WOLT: Transparent Deployment of ML Workloads on Lightweight Many-Accelerator Architectures

Kuan-Lin Chiu[1], Guy Eichler[1], Chuan-Tung Lin[2], Giuseppe Di Guglielmo[1], Luca P. Carloni[1]
Department of Computer Science[1], Department of Electrical Engineering[2]
Columbia University in the City of New York., New York, NY
{chiu@cs, guyeichler@cs, cl4030, giuseppe@cs, luca@cs}.columbia.edu

2024 IEEE 42nd International Conference on Computer Design (ICCD) | 979-8-3503-8040-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICCD63220.2024.00102

***Abstract*—Most available frameworks to develop machine learning applications target software deployment on general-purpose processors or GPUs. We propose WOLT, an end-to-end efficient solution to run TFLite application workloads on lightweight system-on-chip (SoC) architectures that feature many fixed-function hardware accelerators. WOLT enables the execution of TFLite operations on pre-designed accelerators without requiring any modification of the application source code. As it establishes an interface between the high-level framework and the device drivers of the accelerators, WOLT includes a resource manager that allows multi-tenant and conflict-free hardware acceleration of many TFLite applications running in parallel on the SoC. We evaluated WOLT with a comprehensive set of FPGA-based experiments by profiling and running 13 different TFLite workloads on a variety of complete SoC prototypes. We designed these prototypes by combining many CVA6 RISC-V processors with multiple accelerators for vector-matrix multiplication and two-dimensional convolution. For these workloads, WOLT delivers up to 23.2× performance speedup and up to 14.6× energy-efficiency gains compared to a purely software execution. When running multiple TFLite workloads in parallel, WOLT achieves up to 4× of additional performance gain compared to basic hardware acceleration, thanks to efficient resource management.**

***Index Terms*—TensorFlow, TFLite, delegate, accelerator.**

I. INTRODUCTION

In the application domain of machine learning (ML) [1]–[4], TensorFlow, the popular development framework for deep learning [5], has been extended with TensorFlow Lite (TFLite), a specialized framework designed for running "ML at the edge" on resource-constrained devices. Originally, TFLite focused on enabling the execution of ML inference with pre-trained models compressed to their lightweight versions, thereby trading off numeric precision for energy saving. The concept of *TFLite delegates* was introduced to utilize hardware components other than central processing unit (CPU) with TFLite applications [6]. TFLite delegates are software mechanisms that allow the offloading of TFLite operators (e.g., matrix-matrix multiplications and convolutions) from CPUs to commercial graphic processing units (GPUs).

Domain-specific system-on-chip (SoC) architectures integrate a growing number of *specialized fixed-function hardware accelerators* (or, simply, *accelerators*) next to CPUs and GPUs [7]–[12]. Optimized for a key computational kernel, an accelerator computes it with better performance and energy efficiency than software. Benefiting from accelerators, however, typically requires a time-consuming effort to understand the underlying SoC architecture and to modify existing software applications so that they can invoke them via device drivers.

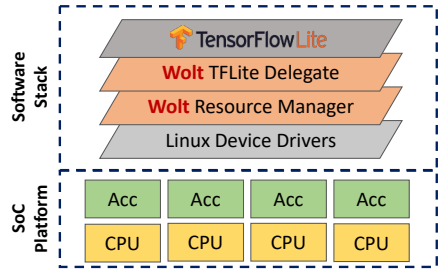


Fig. 1: The WOLT software stack.

WOLT extends the benefits of TFLite delegates to heterogeneous many-accelerator SoCs. As shown in Fig. 1, WOLT enables the deployment of TFLite workloads on an SoC platform with many accelerators. WOLT supports *complete transparency*, which implies no modifications to the high-level software applications that run on the SoC. WOLT includes a new TFLite delegate with a software interface that translates the calls of TFLite operators in the application into the configuration and invocation of the accelerators. WOLT supports the multi-tenant execution of applications by implementing a resource manager with a locking mechanism. The resource manager receives requests from software threads to invoke the accelerators and grants them among the available resources. In this way, WOLT eliminates the requirement to bind specific accelerators to the software threads [13] and promotes efficient and balanced utilization of the available accelerators.

We integrate WOLT into the software architecture of ESP [27], an open-source platform for heterogeneous SoC design, and develop our TFLite delegate and resource manager to control the configuration and invocation of the available accelerators. We modify ESP-based accelerators to support additional TFLite operators and more data types for ML models. We evaluate WOLT by designing multiple FPGA-based SoC prototypes with up to eight accelerators and two 64-bit CVA6 RISC-V CPUs [28]. Our FPGA-based experiments demonstrate the benefits of using WOLT to execute 13 different TFLite workloads from popular ML application domains, comparing it to regular software execution on the CVA6 CPU.

Overall, WOLT eliminates the need to customize TFLite applications for multi-tenant execution on many-accelerator SoCs, while providing significant gains in performance and energy efficiency. These are our main contributions:

1) The development of a TFLite delegate to support offloading TFLite operations to specialized hardware ac-

2576-6996/24/$31.00 ©2024 IEEE
DOI 10.1109/ICCD63220.2024.00102

TABLE I: Characteristics of ML workloads with the performance comparison between different CPU architectures [msec].

