] and SpARCH [42] whose basic primitive is the GEMM operation, and therefore, the sliding window operation has no longer effect.

*3) **Reduction Networks (RNs)**:* These networks are composed of adders whose purpose is to accumulate the different clusters of partial sums that are generated by the MN. Currently, we support the following RNs:

- **Reduction Tree (RT)** and **Augmented Reduction Tree (ART+DIST)** (Figure 3b(a)): An ART integrates a tree-based topology built upon a reduction tree but augmented with one 3:1 adder unit per node 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. This augmented tree enables flexible support of multiple and non-blocking virtual reduction trees over a single physical tree hardware substrate [8].
- **ART + Accumulation Buffer (ART+ACC)** (Figure 3b(b)): This RN is similar to ART, but allocates a set of accumulators at the output of the reduction network, allowing partial sums from consecutive iterations to be temporarily accumulated in the accumulators, and enabling them to be pipelined.
- **Forwarding Augmented Network (FAN)** (Figure 3b(c)): As it is demonstrated in SIGMA [9], the ART topology is inefficient in terms of area and power due to the 3:1 adders. SIGMA proposed a more sophisticated RN called FAN, which equivalently to ART, allows to create any arbitrary number of dynamic-size clusters, but replaces the inefficient 3:1 adder switches by simpler 2:1 adders.
- **Linear Reduction Network (LRN)** (Figure 3b(d)): In order to support all types of accelerators, in STONNE we also implement a linear reduction network which is typically used in rigid accelerators such as the TPU [2], Eyeriss [4], Eyeriss-v2 [43] or ShDianNao [12], to perform the cluster reductions.

B. Memory Hierarchy and Memory Controllers

STONNE implements the typical configurable memory hierarchy found in most DNN accelerators composed of local storage, some on-chip global storage (i.e., the Global Buffer, GB), and the off-chip DRAM memory. These three levels of the hierarchy are configurable by the user through the STONNE configuration file, which defines parameters such as bandwidth, different FIFO sizes, GB size or DRAM size and technology (e.g., HBM). Data orchestration between

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

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

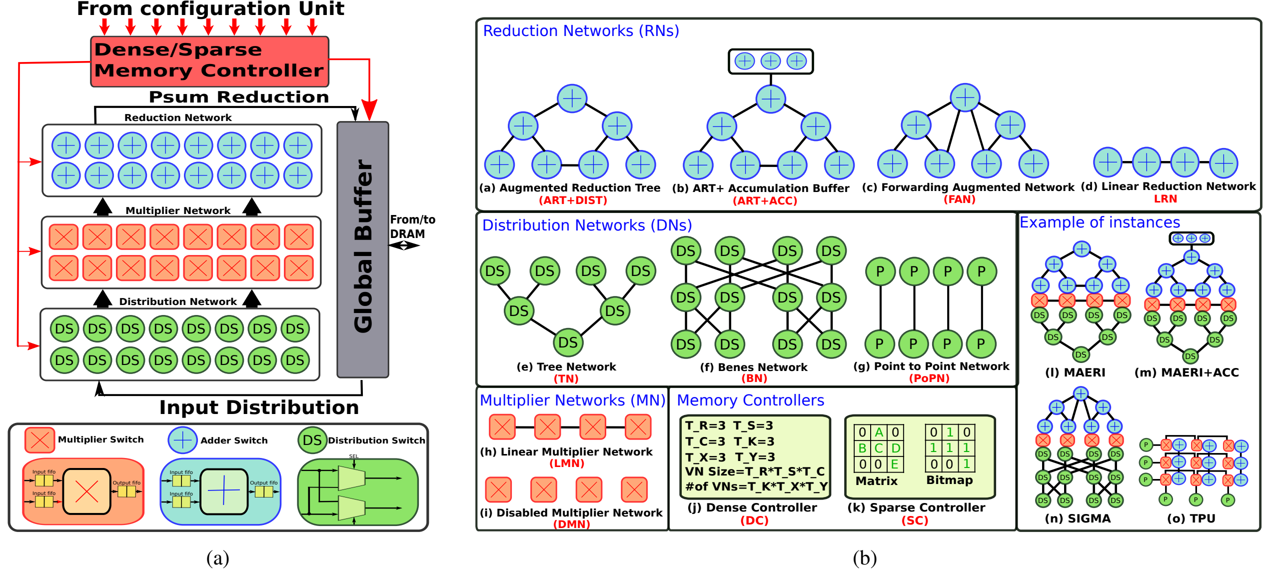


Fig. 3: (a) Overview of the general flexible DNN inference accelerator considered in STONNE. (b) Basic building blocks.

the GB and the distribution and reduction networks is performed by a memory controller (i.e., control unit) which is also selected by the user based on their preferences. As data movement differs depending on both the dataflow and whether the execution is dense or sparse, STONNE implements different types of memory controllers which are configurable and interact with DRAM memory assuming double-buffering prefetching at the Global Buffer. These memory controllers use internal counters to calculate the next addresses that the accelerator will read or write and their implementation is inspired by Buffets [44]. We described these memory controllers as follows:

- **Dense controller (DC)**: it takes inspiration from mRNA [45] and hence, orchestrates the data based on a fixed tile partition that cannot change during the execution of the layer (see Figure 3b(j)). First, a DNN 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, G is the number of groups (i.e., to give support for factorized convolutions), N is the batch size, and X' and Y' are the number of rows and columns in the output respectively. 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 dot product. 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 clusters that are mapped onto the architecture. Note that, if the size of the cluster is smaller than the filter size (i.e., $(T_R/R \times T_S/S \times T_C/C) > 1$), then the architecture will have to enable folding as it will be necessary to iterate over the same cluster to process the entire filter.
- **Sparse Controller (SC)**: The use of the sparse controller (Figure 3b(k)) changes drastically the way in which the data flows throughout the elements of the architecture as when sparsity is enabled, the size of the dot products varies according to the sparsity of the data. The sparse controller implemented in STONNE runs GEMM operations (any CONV operation can be mapped to GEMM using the *img2col* function) and supports both bitmap and CSR formats to represent the sparsity of the MK and KN matrices.

