

# Assessing and Improving the Suitability of Model-Based Design for GPU-Accelerated Railway Control Systems

Alejandro J. Calderón<sup>1,3</sup>[0000–0003–2426–306X], Leonidas Kosmidis<sup>1,2</sup>[0000–0001–9751–1058], Carlos F. Nicolás<sup>3</sup>[0000–0002–2117–913X], Javier de Lasala<sup>4</sup>[0000–0001–7052–553X], and Ion Larrañaga<sup>5</sup>[0000–0003–2399–7168]

<sup>1</sup> Universitat Politècnica de Catalunya, Barcelona, Spain

<sup>2</sup> Barcelona Supercomputing Center (BSC), Barcelona, Spain

<sup>3</sup> Ikerlan Technology Research Centre, Arrasate-Mondragón, Spain

<sup>4</sup> CAF Research Department, Beasain, Spain

<sup>5</sup> CAF Power & Automation, San Sebastián, Spain

**Abstract.** Model-Based Design (MBD) is widely used for the design and simulation of electric traction control systems in the railway industry. Moreover, similar to other transportation industries, railway is moving towards the consolidation of multiple computing systems on fewer and more powerful ones, aiming for the reduction of Size, Weight and Power (SWaP). In that regard, Graphics Processing Units (GPUs) are increasingly considered by critical systems engineers, seeking to satisfy their ever increasing performance requirements. Recently, MBD tools have been enhanced with GPU code generation capabilities for machine learning acceleration, however, there is no indication whether these tools are ready for the design of time-sensitive systems. In this paper we analyse the suitability of commercial MBD toolsets by designing and deploying a model-based parallel control case study on embedded GPU platforms. While our results show promising feasibility evidence, they also reveal shortcomings which should be addressed before these toolsets become fit for developing critical systems. We propose certain improvements that have to be incorporated in these tools to achieve this goal. By implementing our proposals in the generated code, we experimentally show their effectiveness on two NVIDIA-based embedded GPUs.

**Keywords:** Model-Based Design · GPU · Control Systems · Railway.

## 1 Motivation and Introduction

The control of railway electric traction systems requires reading different types of sensors to feed the control algorithms, executing these algorithms and applying their results with precision to guarantee optimal and safe operation. For this purpose, CAF group's traction control electronics use a combination of boards with microprocessors, Digital Signal Processors (DSPs) and Field Programmable Logic Arrays (FPGAs) in a modular way, to adjust costs and space to the needs of each client.

MBD tools have been used in the railway industry for many years to design electric traction control systems, allowing the suitability of the design to be verified by simulation at an early stage. It has also been possible for a long time to transfer the models into code that will be executed by the control electronics, thus facilitating validation through Hardware-in-the-Loop (HIL) simulation and also avoiding the introduction of possible errors inherent to manual intervention. CAF group employs MATLAB-Simulink capabilities to generate, from the model, C code for DSPs and Hardware Description Language (HDL) code for FPGAs. Thus, engineers can work at a higher level focusing on the control algorithms instead of their implementation in the target platforms.

Given the interest of the automotive industry in GPUs for Advanced Driver-Assistance Systems (ADAS) and autonomous driving, several GPU manufacturers are launching products that may be interesting for other functional safety sectors such as railways, especially if they facilitate reductions in costs and SWaP requirements, while allowing the code to still be generated from the models. There are already on the market boards based on System on Chips (SoCs) that integrate a microprocessor and a GPU, which in our railway traction system could replace a microprocessor board and many DSP boards, if real-time behaviour of the code generated from a model could be guaranteed. To validate this, we assess the state-of-the-art of MBD tools for the implementation of GPU-accelerated parallel control systems using a case study. We identify shortcomings which can be addressed to achieve this goal. In this direction, we propose concrete improvements, which we implement and experimentally show their effectiveness on two NVIDIA embedded GPUs.

## 2 Background and Related Work

### 2.1 Model-Based Design

A key attractor for adopting MBD to develop industrial applications is the separation of concerns, where the desired functionality can be described by a pure mathematical representation, i.e. the model. Afterwards, this will be step-wise refined until achieving a description that will be partitioned and allocated to the computing platform.

Equipment manufacturers relying on third-party embedded computing platforms expect MBD to isolate their own intellectual property, related to the functionality of the embedded software, from the particular features of a given hardware, which could jeopardise portability and make platform replacement costly. This expectation typically sacrifices optimal performance when compared to a hand-written implementation tailored to a specific platform.

MBD product development yields many additional advantages: MBD enables the validation of specifications by analysis and simulation, and allows designers to unveil potential design pitfalls at early project stages, which in turn cuts the overall development cost by lowering the time and effort required to fix undesirable behaviour. Moreover, when provided with trustworthy models of the

system environment, MBD enables the systematic exploration of a multiplicity of what-if scenarios under unlikely conditions, which could be hard or completely impossible to fully reproduce in real-world tests.

