Journal of Parallel and Distributed Computing 173 (2023) 140–151

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

SECDA-TFLite: A toolkit for efficient development of FPGA-based DNN accelerators for edge inference

Jude Haris [a,*], Perry Gibson [a,*], José Cano [a,*], Nicolas Bohm Agostini [b,*], David Kaeli [b,*]

[a] University of Glasgow, UK
[b] Northeastern University, USA

ARTICLE INFO

Article history:
Received 21 April 2022
Received in revised form 20 October 2022
Accepted 8 November 2022
Available online 15 November 2022

Keywords:
DNN accelerator design
Design methodology
Hardware-software co-design
SystemC
Simulation
HLS

ABSTRACT

In this paper we propose SECDA-TFLite, a new open source toolkit for developing DNN hardware accelerators integrated within the TFLite framework. The toolkit leverages the principles of SECDA, a hardware/software co-design methodology, to reduce the design time of optimized DNN inference accelerators on edge devices with FPGAs. With SECDA-TFLite, we reduce the initial setup costs associated with integrating a new accelerator design within a target DNN framework, allowing developers to focus on the design. SECDA-TFLite also includes modules for cost-effective SystemC simulation, profiling, and AXI-based data communication. As a case study, we use SECDA-TFLite to develop and evaluate three accelerator designs across seven common CNN models and two BERT-based models against an ARM A9 CPU-only baseline, achieving an average performance speedup across models of up to 3.4× for the CNN models and of up to 2.5× for the BERT-based models. Our code is available at `https://github.com/gicLAB/SECDA-TFLite`.

1. Introduction

Deep Neural Networks (DNNs) have demonstrated high accuracy in learning tasks, such as image classification [21], speech recognition [51] and many more. However, current solutions that attempt to deploy DNNs on low-power and resource-constrained edge devices (e.g., smartphones, tablets, and wearables) are inefficient [14], presenting challenges that can span several levels of the hardware/software stack to run efficiently [44]. To address these inefficiencies, hardware-based optimizations have been proposed to reduce DNN inference costs. This active research area includes ISA-level extensions to CPUs and GPUs [27,30], as well as TPUs [20] and other custom hardware solutions for FPGAs and ASICs [6,22].

There are a variety of tools available for developing FPGA-based DNN accelerators for edge devices [3,25], some of which feature model-specific tuning [28,45,50]. Flexibility is valuable, since the layers and sizes of DNNs can vary, especially given the rapid introduction of new and novel DNN architectures. Examples of the range of DNNs include the large convolutional layers of InceptionV3 [41], as compared to the small depth-wise separable convolutions of MobileNets [17]; or the convolutional neural network (CNN) architecture of ResNets [16], as compared to the transformer-based architecture [46] of models such as BERT [11].

Since resources are more limited on edge FPGAs, and DNN models are large in terms of their memory footprint and computational demands, a given DNN is unlikely to fit fully on an accelerator [14]. Thus for DNN inference, the accelerator must operate in close communication with the CPU, which requires careful co-design with the host CPU code to ensure that data is managed efficiently. Therefore, an effective design methodology for a DNN accelerator design should, for a given set of hardware resource constraints, produce performant accelerators that effectively leverage available resources and can respond quickly to changing workload requirements (e.g., introduction of new types of layers).

To tackle the challenges of designing DNN hardware accelerators, developers can utilize the SECDA methodology [15] (*SystemC Enabled Co-design of DNN Accelerators*). A key part of SECDA is that most hardware design is performed in simulation and can be easily synthesized on real hardware (i.e., an FPGA) for more robust testing. The approach uses a unified codebase and encourages tight integration of the target Application Framework (i.e., the DNN framework such as TensorFlow's [1] TFLite or PyTorch [31] Mobile) with the accelerator's software driver and hardware design. The hardware-software co-design of the accelerator driver alongside the accelerator reduces the time for deployment to the target hardware once the accelerator design is ready for real hard-

* Corresponding authors.
E-mail address: j.haris.1@research.gla.ac.uk (J. Haris).

https://doi.org/10.1016/j.jpdc.2022.11.005

ware evaluation. The unified codebase is achieved by leveraging SystemC [19], which enables low-cost simulation and High-Level Synthesis (HLS) to an FPGA.

The first step in any instantiation of a SECDA-based workflow is the initial integration with the target Application Framework. This instantiation includes sub-steps such as setting up the simulation environment and providing a path to offload host-side computations to the accelerator designer. From there, developers can begin defining their first accelerator designs and follow the iterative design loops of SECDA to produce optimized designs.

However, our observation is that the number of strong candidate application frameworks for DNN accelerators is limited. TVM [4], TFLite, TensorRT [8], PyTorch [31] Mobile, and ONNXRuntime [10] constitute some of the most relevant DNN inference frameworks used today. Thus, various developers creating their integration for these frameworks may represent redundant work, as they will all perform the same initialization steps. Due to TFLite's inherent connection with the popular Tensorflow ML Framework, and its relatively mature support for features such as quantization, sparsity, and custom operations, we decided to develop our SECDA toolkit for development and deployment using TFLite. Therefore, in this work, we propose SECDA-TFLite, a new open source toolkit that extends the TFLite DNN framework, such that it can be more easily used to develop new DNN hardware accelerators using the SECDA design methodology.

The SECDA-TFLite toolkit leverages the TFLite delegate system to provide a robust and extensible set of utilities for integrating DNN accelerators for any DNN operation supported by TFLite. Ultimately, this increases hardware accelerator developers' productivity, as they can begin developing and refining their design more quickly. To aid in this, the toolkit consists of four key components: SystemC Integration, Simulation Profiling, Data Communication, and Multi-threading libraries. These provide the essential utilities that enable developers to start developing their designs. In future, we hope to develop further SECDA toolkits for other popular DNN frameworks such as TVM, TensorRT, and ONNXRuntime.

We demonstrate the utility of SECDA-TFLite with a case study that provides accelerators for both CNN and transformer-based DNN architectures. We port and improve the two CNN accelerator designs from the original SECDA paper [15], integrating and exploiting the more robust tooling provided by SECDA-TFLite. In addition, we bring a new accelerator design targeting the BERT [11] family of models, which uses a transformer neural architecture. The target device is the PYNQ-Z1 board [32], a platform with a dual-core CPU and an edge FPGA. The contributions of this paper include the following:

- We propose SECDA-TFLite, a new open source toolkit to enable the efficient development of custom DNN hardware accelerators for TFLite, focused on the edge, leveraging the SECDA design methodology.
- We describe the key features of the SECDA-TFLite toolkit (simulation, profiling, and data communication utilities) and how it integrates with the upstream TFLite framework. previously developed using SEDCA with our new toolkit for a newer version of TFLite, as well as developing a new accelerator design targeting transformer models.
- We evaluate the performance of the accelerators developed for our case study using several state-of-the-art DNN models. Our CNN accelerators are comparable to or outperform their original counterparts, demonstrating that SECDA-TFLite does not introduce significant overheads compared to a more ad-hoc integration. For our new BERT accelerator, we outperform the CPU-only inference by an average of 2.1× and 2× in terms of inference time and energy efficiency, respectively.

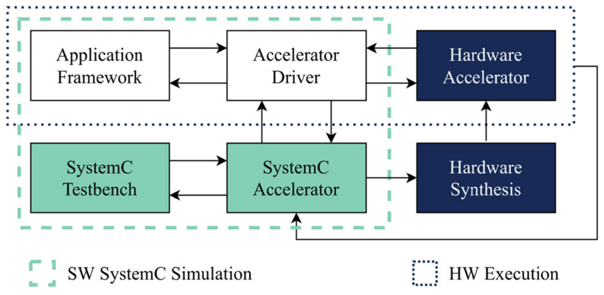

Fig. 1. Overview of the SECDA methodology. Components in the dashed lines correspond to the design in simulation, and components in the dotted lines correspond to the design running on real hardware. *Application Framework* and *Accelerator Driver* software are common to both.

2. SECDA methodology

SECDA (*SystemC Enabled Co-design of DNN Accelerators*) is a hardware-software co-design methodology to efficiently produce optimized DNN inference accelerators for edge devices using FPGAs. SECDA uses SystemC [19] as an accelerator simulation framework, which also allows candidate designs to be iterated upon efficiently. Using SystemC HLS, we can produce a synthesizable design from the same accelerator definition. The co-design of a software accelerator driver and a hardware accelerator is achieved by integrating SystemC's simulation features within the target edge-based DNN framework, allowing designers to quickly test potential optimizations such as varying data transfer and tiling strategies. Embedding the simulation environment and the hardware accelerator into the same software environment reduces the costs of exploring hardware/software co-design trade-offs via simulation relative to synthesizing the design on an FPGA with every change. Fig. 1 shows a high-level overview of the SECDA methodology, with the key components being briefly explained below.