Model	Size [MB]	FC	Conv2d	Depthwise Conv2d	Others	Intel	Intel*	Arm	Arm*	CVA6	CVA6*
Kws [14], [15]	0.043	0.19%	58.24%	37.54%	4.03%	1.98	1.07	14.62	10.40	1,988.05	1,502.86
ResNet10 [15], [16]	0.311	-	95.57%	-	4.43%	5.95	0.16	7.51	1.31	4,587.50	1,378.65
Vww [15], [17]	0.846	0.09%	49.37%	49.12%	1.42%	12.05	0.14	17.62	4.84	3,568.41	2,280.18
EfficientNet-0 [18]	18	0.08%	40.86%	57.24%	1.82%	107.77	14.53	206.58	64.07	152,419.10	61,289.70
EfficientNet-1 [18]	21	1.02%	39.22%	59.66%	0.10%	111.15	25.23	544.23	221.08	213,053.52	86,642.86
MobileNet_v1 [19]	17	-	68.79%	31.14%	0.07%	220.48	140.78	697.36	236.99	305,528.76	75,113.90
MobileNet_v2 [20]	14	1.88%	40.95%	57.08%	0.09%	82.61	18.31	436.39	157.21	121,034.02	39,369.30
MobileNet_v3 [21]	16	-	44.46%	48.46%	7.08%	44.24	8.91	366.05	104.09	64,020.93	20,588.53
ToyCar [15], [22]	1.1	95.8%	-	-	4.20%	0.20	0.07	0.74	0.17	362.91	298.37
SqueezeNet [23]	4.8	-	89.84%	-	10.16%	46.01	26.33	247.99	95.85	295,929.03	96,007.07
MNIST2 [24]	0.4	91.11%	-	-	8.89%	0.12	0.01	0.14	0.08	72.73	62.55
MoViNet [25]	26	-	54.37%	26.92%	18.71%	26.25	11.84	226.05	91.79	29,655.51	16,164.06
MoveNet [26]	9	-	47.54%	45.44%	7.02%	53.93	18.89	409.24	143.84	103,960.04	29,849.13

*Performance when running with XNNPACK

celerators in many-accelerator SoC architectures.
2) The programming of a software layer to enhance resource-management transparency for parallel execution of TFLite workloads.
3) The design of 2D-convolution and general-matrix-multiplication accelerators to better accommodate multiple data types and TFLite operators.
4) The FPGA-based evaluation of running many TFLite workloads on a variety of heterogeneous SoCs with different compositions of accelerators and CPU cores.

II. Background and Workloads Analysis

In ML-based computation, a tensor is a data structure used to represent and process multi-dimensional arrays of numerical data. Tensors are crucial for various operations in deep learning and neural networks. As shown at the top of Fig. 2, an ML algorithm in a user application can be represented as an ML graph consisting of tensors and tensor operations. "Tensor operation" refers to the mathematical process or algorithmic procedure employed, while "tensor operator" refers to the practical realization of the algorithm within a particular ML framework. Fig. 2 includes examples of tensor operation nodes such as 2D convolution (`Conv2d`), element-wise addition (`Add`), and fully connected operations (`FullyConnected`), with input and output tensors labeled as nodes (a to f). The bottom section of Fig. 2 illustrates available processing units. By default, all operations are executed on the CPU, which may not be the most efficient approach. Tensor operations can also be allocated to third-party software libraries for execution on conventional CPUs or to dedicated processing units like GPUs and tensor processing units (TPUs).

Between the user application and the processing units lies a critical component: the TFLite core library. This library offers basic software implementations of tensor operations for CPUs. To optimize performance on other processing units, the core library provides *TFLite delegates*. These delegates enable the execution of specific operations with specialized implementations tailored to the resources available on GPUs and TPUs. This mechanism inherently allows combined implementations of operations, such as delegating `Conv2d` and `Add` to a GPU, while `FullyConnected` continues to be processed on the CPU. Additionally, operations can be delegated to optimized third-party software libraries like XNNPACK [29].

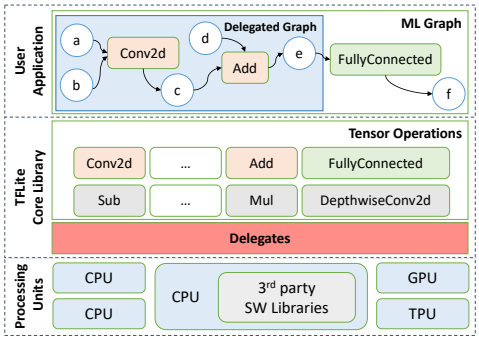


Fig. 2: The concept of TFLite delegate.

The TFLite delegation mechanisms maximizes the utilization of hardware resources for improved inference performance and energy efficiency. However, the utilization of TFLite delegates for specialized fixed-function hardware accelerators presents a set of technical challenges that have limited their effective use. Specifically, the interplay between operations executed by the accelerators and operations executed by the CPU complicates the performance optimization of the TFLite workloads running on the SoC.

To evaluate the impact of delegating different operators in TFLite applications, we profiled the execution of many different ML models across different application domains: keyword spotting [14], image classification [16], [18], [19], [23], binary image classification [17], anomaly detection [22], video recognition [25], and human pose estimation [26].

We analyzed a total of 13 ML models that are executed on eight different datasets, four of which are from the MLPerf Tiny benchmark suite for embedded devices [15]. TABLE I reports the models with their main characteristics. The model sizes range from quite small (43KB) to fairly large sizes (26MB) for lightweight SoC architectures, which are the target of Wolt. We profiled the models according to the utilization of TFLite computational operators. We ran all the workloads on an Intel-i7 CPU and calculated the summed runtime that each workload spent for executing each type of operator. TABLE I reports the runtime of each workload for every operator as a percentage relative to the total runtime of the workload. Notably, `Conv2d`, `FullyConnected` and `DepthwiseConv2d` account for most of the runtime across all workloads. Indeed,

we can improve the total runtime of TFLite workloads by focusing on the acceleration of these three operators. For example, if the average speedup achievable by an accelerator for `Conv2d` is $10\times$, Amdhal's Law [30] indicates that the prospective maximum speedup when executing *SqueezeNet* can reach $5.2\times$. Similarly, given an average speedup of $10\times$ that an accelerator can offer for `FullyConnected`, the potential maximum speedup when running *ToyCar* can reach $7.3\times$.

III. WOLT

Transparency means preserving the existing external interface while modifying internal behavior to shield other systems or users from the impact of the changes. In order to achieve real system transparency when deploying TFLite workloads on many-accelerator SoC architectures, we developed WOLT as a vertically-integrated approach that combines the following software and hardware elements:

1) A dedicated TFLite delegate for offloading computation into specialized fixed-function hardware accelerators.
2) A software mechanism to manage the accelerator assignments for multiple TFLite applications.
3) Accelerators optimized for specific tensor operations.
4) A heterogeneous SoC platform that simplifies the integration and invocation of TFLite-based ML accelerators.

