Autotuned Distribution of Multi-DNN Workloads on Multi-Accelerator SoCs

Federico Nicolás Peccia, Avik Bhatnagar, Oliver Bringmann
FZI Research Center for Information Technology, University of Tübingen
Germany
peccia@fzi.de, bhatnagar@fzi.de, oliver.bringmann@uni-tuebingen.de

Abstract— **Many machine learning applications require heterogeneous Deep Neural Networks (DNNs) to work collaboratively. Although several works have focused on how to serve these systems using cloud solutions, less attention has been paid to the edge scenarios. Particularly, coordinating heterogeneous AI workloads across a custom application-specific System-on-Chip (SoC) with multiple accelerators presents significant challenges. This work first demonstrates how to implement such a baseline system using a SoC generator framework, performs an ablation study prototyping different versions on an FPGA, details how an RTOS can be used to achieve model parallelism on multiple accelerators, and identifies gaps and limitations by executing a multi-DNN autonomous driving application. To improve the utilization of the system and increase throughput, we propose a method to distribute the execution of individual layers across accelerators. Instead of partitioning all layers in the same manner and statically allocating them at compile time, we select the ideal partitioning for each one during compilation using an autotuning process, and then dynamically assign them to the available accelerators during runtime. We analyze the variability in layer execution in a system with multiple accelerators and use this information to guide the runtime allocation of partitions. We demonstrate that our method achieves a mean 29 % and 40 % improvement in accelerator utilization and throughput over the model parallelism baseline, and a 10 % and 9 % improvement over a round-robin runtime distribution of partitions.**

Multi-tenant DNN, FPGA, RTOS, SoC

I. INTRODUCTION

Machine learning applications are becoming increasingly complex, with some of them composed of multiple heterogeneous DNNs, each one trained for a particular task. Some edge multi-DNN application examples are autonomous driving [1] (object detection, tracking, and trajectory prediction) or AR/VR [2] (depth estimation and face recognition).

Multiple works have focused on INFerence-as-a-Service (INFaaS) solutions, where multi-accelerator SoCs are used as backend components of cloud inference systems [3],[4]. These works focus on optimizing cloud-specific metrics like Quality of Service (QoS) or service level agreement (SLA) satisfaction rates. Works that focus on edge solutions usually use commercial-off-the-shelf (COTS) embedded GPUs to deploy multi-DNN systems [5]}.

But the orchestration of heterogeneous DNNs on multi-accelerator custom application-specific SoCs has not been explored as thoroughly. Moreover, both edge and cloud works have only focused on evaluation at the simulator level and ignored implementation challenges. For example, the computational complexity of some of the runtime scheduling algorithms proposed by cloud solutions to dynamically allocate and distribute the execution of layers across the available accelerators typically prevents their deployment on the edge.

We present how such a multi-DNN multi-accelerator custom application-specific SoC for edge applications can be realized using an open-source hardware (HW) generation workflow based on Chipyard [6] and the Gemmini accelerator [7]. We show how the Zephyr RTOS [8] can be configured to achieve model parallelism using a reference application for autonomous driving requiring two heterogeneous DNNs. By executing the multi-DNN system on different HW versions, we identify that model parallelism alone limits the utilization of the available accelerators, resulting in idle resources.

To increase the utilization of the SoC's accelerators, this work proposes to parallelize the execution of individual DNN layers across multiple accelerators. But instead of relying on heuristics to decide how to partition each layer, we propose a workflow based around the TVM Deep Learning compiler [9] to automatically search for the best partitioning configuration for each layer at compile time (autotuning). During runtime, a lightweight allocation algorithm dynamically selects to which accelerators to assign the partitions of each layer. We show how the latency of a layer varies when multiple accelerators share the same memory subsystem, and how this distribution of possible latencies needs to be taken into account when calculating the pending load of each accelerator.

Our method achieves a mean 29 % improvement in accelerator utilization when compared against the model parallelism baseline. Compared to distributing the layer partitions in a round-robin fashion, our method achieves a mean 10 % increase in accelerator utilization. Additionally, the throughput of the partitioned DNN is increased by 40 % and 9 % against the model parallelism baseline and round-robin distribution.

This research was funded by the German Federal Ministry of Research, Technology, and Space (BMFTR), and the EU ChipsJU as part of the project RIGOLETTO under Grant 16MEE0547 and 101194371, respectively.

Manuscript received May 06, 2026; revised July 15, 2026; accepted August 07, 2026; published August 25, 2026.
Issue category: Special Issue on DSD/SEAA 2026 on Works in Progress (WiP) Session, Krakow, Poland, Sept. 2026.
Paper category: Full
DOI: doi.org/10.64552/wipiec.v12i2.134

The rest of this paper is organized as follows. First, Section II discusses previous cloud and edge works. Then, Section III shows how such a multi-DNN multi-accelerator custom edge system can be realized using an RTOS and provides the ablation study measurements showcasing its limitations. Section IV presents our proposal to better utilize the idle accelerators of the system, which is then evaluated in Section V. Finally, Section VI concludes the work.

II. Related Work

Initial works on the deployment of multi-DNN workloads focused on interleaving the execution of DNNs on the same accelerator (*temporal* multi-tenancy). For example, AI-MT [10] proposed a HW scheduler to achieve this. On the other hand, Layerweaver [11] implemented a software (SW) scheduler to achieve the same goal. [12] used a preemption mechanism for accelerators and then proposed PREMA, an algorithm that leverages this preemption feature to manage DNN workloads with different priorities.

