Design Space Exploration of Accelerators and End-to-End DNN Evaluation with TFLITE-SOC

Nicolas Bohm Agostini†, Shi Dong†, Elmira Karimi†, Marti Torrents Lapuerta*,
José Cano‡, José L. Abellán§, David Kaeli†
†*Northeastern University*, Boston, MA, USA **Barcelona Supercomputing Center*, Barcelona, Spain
‡*University of Glasgow*, Glasgow, UK §*Universidad Católica San Antonio de Murcia*, Murcia, Spain

Abstract—**Recently there has been a rapidly growing demand for faster machine learning (ML) processing in data centers and migration of ML inference applications to edge devices. These developments have prompted both industry and academia to explore custom accelerators to optimize ML executions for performance and power. However, identifying which accelerator is best equipped for performing a particular ML task is challenging, especially given the growing range of ML tasks, the number of target environments, and the limited number of integrated modeling tools. To tackle this issue, it is of paramount importance to provide the computer architecture research community with a common framework capable of performing a comprehensive, uniform, and fair comparison across different accelerator designs targeting a particular ML task.**

To this aim, we propose a new framework named TFLITE-SOC (System On Chip) that integrates a lightweight system modeling library (SystemC) for fast design space exploration of custom ML accelerators into the build/execution environment of Tensorflow Lite (TFLite), a highly popular ML framework for ML inference. Using this approach, we are able to model and evaluate new accelerators developed in SystemC by leveraging the language's hierarchical design capabilities, resulting in faster design prototyping. Furthermore, any accelerator designed using TFLITE-SOC can be benchmarked for inference with any DNN model compatible with TFLite, which enables end-to-end DNN processing and detailed (i.e., per DNN layer) performance analysis. In addition to providing rapid prototyping, integrated benchmarking, and a range of platform configurations, TFLITE-SOC offers comprehensive performance analysis of accelerator occupancy and execution time breakdown as well as a rich set of modules that can be used by new accelerators to implement scaling up studies and optimized memory transfer protocols.

We present our framework and demonstrate its utility by considering the design space of a TPU-like systolic array and describing possible directions for optimization. Using a compression technique, we implement an optimization targeting reducing the memory traffic between DRAM and on-device buffers. Compared to the baseline accelerator, our optimized design shows up to 1.26× speedup on accelerated operations and up to 1.19× speedup on end-to-end DNN execution.

Index Terms—**DNN accelerator framework, Systolic array, Memory compression, Hardware-software co-design**

I. Introduction

A rapidly growing number of domain-specific accelerators [1]–[6] were delivered to the market in recent years. Specifically, the growth of Machine Learning (ML) accelerators has been fueled by the demonstrated benefits of sophisticated deep learning (DL) applications. Enabled by hardware acceleration, optimized DL algorithms are able to reduce the time required to produce accurate inference results. Accelerating these computations has been the focus of many hardware and software innovations. In terms of ML hardware, vendors have designed and deployed ML accelerators for a range of systems [1], [2], [7], [8]. Some of these accelerators can be easily benchmarked and compared, but problems arise when attempting to do the same with new accelerators due to lack of support in ML frameworks [9]–[11] and associated benchmark suites [12].

Recent work has proposed a number of microarchitectural improvements on existing spatial accelerator designs [13]–[16], including the use of optimized systolic arrays and other customized solutions [17]–[19]. It is challenging to decide which architectural features are most beneficial for a given application or how new features compare to existing designs. In many of these studies, runtime performance is reported using simulation models or by building implementations in a particular technology node.

There is a real need for an end-to-end framework that enables prototyping and fair comparison of new accelerator designs. Currently, new designs are compared based on their potential compute throughput, estimated power consumption and operating frequency, as shown in Table I. However, such approach is not fair since each design is evaluated in a unique environment, as described in Table II. We can see that each design is evaluated using different technology nodes, number of cores, precision and, sometimes, different workloads. This makes direct comparisons challenging and limits our ability to measure the impact of the proposed architectural enhancement.

Modeling and quantitatively evaluating the benefits of different DNN accelerator designs should be the first step before investing additional time and money on a production prototype. Recent work [23]–[26], discussed in detail in Section V, highlights how design space exploration and end-to-end evaluation of DNNs can be implemented at different levels of abstraction and integrated with custom made [24]–[26] or production [23] frameworks. These contributions effectively provide a path for fair comparison of accelerator enhancements. However, we observe that no recent work has yet enabled system modeling at all levels of a Hardware-Software (HW/SW) co-design project [27], [28]. Instead, they usually provide analytical modelling (the highest level of abstraction) or cycle-based modelling (the lowest level of abstraction before RTL). To address this gap, we propose TFLITE-SOC (System On Chip)

TABLE I: Performance numbers of DNN accelerators proposed by academia or industry. GOPS/W and GOPS/MHz were calculated based on the numbers on the left side of the table. Missing values are due to unreported metrics or could not be inferred from the orignal work.

Platform	Year	GOPS	Power [W]	Clock [MHz]	GOPS/W	GOPS/MHz
TPU V1 [1]	2017	92,000.0	75.00	700.0	1,226.7	131.43
TPU V2 [20]	2017	184,000.0	1120.00	700.0	164.3	262.85
TPU V3 [20]	2018	492,000.0	1800.00	940.0	273.3	523.40
Edge TPU [21]	2018	4,000.0	2.00	500.0	2,000.0	80.00
EIE [22]	2016	102.0	0.59	800.0	172.9	0.12
SCNN [13]	2017	2,000.0	-	1,000.0	-	2.00
MAERI [17]	2018	33.6	4.20	200.0	8.0	0.17
Eyeriss V2 [18]	2018	-	-	200.0	-	-
OuterSPACE [14]	2018	2.9	23.99	1,500.0	0.1	0.00
SIGMA [19]	2020	10,880.0	22.33	500.0	487.2	21.76
V100 Tensor Cores [2]	2017	130,000.0	75.00	-	1,733.3	-
V100 GPU [2]	2017	15,700.0	300.00	1,246.0	52.3	12.60

TABLE II: Environments on which different accelerators have been benchmarked. The observation column points out how Table I metrics were obtained. Missing values are due to unreported metrics or could not be inferred from the orignal work.

