UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Universitat Politècnica de Catalunya

Computer Architecture Department

Real-Time High-Performance Computing for Embedded Control Systems

Alejandro Josué Calderón Torres

A thesis submitted for the degree of
Doctor of Philosophy in Computer Architecture

Advisor: Leonidas Kosmidis
Barcelona Supercomputing Center (BSC)
Computer Architecture and Operating Systems Group

Co-advisor: Carlos Fernando Nicolás Ramírez
Ikerlan Technology Research Centre
Dependable Embedded Systems Department

Tutor: Francisco Javier Cazorla Almeida
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center (BSC)
Computer Architecture and Operating Systems Group

June 2022

Alejandro Josué Calderón Torres
Real-Time High-Performance Computing for Embedded Control Systems
June 2022
Advisors: Leonidas Kosmidis and Carlos Fernando Nicolás Ramírez
Tutor: Francisco Javier Cazorla Almeida

Universitat Politècnica de Catalunya
Computer Architecture Department
Carrer de Jordi Girona, 1, 3
08034 Barcelona

The work reported in this thesis has been conducted in collaboration between the Universitat Politècnica de Catalunya, the Dependable Embedded Systems department of Ikerlan Technology Research Centre and the Computer Architecture and Operating Systems (CAOS) group of the Barcelona Supercomputing Center (BSC).

Abstract

Critical real-time systems include a wide spectrum of computer systems whose correct behavior is dictated not only by correct functionality but also by their timely execution with respect to predefined deadlines. The increasing demand for higher performance in these systems has led the industry to recently include embedded Graphics Processing Units (GPUs), mainly for machine learning and computer vision tasks. In the real-time control systems industry, there is a trend moving towards the consolidation of multiple computing systems into fewer and more powerful ones, aiming for the reduction of Size, Weight and Power (SWaP). The highly parallel architecture of GPUs could help to develop more advanced, energy efficient, and scalable control systems. However, GPUs are known about their closed source and non-deterministic nature, which complicates the resource provisioning analysis for the implementation of real-time systems. On the other hand, in the real-time time control systems industry, a Model-Based Design (MBD) approach is used for the development and testing of this kind of systems to manage their diversity and complexity. Recently, MBD tools have been enhanced with GPU code generation capabilities for machine learning acceleration, a feature that could also be leveraged for the development of real-time control systems. However, there is no indication whether these tools are ready for the design of time-sensitive systems.

This thesis addresses these problems. First, we present a methodology and an automated tool to extract the properties of GPU memory allocators. This tool allows the computation of the real amount of memory used by GPU applications, facilitating a correct resource provisioning analysis. Then, we present a library which allows the characterization of the use of dynamic memory in GPU applications. We use this library to characterize GPU benchmarks and we identify memory allocation patterns that could be modified to improve performance and memory consumption in embedded GPUs. Based on these results, we present a tool to optimize the use of dynamic memory in legacy GPU applications executed on embedded platforms. Afterwards, we analyze the timing of control algorithms executed in embedded GPUs and we identify techniques to achieve an acceptable real-time behavior. Finally, we evaluate MBD tools in terms of integration with GPU hardware and GPU code generation, and we propose a source-to-source transformation tool to improve the model-based generated GPU code.

Acknowledgement

I would like to express my deepest gratitude towards the many people who have contributed in making this thesis successful. I want to show my greatest appreciation for my advisors, Dr. Leonidas Kosmidis and Dr. Carlos Fernando Nicolás, and my tutor Dr. Francisco Javier Cazorla. I am forever grateful for their knowledge, guidance and encouragement through this whole process. Their recurrent support and contributions have made this thesis a reality.

I would like to extend my gratitude to Mr. Peio Onaindia, Dr. Mikel Azkarate-askatsua and Dr. Jon Perez, for taking on the role of supervisors, always lending a helping hand and words of advice when I needed it most. Their encouragement kept me focused and motivated throughout challenging times. I am thankful to Ikerlan for allowing me to be a part of their team and providing funds for my thesis, this is truly an opportunity of a lifetime. I wish to thank the wonderful people from the Dependable Embedded Systems area, especially my workmates from the Real-Time Systems team. Daily challenges are much more enjoyable to work on when you have such a reliable team by your side, I am constantly learning from all of them.

I am truly grateful for the warm welcome the CAOS group from the BSC extended to me while I worked with them in Barcelona. It was a rich learning experience to be able to work with such qualified professionals.

I would like to thank my wonderful family and friends for being by my side during this process. I am forever grateful to my mother and my father, my brothers and sisters, who have always believed in me, this is also their achievement. Many thanks to my amazing friends, my *Kuadrila*, I feel truly blessed to count with their support. Finally, I wish to thank my loving wife Gina, her unconditional support and encouragement made this thesis possible.

This work was supported with a PhD fellowship at Ikerlan Technology Research Centre, through the EMAITEK program. The publications of this thesis have also received partial support by the Spanish Ministry of Economy and Competitiveness under grants PID2019-107255GB and FJCI-2017-34095, the HiPEAC Network of Excellence and the European Commission through the H2020 UP2DATE project (grant agreement 871465).