A typical MBD development process starts with a pure simulation model, known as Model-in-the-Loop (MIL), which is refined and analysed in relevant simulated scenarios until an acceptable behaviour is achieved. Then an initial model-to-code transformation yields a second executable implementation. Depending on whether this implementation could be (cross-)compiled and run on the same host platform or has to be executed on the final target, the configuration is named Software-in-the-Loop (SIL) or X-in-the-Loop (XIL). In addition, depending on the target used in XIL it can be specialised as Processor-in-the-Loop (PIL), FPGA-in-the-Loop (FIL) or in this paper GPU-in-the-loop (GIL).

Recently, GPUs started attracting a strong interest for developing embedded control applications, particularly for real-time controllers involving non-conventional computations –e.g., Deep Neural Networks (DNNs) or Convolutional Neural Networks (CNNs). This also motivated the introduction of MBD toolsets intending to ease the development of GPU applications using the same modelling environments already in use for the other types of computing platforms mentioned before. The novelty of these GPU toolsets, coupled with the special programming patterns required by them, pose several challenges and face many limitations. For example, ensuring the suitability of the languages to describe functionality in a way that can be unambiguously translated to a parallel programming language adequate for coding a GPU, such as CUDA.

## 2.2 Control Systems

Control systems regulate the behaviour of devices or equipment using feedback control loops. Most of modern control systems implementations involve digital embedded microprocessors. Such embedded microprocessors provide interfaces to sensors and actuators, and hard real-time scheduling to guarantee the timely execution of synchronous feedback algorithms. Embedded controllers for real-world applications typically execute multiple cascaded control loops. This control structure tends to require multiple sampling rates; the inner the loop in the code control structure, the higher its required sampling rate is.

An unexpectedly long execution time could delay the action of the controller on the system under control, eventually bringing the system to an inconvenient or risky situation. Therefore, determining the worst-case execution time of the control loop is of uttermost importance for controllers operating critical or protection systems.

Scenarios involving the parallel control of multiple systems can be found in industrial controllers in applications demanding high computational throughput and scalability –e.g., as required by power converters for distributed propulsion systems comprising tens of motors, as proposed for electrification of airborne vehicles [10][12], or power-related applications such as distributed power converters for charging stations or distributed power generation.

### 2.3 GPUs in Critical Systems

GPUs have been initially introduced as special purpose accelerators, in particular for the production of visual content. However, their massively parallel architecture and the introduction of general purpose programmability allowed their use for computationally intensive tasks, including the extremely demanding AI processing, enabled with deep learning.

Autonomy is becoming an important aspect of future critical systems for sectors such as the automotive with the introduction of autonomous driving vehicles, avionics with Unmanned Aerial Vehicles (UAVs), space and planetary exploration with autonomous navigation as well as in industrial automation in industry 4.0 applications to name a few. For this reason, GPU manufacturers started addressing these sectors with the introduction of embedded GPU designs incorporating functional safety features. However, this is still in its infancy with several open challenges currently addressed by the research community.

Several works in the literature address the real-time behaviour of GPUs. As a complex hardware design with a black box non-preemptive behaviour, GPU requires novel approaches for scheduling of real-time tasks [5][8], reduction of offloading overheads [7], characterisation of contention [6] as well as the computation of worst case execution times [3]. Other works analyse necessary properties which need to be taken into account when GPUs are used in the context of critical systems. For example, [14][2] reverse engineered non-obvious aspects of the GPU behaviour which need to be taken into account when GPUs are used in real-time systems, while [4] reverse engineered the memory allocation of GPUs in order to achieve predictable resource consumption with regard to memory and timing. Other authors address the compliance of GPUs with regard to functional safety certification [13][1][11] by proposing the use of language subsets or the adaptation of safety standards. In this work we explore another dimension in this aspect, by using MBD to facilitate certification.

In brief, so far GPUs are mainly employed for high throughput computations but not latency sensitive ones. However, embedded GPUs targeting particularly the automotive and other critical domains are constantly improving in that matter. For example, in the keynote of the GPU Technology Conference 2020 a new GPU architecture and software infrastructure was presented, which will allow to deliver low-latency conversational AI for use in the automotive sector. This is an indication that the latency capabilities of embedded GPUs will be soon competing with other architectures, which were preferred so far for the implementation of such tasks. For this reason, in this work we propose an even more ambitious latency-sensitive parallel case study from the control domain. We assess whether such an application is feasible with the existing technology (both MBD tools and hardware capabilities) or if it will be likely available in the near future according to the current technology trends. In fact, we show that even though existing MBD tools are not yet there, with our proposals this could be achieved even with existing embedded GPUs.

### 3 Case Study: Design and Implementation of a GPU-Accelerated Parallel Control System

#### 3.1 Preliminaries

The primary objective of this work is to assess the capabilities of MBD tools regarding GPU code support for the implementation of real-time parallel control systems. To carry out a comprehensive assessment we opted to implement a case study from the control domain, which is representative of such systems and in addition has not been considered so far for GPU acceleration.