Platform	Tech node	# of PEs	Precision	Observation
TPU V1	28nm	256x256 PEs	INT8	TOPS reported for theoretical maximum throughput.
TPU V2	>12nm	4x2x128x128 PEs	BF16	TOPS reported for theoretical maximum throughput per board. BF16 stands for brain floating format. A board has 4 chips, with 2 cores, with 1 matrix multiply units of 128x128 PEs.
TPU V3	>12nm	4x2x2x128x128 PEs	BF16	TOPS reported for theoretical maximum throughput per board. A board has 4 chips, with 2 cores, with 2 matrix multiply units.
Edge TPU	-		INT8	TOPS reported for theoretical maximum throughput.
EIE	45nm	64 PEs	FP32	TFLOPS reported from evaluation of the FC7 layer of AlexNet. It considers specific weight and actiavation sparsity.
SCNN	16nm	32x32 PEs	INT8	TOPS reported for theoretical maximum throughput.
MAERI	28nm	168 PEs	INT16	TOPS estimated from 100% utilization of all PEs. Power extrapolated based on 16 PEs implementation. Experiments performed using individual convolution layers of VGG16 and AlexNet.
Eyeriss V2	65nm	32x32 PEs	FixedP16	TOPS could not be calculated due to missing information. Paper reports 8x-256x speedup over Eyeriss v1. But Eyeriss v1 paper does not report TOPS or provide a proxy for calculation.
OuterSPACE	32nm	256 PEs	FP32	TFLOPS reported for average compute throughput on Florida SuiteSparse and Stanford Network Analysis Project matrices.
SIGMA	28nm	128x128 PEs	FloatingP	TFLOPS reported by multiplying the base dense TFLOPS with average efficiency computed across different sized GEMMs.
V100 Tensor Cores	12nm	640 Tensor cores	FP16	With support for higher precision at the cost of compute throughput.
V100 GPU	12nm	5376 CUDA cores	FP32	With support for FP64 at half the compute throughput

as an open-source project (available at https://github.com/tflite-soc/tflite-soc) that uses the SystemC [29] library for rapid system modeling and simulation within the TFLite [30] framework. The contributions of this paper are the following:

- We present TFLITE-SOC, a new framework that integrates an event-driven simulator (SystemC) in the TFLite ML framework, providing a path for HW/SW co-design of accelerators at different levels of modelling abstraction. It enables fast prototyping and possible path to synthesis of modeled designs by leveraging the SystemC's hierarchical design capabilities and high-level synthesis.
- TFLITE-SOC enables benchmarking and comparison of new accelerators while leveraging an incremental specification/architecture/communication refinement approach. As an use case of TFLITE-SOC, we implement a TPU-like [1] accelerator that can be used as baseline in the future.
- The framework incorporates the development of useful parameterized SystemC modules that can be used by new accelerators to evaluate scalability or enhanced communication protocols such as data compression of DRAM-accelerator traffic. TFLITE-SOC can model different system organizations (peripheral and embedded) and platform configurations (high performance, low power, and custom specification).
- We provide end-to-end evaluation of our simulated TPU-like accelerator and report performance results, i.e. latency breakdown and Processing Element (PE) utilization, for the execution of three state-of-the-art DNN models (ResNet101, MobileNet_V1 and MobileNet_V2). Our experiments uncovered insights for future accelerator design directions. In particular, latency related to memory traffic between DRAM and on-device buffers was high for specific platform configurations. We implemented a compression scheme that aimed to mitigate this performance bottleneck. This optimization provided up to 1.19× speedup on end-to-end DNN execution over the baseline accelerator.

The rest of the paper is organized as follows. Section II presents our target ML framework and overall runtime breakdown of the DNN models under study. In Section III we expose our TFLITE-SOC system, and we present its use cases in Section IV. The related work is presented in Section V. Finally, in Section VI we summarize our contributions and discuss directions for future work.

II. Target ML Framework and DNN models

Many ML frameworks have emerged to enable training and/or inference of DNN models and this work integrates with one of them, namely TensorFlow Lite (TFLite). Next, we justify our choice for the ML framework and present the runtime breakdown of different contemporary DNN models highlighting the importance of the 2D convolution (CONV2D) for acceleration.

A. *TensorFlow Lite Framework*

TFLite [30] is Google's official ML framework to run inference with TensorFlow models on mobile/embedded edge devices. As of 2020, TFLite has been deployed on more than 4 billion devices worldwide [31]. These platforms include Android, iOS, Linux IoT-enabled devices, and microcontrollers. TFLite also provides multi-threaded execution, GPU optimized kernels, and can be compiled to x86 platforms.

TFLite is simpler than other ML frameworks (e.g., TensorFlow, PyTorch, or Caffe) in many ways. TFLite focuses only on inference. It currently supports only a subset of the operations provided by TensorFlow. It is programmed primarily in C++, relying on the *Bazel* build system [32], compiler of choice, and optimized third party libraries to generate binaries for different target platforms.

Additionally, there are two key features in TFLite that helped us select it over other frameworks. First, to execute a DNN model, the only requirements to perform an inference are the TFLite runtime and the *model.tflite* file. The *model.tflite* file can be easily generated by TensorFlow using any pre-trained DNN model (a set of pretrained models can be downloaded from Google [33]). The second key feature is that TFLite has a built-in benchmarking infrastructure, which enables comprehensive per-layer analysis as well as the performance of the overall DNN execution. Overall, the framework's simplicity, wide deployment, and the key features discussed were primary factors in its selection.

TABLE III: Models used in this work and their inference latencies.

Model	Layers*	Size	†Latency	‡Latency
ResNet101_FP32 [34]	101	178.3MB	526.0ms	103.9ms
MobileNetV1_FP32 [35]	30	14.0MB	17.5ms	8.6ms
MobileNetV2_FP32 [36]	53	16.9MB	24.0ms	9.0ms
MobileNetV1_INT8	30	4.3MB	13.0ms	12.3ms
MobileNetV2_INT8	53	3.4MB	12.0ms	13.8ms

† Previously reported latency on Pixel 3 CPU with 4 threads [33].
‡ Measured latency on i7-8700 CPU with 4 threads.
* Number of CONV2D and/or DEPTHWISE_CONV2D layers.

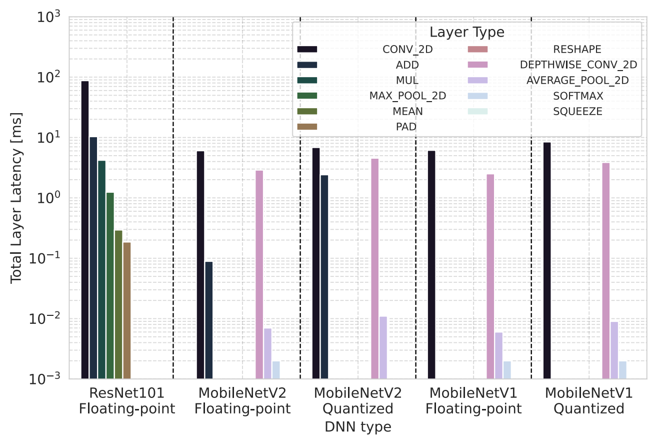


Fig. 1: Overall time spent during DNN inference per layer-type reported for the different DNN models of Table III.

