Generating and Exploiting Deep Learning Variants to Increase Utilization of the Heterogeneous Resources in Autonomous Driving Platforms

Author:
Roger Pujol Torramorell

Supervisor:
Hamid Tabani — Barcelona Supercomputing Center

Co-supervisor:
Francisco J. Cazorla — Barcelona Supercomputing Center

Tutor:
Leonidas Kosmidis — Department of Computer Architecture
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

Advanced Computing Specialization
Master in Innovation and Research in Informatics

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

Computer Architecture - Operating Systems Departament
Barcelona Supercomputing Center

June 24, 2020

Acknowledgements

First, I want to sincerely thank my advisors Hamid Tabani, Francisco J. Cazorla, and Jaume Abella for their guidance and mentoring through the development of this Thesis. I also want to thank the rest of the people of the Computer Architecture and Operating Systems group (CAOS) at the Barcelona Supercomputing Center (BSC). They always offer help when I needed it.

Furthermore, I would like to acknowledge BSC for financially supporting my master studies and also to the following institutions that have partially supported this work: the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the UP2DATE European Union's Horizon 2020 (H2020) research and innovation programme under grant agreement No 871465, the SuPerCom European Research Council (ERC) project under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773), and the HiPEAC Network of Excellence.

Last but not least, I would like to thank my parents, brother, closest friends, and girlfriend for their unconditional support in my studies and life. My most heartfelt thanks for the great confidence you have shown in me.

Abstract

Nowadays, Deep learning-based solutions and, in particular, deep neural networks (DNNs) are getting into several core functionalities in critical real-time embedded systems (CRTES), like those in planes, cars and satellites, from vision-based perception (object detection and object tracking) systems to trajectory planning. As a result, several deep learning instances are running simultaneously at any time on the same computing platform.

However, while modern computing platforms offer a variety of computing elements (e.g., CPUs, GPUs, and specific accelerators) in which those DNN instances can be executed depending on their computational requirements and temporal constraints. Currently, most DNNs are mainly programmed to exploit one particular computing element, regular cores of the GPUs. This lack of variety causes a resource imbalance and under-utilization of the various computing element resources when executing several DNN instances, causing an increase in DNN tasks' execution time requirements.

In this Thesis, (a) we develop different variants (implementation) of well-known DNN libraries used in the Apollo Autonomous Driving software for each of the computing elements of the latest NVIDIA Xavier system-on-chip. Each variant is configured to balance resource requirements and performance: the regular CPU core implementation that can run on 2, 4, and 6 cores (always leaving 2 cores free for other computations); the GPU with regular and Tensor cores variants that can run on 4 or 8 GPU's Stream Multiprocessors (SM); and 1 or 2 NVIDIA's Deep Learning Accelerators (NVDLA); (b) we show that each particular variant/configuration offers different resource utilization/performance point. (c) we show how those heterogeneous computing elements can be exploited by a static scheduler to sustain the execution of multiple and diverse DNN variants on the same platform.

Contents

List of Figures

List of Tables

Acronyms

AD Autonomous Driving.

ADAS Advanced Driver-Assistance System.

AI Artificial Intelligence.

AV Autonomous Vehicle.

CBS Constant Bandwith Server.

CCPLEX CPU Complex.

CE Computing Element.

CRTES Critical Real-Time Embedded System.

DL Deep Learning.

DNN Deep Neural Network.

EDF Earliest Deadline First.

GEMM GEneral Matrix Multiplication.

LiDAR Light Detection And Ranging.

LP Linear Programming.

MPS Multiprocessing Service.

NVDLA NVIDIA Deep Learning Accelerator.

ReLU Rectified Linear Unit.

RNN Recurrent Neural Network.

SAE Society of Automotive Engineers.

SM Streaming Multiprocessors.

SoC System on Chip.

TLP Thread-Level Parallelism.

TOPS Tera-Operations Per Second.

Chapter 1

Introduction

Computing systems require functional correctness so that the outputs provided correspond to the system specification. A subset of the computing systems, known as real-time systems, also needs timing correctness. Timing correctness refers to the execution of the corresponding functionalities before specific deadlines. For instance, a video decoding processor has to process each frame at a given speed; otherwise, the user might see distortions on the result. Similarly, the braking system of a car has to work with a tight deadline; if not, it might respond too late and cause a fatal accident. The difference between these two examples is the criticality of the result; if the video has a failure, it at most results in an inconvenience to the user, whereas if the brakes fail (either functional error or missed deadline), it can result in a severe fatality. Therefore, the systems with critical functionalities are called Critical Real-Time Embedded Systems (CRTES).

Nowadays, most cars include some systems to help drivers to drive better and in a safer way. These cover basic things like the Anti-Lock Braking System (ABS) to more complex things like automated parking or lane following assistance. As stated previously, these systems help the driver with the control, but ultimately the driver always has to make the decisions; these are called Advanced Driving-Assistance Systems (ADAS). Autonomous Driving (AD) is even more ambitious; it seeks to relieve the driver from decision tasks and, ultimately, from any driving responsibility. AD requires more sophisticated algorithms due to its complexity; for example, understanding a real-world environment through sensors and acting according to it, this has to be done with Artificial Intelligence (AI). For the most complex tasks, like object detection (computer vision), it is needed the most powerful tool from AI, which is Deep Learning.

1.1 Motivation

Although deep learning algorithms have been around for more than three decades, in the past few years, they revolutionized the computer science world due to the impressive accuracy improvement

of DL over the traditional algorithms based on shallow learning [28, 33, 48]. This improvement, caused it to spread rapidly to the mainstream computing domain in a wide variety of areas ranging from pattern recognition to natural language processing, data science to new a few. CRTES are not exception to other domains as they also follow this trend with DL-based algorithms used in areas like robotics and AD.

In terms of the latter, AD software solutions heavily build on DNN-based solutions, with hundreds of thousands of vehicles (only Tesla has sold more than 825,970 autopilot-capable vehicles [43]). These vehicles are equipped with the DNN-based partial automation (level 2 and level 3 specified by Society of Automotive Engineers [31]) and most car/truck manufacturers announcing their roadmap towards (DNN-based) highly- and fully-automation in the near future (levels 4 and 5 according to SAE International [31]). In fact, DL has emerged as the reference algorithmic solution for the realization of several functionalities included in AD such as computer vision (e.g., object detection and object tracking), path planning, driver-monitoring systems, gesture-activated AI assistants, and voice-based command and control [40].

On the other hand, increased accuracy in DL algorithms came with a substantial increase in the required computational demands. At the hardware level, high-performance platforms is required to satisfy the massive computation needs of DL workloads [30, 46, 47], with GPUs at the forefront of those solutions and lately being extensively evaluated by OEMs and top-tier companies in the automotive domain [36]. At the software level, highly optimized tools, frameworks, and low-level libraries are deployed to significantly improve the hardware utilization and facilitate the software development process [1, 17, 18, 35].

Despite these efforts, autonomous driving – the target domain of this Thesis – still challenges the computational capabilities of existing solutions. Just for ADAS, which arguably require much lower performance than fully AD systems, ARM projects a 100x increase in computation needs from 2016 to 2024 [7]. Capturing these demands requires a computation capacity of tens of tera-operations per second (TOPS), which can theoretically be achieved by having a variety of accelerators (specialized computing elements such as deep learning accelerators) in automotive system-on-chips (SoC) and high-end GPU platforms with features such as Tensor cores [41] to accelerate deep learning operations.

1.2 Problem Statement

Most of the AD systems consist of a set of sophisticated and highly-coupled modules. Each of the modules implements a specific functionality of autonomous vehicles (AV). For example, Apollo[1] AD system [9] has complex modules such as *Perception* to identify the surrounding area around the

[1] *Apollo* is an industrial and practically implemented project with more than 130 industrial partners, most of them top-tier AI and tech companies and car manufacturers.

vehicle or *Planning* to plan the trajectory that the autonomous car has to take. Due to extensive use of deep learning solutions in various of these modules for different AD functionalities, and according to our observations in an industry-level AD system, several DL instances run simultaneously on the underlying SoC.

For instance, while the object detector module analyzes the current time-frame, the tracking module processes the objects recognized in previous time-frames and matches them with the objects in the current time-frame. At the same time, the planning module calculates the best path trajectory. In more sophisticated systems, each module can require several DNN instances to implement the required functionality, and the module can be instantiated several times, one or several instances per each input sensor, e.g., cameras, LiDARs[2], and radars. However, while modern heterogeneous platforms offer several types of (accelerating) computing elements (CE), current DL implementations mostly exploit the traditional elements, which at the time of writing this Thesis are the regular (graphic) computing cores in a GPU. Regardless of the specific CE, the fact that just one type of CE is used, heavily under-exploits current heterogeneous SoC computation capacity.

The ability to run DL-based *variants*, each using different CEs, would improve timing and throughput, and would pay off the extra effort required to implement those different variants. Vendors such as NVIDIA designed sophisticated SoCs in which they integrated powerful deep learning accelerators (e.g., NVDLA) designed and specialized for DL workloads, high-end GPUs and state-of-the-art multicore CPUs in a single chip. The GPUs are featuring new Tensor cores for DL inference, which are capable of different data type operations, from `int8` to `fp16` and `fp32`, and provide massive and flexible computation capacity.

1.3 Contributions

In this Thesis, we aim at providing efficient computation solutions for AD frameworks, focusing on one of the most sophisticated and industry-level open-source AD framework, the Apollo AD framework [9], when deployed on high-performance SoCs, such as the latest and state-of-the-art SoC with multiple heterogeneous computing elements that NVIDIA designed for high-level automation, the Xavier SoC [42]. Our main contributions are the followings:

1. We exploit the use of DL-based algorithms in Apollo AD system. We perform an analysis of the number of DL instances that can be running during Apollo's execution. We show that, at least, seven instances can be active at the same time, and each instance comes with different computation needs and are subject to different time constraints. Based on observed indicators, we conclude that the number of DL instances is expected to increase in future AD systems.

[2]LiDAR, which stands for Light Detection and Ranging, uses laser pulses to build a 3D model of the environment around the car. Essentially, they help autonomous vehicles "see" other objects, like cars, pedestrians, and cyclists.

2. We implement distinct variants of different DL libraries so that each of them can be executed on different CEs in the NVIDIA's Xavier SoC: CPU, GPU regular cores, GPU Tensor cores, and NVIDIA's specialized deep learning accelerator (NVDLA). Our variants are programmed such that they can be executed under different thread-level parallelism (TLP) degrees, which allows more flexibility when exploiting the existing CEs.

3. We make an in-depth analysis of the implications of running the different variants of the DL libraries on the NVIDIA's Xavier SoC and show that each implementation/TLP-setup offers a different design point in terms of used resources and performance.

4. We model a multicore cyclic executive scheduler as a linear programming (LP) problem to assess the increase in guaranteed performance enabled by heterogeneous resources. We show how the variable execution requirements exhibited by tasks on the different heterogeneous computing elements can be exploited to increase the number of advanced neural network-based functionalities on the same SoC, with clear advantages in terms of reducing procurement costs and reliability concerns.

1.4 Thesis Organization

The rest of this Thesis is structured as follows: Chapter 2 introduces DNNs, Apollo, and also presents the main details of our target platform, the NVIDIA's Xavier SoC. Chapter 3 analyzes the DNNs used in Apollo and the projection in the use of DNN in CRTES. Chapter 4 details the different implementations we developed for different DL libraries and their resource usage and performance in the Xavier SoC. Chapter 5 shows how scheduling can benefit from these TLP-configurable implementations to increase system load or adapt DL execution time requirements to its allocated time budget. Chapter 6 presents the most relevant related works. Chapter 7 summarizes the main conclusions of this work, and, finally, Chapter 8 reviews the published part of this Thesis.

Chapter 2

Background

In this chapter, we first present a brief introduction to deep neural networks and their use in autonomous driving systems. Then, we introduce Apollo autonomous driving software as our reference framework in this Thesis. Lastly, we provide more detail on our reference state-of-the-art hardware platform, NVIDIA AGX Xavier.