More specifically, we are interested in the evaluation of parallel code generation capabilities and the integration with GPU hardware for PIL or HIL testing. After researching the state-of-the-art in these tools, we concluded that currently, there are only two MBD tools which provide that kind of support: MathWork's MATLAB-Simulink through its GPU Coder toolbox and LabView with its GPU Analysis Toolbox. However, the GPU support in the latter is very limited. It only uses the GPU for the acceleration of some computations such as matrix operations and Fast Fourier Transform (FFT) through the CUDA provided libraries, but does not support custom code generation for GPUs. For this reason, we focus our analysis exclusively on MATLAB-Simulink, which is the only industry-ready MBD tool to support this functionality. However, if in the future any additional existing or new MBD tool includes GPU code generation, our methodology and proposed case study can be applied in order to benchmark its capabilities.

GPU Coder is a MATLAB-Simulink toolbox oriented to the generation of optimised CUDA code for NVIDIA GPUs, with special focus on tasks related to deep learning, embedded vision and autonomous systems. However, to the best of our knowledge, there is no previous analysis on the application of GPU Coder – or any other industrial or academic GPU-capable MBD tool – towards the development of a real-time application, such as a control system.

#### 3.2 The Model

To evaluate the capabilities of GPU Coder and the integration with MATLAB-Simulink, we created an initial model to simulate the control of 8 parallel Permanent Magnet Synchronous Motors (PMSMs), as shown in Figure 1a. We see suitable to start with controlling 8 motors in order to justify the use of the GPU in the system, since a lower number of cores can already be accelerated with existing platforms, such as the TI Delfino platform which supports control of 2 motors. Note however that while our initial model which is described next uses 8 motors, in Section 4.5 we perform a full scalability study for the control of up to 1024 motors. This is a reasonable upper bound of the number of potential motors which can realistically be present in a cyber-physical railway system and controlled together. Moreover, this is the maximum number of threads supported by a single Streaming Multiprocessor (SM) in a GPU and it is the number of threads supported by our embedded GPU platforms, since the embedded GPU of the smaller of them contains a single SM.

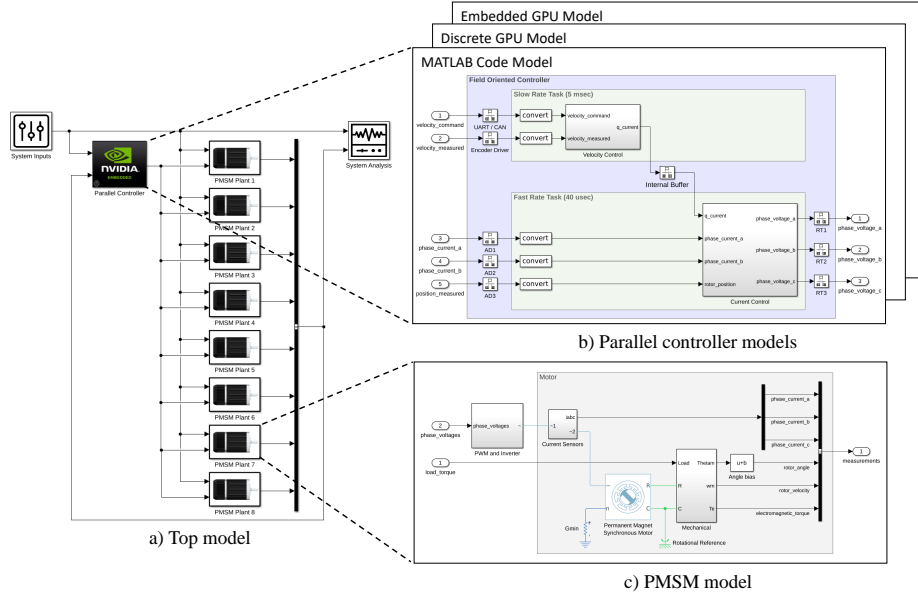


Fig. 1: Simulink model of a parallel PMSM FOC controller

**Algorithm 1: Velocity control algorithm**


---

<b>Input</b>	: reference_speed, measured_speed
<b>Output</b>	: reference_iq
<b>InOut</b>	: accum_error
<b>Parameter</b>	: kp, ki, dt

- 1  $speed\_error \leftarrow reference\_speed - measured\_speed$
- 2  $accum\_error \leftarrow accum\_error + speed\_error * dt$
- 3  $reference\_iq \leftarrow kp * speed\_error + ki * accum\_error$

---

In our implementation we follow the classic model-based development process with gradual refinement as introduced in Section 2.1. First we start with a mathematical model validating its correct behaviour through a MIL simulation. Then we refine the model by generating code executed in the discrete GPU of the host computer where MATLAB-Simulink is installed. This way we perform a HIL / GIL validation, ensuring that its behaviour is identical to the model. Finally, we create the final model which is executed on our target embedded GPU platforms, where the actual evaluation is performed. In that model in particular, we assess not only its identical functionality with the previous models, but also analyse its memory and timing properties.