B. *DNN Models and Runtime Breakdown*

Any model that can be converted into a *model.tflite* file can be executed with the TFLite runtime. In the DNN characterization experiments presented in this section and the accelerator evaluation presented in Section IV, our framework uses mainstream and publicly available Image Classification models with floating-point (FP32) and quantized (INT8) weights. The list of models is shown in Table III and were extracted from [33].

To identify which DNN operations are most time consuming, we used the TFLite benchmarking tool to execute the models in Table III. We ran this analysis on a system with an Intel i7-8700 6/12 cores/threads, and 64GB of DDR4@2666MHz memory. TFLite was compiled with the gcc-5.4.0 compiler and *-march=native* optimizations enabled, which include SSE4 and AVX2 extensions. These special x86 instructions only work with floating point datatypes (FP32) and, as observed in Table III, can help floating point models execute faster than their quantized (INT8) counterparts. Upon workload analysis, we observe that the CONV2D layers – as well as DEPTHWISE_CONV2D layers from MobileNet V1 and V2 models – are the most frequently executed and most time consuming layers. Figure 1 shows that, regardless of the network, the majority of the inference time is spent in the execution of these layers, which consume more than 82% of the total execution time in all five profiled models. In DNN models using FP32 precision, CONV2D and DEPTHWISE_CONV2D layers are at least one order of magnitude more expensive than other operations.

To improve the execution of the convolutional layers, most prior studies on ML acceleration have focused on optimizing individual kernels (i.e., the underlying algorithm used to implement each layer's operation) [37]–[41]. Since TFLite uses Multiple Channel Multiple Kernel (MCMK) convolutional algorithms that are implemented with General Matrix Multiplication (GEMM) kernels [37], [42], we design our accelerator around the execution of these GEMM calls. For instance, if we consider a simple matrix multiplication kernel ($C = A \times B$), the filter input for the convolution operation becomes matrix A, and the input activations of the convolution operation are expanded into a matrix composed of column-transformed windows (*im2col*), which becomes matrix B. TFLite automatically transposes matrix B in memory as a data layout optimization, resulting in a more efficient memory access pattern during the GEMM call.

III. TFLITE-SOC

To enable accelerator prototyping and benchmarking, we have extended TFLite [30] by modifying specific kernels, building the environment in which SystemC simulation executes. Figure 2 presents a high-level diagram of our framework. TFLITE-SOC is comprised of the SystemC modules and bi-directional interfaces displayed inside the dashed box. Our framework required small modifications to *Bazel* build files to support SystemC, and minor additions to TFLite methods. These additions include the *Stimulus* module, responsible for intercepting the data that will be sent to the accelerator and to initiate simulation; and the *Monitor* module, responsible for sending results back to TFLite and producing/managing simulation reports. These modules are connected to the Device Under Test (DUT), i.e. the accelerator, using selected interfaces. We can model different types of interface timing specifications, such as PCI-Express when modeling an external/peripheral accelerator, or an AXI/HyperTransport bus for an embedded accelerator. The DUT module can also be used outside of this framework – this module is developed independently of any of TFLite's classes or methods. This design decision can be useful if the user wants to synthesize the design implemented by the DUT.

In a typical DNN execution, the *TF Operations of Interest* that make use of the underlying GEMM calls (e.g., CONV_2D or MUL) are "intercepted" and executed by the modeled accelerator. Currently, TFLITE-SOC only intercepts GEMM calls, but future work could also accelerate other types of calls/DNN operations.

Finally, TFLITE-SOC reports simulated accelerator performance results for the layers/kernels offloaded to the DUT in isolation. It can also account for x86 latency results of layers executed outside of the DUT, which enables end-to-end inference runtime evaluation. TFLITE-SOC is an open source project available to the community.

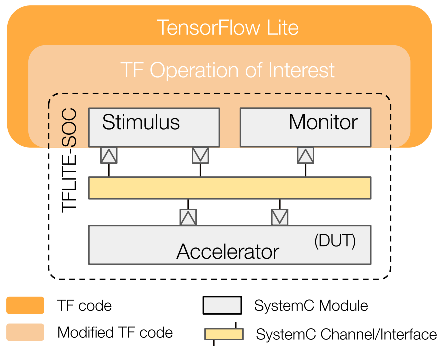


Fig. 2: TFLITE-SOC - Integration of SystemC with TFLite. Arrows in SystemC modules represent initiators or targets of TLM transactions.

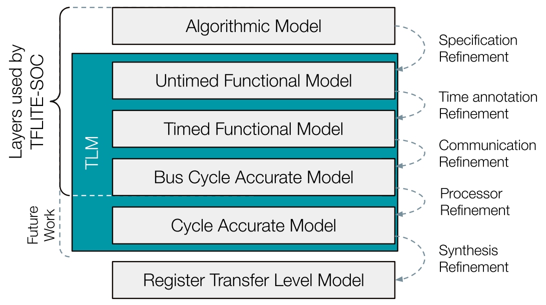


Fig. 3: SystemC incremental modeling in TFLITE-SOC. The labels on the right side specify the system refinements required to achieve the accuracy of different layers of abstraction. Currently, TFLITE-SOC uses abstractions based on the first four layers.

A. SystemC

SystemC is an ANSI C++ class library that has been developed to support system-level design [29]. The SystemC library provides a set of classes and macros to facilitate seamless HW/SW co-simulation of different stages of a HW/SW project. SystemC also offers many features, including a set of abstractions that facilitate system modeling:

- **Modules** are C++ classes containing processes or other modules as part of a hierarchy. They are used to describe the structural connection of the design's components.
- **Processes** are class methods that describe functionality/behaviour of a module. They model independent functionality that can happen currently with other processes.
- **Events** enable synchronization between processes, allowing for cycle/timing-based abstractions. *Events* at a given simulation step can trigger the execution of processes in different modules.
- **Interfaces, Channels and Signals** convey communication between different modules. *Interface classes* are used to declare the access methods that a channel implements. *Channels* provide a concrete implementation of the access methods declared by one or more interfaces. This implementation can be time annotated during the communication refinement step. *Signals* model "wires" with instantaneous propagation.