As illustrated in Fig. 3, WOLT serves as the connection between software ML applications, TFLite, device drivers responsible for invoking and configuring accelerators, and a resource manager that orchestrates the accelerator assignment on many-accelerator SoC architectures.

WOLT TFLite Delegate. As part of WOLT, we propose a TFLite delegate that links ML software applications and SoC accelerators. By doing this, WOLT improves the speed and efficiency of running TFLite models at the edge or in other resource-constrained environments. Fig. 3 shows the main components of our proposed TFLite delegate. The TFLite framework incorporates basic interfaces that enable the execution of TFLite models on specialized processing units. The *TfLiteDelegate* serves as the foundational interface, facilitating the delegation of model operations to these processing units. We developed the WOLT TFLite delegate by implementing two functions: *WoltDelegate* and *WoltDelegateKernel*.

WoltDelegate is responsible for constructing and destructing delegated graph, as shown at the top of Fig. 3. We implemented the constructor and the destructor for the WOLT delegate: `Create()` and `Delete()`. In addition, we implemented the `SupportedNode()` function to indicate which operators can be delegated to the corresponding accelerators.

In the TFLite framework, there are functions for initialization, preparation, and running of the delegated graph. To design our WOLT delegate, we followed the general software structure as in the TFLite framework. We built *WoltDelegateKernel* to execute the delegation process that is specific to the accelerators. We implemented the following three functions within the *WoltDelegateKernel*: The `Init()` function is called only once and performs one-time initialization procedures, to initialize the subgraph that can be delegated from within the complete application graph. It stores the indices of all the subgraph nodes that are part of the delegate kernel, and

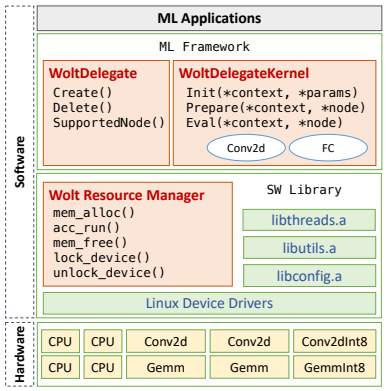


Fig. 3: The architecture of WOLT.

initializes the memories needed for storing the inputs and outputs of each node. It also initializes a hardware buffer for loading and storing the data for the accelerators. The `Prepare()` function handles the preparation of the delegated subgraph. It sets the parameters of each operation node, e.g. the feature map dimensions and filter dimensions for `Conv2d`, and it configures the accelerators for the specific operations. The `Eval()` function executes the delegated subgraph by invoking the relevant accelerators for the required operations.

The information of the delegated graph is stored in three data structures: The *context* data structure refers to the environment of a specific operation in the computational graph. It stores information related to the execution state during the inference process, such as whether the node has been invoked, input and output tensors, and workspace memory. The *params* data structure holds the configuration and settings of a particular operation within the graph. For instance, `Conv2d` contains the parameters like kernel size, stride, padding, activation function, and so on. It also contains the information on the number of nodes to be delegated. The *node* data structure is used to represent the individual operator within the graph. It is the fundamental building blocks of the computational graph. It contains reference to the operation type, for example, the code for `FullyConnected` and `Conv2d`.

Given the workload-profiling results presented in TABLE I, we focused on the `Conv2D` and `FullyConnected` operators in our WOLT delegate as they are the most time consuming operations among the different workloads.

WOLT Resource Manager and Software Interface. Linux device driver and software APIs are used to invoke accelerators [27], [31]. They do it by means of three main primitives: `mem_alloc()`, `acc_run()`, and `mem_free()`. `mem_alloc()` sets up a memory buffer for the accelerator while `mem_free()` releases this buffer. `acc_run()` invokes the accelerator through its device driver.

The device drivers and associated software methods are written primarily in C, while the TFLite source code is in C++. To enable C linkage for the C++ compiler, we used a two-step approach [32]. First, we encapsulated the necessary functions into static C libraries (*libthreads.a*, *libutils.a*, and *libconfig.a*), as shown in Fig. 3. Then, we provided a corresponding header file in TFLite, with APIs wrapped with the *extern "C"* keyword to ensure C linkage. This prevents name mangling and allows

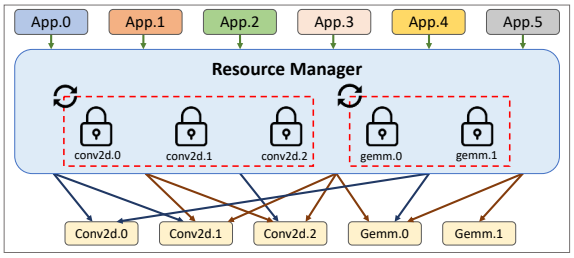


Fig. 4: The architecture of WOLT resource manager.

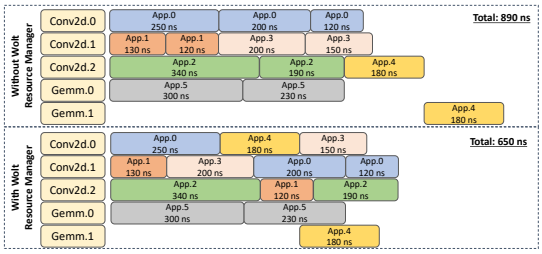


Fig. 5: An example of using WOLT resource manager.

direct invocation of software APIs within the WOLT TFLite delegate, thus enabling seamless interfacing with WOLT.

In typical scenarios, when an application invokes an accelerator, it is necessary to specify the type of the accelerator and the specific accelerator instance. For example, if the SoC contains two acc_Conv2d accelerators, the user must specify which accelerator to invoke, i.e. acc_Conv2d.0 or acc_Conv2d.1. This limits the system transparency from the application's viewpoint because the software must be aware of the hardware in the SoC and indicate the specific accelerator to be used. Furthermore, for the case of parallel execution of applications, this is problematic because each application will have a statically allocated accelerator at design time, which can damage performance and cause runtime bottlenecks.