2.1. Software components

The *Application Framework* is the DNN software framework that runs the target DNN models, from which we offload work to the accelerator. The *Accelerator Driver* is the software component in the co-design methodology, the bridge between the *Application Framework* and the hardware accelerator.

2.2. SystemC simulation

SystemC Simulation is the cornerstone of the SECDA methodology. Using simulation combined with HLS, we can gain insight into candidate designs. In SECDA, we use testbench and end-to-end SystemC Simulation. **SystemC Testbench simulation** is based on unit testing the accelerator design and its components on sets of input data, which enables developers to iteratively design accelerator components without running a full workload. **End-to-end SystemC Simulation** runs entire DNN models using our candidate accelerator designs, with the integration of the *Application Framework* via the *Accelerator Driver*. This tests the correctness of the full system and leverages the accelerator's per-component performance estimates to show metrics for full DNN workloads.

2.3. Design loop

The SECDA methodology relies on two different iterative design loops to explore the accelerator design space for DNNs. The most frequently used design loop is iterations in cheap SystemC Simulation (SW SystemC Simulation) [19], with the second loop being hardware benchmarking on FPGAs (HW Execution). Hardware benchmarking requires logic synthesis which is expensive in terms

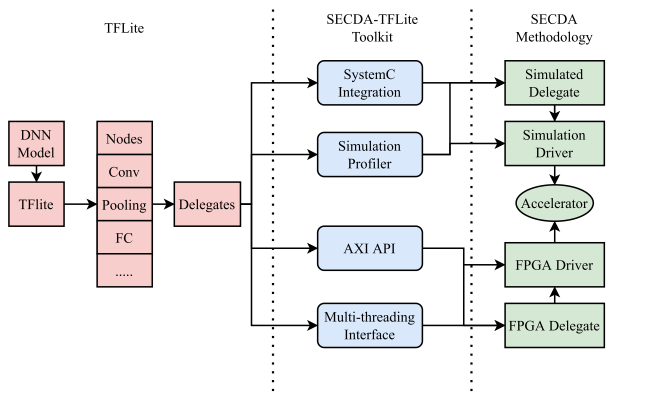

Fig. 2. Overview of the SECDA-TFLite toolkit and how it is used within the SECDA design methodology for TFLite.

of time, thus SECDA aims to minimize the number of times this occurs. SECDA achieves this by having the majority of the accelerator design performed in low-cost SystemC Simulation, which provides performance estimates of the accelerator design for a given DNN model. After a major design iteration, promising designs are fully synthesized and evaluated on the target FPGA.

3. SECDA-TFLite

SECDA is a generic methodology that can be applied to a variety of application frameworks (i.e., DNN inference frameworks), with the first step being to provide integration with the target framework. However, our observation is that there is a limited number of popular DNN inference frameworks, and thus this initial setup step can be reused between accelerator designs defined on the same framework, further reducing the barriers to developing new accelerators.

SECDA-TFLite is a TFLite specific toolkit that provides the initial development environment when following the SECDA methodology within TFLite, and a set of utilities to aid development. This enables the developer to begin prototyping and integrating their new design with significantly reduced initial setup costs. While the original SECDA case study was embedded within TFLite [15], the integration was ad-hoc since it was focused on providing support for a single accelerator rather than a generic integration for future developers. SECDA-TFLite aims to be a robust open source toolkit for anyone who wants to develop new DNN accelerators within TFLite.

The rest of the section expands on the key aspects of SECDA-TFLite and how it streamlines FPGA-based DNN accelerator development for TFLite using the SECDA methodology. Section 3.1 gives context on where the integration with TFLite occurs, Section 3.2 describes the 4 main components of the toolkit, and Section 3.3 discusses the toy accelerator design that we provide as a starting point for developers, which leverages all features of the SECDA-TFLite toolkit.

3.1. SECDA-TFLite delegates

The SECDA-TFLite toolkit provides integration with the TFLite DNN inference framework and provides a starting point for developers to produce new DNN hardware accelerators. This integration exploits TFLite's so-called "delegate" system, where operations from DNNs can be efficiently offloaded to the target accelerator while still providing the original CPU inference for non-accelerator layers. Fig. 2 shows an overview of the key aspects SECDA-TFLite's integration in TFLite. For future SECDA toolkits targeting other frameworks, we will exploit similar subsystems such as TVM's

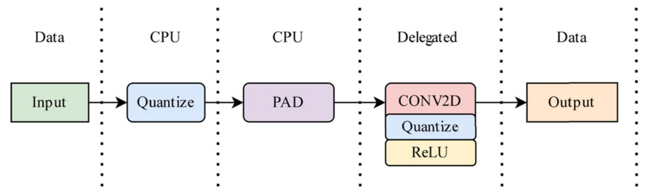

Fig. 3. A simplified example of a DNN running on TFLite, with some nodes running on the CPU, and a group of three running on a delegate.

BYOC,[1] or ONNX Runtime's *ExecutionProvider*. Section 3.1.1 gives a brief overview of the TFLite delegate system, while Sections 3.1.2 and 3.1.3 discuss the delegates that SECDA-TFLite provides to developers for developing their hardware designs.

3.1.1. TFLite delegate system

The TFLite delegate system [43] is available in later versions of TFLite with the purpose of providing simplified support for different hardware and software backends for DNN operations. While TFLite provides some delegates for Android and iOS devices, creating custom delegates is required for deploying new custom hardware accelerators. For DNN inference, TFLite delegates can be used to offload individual (or groups of) TFLite operations within a DNN model to other backends, including hardware accelerators. Fig. 3 shows an example of three operations in a DNN being executed on a delegate, with the rest of the operations being run using the default CPU runtime. In the example, the three nodes are grouped together, enabling the potential for further optimizations through the delegate.

In addition, creating custom delegates in TFLite can be cumbersome and require expertise to connect TFLite with low-level hardware drivers. Hence, the SECDA-TFLite toolkit provides the bulk of the initial delegate system required for developers to implement new DNN hardware accelerator designs following the SECDA methodology.

3.1.2. SECDA-TFLite simulation delegate

The purpose of the SECDA-TFLite simulation delegate is to allow end-to-end simulation as defined within the SECDA methodology within TFLite. The simulation delegate connects TFLite to simulated DNN accelerator hardware designs. Under SECDA, the SystemC simulation is required to provide a fast evaluation time for changes to the accelerator design, enabling verification of correctness and resource efficiency. To this end, the simulation delegate provides profiling tools to the developer in a sub-system called the *Simulation Profiler*, which is described in Section 3.2.2. These profiling tools can be extended to meet the developers' needs while providing the essential features required.

The simulation delegate is also used to co-design the accelerator driver, which is used to communicate with the accelerator. The accelerator driver is responsible for managing aspects such as tiling strategies, data preparation, output data unpacking, control flow, and generating accelerator instructions. Overall the efficiency of the accelerator driver can be very impactful on the overall runtime performance [47,49], hence why the co-designing of the accelerator and the accelerator driver is prioritized within the SECDA methodology.

3.1.3. SECDA-TFLite FPGA delegate

The purpose of the FPGA delegate is to provide the interface to versions of the accelerator running on real hardware, namely an

[1] "Bring Your Own Codegen" https://tvm.apache.org/2020/07/15/how-to-bring-your-own-codegen-to-tvm.

```
// SystemC Accelecrator.h
// Define profiling for hardware accelerator
ClockCycles *total = new ClockCycles("total", true);
ClockCycles *read = new ClockCycles("read", true);
ClockCycles *compute = new ClockCycles("compute", true);
ClockCycles *send = new ClockCycles("send", true);
std::vector<Metric *> profiling_vars = {total, read,
compute, send};
// -------------------------------------------------------
// C++ SimulationDelegate.cc
// Save profile for target hardware module after simulation
profile.saveProfile(accelerator.profiling_vars);
...
...
// Save all profiled data to csv
profile.saveCSVRecords(profile_output_file_name);
```

Listing 1: Example of using the Simulation Profiler to define, profile and export specific clock cycle metrics across simulation.

FPGA. The simulation delegate connects TFLite to the DNN accelerator hardware designs on an FPGA. Under SECDA, the purpose of running on real hardware is to identify bottlenecks that are not revealed through simulation, for instance, the impact of off-chip memory accesses. To this end, SECDA-TFLite provides the *Data Communication library*, discussed in Section 3.2.3. This provides approaches for the designer to convert simulation constructs defined for data communication into real hardware AXI data transfer implementation. Similarly, accelerator driver code can be threaded to improve the CPU-side performance, thus SECDA-TFLite provides a *Multi-Threading API* (see Section 3.2.3).