2.1 Introduction to Deep Neural Networks

Deep learning (DL) methods are part of the machine learning methods, which are mainly inspired by the structure and function of the human brain, and are called artificial neural networks.

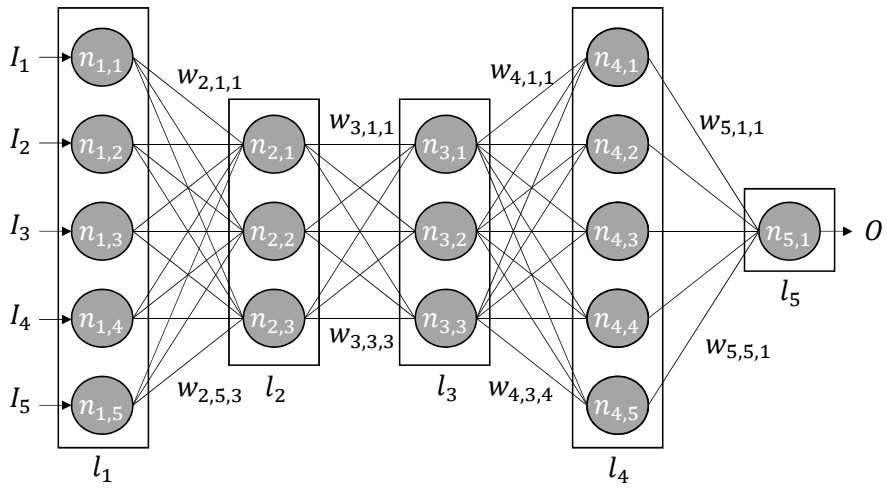


Figure 2.1: Representation of a Neural Network

Deep Neural Networks (DNNs) follow a well-known DL architecture for those networks, which has an input layer, an output layer and, at least, one hidden layer in between. Figure 2.1 is a

representation of a Neural Network where it shows the input layer l_1, the hidden layers (l_2, l_3 and l_4), and lastly, the output layer l_5. Each of the nodes in Figure 2.1 are called a "neuron". The value of each neuron (node) is computed as presented in Equation 2.1, where $n_{i,j}$ indicates a neuron in layer i position j, $w_{i,k,j}$ is the weight of the network between $n_{i-1,k}$ and $n_{i,j}$, $max(l_i)$ indicates the last position of layer l_i and $b_{i,j}$ is a bias of $n_{i,j}$.

$$n_{i,j} = b_{i,j} + \sum_{k=1}^{max(l_{i-1})} n_{i-1,k} \times w_{i,k,j} \tag{2.1}$$

In Figure 2.1 as a particular example, each neuron is connected to all the neurons from the previous layer, however, this might be different in other types of layers. Each layer performs specific types of sorting and ordering in a process that some refer to as "feature hierarchy" [27]. These networks have to go through a training process with labeled data to tune the appropriate weights in order to work properly.

One of the key uses of these sophisticated neural networks is dealing with unlabeled or unstructured data. DL, used to describe these (deep) neural networks, uses aspects of artificial intelligence which seek to classify and order information in ways that go beyond simple input/output protocols. DNNs provide high accuracy solutions in several domains including computer vision for functions such as image classification and object detection. Nowadays, DNNs are widely used in a variety of areas, and CRTES are not an exception to this.

Recurrent neural networks (RNNs) are another class of artificial neural networks (which have an internal state with information from previous executions) that are very successful for history-based workloads such as speech recognition, path planning, and machine translation. RNNs are used as the state-of-the-art approach for path planning in industrial autonomous driving systems.

2.2 Apollo Autonomous Driving Software

We study an industrial autonomous driving software to illustrate the benefits of our DNN variant-based approach. In particular, we use *Apollo* [9], an industrial and practically implemented project with more than 130 industrial partners, most of them top-tier AI and tech companies and car manufacturers. Apollo is arguably the most sophisticated open-source autonomous driving framework available implementing the entire AD software stack and already deployed on a variety of vehicles (such as Robo-taxies, trucks and passenger cars). It is also used as a representative case study in research [3, 4, 49, 53]. Apollo has been widely used recently to help with the COVID-19 situation in China [10]. Apollo supports state-of-the-art hardware such as latest LiDARs and cameras from Velodyne and other vendors, as well as GPU acceleration.

In this Thesis, we study Apollo v3.0 which comprises 8 main modules and several sub-modules, as

shown in Figure 2.2. These software modules operate in a software-pipelined fashion and work in general at time-frame level:

M_0 *Speech recognizer* processes the voice-based commands from the driver/passengers and transmit them to the control unit.

M_1 *Perception* identifies the surrounding area around the autonomous car.

$M_1.d$ The *detection* submodule is in charge of detecting obstacles and objects from different sensors.

$M_1.f$ *fusion* takes the results of all detected objects from different sensors and combines them by a sensor fusion algorithm.

$M_1.t$ *tracker* follows the detected objects and matches them with the objects detected in previous frames.

M_2 The *Planning* plans the spatio-temporal trajectory for the vehicle to take.

M_3 *Localization* leverages information received from different input sensors to estimate the precise position of the vehicle.

M_4 The *Map* provides ad-hoc structured information regarding the roads.

M_5 *Prediction* anticipates the future motion trajectories of perceived obstacles/objects.

M_6 *Control* generates control commands such as accelerating/braking and steering.

M_7 *CAN Bus* passes all the control commands to the vehicle hardware and provides information back to the autonomous system.

In this Thesis, we used Apollo default input datasets which are real data from sensors of an AD car collected and provided by the Apollo team. Besides, we used similar neural network architectures that Apollo employs in its different stages.

2.3 The Jetson AGX Xavier and it's Main Computing Elements

NVIDIA has recently introduced the Jetson AGX Xavier SoC [42] as the cornerstone of its automotive platforms. Xavier delivers over 30 TOPS for DL applications while consuming less than 30 Watts. Xavier is, to date, the largest SoC ever built with more than 7 billion transistors.

Xavier comprises four main computing elements (CEs) capable of processing deep learning workloads: traditional CPU cores, GPU regular cores (known as CUDA cores), GPU Tensor cores, and the NVDLAs. The Xavier SoC also integrates several other accelerators such as vision accelerator, video encoder, etc. However, these accelerators cannot be used for DNN/RNN inference due to

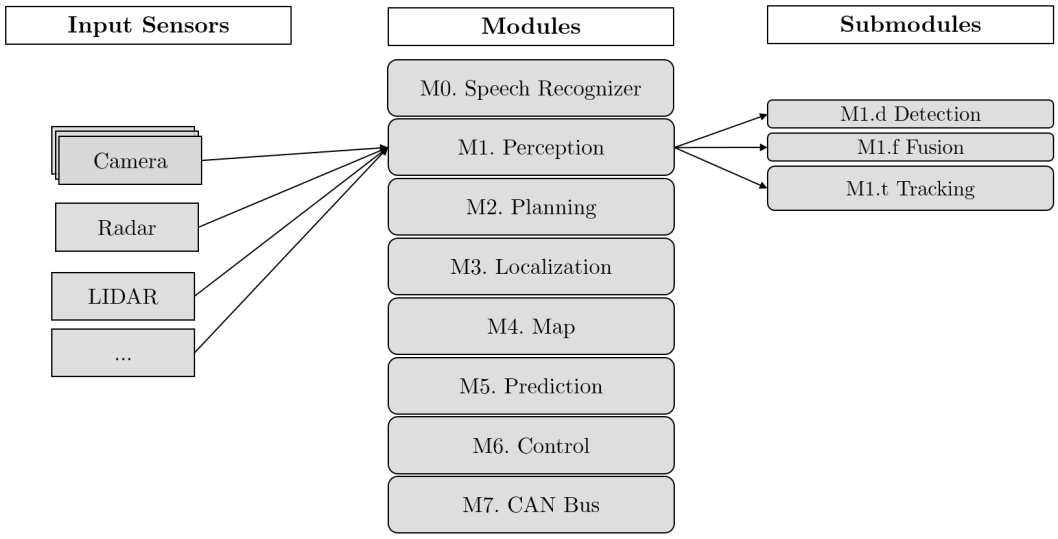


Figure 2.2: Modules, sub-modules and input sensors of Apollo.

their limited programmability. Therefore, in this Thesis, we focus only on CPU, GPU (regular and Tensor) cores, and the NVIDIA Deep Learning Accelerators (NVDLA).

1. CPU cores. The CPU complex (CCPLEX) comprises eight homogeneous Carmel ARMv8.2 processors. Each core has its private instruction and data caches. In each cluster of two cores (4 clusters in total), an L2 cache is shared between both cores. An L3 cache is shared between all CPU cores.

2. GPU regular and Tensor cores. The Volta GPU microarchitecture comprises 512 regular cores (CUDA cores in NVIDIA terminology) and 64 Tensor cores. The GPU is structured in 8 Streaming Multiprocessors (SMs) each containing 64 regular and 8 Tensor cores.

   Tensor cores [41] accelerate large matrix operations, which are at the heart of many AI functions. While each regular core can perform up to one single-precision multiply-accumulate operation per 1 GPU clock, each Tensor core can perform one matrix multiply-accumulate operation per 1 GPU clock. The Tensor core can multiply two `fp16` 4×4 matrices and adds the multiplication product `fp32` matrix to the accumulator, which is also a `fp32` 4×4 matrix.

   In each SM, threads can use either the regular cores or the Tensor cores. Hence, at most, 512 regular or 64 Tensor cores can be used in parallel.

3. NVDLA provides a flexible, robust inference acceleration solution. Xavier SoC has two NVDLAs which can be configured to run deep learning workloads. This is the very first work that considers NVDLAs in the real-time domain to the best of our knowledge.

Overall, the NVIDIA Xavier SoC offers four different CEs (counting GPU Regular and Tensor cores as separate CEs), as can be seen in Figure 2.3, that we use to illustrate our proposal's benefits. In

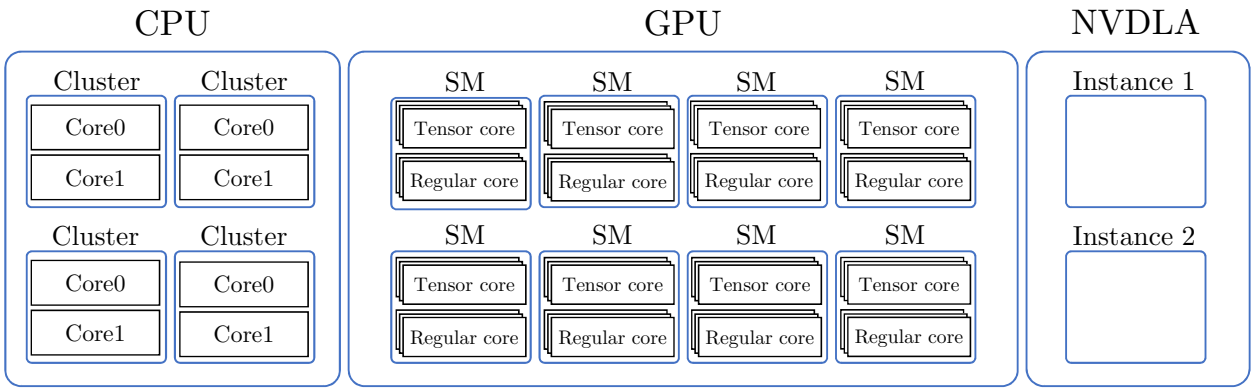


Figure 2.3: CEs in the Xavier SoC

Table 2.1: Configurations which we exploit Xavier's CEs

CPU	GPU	NVDLA
1 cluster	4 SMs	1 instance
2 clusters	8 SMs	2 instances
3 clusters		

order to reduce the exploration space, we only consider the configurations in which we explore each CE, as shown in Table 2.1.

- At the CCPLEX level, we restrict our approach to core clusters. Also, since all DNN/RNN instances using the GPU or the NVDLA are initiated from the CPU, we reserve 2 CPUs for them. Overall, at the CPU level, a DNN/RNN instance can use 2, 4, or 6 of the remaining cores.
- At the GPU level, we set up a minimum granularity of 4 SMs. Hence, a DNN/RNN task can use either 4/8 SMs to exploit 256/512 GPU regular or 32/64 Tensor cores, respectively.
- Each task can use one or two NVDLA accelerators.