	TPU-like	MAERI-like	SIGMA-like
Memory Controller	Dense	Dense	Sparse
Distribution Network	PoPN	TN	BN
Multiplier Network	LMN	LMN	DMN
Reduce Network	LRN	ART	FAN

TABLE IV: Modeling DNN Accelerators in STONNE.

Obviously, the configured memory controller must always be compatible with the hardware substrate selected to be modelled. In terms of dataflows, STONNE implements the weight-stationary, output-stationary and input-stationary dataflows. Other alternatives could also be easily implemented from the existing memory controllers. DRAM is modeled using DRAMsimv3 [46].

C. Modeling DNN Accelerators in STONNE

Cycle-level simulation: Figure 4 shows the class diagram used in the STONNE Simulation Engine to model each component. As can be observed, all the components contain a `cycle()` method which implements their behaviour during a clock cycle. To enable the abstraction and allow the user to configure its own accelerator, we employ a hierarchical abstract class implementation whose specific instances are selected at runtime by the main `Accelerator` class. This top class iterates over every configured component in the accelerator and runs its `cycle()` method, emulating a cycle-by-cycle microarchitectural behaviour.

Variability: Using the building blocks shown in Figure 3b(a-k), STONNE is able to model a variety of DNN accelerator architectures. Examples of particular architectures directly supported in STONNE and the basic building blocks used in each case are given in Table IV (also drawn in Figure 3b(l-o)). Moreover, STONNE can be easily extended with additional models of DNs, MNs, RNs and memory controllers, giving rise to new accelerator architectures.

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

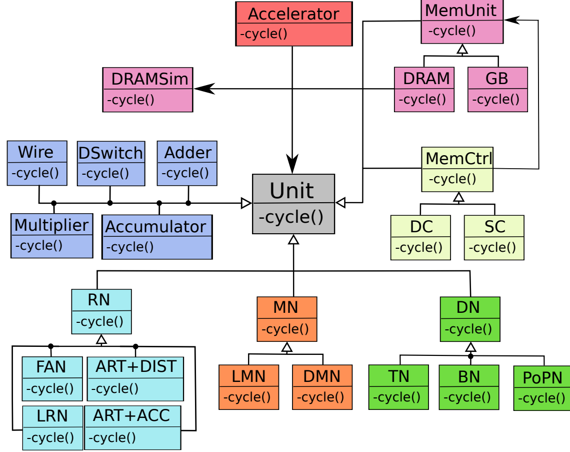


Fig. 4: Simulation Engine class diagram. Acronyms from Figure 3b.

Data-dependent optimizations: Since STONNE connects with DL frameworks, the aforementioned building blocks can be extended to precisely evaluate data-dependent architectural optimizations. The last two use cases presented in Section VI showcase this.

Limitations of STONNE: In essence, STONNE is aimed to model MAC-based accelerators. Modelling other types of accelerators (e.g., bit-wise or analog ones) could require major changes to the Simulation Platform component of STONNE.

V. Validation and Results

Timing validation: To validate the timing accuracy of STONNE against real hardware, we focus on three open-source implementations of DNN accelerators: the MAERI BSV code, the SIGMA Verilog code, and the TPU RTL implementation used to validate SCALE-Sim [33] implemented in Verilog. This helps us also validate the experimental results performed in our three use cases in Section VI. For the validation process, we configure three instances of a MAERI-like, a SIGMA-like and an output stationary TPU-like architecture using their corresponding building blocks (see Figures 3b(l), 3b(n) and 3b(o)).

Since these RTL versions do not provide the large flexibility of our cycle-level architectural simulator–which can model any combination of the parameters of the accelerator (e.g., number of MSs, number of trees in the DN, number of input/output ports in the Global Buffer)– we are heavily constrained in the number of validation experiments that we can carry out. This way, for the MAERI-like architecture, we have configured both STONNE and BSV versions with 32 MSs and 4 DN/RN elements/cycle bandwidth parameters. In addition, the MAERI BSV version can only execute the three different types of layers listed in Table V, 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 the SIGMA-like version, we have configured both the RTL and STONNE versions with 128 MSs and 128 DN/RN elements/cycle bandwidth parameters running 4 layers. For the TPU-like, we have configured both the RTL model and STONNE using a 16×16 PE-array and full bandwidth. Given this set of microbenchmarks targeting specific layer types, we run STONNE using its direct user interface (the `STONNE User Interface` in Figure 2).