Other works focused on *spatial* multi-tenancy, where different workloads are assigned to different accelerators, adding parallelism to the execution. Planaria [13] proposed a HW design that allows the creation of sub-accelerators dynamically, and used this dynamic reconfiguration to run different workloads in parallel. Herald [14] proposed a way to orchestrate DNNs on heterogeneous dataflow accelerators by assigning each layer to the accelerator best suited for it. MAGMA [15], [17] and COMB [18] use Genetic Algorithms (GA) to find an optimized mapping for a set of DNNs on multiple accelerators sharing the same memory interface and take into account bandwidth allocation during the optimization. Layer-Puzzle [4] uses a dynamic scheduler framework to parallelize the execution of individual layers. CD-MSA [3] uses a deadline-aware scheduler to improve Quality of Service (QoS) guarantees for cloud use cases. Finally, works like Versa-DNN [19] proposed versatile HW accelerators able to reconfigure themselves into sub-accelerators during runtime, and present the appropriate algorithms to utilize them.

Table I compares our work against the literature. **Parallelism level** refers to the scheduling granularity: allocating entire *layers* to each accelerator, or partitioning the execution of a layer and distributing the resulting *tensor* operations. **Allocation** references the runtime behavior: some works decide the accelerator assignment offline (*static*); others do so during runtime (*dynamic*). The target **Application** modifies design decisions, particularly in dynamic allocation works: given its complexity, most dynamic algorithms developed for INFaaS would not be suitable for edge cases.

Other works have also leveraged TVM to distribute DNN execution across multiple accelerators in custom SoCs, such as MATCH [20]. However, our approach differs fundamentally in three key aspects: 1) while MATCH distributes the execution of complete layers, our work partitions individual layers into finer-grained tensor operations and distributes them, 2) MATCH performs accelerator mapping at compile time, whereas our distribution decisions occur at runtime, and 3) MATCH supports single DNN execution, while our runtime enables the concurrent execution of multiple DNNs.

TABLE I. Our work compared against others

	Year	Parallelism level	Allocation	Application	Evaluated on
PREMA [12]	2020	-	-	INFaaS	Simulator
AI-MT [10]	2020	-	-	INFaaS	Simulator
Planaria [13]	2020	Tensor	Dynamic	INFaaS	Simulator
Herald [14]	2021	Tensor	Static	Both	Simulator
Layerweaver [11]	2021	Layer	Static	INFaaS	Simulator
MAGMA [15]	2022	Layer	Static	INFaaS	Simulator
H2H [16]	2022	Layer	Static	INFaaS	Simulator
Layer-Puzzle [4]	2023	Tensor	Dynamic	INFaaS	Simulator
[17]	2023	Tensor	Static	Edge	Simulator
CD-MSA [3]	2023	Tensor	Dynamic	INFaaS	Simulator
COMB [18]	2023	Layer	Static	Edge	Simulator
Versa-DNN [19]	2024	Tensor	Dynamic	INFaaS	Simulator
Ours	**2026**	**Tensor**	**Dynamic**	**Edge**	**FPGA**

III. Baseline multi-DNN multi-accelerator system implementation

A generic multi-DNN multi-accelerator system is comprised of a set of M accelerators $A = \{acc_0, \dots, acc_{M-1}\}$, and executes N DNN workloads $G = \{G_0, \dots, G_{N-1}\}$. During runtime, requests are received for each DNN workload, and the system orchestrates the execution of each one of them on the available accelerators. The request arrival rate is application-specific: it could be periodic (like images coming from a camera stream) or batched (a one-time batch of requests). From these definitions, several challenges arise.

First of all, the number and size of the available accelerators need to be defined. These are usually constrained by area or power requirements, but the application to be executed on them should also be taken into account. For example, an application requiring high parallelism may benefit more from multiple smaller accelerators. On the other hand, an application focused on throughput may already fulfil its requirements with only one powerful accelerator.

Second, some access control mechanism needs to be provided to avoid concurrent SW access to the same HW resource. This can be realised using an RTOS resource, like a mutex, to block each accelerator until it finishes its computation. There are two possible blocking granularities. In the first one (model-wise), each request blocks the accelerator until it executes all its layers. From a request point of view, this is the *non-preemptible* case: once a request starts to execute on one accelerator, the RTOS cannot select another request to run on the same accelerator. In the second option (layer-wise), each request blocks the accelerator only until it finishes the execution of its current layer. This is the *preemptible* case: the RTOS is allowed to interleave the execution of layers from different requests on the same accelerator.

A third execution granularity exists, where the execution of a layer on an accelerator is interrupted before finishing its execution, in order to execute a new layer on it. But this would require storing the current accelerator state and then retrieving it once the layer can continue its execution. If a HW mechanism is not available to take care of this, it can result in a considerable

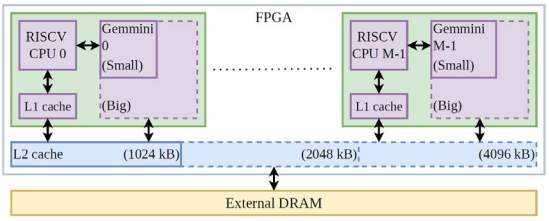


Figure 1 Multi-accelerator SoC under analysis

time overhead. So we focus our analysis only on the first two granularities.

Finally, once a new request arrives, the system needs to select which of the available accelerators should execute the model for this particular request. New requests should ideally be assigned to idle (or soon-to-be) accelerators.