3.2. Toolkit

Along with the two TFLite delegates, which provide starting points for the integration and development of DNN hardware accelerator designs with TFLite, the SECDA-TFLite toolkit also provides four core components: SystemC integration, Simulation Profiler, Data Communication, and Multi-threading API; which helps the designer to develop their hardware designs. We describe the four components below.

3.2.1. SystemC integration

SystemC end-to-end simulation is key to the SECDA methodology, however TFLite does not natively support the SystemC simulation environment. Thus, with SECDA-TFLite, we define a software library within the TensorFlow Bazel [9] workspace, which can be included as a dependency when compiling any TensorFlow binaries and delegates. This provides the user with the necessary components for running SystemC simulation, enabling the SystemC API to initialize and bind hardware modules defined in SystemC to the simulation environment via the delegate calls.

As well as reducing the work developers must do to integrate SystemC, we also provide the *SystemC integration API* that allows developers to define data transfer between the simulated accelerator and the TFLite allocated data tensors. Thus, a key benefit of the SystemC integration library is that we can ensure SystemC simulation constructs are easily accessible throughout the delegate code, with standard boilerplate code predefined for TFLite.

3.2.2. Simulation profiler

End-to-end SystemC simulation can be used to quickly evaluate the potential performance impact of changes to the hardware and software components of the accelerator design, as well as verifying the correctness of the implementation. In order to profile the end-to-end simulation, the developer needs to add additional code to keep track of hardware and software metrics (such as simulated clock cycles spent), throughout the end-to-end DNN inference. SECDA-TFLite provides a system called the *simulation profiler*, which provides a method to define the different types of metrics to capture from the accelerator and software driver, along with common metrics developers will be interested in, such as the number of clock cycles.

Listing 1 shows an example of how a developer can use the simulation profiler to capture various clock cycle metrics for their hardware accelerator design. This saves the developer time by not having to define, capture, and process common profiling metrics manually. However, the simulation profiler is extensible to bespoke metrics not defined by SECDA-TFLite, such as accelerator instruction count. In addition, the simulation profiler provides an export function to a CSV file for analysis.

3.2.3. Data communication

Edge FPGA-based DNN accelerators require a high degree of data transfer between on-chip and off-chip memory, as the on-chip memory will have insufficient capacity to store all the weight and input data required through inference. Hence, during inference, new sets of data need to be sent to the accelerator to be processed, and the resultant data needs to be transferred back to the main memory. The Advanced eXtensible Interface (AXI) is the standard data interface for data movement between the FPGA and ARM CPU cores. Within SECDA-TFLite, we provide a simple AXI API to allow the designer to quickly implement data transfers between CPU and accelerator through the three main types of AXI data transfers: AXI4, AXI4Lite, and AXI-Stream. Much as with the other components of the SECDA-TFLite toolkit, this avoids the need for the hardware developer to define their own data communication methods while still providing the ability to adapt the system if required.

3.2.4. Multi-threading API

The multi-threading API consists of simple classes which help the delegate developer to define CPU-side tasks that need to be performed in a multi-threaded fashion. Most commonly, this will be in the *accelerator driver*, which needs to efficiently transfer data between the hardware accelerator and TFLite. The API allocates tasks to worker threads to execute. While threading is not required to improve the performance of the accelerator, it can improve the performance of any computationally expensive calls, such as data packing and unpacking to reduce the bottleneck of data preparation and storing. In addition, the API does not exclude the use of other common threading libraries, however, it is provided so developers can benefit from multi-threading in the context of TFLite without having to produce a custom solution.

3.3. Template delegate and SystemC DMA-engine

Along with the SECDA-TFLite toolkit, we provide a set of simulation and FPGA delegates for a toy accelerator. This simple design can serve as a quick starting point for developers, as well as showcase the usage and main features of SECDA-TFLite. The simulation and FPGA delegate also highlight the differences between the simulation driver and the actual AXI-based FPGA driver. We also provide a hardware definition of a simple DMA engine that can be used to simulate any data communication using the AXI-Stream interface.

4. Case study

To demonstrate the value of the SECDA-TFLite toolkit and how it provides a foundation for efficiently developing DNN accelerators within TFLite using the SECDA methodology, we develop three different FPGA-based DNN accelerator designs targeting Convolutional and BERT-based DNN models.

We develop the designs for resource-constrained edge devices such as our target device, the PYNQ-Z1 board. For Convolutional

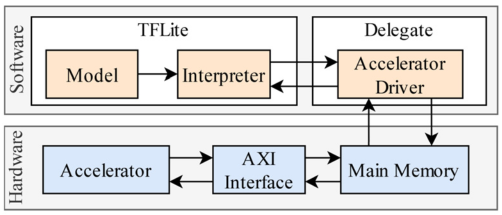

Fig. 4. Runtime model, common for all accelerators and DNN model types.

Neural Network (CNN) models, we port, improve, and integrate the Vector MAC (VM) and Systolic Array (SA) based-designs, which were previously defined within the original SECDA case study [15], using the SECDA-TFLite toolkit. Since the original case study, TFLite has changed significantly (from v2.2, to v2.7+), thus the newly defined accelerators now target signed 8-bit inference rather than unsigned 8-bit inference used in earlier versions of TFLite, which has been deprecated. Thus, our accelerators use signed 8-bit quantized DNN models, a popular machine learning approach that can reduce the inference time with a low accuracy penalty [52].

We accelerate convolutional layers, which in TFLite are implemented using the *GEMM convolution* algorithm. Thus, we develop the aforementioned custom accelerators and their respective drivers to reduce the inference time of the model. For models in the BERT family, we note that they contain high-level DNN layer structures commonly referred to as *transformer layers*. However, these transformer layers can be decomposed into several Matrix Multiplication operations, which are represented as Fully Connected (FC) layers within TFLite models. From our experiments, these FC layers are the most expensive in terms of computational requirements taking up to 66% of the overall inference time. Thus for BERT-based models, we develop a new GEMM-based accelerator design targeted to accelerate the FC layers within TFLite.

Fig. 4 shows the execution flow when performing DNN inference using our custom accelerators. The offloading of the computation to our accelerators is integrated through TFLite delegates, which we develop using the SECDA-TFLite toolkit. We describe the improved development environment made possible through the SECDA-TFLite toolkit and how it is used throughout the case study and provide details of the designs in the following sections.

In Section 4.1, we give a brief overview of how the SECDA design methodology is followed when using SECDA-TFLite. In Section 4.2 we discuss our three accelerator designs, with Section 4.3 giving details of their components. In Section 4.4, we briefly discuss some of the relevant features of the accelerators' supporting software.

4.1. SECDA-TFLite workflow

To develop new accelerators using the SECDA methodology, we need to instantiate the development environment within the application framework so that we can load and run models, as well as run simulations and synthesis of our candidate hardware designs. As discussed in Section 3, SECDA-TFLite provides the instantiation of the SECDA methodology within TFLite, thus avoiding many manual steps. Once this occurs, the developer can follow the SECDA methodology to develop and define their chosen accelerators, leveraging the utilities available in SECDA-TFLite. In the following sections, we briefly give an overview of how we did this for our case study.

4.1.1. Initialization

After setting up their environment using the utilities provided by SECDA-TFLite (as described in Section 3.2.1), the developer uses the SECDA-TFLite simulation delegate to make a delegate for their target accelerator, specifying features such as the operation type(s) they want to accelerate.

As in a typical SECDA workflow, for the first iteration of development, the developer may define a native C++ implementation of the target operation. The native C++ implementation can be a useful starting point from which the designer can develop the hardware design. Over time, the developer can replace this stub with SystemC hardware definitions, producing a viable first accelerator design.

4.1.2. SystemC simulation co-design/co-verification

After adding simple SystemC constructs to our accelerator module, we develop hardware components such as hardware buffers or processing units in SystemC to replace the initial implementation. Through developing the simulation testbench, a part of the SECDA methodology, we implement hardware components for computing the target operation (e.g., weight buffers, multipliers). Using the testbench and the end-to-end simulation environment, we go through several iterations where we fine-tune the design of our accelerator components. This fine-tuning is used to ensure each hardware component is efficient in terms of the hardware resources utilized and clock cycles spent. Additionally, this also includes making sure that the overall behavior of the accelerator architecture is efficient and that no component is creating a bottleneck.

To help perform this fine-tuning, we can use the profiling tool within SECDA-TFLite, as discussed in Section 3.2.2. Our pre-defined metrics, such as buffer utilization or clock cycle counts, can be used, or custom metrics can be defined. Using these metrics, the developer is able to adapt the hardware design and software driver iteratively. For example, reducing the number of clock cycles or changing the behavior of the *Accelerator Driver* to improve data reshaping.