**Mathematical Description:** A PMSM is a rotating electrical machine which has phase windings in the stator and permanent magnets in the rotor. To operate, it requires the interaction of the magnetic field created by the stator coils and the magnetic field created by the permanent magnets. The three stator

**Algorithm 2:** Current control algorithm

---

```

Input      : reference_iq, ia, ib, angle
Output    : va, vb, vc
InOut     : accum_error_id, accum_error_iq
Parameter : kp, ki, dt
1  ialpha ← ia;
2  ibeta ← (1/sqrt(3)) * (ia + (2 * ib))
3  id ← cos(angle) * ialpha + sin(angle) * ibeta
4  iq ← cos(angle) * ibeta - sin(angle) * ialpha
5  id_error ← -id
6  accum_error_id ← accum_error_id + id_error * dt
7  vd ← kp * id_error + ki * accum_error_id
8  iq_error ← reference_iq - iq
9  accum_error_iq ← accum_error_iq + iq_error * dt
10 vq ← kp * iq_error + ki * accum_error_iq
11 valpha ← cos(angle) * vd - sin(angle) * vq
12 vbeta ← sin(angle) * vd + cos(angle) * vq
13 va ← valpha
14 vb ← (-valpha + (sqrt(3) * vbeta))/2
15 vc ← (-valpha - (sqrt(3) * vbeta))/2

```

---

coils are permanently energised with a sinusoidal current which is 120 degrees apart on each phase. This creates a rotating North / South magnetic field.

In our top model shown in Figure 1a, each one of the 8 PMSM plants has the internal structure shown in Figure 1c. The core of this structure is a Simscape PMSM block connected to a mechanical circuit which is necessary to simulate the physical properties of the motor. The structure also includes a subsystem used for the Pulse-Width Modulation (PWM) generation and a three-phase inverter circuit to simulate the commutation needed to produce rotation in the motor.

To simplify the control of PMSMs a vector control technique known as Field Oriented Control (FOC) is used. The FOC algorithm consists of two control loops which execute at different frequencies. The first control loop is the slower one and is a simple Proportional-Integral (PI) controller, which is used to control the velocity of the motor. Algorithm 1 shows the steps executed in the velocity control loop. The output of this control loop is used as reference value for the quadrature current (*reference\_iq*) in the next control loop. The second control loop is more complex and runs at a higher frequency. This control loop consists of a series of coordinate transforms of the currents to determine the time invariant values of torque and flux of the motor. These values can then be controlled using PI controllers. Algorithm 2 shows the steps executed in the current control loop.

Based on these two control loops, we defined the structure of a field oriented controller subsystem, as shown in Figure 1b. In this configuration, the output of the speed controller is used as reference value in the current controller. However, since both control loops are executed at different frequencies, a rate transition buffer is used to hold the reference value.

In the model shown in Figure 1a, the Parallel Controller block is a Simulink *variant* subsystem on which we implemented three different versions of the field oriented control structure, as shown in Figure 1b.

**MATLAB Code Model:** In the first model, we implemented the velocity and current control algorithms using MATLAB code. The reason for using MATLAB code instead of Simulink blocks is that GPU Coder has the limitation that it can only generate CUDA code from MATLAB code. To be able to control multiple PMSMs in parallel, we created parallel versions of Algorithms 1 and 2 using vectors instead of scalar variables and replacing the scalar operators with MATLAB element-wise operators. To integrate the parallel MATLAB code into the Simulink model, we used *MATLAB Function* Simulink blocks. This first version of the field oriented controller allowed us to validate the correctness of our setup and to register the behaviour of the PMSM plants when interacting with the controller, using a MIL simulation. For this task, we applied different reference input signals for the 8 PMSM plants, as shown in Section 4.

**Discrete GPU Model:** For the second model, we used GPU Coder to generate CUDA code from the MATLAB version of the control algorithms. First, we generated a dynamic-link library with the CUDA version of the velocity and current controllers. Then, we invoked this external library from MATLAB. In the Simulink model, we included two MATLAB Function blocks to invoke the corresponding CUDA functions. This way, on each step of the Simulink simulation, the velocity and current control calculations are executed in the GPU of the host computer and their output is returned to Simulink to drive the simulated motors. In the generated code, each thread in the GPU is in charge of driving a different motor.

**Embedded GPU Model:** For the third model, we executed the generated CUDA code in the target embedded GPUs, to evaluate the capabilities of GPU Coder to interact with external hardware. GPU Coder includes a support package for the deployment of CUDA code in embedded NVIDIA GPUs such as the Jetson and DRIVE platforms. Moreover, the support package provides the functionality to create a PIL session between MATLAB-Simulink and a target embedded GPU platform, which allows the remote execution of code. We implemented this model using a similar approach to the previous one, but creating a PIL session as opposed to creating a dynamic-link library. In addition to the equivalence checking between the MATLAB-Simulink and the generated code for all models, we also evaluate the performance and memory consumption of this version of the application in a standalone setup instead of PIL, as described in Section 4.4. Then, in Section 4.5 we propose improvements for the generated code, which we implement and evaluate experimentally, showing their effectiveness.