A. *Selected application*

We choose a 3D object detection application to demonstrate a multi-DNN workload use case, where the 3D object properties are estimated from a single camera image by leveraging deep learning and geometric constraints [1]. Our implementation of the paper consists of two DNNs: a YOLOv7-tiny model to obtain all 2D object detections within an image in a single-shot operation, and a VGG-19-based DNN that regresses the 3D object parameters *for each* detected object.

Since the number of times the VGG model needs to be executed is dependent on the number of detections found by the YOLO model, the exact number of VGG requests is **unknown at compile time**. Therefore, we cannot statically define where each VGG model request needs to be executed. This is why some method is needed during runtime to dynamically assign requests to different accelerators.

B. *Hardware/software baseline*

Our SoC (Fig. 1) is built around the Chipyard generator framework. We implement *M* instances of the Gemmini accelerator, which contains a parametrized systolic array of DIMxDIM Processing Elements (PEs). This accelerator was selected as its applicability for real-time applications was already analyzed in [21]. The Zephyr RTOS is used to orchestrate the execution of the DNNs in *G*. To map the DNN operations onto the accelerator, we used the open-source integration in [22].

We configure the RTOS with a set of threads and use semaphores to exchange data between them. One thread is instantiated for each combination of workload in *G* and accelerator in *A*. Threads can only execute on the accelerator they are assigned to. Each new workload request is signalled through a semaphore to all the threads that can manage the corresponding model. This results in *automatic* model parallelism: if multiple requests are received, no matter the workload, they can be processed by different threads and therefore be executed on different accelerators.

Fig. 2 shows how these SW aspects modify the execution when one or two accelerators are available, and the metrics that

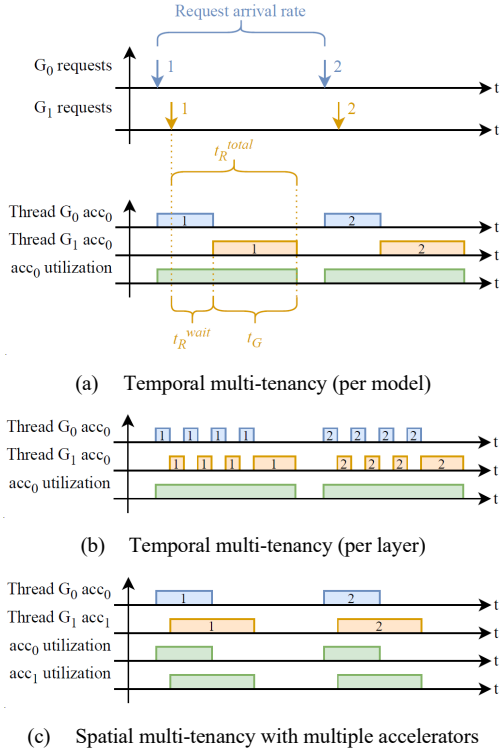


Figure 2 Different access control configurations in a multi-DNN multi-accelerator system.

can be evaluated in such a system (t_R^{wait}: request arrived but is waiting to be executed, t_G: actual model execution time, $t_R^{total} = t_R^{wait} + t_G$). For example, when only one accelerator is available, model-wise (Fig. 2a) or layer-wise blocking (Fig. 2b) modifies both t_R^{wait} and t_R^{total}. When two accelerators are available (Fig. 2c), threads assigned to different accelerators can execute the requests in parallel. The last configuration with two accelerators already shows one of the limitations of this approach, which we address in Section IV: once acc_0 finishes executing the G_0 request, it sits idle when it could be executing part of the G_1 request.

To explore the design space of possible SoC configurations, we parameterize 1) the number of accelerators M, 2) the size of the accelerators, between *Small* (16x16 PEs, 256 kB scratchpad) and *Big* (32x32 PEs, 512 kB scratchpad), and 3) the L2 cache size (1024, 2048, and 4096 kB). From the SW point of view, we also parameterize the mutex access control per accelerator (model- or layer-wise). All HW versions were implemented on a ZCU111 AMD FPGA development board, with a clock of 100 MHz[1].

For each combination of HW/SW parameters, we explore the behavior of the system under different benchmarks, each with different request arrival rates: 1) *Simple*, where one request

[1] We fixed the same frequency for all of them to focus our analysis only on the SoC-level parameters. Also, some configurations are excluded because these did not fit onto the FPGA

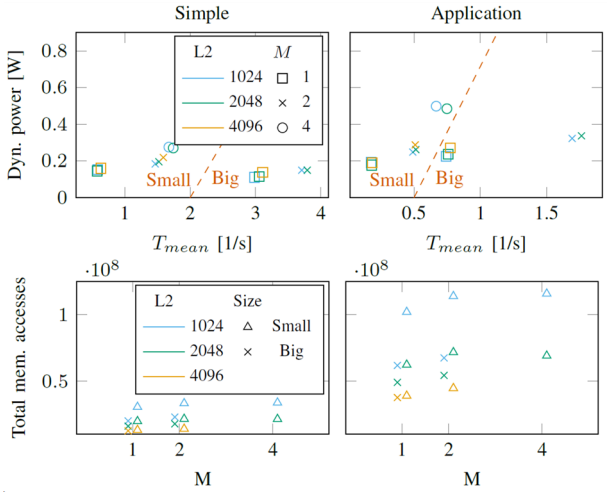


Figure 3 Measurements for different request arrival rates, grouped by L2 cache size, amount of accelerators available (*M*) and accelerator size.