Contents

List of Figures

List of Tables

Listings

Introduction 1

1.1 Real-Time Systems and the Need for High Performance

Real-time systems are computing systems whose correct behavior is dictated not only by correct functionality, but also by their timely execution with respect to predefined deadlines. In real-time systems, each task has an activation period and a known worst-case execution time. The time between the activation of a task and its completion must be bounded by its deadline. A real-time task that always has to meet its deadlines is considered to have *hard real-time* requirements. On the other hand, if it is acceptable for a real-time task to miss some deadlines, the task is considered to have *soft real-time* requirements.

In the domain of critical real-time systems, we find a wide spectrum of computer systems. On one end of the spectrum we have safety critical systems, ranging from transportation to medical and control systems. Since human lives are at stake, such systems usually have hard real-time requirements. On the other end, we find business and mission critical systems which although do not impose a threat to human safety, their correct and timely execution is essential to fulfill their mission, typically to provide valuable services to science, society and economy. Examples of such systems are banking and commerce services, communications and scientific space missions, which have somewhat less strict timing requirements, but they are still important for their operation and justification of their high cost.

A recent trend is the adoption of GPUs in the critical domains, in order to satisfy the performance demand of advanced features. Probably the most well-known case is in automotive, where automakers are working on autonomous driving prototype vehicles [1] powered by GPUs mainly for cognitive tasks and artificial intelligence. The medical domain and finance are also employing GPUs [2] mainly for image processing and high-computational capacity, as well as the space domain [3]. Other critical domains are expected to follow as well, especially whenever there is a need for inference based on artificial intelligence (AI) or high compute performance.

To this aim, the GPU market lead vendor NVIDIA has performed significant investments in the automotive and industrial automation sector by designing embedded GPU systems meeting the temperature and reliability needs of these markets, such as the NVIDIA PX2 and its development board Jetson TX2, the NVIDIA Jetson Nano, the NVIDIA AGX Xavier and its latest addition the NVIDIA Xavier NX. Other vendors such as Imagination Technologies have also automotive compliant products like the PowerVR Series6XT GX6650 GPU incorporated in the Renesas R-CAR H3 platform or the latest product lines PowerVR Series8 and Series9, as well as ARM which collaborated with Samsung to produce the GPU platform Exynos Auto V9 to be used in Audi's cars, based on its Mali-G76 GPU [4].

Despite the important performance benefits provided by GPUs, they are notoriously known about their closed source and non-deterministic nature, which introduces several challenges in the development of GPU-accelerated critical real-time systems. In this thesis, we address some of these challenges, focusing particularly on GPU suitability for real-time control systems.

1.2 Resource Provisioning in Critical Real-Time Systems

Critical real-time systems are very diverse and have very different particular requirements, however, all of them have a common property: they require high availability. The key to achieve high availability is the careful resource provisioning of the system, in order to guarantee that each of the tasks of the system has enough resources to be efficiently executed, at the same time preventing from exceeding a limit that can jeopardize the entire system or impact the other tasks.

In particular, one of the most extreme cases of resource provisioning is found in avionics [5], whose operating system standard, namely ARINC653 [6] enforces strict memory and time budgets for each task. This requires that the system engineer needs to figure out the exact memory usage of each task and ensure that the total memory usage does not exceed the size of the system memory. Similarly in timing, the worst case execution time of each task has to be determined, and ensure that it is smaller than its deadline and that the overall system has enough capacity to accommodate the execution of all tasks. Automotive operating systems, AUTOSAR-compliant [7], follow a similar approach in resource allocation, as well as the operating systems in other critical domains like the Integrity RTOS which is used in industrial control systems [8].

In less critical systems built on general purpose operating systems like Unix-based ones, although the operating systems do not impose these limitations for each task, system engineers still perform the same type of analysis. For example, although these operating systems do allow the use of more memory than the one physically present in the system, based on virtual memory and disk-backed memory (a feature known as *paging* or *swap*) and/or compression, the performance of the system is severely affected when this feature is used, compromising its timing behavior and under heavy memory pressure even the stability of the system is jeopardized. Therefore, the accurate resource provisioning allows to prevent such scenarios, guaranteeing that the total capacity of the system is not exceeded.

The introduction of GPUs in critical real-time systems creates new challenges in terms of resource provisioning. In particular, NVIDIA GPUs are programmed in CUDA, a proprietary programming language developed by NVIDIA. The GPU execution model in its rudimentary form follows an accelerator approach, in which the programmer has to explicitly allocate GPU memory and manage transfers between the CPU and the GPU, as well submitting code to be executed in the GPU, in the form of tasks known as *kernels*. Although this explicit resource allocation provides the delusion of full control over the resource management, the actual resource consumption both in memory and timing is larger, hidden behind closed source layers. The reason is that the actual resource management takes place within the CUDA runtime and GPU driver, which are closed source. The GPU products from other vendors are programmed in OpenCL, which despite its name is not more open. OpenCL has a similar programming model to CUDA, which is also implemented in a closed source runtime and GPU driver provided by each vendor.