To address this limitation, we developed a software resource manager and integrated it into the Linux device driver that allocates accelerators for runtime applications. Therefore, the applications only need to initiate a request for an accelerator type and pass the computation parameters. When booting Linux, the WOLT resource manager probes the available accelerators on the SoC by using Linux device drivers. Then, it assigns a lock for each accelerator to indicate if the accelerator is in-use or idle. Multiple applications running simultaneously may compete for the available accelerators by requesting a corresponding lock. The WOLT resource manager is intentionally designed to be simple, enabling potential portability to different platforms. It checks the locks in a round-robin fashion in order to find an idle (available) accelerator. This fair round-robin allocation ensures balance in the system by avoiding the overuse of any single accelerator; it also supports generality across SoC architectures with different accelerator combinations. Moreover, the resource manager remains independent from the TFLite operators and can be extended with other allocation algorithms that fit a chosen SoC architecture.

We used a file as a lock for each accelerator instead of interprocess communication by design choice. During the executing of the applications, (1) if an idle accelerator is found, the resource manager invokes the accelerator for the application and marks it as busy by using the lock (`lock_device()`). (2) if all the accelerators are in use, the application is stalled until one of the locks becomes available (`unlock_device()`). With the WOLT resource manager, running multiple applications in parallel on a heterogeneous many-accelerator SoC becomes much more efficient, while the software application remains unaware of the underlying hardware.

Fig. 4 shows an example that includes six applications with three acc_Conv2d and two acc_Gemm accelerators. Each application has different requirements for accelerators. Fig. 5 shows the executions of the six applications with and without the WOLT resource manager. When the WOLT resource manager is not enabled, specific accelerators are manually assigned to the applications. For example, App.0 always uses acc_Conv2d.0, App.3 always uses acc_Conv2d.1. Fig. 5 shows how the WOLT resource manager enables seamless accelerator invocation from different applications, improving overall performance from 960 ns to 650 ns.

WOLT Specialized Hardware Accelerators. The majority of TFLite learning models heavily use the `FullyConnected` and `Conv2D` operators, as shown in TABLE I. Consequently, for WOLT, we designed an accelerator for general matrix-matrix multiplication (acc_Gemm) and an accelerator for 2D-convolution (acc_Conv2d) to be called by the `FullyConnected` and `Conv2D` operators, respectively. We designed the accelerators using high-level synthesis (HLS) and then specialized them with further optimizations using techniques such as loop pipelining, loop unrolling, and so on. The accelerators use 32-bit fixed-point and 32-bit floating-point data types, whereas TFLite models may be quantized with diverse data types, including the popular 8-bit integer data type. To address this versatility in ML model data types, we further developed accelerators optimized to operate with 8-bit integers, namely, (acc_GemmInt8 and acc_Conv2dInt8). Compared to the 32-bit versions, these specialized 8-bit accelerators have smaller area and power consumption, which are ideal for lightweight edge devices. We also added compatible device drivers for the new accelerators.

IV. EXPERIMENTAL EVALUATION

We performed a variety of FPGA-based experiments to analyze the performance and energy-efficiency gains obtained by using WOLT and delegating the execution of TFLite operators to accelerators. Our experiment show also the benefits of the WOLT resource manager when running TFLite workloads.

FPGA-Based Experimental Setup. To implement our SoC prototypes we use the open-source ESP platform [27], as it provides multiple flows for the design of accelerator [33], their agile integration in a flexible SoC architecture, and a robust software ecosystem [34]. The ESP heterogeneous tile-based architecture includes accelerator, processor, I/O and memory tiles [35]. The accelerator tiles follow a loosely-coupled accelerator model [36], [37]. A processor tile has a CPU chosen from the RISC-V 64-bit CVA6 [28], the SPARC 32-bit LEON3 [38], and the RISC-V 32-bit Ibex [39]. Each memory tile has a channel to main memory.

TABLE II: FPGA Resource Utilization.

	BRAMs	DSPs	FFs	LUTs	Power [W]
MEM	540	253	189K	188K	1.234
IO	160	0	14K	12K	0.084
CPU	36	27	42K	54K	0.200
Conv2d	17	70	23K	22K	0.081
Conv2dInt8	15	38	19K	19K	0.057
Gemm	19	44	19K	25K	0.075
GemmInt8	16	13	18K	21K	0.072

We designed multiple tile-based SoC prototypes with different compositions of accelerators and CPU cores. These prototypes include multiple CVA6 CPUs, a memory controller, an I/O tile, and many accelerator tiles. By using SystemC and Cadence Stratus HLS 20.25, we designed four different accelerator types, representing the TFLite operators and the supported data types (32-bit fixed-point or 8-bit integer): acc_Conv2d, acc_Conv2dInt8, acc_Gemm, and acc_GemmInt8. We applied several optimization techniques such as loop pipelining, loop unrolling, double buffering, and data chunking in order to balance the local computation and data-communication tasks. We implemented our FPGA prototypes by using Xilinx Vivado 2019.2 with a clock frequency of $78MHz$, which is determined by the critical path of the RISC-V CVA6 CPU. We deployed the SoCs on a Virtex Ultrascale XCVU440 FPGA board. TABLE II lists the FPGA resource utilization (i.e., BRAM, DSP, FF, and LUT counts), as well the power of each tile, as reported after synthesis by the Xilinx Vivado tool.

TFLite Workload Performance. The right-hand side of TABLE I reports the execution times (in ms) of many TFLite workloads with different data types running on different CPUs: the Intel i7-8700K (@3.7GHz), the Arm Cortex-A53 (@1GHz), and the RISC-V CVA6 (@78MHz). The columns with XNNPACK report the performance when running the workloads in software while leveraging the XNNPACK library [29]. XNNPACK provides software-based optimizations of low-level primitives for accelerating the execution of neural networks specified in high-level frameworks, such as TFLite, on architectures based on x86, Arm, and RISC-V CPUs.