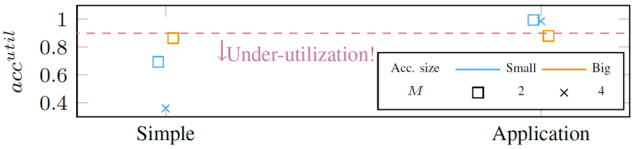


Figure 4 Mean accelerator utilization comparison.

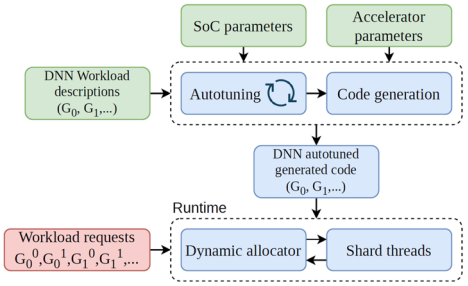


Figure 5 Proposed workflow for the autotuned-based distribution of DNNs across multiple accelerators.

is generated for each workload every second, and 2) *Application*, simulating the application's behavior, where every second one request for the YOLO model and a batch of four requests[2] for the VGG workload is generated (similar to the multi-stream arrival rate used by MLPerf Tiny [23]).

C. Measurements

We executed each benchmark for 3 seconds and then waited until all requests were processed (we call the total elapsed time $t_{benchmark}$). We also recorded the power consumption of the FPGA and the traffic between the L2 cache and the external DRAM during the entire execution. For each model's request, we record all metrics defined in Fig. 2, and define the mean throughput across all N models as Equation (1), with R_n being the amount of requests received for model *n* and $t_{R_r}^{total}$ the total time needed to finish request R_r.

$$T_{mean}[1/s] = \frac{1}{N}\sum_{n=0}^{N}\left(\frac{1}{R_n}\sum_{r=0}^{R_n}\frac{1}{t_{R_r}^{total}}\right) \quad (1)$$

Fig. 3 shows the behavior of different HW versions under different request arrival rates[3]. Increasing *M* brings a significant throughput improvement, especially during the *Application* benchmark, where several simultaneous requests can be parallelized. Similarly, increasing the size of the accelerators brings even bigger benefits: for example, having only one *Big* accelerator is already better than having 4 *Small* ones both in terms of mean throughput *and* power.

Increasing the L2 cache size improves the mean throughput, as less data needs to be exchanged with the external memory. This can also be appreciated in terms of external memory accesses (reads + writes), as these are reduced when the L2 cache is increased. But interestingly, these also increase when more accelerators are introduced. This is attributed to the shared nature of the L2 cache; concurrent execution induces cache line evictions, and data utilized by one accelerator may be displaced by the memory demands of others.

Although we do not observe a change in T_{mean} when blocking the accelerators per layer or per model, we do observe changes in t_G and t_R^{wait} [4]. The behavior is in line with the expected one presented in Fig. 2.

$$acc^{util} = \frac{1}{M}\sum_{i=0}^{M}\frac{t_{acc_i}^{busy}}{(t_{benchmark} - t_{idle})} \quad (2)$$

Finally, the mapping tool in [22] takes care of the individual PE utilization, so we analyze the mean accelerator utilization from a systems perspective (Equation (2)). $t_{acc_i}^{busy}$ represents the time accelerator *i* was busy processing a layer, and t_{idle} the time no accelerator is busy because there are no more requests to process. By subtracting t_{idle} from $t_{benchmark}$, acc^{util} only measures utilization while there are pending requests to process. Conceptually, if an accelerator is not being used while others are, the utilization for this accelerator decreases, as it could be potentially used to offload some computation from the other busy accelerators.

Fig. 4 shows that some configurations present an under-utilization of their accelerators ($acc^{util} < 0.9$). This happens in the *Simple* benchmark with both accelerator sizes, and in the *Application* benchmark with the *Big* size. In the rest of the cases, the accelerators are already constantly occupied because the arrival rate is too fast compared to the time needed by one accelerator to process one request.

IV. Proposal

Fig. 5 presents our proposal to improve the utilization of the idle resources identified in Section III-C. The compilation process receives the description of each DNN workload, the parameters of the SoC (L2 cache size, number of accelerators), and of the accelerator (for our example with Gemmini accelerators: number of PEs, size of internal scratchpad

[2] We fixed the VGG model requests to four to ensure reproducible results across different benchmark executions with different parameters and HW configurations.

[3] We only report dynamic power, as in an FPGA, the static power is fairly constant across all HW versions.

[4] Figure is omitted because of space limitations.