To sum up, the possible hardware configurations for a task are: 2, 4, or 6 CPU cores; 256 regular or 32 Tensor cores, 512 regular or 64 Tensor cores; and 1 or 2 NVDLA instances.

Chapter 3

Analysis of the Deep Learning elements in Apollo

In the literature, we can find a wide range of DNNs and other DL algorithms. In this Thesis, we focus on those generally used for DL-based solutions in AD systems. In this line, Table 3.1 shows different types of (state-of-the-art) neural networks widely used in key domains for AD functionalities. We can observe that three modules (M_0, M_1, and M_5) and 2 sub-modules ($M1.d$ and $M1.t$) use neural networks, DNNs and RNNs in particular. Table 3.1 shows:

Table 3.1: Neural networks used in different modules of the Apollo autonomous driving system.

	Deep Learning Software	**Description**
M0. Speech Recognition	Voice Command and Control	A DNN-based accurate speech recognition application to process speech commands
M1. Perception	Camera Object Detection	A DNN-based algorithm to identify objects and traffic signals from camera sensors
	LiDAR object Detection	A DNN-based algorithm to identify objects from LiDAR sensors
	Object Tracker	A DNN-based algorithm to track identified objects in consecutive frames
M5. Prediction	Lane sequences (RNN1)	A RNN for lane sequence-based prediction
	Obstacle status (RNN2)	A RNN for obstacle status
	A RNN using the output produced by RNN1 and RNN2	A RNN to compute the probability of each lane sequence based on RNN1 and RNN2

- The perception module, M_1, relies on different DNNs for detecting ($M1.d$) obstacles and objects from different sensors. The results of all detected objects are fused by a sensor fusion algorithm ($M1.f$) that does not use DNNs. As the last step, an object tracker ($M1.t$) deploys

a DNN to track and follow detected objects.

- The prediction module, M_5, uses RNNs to build a model to predict the target lane that the vehicle should take. One RNN model is for lane sequences and another RNN model for the associated object states. The concatenation of these two RNNs is fed into another neural network to estimate the probability for each lane sequence. Interestingly, the modules using neural networks, *Perception* and *Prediction*, are the most compute-intensive modules: they consume more than 70% of the time Apollo uses to process each frame.
- Some AD systems suggest deploying AI-assistant applications to be implemented inside the cabin, which are all based on neural networks [40]. Such applications are proposed for driver-monitoring, and command and control using gestures and voice.

Table 3.2 summarizes the modules using DNNs and RNNs.

Table 3.2: Modules of Apollo using DNNs (⊛) and RNNs (◎).

	M1 (Perception)			**Other Modules**						
Input	M1.d	M1.f	M1.t	M0	M2	M3	M4	M5	M6	M7
Camera	⊛									
LiDAR	⊛		⊛	⊛				◎ ◎ ◎		
Radar										

3.1 Real Execution Trace

Figure 3.1 shows part of a trace collected from the actual execution of Apollo when all DNN and RNN instances in the different modules/sub-modules use the regular GPU cores in the Jetson AGX Xavier. Each rectangle shows the span of execution of each DNN/RNN instance, i.e., since it starts running (i.e., it is executable) until its execution finishes. As Figure 3.1 shows, at a given time t_i, several instances of different neural networks are executed concurrently. Also, each particular DNN/RNN has diverse computational requirements with more than 12× variability among them: the voice command runs for 4.5ms, whereas the different RNNs in the prediction module range from 48.61ms to 61.06ms; finally in the perception module DNN instances span goes from 38.96ms to 50.18ms. It is also the case that temporal constraints vary up to 10x across DNN/RNN instances. This variance comes from the fact that the rate at which frames arrive across different input sensors can vary from 10ms for Radars to 100ms for LiDARs.

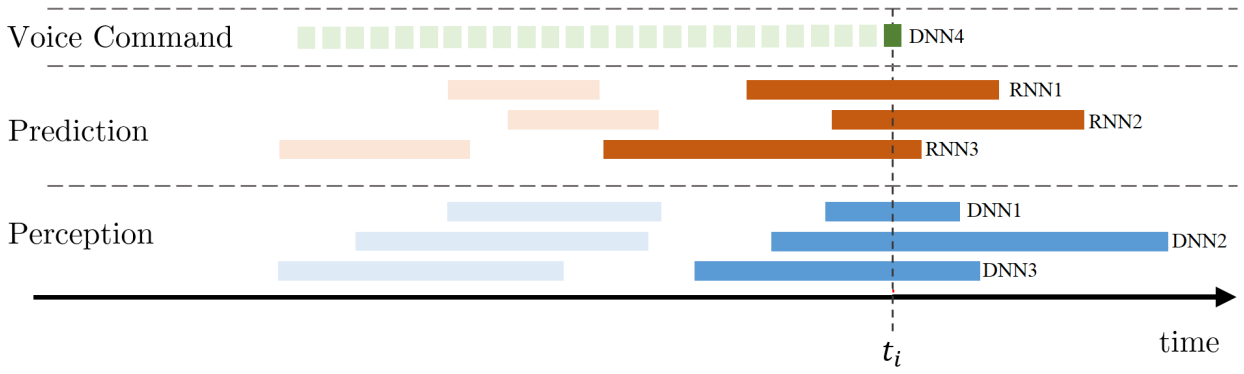


Figure 3.1: Concurrent execution of different modules or instances of a module in Apollo, an industrial autonomous driving software.

3.2 DNN instances

Current trends in the sophisticated AD systems show that the number of concurrent neural network instances can easily reach dozens. The main reasons for such an increase are the followings:

- *More input sensors.* Moving towards fully autonomous driving (Level 5 [31]) naturally requires increasing the number of sensors to cover the car's surrounding more accurately. Today, some of the AD systems, which are still far from a Level 5 system, use more than 8 cameras and radars (e.g., Telsa [19] uses 8 cameras and 12 ultrasonic sensors, and NVIDIA autopilot [40] uses 8 high-resolution cameras, 8 radars and optionally up to 3 LiDARs). Therefore, more DNN-based workloads have to be processed, increasing the computation demand significantly.
- *More sophisticated algorithms.* Perception submodules tend to use more sophisticated DNNs, with a larger number of layers and higher computational needs for further improvements in the accuracy of object and obstacle detection, especially in conditions with reduced visibility such as fog night, rain, and snow. The *Prediction* module already uses 3 different neural networks either to achieve higher accuracy or to cover more complex scenarios. Indeed, this type of module usually uses sophisticated neural network architectures [57, 58].
- *More functionalities.* Besides the main functions of an AD system, extra features are introduced to improve driving quality and safety: from gesture detection and speech-based command and control up to driver-monitoring to predict take-over readiness [21].

This trend towards exploiting multiple neural networks running in parallel and the increasing number and type of accelerators we witness in modern GPUs motivate our idea of assessing the benefits of DNN/RNN variants in modern GPUs.

Chapter 4

Deep Neural Network Implementations

In this Thesis, we adapt the implementation of DNN-based Apollo modules in order to make it work at any CE of the Xavier SoC with the best possible performance. To that end, we change the implementation in the baseline source code, which is based on x86 and GPU. Depending on the target CE, we need to use appropriate libraries and re-implement Apollo modules. In general, DL workloads are implemented layer by layer by defining specific functions for each layer. Then, depending on the layer and the highly-optimized low-level target library [51] (e.g., cuBLAS [35]), input data needs to be transformed to match the proper format expected by the low-level library function.

Apollo v3.0 which is used in this Thesis exploits only regular cores in the GPU for DNN inference. The most computationally-intensive part of inference, such as convolution or fully connected layers, can usually be reduced to GEneral Matrix Multiplication (GEMM), which are implemented with cuBLAS.

For completeness, we have performed several experiments comparing the same DNN operations using cuDNN and cuBLAS. Our results show that cuBLAS achieves very competitive results w.r.t. cuDNN. However, note that the main idea of the Thesis, i.e., having diverse DNN implementations, does not depend on the particular library used.

4.1 Specialized per-CE libraries

Table 4.1 presents the optimized libraries that we have used to implement our software. As it can be seen, for each specific CE, we used different libraries. In addition, we modified the baseline code in order to run the optimized code. We exploit CEs by implementing and adapting our target

Table 4.1: Optimized libraries used to implement the Apollo software for each particular CE.

CE	Optimized Libraries
CPU	We used OpenMP [20] to implement all the functions to run on the CPU cores. Our implementation allows fixing the maximum number of cores that can be used.
GPU Regular Cores	The baseline implementation targets regular cores to run the kernels.
GPU Tensor Cores	We used specific libraries and adapted our code to exploit the Tensor cores. Some of our target deep neural networks consist of 100+ layers. The implementations of all the layers had to be modified.
NVDLA	We adapted each neural network configuration to be compatible with TensorRT [37], except the RNNs as they are not supported by NVDLA [38]. We use the TensorRT framework to launch applications on the NVDLAs.

software for each particular CE. To obtain maximum performance benefits we use specific and specialized library variants.

Along with the introduction of Tensor cores, NVIDIA provided some low-level libraries to support their use. For using the NVDLA, the TensorRT [37] library should be used which is a platform developed and provided by NVIDIA for high-performance deep learning inference. TensorRT offers a DL inference optimizer and runtime that can deliver low latency and high-throughput for DL applications.

It is worth mentioning that, to our knowledge, the version of Apollo that we studied in this Thesis does not use TensorRT.

4.2 Implementation for different CEs

We illustrate the required effort to modify all the functions in the source code to run the entire workload on a specific CE, by focusing on a small function performing a matrix multiplication (GEMM) operation without transposing any of the operand matrices. It is worth noting that each of the functions that implement the different layers of the neural networks is functionally different, and therefore, each of them requires different modifications.

We chose GEMM since it is the function that consumes most of the execution time in the DNN inference and is also one of the most parallelizable functions. In this example, the matrix multiplication function builds on the following formulation, in which A, B, and C are matrices, and α and β are floating-point coefficients.

$$C = \alpha A \times B + \beta C \tag{4.1}$$

4.2.1 CPU implementation

Figure 4.1 shows the CPU version of the matrix multiplication operation presented in Equation 4.1. As input parameters the function takes $ALPHA(\alpha)$ and $BETA(\beta)$ as shown in Equation 4.1; M, N, and K that are the dimensions of the matrices; and lda, ldb, and ldc are leading dimensions of matrices A, B, and C respectively. In other words, lda, ldb, and ldc determine the forward movement in memory when it reached the end of a row (in row-major order) or column (in column-major). In fact, these parameters define strides that provide plenty of flexibility to work with smaller tile sizes inside a larger matrix.

In the first loop, lines 4-8, the βC operation is executed according to Equation 4.1. In lines 10-17, the main loops are implemented to perform the matrix operations. The OpenMP pragma at line 9 automatically parallelizes the outer loop so that independent loop iterations can be executed simultaneously.

```
1  void OpenMPgemmNN(int M, int N, int K, float ALPHA, float const *A, int lda, float const *B,
       int ldb, float BETA, float *C, int ldc)
2  {
3      int i, j,k;
4      for(i = 0; i < M; ++i){
5          for(j = 0; j < N; ++j){
6              C[i*ldc + j] *= BETA;
7          }
8      }
9      #pragma omp parallel for
10     for(i = 0; i < M; ++i){
11         for(k = 0; k < K; ++k){
12             register float A_PART = ALPHA*A[i*lda+k];
13             for(j = 0; j < N; ++j){
14                 C[i*ldc+j] += A_PART*B[k*ldb+j];
15             }
16         }
17     }
18 }
```

Figure 4.1: CPU implementation of the reference matrix multiplication (gemm) operation.

4.2.2 GPU regular core implementation

Figure 4.2 shows the implementation for the GPU regular cores, with the function requiring the same parameters as for the CPU version. Also note that in this example, we assume that the matrices are already in the device's memory space.