Fig. 6 shows the performance that we obtained when running the workloads with WOLT compared to running the workloads with XNNPACK on the CVA6 CPU. Our baseline is the execution of the same workloads on the CVA6 CPU without XNNPACK (marked with a red line). XNNPACK (green bars) outperforms the baseline with a maximum of $4.07\times$ speedup for *MobileNet_v1*. However, WOLT (blue bars) achieves a better performance for all of the workloads. For some workloads (*ResNet10* and *SqueezeNet*), WOLT provides a speedup over $20\times$ thanks to the high utilization of the `Conv2d` operator (TABLE I). For the *ToyCar* and *LeNet*) workloads, which utilize the `FullyConnected` operator extensively, WOLT gives speedups of $4\times$ and $6\times$, respectively.

TFLite Workload Energy Efficiency. We defined the energy-efficiency gain as the ratio of the energy dissipated when running the workloads entirely on the RISC-V CPU over the energy dissipated when running them with WOLT:

$$\frac{E_{cpu}}{E_{acc}} = \frac{P_{cpu} * T_{cpu}}{P_{cpu} * T_{cpu} * F_{cpu} + P_{acc} * T_{acc} * F_{acc}} \quad (1)$$

where E_{cpu}, P_{cpu}, and T_{cpu} are the CPU's energy consumption, power dissipation, and execution time, respectively; F_{cpu} is the fraction of the workload computation executed on the CPU; similarly, E_{acc}, P_{acc}, T_{acc} and F_{acc} are the corresponding values obtained by running the workloads on the accelerators through WOLT's hardware delegation.

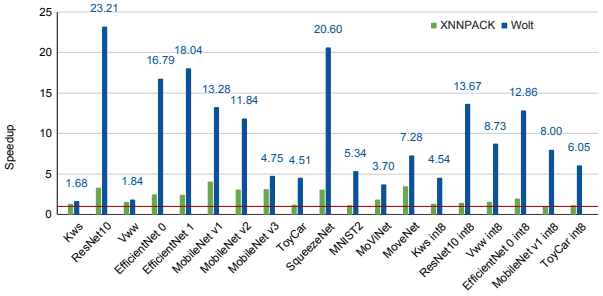


Fig. 6: Normalized speedup.

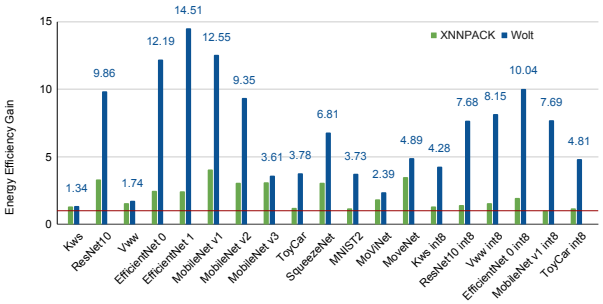
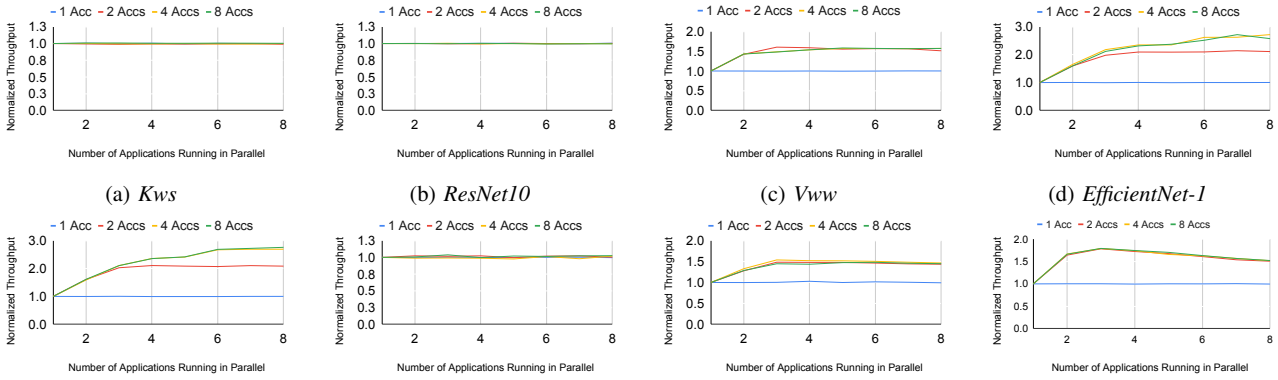


Fig. 7: Normalized energy-efficiency gain.

Fig. 7 illustrates the energy-efficiency gain obtained with WOLT for each workload. Similar to Fig. 6, the red line indicates the baseline where the workloads are executed on the RISC-V CPU alone and without XNNPACK. The results show that WOLT achieves higher energy efficiency than XNNPACK for all workloads; in particular, WOLT provides a maximum gain of $14.51\times$ in energy efficiency for *EfficientNet-1*.

WOLT Resource Manager: One CPU. For the next set of experiments, we applied the WOLT resource manager to the execution of real applications by using many different ML models. TABLE III lists these models together with the fractions of each application running on the CPU and on the accelerator, respectively. Fig. 8 reports the experimental results. The data-processing throughput, which is defined as the number of tasks executed per unit of time, is normalized with respect to the execution of the given workloads without using the WOLT resource manager on an SoC featuring one CPU and a single accelerator. For *MobileNet_v1* with two accelerators (red curve in Fig. 8e), one can note that the throughput reaches about $2\times$ and plateaus after running four workloads in parallel. If the number of available accelerators increases to four or eight (the yellow and green curves), the normalized throughput reaches $2.8\times$ with six or more applications running in parallel. On the other hand, we don't see a similar behavior when running multiple *ResNet10*, *Kws*, and *ToyCar* applications. The throughput remains constantly equal to 1. As reported in TABLE III, for *ResNet10*, *Kws*, *ToyCar* the time to run on a CPU



(a) *Kws* (b) *ResNet10* (c) *Vww* (d) *EfficientNet-1*

(e) *MobileNet_v1* (f) *ToyCar* (g) *MoviNet* (h) *MoveNet*

Fig. 8: The normalized throughput of running workloads in parallel with WOLT and one CPU core.