These features are part of the SystemC IEEE standard and are designed to separate the details of communication between modules and their specific implementation. This design approach is further enabled by Transaction Level Modeling (TLM) [43] and is used at the different levels of abstraction, as shown in Figure 3. With TLM, modules communicate through transactions using the access methods of a connected channel. In a TLM system, the interfaces and channels are what separate communication from computation.

Another feature of our framework is that SystemC is based on an event-driven simulation engine. Working at higher levels of abstraction (above RTL), event-driven simulation is faster than cycle-based simulation [29], [44], [45].

B. Developing with TFLITE-SOC

With SystemC, TFLITE-SOC supports incremental design refinement at different system scales for the modeled accelerator. This is highly efficient, as the HW/SW designer can focus on the desired functionality at scale (e.g., processing element array, scratchpad module, control unit, etc.) without impacting other modules expressed at different levels of abstraction. Incremental refinement is supported within TFLITE-SOC as shown in Figure 3. This approach enables us to evaluate a new accelerator architecture by using the following steps:

1) Identify the set of modules that we can leverage from existing TFLITE-SOC accelerators and the modules that need to be developed to implement new micro-architectural modifications.
2) Create the Algorithmic and Untimed Functional models for the new modules.
3) Time-annotate the new modules to make them operate in the Timed Functional Model abstraction.
4) Select the proper underlying implementations (i.e. bus type and specifications) for channels/interfaces to make them operate in the Bus Cycle Accurate abstraction.

Existing TFLITE-SOC modules are configurable by modifying module/accelerator parameters. These parameters control the architecture topology, memory and compute dimensions, and timing behaviour. Some of the available configurations will be discussed later in Section IV-C.

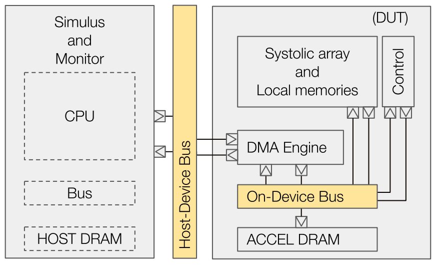
IV. Case Studies and Analysis

To demonstrate the utility of TFLITE-SOC's modeling capabilities and explore the range of analysis available, we consider the design space of an accelerator and evaluate different design specifications leveraging TFLITE-SOC's parameterized modules. We validated the functional correctness of the simulations of the accelerator with its different configurations. We match accelerator outputs with those produced by standard TFLite execution. Functional verification was performed on both FP32 and INT8 data types.

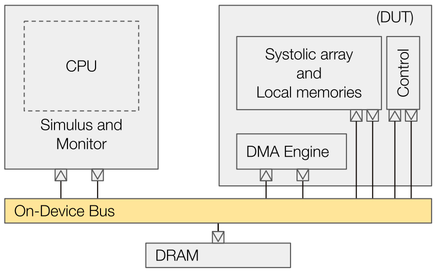
A. Baseline Accelerator Model

Similar to other accelerator studies [13], [15], [16], we select a TPU-like accelerator [1] as a baseline model and implement it in TFLITE-SOC. Our baseline accelerator uses a weight stationary [46] systolic array architecture that, depending on the DNN model, uses either FP32 or INT8 data types. Using our framework, this accelerator can be modeled as a peripheral device, connected to the host system over a peripheral interface (similar to PCIe), or as an embedded device, connected to the CPU using a standard bus interface (similar to AXI, or HyperTransport). Figures 4a and 4b highlight the key structural differences between these two system organizations, which are representative of different use cases of DNN accelerators [1], [21].

Figure 5 shows the module-based implementation of the devices shown in Figures 4a and 4b. It provides details of the



(a) Baseline TPU-like accelerator as a peripheral device.



(b) Baseline TPU-like accelerator as an embedded device.

Fig. 4: Block diagram for the different system organizations of the baseline accelerator. Dashed modules are not implemented and their high-level functionality is abstracted in the first parent module (i.e., the Stimulus and Monitor module).

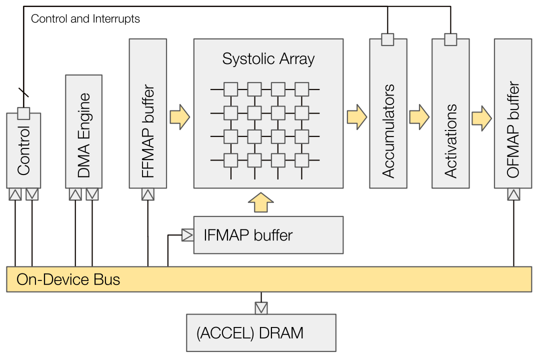


Fig. 5: Details of the baseline accelerator. The arrows represent the FIFO channels.

architecture of the Systolic Array block of Fused Multiply-Add (FMA) elements (i.e., Processing Elements or PEs) as well as key buffers and other modules present in the TPU design. Similar to [1], systolic execution expects the data to arrive for processing at regular time intervals. CISC-like instructions are sent to the TPU to be decoded and then the control unit guides the TPU's execution. Since we lack a driver API for this accelerator, our control unit, implemented as an Untimed Functional Model, receives the target operation and input data dimensions and generates the correct signals and events to carry out computation. Aside from the control unit, the other modules displayed in Figure 5 have undergone time annotation and communication refinements (see Figure 3). The modules shown in Figures 4 and 5 are parameterized and allow to select the number of PEs for the **Systolic Array** block. Note that proper tiling is performed to execute operations using data with dimensions larger than the PE array or the

TABLE IV: Pattern Encoding for FPC algorithm. Courtesey of [47].

Prefix	Pattern#	Encoded Pattern	Original Data	Compressed Data	Total Data Size (data + metadata)
001	1	Zero word	Z_{32}	-	0 + 3 bits
010	2	Word with repeated bytes	$N_8N_8N_8N_8$	N_8	8 + 3 bits
011	3	4-bit sign-extended	$X_{28}N_4$	N_4	4 + 3 bits
100	4	One byte sign-extended	$X_{24}N_8$	N_8	8 + 3 bits
101	5	Halfword sign-extended	$X_{16}N_{16}$	N_{16}	16 + 3 bits
110	6	Halfword padded with zero halfword	$N_{16}Z_{16}$	N_{16}	16 + 3 bits
111	7	Two halfword, each a byte sign-extended	$X_8N_8X_8N_8$	N_8N_8	16 + 3 bits
N/A	8	Uncompressed	N_{32}	N_{32}	32 + 0 bits

Z: Zero bits, **X**: All ones or all zeros bits, **N**: No any specific pattern,
The subscripts show the number of bits each symbol represents. Total Data size shows the size of the compressed data.

on-device memory (IFMAP, FFMAP, OFMAP buffers). The architecture of the baseline accelerator uses two levels of tiling. The first level tries to best utilize on-device buffers and minimize DRAM reads. The second level focuses on improving utilization of the PE array, while minimizing reads to the on-device buffers. Tile sizes are selected based on on-device buffer sizes and PE array sizes. Additionally the weight stationary nature of the modeled accelerator is taken into consideration to try to minimize reads from on-device buffers.