To evaluate the accuracy of timing simulation, Table V shows a comparison of the total number of executed cycles reported by the RTL versions and STONNE after running the eleven layers

Design	Layer	M	N	K	RTL # cycles	STONNE # cycles	Error %
MAERI	MAERI-1	6	25	54	1338	1381	3.10%
	MAERI-2	20	25	180	16120	16081	0.24%
	MAERI-3	6	400	54	26178	26581	1.51%
SIGMA	SIGMA-1	64	128	32	2321	2304	0.73%
	SIGMA-2	256	64	64	8594	8448	1.72%
	SIGMA-3	256	128	64	17192	16896	1.75%
	SIGMA-4	128	1	64	139	138	0.72%
TPU	TPU-1	16	16	32	66	67	1.50%
	TPU-2	16	16	16	50	51	2.00%
	TPU-3	32	32	16	200	204	2.00%
	TPU-4	64	64	32	1056	1072	1.50%

TABLE V: Timing accuracy of STONNE using RTL versions of MAERI, SIGMA and an OS-dataflow TPU.

supported by the RTL versions. As we can see, the differences in the total number of executed cycles obtained with STONNE and the RTL versions range from 0.14% to 3.10% (1.53% on average), demonstrating that STONNE closely mimics the characteristics of the hardware versions.

Functional validation: Since STONNE simulator is a back-end compute platform of PyTorch, it also outputs the result of the inference (the prediction) when running a particular DNN model for certain input data. To validate the functionality of STONNE, we have configured and run the three DNN accelerators presented in Table IV with 256 processing elements and full bandwidth (i.e., 256 elements/cycle). We have executed the seven DNN models listed in Table I with a test set of 50 samples (e.g., an image or a sentence) from their respective datasets, and for every sample, we have compared the output of the last DNN layer (e.g., the score digits of a fully-connected layer) reported by PyTorch when running natively on the CPU, with the obtained for the executions with STONNE. They perfectly match for all cases.

Accuracy of energy and area estimates: We ran synthesis using Synopsys Design-Compiler and place-and-route using Cadence Innovus on each module inside the TPU, MAERI and SIGMA RTL to obtain the real energy and area numbers. We used those numbers to derive the energy and area models implemented in STONNE.

VI. Examples of Use Cases of STONNE

Through three use cases, we demonstrate how STONNE can be used to conduct comprehensive evaluations of several DNN accelerator architectures running complete DNN models. For the three use cases we assume the next system parameters: 28-nm technology node, 1 GHz clock, FP8 datatype, 108-KB Global Buffer (GB) size and two 256 GB/s 512-MB HBM2 DRAM modules. Other relevant parameters that are specific to each use case are given below.

A. Evaluation of DNN inference in TPU, MAERI and SIGMA

The aim of the first use case is to directly compare three different accelerator architectures (namely, TPU, MAERI and SIGMA) considering their achievable performance, energy consumption and required area. All the simulations were performed considering the complete inference processing of the 7 DNN models presented in Table I.

Methodology and Configuration parameters. We assume the next system parameters for the three architectures: For both MAERI and SIGMA, we assume 256 multipliers and adders, and 128 elements/cycle GB read/write bandwidth. For the TPU, we have configured 256 processing elements and full bandwidth (as this architecture requires). Note that, configuring these three architectures in STONNE does not require any modifications in the simulation

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

framework as STONNE directly supports the required hardware modules and dataflows for all of them (see Table IV).

Results. Figure 5a shows the number of cycles obtained for the three simulated architectures. We observe that a MAERI-like architecture reaches average performance improvement of 20% over the TPU-like architecture for the execution of the seven DNN models, with a maximum of 231% for Mobilenets and a minimum of 9% for Resnets-50. Besides, we found that a SIGMA-like architecture is 91% faster on average than a MAERI-like one thanks to the sparsity support.

Figure 5b shows a breakdown of the total amount of energy consumed (μJ) in each case, distinguishing the main architectural components: Global Buffer (GB), Multiplier Network (MN), Distribution Network (DN) and Reduction Network (RN). As we can appreciate, the energy consumption is dominated by the RN as it reaches 84%, 58% and 43% of the total energy on average across the DNN models for the TPU-like, MAERI-like and SIGMA-like architectures, respectively. In general, STONNE finds that the SIGMA-like architecture is 70% and 54% more energy efficient than the MAERI-like and TPU-like architectures, respectively. This is due to the capacity of SIGMA to exploit sparsity, which reduces the number of operations by 77%, thus bringing significant dynamic energy savings. Finally, in terms of area, the SIGMA-like architecture is 13% more efficient than the MAERI-like one, while the TPU-like architecture is 17% and 6% more efficient with respect to the MAERI-like and SIGMA-like architectures, respectively. As we observe, the differences in area are not as noticeable as those in the energy and runtime metrics. This is due to the area required in the three cases is mainly dominated by the SRAM structure of the GB, which is the same for the three architectures, and that represents 70%, 77% and 82% of the total area of the MAERI-like, SIGMA-like and TPU-like architectures, respectively.

These results are consistent with the trends pointed in prior works [8], [9] and validate that flexible architectures can adapt much better to the current diversity of DNN layers.

B. Back-End Extension for Data-Dependent HW Optimizations

This second use case proves how the back-end of STONNE can be easily extended to model other accelerators. In particular, we will use STONNE to model the data-dependent accelerator SNAPEA [6]. This architecture that aims to optimized CNN processing, exploits a property in which all the activation values in the convolution operations are either zero or positive. Any negative value calculated during the convolution is directly converted into zero by the subsequent ReLU operation. This means that the weights can be statically reordered based on their signs so that the architecture can perform at runtime a single-bit sign check on the partial sum. Once the partial sum drops to zero, the rest of computations and memory accesses can be avoided, since the output value will unfailingly be zero.