4.1.3. Design loop

SECDA-TFLite provides developers with the utilities required to quickly instantiate an accelerator development environment. Once this is done, the main design loops of SECDA begin, where developers switch back and forth between evaluation on both simulated and FPGA-synthesized versions of their hardware designs, noting bottlenecks and making design iterations. A key benefit of using the SECDA-TFLite toolkit is that switching between simulation and FPGA evaluation is simplified by leveraging the delegate system.

4.2. Accelerator designs

Following the workflow described in Section 4.1, we developed and integrated three hardware accelerator designs within SECDA-TFLite. All three accelerator designs follow an output-stationary dataflow approach [23], which was chosen to remove the need to store many intermediate results on valuable on-chip memory, or incur costs associated with storing them off-chip. From the original SECDA case study, both the Vector Mac (VM) and Systolic Array (SA) designs have been updated to support per-axis quantization, as well as the signed-integer quantization scheme required by newer versions of TFLite. The new FC-GEMM accelerator used for the transformer model was developed entirely from scratch using the SECDA-TFLite toolkit. Table 1 contains the resource utilization for each accelerator design, which was estimated after HLS; all three designs were configured to run at 200 MHz. Note that although the number of DSPs varies across designs, all three designs contain the same number of MAC processing elements (PEs); due to DSP limitations, some of the PEs are instantiated using LUTs instead of DSPs. The following sections give a brief overview of each accelerator design.

Table 1
PYNQ Z1 FPGA resources utilized per accelerator design.

Accelerator \ Resource	BRAM	DSP	FF	LUT
Vector Mac	221	188	61127	50298
Systolic Array	160	196	59585	33139
FC-GEMM	224	164	35580	26585

4.2.1. Vector mac design (VM)

Fig. 5 shows an overview of the VM accelerator design that consists of four SIMD-style compute units, which we refer to as GEMM Units. We are limited to four GEMM Units by the resource constraints of the target device PYNQ-Z1. Each GEMM Unit broadcasts sets of weights and inputs to its internal MAC Units to produce a 4×4 output tile. Each output value is calculated using a set of four MAC Units, with the intermediate results reduced to the final output value through an adder tree. For off-chip communication, the design uses the AXI-Stream interface, leveraging utilities from SECDA-TFLite's Data Communication module. The AXI-Stream interface enables a simpler accelerator design that does not require specialized instructions.

4.2.2. Systolic array design (SA)

Fig. 6 shows an overview of the SA accelerator design. The design contains a single Compute Unit built as a 16×16 MAC-based systolic array, where each MAC Unit accumulates towards a single output value. MAC Units work by reading and storing the input and weight values of the neighboring MAC Units into their own registers. Hence, the systolic array moves input and weight values vertically and horizontally, respectively, once at the start of each step. The new data for the starting row and column of the MAC Units are read from a set of 32 data queues which are filled by the scheduler. Similar to the VM design, the SA design utilizes the AXI-Stream interface for data communication.

4.2.3. Fully connected GEMM accelerator (FC-GEMM)

Fig. 7 shows an overview of the FC-GEMM accelerator design, which we use to accelerate FC layers within transformer models such as BERT. The data movement between main memory and on-chip memory is performed with the AXIM interface, thus allowing the accelerator to directly access the main memory to load and store data. The AXIM interface decreases the need for complicated data packing and ordering from the accelerator driver, which is otherwise required by the two previous designs that used the AXI-Stream interface. This comes at the cost of potentially higher data transfer latency.

The design contains five key hardware modules: Fetch, Load, Scheduler, Compute, and Store Units. For further details on these modules, refer to Sections 4.3 and 4.3.5. The computation is performed by a single *Compute Unit*, that contains a $4 \times 4 \times 16$ MAC array. For FC-GEMM, we opt for a single large *Compute Unit* as the BERT models we target consistently contain large MatMul dimensions, which ensure the *Compute Unit* is fully utilized during GEMM, as opposed to multiple smaller compute units which are more efficient for smaller sized GEMM more prevalent in CNN models. This additionally simplifies the work of the *Scheduler Unit* as it only needs to decode the compute instructions for a single *Compute Unit*.

4.3. Accelerator components

Our accelerator designs are constructed with a variety of basic components, developed and tested both individually in the SystemC testbench and together in end-to-end simulation. All three designs contain similar components, although their behavior and connections vary. Adapting, reusing, and recomposing these components for new designs is a valuable feature of any hardware

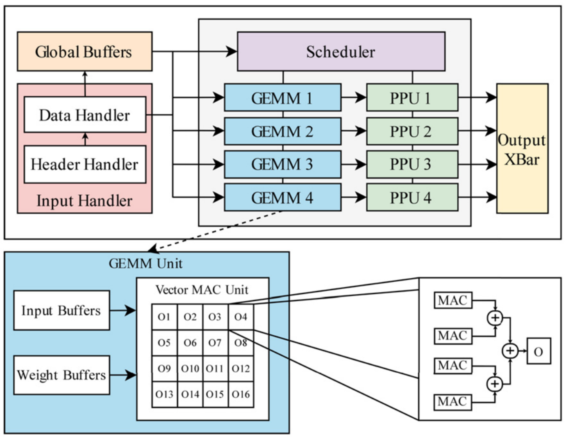

Fig. 5. Vector Mac accelerator design, featuring four GEMM Units.

design methodology, especially in DNNs where a given design may lose relevance quickly due to novel DNN workloads emerging. Next is a brief description of the major components.

4.3.1. The scheduler

The *Scheduler* orchestrates computations that occur within the processing units of each design. For the VM design, the *Scheduler* assigns work to each GEMM Unit, broadcasting weight data tiles to all GEMM Units and ensuring maximum weight data tile reuse to minimize redundant loads. For the SA design, the *Scheduler* feeds input and weight data to the corresponding data queues, which feed the outer MAC Units within the array. For both the VM and the SA designs, the *Scheduler* transfers quantization parameters for each data tile to the Post Processing Units. For the FC-GEMM design, the *Scheduler* simply decodes the compute instructions and directs the MAC Array to perform the GEMM computation.

4.3.2. The input handler

The *Input Handler* is only found in the VM and SA designs, as the FC-GEMM design has its own way of handling input data. The component receives all data sent by the driver from main memory via the AXI-DMA engine, as shown by the "AXI" component in Fig. 4. Metadata added by the driver is used to direct the incoming data to the appropriate accelerator buffers. The arrangement of the buffers varies between both VM and SA designs. The VM design makes use of local buffers within each GEMM Unit to store all input values and the active tile of weight data, with the global buffers used for storing all weights tiles. The SA design only uses global buffers for both input and weight data. Additionally, the *Input Handler* for both VM and SA designs unpacks and stores the quantization parameters required for post-processing.

4.3.3. Post processing unit (PPU)

The PPU is only used in the VM and SA designs, although the FC-GEMM's *Store* component performs some of the same functions. The PPU receives `int32` output tiles from their adjacent processing unit and applies the post-processing pipeline to obtain the quantized `int8` result tiles. Due to the change in the quantization scheme from earlier TFLite versions, the PPU was updated to read additional quantization parameters (e.g., the axis scaling factor) and applies them per output tile. By performing the post-processing steps within the accelerator rather than the CPU, we reduce the size of our output data by a factor of 4, which translates to significant inference time savings.

Additionally, the PPU performs all other functionality required by the quantization algorithm that needs to be applied to the outputs, including bias addition, scaling, and applying the activation function. For the VM design, there are multiple smaller PPUs that

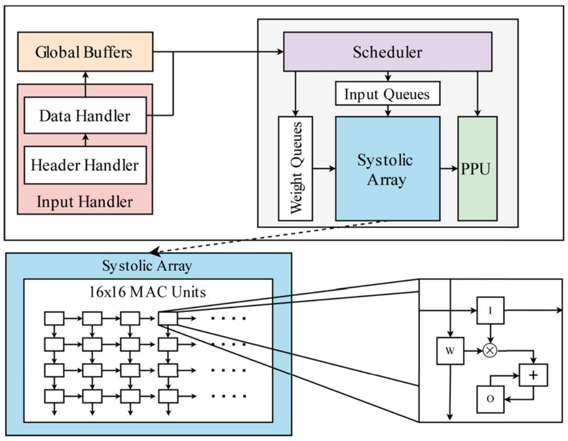

Fig. 6. Systolic Array accelerator design, featuring a 16x16 MAC Array.

process the output from each GEMM Unit, this ensures the GEMM Units can move to the next set of computations without waiting for PPU Units to become available. The PPU outputs are combined later by the *Output Crossbar*. In comparison, the SA design contains a single PPU that processes all the 16 × 16 output tiles and sends them back to the main memory.