## 4 Evaluation

### 4.1 Experimental Setup

We used the MathWorks MATLAB-Simulink toolset release 2021a with GPU Coder 2.1 to develop our parallel control case study, running on a computer

Table 1: Embedded GPU Platforms

Platform	GPU architecture	Compute capability	SMs	CUDA cores	Max. threads per block	RAM
Jetson Nano	Maxwell	5.3	1	128	1024	4GB
Jetson TX2	Pascal	6.2	2	256	1024	8GB

equipped with an NVIDIA GeForce GTX 1650Ti Max-Q discrete GPU. For the final embedded GIL evaluation we used 2 different versions of the NVIDIA Jetson family of embedded GPU single board computers. The details of each embedded platform are provided in Table 1. For the performance evaluation on the embedded platforms, we installed Linux for Tegra (L4T) 32.5 with the PREEMPT RT patches. Moreover, to avoid external interference, all the experiments have been executed with a real-time priority of 98, on an isolated CPU core and with paging disabled. To guarantee the maximum performance, `jetson_clocks` has been enabled with the maximum `nvmodel` profile for each embedded platform.

## 4.2 Validation of the Models

For the models validation task, we designed 8 different reference input signals for the PMSM plants, which represent changes in the target speed of each motor, as shown in Figure 2 with a continuous red line. We applied the reference signals to the controller based on MATLAB code and registered the outputs of the simulation. With this MIL simulation, we validated that the configuration was correct and that the parallel controller was in fact capable of controlling different plants with different set-points. Figure 2 shows also the response of the system with a blue dashed line, trying to adapt the speed of each motor to the requested speed. In Figure 3 we can see the changes in the phase-voltages of each motor in response to the changes of the requested speed. As expected, identical results have been obtained also with the GIL simulations of the discrete and embedded GPUs of the target platforms, which are omitted for the sake of space, as well as because they do not offer any additional value except confirming the equivalence of the MATLAB model and the generated CUDA code.

## 4.3 Integration with External Hardware

In the third model, we used the GPU Coder support package for embedded NVIDIA GPUs to establish a PIL session between MATLAB-Simulink and the Jetson boards to run the simulation. In this setup, Simulink only simulates the physical motors, while their parallel control algorithm is executed in the embedded GPU. As stated in Section 4.2 this is functionally equivalent to the other models. However, we identified two important limitations:

First, in the PIL simulation mode the system establishes a single communication channel at a time to execute a single CUDA kernel on the target. This

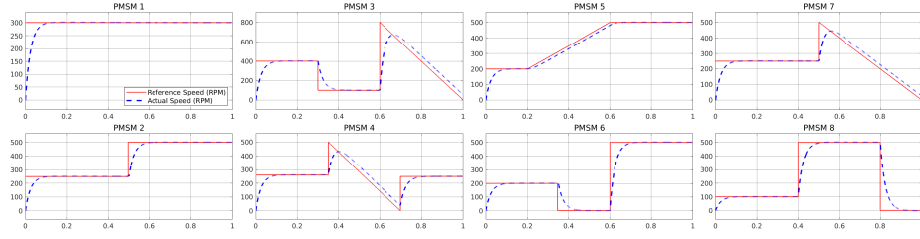


Fig. 2: Reference and actual speeds of parallel PMSMs

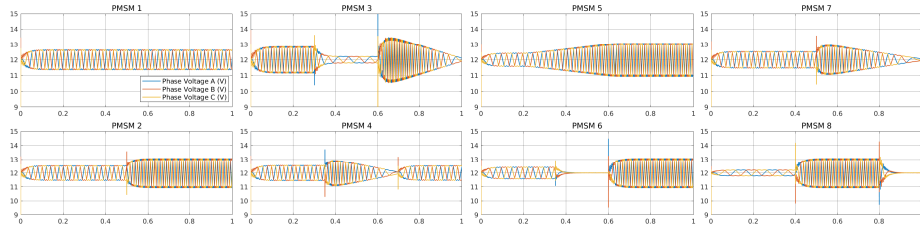


Fig. 3: Phase voltages of parallel PMSMs

means that in cases such as our application where multiple CUDA kernels are used, their execution is serialised. Second, for this same reason, the execution frequency of the target GPU is limited by the communication latency between the host and the target which initiates each kernel execution, increasing the physical execution time required for the simulation.

#### 4.4 Evaluation of Generated CUDA Code

**Performance of generated CUDA code:** In order to evaluate the actual performance of the generated code, we ran our case study directly on the target platforms, without a Simulink PIL setup but as a standalone application. From a control systems perspective, we are interested in measuring the execution time of the instructions that will be executed on each iteration of the control loops: copying values from CPU to GPU, launching the kernel, executing the kernel, and copying the results back from GPU to CPU. On each experiment we execute one million control iterations and we register the average execution time, the maximum execution time and the standard deviation (appearing as Avg, Max and S.D. respectively in the following Tables). To measure the execution time of each iteration, we used the `clock_gettime` function with the `CLOCK_MONOTONIC` clock, which has nanosecond resolution.