Implementation. The changes that have been introduced in STONNE to model this architecture mainly affect its back-end, and are as follows:

1) Inclusion of a prior-simulation function in the input module (see Figure 2a) to reorder the weights as explained in SNAPEA [6] and that creates a table of indexes to locate the inputs. This table is passed to the memory controller (i.e., control unit) which will use it to match every sorted weight with its activation.
2) A new memory controller (i.e., Control Unit) in the Simulation Engine (see Figure 2a) that utilizes this table of indexes to correctly deliver the weights and inputs to the multipliers. This unit is just an extension of the previous dense memory controller already provided in STONNE and explained in Section IV-B.
3) We use the current linear multiplier network (see MNs Section IV-A) configured to use the output-stationary dataflow.
4) We have extended the accumulation logic in the processing units to detect when the results are negative. As soon as this event is triggered, the data is sent out to the Global Buffer, cutting out the computation earlier, and thus, saving energy and time. We implement the *exact mode* explained in SNAPEA.
5) To estimate energy consumption, we have included in the Output Module a new table with the energy model of SNAPEA based on the published energy numbers provided in the SNAPEA paper.

Methodology and Configuration parameters. Similar to the SNAPEA work [6], for this use case we model 64 multipliers and adders, and 64 elements/cycle GB read/write bandwidth. We have configured two different versions of our SNAPEA implementation: the Baseline, which models the SNAPEA architecture but that excludes the negative detection logic, and therefore, runs the entire execution; and the full SNAPEA architecture (we call it SNAPEA-like) which adds this logic, cutting out the computation earlier whenever possible. We have configured and run these two versions of the accelerator with the aforementioned parameters and we have executed the four purely CNN models of those listed in Table I (i.e., Alexnet, Squeezenet, VGG16, and Resnets-50) with a set of 20 input images extracted from ILSVRC-2012 validation dataset. For each input image, we have compared the output of the last DNN layer (e.g., the score digits of a fully-connected layer) reported by PyTorch when running only on a CPU, with the one obtained for the executions with STONNE simulating the two SNAPEA implementations to corroborate that they perfectly match.

Results. Figure 6a plots the speedups achieved by the SNAPEA-like architecture against the baseline for the considered four CNN models. STONNE shows that SNAPEA can bring average speedups of 35%, closely approaching the 30% originally reported in [6]. On the other hand, Figure 6b shows the energy consumed when running the benchmarks on the two architectures. The results are normalized to the baseline. Similarly, these numbers demonstrate that the speedups mentioned previously translate into significant energy savings (21% on average). These results can be explained by observing Figures 6c and 6d, which show the number of operations and memory accesses performed during the execution of the CNN models on both the SNAPEA-like architecture and the baseline. In particular, we can observe that on average the technique exploited by SNAPEA is able to reduce the number of computations and memory accesses by 30% and 16%, respectively, being Squeezenet (S) the CNN model with the highest reductions (30% in operations and 22% in memory accesses) which correlates with the highest improvements in energy consumption. As it can be appreciated, these results closely follow the trend reported in [6], confirming that SNAPEA is a promising optimization to be applied to CNN accelerators. Obviously, we found timing and energy differences between the original paper and the results obtained with STONNE. These differences mainly stem from slight variances in the methodologies used in each case: older CNN models are used in [6], potential differences in the weights of the CNN models are possible (the SNAPEA paper does not specify how the weights of the CNN models have been obtained), and presumably, different images are used as inputs in both cases. More importantly, however, is that we see the same order of magnitude in the gains that SNAPEA is able to achieve, demonstrating STONNE's

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

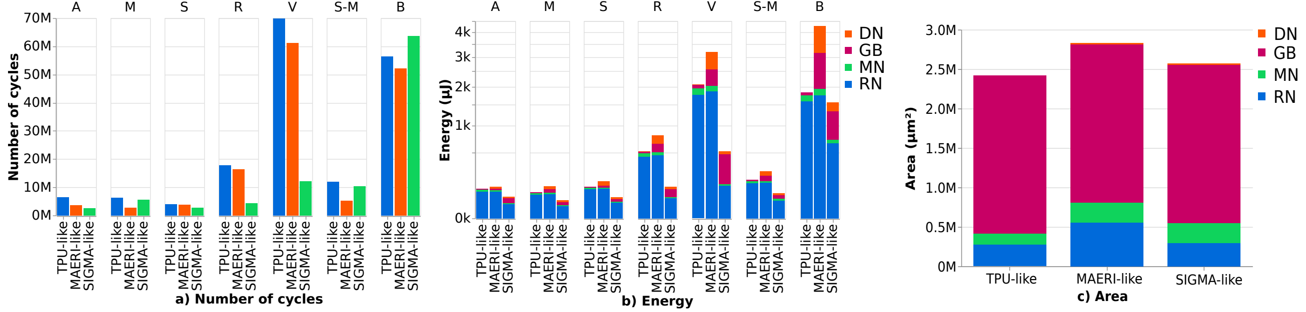


Fig. 5: Number of cycles (a) and energy consumption (we use an *sqrt* log y-axis) in μj (b) reported by STONNE after running the inference procedure of the DNN models listed in Table I on MAERI, SIGMA and TPU. Area estimations in μm^2 (c) for MAERI, SIGMA and TPU.