B. Enhancing the Baseline Accelerator

In this work, we explore a number of modifications to the baseline platform. The modifications were selected to showcase TFLITE-SOC's modeling flexibility and to evaluate the impact of Frequent Pattern Compression [48] applied to *On-Device Bus* communication, a feature not seen in previous DNN accelerator work. Later, in Section V, we summarize other architectural features proposed in the literature that could be implemented within TFLITE-SOC. Here we consider changes to the number of PEs, different platform specifications (e.g. interface bandwidth), and compression/decompression of input and output data transfers to reduce the time spent during DRAM-accelerator communication over the *On-Device Bus*.

Typically, compression techniques exploit common data patterns to reduce the effective number of bytes stored and/or transferred. Patterns are regularities in data that can arise from the intrinsic properties of what is stored in the data or are generated as a result of specific data transformations. Next, we describe some well-known sources that lead to predictable patterns in DNN weights and input activations:

- In order to prevent many output activations from saturating due to using a large single weight value, DNN training techniques can induce weights to converge to smaller values. This can result in weights with a limited dynamic range.
- ReLu operations transform negative numbers into zeros.
- Since the data sent to the accelerator was generated by an *im2col* transformation, the final representation requires repeated values and zero values.
- Many DNN applications present input data with spatial correlation (e.g., images or text) or temporal correlation (e.g., sound and video). Regions in these inputs may vary a little when compared to their neighbours.

Frequent Pattern Compression (FPC) [48] encodes multiple common data patterns that are observed in scientific applications. It uses a smaller number of bits that are pattern dependent to represent the input value. Zero words, repeated words, and narrow words encoding examples are shown in Table IV. For instance, for Pattern#1, blocks of 32 zero bits can be represented by 3 bits of metadata; with Pattern#2, four repeated words of 8 bits (32 bits total) can be represented by 11 bits (8 bits of data + 3 bits of metadata).

According to Almadeen and Wood [48], FPC can compress and decompress 16-bit, 32-bit, 512-bit data (depending on the pattern) with a compression latency of 3 cycles and a decompression latency of 5 cycles. Given this order of magnitude, FPC may not be ideal when compressing data transfers from first-level caches, but represents negligible latency when considering transfers from DRAM which typically take hundreds of cycles to complete. Therefore, we leverage FPC to optimize streamed data transfers over the *On-Device Bus* shown in Figures 4a and 4b. These transfers occur whenever an input, filter or output feature map (IFMAP, FFMAP, OFMAP) is read from or written to *DRAM* memory. We later evaluate this optimization by accounting for the impact of FPC compression and decompression on streamed transactions sent across the *On-Device Bus*. During modeling, for simplicity, we focused on 32-bit windows. This optimization was implemented as an alternative communication refinement.

C. Experiments

To explore the features available in TFLITE-SOC, we present two use-case studies that evaluate the accelerator and the associated system organizations previously described in Section IV-A. In our simulations, we model technology parameters based on recent TPU implementations [1], [21] and the specifications found for low power devices, such as FPGAs made for small embedded applications [49]. These parameters are shown in Table V. They are used as simulation configurations that dictate timing behaviour of the time-annotated SystemC modules. For end-to-end evaluation, we consider the DNN models listed in Table III.

In the following experiments, we modify the accelerator using the TFLITE-SOC framework in order to identify bottlenecks during end-to-end DNN execution. The TFLITE-SOC output also provides the total latency of the DNN layers that are not executed on the accelerator. This information has been obtained using the x86 host system described in Section II.

TABLE V: Specifications of DNN accelerators modeled with TFLITE-SOC.

Configuration	**PER**	**MOB**	**LPW**
Topology (Figure 4)	Peripheral	Embedded	Embedded
Host-Device Bus BW	14 GiB/s	N/A	N/A
On-Device Bus BW	14 GiB/s	14 GiB/s	0.6 GiB/s
FMAP buf. to PE BW	167 GiB/s	167 GiB/s	2 GiB/s
FMAP buf. size (total)	12 MiB	6 MiB	0.5 MiB
DRAM size	8 GiB	1 GiB	0.5 GiB
PE latency	1 Cycle	1 Cycle	1 Cycle
Accelerator frequency	700 MHz	200 MHz	200 MHz
Example device	TPU	Smartphone	FPGA (Low Power)

The systolic array uses PEs that support the same basic data-type assumed in the DNN model (i.e., FP32 and INT8). This impacts DRAM/FMAP tiling, chip area and energy consumption. Tiling is modeled accordingly, but area and energy consumption analysis are not yet supported by TFLITE-SOC.

During the experiments, we explore three different configurations which are explained below. The specifications details are provided in Table V.

- **PER** - representing a peripheral device similar to a TPUv1.
- **MOB** - representing an embedded device that would be part of a mobile SoC device chipset, such as a smartphone.
- **LPW** - representing an embedded device that has power/frequency limitations, such as an FPGA or small microcontroller.

Scaling up experiment - We use parameterized SystemC modules implemented in TFLITE-SOC to change the number of PEs in the *Systolic Array* block. We model blocks of 256×256, 128×128 and 64×64 PEs, often used in production designs. We report on the performance of different system organizations of Figure 4 and configurations of Table V. Finally we discuss how scaling-up impacts total latency and PE utilization.

On-Device Bus trafic with FPC experiment - Here, we analyze the impact of using FPC compression of DRAM memory transfers streamed over the On-Device Bus. In this scenario, our baseline implementation is a systolic array accelerator with 256×256 PEs (similar to TPUv1 [1]) using an embedded system organization (refer to Figure 4b). The baseline implementation is compared to an accelerator design enhanced with an On-Device Bus. We focus on specific configurations/DNN models that present high DRAM latency, and so stand to benefit greatly from this optimization.