GPU Coder can generate CUDA code which uses either the discrete or the unified memory modes. Unified memory allows the CPU and the GPU to share the same address space, which matches the physical memory configuration of the embedded Jetson boards. However, some researchers consider this feature

Table 2: Execution time of generated CUDA code ( $\mu s$ )

Platform	Memory mode	Velocity			Current		
		Avg	Max	S.D.	Avg	Max	S.D.
Jetson Nano	Discrete	265.2	350.9	9.7	489.1	606.0	30.4
	Unified	663.8	1086.7	13.5	1335.1	2045.5	37.3
	Unified fixed	80.7	152.7	2.8	90.6	167.2	3.1
Jetson TX2	Discrete	154.2	273.4	5.7	296.6	383.6	9.5
	Unified	395.1	574.3	14.3	816.5	1013.5	17.9
	Unified fixed	49.7	92.1	2.0	65.6	111.9	8.0

not suitable for critical systems [4] due to its black-box behaviour. Regardless, we perform our evaluation with both memory modes. In the evaluation, we identified that GPU Coder generates `cudaMemcpy` calls to transfer data between CPU and GPU when using the unified mode, which are unnecessary in this mode and costly as we show, given that the CUDA system automatically migrates data between CPU and GPU. This is the first shortcoming we identified. For comparison purposes, we created a fixed version of the code generated for the unified memory mode, removing these unnecessary `cudaMemcpy` calls.

Table 2 shows the execution times of the code generated for the discrete and unified memory modes on the two target platforms. We also include the execution times of the fixed version of the unified memory code. Note that the execution times of the original unified memory code are significantly higher than the execution times of the fixed version, due to the overhead caused by the extra `cudaMemcpy` calls. Therefore, there is room for improvement in the GPU Coder code generation in order to achieve the requirements of a control system.

In control systems, the maximum execution times of the control algorithms will define the maximum sampling rates the controller will be able to achieve. Unexpectedly long execution times could delay the action of the controller on the system under control, which can cause a risky situation in the system. In the case of FOC, the needed sampling rates depend on the physical characteristics of the inverters and the motors. In general, the period of the velocity control loop usually can be around a few milliseconds, while the current control loop typically has a period inferior to 100 microseconds.

As shown in Table 2, while the maximum execution times of the velocity controller are in an acceptable range, the maximum execution times of the current controller are very high, limiting the maximum control frequency of the system.

To better understand the maximum execution times, we executed some iterations of the control loops using the NVIDIA `nvprof` profiler. Tables 3 and 4 show the results reported by `nvprof` for the discrete and unified memory modes.

In both versions, the maximum execution time for the actual kernel execution is in the target range for both the velocity and current controllers. The rest of the time is spent on `cudaMemcpy` and kernel launch / synchronisation calls. Therefore, if the time spent on these calls is reduced or eliminated, it is feasible

Table 3: Profiling results for discrete memory mode ( $\mu s$ )

Platform	Call name	Velocity			Current		
		Avg	Min	Max	Avg	Min	Max
Jetson Nano	Kernel execution	2.0	2.0	2.0	2.6	2.5	2.6
	<code>cudaLaunchKernel</code>	50.3	45.5	60.0	56.0	47.2	69.8
	<code>cudaMemcpy</code>	46.0	30.9	74.9	50.3	32.0	89.0
Jetson TX2	Kernel execution	1.5	1.4	1.6	2.0	1.9	2.1
	<code>cudaLaunchKernel</code>	30.8	27.4	43.3	40.0	31.7	51.9
	<code>cudaMemcpy</code>	29.1	20.6	52.9	31.0	19.7	84.7

Table 4: Profiling results for unified memory mode ( $\mu s$ )

Platform	Call name	Velocity			Current		
		Avg	Min	Max	Avg	Min	Max
Jetson Nano	Kernel execution	2.8	2.7	3.1	3.9	3.7	4.1
	<code>cudaLaunchKernel</code>	76.9	66.6	96.7	86.6	77.0	98.9
	<code>cudaMemcpy</code>	101.4	83.3	142.6	116.1	93.5	173.7
	<code>cudaDeviceSynchronize</code>	46.7	44.9	50.6	53.7	50.5	57.3
Jetson TX2	Kernel execution	2.4	2.4	2.5	3.5	3.4	3.7
	<code>cudaLaunchKernel</code>	56.6	50.5	71.9	64.6	48.4	86.9
	<code>cudaMemcpy</code>	64.6	50.0	114.0	73.2	57.0	120.4
	<code>cudaDeviceSynchronize</code>	29.9	26.9	40.3	33.7	28.4	44.7

to achieve the timings required for the control system. Based on this analysis, on Section 4.5 we propose some further improvements for the generated code.