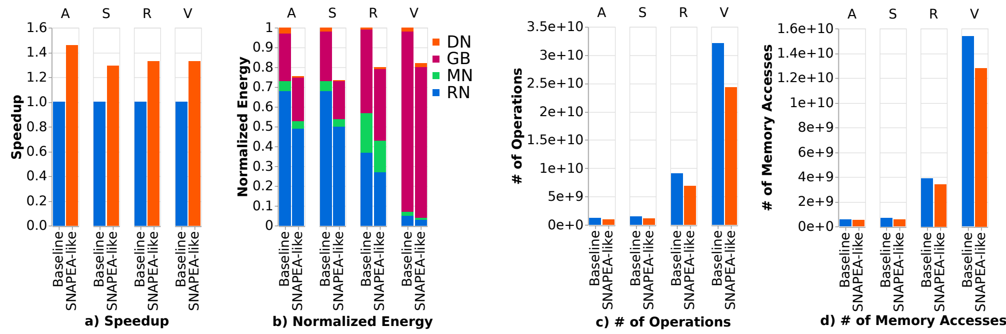


Fig. 6: Speedups (a) and normalized energy (b) of SNAPEA against the baseline after running four CNN models. Number of computed operations (c) and performed memory accesses (d) during the execution of the four CNN models for SNAPEA and the baseline.

ability to quickly and faithfully quantify the benefits of applying data-dependent optimizations to DNN accelerators.

C. Front-End Extension for Filter Scheduling in Sparse Accelerators

Through the third use case, we demonstrate that precise, full-model evaluation is required to expose the particular values used during inference. This is needed for some optimization techniques such as filter scheduling in flexible sparse DNN accelerators that we present here. The modifications now focus on the front-end of the simulator.

Motivation and idea: When we consider the large amount of sparsity in the filters of contemporary trained DNN models (from 60% to 90%, as shown in Table I), the amount of computation involving a certain filter can be largely reduced by only mapping the non-zero weights onto the accelerator's processing elements. In addition to this well-known optimization, in this use case, we demonstrate for the first time that *the way in which the filters of a sparse DNN model are scheduled onto a DNN inference accelerator might have significant impact on performance*. A prior work [47] examined this idea, but with a focus on GPUs. Here, we prove that this reordering has also significant impact on DNN accelerators. To do so, we use STONNE to simulate a flexible and sparse 256-MS SIGMA-like architecture [9].

First, to analyze scheduling opportunities of variable size filters onto the SIGMA's MN fabric, we pay attention to the diversity of filter sizes for the seven DNN models under study when sparsity is exploited. Particularly, Figure 7a shows the average number of entire filters that could be mapped simultaneously onto a 256-MS flexible architecture for every single DNN layer depending on each DNN model. As can observed, between 4 and 8 filters can be entirely mapped simultaneously in most cases. The only exceptions are Alexnet and BERT, that features larger filter dimensions by design (e.g., up to 4.3× larger filters compared to Mobilenets-V1, which comes next in terms of filter size). Moreover, we have further looked into the size of the filters required to compute each of the DNN layers, finding out huge variability between them. As an illustrative example, Figure 7b shows the size (*y-axis*) for every mapped filter onto the 256-MS architecture for the first layer of each DNN model[5] (*x-axis*).

Next, we show that the large filter size variability found in contemporary DNN models can be exploited to optimize the DNN inference procedure. In particular, we observe that the specific order in which the filters are scheduled to be mapped onto the DNN accelerator can impact the overall compute utilization, and in consequence, overall performance. Figure 8 illustrates this optimization opportunity for an example 8-MS SIGMA-like architecture. At the top of the figure, we consider an example layer composed of a 1×5 vector of inputs and four 1×5 sparse filters (F_0 and F_2 have an effective size of 4, while it is 2 in the case of F_1 and F_3) utilized to compute four dot products (producing the outputs from O_0 to O_3). We show how changing the order of the filters (which could be done either statically by the compiler or dynamically by the accelerator's memory controller) yields a variable number of cycles to complete the computation of the four dot products in the example.

Figure 8a shows that computing this layer completely ignoring

[5]The maximum mapping size is 256 because of the 256-MS SIGMA architecture.

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

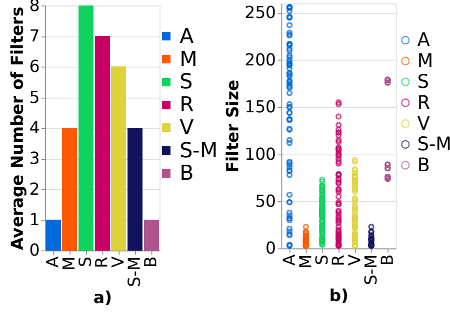


Fig. 7: (a) Average number of entire filters that can be mapped simultaneously in a 256-MS flexible sparse architecture. (b) Filter sizes for the first layer of the DNN models.

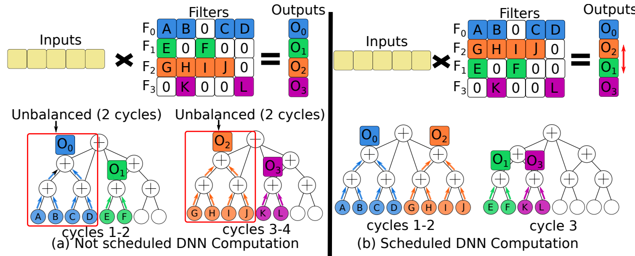


Fig. 8: An example filter scheduling heuristic to optimize four dot products in a SIGMA-like accelerator architecture.

the opportunities of scheduling the filters can lead to an unbalanced scenario, requiring a total of 4 cycles. On the other hand, Figure 8b illustrates how faster layer processing can be achieved if the computation of the four filters is scheduled differently. For instance, we can consider a simple scheduling heuristic to achieve perfect load-balancing in this example for computing the dot products. In particular, we can rearrange the filters to be mapped at every iteration depending on filter size following a Largest Filter First policy as follows (more details next). In the first iteration, F_0 and F_2 with 4 non-zero values can be mapped together, thus taking 2 cycles to compute their two outputs. Then, in the next iteration, the 2-size filters (F_1 and F_3) can also be mapped together, but in this case the computation of their associated outputs would be completed in just 1 cycle. Therefore, after applying this simple reordering of the filters, we can balance the computation of the outputs and make better use of the accelerator resources, which in the end results in fewer clock cycles required (25% less in this simple example).