First, we get the device ID and check whether we have initialized a cuBLAS handle for it. If it is not the case, we create a new one. Once we obtain the handle, we call `cublasSgemm` but with the matrices in reversed order. This is because C/C++ assumes a row-major layout, whereas CUDA

```
void GRCSgemmNN(int M, int N, int K, float ALPHA, float const *A,
int lda, float const *B, int ldb, float BETA, float *C, int ldc)
{
    static int init[16] = {0};              // Vector for initialized handles
    static cublasHandle_t handle[16];       // Vector of actual handles
    int i;
    cudaGetDevice(&i);                      // Get current device
    if(!init[i]) {                          // If not initialized
        cublasCreate(&handle[i]);           // Creates the handle
        init[i] = 1;
    }
    cudaError_t status = cublasSgemm(handle[i],
            CUBLAS_OP_N, CUBLAS_OP_N,       // Select the non-transpose matrices
            N, M, K,                        // Sizes of the matrices
            &ALPHA,
            B, ldb,                         // B and it's leading size
            A, lda,                         // A and it's leading size
            &BETA,
            C, ldc);                        // C and it's leading size
    if (status != cudaSuccess)              // Check if there is any error
        printf("CUDA Error: %s\n",cudaGetErrorString(status));
}
```

Figure 4.2: GPU implementation for regular cores.

assumes a column-major layout, which means that CUDA is reading the matrices in a transposed manner. Then, since everything is transposed, we can reverse the operators:

$$A \times B = C \iff B' \times A' = C'$$

Finally, we check whether cuBLAS triggered any error during the GEMM.

4.2.3 GPU Tensor core implementation

Reprogramming the GPU code to be run on the Tensor cores requires to change the math mode to `CUBLAS_TENSOR_OP_MATH`. Nonetheless, this implementation builds on some preconditions to run on the Tensor cores: K, lda, ldb, and ldc have to be multiple of 8, and N has to be multiple of 4. Figure 4.3 shows the implementation for the GPU Tensor cores.

4.2.4 NVDLA

The steps we have followed to run the neural network workload on the NVDLAs are shown in Figure 4.4. As a first step, the DNN configuration needs to be in the proper format, *prototxt* that is compatible with TensorRT. To that end, we developed a script that goes layer by layer in the configuration file of the neural network and changes its format to *prototxt*.

```
void GTCSgemmNN(int M, int N, int K, float ALPHA, float const *A, int lda, float const *B,
    int ldb, float BETA, float *C, int ldc)
{
    static int init[16] = {0};                  // Vector for initialized handles
    static cublasHandle_t handle[16];           // Vector of actual handles
    int i;
    cudaGetDevice(&i);                          // Get current device
    if(!init[i]) {                              // If not initialized
        cublasCreate(&handle[i]);               // Creates the handle
        init[i] = 1;
    }
    cublasSetMathMode(handle[i], CUBLAS_TENSOR_OP_MATH);     // Set math mode to enable
    Tensor cores
    cudaError_t status = cublasSgemm(handle[i],
            CUBLAS_OP_N, CUBLAS_OP_N,           // Select the non-transpose matrices
            N, M, K,                            // Sizes of the matrices
            &ALPHA,
            B, ldb,                             // B and it's leading size
            A, lda,                             // A and it's leading size
            &BETA,
            C, ldc);                            // C and it's leading size
    if (status != cudaSuccess)                  // Check if there is any error
        printf("CUDA Error: %s\n",cudaGetErrorString(status));
}
```

Figure 4.3: GPU implementation for Tensor cores.

It is worth mentioning that some layers in the original format are translated into several layers in *prototxt*. For instance, a *Convolutional* layer that has *Batch Normalization* and an activation of type *Leaky* is divided into four different layers: a regular convolution, a *Batch Normalization*, a scale, and a *ReLU* (Rectified Linear Unit) with negative slope.

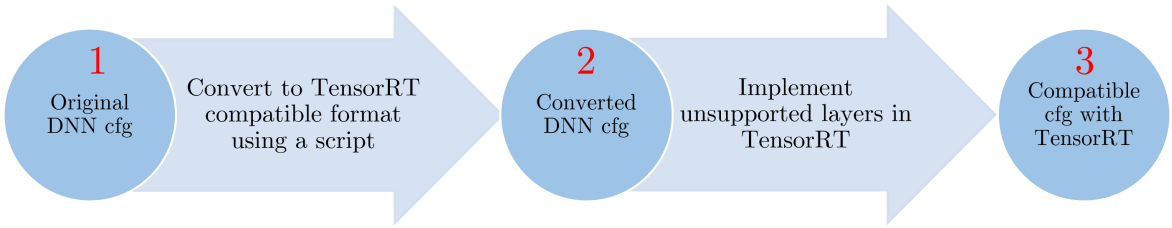


Figure 4.4: The steps required to specify neural network layers in order to be run on the NVDLAs.

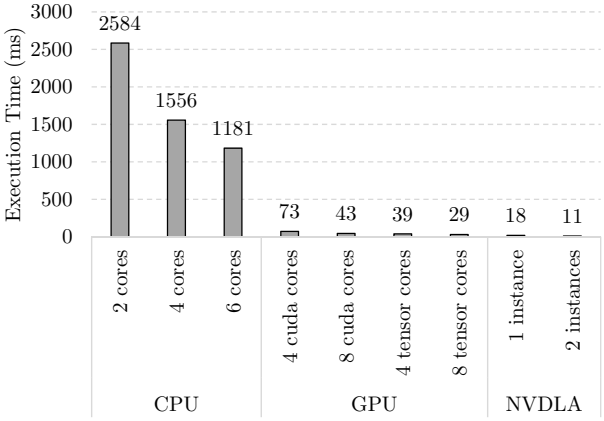
Following this step, we obtain a functional configuration file in the proper format. However, in most cases, some layers are not supported yet by TensorRT. At the time of writing this Thesis, several types of layers, especially for RNNs, are not implemented and cannot be executed on the NVDLA. To overcome this limitation, we adapted some of the available layers using equivalent and currently supported methods. The conflicting layers in our neural networks are *Upsample* layer and *Leaky ReLU* layer. To solve this issue, we implemented the layers according to TensorRT specifications. After these modifications, configuration files are in the correct format required by

TensorRT. Using the generated configurations, TensorRT can parse and build the neural network model and run it in the NVDLAs. Note that TensorRT can allow unimplemented layers to fallback and run on the GPU cores. In our experiments for this Thesis, we avoid any fallback by adapting all the unimplemented layers using other equivalent layers (this process does not include RNN layers that are not supported), and the entire neural network is not supported running on the NVDLA.

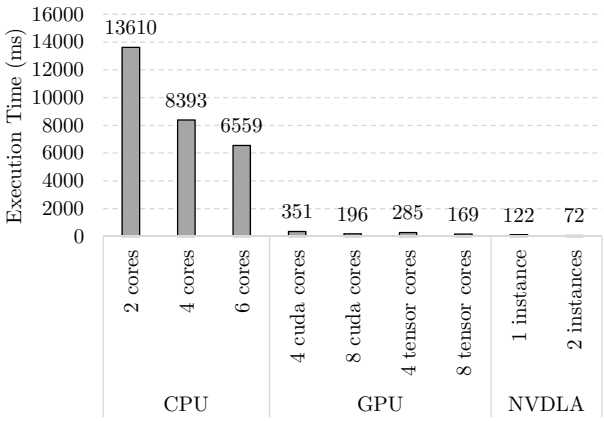
4.3 Timing Analysis Results

We analyze the results obtained for each DNN variant under different CE and TLP levels: CPU cores (2 cores, 4 cores, and 6 cores), GPU regular cores (4 SMs, 8 SMs), GPU Tensor cores (4 SMs, 8 SMs), and NVDLAs (1 or 2 NVDLAs). Note that we always reserve 2 cores of the CPU for managing the operating system tasks and the tasks that are running in the GPU or NVDLA, since their corresponding CPU processes also trigger them.

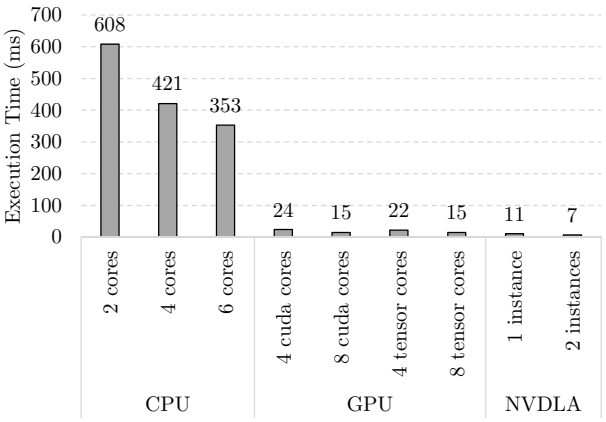
Figure 4.5 shows the timing results of different neural network instances of Apollo running on each CE. Timing characterization has been performed with other DNN/RNN instances run in parallel. While we did not run specific experiments to hit the worst-case timing interference among computing elements, we assume the obtained results also factor in contention effects. We can derive the following conclusions:



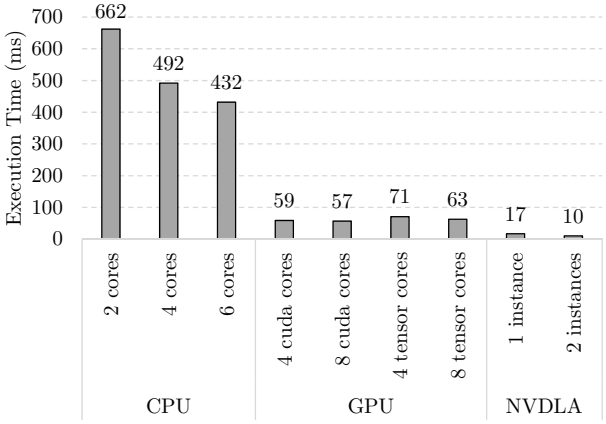
(a) M1.d Normal resolution camera



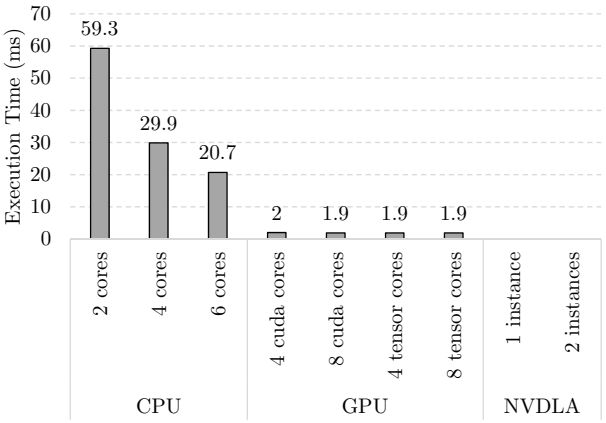
(b) M1.d High resolution camera



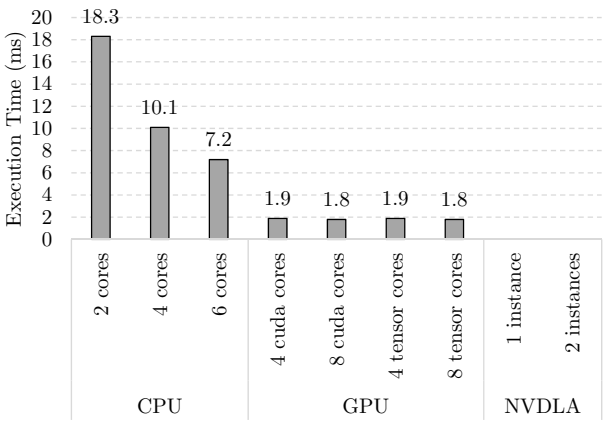
(c) M1.d LiDAR



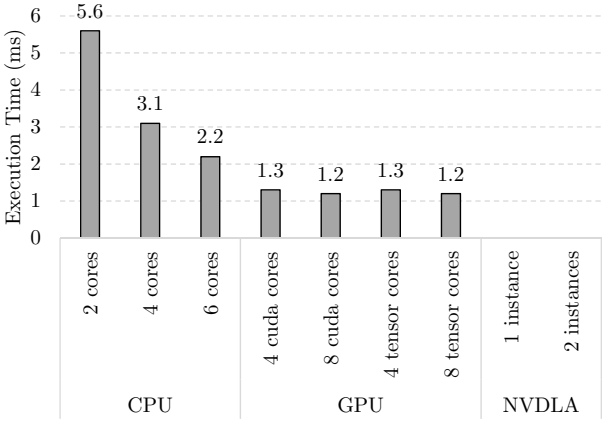
(d) M1.t Object Tracker



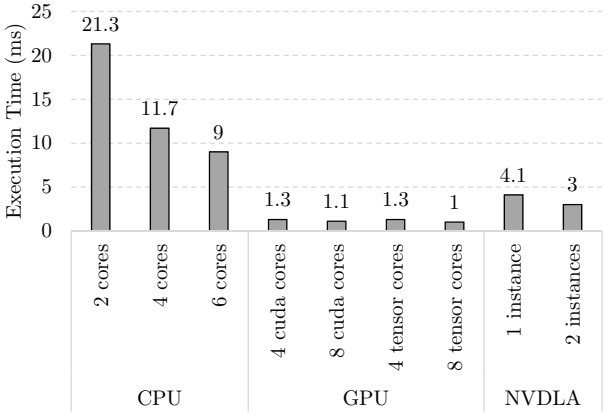
(e) M5 (RNN1)