TABLE III: Workload CPU time vs. Accelerator time

	CPU		Accelerator	
	[sec]	F_{cpu}	[sec]	F_{acc}
Kws	2.61	81.31%	0.60	18.69%
ResNet10	2.31	63.41%	1.33	36.59%
Vww	3.6	46.57%	4.14	53.43%
EfficientNet-0	2.81	27.99%	7.23	72.01%
EfficientNet-1	3.6	26.25%	9.44	73.75%
MobileNet_v1	22.39	24.17%	70.27	75.83%
MobileNet_v2	25.2	31.97%	53.62	68.03%
MobileNet_v3	35.91	35.26%	65.94	64.74%
ToyCar	2.91	95.72%	0.13	4.28%
SqueezeNet	8.03	67.59%	3.85	32.41%
MNIST2	2.11	94.20%	0.13	5.80%
MoViNet	13.61	46.15%	15.88	53.85%
MoveNet	4.40	28.74%	10.91	71.26%

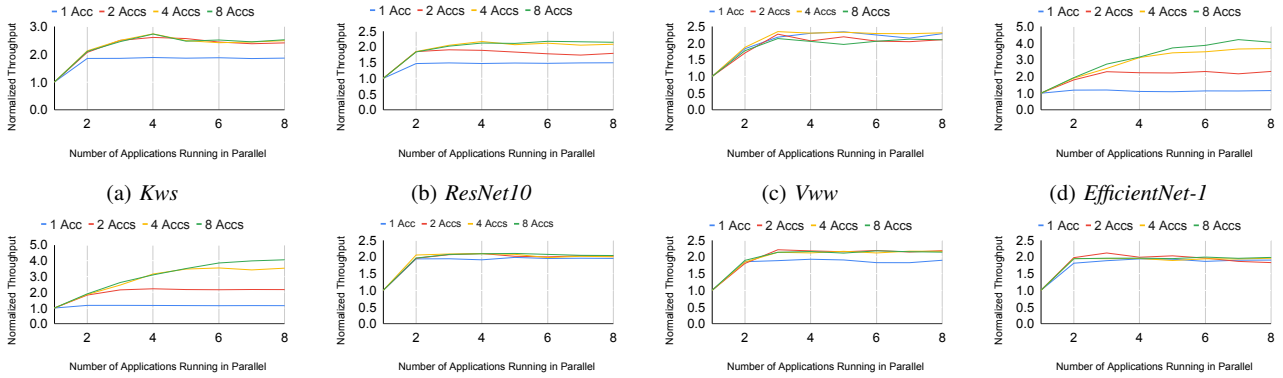
is greater than the time to run on the accelerators. Therefore, running multiple applications in parallel does not benefit from increasing the available accelerators. In the case of *Vww*, the throughput curves when more than one accelerator is available plateau after running two applications, as shown in Fig. 8c; this means that adding more than two accelerators does not help with the parallelization. Conversely, using more accelerators for *EfficientNet-1* yields better normalized throughput gains because, as reported in TABLE III, more than 73.5% of the execution time for this ML model is spent on the accelerators; specifically, running with two, four, eight accelerators yields $2\times$, $2.6\times$, and $2.7\times$ gains, respectively.

From the experiments and analysis above, we observe that when the portion of execution time spent on the CPU is greater than portion spent on one accelerator, the CPU is essentially the bottleneck. Therefore, adding more accelerators does not improve the performance when running multiple applications in parallel. Conversely, when the portion of execution time spent on the CPU is less than the portion spent on one accelerator, a performance gain can be obtained by increasing the number of accelerators running in parallel up to a given number that is application specific. This observation can be formalized as follows. When a workload spends more time on CPU than on accelerators ($F_{cpu} > F_{acc}$), using one accelerator brings all the possible performance gain. On the other hand, when a workload spends more time on accelerators than on CPU ($F_{acc} > F_{cpu}$), additional accelerators enhance performance until reaching a plateau, where the number of accelerators (N) exceeds the ratio of accelerator execution time to CPU execution time ($N > F_{acc}/F_{cpu}$).

WOLT Resource Manager: Two CPUs. Since we know from TABLE III that the performance of running some ML models is capped by the execution on the CPU, we investigated the impact of increasing the number of CPU cores. Fig. 9 shows the experimental results of analyzing SoC configurations that have 2 CPU cores and growing numbers of accelerators. The throughput is normalized with respect to the baseline case when the applications run on an SoC without using the WOLT resource manager. In the case of *ResNet10*, compared to Fig. 8b, Fig. 9b shows that the additional CPU cores allow reaching higher normalized throughput values: up to $1.5\times$ when there is only one accelerator available and up approximately $2.3\times$ when there are two or more accelerators available. Similarly, for *MobileNet_v1*, the normalized throughput reaches $4\times$ when utilizing 8 accelerators. In general, the comparison of the results of Fig. 9 with the corresponding ones of Fig. 8 confirms the hypothesis that, for certain applications where the CPU execution is the bottleneck, increasing the number of CPUs allows a better use of accelerators for an overall performance improvement.

WOLT Resource Manager: Multiple Workloads. In real-world applications like autonomous vehicles, a single SoC often needs to concurrently execute multiple ML workloads [40]. For example, in autonomous navigation, convolutional neural networks (CNNs) are used for tasks such as object detection and semantic segmentation, while deep reinforcement learning (DRL) learns optimal navigation policies. Visual simultaneous localization and mapping (VSLAM) utilizes CNNs for object detection, semantic segmentation, and image classification, while DRL is used for learning navigation strategies [41]. We investigated this multi-workload scenario using an FPGA-based SoC prototype with one CPU core and four `acc_Conv2d` accelerators, whose execution is dynamically orchestrated by the WOLT resource manager. We tested combinations of *ResNet10* (R), *Vww* (V), and *MobileNet_v1* (M) workloads.

Fig. 10 shows three combinations: (1) R and V, (2) R and M, and (3) R, V, and M. We compared the normalized throughput with and without the WOLT resource manager (blue bars in Fig. 10). TABLE III shows that F_{cpu} for *Vww*



(a) *Kws* (b) *ResNet10* (c) *Vww* (d) *EfficientNet-1*

(e) *MobileNet_v1* (f) *ToyCar* (g) *MoviNet* (h) *MoveNet*

Fig. 9: The normalized throughput of running workloads in parallel with WOLT and two CPU cores.