4.3.4. Output crossbar

The Output Crossbar is unique to the VM design and is used to collect the output tiles from all PPUs. It rearranges the tiles such that the results are sent back to the main memory in the desired order (i.e., row-major ordering with a striding factor of 4).

4.3.5. FC-GEMM specific hardware

The FC-GEMM design differs significantly from the VM and SA designs, and it is a completely new accelerator design in this work. It features five key components: Fetch, Load, Scheduler, Compute, and Store Units.

- The **Fetch** Unit receives control signals to start the accelerator along with the number of instructions to load from main memory.
- The **Load** Unit receives instructions from the *Fetch* Unit to load data into either input, weight, or bias buffers. The instruction contains the main memory address along with the size and stride of the data that needs to be loaded.
- The **Scheduler** component is discussed in Section 4.3.1.
- The **Compute** Unit is a 4 × 4 × 16 MAC array, which simplifies the data orchestration within the accelerator.
- The **Store** Unit receives output data from the Compute Unit, and similarly to the PPU, performs post-processing steps, including quantization, bias addition, and application of the ReLU activation function. The *Store* Unit reads instructions from the *Fetch* Unit to get the destination address within the host-side memory to write back the processed data and finally coordinates this data transfer to the main memory using AXI4M.

4.4. Accelerator software

For each accelerator design, there is a need for the software driver to communicate with the accelerator and ensure the correct instructions and data are passed to the accelerator. Additionally, careful co-design of the accelerator driver and accelerator hardware is required to optimize the performance effectively. In this section, we give a brief overview of the software driver used in our three designs.

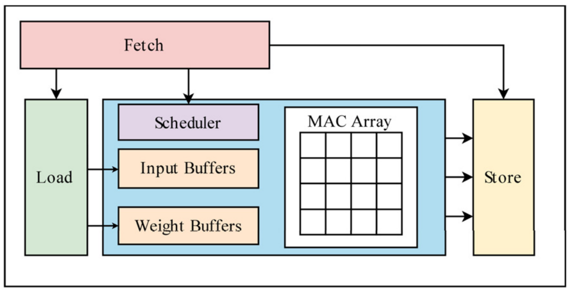

Fig. 7. FC-GEMM accelerator design, featuring a single Compute Unit.

4.4.1. Convolutional layers

The VM and SA accelerators both target the same operation type, namely convolutional layers. Hence, the two designs share very similar driver software. First, the accelerator driver applies the *im2col* operation to the input tensor, which is a required step for our definition of GEMM convolution. Next, the accelerator driver maps the data into the format required by our accelerator designs, coordinates transfer to the accelerator, and receives the processed output data. Note that the key difference between the drivers for the VM design and the SA design is the handling of output data, as the output layouts differ.

We pipelined the execution of the operations within the GEMM driver across multiple batches of GEMM operations within each layer to ensure that the CPU is not idle while the accelerator is processing inputs. In later design iterations, we found that the bottleneck was no longer the accelerator's GEMM operations, but instead, it was the quantization and other post-processing steps, which were originally performed on the CPU. Hence we moved these post-processing steps to the accelerator with the creation of the PPU (see Section 4.3.3), with the driver managing the accelerator's new functionality.

4.4.2. Fully connected layers

The driver for the FC-GEMM accelerator has a number of key differences since it targets a different operation type than the VM and SA designs. However, ultimately it performs the same function, namely processing data passed between TFLite and the accelerator and vice-versa. In addition, since the accelerator reads and writes host memory directly, the driver needs to identify and pass the relevant memory addresses to the accelerator.

Overall, the key responsibilities of the FC-GEMM accelerator's driver are: *i*) Computing quantization parameters and managing the bias and activation function metadata, which will be passed to the accelerator; *ii*) Padding and copying TFLite inputs tensors to memory mapped input buffers; *iii*) Generating the instructions required for the accelerator to compute the given FC layer; *iv*) Copying back computed results from the memory mapped output buffer to the TFLite outputs tensors; *v*) Managing all the accelerator control signals, such as initializing the accelerator and waiting for the accelerator to finish the computation.

5. Evaluation

5.1. Experimental setup

We evaluate the three accelerator designs described in our case study (see section 4.2) on the PYNQ-Z1 board, which includes an edge FPGA and a 650 MHz dual-core ARM Cortex-A9 CPU. We utilize the TFLite Model Benchmarking tool[2] to run all experiments.

[2] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark.

Table 2
Details on the DNN models used throughout our evaluation. For CNN models, CONV layers are shown and for BERT models, FC layers are shown.

Model	Layers	MACs (M)	Parameters (M)	Model Size (MB)
MobileNetV1	14	569	4.2	11.4
MobileNetV2	35	300	3.8	12.8
InceptionV1	57	1502	6.6	20.5
InceptionV3	94	5725	23.9	75.2
ResNet18	20	1800	11.5	30.5
ResNet50	54	3,800	25.6	61.5
EfficientNet-L	61	2,550	13.0	42.6
MobileBert	362	2,850	25.3	345.4
Albert	554	2,609	11.0	178.3

First, we benchmark seven widely used CNN models quantized to signed 8-bit integers: MobileNetV1 [17], MobileNetV2 [36], InceptionV1 [40], InceptionV3 [41], ResNet18, ResNet50 [16], and EfficientNet-Lite [42], all trained for the ImageNet dataset [35]. For each CNN model, we evaluate CPU-only inference times in TFLite using one and two CPU threads, taking the median of 100 runs to compare against our VM and SA designs implemented through the TFLite delegate system. We then benchmark two popular BERT models quantized to signed 8 bits integers: MobileBert [38] and Albert [24], both defined for the SQAUD dataset [34] with sequence lengths of 384 and 128 for the respective models. For the CNN models, in addition to comparisons against our CPU-only baseline, we compare our performance against a state-of-the-art VTA accelerator on the same device, as discussed later in Section 4.4. For each BERT model, we again evaluate CPU-only inference times in TFLite using one and two CPU threads, taking the median of 100 runs to compare against our FC-GEMM accelerator. Table 2 contains some key details of all the DNN models used throughout our experiments.

Note that non-accelerated TFLite layers for both sets of benchmarks use their native C++ implementations and are compiled with the recommended TFLite optimizations for the target platform. This included enabling NEON-based vector instructions, multithreading, and utilizing the gemmlowp [12] backend library. Conversely, our accelerated layers use custom-designed CPU-side accelerator drivers (see Section 4.4) that include handwritten software optimizations (e.g., vector instructions) developed using the SECDA-TFLite toolkit. We gather energy metrics using a COOWOO digital USB power meter [7].

5.2. CNN results

Table 3 shows the breakdown of inference time and energy consumption for the CNN models under study for a single image using the CPU with one (CPU-1t) and two (CPU-2t) threads and the two accelerator designs (VM and SA) with a stddev of 0.5%. The time is split between convolutional layers (T_{CONV}), which our accelerators target, and all other layers ($T_{Non\text{-}CONV}$), which run on the CPU.

5.2.1. Performance across models

For the VM accelerator, we observe an average speedup across models of 2.9× and 1.8× and an average energy saving of 2.6× and 1.7× for one and two threads, respectively, in each case when compared to CPU-only inference. Similarly, for the SA accelerator, we observe an average speedup across models of 3.4× and 2.0× and an average energy saving of 2.9× and 1.9× for one and two threads, respectively, in each case when compared to CPU-only inference.

We also observe that InceptionV1 achieves the best speedup relative to the CPU-only version, with 3.8× and 2.1× speedup for one and two threads, respectively for VM, and 4.3× and 2.4× respectively for SA. Compared to MobileNetV1 and MobileNetV2, which feature depthwise separable convolutions (meaning that

Table 3
Inference time (milliseconds, speedup) and energy consumption (joules, savings) for the seven CNN models under study when using one or two CPU threads and the accelerator designs.