As a consequence, an accurate resource provisioning of GPU applications is complicated, leading either to underestimation or overestimation of resource provisioning. Although this problem is not yet very evident in the existing under-utilized prototype systems, based on Unix-like operating systems such as Linux, it will soon be a roadblock as these systems will require the deployment of more software functionalities in the same platform. Even more important, the problem will be more pronounced when these systems will be moved to operating systems for critical systems with strict and explicit resource provisioning per task like AUTOSAR and ARINC653.

1.3 Model-Based Design for Critical Real-Time Systems

Critical real-time systems have strong safety and security requirements, which in case they are violated can have severe consequences like endangering human lives. In particular, the safety of these systems depends on their correct functionality as well as their timely response. Thus, it is necessary to prove or collect sufficient evidence that safety and security demands are met in order to achieve *functional safety certification*, so that the system is permitted to be used according to the law.

Each critical domain has a different *functional safety standard* which describes the procedures to be followed for the design and verification of its systems for each criticality level. For example, in the automotive domain, the ISO 26262 highly recommends the use of MBD tools for the highest criticality (ASIL D) software functionalities. Other standards, like the DO-178C used in avionics, the IEC 61508 used in industrial automation and the IEC 62279 used in railway have similar recommendations.

MBD allows the automatic generation of software based on a high-level description of some form. This way, the absence of the human factor prevents the introduction of model to code translation errors which can compromise the safety of the system. In addition, the mathematical properties of the model can be verified and most importantly the generated software is correct by construction. Probably the most prominent example of MBD is SCADE, a software generator based on the Lustre language, which is extensively used in avionics and other domains for the highest criticality software of their systems. Other widely used MBD tools across multiple industries are MATLAB-Simulink and LabView.

In addition to safety, security is very important in critical domains since a compromised system can have adverse consequences varying from dangerous operations and system destruction to a complete denial of service of essential infrastructure in sectors such as energy, telecommunications, transportation etc. In the recent years, several attacks on industrial systems have been experienced, halting industrial control systems [9] or disabling energy providers [10]. MBD methodologies can improve the security of critical infrastructures [11][12] by enforcing additional security policies and allowing strong verification methods such as model-checking [13][14] to be used.

Modern MBD tools began supporting GPU code generation in their newest releases to leverage the computational power of these graphics processors, which have been successfully used to accelerate the machine learning or AI inference workloads that are increasingly needed in many industrial applications. Moreover, GPUs have been started being considered for use in safety critical systems, since they can provide the increased performance required for the implementation of advanced features as well as for the consolidation of several functionalities in a single computing platform in order to achieve reduced SWaP.

While MBD tools have been traditionally used for the design and implementation of critical systems based on single core platforms, there is no study of their GPU code generation capabilities for use in such critical context beyond the obvious field of machine learning.

1.4 Contributions

In this section, we present an overview of the contributions of this thesis regarding overcoming the aforementioned challenges introduced by the use of GPUs in high-performance real-time control systems, in terms of resource provisioning and model-based design.

1.4.1 Reverse Engineering and Analysis of GPU Memory Allocator Properties

GPUs are known about their closed source nature. The real resource consumption of GPU applications is hidden behind closed source layers. As a consequence, an accurate resource provisioning of GPU applications is complicated. In Chapter 4, we present a methodology to reverse engineer GPU memory allocators and extract their internal properties. Based on this methodology, we present a tool to automatically extract the memory allocator properties and calculate the real amount of memory used by GPU applications. At first, our analysis is oriented to NVIDIA GPUs [15]. Then, we show that our methodology is general enough to work with GPUs of other vendors, which are programmed with OpenCL [16]. With these contributions, we allow the accurate resource provisioning for GPU-based critical real-time systems.

1.4.2 Characterization of Dynamic Memory Use in GPU Applications

GPU dynamic memory allocation can have a negative impact on the launch time of GPU kernels due to the overhead added by the memory allocation system when creating new memory pools. Knowing how dynamic memory is allocated and deallocated can help system engineers to identify patterns that can potentially harm the execution time of GPU tasks. On the other hand, GPU applications can use different types of dynamic memory. Knowing how much of each type of dynamic memory is used in a GPU application could be useful to optimize the memory utilization when porting code written for a discrete GPU to an embedded GPU platform. In Chapter 5, we present a library that allows the characterization of the use of dynamic memory in GPU applications. We use our library to characterize three popular GPU benchmark suites, and we identify memory allocation patterns that could be modified to improve performance and memory consumption when deploying these applications in embedded GPUs.

1.4.3 Optimization of Dynamic Memory Use in Embedded GPUs

Using GPUs in critical systems presents several challenges, since GPU programming models rely on explicit dynamic memory management. Moreover, when dynamic memory allocation is used, it is critical to compute the exact amount of memory used as well as to minimize it. In embedded GPU platforms, both the CPU and the GPU share the same DRAM, having both host allocations and device allocations served from the same physical memory. In this scenario, using the traditional GPU memory model would cause duplicate memory allocations and unnecessary memory transfers. In Chapter 6, we present a tool that optimizes the use of dynamic memory in GPU applications that use the traditional memory model when they are executed in embedded GPUs. With this contribution, we allow legacy GPU applications to be used in a critical setup without rewriting them, while at the same time minimizing their memory consumption and memory management runtime overhead.