D. Results

Figures 6 and 7 present stacked bars to break down per-module latency contributions for end-to-end DNN inference. The bars are grouped by network model, datatype, and accelerator configuration, with the baseline (256×256 PEs) on the left of each group of bars. Note that the absolute runtime contributions of the "*Other (x86)*" segments are the same for the same DNN model/Datatype, but different across different DNN models/Datatypes combinations.

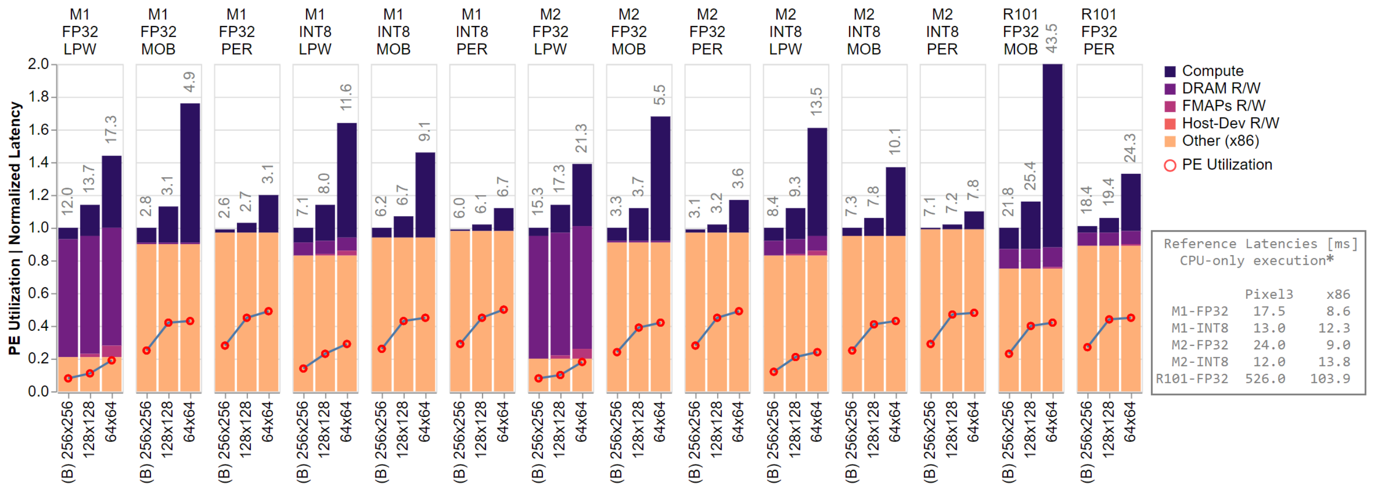
In Figures 6 and 7 we can see how the per-module latency is affected by the PE array size/configuration and modeled enhancements. The average PE array utilization (percent of active PEs) is also shown on the y-axis (e.g., the PE utilization of the simulated baseline for the configuration M1-FP32-LPW is 8%) and end-to-end latency is annotated on top of each bar. All of the metrics in these figures are generated by TFLITE-SOC. These figures showcase the wealth of information generated by our framework. After simulation completes, similar information is generated on a per-layer basis. In the discussion below, we attempt to highlight insights derived from the observed trends.

We first analyze the scale-up results in Figure 6. They show that when we use larger systolic arrays (approaching the baseline size) the average PE array utilization always decreases. This behaviour has been documented in prior studies [20], [24] and occurs due to difficulties when mapping GEMM operations with irregular shapes. Input matrices fall into this category, when they have a dimension smaller than the width of the PE array. This is common in the first convolutional layers of a DNN because of the small dimensions of the IFMAPs (or FFMAPs). Increasing the size of the systolic array increases the number of idle PEs during these stages of DNN execution. However, this is not the case for the entire run as input dimensions increase beyond PE array dimensions after the first (10%-25%) convolutional layers.

Even though utilization is lower for the baseline configuration, *(B)* 256×256 PEs, it always achieves the lowest latency thanks to higher compute throughput. However, at ~1.6× better PE utilization on average, a 128×128 array presents as a great candidate for a scaled-out (multiple PE arrays) accelerator design. Supporting scale-out experiments will be integrated in TFLITE-SOC in a future release. All of these example accelerator configurations are available in TFLITE-SOC to jumpstart future design exploration.

Next, we discuss the effects of the FPC compression feature applied to configurations that presented high *DRAM R/W* latency. The benefits of this technique are shown in Figure 7. First, we observe that the largest performance gains happen on the *LPW* configuration executing floating-point models. While we can achieve a speedup of up to 1.19× on DNN inference, *LPW* is a configuration more commonly used for quantized models. In more common use cases, such as the *MOB* configuration executing a floating-point model, or the *LWP* configuration executing quantized models, the end-to-end latency speedup is smaller and ranges from 1.02-1.05×. However, not all layers are executed on the accelerator and will be computed by the CPU. The latency associated with CPU layers is represented by the "*Other (x86)*" segment. If we ignore this latency, we observe a peak speedup of 1.26× with the *MOB* configuration running ResNet101. In ResNet101, if we compare our mobile configuration execution (R101-FP32-MOB) to the Pixel3 execution, we observe a speedup of 25×.

This speedup is achieved thanks to a significant reduction of On-Device Bus traffic. Reducing memory traffic also helps to increase PE utilization across different model configurations.



Reference Latencies [ms]
CPU-only execution*

	Pixel3	x86
M1-FP32	17.5	8.6
M1-INT8	13.0	12.3
M2-FP32	24.0	9.0
M2-INT8	12.0	13.8
R101-FP32	526.0	103.9

Fig. 6: Scale-up results and breakdown of inference time spent in each accelerator module during DNN execution. The line plots show the average PE utilization. The end-to-end latency in ms is provided atop each bar. **M1**: MobileNetV1, **M2**: MobileNetV2, **R101**: ResNet101, **FP32**: Floating-point model, **INT8**: Quantized model, **PER**: peripheral configuration, **MOB**: mobile configuration, **LPW**: low power configuration, **(B)**: baseline to which other bars are normalized, $*$: CPU latencies as reported in Table III.

Upon deeper investigation, we observed that this increase is more evident for accelerator calls operating on large IFMAP and FFMAP inputs, which were previously staling because needed tiles were not readily available in their respective on-device-buffers.