This observation opens up a new avenue to optimize the inference procedure of sparse DNN models in flexible sparse DNN accelerators. In particular, the exploration of novel scheduling strategies for the sparse filters to better balance the mapping of the dot products onto the DNN accelerator, so that compute unit utilization is maximized and processing time reduced. We focus on exploring static scheduling heuristics, where the sparse memory controller issues the filters for execution in the same order that is determined by a certain static scheduling strategy directly applied layer by layer to each of the DNN models. To guarantee the correctness of the executions, a final reordering step is carried out after the last fully-connected layer of each DNN model.

Implementation. Implementing new scheduling approaches in STONNE just requires modifications in the front-end (i.e., Input Module) of the simulator. To do so, we have incorporated a prior-simulation function that reorders the filters based on its size and on the scheduling technique.

Methodology and Configuration parameters. We model and simulate a single Flexible Dot Product Engine (Flex-DPE) in a SIGMA-like flexible architecture that supports sparse and irregular matrix formats based on weights and/or activation sparsity. The on-chip network choices are shown in Table IV. We have run the seven DNN models whose sparsity levels are shown in Table I. We model these system parameters: 256 multipliers and adders and 128 elements/cycle Global Buffer (GB) read/write bandwidth.

In this work, we consider a simple static heuristic: *Largest Filter First (LFF)*. In LFF, the filters are reordered so that the sparse controller always selects the largest available filter (i.e., of those not yet used for computation) that can be mapped onto the Multiplier Network (256 MSs in this case). To cover the rest of the available MSs, the scheduler selects as many available filters as possible in descending size order. For the sake of completeness, we also present the results obtained for a *Random (RDM)* ordering.

Results. We compare the results for LFF and RDM against those obtained when running the sparse filters in their natural order (i.e., without performing any kind of reordering). We call this approach the *No Scheduling (NS)* ordering. As observed in Figure 9a, using the random scheduling strategy does not yield any performance improvement as the MS utilization does not increase at all. Alternatively, LFF is capable of both balancing the processing of the clusters during most of the execution and selecting a smaller filter when another one does not fit. This leads to increased MS utilization (2.5% on average), which translates into performance advantages ranging between 11% for the most sensitive DNN models (Squeezenet, VGG-16, Resnets-50 and ssd-Mobilenets) and 1% in models such as BERT, whose large filter sizes and low sparsity ratio (60%) often prevent multiple clusters from being processed simultaneously (see Figure 7a). On average, we observe a performance gain of 7% across the seven DNN models.

Figure 9b plots a breakdown of the total amount of energy consumed in each case, distinguishing between the main components of the architecture: Global Buffer, Multiplier Network, Distribution Network and Reduction Network. As we can see, energy reductions are not very significant, ranging between 1% and 6% (4% on average). The energy consumption in this case is mainly dominated by the number of operations carried out during the execution, which is the same regardless of how the filters are rearranged. In this case, most of the observed energy gains come from both the reduction of static energy due to the decrease in execution time, and the reduction in the number of messages to be sent through the DN (i.e., running more clusters simultaneously increases the ability to exploit the DN's multicast package delivery).

Another important observation that we would like to make is that we have found out a huge difference in terms of layers sensitivity when it comes to LFF. In particular, there are some layers that experience a large impact in terms of energy and performance when LFF is applied, but this benefit is subsequently hidden by other low-sensitive layers. As an illustrative example, Figure 9c shows the performance and energy gains for 14 representative layers (in terms of LFF sensitivity) of Resnets-50. The results are normalized to the obtained for the non-scheduled execution. Here, we can see how the layers can be divided into three different groups according to their sensitivity to the LFF scheduling heuristic. The first five layers show no benefit at all as LFF is not capable to leverage the MSs better (gains of 0.01% on MS utilization), so they fall into the *low-sensitive*

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

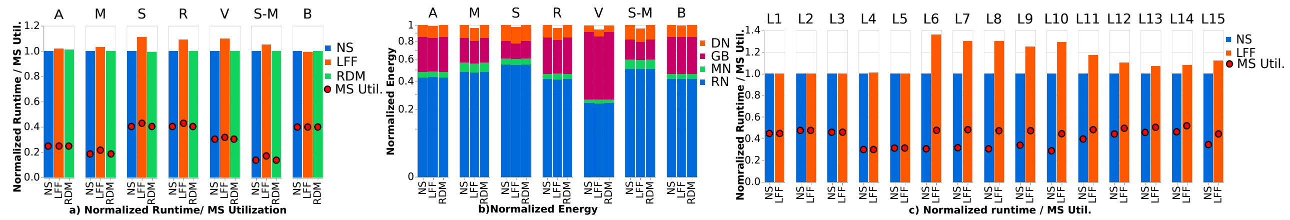


Fig. 9: (a) Normalized runtime for the LFF static scheduling strategy with respect to a non-scheduled execution. (b) Normalized energy for the LFF static scheduling strategy with respect to a non-scheduled execution. (c) Normalized runtime and energy for the LFF static scheduling strategy with respect to a non-scheduled execution for 14 layers of Resnets-50.