(f) M5 (RNN2)



(g) M5 (RNN3)



(h) M0 (Speech and comm. control)

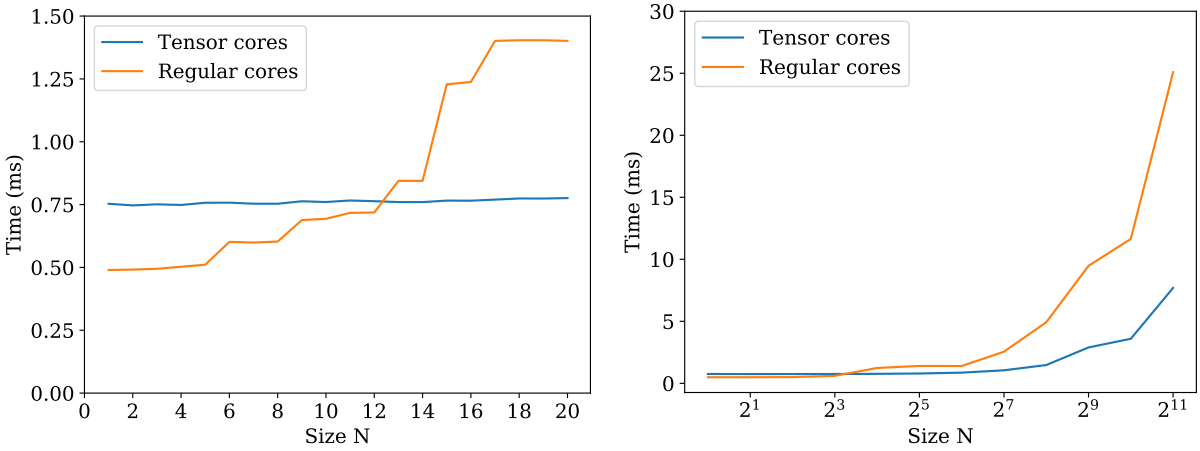
Figure 4.5: Timing results of different Apollo neural network instances on different CEs.

- For the camera object detector ($M1.d$), Figure 4.5a shows that using more CPU cores significantly improves performance. Regarding the timings on GPU CEs, as expected for this workload, Tensor cores provide better performance in comparison to the regular cores, with 8 SMs providing significantly higher performance in comparison to 4 SMs. Also, the NVDLA accelerates this NN, achieving the best performance results.

- Figure 4.5b shows the results for ($M1.d$) under another configuration for the object detector with the same neural network architecture, but with a higher camera resolution. As the results show, we have the same trends as in Figure 4.5a; however, due to the increase in the workload size, execution times increase.

- Figure 4.5c shows the results for the LiDAR object detector, $M_1.d$. Similar to the previous results, GPU CEs provide better performance than CPU cores, though this time, Tensor cores do not result in significant improvements over GPU regular cores. NVDLA again provides the best results. As the workload is smaller than for the camera detector, the times are proportionally reduced.

- Figure 4.5d shows the timing results for object tracker, $M_1.t$. For this specific workload, GPU regular cores provide higher performance than the Tensor cores. After a detailed analysis and designing some experiments, we find out that Tensor cores achieve worse performance than regular cores whenever we run a GEMM of $A_{M\times K}B_{K\times N}$, in which N has a minimal value. More specifically as Figure 4.6a shows for $N \leq 12$ Tensor cores exhibit worse performance than regular cores. This particular case directly affects the Object Tracker (M1.t) since all the GEMMs performed by this DNN have $N = 1$. However, by increasing the value of N, (in this particular experiment, for $N > 12$) Tensor cores provide considerably better performance, see Figure 4.6b.



(a) N ranging from 1 to 20.

(b) N ranging from 1 to 2^{11}.

Figure 4.6: Time spent in a GEMM ($A_{M\times K}B_{K\times N}$) where $M = K = 1024$.

- Figures 4.5e, 4.5f, and 4.5g show the timing results for the three recurrent neural networks

in the prediction module. Due to the nature of the RNNs, these workloads cannot highly benefit from increased parallelization. We still observe a performance improvement using more CPU cores or the GPU, but in the GPUs case, even though they provide an order of magnitude better performance in comparison with CPUs, the utilization of the resources is far from the full potential of the GPU. In terms of the NVDLA, since several key layers of the RNN networks are not implemented in the TensorRT, we are unable to run these workloads on the NVDLA [39].

- Finally, Figure 4.5h shows the speech recognizer module ($M0$), which uses a DNN as discussed in previous sections. This DNN network improves performance by one order of magnitude with GPU cores. Instead, the NVDLA, while better than the CPU cores, performs significantly worse than the GPU cores.

Overall we can see that,

1. For some DNNs the NVDLA variant and the GPUrc (regular cores) and GPUtc (Tensor cores) variants offer comparable performance.
2. In some cases the GPUrc variant provides better results than GPUtc.
3. It is also the case that, in some cases, the performance of CPU is relatively close ($\approx 2x$) of that obtained with the GPUtc and GPUrc variants.
4. Across the different neural networks, we see that the CPU time requirements for some of them (e.g. 4.5e, 4.5f, 4.5g, and 4.5g) is comparable to that required by others in the GPU and NVDLA (e.g. 4.5a, 4.5b, 4.5c, and 4.5d).

This observations and obtained results makes it worth exploiting all CEs.

4.4 Other Considerations

TLP controllability. Fully exploiting variants requires exercising control on TLP as provided by NVIDIA's MPS (Multiprocessing Service). This control allows multiple kernels from different processes to be executed concurrently in the GPU while limiting their resource usage, i.e., how many SMs each kernel uses. The use of MPS has been shown to provide positive results in real-time systems [55], which paves the way for its ubiquitous adoption in all GPUs. Furthermore, MPS only requires driver updates. As this feature is not present in the Xavier SoC, we emulate its effect in our experiments by executing the GPU tasks in isolation and using Xavier's capability to enable only a certain number of SMs in the GPU.

Contention effects on timing behavior is a widely studied topic in the real-time community mainly for CPUs, with few proposed techniques for GPUs [16] to reduce contention bounds. Contention

bounding techniques, e.g., [23, 34] produce a factor Δ_{cont} to be added on top of the in-isolation timing estimates. In this Thesis, we assume that the observed execution time factor in relevant contention effects.

Accuracy. Different implementations may use different standards for floating-point (FP) number representation (e.g., 16-bit or 32-bit representations), different FP operations, or, at least, different FP operation orders. Due to rounding effects, this may lead to slightly different numerical results, whose impact on the system-level functionality needs to be assessed. However, functional results (i.e., objects detected, driving decisions, etc.) match since those tiny numerical variations has no impact in the semantics of the framework. For instance, whether the probability of recognizing an object varies by ±0.1% makes no practical difference in general (e.g., 90.7% vs. 90.8%). Hence, despite the different implementations across CEs, the results of all implementations match functionally.

Multi-CE variants. In our current implementation, each neural network instance exploits a single CE. As future work, we consider adding a multi-CE capability so that a single instance can exploit several CEs, e.g., 4 cores, 1 SM, and 1 NVDLA. While this offers more flexibility than our current single-CE per neural network instance approach, it already shows significant improvements over the baseline in which all instances use the same CE.

Chapter 5

Exploiting Diversity to Increase Schedulability

With platforms supporting 'diverse' computing elements and TLP degrees, the timing behavior of an application is inherently dependent on the deployed configuration. Applications exhibit different execution time bounds depending on the actual CE in which they are mapped. Thus, the overall mapping strategy is fundamental to determine the schedulability of a given set of applications as a whole. The identification of optimal mapping strategy is not a specific requirement for heterogeneous platforms [11], but is a well-studied problem at the basis of several scheduling approaches for homogeneous systems, from partitioned to cyclic-executive scheduling approaches (e.g., [22,29]). Computing an optimal partitioning is NP-hard in the general case: depending on the complexity of the problem instance, provided solutions range from exact optimization frameworks to heuristic-based approaches.

In this Thesis, we are interested in evaluating the benefits of system schedulability, that can be appreciated with execution platforms supporting diverse CE/TLP configurations. As a common characteristic, different DNN instances realizing AD framework functionalities can be modeled as *recurring applications* that run periodically according to a given frame rate. The frame rate depends on the frequency with which the inputs must be processed.

Static programming or cyclical-executive approaches are particularly suitable for these types of systems: despite their known limitations in terms of flexibility and scalability, they are relatively easy to implement and predictable, even in multi-cores. For this reasons, cyclic-executive is still widely adopted in the critical embedded real-time system domains, and is at the basis of standard frameworks (e.g., AUTOSAR [8], ARINC [6]) in critical embedded real-time system domains.

A static schedule results in the repeated execution of a sequence of intervals or frames. Tasks associated with a frame must execute and complete within that frame (i.e., performance guarantees are enforced at each frame boundary). A sequence of frames is then periodically repeated as part

of a major frame, corresponding to the hyper-period. The repetitive behavior of the diverse DNN instances (and the relative independence between them) is naturally modeled with a static schedule. Constructing a schedule for a cyclic executive involves finding a task-to-core mapping that allows all tasks to be completed within their frame (or, reciprocally, that the cumulative utilization of all tasks in a frame does not exceed 1).

While there exist specific rules to define appropriate frame number and size, the schedulability of cyclic executive systems reduces to showing that all computations have completed within the frame. A common approach to construct a valid static schedules consists in formulating the scheduling problem as a *linear programming* (LP) model.

In the scope of our evaluation, we model the problem of scheduling a heterogeneous workload of several diverse DNN instances as a cyclic executive system.

We exploit a LP-based representation of the problem to assess the increase in schedulability that can arise when multiple CE/TLP configurations are supported. Without loss of generality, we assume in this Thesis that all DNN/RNN variants share a standard time frame: the ILP formulation allows intercepting those deployment scenarios where it is impossible to schedule all the DNN variants within a frame.

In the following, we first discuss our assumptions in terms of schedule constraints and LP formulation and then present the experimental set-up.

5.1 Task Model

We consider a periodic task system $\mathcal{T}$ and we model DNN variants as a set of n independent periodic tasks $\tau_1, \ldots, \tau_n \in \mathcal{T}$ that have to be statically scheduled on a multiprocessor platform, comprising a set of heterogeneous m cores. We assume an implicit-deadline periodic task model where each task τ_i is characterized by a period p_i and a relative deadline d_i (in this work we assume implicit-deadline tasks, thus $d_i = p_i$).