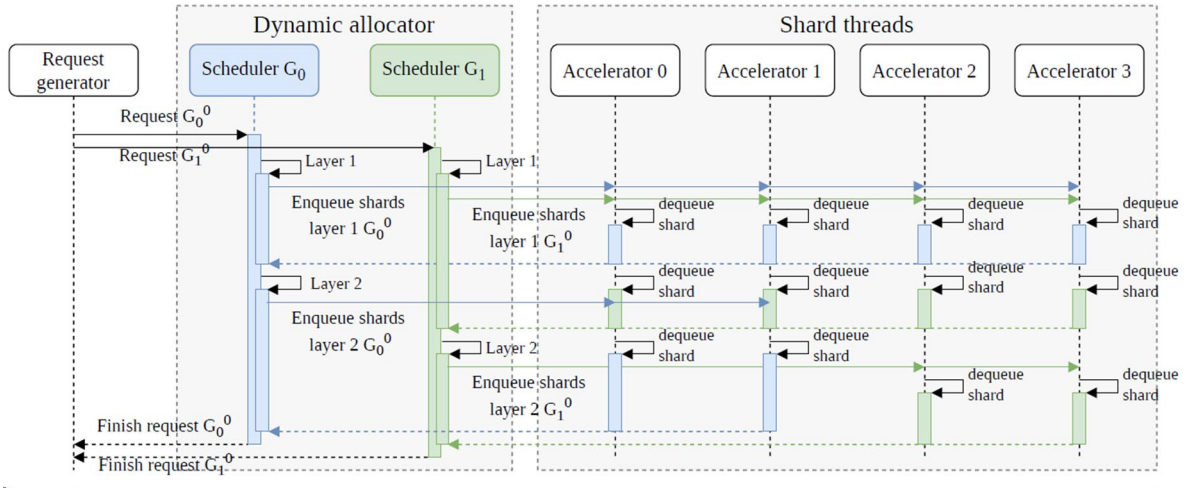
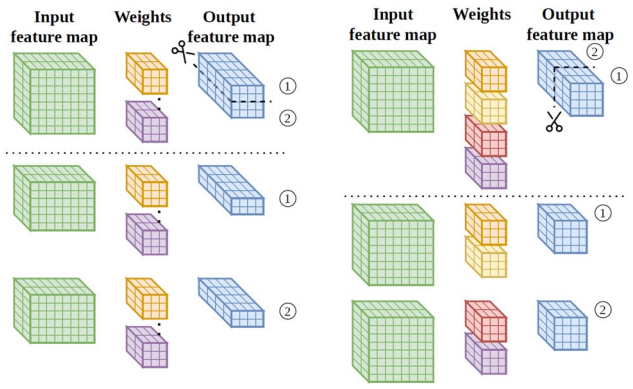


Figure 6 Example of the message passing between threads in the proposed RTOS runtime configuration.



(a) Row partition (b) Channel partition

Figure 7 Partition options examples for convolutions.

memories, etc). Then, the autotuning process finds the best way to partition each layer of the model across the available accelerators using AutoTVM [9]. During runtime, requests are received for each workload, and the dynamic allocator assigns the partitions (or *shards*) of each layer to a different accelerator.

A. Autotuning for partitioning point identification

AutoTVM has been proven to be a reliable tool for tuning the execution of tensor operations on both commercial and custom HW. This tuning process modifies the tensor operation by applying loop transformations and evaluates each candidate on the actual HW to find the one that minimizes its execution latency. Indeed, this was already applied to the Gemmini accelerator for the tuning of convolutions and other layers [24], but it only focused on one accelerator.

Although initially developed for the design exploration of loop transformations like tiling and reordering (internal to each tensor operation), we extended AutoTVM to consider two new optimization dimensions: how the Output Feature Map (OFM) of a convolutional layer is partitioned (channel- or row-wise [25]) and how many partitions are generated (limited to the maximum number of accelerators). Instead of using heuristics to decide how to partition each layer, we integrate this decision into the AutoTVM optimization framework. AutoTVM then finds the partition methodology that minimizes the latency for each layer (these measurement already encompasses the memory traffic overhead of sharing data between multiple accelerators). The one-time overhead of running this process for a particular DNN is analyzed in Section V-B.

Fig. 7 illustrates the differences between the partitioning configurations explored by our method. Using row partition (Fig. 7a) reduces the size of the input and output feature maps needed by each shard, but channel partition (Fig. 7b) can also be used to reduce the size of the weight tensors needed by each partition. The first layers from a DNN, which tend to have smaller weight tensors but larger OFMs, may benefit from row partitioning, in contrast to the last layers from the network, which may be better suited for channel partitioning.

This step only finds the best partitioning configuration for each layer. The actual shard-accelerator assignment is done at runtime, as explained in the next Section.

B. Runtime dynamic allocation of shards

To be able to assign these shards to the available accelerators dynamically during runtime, we expanded the RTOS configuration presented in Section III-B, originally thought for model parallelism. For each model, we instantiate a *scheduler* thread, which receives a request and enqueues shards into the available accelerators through the *shard* threads' queues. Fig. 6 presents a basic example of how this system would behave during runtime. Each scheduler thread first enqueues the shards of its respective first layers. The shard threads then process them until their queues are empty, and report back to the original scheduler threads every time a shard is finished. During the execution of both models' second layers, because there are two accelerators idle, the scheduler thread for

```
Require: accum_load, accum_lat, method, next_acc
 1: for layer in G_x do
 2:     assigned_acc = []
 3:     for shard in layer.shards do
 4:         ▷ Get next accelerator ◁
 5:         if method = ROUNDROBIN then
 6:             acc_idx = (next_acc + 1) % M
 7:             next_acc += 1
 8:         else if method = LOAD then
 9:             acc_idx = idxmin(accum_load)
10:             accum_load[acc_idx] += 1
11:             assigned_acc.push(acc_idx)
12:         else
13:             acc_idx = idxmin(accum_lat)
14:             lat = get_mean_lat(shard)
15:             accum_lat[acc_idx] += lat
16:             assigned_acc.push((acc_idx,lat))
17:         ▷ Add shard to specific accelerator queue ◁
18:         enqueue(acc_idx, shard)
19:     ▷ Wait for all shards to finish ◁
20:     wait_for_shards_ready()
21:     ▷ Reset accelerator usage ◁
22:     if method = LOAD then
23:         for acc_idx in assigned_acc do
24:             accum_load[acc_idx] -= 1
25:     else if method = MEAN then
26:         for (acc_idx, lat) in assigned_acc do
27:             accum_lat[acc_idx] -= lat
28: return
```

Algorithm 1 Scheduler thread pseudocode.