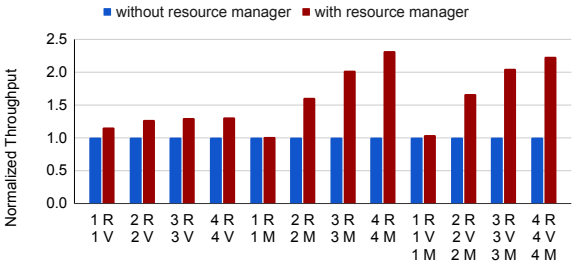


Fig. 10: Normalized throughput of running a combination of *ResNet10* (R), *Vww* (V), and *MobileNet_v1*(M).

(53.43%) is larger than for *ResNet10* (36.59%). *ResNet10* and *Vww* have comparable CPU performance with execution times of 4.5 sec and 3.5 sec, respectively. Combining *ResNet10* and *Vww* resulted in a smaller throughput gain (1.3× for 2R and 2V) compared to *Vww* alone (1.5× for 4V) (Fig. 10). However, combining *ResNet10* with *MobileNet_v1* aligns with trends observed for *MobileNet_v1*, due to its longer execution time, which dominates the overall performance. Similarly, running *ResNet10*, *Vww*, and *MobileNet_v1* together mirrors the *ResNet10* and *MobileNet_v1* combination due to the significant execution-time differences.

In summary, the WOLT resource manager improves performance when running multiple workloads in parallel, especially when the workloads have unbalanced execution times.

V. RELATED WORK

TFLite Delegates and Hardware Accelerators. TFLite delegates support the execution of TFLite workloads on heterogeneous SoCs for mobile devices. Lee *et al.* pioneered the TFLite-GPU delegate to use mobile-phone GPUs [6], showing its compatibility across diverse devices and comparable or superior performance to other SDKs tailored for specific hardware architectures. Jiang *et al.* profiled deep learning inference on smartphone GPUs with TFLite and the TFLite-GPU delegates [42], emphasizing the need for improved hardware and software solutions for optimal deep learning hardware delegation with TFLite. Different from the TFLite-GPU delegate, our WOLT TFLite delegate targets many-accelerator heterogeneous SoCs, which are typically more resource constrained.

Several tools and workflows were proposed for designing hardware accelerators targeting neural networks. DNNBuilder [43] is an automated tool for building hardware accelerators for DNN workloads on FPGAs. It generates accelerators for models implemented with frameworks like TensorFlow and Caffe by generating RTL components for each layer, and combining them into a single accelerator. Moreau *et al.* proposed the VTA [44], a programmable architecture for designing various deep learning model accelerators. Genc *et al.* introduced Gemmini [45], a systolic array accelerator that facilitates smooth integration in heterogeneous SoCs and manages different system-level scenarios. Harris *et al.* created the SECDA-TFLite toolkit [46], leveraging the TFLite delegate system to integrate DNN accelerators, providing an initial development environment within TFLite for designing hardware accelerators for specific workloads.

Unlike these projects, which focus on designing accelerators for neural networks with a preliminary software interface, WOLT enables the integration of real-world applications on many-accelerator heterogeneous SoCs with multi-core RISC-V CPUs. The WOLT TFLite delegate is based on a full SoC architecture with a multi-plane NoC as the main on-chip interconnect [27]. It supports executing multiple TFLite workloads in parallel, allocates several types of accelerators simultaneously, and can be tested on FPGA. Future work could integrate other state-of-the-art accelerators within WOLT as accelerators invoked by the WOLT TFLite delegate.

Resource Manager for Parallel Execution. Many studies address parallel execution of ML workloads on heterogeneous SoCs. Hill and Reddi re-targeted the Roofline model to capture task distribution and performance estimation of the accelerators in an SoC [47]. Kim *et al.* [13] developed AuRORA, a hardware-based resource manager that binds ML workloads to accelerators, requiring ISA extensions for CPU-client interaction. In contrast, WOLT offers a straightforward software-based solution for SoC accelerator management, requiring no ISA changes or additional hardware, yet significantly improving performance. Other works, e.g. Yu *et al.* [48] and Zhang *et al.* [49], target resource management in cloud-based systems.

TABLE IV summarizes the comparison with related works. To our knowledge, no previous solution executes multi-tenant

TABLE IV: Comparison with Related Work.

	platform	specialized acc	parallel exec	transparent deployment	FPGA
WOLT	edge	✓	✓	✓	✓
VTA [44]	edge	✓	-	-	✓
Gemmini [45]	edge	✓	-	-	-
SECDA-TFLite [46]	edge	✓	-	✓	✓
Aurora [13]	edge	✓	✓	-	-
AvA [48]	cloud	✓	✓	✓	-
Sinan [49]	cloud	-	✓	✓	-

ML workloads on lightweight many-accelerator architectures.

As pipeling accelerators with point-to-point communication offers better performance than memory-based communication [50], future work could explore its integration with the WOLT resource manager for resource-constrained SoCs.

VI. CONCLUSIONS

We presented WOLT, an end-to-end open-source solution for leveraging specialized hardware accelerators in TFLite applications on heterogeneous SoC architectures for embedded devices. WOLT decouples TFLite model design from accelerator implementation, thus enhancing flexibility and ease of deployment. Experiments on an FPGA-based SoC prototype show that WOLT effectively delegates TFLite operators to accelerators, resulting in performance and power gains over software execution on open-source embedded processors. WOLT advances edge computing by enabling efficient deployment of TFLite workloads on diverse SoC architectures. We released the contribution of this work in the public domain[1].