1.4.4 Timing Characterization of GPU- accelerated Control Algorithms

Control systems are real-time systems that regulate the behavior of devices or equipment using feedback control loops. These control loops must be executed periodically, with sampling rates according to the needs of the system under control. An unexpectedly long execution time of a control algorithm could delay the action of the controller on the system under control, possibly causing an unwanted situation. Most real-world control algorithms are simple enough to be executed on embedded microprocessors. The highly parallel architecture of embedded GPUs could be leveraged to replace multiple embedded microprocessors for the implementation of scalable parallel control systems. However, GPUs are known about their non-deterministic nature, which complicates their use for implementing real-time systems. In Chapter 7, we characterize the timing of control algorithms executed in embedded GPUs using multiple configurations and we identify techniques to achieve an acceptable real-time behavior. Some of our proposed configurations have served for the selection of the baseline research platforms in [17].

1.4.5 Assessment and Improvement of Model-Based Design for GPU Control Systems

In industry, MBD is the preferred choice for developing and testing real-time control systems. Recently, MBD tools have been enhanced with GPU code generation capabilities for deep learning and computer vision acceleration, a feature that could also be leveraged for the development of parallel real-time control systems. However, there is no indication whether these tools are ready for the design of time-sensitive systems. In Chapter 8, we analyze the suitability of commercial MBD toolsets for the development of real-time control systems. We evaluate the integration with GPU hardware and the GPU code generation capabilities [18]. We propose improvements for the generated GPU code, and we present a source-to-source transformation tool that automatically applies our proposed improvements to the generated GPU code. With this contribution, we facilitate the inclusion of embedded GPUs in the classical control system development cycle.

1.5 Thesis Organization

Each of the major contributions of this thesis is presented in a different chapter, following the next structure:

- In Chapter 2 we present the necessary background on GPU programming, model-based design and control systems. We also present some previous works related to the use of GPUs in critical systems.
- In Chapter 3 we describe the methodology and the experimental setup used for the evaluation of the contributions of this thesis. In particular, we describe the hardware platforms, benchmarks, and tools used to carry out the evaluations in the rest of the chapters.
- In Chapter 4 we present a methodology and an automated tool to extract information about the internal properties of the GPU memory allocators. We apply our tool in two automotive case studies to show how a system engineer can be benefited by this information, in order to provision the correct amount of memory.
- In Chapter 5 we present a tool to characterize the use of dynamic memory in GPU applications. We use our tool to characterize three popular GPU benchmark suites and we identify potential improvements to memory allocation patterns.
- In Chapter 6 we present a tool to optimize the use of dynamic memory in legacy GPU applications executed on embedded platforms.
- In Chapter 7 we analyze the timing of control algorithms executed in embedded GPUs and we identify techniques to achieve an acceptable real-time behavior.
- In Chapter 8 we evaluate MBD tools in terms of integration with GPU hardware and GPU code generation, and we propose a source-to-source transformation tool to improve the model-based generated GPU code.
- Finally, in Chapter 9 we present the main conclusions and potential future directions of research of this thesis.

1.6 List of Publications

In this section, we list the publications which were produced as a direct contribution of the research carried out during the development of this thesis.

1.6.1 Accepted Publications

- **Understanding and Exploiting the Internals of GPU Resource Allocation for Critical Systems**
  Alejandro J. Calderón*, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, Peio Onaindia*
  IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2019
- **GMAI: Understanding and Exploiting the Internals of GPU Resource Allocation in Critical Systems**
  Alejandro J. Calderón*, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, Peio Onaindia*
  ACM Transactions on Embedded Computing Systems (TECS), 2020
- **Assessing and Improving the Suitability of Model-Based Design for GPU-Accelerated Railway Control Systems**
  Alejandro J. Calderón*, Leonidas Kosmidis, Carlos F. Nicolás, Javier de Lasala, Ion Larrañaga*
  International Conference on Architecture of Computing Systems (ARCS), 2021

1.6.2 Other Publications

In addition, although they do not constitute explicit contributions of this Thesis, the following publications are tightly related to it.

- **The UP2DATE Baseline Research Platforms**
  Alvaro Jover-Alvarez, ***Alejandro J. Calderón****, Ivan Rodriguez, Leonidas Kosmidis, Kazi Asifuzzaman, Patrick Uven, Kim Grüttner, Tomaso Poggi, Irune Agirre*
  Design, Automation & Test in Europe (DATE), 2021

- **GPU Devices for Safety-Critical Systems: A Survey**
  Jon Perez-Cerrolaza, Jaume Abella, Leonidas Kosmidis, ***Alejandro J. Calderón****, Francisco J. Cazorla, Jose Luis Flores*
  ACM Computing Surveys, 2022

- **On the Safe Deployment of Matrix Multiplication in Massively Parallel Safety Systems**
  Javier Fernández, Irune Agirre, ***Alejandro J. Calderón****, Jon Perez, Jaume Abella, Francisco J. Cazorla*
  Applied Sciences - Special Issue in Machine Learning and Software Intensive Systems: Theory, Methods and Applications, 2022

Finally, the following publication has been recently submitted and is currently under review.