**Memory overhead of generated CUDA code:** In addition to the performance of the generated code, we also evaluated its memory consumption. For this task we employ the open source GPU memory allocation inspector GMAI, proposed by [4]. GMAI reports that the generated CUDA code performs 20 individual allocations and memory copies for each of the GPU variables, which is inefficient. Although the allocations occur at the application startup and thus do not affect timing, the individual memory copies significantly impact the timing, since they are quite costly as shown in Table 3. On the other hand, in terms of absolute memory consumption, this allocation strategy is beneficial, since all individual memory allocations are quite small and of the same size, so they are allocated from the same size class of the memory allocator, occupying a single memory pool which has a size of 1 MB in the Nano and 2 MB size in the TX2. Note that each of the generated GPU variables corresponds to an array with size as many elements as the number of the motors which are controlled. In total, for the 8 motor configuration, the total requested size for GPU memory from the application is less than 1 KB.

Table 5: Execution time of improved CUDA code versions ( $\mu s$ )

Platform	Improvement	Velocity			Current		
		Avg	Max	S.D.	Avg	Max	S.D.
Jetson Nano	Zero-copy memory	32.7	74.1	3.0	34.4	81.2	2.9
	Persistent kernel	2.9	8.8	0.2	4.0	9.8	0.3
Jetson TX2	Zero-copy memory	20.3	55.5	1.3	31.5	61.9	7.3
	Persistent kernel	3.0	8.8	0.2	4.1	9.8	0.2

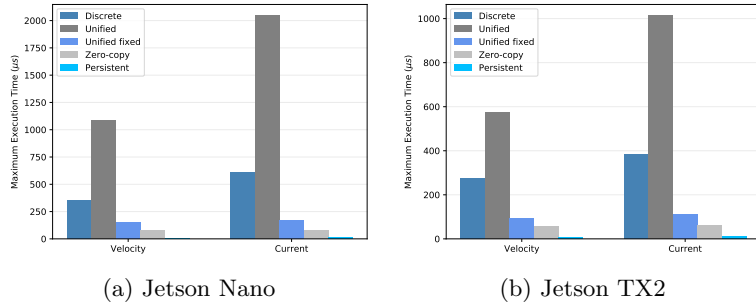


Fig. 4: Maximum execution times of generated code and proposed improvements.

#### 4.5 Improvement of Generated CUDA Code and its Evaluation

In embedded platforms where CPU and GPU share the same physical memory, the memory copy overhead can be eliminated using an alternative memory configuration known as *zero-copy*. This feature allows the allocation of memory regions shared between CPU and GPU, eliminating redundant allocations as well as the copying task itself.

Regarding the kernel launch overhead, it can be reduced using the persistent threads model [9]. In this model, a persistent kernel is launched only once, which iterates waiting for work. Then, the CPU can assign new work to the persistent kernel by just changing values in memory, avoiding the kernel launch process.

Based on these two approaches we modified the generated code in two steps, creating two versions in order to evaluate the benefit obtained from each one. In the first step we replaced the traditional memory allocations with zero-copy allocations to avoid using `cudaMemcpy` calls. In the second step, besides using zero-copy memory, we replaced also the kernel launch / synchronisation with a persistent kernel launch. Table 5 shows the resulting execution times of the control algorithms with these improvements. Figure 4 shows a comparison of the maximum execution times for the different versions of velocity and current controllers on the target platforms.

Note that using zero-copy memory is enough to get maximum execution times in the target range for both control algorithms. Furthermore, when this solution is combined with a persistent kernel, the control loops can be executed

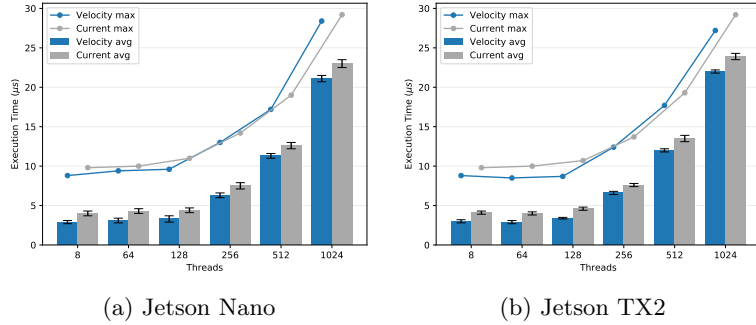


Fig. 5: Performance scalability of improved CUDA code

significantly faster on both platforms. This improvement by an order of magnitude can be beneficial for even tighter control scenarios. In terms of memory, since we replace a host allocation and a GPU allocation with a single zero-copy allocation, the memory consumption is reduced up to 50%.

Finally, to evaluate the scalability of the improved code, we executed the control algorithms with different threads configurations, to control up to 1024 motors in parallel. Figure 5 shows the execution times on the target platforms. Note that the most stable execution times are obtained with up to 128 threads, which is the amount of CUDA cores per SM in both platforms. Even so, in all the cases, the maximum execution times do not exceed 30 microseconds.