Given the platform's heterogeneous nature, a task cannot be associated with a single-valued computational requirement. Moreover, it is not even sufficient to model the variation in the time as a function of the specific core the task is executing on. As an example, the GPU in the Xavier SoC include 8 Streaming Multiprocessors, which can be used as regular (CUDA) cores or can be configured to exploit also the Tensor cores, and an application (e.g., a DNN instance) may be executed on a variable number of SMs. Therefore, each task may exhibit different time bounds depending on the computational element (and mode) it is executed, and the TLP degree that it is granted. We capture this dimensions as a set of CE/TLP configurations $\mathcal{CE} := \{ce_1, \ldots, ce_k\}$ so that each task τ_i is associated a set of time bound $\mathcal{C} = \{c_{i,1}, c_{i,2}, \ldots c_{i,k}\}$ with $c_{i,j}$ denoting the time of task τ_i under configuration $ce_j \in \mathcal{CE}$.

In line with our assessment objective, we are not interested in modeling a full static schedule over a full major frame. We limit our scope to finding a feasible schedule (if it exists) at the smallest time interval (frame) with enforced timing constraints.

Given a set of tasks $\mathcal{T}' \subseteq \mathcal{T}$ to be statically scheduled on a set of CE/TLP configurations $\mathcal{CE}$ within a frame f, a static schedule for $\mathcal{T}'$ in f under configuration $ce_j \in \mathcal{CE}$ is valid only if the cumulative task utilization does not exceed f.

5.2 Linear Programming Model

We modeled the cyclic executive scheduling problem on multiple CE/TLP configurations as an LP problem. LP-based approaches have been exploited for deriving static schedules in both homogeneous [22] and heterogeneous [11] multiprocessor systems. While the considered optimization problem is NP-hard [32], LP approaches are effective in most cases; heuristic-based methods have been proposed to overcome scalability concerns.

An LP model comprises a set of decision variables (possibly constrained to assume only integer values), a set of linear constraints, and an objective function. Constraints and objective function are expressed as (linear) inequalities over the decision variables.

The cyclic executive schedule can be modeled as an instance of a 0/1 optimization, as the sought solution models whether or not a task is mapped to a given computational element.

Intuitively, the objective function aims to minimize the total utilization, and if it fails to find a solution to the LP problem, it means that the task set is not schedulable under any feasible configuration. Other criteria can be specified in the form of weights to guide mapping decisions.

It is worth noting that, in our particular case, we are not interested in finding an optimal solution, but only in proving or disproving the task set schedulability.

To instantiate the task model to the Xavier SoC, and consistently with the investigation conducted in the Thesis, we consider a sub-set of all the supported CE/TLP configurations $\mathcal{CE}_{Xavier} = \{\texttt{CPU}, \texttt{GPU}^{\texttt{RC}}, \texttt{GPU}^{\texttt{RC-comb}}, \texttt{GPU}^{\texttt{TC}}, \texttt{GPU}^{\texttt{TC-comb}}, \texttt{GPU}^{\texttt{RC+TC}}, \texttt{NVDLA}, \texttt{NVDLA}^{\texttt{comb}}\}$. Here `comb` configurations for the NVDLA and GPU cores hints at the possibility of being constrained to always use the multiple instances of the CE as a block.

The set of tasks' timing bounds per configuration is given in input to the ILP as a static bi-dimensional matrix $U[\tau_i \in \mathcal{T}][ce_j \in \mathcal{CE}]$ holding the timing budget of task τ_i when deployed to node ce_j. The main decision variable consists in a bi-dimensional boolean matrix $B[\tau_i \in \mathcal{T}][ce_j \in \mathcal{CE}]$ representing whether τ_i is deployed to ce_j. Accordingly, the objective function would consist in minimizing the cumulative utilization, $\sum_{\tau_i \in \mathcal{T},\ ce_j \in \mathcal{CE}} U[\tau_i][ce_j] * B[\tau_i][ce_j]$.

A set of LP constraints has been defined to guarantee a task can only be mapped to one CE/TLP

(we assume tasks cannot be deployed to multiple CEs) and to enforce the maximum utilization on a CE/TLP configuration not to exceed 100% [22]. Constraints also handle the inter-correlations between CE/TLP, such as the fact that while two applications can be mapped to `NVDLA` at the same time, $\texttt{NVDLA}^{\texttt{comb}}$ is a configuration that implies exclusive use of the `NVDLA`.

5.3 Experimental setup

In our experiments, we assess the impact of supporting different CE and thread-level parallelisms (TLP) on the schedulability of several DNN instances on the same platform. We build on the information we derived from the Apollo software to perform a scenario-based evaluation. We used the timing profile of the applications analyzed in Chapter 4 to derive a predefined set of DNN (or RNN) applications (DNN_{1-5}, RNN_{1-3}), with varying computational and timing requirements under the different CE/TLPs configuration, where the set of CE/TLP configurations matches the one considered in Chapter 4.

Each application is represented as a recurrent task with a worst-case execution time distribution, in the range $[U_{max}, U_{min}]$ milliseconds, which depends on the particular CE/TLP configuration. The time interval has been derived by applying a $\pm 15\%$ inflation factor to the values observed on the Xavier SoC reported in Figure 4.5. As observed in Section 4.3, those reference values also factor in contention effects, and we assume the timing requirements are not changing depending on the deployment configuration of co-running applications.

For each CE/TLP configuration, we generated 16,000 synthetic task sets under different overall utilization thresholds (with a mechanism similar to UUnifast [14]). Task set were generated by randomly selecting several instances of the diverse DNN/RNN types. The utilization of each DNN/RNN is drawn from the intervals reported in Table 5.1 above (values are in milliseconds), which is, in turn, built on the timing characterization results in Section 4.3.

DNN_2, the object detector version working with high-resolution images in Figure 4.5, was not included in the evaluation as it corresponds to a high-resolution variant of object detection application that is clearly over-demanding for the target platform. We use instead DNN_1, the object detector working with standard resolution images.

Still on Table 5.1, it is also worth noting that applications (DNN_1, DNN_3, DNN_4) could not be scheduled on CPU cores (utilization larger than 100%), and RNNs execution is not supported on NVDLA). This is supported in the ILP model by forcing $B[\tau_i][ce_j] = 0$ for specific combinations.

As commented above, focusing on a single scheduling frame is sufficient to fulfill our evaluation objective. We assumed all applications to fit in the same frame, with a reference size of 100 ms.

Table 5.1: Utilization distributions for the DNN/RNN types and CE/TLP, values represented in milliseconds.

		CPU			GPU				NVDLA	
					$\texttt{GPU}^{\texttt{RC}}$		$\texttt{GPU}^{\texttt{TC}}$			
		2	4	6	4	8	4	8	1	2
DNN_1	U_{max}	×	×	×	83.95	49.45	44.85	33.35	20.70	12.65
	U_{min}	×	×	×	62.05	36.55	33.15	24.65	15.30	9.35
DNN_3	U_{max}	×	×	×	27.60	17.25	25.30	17.25	12.65	8.05
	U_{min}	×	×	×	20.40	12.75	18.70	12.75	9.35	5.95
DNN_4	U_{max}	×	×	×	67.85	65.55	81.65	72.45	19.55	11.50
	U_{min}	×	×	×	50.15	48.45	60.35	53.55	14.45	8.50
RNN_1	U_{max}	68.19	34.38	23.80	2.30	2.19	2.19	2.19	-	-
	U_{min}	50.41	25.42	17.60	1.70	1.62	1.62	1.62	-	-
RNN_2	U_{max}	21.05	11.62	8.28	2.19	2.07	2.19	2.07	-	-
	U_{min}	15.56	8.59	6.12	1.62	1.53	1.62	1.53	-	-
RNN_3	U_{max}	6.44	3.57	2.53	1.50	1.38	1.50	1.38	-	-
	U_{max}	4.76	2.64	1.87	1.11	1.02	1.11	1.02	-	-
DNN_5	U_{max}	24.50	13.46	10.35	1.50	1.27	1.50	1.15	4.72	3.45
	U_{max}	18.11	9.95	7.65	1.11	0.94	1.11	0.85	3.49	2.55

5.4 Schedulability results

We used our LP formulation to assess the schedulability of the task sets under specific CE/TLP configurations. All DNNs/RNNs are required to run concurrently on the same system, as observed in Apollo's case. A task set is considered to be infeasible if the LP problem admits no solution. We consider different CE/TLP setting, ranging from single-CE configurations ($\texttt{CPU}, \texttt{GPU}^{\texttt{RC}}, \texttt{GPU}^{\texttt{TC}}$, and `NVDLA` only), to mixture configuration, up to the most flexible setting where all CE/TLP configurations are supported.

The experiments aim at confirming that being able to configure and exploit different computing elements with different task-level parallelism is a fundamental enabler for successfully deploying multiple DNN variants on the same system.

We assess how support for different CE/TLP can be leveraged to sustain the schedulability of systems that would have been not schedulable otherwise. Besides, when a system admits multiple feasible schedules, the ILP could also be instructed to identify, among the existing feasible CE/TLP configuration, the one satisfying a predefined criterion, such as maximizing performance.

To analyze the benefits of our neural network variant proposal, we use, as a baseline reference, single-CE setups, where only one CE is exploited. We create several scenarios in which an increasing

subset of all CEs are used ($\texttt{CPU}, \texttt{GPU}^{\texttt{RC}}, \texttt{GPU}^{\texttt{TC}}, \texttt{NVDLA}$). In each scenario, the utilization thresholds considered for the experiments are computed on the reference utilization of the CE providing the highest performance.

The scenarios we addressed are the following:

- `nvdla+gpu_rc+gpu_tc+cpu`: takes $\texttt{NVDLA}^{\texttt{comb}}$ as reference highest-performance CE, and considers the CE/TLP configurations $\texttt{CPU}, \texttt{GPU}^{\texttt{TC}}, \texttt{GPU}^{\texttt{RC+TC}}, \texttt{NVDLA}^{\texttt{comb}}, \texttt{NVDLA}$;
- `gpu_tc+gpu_rc+cpu`: takes $\texttt{GPU}^{\texttt{TC}}$ as reference highest-performance CE, and considers the CE/TLP configurations $\texttt{CPU}, \texttt{GPU}^{\texttt{TC}}, \texttt{GPU}^{\texttt{RC+TC}}$;
- `gpu_rc+cpu`: only uses `CPU` and $\texttt{GPU}^{\texttt{RC}}$, with the latter being the highest-performing CE.

This approach lets us assess our variants approach under different scenarios with an increasing number of supported CEs, each with specific performance characteristics. Additionally, we assess the flexibility of considering all the units of the highest-performing CE as a single element with their combined performance ($\texttt{NVDLA}^{\texttt{comb}}$) versus providing the scheduler the flexibility to allocate NN instances to independent CE units (`NVDLA`).

As explained in Section 5.3, a large set of workloads with different NN instances has been generated for each scenario, using the cumulative utilization relative to the highest-performing CE as a threshold. In all scenarios, we considered such threshold to vary in 100% to 400% utilization over the scheduling interval.

NVDLA. Figure 5.1 shows the ratio of feasible task sets under the considered utilization thresholds (relative to `NVDLA`) and CE/TLP configurations in the `nvdla+gpu_rc+gpu_tc+cpu` scenario.

Under 100% `NVDLA` utilization, the NVDLA alone can always schedule the task set: both in the $\texttt{NVDLA}^{\texttt{comb}}$ and `NVDLA` setups, we observed a 100% success ratio. This schedulability is obviously the case for $\texttt{NVDLA}^{\texttt{comb}}$, as it is the scenario used to compute the utilization threshold. But it also normally holds for two separate instances of `NVDLA` as the combined use of the $\texttt{NVDLA}^{\texttt{comb}}$ does not necessarily exploit full parallelism. Clearly, with increasing utilization, $\texttt{NVDLA}^{\texttt{comb}}$ cannot schedule any workload. `NVDLA` instead still exhibits a high success ratio at 120% utilization, which only falls rapidly at 140% and becomes zero after 160%. This difference between $\texttt{NVDLA}^{\texttt{comb}}$ and `NVDLA`, is explained by the fact that the utilization is relative to the *combined* use of NVDLA, which does not provide precisely double performance when compared to a single NVDLA instance.