G_1 can decide to execute the shards of its second layer on them, thus improving utilization.

Algorithm 1 presents the pseudocode for each scheduler thread. Variables *accum_load*, *accum_lat*, and *next_acc* are shared across all scheduler threads, and their access is adequately protected to avoid concurrent access. Variable *method* selects how to obtain the accelerator to which the next shard will be assigned.

The dynamic allocation comes into play during the selection of *where* to enqueue the shards of a particular layer (Line 4). A simple load balancing could be achieved by assigning shards to accelerators in a round-robin fashion (Line 6). But this method would neglect how many shards are still pending execution on each shard thread, and therefore, how many shards need to be executed on the selected accelerator before the current shard is executed.

A second approach is to assign the shards based on the already available load of each shard thread queue, and by assigning each shard to the queue with fewer pending shards (Line 9). List *accum_load* keeps track of the assigned shards per accelerator, and then resets it accordingly when the shards finish (Line 24). But this method ignores the latency of each pending shard in the queue, which is necessary to know which shard thread will become idle first.

To solve this, the assignment should then take into account the load of each queue, but in terms of *expected latency* (Line 13). The dynamic allocator can then estimate which threads will finish their pending shards faster and assign new shards to them to distribute the load.

But calculating this *expected latency* is not trivial, as variations exist in layer execution time when more than one accelerator is available in the system. Fig. 8 presents the latency of individual layers obtained from the baseline measurements from Section III-C[5]. The more accelerators there are available in the system, the more the latency of a particular layer varies across different requests. For systems with 2 and 4 available accelerators, the latency of a layer cannot be represented by a single value anymore, but by a *distribution* of possible latencies. This behavior is a result of the sharing of the memory subsystem across all accelerators, as described in Section III-C, and is in line with the observed increase in memory accesses when *M* increases shown in Fig. 3.

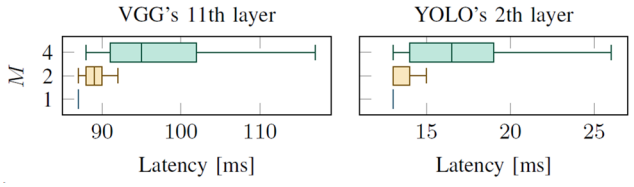


Figure 8 Distribution of latencies measured on a SoC with *Small* accelerator/s and 2048 kB L2 cache size..

To keep track of how much time each shard thread will still be busy during runtime, we need to estimate each shard's latency *every time* one is allocated. Based on the analysis of Fig. 8, we propose to use the *Mean* of previous measurements for this (Line 14}). List *accum_lat* keeps track of this *expected latency* per accelerator, and then resets it accordingly when the shards finish (Line 27).

The decision of where to assign each shard needs to be simple, as this needs to be executed by each scheduler thread every time a shard needs to be executed, and thus cannot insert unnecessary overhead in the execution of each request. The *Round-robin* allocation method is the simplest, as it only requires returning the next accelerator. The other two methods (*Load* and *Mean*), although comparing different metrics, are also simple, as these only require *M*-1 lookups to find the accelerator with the least pending latency or the least queued shards. This makes these three methods lightweight and suitable for implementation on resource-constrained devices.

V. Evaluation

In this section, we evaluate the impact of our proposal by applying our method to the VGG model of our representative use case. We only use our method on the VGG model because in a real scenario this model would be executed *for each* detected object in the scene, so it would benefit more from a faster completion time thanks to distributed execution across multiple accelerators. We only focus on hardware setups where the utilization is lower than 0.9 (Fig. 4), as these are the cases that can be improved by taking advantage of idle accelerators. The model parallelism measurements presented in Section III are used as baselines.

A. Throughput and utilization improvements

Fig. 9 shows, for each HW version and benchmark, how much the mean accelerator utilization and throughput are

[5] We observe the same behavior in all layers and across all HW versions. On average, because of competing access to memory, using two and four accelerators increases the mean measured latency for each layer by 31% and 91% respectively.

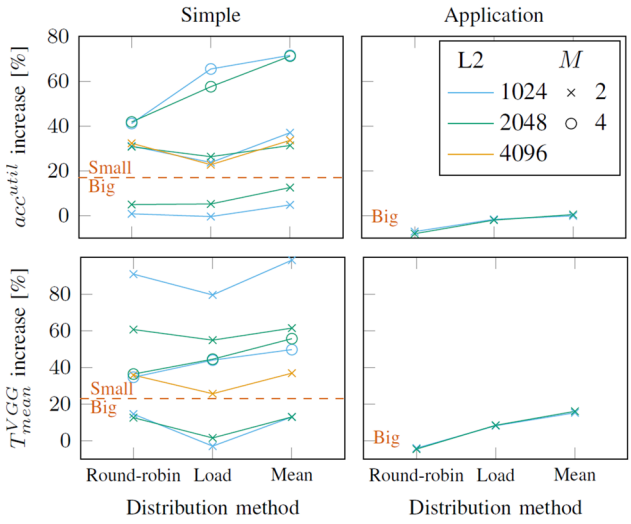


Figure 9 Improvements in acc^{util} and throughput for the VGG model using different distribution strategies for both benchmarks (compared against the corresponding best result from Section III).