## 5 Conclusion and Outlook

In this paper we assessed for the first time the GPU code generation capabilities of MATLAB-Simulink for the design of real-time parallel control systems. We performed this evaluation by designing a novel GPU-accelerated parallel control case study, as a representative application of future railway parallel control systems, which we evaluate in 2 embedded GPU platforms.

Our results show that existing embedded GPU hardware can already support the timing requirements of such a case study, scaling up to 1024 motors, provided that the generated code is optimised according to our proposals. However, due to code generation inefficiencies, the original MBD generated code cannot meet the performance required by the control application. In particular, while the actual GPU generated code is functional, we noticed inefficiencies in the API calls which control the interaction of the GPU with the CPU part of the application. In terms of memory consumption the generated code is reasonable, however the implementation of the memory allocations and transfers is the limiting factor of the control loop frequency, together with the kernel launch overhead.

For these reasons, we conclude that to enable the use of the MathWorks toolset for model-based designing GPU-accelerated real-time control applications, it yet requires to enhance its GPU code generation capabilities at least in

the following aspects: a) add support for zero-copy memory configuration which can eliminate the overhead of memory copies, and b) add support for a method to reduce or eliminate the kernel launch overhead, such as persistent threads.

### Acknowledgments

This work was partially supported by the European Commission's Horizon 2020 programme under the UP2DATE project (grant agreement 871465), by the Spanish Ministry of Economy and Competitiveness under grants PID2019-107255GB and FJCI-2017-34095 and the HiPEAC Network of Excellence.

### References

1. Alcaide, S., Kosmidis, L., Tabani, H., Hernandez, C., Abella, J., Cazorla, F.J.: Safety-related Challenges and Opportunities for GPUs in the Automotive Domain. *IEEE Micro* **38**(6), 46–54 (2018)
2. Amert, T., Otterness, N., Yang, M., Anderson, J.H., Donelson Smith, F.: GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In: *Real-Time Systems Symposium, RTSS*. vol. 2018-January, pp. 104–115 (2017)
3. Berezovskyi, K. et al.: Makespan computation for GPU threads running on a single streaming multiprocessor. In: *Euromicro Conference on Real-Time Systems* (2012)
4. Calderón, A.J., Kosmidis, L., Nicolás, C.F., Cazorla, F.J., Onaindia, P.: GMAI: Understanding and Exploiting the Internals of GPU Resource Allocation in Critical Systems. *ACM Transactions on Embedded Computing Systems* **19**(5) (2020)
5. Capodiecì, N., Cavicchioli, R., Bertogna, M., Paramakuru, A.: Deadline-Based Scheduling for GPU with Preemption Support. In: *Proceedings - Real-Time Systems Symposium, RTSS*. vol. 2018-December, pp. 119–130 (2019)
6. Cavicchioli, R. et al.: Memory Interference Characterization between CPU Cores and Integrated GPUs in Mixed-criticality Platforms. In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. pp. 1–10 (2017)
7. Cavicchioli, R. et al.: Novel Methodologies for Predictable CPU-to-GPU Command Offloading. In: *Euromicro Conference on Real-Time Systems, ECRTS 2019* (2019)
8. Elliott, G.A., Ward, B.C., Anderson, J.H.: Gpusync: A framework for real-time gpu management. In: *Proceedings - Real-Time Systems Symposium*. pp. 33–44 (2013)
9. Gupta, K. et al.: A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In: *2012 Innovative Parallel Computing, InPar* (2012)
10. Kim, H.D., Perry, A.T., Ansell, P.J.: A Review of Distributed Electric Propulsion Concepts for Air Vehicle Technology. In: *2018 AIAA/IEEE Electric Aircraft Technologies Symposium, EATS 2018* (2018)
11. Saidi, S., Steinhorst, S., Hamann, A., Ziegenbein, D., Wolf, M.: Future Automotive Systems Design: Research Challenges and Opportunities. In: *International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS* (2018)
12. Schmollgruber, P., Döll, C., Hermetz, J., Liaboeuf, R., Ridet, M., Cafarelli, I., Atinault, O., François, C., Paluch, B.: Multidisciplinary Exploration of DRAGON: an ONERA Hybrid Electric Distributed Propulsion Concept. In: *AIAA Scitech 2019 Forum* (2019)
13. Trompouki, M.M., Kosmidis, L.: BRASIL: A High-integrity GPGPU Toolchain for Automotive Systems. In: *Proceedings - 2019 IEEE International Conference on Computer Design, ICCD 2019*. pp. 660–663 (2019)
14. Yang, M., Otterness, N., Amert, T., Bakita, J., Anderson, J.H., Smith, F.D.: Avoiding Pitfalls When Using NVIDIA GPUs for Real-time Tasks in Autonomous Systems. In: *Euromicro Conference on Real-Time Systems, ECRTS* (2018)