- **Unraveling the Mystery of NVIDIA's Unified Memory for Safety-Critical GPU Systems**
  Xabier Arauzo, ***Alejandro J. Calderón****, Leonidas Kosmidis, Irune Yarza*
  IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2022

Background

2

In this chapter, we provide the background related to this thesis in terms of control systems, model-based design and Graphics Processing Units (GPUs). We introduce basic terminology and fundamental concepts to understand the context of this thesis. We also provide references to some previous works related to the use of GPUs in critical systems.

2.1 Control Systems

Control systems regulate the behavior 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 [19][20], or power-related applications such as distributed power converters for charging stations or distributed power generation.

2.2 Model-Based Design

In industry, model-based design is the preferred choice for developing and testing control systems. 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 jeopardize portability and make platform replacement costly. This expectation typically sacrifices optimal performance when compared to a hand-written implementation tailored to a specific platform. To this end, several authors proposed similar approaches to preserve the platform independence, for example, the Y-development process [21], or the Platform-Based Design (PBD) [22][23] among others.

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 behavior. 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 analyzed in relevant simulated scenarios until an acceptable behavior 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 specialized as Processor-in-the-Loop (PIL), FPGA-in-the-Loop (FIL) or in the context of this thesis GPU-in-the-loop (GIL). Moreover, with the advent of increasingly capable platform simulation tools, these configurations can be exercised in a co-simulation environment coupling both the modeling environment – e.g., Simulink – and a virtual platform simulator.

Among the latter, we can mention Mentor Graphics QuestaSim or ModelSim to host-simulate the behavior of programmable logic, while QEMU, Imperas Open Virtual Platform, or WindRiver Simics could be used to simulate diverse microprocessors. At this stage, the trustworthiness of the verification results depends on the reliability of the simulators used, thus the immediate afterwards step is to exercise the obtained application in the final platform by injecting the same stimuli from the simulated scenarios to the implementation under test and verifying the equivalence of the outputs with regard to the previously validated realizations.

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 modeling 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.

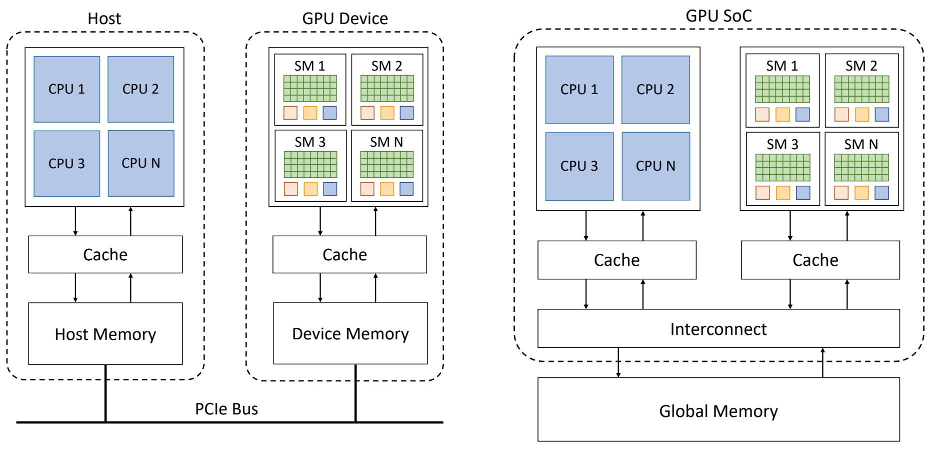
2.3 Graphics Processing Units (GPUs)

GPUs are high-performance co-processors initially developed for handling computationally intensive tasks related to rendering computer graphics. Modern GPUs have become more powerful and generalized, enabling them to be applied to accelerate time-consuming general-purpose parallel computing tasks with excellent performance and high power efficiency. This section presents GPU-related terminology and fundamental concepts relevant to understanding the rest of the thesis.

2.3.1 GPU Architecture and Terminology

GPUs are massively parallel computing devices with a many-core architecture. The term many-core is usually used to describe multi-core architectures with tens or hundreds of cores. Even though the term core can be used to describe the processing elements of CPUs and GPUs, a GPU core is very different from a CPU core. A CPU core is a robust unit designed for complex control logic, seeking to optimize the execution of sequential programs, while a GPU core (scalar core) is a lightweight unit optimized for data-parallel tasks with simple control logic, focusing on the throughput of parallel programs [24].

Currently, GPUs are not independent platforms but devices that operate as co-processors of a CPU. Therefore, GPUs must work as a part of a heterogeneous architecture in conjunction with a CPU host. For this reason, in GPU computing terminology, the CPU side of this architecture is called the *host* and the GPU side is called the *device*. As shown in Figure 2.1, there are two different versions of the heterogeneous CPU-GPU architecture.



(a) *Discrete architecture* **(b)** *Embedded architecture*

Figure 2.1: Heterogeneous CPU-GPU architectures

In a discrete architecture (Figure 2.1a), GPUs are separate adapter cards that are plugged into the host computer using the PCIe bus. The GPU cards have their own DRAM memory, which is different from the memory of the host computer. To operate, the data must be copied from host memory to device memory (and vice-versa) through the PCIe bus. This is the typical architecture for servers and desktop computers. In an embedded architecture (Figure 2.1b), the CPU and GPU are integrated into the same System on Chip (SoC) and share the same DRAM memory. This architecture is common in mobile and embedded devices.