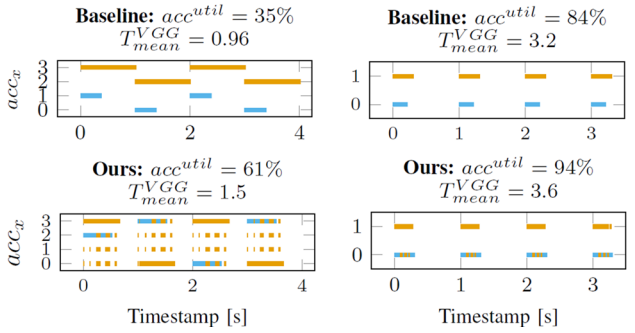


Figure 10 Execution traces for the Simple benchmark for an SoC with 4 Small accelerators (left) and one with 2 Big accelerators (right), showcasing the increase in acc^{util} and T^{VGG}_{mean} by using our method.

improved when compared against the baseline from Section III-C. The dynamic scheduler using the *Mean* latency estimation for the allocation of shards is constantly better than both the *Round-robin* distribution of shards and the one based only on the *Load* of each queue. The improvement increases as more accelerators are available, which makes sense, as these are the cases with more idle accelerators, which provide more opportunities for improvement. For the *Simple* benchmark, our dynamic allocator using the *Mean* method is able to achieve a mean 37 % increase in acc^{util}. For the *Application* benchmark, the increase is only 0.2 %, as these HW versions already had a high accelerator utilization (Fig. 4). Overall, our dynamic allocator is able to achieve a 29 % mean increase in acc^{util} when compared against the model parallelism baseline, and a 10 % increase when compared against the *Round-robin* method.

In terms of throughput, our dynamic allocator method using the *Mean* method achieves a mean 40 % increase when compared against the model parallelism baseline, and a 9 % improvement against the *Round-robin* distribution method.

For both accelerator utilization and throughput, the *Load* method is not reliable, as only counting the number of shards assigned to each accelerator is not enough to adequately decide which accelerator will become idle faster. Our *Mean* method always surpasses it.

Improvements are significant for the SoCs with smaller accelerators, but lower for the other HW versions. A significant reason for the underutilization of the *Big* accelerators comes from the more similar execution time of both models, which results in little time where one accelerator is idle while the others are still busy. This can be seen more clearly in Fig. 10. On the left, during the first request, accelerators 0 and 2 are idle during the entire duration of the request, and there is a significant amount of time accelerator 3 is still executing the VGG model (orange) while accelerator 1 has already finished the execution of the YOLO model (blue). On the right, because the VGG model is executed faster (as this model is mapped more efficiently onto the *Big* accelerators), there is less time one accelerator is still busy, and the other one idle (as defined in Equation (2), the time where no request is available is not taken into account as idle time for the utilization calculation). The impact of our work can be seen more clearly in the case with 4 *Small* accelerators, but it is still able to increase both utilization and throughput for the case with 2 *Big* accelerators.

B. *Autotuning overhead*

Executing autotuning processes like AutoTVM introduces a one-time compilation overhead because the tuning process evaluates each possible candidate on the HW to find the one that minimizes the latency. Given our new autotuning parameters defined in Section IV-A (type of partition, row or channel, and number of partitions, limited to *M*), the maximum number of possible partitioning candidates that need to be measured on the HW can be calculated. For a given DNN workload *G* containing *L* layers, for an SoC with *M* accelerators, the maximum number of candidates to measure is defined by Equation (3).

$$candidates = L \times (2 \times M - 1) \quad (3)$$

For the VGG model (*L*=16, only convolutions) on a SoC with *M*=4, this yields 112 possible candidates. In our experiments running AutoTVM with our modifications on a 12-core i7 CPU (3.20GHz) server with 32 GB of RAM, the entire process required approximately 45 minutes. Given the performance improvements demonstrated in Section V-A, this one-time cost is well justified.

VI. Conclusion

This paper first explored the HW design space for a multi-DNN multi-accelerator SoC system and proposed an RTOS configuration to achieve automatic model parallelism. From the FPGA implementation results, we extracted insights and identified that model parallelism underutilized the available accelerators. In order to improve this, we proposed a joint compilation and runtime methodology to first find the optimal

partitioning points for each tensor operation during compile time using autotuning, and then dynamically distribute these partitions across the available accelerators using a lightweight scheduler. We explored different decision techniques for the proposed dynamic allocator scheduler, and demonstrated that we can achieve a 29 % and 40 % improvement in mean accelerator utilization and throughput, respectively, over the model parallelism baseline. When compared against a round-robin distribution of partitions, our dynamic allocator is able to achieve a 10 % and 9 % increase in mean utilization and throughput, respectively.

Although this work focused mostly on the object detection aspect of an autonomous driving application, a future extension could evaluate the proposed methodology on a full-scale autonomous driving application, where additional DNN workloads need to be orchestrated for other tasks like object tracking and trajectory prediction. Additionally, only the mean of the distribution of measured latencies was used to guide the runtime allocation of shards. Other metrics of the distribution (e.g., different quantiles) may present different trade-offs and are worth analyzing.