DNN	Hardware	T_{CONV}	$T_{Non\text{-}CONV}$	Total time		Energy	
MobileNetV1	CPU-1t	566	165	732	1.0x	2.05	1x
	CPU-1t + VM	107	164	271	**2.7x**	0.79	**2.6x**
	CPU-1t + SA	107	164	271	**2.7x**	0.83	2.5x
	CPU-2t	288	103	391	1.0x	1.26	1.0x
	CPU-2t + VM	107	102	209	1.9x	**0.65**	1.9x
	CPU-2t + SA	107	101	**208**	1.9x	**0.65**	1.9x
MobileNetV2	CPU-1t	430	204	634	1.0x	1.84	1.0x
	CPU-1t + VM	124	203	327	**1.9x**	0.94	**2x**
	CPU-1t + SA	126	199	325	**1.9x**	0.94	**2x**
	CPU-2t	219	128	347	1.0x	1.08	1.0x
	CPU-2t + VM	124	128	**252**	1.4x	**0.76**	1.4x
	CPU-2t + SA	126	127	253	1.4x	**0.76**	1.4x
InceptionV1	CPU-1t	1300	70	1370	1.0x	3.64	1.0x
	CPU-1t + VM	291	67	358	3.8x	1.08	3.4x
	CPU-1t + SA	254	66	**320**	**4.3x**	**1.01**	**3.6x**
	CPU-2t	687	72	759	1.0x	2.05	1.0x
	CPU-2t + VM	291	67	358	2.1x	1.08	1.9x
	CPU-2t + SA	255	66	321	2.4x	**1.01**	2.0x
InceptionV3	CPU-1t	5182	312	5494	1.0x	15.44	1.0x
	CPU-1t + VM	1213	283	1496	3.7x	4.72	3.3x
	CPU-1t + SA	1001	279	**1280**	**4.3x**	4.32	**3.6x**
	CPU-2t	2673	310	2983	1x.0	9.04	1.0x
	CPU-2t + VM	1215	281	1496	2.0x	4.75	1.9x
	CPU-2t + SA	1003	277	**1280**	2.3x	**4.25**	2.1x
ResNet18	CPU-1t	1680	53	1733	1x	3.67	1.0x
	CPU-1t + VM	558	48	606	2.9x	1.80	2.0x
	CPU-1t + SA	356	46	**402**	**4.3x**	**1.30**	**2.8x**
	CPU-1t + VTA	-	-	1369	1.3x	3.28	1.1x
	CPU-2t	876	53	929	1.0x	2.45	1.0x
	CPU-2t + VM	558	49	607	1.5x	1.76	1.4x
	CPU-2t + SA	356	46	**402**	2.3x	**1.30**	1.9x
	CPU-2t + VTA	-	-	737	1.3x	1.51	1.6x
ResNet50	CPU-1t	3200	475	3675	1.0x	10.08	1.0x
	CPU-1t + VM	798	328	1126	3.3x	3.35	3.0x
	CPU-1t + SA	588	324	912	**4.1x**	2.84	**3.5x**
	CPU-1t + VTA	-	-	1759	2.1x	4.39	2.3x
	CPU-2t	1648	410	2058	1.0x	5.76	1.0x
	CPU-2t + VM	799	328	1127	1.8x	3.31	1.7x
	CPU-2t + SA	586	324	**910**	2.3x	2.84	2.0x
	CPU-2t + VTA	-	-	1036	2.0x	**2.81**	2.1x
EfficentNet-L	CPU-1t	2665	1230	3895	1.0x	10.98	1.0x
	CPU-1t + VM	566	1241	1807	**2.2x**	5.33	**2.1x**
	CPU-1t + SA	555	1225	1780	**2.2x**	5.44	2.0x
	CPU-2t	1335	667	2002	1.0x	6.12	1.0x
	CPU-2t + VM	567	650	1217	1.6x	**3.78**	1.6x
	CPU-2t + SA	555	651	**1206**	1.7x	3.92	1.6x

each convolutional layer performs fewer MACs per input), InceptionV1's standard convolutions have greater potential for GEMM acceleration since the relative cost of its data preparation stage is smaller. Additionally, for InceptionV1, InceptionV3, ResNet18, and ResNet50, we observed negligible speedups for multi-threaded execution relative to the other models due to the larger number of GEMM operations, coupled with our pipelined execution. This means that the CPU-side latency, due to data format conversions, is "hidden" by the accelerator's computation, and thus results in minimal benefits from two threads.

5.2.2. Performance breakdown

We observe less speedup and energy savings with dual-thread execution as expected, since the compute capacity of the CPU doubles while both accelerator designs remain the same. However, our accelerated runtime using two threads improves inference time since the CPU-side *Accelerator Driver* can leverage threads. While analyzing our designs, we observed that we hit a threshold for performance gains achieved by our hardware designs, with the

Table 4
Inference time (seconds, speedup) and energy consumption (joules, savings) for the two BERT models under study when using one or two CPU threads and the FC-GEMM accelerator.

DNN	Hardware	T_{FC}	$T_{Non\text{-}FC}$	Total time		Energy	
MobileBert	CPU-1t	8.62	3.42	12.04	1.0x	3.02	1.0x
	CPU-1t + FC-G	2.74	3.41	6.15	**2.0x**	**1.58**	**1.9x**
	CPU-2t	4.44	3.47	7.91	1.0x	2.09	1.0x
	CPU-2t + FC-G	2.64	3.39	**6.04**	1.3x	1.62	1.3x
Albert	CPU-1t	9.88	2.59	12.46	1.0x	3.13	1.0x
	CPU-1t + FC-G	1.42	2.57	4.00	**3.1x**	**1.08**	**2.9x**
	CPU-2t	4.47	3.07	7.54	1.0x	2.02	1.0x
	CPU-2t + FC-G	1.40	2.57	**3.96**	1.9x	**1.08**	1.9x

bottleneck for inference performance shifting to two other areas. Namely, (i) CPU-side CONV data preparation and result unpacking; (ii) and non-accelerated layers.

For (i), breaking down the single-threaded CONV time for VM, we observe that only 37% of the time is spent performing off-chip data transfers and the accelerator computations. The CPU-side data preparation and resulting unpacking represents the majority of the CONV time, 63%, which highlights the importance of hardware/-software co-design to ensure that additional hardware changes cannot further reduce this time.

For (ii), in single thread CPU-only inference, Non-CONV layers only represent 16% of the inference time on average. However, by accelerating the CONV layers, the relative importance of Non-CONV layers increases, representing 38% and 40% of single thread inference time for VM and SA accelerators, respectively.

5.2.3. SA vs. VM

Comparing our two designs, SA achieves slightly better performance, 16% on average in latency and up to 9% in energy savings. This performance difference between SA and VM can be attributed to the different strategies used to perform GEMM and due to the different configurations of the MAC PEs. The SA design contains a single large array of PEs, whereas the VM design consists of four smaller GEMM units, introducing higher overhead for scheduling GEMM operations. We can also conclude that while VM's smaller GEMM units allow for smaller tiles sizes, the SA design's single large systolic array leads to higher data reuse, lowering the number of BRAM reads. While there is a slight variation in performance due to the differences between these designs, the overall end-to-end performance is similar due to the shift in the inference performance bottlenecks to the CPU-side.

Finally, note that both accelerator designs could still be further refined. However, the purpose of the case study is to highlight that by using SECDA-TFLite, we were able to quickly port, redesign, and further iterate upon the previously defined VM and SA accelerators while improving inference time performance and energy consumption against the CPU-only case.

5.3. BERT results

Table 4 shows the breakdown of inference time and energy consumption for the two BERT models under study using the CPU with one (CPU-1t) and two (CPU-2t) threads and the FC-GEMM (FC-G) accelerator with a stddev 0.1%. Time is split between Fully Connected layers (T_{FC}), which our accelerator targets, and all other layers ($T_{Non\text{-}FC}$), which run on the CPU.

For our FC-GEMM accelerator, we observe an average speedup across models of 2.5× and 1.6× and an average energy saving of 2.4× and 1.6× for one and two threads, respectively, in each case when compared to CPU-only inference.

Similar to the CNN experiments, we observe less speedup and energy savings with dual-thread execution as expected, since the compute capacity of the CPU doubles while the accelerator designs remain the same.

Comparing the performance of the two models, we find that the accelerator provides a 3.1× speedup for Albert while only providing a 1.9× speedup for MobileBert for single thread execution. We perform further analysis of the single thread inference of the two models to identify the reason for the difference in performance improvement between the two models while using the accelerator.

We observe that the time spent during inference on FC layers goes from 79% to 36% when comparing CPU-only inference against accelerated inference for Albert. Similarly, FC layers inference time percentage goes from 72% to 46% for MobileBert. These percentage shifts highlight that when accelerated, the remaining layers of the models become the majority of the inference time, especially for Albert.

We investigate further and break down how the FC layers are delegated during inference utilizing the accelerator. Note that the FC delegate used for the FC-GEMM accelerator is capable of grouping consecutive FC layers within a given model before inference. We observe that by using the accelerator delegate, we reduce the number of FC layers by 5× and 2.5× for Albert and MobileBert, respectively. Hence this grouping optimization is the key reason behind the difference in the performance boost achieved by utilizing the accelerator for Albert and MobileBert.

5.4. Comparison with state-of-the-art DNN accelerators

We now validate that our designs are competitive with another state-of-the-art DNN accelerator in terms of inference time, our main design goal. We compare our designs against VTA, which is supported through the state-of-the-art DNN compiler framework TVM [4]. We chose it over other accelerator frameworks due to its recent release, support from an active open source community, and its use of 8-bit quantization similar to our designs.