Analyzing the benefits of our variants approach, we can see that enabling the use of other CEs allows us to sustain the execution of all DNN instances (100% success ratio) for loads up to 2.8, significantly beyond what is observed with NVDLAs only. In between 2.8 and 3.4, the flexibility of CE/TLP deployment is exploited at most, allowing to schedule some task sets successfully.

The average numbers of DNN-base functionalities successfully scheduled under the considered work-

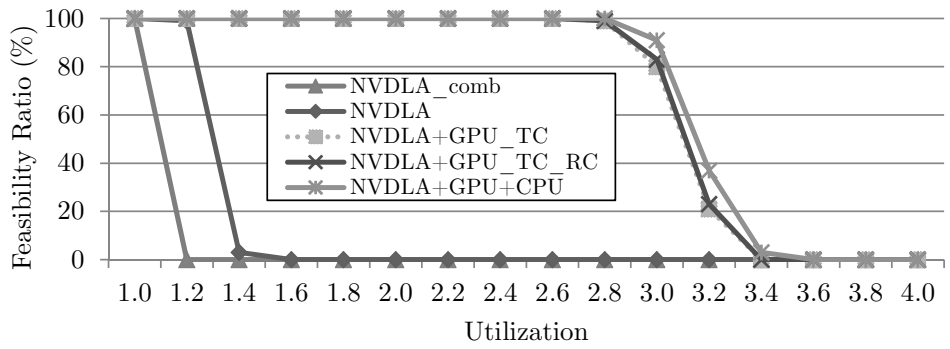


Figure 5.1: Percentage of schedulable workload when the NVDLA is the highest-performance CE.

Table 5.2: Average number of DNN/RNN instances per workload in the `nvdla+gpu_rc+gpu_tc+cpu` scenarios.

	$\texttt{NVDLA}^{\texttt{comb}}$	NVDLA	NVDLA + $\texttt{GPU}^{\texttt{TC}}$	NVDLA + $\texttt{GPU}^{\texttt{TC+RC}}$	NVDLA + $\texttt{GPU}^{\texttt{TC+RC}}$ + CPU
1.0	12.16	12.16	12.16	12.16	12.16
1.2	×	14.66	14.66	14.66	14.66
1.4	×	16.50	17.23	17.23	17.23
1.6	×	×	19.83	19.83	19.83
1.8	×	×	22.46	22.46	22.46
2.0	×	×	24.93	24.93	24.93
2.2	×	×	27.56	27.56	27.56
2.4	×	×	30.13	30.13	30.13
2.6	×	×	32.63	32.63	32.63
2.8	×	×	35.29	35.29	35.24
3.0	×	×	38.46	38.31	38.13
3.2	×	×	43.30	43.22	42.72
3.4	×	×	×	×	49.50

loads and CE/TLP configurations are reported in Table 5.2. Within the feasibility region, all scenarios behave quite similarly as the average task set population grows as long as the computational load increases. Still, within the feasibility region, the average number of instances does not increase when adding more CEs. The only minimal variation happens at 140% utilization, where enabling the GPU allows for one additional DNN-based functionality to be successfully deployed in the average case. When the NVDLAs are saturated, the GPU elements alone are capable of providing up to 340% utilization (NVDLA-defined) and changing the GPU configuration (enabling regular CUDA cores) or introducing the CPUs is slightly affecting both schedulability and number of allocated DNNs.

GPU Tensor cores. The ratio of feasible task sets under the `gtc+grc+cpu` scenario is reported in Figure 5.2. The considered utilization thresholds are relative to the use of 8 $\texttt{GPU}^{\texttt{TC}}$ as a block.

Similarly to the NVDLA, the more flexible configuration, where GPU cores are used as two separate clusters, guarantees an improved schedulability ratio.

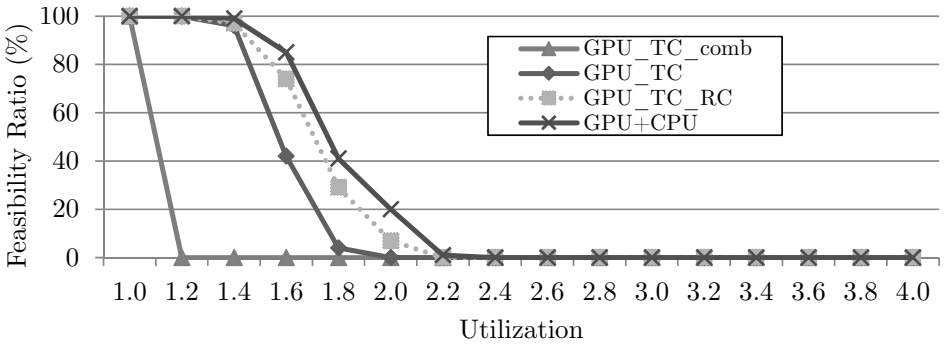


Figure 5.2: Percentage of schedulable workload when $\texttt{GPU}^{\texttt{TC}}$ is the highest-performance CE.

Table 5.3: Average number of DNN/RNN instances per workload in the `gpu_tc+gpu_rc+cpu` scenarios.

	$\texttt{GPU}^{\texttt{TC}-\texttt{comb}}$	$\texttt{GPU}^{\texttt{TC}}$	$\texttt{GPU}^{\texttt{TC}+\texttt{RC}}$	$\texttt{GPU}^{\texttt{TC}+\texttt{RC}} + \texttt{CPU}$
1.0	10.52	10.52	10.52	10.52
1.2	×	11.27	11.27	11.27
1.4	×	11.55	11.58	11.60
1.6	×	11.41	11.72	12.45
1.8	×	11.33	11.90	12.47
2.0	×	×	13.00	14.10
2.2	×	×	×	9.00

The schedulability improvement is even more significant than in the NVDLA case, as the GPU's flexible use allows us to schedule more than almost 80% of the task sets even under a 150% workload. When other CEs are enabled, as suggested by our approach, the schedulability ratio further improves and reaches 85% at 160% utilization.

It is interesting to note the performance improvement obtained by moving from using only $\texttt{GPU}^{\texttt{TC}}$ as two independent clusters to using potentially both $\texttt{GPU}^{\texttt{TC}}$ and $\texttt{GPU}^{\texttt{RC}}$. In fact, one would expect regular cores not to bring any improvement over the Tensor cores scenario, being the Tensor a more advanced accelerator than regular GPU cores. However, while being more advanced, Tensor cores are also more specialized, and their use can be counter-productive for generic applications, as can also be observed in Table 5.1. Exploiting the CPU, instead, allows a comparatively smaller increase in computational power, as expected.

The average numbers of DNN-based functionalities successfully scheduled under the considered workloads and CE/TLP configurations (see Table 5.3) show substantially similar values for all configurations. The configuration using Tensor cores in clusters of four shows slightly different values than those observed when enabling the regular cores and the CPU.

Similarly to the NVDLA scenario, flexible use of the $\texttt{GPU}^{\texttt{TC}}$ alone allows sustaining up to 200% utilization. Again, introducing the CPUs does not affect schedulability and does not allow a larger number of DNN instances.

GPU Regular cores. As the final step in our incremental evaluation, we assess the benefits of our approach in a CE/TLP configuration where only the CPU and the GPU regular cores are made available. In this case, the benefit of the flexible approach $\texttt{GPU}^{\texttt{RC}}$ over the combined $\texttt{GPU}^{\texttt{RC}-\texttt{comb}}$ is remarkable. Conversely, the benefit offered by enabling additional CEs is less consistent when compared to the flexible use of the reference CE. The reason is that regular GPU cores do not seem to support a reasonable degree of parallelism for DNN-based functionalities, as confirmed by relatively close performance between using 4 or 8 GPU cores in Table 5.1. Enabling the use of CPUs makes a negligible difference in the success ratio, which is explained by the relatively small increase in computational power provided by the CPU cores.

The average number of DNN-based functionalities scheduled under these configurations (see Table 5.4), confirms the trend observed for the NVDLA and Tensor cores scenarios. The number of scheduled instances increases with utilization. Only a few DNN instances are added in the average after enabling the use of CPUs.

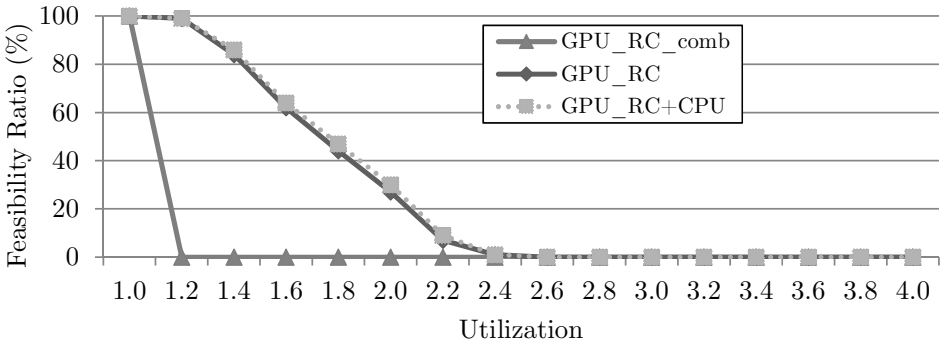


Figure 5.3: Percentage of schedulable workload when $\texttt{GPU}^{\texttt{RC}}$ is the highest-performance CE.

Table 5.4: Average number of DNN/RNN instances per workload in the `gpu_rc+cpu` scenarios.

	$\texttt{GPU}^{\texttt{RC}-\texttt{comb}}$	$\texttt{GPU}^{\texttt{RC}}$	$\texttt{GPU}^{\texttt{RC}} + \texttt{CPU}$
1.0	9.59	9.59	9.59
1.2	×	10.54	10.54
1.4	×	11.35	11.45
1.6	×	11.96	12.08
1.8	×	12.49	12.65
2.0	×	12.88	13.39
2.2	×	14.01	14.71
2.4	×	18.00	18.35

Chapter 6

Related Works

In this Thesis, we focus on a particular AD framework, Apollo, since it is arguably one of the most sophisticated open-source AD framework implementing the entire AD software stack and has support from many top-tier companies. There are some other interesting open-source AD frameworks such as Autoware [52], which claims to be a full-stack implementation, however, are mainly developed in academia. NVIDIA Drive program is another interesting AD system from NVIDIA. This framework implements all the software stack including OS oriented for AD, tools for getting the information from sensors, and the processing of the data. Also, they system includes some specific hardware designed for AD in which it is included in the Xavier SoC, but the software is intended to work only on the more high-end chips such as the NVIDIA Drive PX Pegasus [36]. Furthermore, it is worth mentioning Tesla's Autopilot, which is already deployed in many real cars with boards that are variants from the Drive PX Pegasus but still are working on a level 2 of SAE autonomy. Note the latter systems from NVIDIA and Tesla are not fully available in detail for the sake of studies performed in this thesis.

DL techniques are increasingly used in CRTES as they deliver more precise functional results than other approaches. GPUs are considered for the execution of DL software because of their capability of performing massively-parallel general-purpose computations and efficiency supporting DL libraries [13]. However, GPUs in CRTES bring several challenges for safety [2, 50], and timing. The latter, which can be categorized into three groups, have been addressed by different works:

1. Works with a focus on the implications of GPUs in the real-time properties of the system.
2. Works improving the utilization/efficiency of the existing DL and computer vision software.
3. Works that propose low-level modifications to support DL, such as scheduling algorithms or hardware support.

Research on the real-time properties of GPUs has been conducted for almost a decade. Initial

works focused on scheduling proposals for the peculiar timing behavior of GPUs, which is based on interrupts [24], and deal with their non-preemptive nature, which requires task synchronization [26]. Multiple CPU-GPU allocation strategies have been considered in [25], where the authors evaluate different partitioning and clustering schemes to enable sharing various instances of the same GPU across multiple cores.

In our work, we deal with a heterogeneous set of accelerators and GPU regular/Tensor cores. We consider a set of diverse parallel tasks that can be scheduled under various TLP through multiple CPUs, GPU SMs, and other specialized accelerators; we study how their execution requirements vary depending on the computing element on which they are scheduled.