In order to better understand how FPC compression of the bus data provides performance benefits, Figure 8 shows a comparison between using uncompressed data (our baseline) versus compressed data. Each bar plots the sum of each individual feature map's size required by the DNN computations in the accelerator. Values are normalized to the size of the uncompressed data, with the effective size shown in GB above each bar. The presented patterns were described in Table IV. We observe that in floating-point DNN models, the most frequent pattern is zero words (Pattern 1) which can be significantly compressed, resulting in a 30% savings in terms of communication traffic during inference. The 8-bit quantized DNNs use feature maps with more complex patterns, including sign extensions (Patterns 4, 5 and 7) and zero padding (Pattern 6). The effective savings for the quantized networks is around 10%. The presence of dynamically changing patterns in each network highlights the importance of a dynamic compression scheme such as FPC. However, our analysis shows that accelerators specialized to run DNNs with floating-point values should focus primarily on zero words pattern (i.e., 32 consecutive zero bits).

Figure 8 shows the aggregated contributions, but does not show the contributions of individual feature maps across different kernel calls. Looking deeper into this question, we observed that the most prevalent pattern types remain consistent over the same DNN model/data type combinations, but exhibit some variation in pattern frequencies across different layers. Because the benchmarking tool uses a random input to feed the DNNs, most of the observed patterns are present in the weights (FFMAP) and outputs (OFMAP). We believe that this is the worst case scenario, i.e. inputs that are not random would present more compressible patterns in the IFMAPs, possibly providing even higher speedups with our technique.

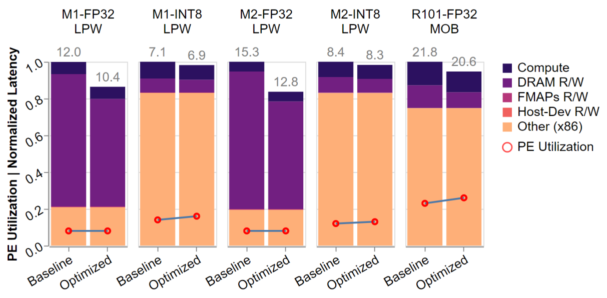


Fig. 7: FPC enhancement results and breakdown of inference time spent in each accelerator module during DNN execution. All experiments use a 256×256 systolic array in the embedded system organization. The line plots show the average PE utilization. The end-to-end latency is shown atop each bar.

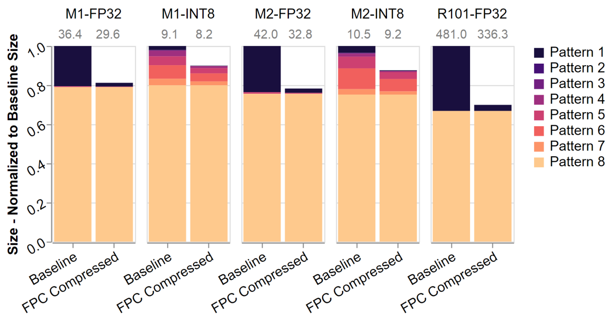


Fig. 8: Size of the uncompressed and compressed feature maps for each individual DNN model. Note that the y-axis begins at 50%. Values atop of each bar indicate the total data size represented by the bar in MB.

V. Related Work

Recent work has explored enhancements to systolic array accelerators. Chen et al. [18] proposed a network of PEs to perform spatial tiling of input data, across all dimensions, to better utilize the accelerators when executing convolutional kernels. Parashar et al. [13] describe a data flow that maintains sparse weights and activations using a sparse encoding, eliminating unnecessary data transfers, and reducing storage requirements. Kung et al. [15] improve sparse convolutional neural networks by combining sparse columns of a convolutional filter into a dense column, effectively improving utilization efficiency. Other studies [50], [51] explore optimizing the mapping and scheduling of computations, either statically or dynamically. These features can be implemented in our baseline accelerator by modifying the control logic and module interconnections.

In the field of DNN accelerator frameworks, Xi et al. [25] proposed SMAUG, the first DNN framework for end-to-end DNN model simulation based on gem5-Aladdin, a SoC pre-RTL simulator. SMAUG consists of a custom Python API that describes the DNN model and includes a complete software stack that finds optimized tiling strategies to best utilize the modeled accelerator in gem5-Aladdin. Instead of using a custom DNN framework, TFLITE-SOC integrates with Tensorflow Lite, which already has wide deployment and will be supported for the foreseeable future. SCALE-SIM is a Python-based cycle-accurate and configurable systolic array accelerator simulator proposed by Samajdar et al. [24]. They use SCALE-SIM to provide analytical modeling to estimate the runtime of individual DNN operations on a systolic array. This information is used to determine the best size, aspect ratio, and number of partitions to achieve the best performance for a given operation. STONNE [23] is also a cycle-accurate simulation framework written in C++ that enables end-to-end evaluation of accelerator architectures. It provides a path to build DNN accelerators based on general building blocks that feature flexibility. STONNE is the first simulator framework integrated with a commercial DNN framework (Caffe). Unfortunately, SCALE-SIM and STONNE only target simulation. Future accelerator synthesis would require translation or re-implementation of the architecture with a language that enables high-level-synthesis, such as SystemC, or with hardware description languages, such as Verilog, SystemVerilog or VHDL.

MAESTRO [26] is an analytical model that describes the behavior of DNN accelerators. It can estimate execution time, energy efficiency, and hardware costs of a design without requiring explicit RTL/cycle-level simulation. Unlike other frameworks that are based on compute-centric notations, MAESTRO uses data-centric notation to represent dataflows. Similar to MAESTRO, Timeloop [51] uses a data-centric model and can aid in the search for optimal dataflow configurations in accelerators with different architectures. We believe that MAESTRO or Timeloop, which only provide analytical models, could be used along with TFLITE-SOC for the first step of the design space exploration.

VI. Conclusion

In this paper we present the design and use-cases for TFLITE-SOC. TFLITE-SOC is fully integrated in TensorFlow Lite, an industry-leading ML framework, enabling end-to-end DNN inference evaluation of new accelerators. Under the hood our framework uses SystemC to provide hierarchical modeling, simulation capabilities and rapid design prototyping. Testing out new architectural features can be done with minimal effort given TFLITE-SOC's incremental design refinement model. We support the simulation of a TPU-like systolic array accelerator with different PE array dimensions, system organizations (peripheral and embedded) and platform configurations (high performance, low power, and custom specification). To demonstrate TFLITE-SOC's utility and flexibility, we investigate our accelerator's design space, identifying bottlenecks and suggesting improvements. In a case study, we implemented and evaluated an optimization focusing on reducing the increased latency associated with streamed memory transfers. In future work we plan to improve TFLITE-SOC by providing energy estimation reports using Accelergy [52], the capability of performing scale-out studies, enabling a streamlined path for design synthesis, and implementing baselines for new DNN accelerator architectures.