Table 3 shows the performance of VTA for both ResNet18 and ResNet50, taking the median of 100 runs on the PNYQ-Z1 board. These two models were the only two publicly available models which were compatible with both TVM-VTA and TFLite. To be able to compare results, we performed our VTA experiments on the PYNQ Z1. We mapped the open source VTA accelerator to PYNQ's FPGA and performed DNN inference. We carefully followed the instructions provided by the TVM-VTA team,[3] which includes a pre-compiled FPGA mapping for VTA for the PYNQ Z1.

The results show that the designs developed using SECDA with the aid of the SECDA-TFLite toolkit are competitive with VTA. In single thread comparison with the VTA accelerator, the VM and SA accelerators are 1.61x and 2.1x faster on average across the models, respectively, and similarly, the VM and SA accelerators report 39% and 12% less energy consumption on average, respectively. In dual thread comparison, our VM design is on par for ResNet18 and only 16% worse for ResNet50 in terms of latency, while VTA reports 43% and 23% less energy consumption for ResNet18 and ResNet50, respectively. Our SA design outperforms VTA by 35% and 6% in terms of latency for ResNet18 and ResNet50, respectively, but VTA reports 14% and 5% less energy consumption for ResNet18 and ResNet50, respectively.

Note that VTA runs more of its layers on the accelerator, for example, the intermediate pooling layers, which results in fewer off-chip data transfers and therefore achieves a greater energy efficiency than our design. In terms of our target performance metric, inference time, we have demonstrated our accelerator designs developed utilizing SECDA-TFLite can be competitive with a state-of-the-art accelerator.

[3] https://tvm.apache.org/docs/topic/vta/tutorials/frontend/deploy_classification.html#sphx-glr-topic-vta-tutorials-frontend-deploy-classification-py.

For BERT acceleration, we note that to the best of our knowledge, there are no publicly available FPGA-based DNN accelerator inference designs targeting transformer models available that we can compare against. Hence, we only compare against CPU-only inference.

6. Related work

While there is a range of design tools for designing DNN accelerators, there are fewer tools available to enable the developer to easily integrate their design workflow and design with a pre-existing DNN framework. STONNE [29] provides cycle-accurate simulation for deep learning accelerator designs such as MAERI [22] and SIGMA [33]. However, it does not have a direct path to map candidate designs to hardware and run hardware evaluation on target FPGA devices with the chosen DNN framework. TFLITE-SOC [2] features aspects of the SECDA methodology by tightly integrating TFLite with a SystemC system-level simulation, but it lacks the additional support that the SECDA-TFLite toolkit provides, which enables the easy integration of both simulation and hardware inference with TFLite. SMAUG [49] provides a simulation-based design methodology that uses gem5-Aladdin [37] to perform full system simulation of the host system, the off-chip memory accesses, and the accelerator design itself. While this approach provides high fidelity in terms of design performance insights, the simulation speed is very slow due to the simulation of the entire system (e.g., several hours for ResNet50). Rather than integrating with an existing DNN framework, as SECDA-TFLite does, models must be redefined using SMAUG's Python API. In addition, SMAUG does not offer an approach where a design can be directly synthesized to a target FPGA and integrated with a DNN framework of choice.

While the SECDA methodology and SECDA-TFLite toolkit focus on the development and integration of new DNN hardware accelerator architectures, there are other tools that enable automated hardware space exploration of templated DNN accelerator architecture. This automated hardware space exploration acts as an optional design stage to further improve performance for a given DNN model. Examples include VTA [28], which defines a GEMM unit and high-level task ISA, built on top of the TVM [4] compiler stack, to produce specific designs for a given DNN architecture. VTA can optionally leverage the AutoTVM tuning tool [5] for additional design space exploration. DNNBuilder [50] produces DNN specific architectures and permits exploration of alternative quantization schemes, as adopted by other schemes [13,45,48]. These automated design exploration tools require the definition of the template hardware architectures which are modular and tunable, and with the SECDA methodology we are able to produce the template hardware architecture definitions. Additionally, with SECDA-TFLite, we can easily integrate our template hardware designs within TFLite to measure the end-to-end performance of the design before integrating the design with the automated design exploration tools.

Finally, a general objective for DNN acceleration is to improve the efficiency of MAC operations. Approaches include improving data reuse and different dataflow strategies [39] and leveraging sparsity in weights/activations to reduce MACs, supported by efficient hardware designs [18,26]. The SECDA methodology, along with the SECDA-TFLite toolkit, enables shorter development and integration time with TFLite and more efficient exploration of hardware solutions for these opportunities.

7. Conclusion

In this paper, we presented SECDA-TFLite, a new open source toolkit that increases the ease of developing new hardware accelerators for edge DNN inference, following the hardware/software co-design SECDA methodology. SECDA-TFLite provides the initial environment setup within the TFLite DNN framework as well as a set of tools and utilities to aid in the development of accelerator designs, providing the infrastructure to initialize and follow the SECDA design loop within TFLite. As a result, developers are able to co-design new accelerator designs for TFLite, bypassing many of the initial setup costs.

The SECDA-TFLite toolkit enables tight integration of accelerator designs with TFLite while also enabling the developer to follow the SECDA design loop easily within TFLite, thus improving opportunities for co-design of the accelerator delegate and driver. We provide utilities for SystemC interfacing, simulation profiling, data communication, and multi-threading for the driver.

By simulating the accelerator behavior using SystemC, we reduce development costs by minimizing the number of synthesis iterations, allowing fine-grained benchmarking and co-verification of accelerator components. Furthermore, we can directly map our simulated designs onto an FPGA without re-implementation. As a case study, we proposed three GEMM-based accelerator designs for optimizing the inference of seven CNN models and two BERT models on a PYNQ-Z1 board. The accelerated models outperform the CPU baseline in all cases. As future work, we plan to develop SECDA-based toolkits for other popular DNN inference frameworks, and explore the re-usability of hardware designs across different DNN frameworks.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Jude Haris reports financial support was provided by Engineering and Physical Sciences Research Council (grant EP/R513222/1).

Data availability

The code is available through GitHub: `https://github.com/gicLAB/SECDA-TFLite`.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, et al., TensorFlow: a system for large-scale machine learning, in: USENIX OSDI, 2016, pp. 265–283.

[2] N.B. Agostini, S. Dong, E. Karimi, M. Torrents, J. Cano, J.L. Abellán, D. Kaeli, Design space exploration of accelerators and end-to-end DNN evaluation with TFLITE-SOC, in: SBAC-PAD, 2020, pp. 10–19.

[3] M. Alwani, H. Chen, M. Ferdman, P. Milder, Fused-layer CNN accelerators, in: MICRO, 2016, pp. 1–12.

[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, A. Krishnamurthy, TVM: an automated end-to-end optimizing compiler for deep learning, in: OSDI, 2018, pp. 579–594.

[5] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, A. Krishnamurthy, Learning to optimize tensor programs, in: NeurIPS, 2018, pp. 3393–3404.

[6] Y.-H. Chen, T.-J. Yang, J. Emer, V. Sze, Eyeriss v2: a flexible accelerator for emerging deep neural networks on mobile devices, in: IEEE JETCAS, 2019, pp. 292–308.

[7] COOWOO USB Digital Power Meter, http://www.coowootech.com/tools.html, 2022.

[8] N. Corporation, NVIDIA TensorRT: programmable inference accelerator, https://developer.nvidia.com/tensorrt, Apr. 2016.

[9] B. developers, Bazel, https://bazel.build/, 2021.

[10] O.R. developers, Onnx runtime, https://onnxruntime.ai/, 2021.

[11] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, in: NAACL-HLT, 2019, pp. 4171–4186.

[12] Gemmlowp, https://github.com/google/gemmlowp, 2022.

[13] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, J. Cong FP-DNN, An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates, in: FCCM, 2017, pp. 152–159.