More recent works have focused on exposing undocumented or miss-documented features of NVIDIA GPUs and their benchmarking [5, 44, 55]. Moreover, [55] is the first real-time paper evaluating NVIDIA's MPS system, which allows multiple processes to execute kernels concurrently in the GPU, containing their SM usage, which is an essential feature for our work. Similarly to our work, [54] considers fine-grained vision-related schedulable entities that can be executed on CPU or GPU, however, it does not contemplate several accelerators beyond the GPU's SMs.

Authors in [56] apply sensor fusion and propose a supervised scheduling algorithm for multiple DNN layers, considering each one as a separate dynamically schedulable entity on a GPU. Similarly to our work, the proposed approach focuses on multiple DNN instances. The focus, however, is limited to a single computational element and does not include the use of multiple elements and thread-level parallelism configurations. Bateni et al. [12] proposed *ApNet*, an approximation-aware real-time neural network, to guarantee that DNN workloads meet their deadlines by using an efficient approximation. Despite the fact that their proposal can incur some accuracy loss, it can ensure the timing predictability. Our work is orthogonal to the ApNet, and applying both approaches can further improve resource utilization and performance.

In another work, Bateni et al. [13] proposed *Predjoule*, which is a timing predictable energy optimization framework. *Predjoule* targets DNN workloads and guarantees the latency and energy efficiency of such workloads. We believe that this work can be extended to support various hardware resources and, in combination with our work, could improve latency and energy consumption.

Capodieci et al. [15] presented a real-time scheduler for GPU activities on SoC systems such as NVIDIA Jetson TX2. They implement and test the Earliest Deadline First (EDF) for GPU tasks, which is enhanced with a Constant Bandwith Server (CBS) based timing isolation mechanism. On the contrary, our work allows the co-scheduling of different computing resources such as CPU cores, GPU cores and Tensor cores, and DL accelerators.

Overall, to the best of our knowledge, this is the first work to study the performance variability of diverse DNN/RNN variants with different computing elements and TLP setups in the Xavier SoC. Also, we exploit an LP model of a heterogeneous static scheduler to assess the platform's capability to sustain the execution of multiple DNN/RNN instances.

Chapter 7

Conclusions and Future Work

Timing correctness is one of the most crucial design constraints in Critical Real-Time Embedded Systems. In this Thesis, we have focused on the utilization of all the available resources to either be able to increase the software load or reduce the cost of the hardware, in CRTES such as AD.

As the number of DNN/RNN instances running in parallel continues to increase in future AD systems, so does the ability to exploit the heterogeneous computing elements in modern computing SoCs. In this Thesis, in support of the first claim, we have analyzed the neural networks concurrently running in the Apollo AD and the current projections in their number. To sustain the latter claim, instead, we have created distinct variants of the different neural-network libraries used in Apollo.

Our results show high diversity in the performance obtained by each variant in each of the computing elements of the Jetson AGX Xavier. This diversity provides an opportunity for exploiting the scheduling strategy to deploy multiple NN-based instances on the same platform simultaneously.

We used an LP formulation for a multicore cyclic executive scheduler to demonstrate the performance increase potentially enabled by different heterogeneous computing elements, and show how this allows deploying multiple advanced NN-based functionalities SoC.

As a future work, we plan work on adapting the DNNs from other AD frameworks in a similar way to see which resource is the most suitable among the others. Moreover, it would be interesting to consider other heterogeneous hardware platforms with different resources to see which of the resources can provide better performance for various specific modules.

Chapter 8

Publications

Our main contributions in this Thesis has been published in the 31st Euromicro Conference on Real-Time System (ECRTS):

Generating and Exploiting Deep Learning Variants to Increase Heterogeneous Resource Utilization in the NVIDIA Xavier [45]

Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella and Francisco J. Cazorla in Proceedings of the 31st Euromicro Conference on Real-Time Systems, ECRTS '19, (Dagstuhl, Germany),volume 133, pages 1–23, 2019.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. Safety-related challenges and opportunities for gpus in the automotive domain. *IEEE Micro*, 38(6):46–55, 2018.

[3] Miguel Alcon, Hamid Tabani, Jaume Abella, Leonidas Kosmidis, and Francisco J. Cazorla. Enroute: on enabling resource usage testing for autonomous driving frameworks. In Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini, editors, *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, pages 1953–1962. ACM, 2020.

[4] Miguel Alcon, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Timing of autonomous driving software: Problem analysis and prospects for future solutions. In *RTAS '20: IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 267–280, 2020.

[5] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU scheduling on the NVIDIA TX2: hidden details revealed. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 104–115. IEEE Computer Society, 2017.

[6] ARINC. *Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc, 1997.

[7] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. `https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php`, 2015.

[8] AUTOSAR. *Specification of RTE Software - AUTOSAR CP Release 4.3.1*, 2017.

[9] Baidu. APOLLO, an open autonomous driving platform. `http://apollo.auto/`, 2018.

[10] Baidu. How coronavirus is accelerating a future with autonomous vehicles. `https://www.technologyreview.com/2020/05/18/1001760/how-coronavirus-is-accelerating-autonomous-vehicles/`, 2020.

[11] Sanjoy K. Baruah, Vincenzo Bonifaci, Renato Bruni, and Alberto Marchetti-Spaccamela. ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors. *J. Sched.*, 22(2):195–209, 2019.

[12] Soroush Bateni and Cong Liu. Apnet: Approximation-aware real-time neural network. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 67–79. IEEE Computer Society, 2018.

[13] Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 107–118. IEEE Computer Society, 2018.

[14] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[15] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for GPU with preemption support. In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 119–130. IEEE Computer Society, 2018.

[16] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between CPU cores and integrated gpus in mixed-criticality platforms. In *22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017, Limassol, Cyprus, September 12-15, 2017*, pages 1–10. IEEE, 2017.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[18] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[19] Tesla Corp. Tesla Autopilot. `https://www.tesla.com/autopilot`, 2018.

[20] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[21] Nachiket Deo and Mohan M. Trivedi. Looking at the driver/rider in autonomous vehicles to predict take-over readiness. *IEEE Trans. Intell. Veh.*, 5(1):41–52, 2020.

[22] Calvin Deutschbein, Tom Fleming, Alan Burns, and Sanjoy Baruah. Multi-core cyclic executives for safety-critical systems. *Sci. Comput. Program.*, 172:102–116, 2019.

[23] Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the aurix^{tm} tc27x. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 97:1–97:6. ACM, 2018.

[24] Glenn A. Elliott and James H. Anderson. Robust real-time multiprocessor interrupt handling motivated by gpus. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 267–276. IEEE Computer Society, 2012.

[25] Glenn A. Elliott and James H. Anderson. Exploring the multitude of real-time multi-gpu configurations. In *Proceedings of the IEEE 35th IEEE Real-Time Systems Symposium, RTSS 2014, Rome, Italy, December 2-5, 2014*, pages 260–271. IEEE Computer Society, 2014.

[26] Glenn A. Elliott, Bryan C. Ward, and James H. Anderson. Gpusync: A framework for real-time GPU management. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 33–44. IEEE Computer Society, 2013.

[27] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 580–587. IEEE Computer Society, 2014.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[29] Joël Goossens, Pascal Richard, Markus Lindström, Irina Iulia Lupu, and Frédéric Ridouard. Job partitioning strategies for multiprocessor scheduling of real-time periodic tasks with restricted migrations. In Liliana Cucu-Grosjean, Nicolas Navet, Christine Rochange, and James H. Anderson, editors, *20th International Conference on Real-Time and Network Systems, RTNS '12, Pont a Mousson, France - November 08 - 09, 2012*, pages 141–150. ACM, 2012.

[30] Intel. Intel® GO™ Automated Driving Solution Product Brief. `https://www.intel.es/content/dam/www/public/us/en/documents/platform-briefs/go-automated-accelerated-product-brief.pdf`.

[31] SAE International. AUTOMATED DRIVING, Levels of driving automation are deined in new SAE International standard J3016. https://www.sae.org/standards/content/j3016_201806/, 2018.

[32] Richard M. Karp. Reducibility among combinatorial problems. pages 85–103, 1972.

[33] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[34] Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 109–118. IEEE Computer Society, 2014.

[35] NVIDIA. cuBLAS: Implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime. http://docs.nvidia.com/cuda/cublas/.

[36] NVIDIA. DRIVE PX. Scalable supercomputer for autonomous driving. http://www.nvidia.com/object/drive-px.html.

[37] NVIDIA. TensorRT: A platform for high-performance deep learning inference. https://developer.nvidia.com/tensorrt.

[38] NVIDIA. TensorRT Support Matrix. https://docs.nvidia.com/deeplearning/sdk/tensorrt-support-matrix/index.html.

[39] NVIDIA. Deep Learning SDK Documentation. https://docs.nvidia.com/deeplearning/sdk/tensorrt-archived/tensorrt-504/tensorrt-support-matrix/index.html, 2018.

[40] NVIDIA. Self-driving Safety Report. https://www.nvidia.com/en-us/self-driving-cars/safety-report/, 2018.

[41] NVIDIA. Tensor Core, The Next Generation of Deep Learning. https://www.nvidia.com/en-us/data-center/tensorcore/, 2018.

[42] NVIDIA. Jetson AGX Xavier. https://www.nvidia.com/es-es/autonomous-machines/embedded-systems/jetson-agx-xavier/, 2019.

[43] Massachusetts Institute of Technology (MIT). Tesla Vehicle Deliveries and Projections. https://hcai.mit.edu/tesla-vehicle-numbers/, 2020.

[44] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alexander C. Berg, and Shige Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In Gabriel Parmer, editor, *2017 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburg, PA, USA, April 18-21, 2017*, pages 353–364. IEEE Computer Society, 2017.

[45] Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the NVIDIA xavier. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany*, volume 133 of *LIPIcs*, pages 23:1–23:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[46] Qualcomm. QUALCOMM Snapdragon 820 Automotive Processor. `https://www.qualcomm.com/products/snapdragon/processors/820-automotive`.

[47] RENESAS. RENESAS R-Car H3. `https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html`.

[48] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[49] Hamid Tabani, Matteo Fusi, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Intpred: flexible, fast, and accurate object detection for autonomous driving systems. In Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini, editors, *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, pages 564–571. ACM, 2020.

[50] Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Francisco J. Cazorla, and Guillem Bernat. Assessing the adherence of an industrial autonomous driving framework to ISO 26262 software guidelines. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 9. ACM, 2019.

[51] Hamid Tabani, Roger Pujol, Jaume Abella, and Francisco J. Cazorla. A cross-layer review of deep learning frameworks to ease their optimization and reuse. In *ISORC '20: IEEE 23rd International Symposium on Real-Time Distributed Computing*, pages 144–145. IEEE, 2020.

[52] The Autoware Foundation. Autoware.Auto. `https://www.autoware.auto/`.

[53] Sergi Vilardell, Isabel Serra, Hamid Tabani, Jaume Abella, Joan del Castillo, and Francisco J. Cazorla. Cleanet: enabling timing validation for complex automotive systems. In Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini, editors, *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, pages 554–563. ACM, 2020.

[54] Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H. Anderson, F. Donelson Smith, and Shige Wang. Making openvx really "real time". In *2018 IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, TN, USA, December 11-14, 2018*, pages 80–93. IEEE Computer Society, 2018.

[55] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. Avoiding pitfalls when using NVIDIA gpus for real-time tasks in autonomous systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPIcs*, pages 20:1–20:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[56] Husheng Zhou, Soroush Bateni, and Cong Liu. Sˆ3dnn: Supervised streaming and scheduling for gpu-accelerated real-time DNN workloads. In Rodolfo Pellizzoni, editor, *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2018, 11-13 April 2018, Porto, Portugal*, pages 190–201. IEEE Computer Society, 2018.

[57] Alex Zyner, Stewart Worrall, and Eduardo M. Nebot. A recurrent neural network solution for predicting driver intention at unsignalized intersections. *IEEE Robotics Autom. Lett.*, 3(3):1759–1764, 2018.

[58] Alex Zyner, Stewart Worrall, and Eduardo M. Nebot. Naturalistic driver intention and path prediction using recurrent neural networks. *IEEE Trans. Intell. Transp. Syst.*, 21(4):1584–1594, 2020.