layer category. Contrarily, for the next five, significant improvements (up to 36% and 16% performance and energy gains, respectively, in layer *L6*) are observed, and would therefore be those that make up the *high-sensitive* layer category. These significant gains are explained by increased MS utilization, which ranges from 9% to 13% (11% on average). Finally, the *medium-sensitive* layer category would be comprised of the last five layers, for which MS utilization benefits varying from 8% to 4% (5% on average) are obtained, leading to lower performance advantages between 17% and 8% (energy benefits range between 5% and 1% in this case).

The obtained results point out that more intelligent heuristics capable of adapting the filter scheduling strategy to the specific features of each layer could bring large benefits in terms of performance and energy savings when running sparse DNN models. This observation paves the way for the development of much more sophisticated strategies aimed to improve the energy efficiency of next-generation DNN accelerators.

VII. Conclusions

We demonstrate that as the complexity of the microarchitecture of DNN accelerators grows, the analytical models typically used to estimate their performance and energy figures are not able to capture many important subtleties that simulation at cycle level does. STONNE aims to fill this gap, paving the way towards rapid and accurate prototyping of next-generation DNN accelerator architectures. Through three use cases, we demonstrate the huge potential of STONNE to assist the research community in the pursuit of better DNN accelerator architectures.

Acknowledgments

We thank Raveesh Garg for his help adding additional support for sparse computation. We also thank the anonymous reviewers for their feedback that helped improve the positioning of this work. This work was supported by RTI2018-098156-B-C53 (MCIU/AEI/FEDER,UE), NSF OAC 1909900 and US Department of Energy ARIAA co-design center. F. Muñoz-Martínez was supported by grant 20749/FPI/18 from Fundación Séneca.

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] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *44th Int'l Symp. on Computer Architecture*, 2017.

[3] 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," *International Symposium on Computer Architecture (ISCA)*, pp. 27–40, Jun. 2017.

[4] 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.

[5] W. Lu *et al.*, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," *IEEE Int'l Symp. on High Performance Computer Architecture*, pp. 553–564, May 2017.

[6] V. A. et al., "SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks," *int'l Symp. on Computer Architecture*, pp. 662–673, Jun. 2018.

[7] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen, "DaDianNao: A neural network supercomputer," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 73–88, Jan. 2017.

[8] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2018.

[9] E. Q. et al., "SIGMA: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," *Int'l Symp. on High-Performance Computer Architecture*, Mar. 2020.

[10] Y.-H. Chen, J. S. Emer, and V. Sze, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 12–21, Jun. 2017.

[11] H. Kwon *et al.*, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," *Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2019.

[12] 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," *2015 International Symposium on Computer Architecture (ISCA)*, pp. 92–104, Jun. 2015.

[13] V. J. Reddi *et al.*, "MLPerf inference benchmark," *47th Int'l Symp. on Computer Architecture (ISCA)*, May 2020.

[14] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv: 1710.01878v2 (2017)*, Oct. 2017.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," *arXiv preprint arXiv: 1512.02325v5 (2015)*, Dec. 2015.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv: 1810.04805v2 (2019)*, May 2019.

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

[22] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C.Berg, and L. FeiFei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, Dec. 2015.
[23] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollar, "Microsoft coco: Common objects in context," *arXiv preprint arXiv: 1405.0312v3*, Feb. 2015.
[24] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv: 1606.05250v3*, Oct. 2016.
[25] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.
[26] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, "Rsim: simulating shared-memory multiprocessors with ilp processors," *Computer*, vol. 35, no. 2, pp. 40–49, 2002.
[27] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.
[28] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.
[29] N. Binkert and B. et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, 2011.
[30] Y. e. a. Sun, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, p. 197–209.
[31] A. P. et al., "Timeloop: A systematic approach to dnn accelerator evaluation," *Int'l Symp. on Performance Analysis of Systems and Software*, Mar. 2019.
[32] "Pytorch," https://pytorch.org/.
[33] A. S. et al., "SCALE-Sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv: 1811.02883v1 (2019)*, Feb. 2019.
[34] "DNNSim repository," https://github.com/isakedo/DNNsim.
[35] "MAERI code v1," https://github.com/hyoukjun/MAERI.
[36] "SIGMA code v1," https://github.com/georgia-tech-synergy-lab/SIGMA.
[37] S. Xi *et al.*, "SMAUG: End-to-end full-stack simulation infrastructure for deep learning workloads," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, Nov. 2020.
[38] "Caffe website," http://caffe.berkeleyvision.org/.
[39] R. C. Martin, "Design principles and design patterns," *objetcmentor*, Apr. 2000.
[40] 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.
[41] T. Krishna, H. Kwon, A. Parashar, M. Pellauer, and A. Samajdar, "Data orchestration in deep learning accelerators," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 3, pp. 1–164, 2020.
[42] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274, Feb. 2020.
[43] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292 – 308, Jun. 2019.
[44] M. Pellauer *et al.*, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," *International Conference on Architectural Support for Programmling Lenguages and Operating Systems (ASPLOS)*, pp. 137–151, Apr. 2019.
[45] 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.
[46] S. L. et al., "DRAMsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, 2020.
[47] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O'Connor, "Learning sparse matrix row permutations forefficient spmm on gpu architectures," *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 48–58, 2021.