Acknowledgments: This work was supported in part by DARPA (C#: FA8650-18-2-7862), DOE (A#: "DE-SC0024458") and in part by the NSF (A#: 1764000). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

REFERENCES

[1] M. M. Amiri and D. Gündüz, "Machine learning at the wireless edge: Distributed stochastic gradient descent over-the-air," in *ISIT*, 2019.
[2] E. Li *et al.*, "Edge AI: On-demand accelerating deep neural network inference via edge computing," *IEEE Trans. on Wireless Communications*, 2019.
[3] C. Hao *et al.*, "Enabling design methodologies and future trends for Edge AI: Specialization and codesign," *IEEE Design & Test*, 2021.
[4] S. K. u. Zaman *et al.*, "LiMPO: lightweight mobility prediction and offloading framework using machine learning for mobile edge computing," *Cluster Computing*, 2022.
[5] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," *arXiv:1603.04467*, 2016.
[6] J. Lee *et al.*, "On-device neural net inference with mobile GPUs," *arXiv:1907.01989*, 2019.
[7] H. Sun *et al.*, "Joint offloading and computation energy efficiency maximization in a mobile edge computing system," *IEEE Trans. on Vehicular Technology*, 2019.
[8] J. Bi *et al.*, "Energy-optimized partial computation offloading in mobile-edge computing with genetic simulated-annealing-based particle swarm optimization," *IEEE Internet of Things Journal*, 2020.
[9] X. Chang *et al.*, "Interpretable machine learning in sustainable edge computing: A case study of short-term photovoltaic power output prediction," in *ICASSP*, 2020.
[10] G. Eichler *et al.*, "MasterMind: many-accelerator soc architecture for real-time brain-computer interfaces," in *ICCD*, 2021.
[11] M. Zanghieri *et al.*, "sEMG-based regression of hand kinematics with temporal convolutional networks on a low-power edge microcontroller," in *COINS*, 2021.
[12] G. Eichler *et al.*, "MindCrypt: the brain as a random number generator for soc-based brain-computer interfaces," in *ICCD*, 2023.
[13] S. Kim *et al.*, "AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads," in *MICRO*, 2023.
[14] Y. Zhang *et al.*, "Hello edge: Keyword spotting on microcontrollers," *arXiv:1711.07128*, 2017.
[15] C. Banbury *et al.*, "MLPerf tiny benchmark," *arXiv:2106.07597*, 2021.
[16] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.
[17] A. Chowdhery *et al.*, "Visual wake words dataset," *arXiv:1906.05721*, 2019.
[18] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Machine Learning Research*, 2019.
[19] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv:16021704.04861*, 2017.
[20] M. Sandler *et al.*, "MobileNetV2: Inverted residuals and linear bottlenecks," *arXiv:1801.04381*, 2019.
[21] A. Howard *et al.*, "Searching for MobileNetV3," *arXiv:1905.02244*, 2019.
[22] Y. Koizumi *et al.*, "ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection," *WASPAA*, Oct 2019.
[23] F. N. Iandola *et al.*, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *arXiv:1602.07360*, 2016.
[24] TensorFlow Examples, https://github.com/tensorflow/examples/tree/master.
[25] D. Kondratyuk *et al.*, "MoViNets: Mobile video networks for efficient video recognition," *arXiv:2103.11511*, 2021.
[26] R. Bajpai and D. Joshi, "MoveNet: A deep neural network for joint profile prediction across variable walking speeds and slopes," *IEEE Trans. on Instrumentation and Measurement*, 2021.
[27] P. Mantovani *et al.*, "Agile SoC development with open ESP," in *ICCAD*, 2020.
[28] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology," *VLSI*, 2019.
[29] XNNPACK, https://github.com/google/XNNPACK.
[30] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS*, 1967.
[31] D. Giri *et al.*, "ESP4ML: Platform-based design of systems-on-chip for embedded machine learning," in *DATE*, 2020.
[32] K.-L. Chiu *et al.*, "EigenEdge: Real-time software execution at the edge with RISC-V and hardware accelerators," in *CPS-IoT Week*, 2023.
[33] D. Giri *et al.*, "Accelerator integration for open-source SoC design," *IEEE Micro*, 2021.
[34] Embedded Scalable Platform (ESP), www.esp.cs.columbia.edu.
[35] L. P. Carloni, "The case for embedded scalable platforms," in *DAC*, 2016.
[36] J. Cong *et al.*, "Architecture support for accelerator-rich CMPs," in *DAC*, 2012.
[37] E. G. Cota *et al.*, "An analysis of accelerator coupling in heterogeneous architectures," in *DAC*, 2015.
[38] SPARC LEON3, www.gaisler.com/index.php/products/processors/leon3.
[39] RISC-V Ibex - lowRISC, https://github.com/lowRISC/ibex.
[40] S.-C. Lin *et al.*, "The architectural implications of autonomous driving: Constraints and acceleration," in *ASPLOS*, 2018.
[41] S. Mokssit *et al.*, "Deep learning techniques for visual SLAM: A survey," *IEEE Access*, 2023.
[42] S. Jiang *et al.*, "Profiling and optimizing deep learning inference on mobile GPUs," in *APSys*, 2020.
[43] X. Zhang *et al.*, "DNNBuilder: an automated tool for building high-performance dnn hardware accelerators for FPGAs," in *ICCAD*, 2018.
[44] T. Moreau *et al.*, "Leveraging the VTA-TVM hardware-software stack for FPGA acceleration of 8-bit ResNet-18 inference," in *ReQuEST*, 2018.
[45] H. Genc *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *DAC*, 2021.
[46] J. Haris *et al.*, "SECDA-TFLite: A toolkit for efficient development of FPGA-based DNN accelerators for edge inference," *Journal of Parallel and Distributed Computing*, 2023.
[47] M. Hill and V. Janapa Reddi, "Gables: A roofline model for mobile SoCs," in *HPCA*, 2019.
[48] H. Yu *et al.*, "AvA: Accelerated virtualization of accelerators," in *ASPLOS*, 2020.
[49] Y. Zhang *et al.*, "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *ASPLOS*, 2021.
[50] K.-L. Chiu *et al.*, "An analysis of accelerator data-transfer modes in noc-based SoC architectures," in *HPEC*, 2023.

[1] https://github.com/klchiu/esp/tree/wolt