Using CUDA terminology, a GPU is composed by one or more Streaming Multiprocessors (SMs). Each SM contains several *scalar cores* (CUDA cores) and other resources such as registers, shared memories and *warp schedulers*. The *compute capability* of a SM is a version number that identifies the hardware features available on the SM. A scalar core is a pipelined Arithmetic Logic Unit (ALU) capable of executing integer and floating-point operations. The scalar cores execute groups of

Table 2.1: Equivalent terms between CUDA and OpenCL terminology

CUDA terminology	OpenCL terminology
Streaming Multiprocessor	Compute Unit
Scalar Core (CUDA Core)	Processing Element
Warp	Wavefront
Grid	Computation Domain
Block	Work-group
Thread	Work-item

32 lockstep threads known as *warps*, in a Single Instruction Multiple Threads (SIMT) fashion. The warp schedulers decode and issue instructions to be executed by each warp. During the execution, resources like shared memory and registers are shared by the warps being scheduled. The warp schedulers take advantage of memory stalls by swapping a stalled warp with a different warp to run on the set of 32 scalar cores assigned to it. This way, the warp schedulers can hide the latency related to memory accesses.

On the software side, a GPU application consists of two parts: host code and device code. Host code runs on the CPU and is responsible for managing the data and execution parameters before loading the compute intensive task into the GPU. Device code is executed on the GPU in the form of functions known as *kernels*. Before launching a kernel in a discrete architecture, the user prepares the data to be processed using host memory, reserves the corresponding amount of device memory, and triggers a copy operation from host to device memory. Once the kernel execution is finished, it is required to copy the results from device memory to host memory. In an embedded architecture, since CPU and GPU share DRAM memory, this can be done more efficiently by defining a single region of shared memory to avoid memory copy operations. However, in this scenario, memory coherency mechanisms must be taken into account. When launching a kernel to the GPU, the user specifies a *grid* configuration with the number of *blocks* and *threads* per block to be executed, which will vary according to the size of the problem.

So far, we have described the architecture and basic functionality of GPUs using CUDA terminology, which applies to NVIDIA GPUs. However, a similar terminology applies to GPUs of other vendors, which are programmed using the OpenCL language. Table 2.1 shows the equivalent terms used for OpenCL devices. For convenience, in the rest of this thesis we will use CUDA terminology.

2.3.2 GPU Programming Fundamentals

CUDA Programming Model

CUDA is a parallel computing platform and programming language for NVIDIA GPUs. CUDA extends the C/C++ and Fortran languages by allowing the programmer to define functions called kernels, which are executed in parallel by several CUDA threads on a GPU.

```
__global__ void vectorAdd(int *A, int *B, int *C)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    C[id] = A[id] + B[id];
}

int main()
{
    int *h_A, *h_B, *h_C;
    int *d_A, *d_B, *d_C;

    int blocks = 2;
    int threads = 32;
    int n_elements = blocks * threads;
    size_t bytes = n_elements * sizeof(int);

    h_A = (int*)malloc(bytes);
    h_B = (int*)malloc(bytes);
    h_C = (int*)malloc(bytes);

    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);

    vectorAdd<<<blocks, threads>>>(d_A, d_B, d_C);
    cudaDeviceSynchronize();

    cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost);

    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

Listing 2.1: CUDA vector addition program

The structure of a typical CUDA program has the next steps: allocate host memory to prepare data on the host side, allocate device memory on the GPU side, copy data from host memory to device memory, launch a kernel to process the data on the GPU, copy the results from device memory to host memory, and finally, free the allocated host and device memory. To exemplify this process, Listing 2.1 shows a CUDA vector addition program.

In CUDA, both the host code and the device code can be part of the same program. The NVIDIA `nvcc` compiler takes care of each part and divides the code to be executed on the host from the code to be executed on the device. In Listing 2.1, the device code is in lines 1 to 5, which is a CUDA kernel to add the values of two vectors and store the results into a third vector (C = A + B). This kernel does not execute automatically in the GPU, it has to be launched from the CPU side. Lines 7 to 41 show the host code, which executes on the CPU. First, we allocate host memory (lines 17 to 19) and device memory (lines 21 to 23) for the three vectors. Then, we copy the data of the two vectors we want to add from host memory to device memory (lines 25 and 26). For simplicity, we do not show code to initialize the host vectors before copying the values.

When the data is ready to be processed at the GPU side, we launch the GPU kernel (line 28). When launching a kernel, the programmer specifies the distribution of the threads through the threads per block and blocks per grid values. In this case, we are launching a grid of 2 blocks with 32 threads each, for a total of 64 threads, which will calculate in parallel the addition of vectors with 64 elements. We used one-dimensional integer values to specify the number of blocks and threads according to the complexity of our problem. However, CUDA supports 3-dimensional variables (`dim3`) to launch 3-dimensional grids to solve more complex problems. The grid configuration will vary according to the size of the problem being solved and the characteristics of the GPU hardware. Since individual blocks are assigned to specific SMs, and each SM has a determined number of CUDA cores and warp schedulers, the size of the grid could be defined taking into account this information to make use of the resources efficiently.

After launching the GPU kernel, the CPU can continue executing other tasks without waiting for the GPU to finish. However, we can also wait for the GPU using a synchronization function (line 29). When the GPU finishes the kernel execution, we copy the results from device memory to host memory. Finally, we release both host and device memory (lines 33 to 38) before finishing the program to avoid memory leaks.

OpenCL Programming Model

OpenCL follows a similar programming model with CUDA, however it is a lower level language than CUDA. This means that the same functionality is implemented with more API calls which offer finer grained control, at the expense of programming complexity. The structure of a typical OpenCL program has the next steps: initialize the platform, load and compile the kernel code, allocate host memory to prepare data on the host side, allocate device memory on the device side, copy data from host memory to device memory, prepare kernel arguments, launch kernel to process the data on the device, copy the results from device memory to host memory, and finally, free the allocated host and device memory.