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.

A. Artifact Appendix

A.1 Abstract

In this work we present STONNE (Simulation TOol of Neural Network Engines), a cycle-level microarchitectural simulation framework that can plug into any high-level DNN framework as an accelerator device and perform full-model evaluation (i.e. we are able to simulate real, complete, unmodified DNN models) of state-of-the-art rigid and flexible DNN accelerators, both with and without sparsity support. As a proof of concept, we use STONNE in three use cases: i) a direct comparison of three dominant inference accelerators using real DNN models; ii) back-end extensions and iii) front-end extensions of the simulator to showcase the capability of STONNE to rapidly and precisely evaluate data-dependent optimizations. In order to make the tool available for reproducing the experiments for i) and iii), in this artifact we include the first version of the simulator. For ii) we include a different version contained separately in the repository. For the three cases we include the data sets required to generate the charts and the script to carry out the experiments.

A.2 Artifact check-list (meta-information)

- **Program: STONNE.**
- **Compilation: STONNE can be easily compiled using g++ 11 or superior.**
- **Output: data outputs are included in dataset_raw.tar.gz.**
- **Experiments: The kernels and DNNs used during this paper might be reproduced using the Pytorch interface of STONNE and its STONNE user interface (just configure the simulator to run the dimensions indicated in the paper).**
- **How much disk space required (approximately)?: 200MB.**
- **How much time is needed to prepare workflow (approximately)?: 2 hours.**
- **How much time is needed to complete experiments (approximately)?: 5 days (depends on the number of nodes you have to parallize the executions).**
- **Publicly available?: Yes.**
- **Code licenses (if publicly available)?: MIT License.**
- **Archived (provide DOI)?: 10.5281/zenodo.5516222**

A.3 Description

This artifact contains the tools and data sets required to generate the charts provided in the paper.

A.3.1 How to access

The entire artifact can be found by means of the next DOI: 10.5281/zenodo.5516222. Here, we can clearly distinguish between several compressed (and not compressed) files:

- *analytical_model.tar.gz*: Contains the analytical model used to compare against the cycle-level simulator STONNE (see Figure 1).
- *benchmarks_pytorch.tar.gz*: Contains the benchmarks to generate the results of the three use cases (Section VI).
- *dataset_raw.tar.gz*: Contains the raw data generated after running the simulations.

  Figures.tar.gz: Contains the raw figures presented in the manuscript.

  IISWC.ipynb: This jupyter notebook file contains the scripts used to generate the figures from the raw data.

  stonne_original.tar.gz: STONNE simulator modeling MAERI, TPU and SIGMA able to reproduce the experiments of sections VI.A and VI.C.

  stonne_snapea.tar.gz: Adds Snapea functionality to reproduce the experiments of section VI.B.

A.4 Dependencies

STONNE does not require any dependence. However, running the benchmarks with Pytorch requires some previous packages:

- Torchvision `https://github.com/pytorch/vision`
- transformers `https://github.com/huggingface/transformers`
- numpy.
- Cmake.

Please, make sure all the dependencies are solved before running any benchmark. Besides, make sure the installation of the dependencies does not remove the current version of pytorch to install another one. In order to avoid so, it is highly recommended to install both the torchvision and transformers packages from sources. Here, we present an example of how to install torchvision package from source:

```
$ git clone https://github.com/pytorch/vision
$ cd vision
$ python setup.py install
```

Please, follow the official instructions for each dependency if something occurs.

A.5 Installation

The installation of STONNE explained below is applicable for both files *stonne_original.tar.gz* and *stonne_snapea.tar.gz*.

The STONNE User Interface 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 runs it with random tensors. To reproduce the experiments shown in Figure 1 and Section V it is necessary to use this interface. The instalation of STONNE is very simple:

```
$ tar -xzvf stonne_original.tar.gz
$ cd stonne/stonne
$ make
```

These commands will generate a binary file 'stonne/stonne'. This binary file can be executed to run layers and gemms with any dimensions and any hardware configuration. All the tensors are filled using random numbers.

On the other hand, to reproduce the experiments shown in the use cases, it is necessary to execute the real benchmarks. To do so, the STONNE Pytorch frontend interface must be installed.

The pytorch-frontend is located in the folder 'pytorch-frontend' and this basically contains the Pytorch official code Version 1.7 with some extra files to create the simulation operations and link them with the 'stonne/src' code. The current version of the frontend is so well-organized that running a pytorch DNN model on STONNE is straightforward.

Installing Pytorch-frontend will make the same effort as installing the original Pytorch framework. First, you will need Python 3.6 or later and a C++14 compiler. Also, it is highly recommended to install Anaconda environment. Once you have Anaconda installed (`https://www.anaconda.com/products/individual`) you can proceed to the installation. Next, we summarize the installation process on Linux (Please refer to the original Pytorch documentation to learn how to install it in other operating system):

```
$ cd pytorch-frontend/
$ python setup.py install
```

Next, run the next commands, which will install the pytorch_stonne package that will be used for the connection with STONNE.

```
$ cd stonne_connection/
$ python setup.py install
```

A.6 Notes

STONNE simulator and an extended explanation of the tool may be found on `https://github.com/stonne-simulator/stonne`.

Authorized licensed use limited to: Universidad Politecnica de Cartagena. Downloaded on April 04,2024 at 11:08:55 UTC from IEEE Xplore. Restrictions apply.