References

[1] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, 2017, pp. 1–12.

[2] NVIDIA Corporation, "Nvidia Tesla V100 GPU Architecture," *White Paper*, no. v1.1, p. 53, 2017.

[3] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[4] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, "A practical reconfigurable hardware accelerator for Boolean satisfiability solvers," *Design Automation Conference (DAC)*, p. 780, 2008.

[5] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A Genomics Co-processor Provides up to 15,000× acceleration on long read assembly Yatish," *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 5, no. 3, pp. 226–233, 2018.

[6] S. Mookherjee, L. Debrunner, and V. Debrunner, "A low power radix-2 FFT accelerator for FPGA," *Conference Record - Asilomar Conference on Signals, Systems and Computers*, pp. 447–451, 2016.

[7] D. Schor, "Inside Tesla's Neural Processor In The FSD Chip," 2019.

[8] Qualcomm, "SNPE: Snapdragon Neural Processing Engine," 2016.

[9] M. Abadi *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[10] Facebook, "PyTorch: tensors and dynamic neural networks in Python with strong GPU acceleration," 2017.

[11] Xiaomi, "MACE is a deep learning inference framework optimized for mobile heterogeneous computing platforms," 2019.

[12] P. Mattson *et al.*, "MLPerf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, pp. 8–16, 2020.

[13] A. Parashar *et al.*, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," *IEEE International Symposium on Computer Architecture (ISCA)*, may 2017.

[14] S. Pal *et al.*, "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 724–736.

[15] H. T. Kung, B. McDanel, and S. Q. Zhang, “Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining under Joint Optimization,” *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 821–834, 2019.

[16] N. K. Jha, S. Ravishankar, S. Mittal, A. Kaushik, D. Mandal, and M. Chandra, “DRACO : Co-Optimizing Hardware Utilization , and Performance of DNNs on Systolic Accelerator,” *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020.

[17] H. Kwon, A. Samajdar, and T. Krishna, “MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects,” *Architectural Support for Programming Languages (ASPLOS)*, vol. 53, no. 2, pp. 461–475, 2018.

[18] 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 OF SOLID-STATE CIRCUITS*, jul 2018.

[19] E. Qin *et al.*, “SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training,” *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 58–70, 2020.

[20] N. P. Jouppi *et al.*, “A domain-specific supercomputer for training deep neural networks,” *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[21] CORAL-AI, “CORAL System-on-Module datasheet,” 2019.

[22] S. Han *et al.*, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” *ACM SIGARCH Computer Architecture News*, 2016.

[23] F. Munoz-Martınez, M. E. Acacio, J. L. Abellán, and T. Krishna, “STONNE : A Detailed Architectural Simulator for Flexible Neural Network Accelerators,” *ArXiv*, pp. 1–8, 2020.

[24] A. Samajdar, J. Moritz, J. Yuhao, Z. Paul, W. Matthew, and M. Tushar, “A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim,” *IEEE International Symposium on Performance Analysis of Systems and Software*, 2020.

[25] S. L. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, and D. Brooks, “SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads,” *ArXiv*, pp. 1–14, 2019.

[26] 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 Using MAESTRO,” *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 754–768, 2019.

[27] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O’Boyle, and A. Storkey, “Characterising across-stack optimisations for deep convolutional neural networks,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, September 2018, pp. 101–110.

[28] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan, and X. Chen, “Convergence of Edge Computing and Deep Learning: A Comprehensive Survey,” *IEEE Communications Surveys & Tutorials*, pp. 1–1, 2020.

[29] J. Aynsley, “OSCI TLM-2.0 language reference manual,” *Open SystemC Initiative (OSCI), Tech. Rep*, no. July, p. 194, 2009.

[30] Google, “Deploy machine learning models on mobile and IoT devices,” 2018.

[31] K. LeViet, “How TensorFlow Lite helps you from prototype to product,” 2020.

[32] Bazel, “Build and test software of any size, quickly and reliably,” 2020.

[33] Google, “TFLite Hosted models - Image Classification,” 2020.

[34] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” *European conference on computer vision*, pp. 630–645, 2016.

[35] A. G. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” *ArXiv*, 2017.

[36] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” *IEEE Conference on Computer Vision and Pattern Recognition*, jan 2018.

[37] Google, “gemmlowp: a small self-contained low-precision GEMM library,” 2018.

[38] NVIDIA, “NVIDIA cuDNN,” 2017.

[39] L. Cavigelli, M. Magno, and L. Benini, “Accelerating real-time embedded scene labeling with convolutional networks,” *Proceedings - Design Automation Conference*, vol. 2015-July, 2015.

[40] Z. Ji, “ILP-M Conv: Optimize Convolution Algorithm for Single-Image Convolution Neural Network Inference on Mobile GPUs,” *ArXiv*, 2019.

[41] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, “Low-memory GEMM-based convolution algorithms for deep neural networks,” *ArXiv*, 2017.

[42] G. Guennebaud, B. Jacob *et al.*, “Eigen v3,” http://eigen.tuxfamily.org, 2010.

[43] IEEE, “IEEE Standard for Standard SystemC Language Reference Manual,” *IEEE Std 1666-2011*, vol. 2002, no. March, pp. 1–638, 2012.

[44] Y. Sun *et al.*, “MGPUSim: enabling multi-GPU performance modeling and optimization,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 197–209.

[45] R. Dömer and D. D. Gajski, “Comparison of the Scenic Design Environment and the SpecC System,” 1998.

[46] V. Sze, S. Member, Y.-H. Chen, S. Member, T.-J. Yang, and J. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” Tech. Rep., 2017.

[47] M. K. Tavana, Y. Sun, N. Bohm Agostini, and D. Kaeli, “Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-GPU systems,” *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019*, no. February, pp. 664–674, 2019.

[48] A. R. Alameldeen and D. A. Wood, “Frequent pattern compression: A significance-based compression scheme for l2 caches,” *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[49] Xilinx Inc., “Zynq-7000 SoC Data Sheet,” vol. 190, pp. 1–21, 2018.

[50] B. Liu *et al.*, “Addressing the issue of processing element underutilization in general-purpose systolic deep learning accelerators,” *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pp. 733–738, 2019.

[51] A. Parashar *et al.*, “Timeloop: A Systematic Approach to DNN Accelerator Evaluation,” *Proceedings - 2019 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019*, pp. 304–315, 2019.

[52] Y. N. Wu, J. S. Emer, and V. Sze, “Accelergy: An architecture-level energy estimation methodology for accelerator designs,” *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, vol. 2019 Novem, 2019.