```
__kernel void vectorAdd(__global int *A, __global int *B, __global int *C)
{
    int id = get_global_id(0);
    C[id] = A[id] + B[id];
}
```

Listing 2.2: OpenCL vector addition kernel

To compare with CUDA, we use an OpenCL version of the vector addition program. First, in OpenCL, the device code and the host code are always written in different files. Listing 2.2 shows the contents of the OpenCL kernel file for vector addition. We can observe that the OpenCL syntax for writing the device code is very similar to the CUDA syntax.

```
#include <CL/cl.h>
#define MAX_SOURCE_SIZE (0x100000)

int main()
{
    cl_platform_id platform_id;
    cl_device_id device_id;
    cl_uint num_platforms;
    cl_uint num_devices;
    cl_int ret;

    clGetPlatformIDs(1, &platform_id, &num_platforms);
    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &num_devices);
    cl_context context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
    cl_command_queue c_queue = clCreateCommandQueue(context, device_id, 0, &ret);
```

Listing 2.3: OpenCL platform initialization

On the host side, the OpenCL program starts with the platform initialization, as shown in Listing 2.3. The platform initialization is necessary because OpenCL not only works with GPUs but also with other computing platforms like multi-core microprocessors and FPGAs. Since OpenCL assumes that there can be different computing platforms in the system, the first step is to specify the platform ID and the device type (lines 12 and 13). With this information, we create an OpenCL context (line 14) and a command queue (line 15) to send execution commands to the specified device. In CUDA, it is also possible to select a specific device when more than one GPU is present in the system. However, the CUDA context creation is transparent to the user, and it is done automatically by the runtime system.

After the platform initialization, it is necessary to load the source code of the kernel and compile it, as shown in Listing 2.4. Since the source file can contain more than one kernel, it is necessary to specify the name of the kernel when creating the kernel object (line 23).

```
16      FILE *file = fopen("vectorAddKernel.cl", "r");
17      char *source_str = (char*)malloc(MAX_SOURCE_SIZE);
18      size_t source_size = fread(source_str, 1, MAX_SOURCE_SIZE, file);
19      fclose(file);
20
21      cl_program program = clCreateProgramWithSource(context, 1, (const
        ↪  char**)&source_str, (const size_t*)& source_size, &ret);
22      clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
23      cl_kernel kernel = clCreateKernel(program, "vectorAdd", &ret);
```

Listing 2.4: OpenCL vector addition kernel creation

Once the device has been selected, and the context, command queue and kernel objects have been created, we are ready to prepare and launch the kernel execution. Listing 2.5 shows the host code needed to prepare the memory objects and launch the kernel execution into the device. First, we allocate host memory (lines 28 to 30) and device memory (lines 32 to 34) for the three vectors. Then, we enqueue the memory copy of the two input vectors to pass their data from host memory to device memory (lines 36 and 37). Before launching the kernel, it is necessary to set the kernel arguments (lines 39 to 41).

When the data is ready to be processed at the device side, we enqueue the kernel launch (line 43). When launching a kernel, the programmer specifies the distribution of the threads in the form of a computation domain. A computation domain is defined with two sizes: the global size, which is the total number of work-items, and the local size, which is the number of work-items per work-group. Like in CUDA, these sizes can have up to three dimensions to model complex problems. Because of the simplicity of this example, we use a computation domain of one dimension, with 64 total work-items and 32 work-items per work-group, meaning that we will divide the work into 2 work-groups. Each work-item will calculate the addition of one element of the first input vector with the corresponding element of the second input vector.