[14] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, H. Kim, Characterizing the deployment of deep neural networks on commercial edge devices, in: IISWC, 2019, pp. 35–48.
[15] J. Haris, P. Gibson, J. Cano, N.B. Agostini, D. Kaeli, SECDA: efficient hardware/-software co-design of FPGA-based DNN accelerators for edge inference, in: SBAC-PAD, 2021, pp. 33–43.
[16] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: CVPR, 2016, pp. 770–778.
[17] A.G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, MobileNets: efficient convolutional neural networks for mobile vision applications, arXiv, 2017, pp. 1–9.
[18] S. Huang, C. Pearson, R. Nagi, J. Xiong, D. Chen, W. Hwu, Accelerating sparse deep neural networks on FPGAs, in: HPEC, 2019, pp. 1–7.
[19] IEEE, IEEE Standard for Standard SystemC Language Reference Manual, IEEE Std 1666-2011, 2012.
[20] N.P. Jouppi, C. Young, N. Patil, D. Patterson, et al., In-datacenter performance analysis of a tensor processing unit, in: ISCA, 2017, pp. 1–12.
[21] A. Krizhevsky, I. Sutskever, G. Hinton, ImageNet classification with deep convolutional neural networks, in: NeurIPS, 2012, pp. 1097–1105.
[22] H. Kwon, A. Samajdar, T. Krishna, MAERI: enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects, in: ASPLOS, 2018, pp. 461–475.
[23] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, T. Krishna, Understanding reuse, performance, and hardware cost of DNN dataflow: a data-centric approach, in: MICRO, 2019, pp. 754–768.
[24] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, R. Soricut Albert, A lite bert for self-supervised learning of language representations, in: ICLR, 2020, pp. 1–17.
[25] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, Y. Xu, Throughput-optimized FPGA accelerator for deep convolutional neural networks, in: ACM TRETS, 2017, pp. 1–23.
[26] Y. Lu, L. Gong, C. Xu, F. Sun, Y. Zhang, C. Wang, X. Zhou, A high-performance FPGA accelerator for sparse neural networks: work-in-progress, in: CASES, 2017, pp. 1–2.
[27] S. Markidis, S.W.D. Chien, E. Laure, I.B. Peng, J.S. Vetter, NVIDIA tensor core programmability, performance & precision, in: IPDPSW, 2018, pp. 522–531.
[28] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, A. Krishnamurthy, A hardware-software blueprint for flexible deep learning specialization, arXiv, 2019, pp. 1–7.
[29] F. Muñoz-Martínez, J.L. Abellán, M.E. Acacio, T. Krishna, STONNE: enabling cycle-level microarchitectural simulation for DNN inference accelerators, in: IISWC, 2021, pp. 122–125.
[30] G. Ottavi, A. Garofalo, G. Tagliavini, F. Conti, L. Benini, D. Rossi, A mixed-precision RISC-V processor for extreme-edge DNN inference, in: ISVLSI, 2020, pp. 512–517.
[31] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in PyTorch, in: NeurIPS Autodiff Workshop, 2017, pp. 1–4.
[32] PYNQ Z1 Manual, https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual, 2020.
[33] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, T. Krishna, SIGMA: a sparse and irregular GEMM accelerator with flexible interconnects for DNN training, in: HPCA, 2020, pp. 58–70.
[34] P. Rajpurkar, J. Zhang, K. Lopyrev, P. Liang, Squad: 100,000+ questions for machine comprehension of text, in: EMNLP, 2016, pp. 2383–2392.
[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, L. Fei-Fei, ImageNet large scale visual recognition challenge, in: IJCV, 2015, pp. 211–252.
[36] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L. Chen, MobileNetV2: inverted residuals and linear bottlenecks, in: CVPR, 2018, pp. 4510–4520.
[37] Y.S. Shao, S.L. Xi, V. Srinivasan, G. Wei, D. Brooks, Co-designing accelerators and SoC interfaces using gem5-Aladdin, in: MICRO, 2016, pp. 1–12.
[38] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, D. Zhou, Mobilebert: a compact task-agnostic bert for resource-limited devices, in: ACL, 2020, pp. 1–13.
[39] V. Sze, Y. Chen, T. Yang, J.S. Emer, Efficient processing of deep neural networks: a tutorial and survey, in: Proceedings of the IEEE, 2017, pp. 2295–72329.
[40] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: CVPR, 2015, pp. 1–9.
[41] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, Rethinking the inception architecture for computer vision, in: CVPR, 2016, pp. 2818–2826.
[42] M. Tan, Q. Le, EfficientNet: rethinking model scaling for convolutional neural networks, in: ICML, 2019, pp. 6105–6114.
[43] Tensorflow lite delegates, https://www.tensorflow.org/lite/performance/delegates, 2022.
[44] J. Turner, J. Cano, V. Radu, E.J. Crowley, M. O'Boyle, A. Storkey, Characterising across-stack optimisations for deep convolutional neural networks, in: IISWC, 2018, pp. 101–110.
[45] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers, FINN: a framework for fast, scalable binarized neural network inference, in: FPGA, 2017, pp. 65–74.
[46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in: NeurIPS, Curran Associates Inc., 2017, pp. 6000–6010.
[47] Y.E. Wang, C.-J. Wu, X. Wang, K. Hazelwood, D. Brooks, Exploiting parallelism opportunities with deep learning frameworks, in: ACM TACO, 2021, pp. 1–23.
[48] X. Wei, C.H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, J. Cong, Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs, in: DAC, 2017, pp. 1–6.
[49] S.L. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, D. Brooks, Smaug: end-to-end full-stack simulation infrastructure for deep learning workloads, in: ACM TACO, 2020, pp. 1–26.
[50] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, D. Chen, DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs, in: ICCAD, 2018, pp. 1–8.
[51] Y. Zhang, M. Pezeshki, P. Brakel, S. Zhang, C. Laurent, Y. Bengio, A.C. Courville, Towards end-to-end speech recognition with deep convolutional neural networks, in: INTERSPEECH, 2016, pp. 410–414.
[52] A. Zhou, A. Yao, Y. Guo, L. Xu, Y. Chen, Incremental network quantization: towards lossless CNNs with low-precision weights, in: ICLR, 2017, pp. 1–14.

Jude Haris is a Ph.D. candidate in the School of Computing Science at University of Glasgow, Scotland. He completed is Master's Degree in Computer Science at the University of Glasgow in 2020. His research interests include Computer Architecture and Hardware Acceleration of Deep Neural Networks. His current works have been focused around developing FPGA-based hardware accelerators for DNN inference on Edge devices.

Perry Gibson is a machine learning systems researcher, with a focus on domain-specific compilers for deep neural networks (DNNs). Since 2019 he has been undertaking a PhD at the University of Glasgow. In 2019, he completed his Masters of Informatics at the University of Edinburgh. His primary research interests include auto-scheduling, model compression, TVM and MLIR, homomorphic encryption, and generative models.

José Cano is an Associate Professor in the School of Computing Science at the University of Glasgow (Scotland, UK), where he leads the Intelligent Computing Laboratory within the GLASS research Section. His research interests are in the broad areas of Computer Architecture, Computer Systems, Compilers, Machine Learning and Security. José received the MS and PhD degrees in Computer Science from Universitat Politècnica de València (UPV), Spain, in 2004 and 2012, respectively. He was a postdoctoral researcher at Universitat Politècnica de Catalunya (Spain) between 2012 and 2013, and at the University of Edinburgh between 2014 and 2018. He has authored over 40 refereed publications, and has served as a co-organizer, as a chair (publicity and session), and as a TPC in numerous conferences and workshops. He was a member of the IEEE TPDS Review Board between 2020 and 2021, and is currently member of the ACM TACO distinguished reviewer board. He is a senior member of IEEE and ACM, and member of HiPEAC.

Nicolas Bohm Agostini is a Ph.D. candidate in the Department of Electrical and Computer Engineering at Northeastern University. In 2015, he completed his Bachelors's degree in Electrical Engineering at the Universidade Federal do Rio Grande do Sul (UFRGS, Brazil). His primary research interests include Computer Architecture and High-Performance Computing. He enjoys teaching and mentoring, which he exemplified by instructing courses such as Compilers, GPU Programming, and Embedded Robotics during his graduate career. Recently, he has been working on projects focused on accelerating machine learning and linear algebra applications by proposing compiler extensions for different targets and new computer architecture features.

David Kaeli received a BS and PhD in Electrical Engineering from Rutgers University, and an MS in Computer Engineering from Syracuse University. He is a Full Processor on the ECE faculty at Northeastern University, Boston, MA where he directs the Northeastern University Computer Architecture Research Laboratory (NUCAR). Prior to joining Northeastern in 1993, Kaeli spent 12 years at IBM, the last 7 at T.J. Watson Research Center, Yorktown Heights, NY.

Dr. Kaeli has published over 400 critically reviewed publications, including 7 books, and 12 patents. His research spans a range of areas including high performance computer architecture, hardware security and reliability and data management/analytics. He is a member of the NIEHS-supported P42 PROTECT Center, the NSF CHEST IUCRC, and Northeastern's Institute for Experiential Artificial Intelligence. He is the Editor-in-Chief of ACM Transactions on Architecture and Code Optimization and an Associate Editor of Elsevier Journal on Future Generation Computer Systems. Dr. Kaeli is a fellow of both the ACM and IEEE. He can be contacted at kaeli@ece.neu.edu.