REFERENCES

[1] A. Mousavian, D. Anguelov, J. Flynn, and J. Kosecka, "3d bounding box estimation using deep learning and geometry," CoRR, vol. abs/1612.00496, 2016. [Online]. Available: http://arxiv.org/abs/1612.00496

[2] C.-J. Wu, D. Brooks, K. Chen, D. Chen et al., "Machine learning at facebook: Understanding inference at the edge," in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019, pp. 331–344.

[3] C. Wang, Y. Bai, and D. Sun, "CD-MSA: Cooperative and Deadline-Aware Scheduling for Efficient Multi-Tenancy on DNN Accelerators,"IEEE Transactions on Parallel and Distributed Systems, vol. 34, no. 7,pp. 2091–2106, Jul. 2023.

[4] C. Gao, Y. Wang, C. Liu, M. Wang et al., "Layer-Puzzle: Allocating and Scheduling Multi-task on Multi-core NPUs by Using Layer Heterogeneity," in 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). Antwerp, Belgium: IEEE, Apr. 2023, pp. 1–6.

[5] Z. Zhao, N. Ling, N. Guan, and G. Xing, "Miriam: Exploiting elastic kernels for real-time multi-dnn inference on edge gpu," in Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems, ser. SenSys '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 97–110. [Online]. Available: https://doi.org/10.1145/3625687.3625789

[6] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb et al., "Chipyard: Integrated design, simulation, and implementation framework for custom socs," IEEE Micro, vol. 40, no. 4, pp. 10–21, 2020.

[7] H. Genc, S. Kim, A. Amid, A. Haj-Ali et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in Proceedings of the 58th Annual Design Automation Conference (DAC), 2021.

[8] Linux foundation. zephyr [Online]. Available: https://www.zephyrproject.org/[Accessed: 16 April 2025].

[9] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang et al., "Learning to optimize tensor programs," CoRR, vol. abs/1805.08166, 2018. [Online]. Available: http://arxiv.org/abs/1805.08166

[10] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, ser. ISCA '20. Virtual Event: IEEE Press, Sep. 2020, pp. 940–953.

[11] Y. H. Oh, S. Kim, Y. Jin, S. Son et al., "Layerweaver: Maximizing Resource Utilization of Neural Processing Units via Layer-Wise Scheduling," in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Feb. 2021, pp. 584–597.

[12] Y. Choi and M. Rhu, "PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units," in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). San Diego, CA, USA: IEEE, Feb. 2020, pp. 220–233.

[13] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer et al., "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Oct. 2020, pp. 681–697.

[14] H. Kwon, L. Lai, M. Pellauer, T. Krishna et al., "Heterogeneous Dataflow Accelerators for Multi-DNN Workloads," in 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Feb. 2021, pp. 71–83.

[15] S.-C. Kao and T. Krishna, "MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores," in 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA). Seoul, Korea, Republic of: IEEE, Apr. 2022, pp. 814–830.

[16] X. Zhang, C. Hao, P. Zhou, A. Jones, and J. Hu, "H2H: Heterogeneous Model to Heterogeneous System Mapping with Computation and Communication Awareness," Apr. 2022.

[17] S. Karl, A. Symons, N. Fasfous, and M. Verhelst, "Genetic Algorithm-based Framework for Layer-Fused Scheduling of Multiple DNNs on Multi-core Systems," in 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). Antwerp, Belgium: IEEE, Apr. 2023, pp. 1–6.

[18] S. Zheng, S. Chen, and Y. Liang, "Memory and Computation Coordinated Mapping of DNNs onto Complex Heterogeneous SoC," in 2023 60th ACM/IEEE Design Automation Conference (DAC). San Francisco, CA, USA: IEEE, Jul. 2023, pp. 1–6.

[19] J. Yang, H. Zheng, and A. Louri, "Versa-DNN: A Versatile Architecture Enabling High-Performance and Energy-Efficient Multi-DNN Acceleration," IEEE Transactions on Parallel and Distributed Systems, vol. 35, no. 2, pp. 349–361, Feb. 2024.

[20] M. Amine Hamdi, F. Daghero, G. Maria Sarda, J. Van Delm et al., "Match: Model-aware tvm-based compilation for heterogeneous edge devices," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 44, no. 10, pp. 3844–3857, 2025.

[21] M. Kirschner, F. Peccia, F. Th¨ommes, V. P. Betancourt et al., "Characterization of execution time variability in fpga-based ai-accelerators," in 2023 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), 2023, pp. 1–8.

[22] F. N. Peccia and O. Bringmann, "Integration of a systolic array based hardware accelerator into a dnn operator auto-tuning framework," in Proceedings of the 2023 Workshop on Compilers, Deployment, and Tooling for Edge AI, ser. CODAI '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 21–26. [Online]. Available: https://doi.org/10.1145/3615338.3618130

[23] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman et al., "Mlperf tiny benchmark," Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, 2021.

[24] F. N. Peccia, S. Pavlitska, T. Fleck, and O. Bringmann, "Efficient edge AI: Deploying convolutional neural networks on fpga with the gemmini accelerator," in 2024 27th Euromicro Conference on Digital System Design (DSD), 2024, pp. 418–426.

[25] E. Baccour, N. Mhaisen, A. A. Abdellatif, A. Erbad et al., "Pervasive AI for IoT applications: A survey on resource-efficient distributed artificial intelligence," IEEE Communications Surveys & Tutorials, vol. 24, no. 4, pp. 2366–2418, 2022.