```
24      size_t global_size = 64;
25      size_t local_size = 32;
26      size_t bytes = global_size * sizeof(int);
27
28      int *h_A = (int*)malloc(bytes);
29      int *h_B = (int*)malloc(bytes);
30      int *h_C = (int*)malloc(bytes);
31
32      cl_mem d_A = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, &ret);
33      cl_mem d_B = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, &ret);
34      cl_mem d_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, &ret);
35
36      clEnqueueWriteBuffer(c_queue, d_A, CL_TRUE, 0, bytes, h_A, 0, NULL, NULL);
37      clEnqueueWriteBuffer(c_queue, d_B, CL_TRUE, 0, bytes, h_B, 0, NULL, NULL);
38
39      clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_A);
40      clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&d_B);
41      clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*)&d_C);
42
43      clEnqueueNDRangeKernel(c_queue, kernel, 1, NULL, &global_size, &local_size, 0,
        ↪  NULL, NULL);
44      clEnqueueReadBuffer(c_queue, d_C, CL_TRUE, 0, bytes, h_C, 0, NULL, NULL);
45
46      free(h_A);
47      free(h_B);
48      free(h_C);
49      clReleaseMemObject(d_A);
50      clReleaseMemObject(d_B);
51      clReleaseMemObject(d_C);
52      clFlush(c_queue);
53      clFinish(c_queue);
54      clReleaseKernel(kernel);
55      clReleaseProgram(program);
56      clReleaseCommandQueue(c_queue);
57      clReleaseContext(context);
58
59      return 0;
60  }
```

Listing 2.5: OpenCL vector addition kernel execution

After the kernel execution, we copy the results from device memory to host memory (line 44). Finally, we release the host allocations (lines 46 to 48), the device allocations (lines 49 to 51), and we destroy the command queue, context and kernel objects (lines 52 to 57).

2.3.3 Memory Allocation in GPUs

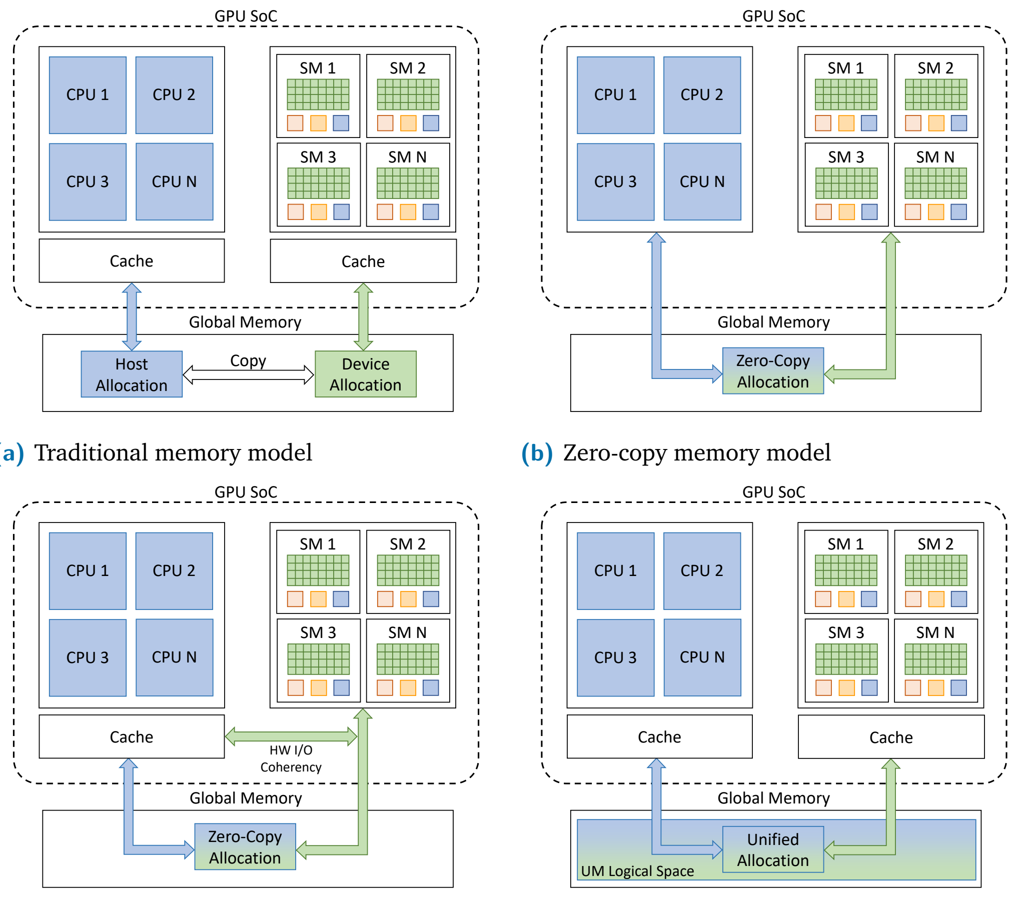
Memory Allocation in CUDA

In the traditional CUDA programming model, the programmer is in charge of explicitly managing memory for both the CPU and the GPU, including allocation, deallocation and transfers between the host memory and the device memory. Regular CPU memory ie. allocated using `malloc` is by default *paged*, which means that the operating system can swap it out to the disk if needed, typically due to memory oversubscription. On the other hand, GPU memory, allocated with `cudaMalloc` is always non-paged, that is, it is always present in the memory. Copies between CPU and GPU memory are performed by DMA (Direct Memory Access) operations. However, as DMA transfers are asynchronous with respect to the CPU execution, they can operate only when the pages are guaranteed to be resident in the memory. Since this is not always the case for paged memory, the transfers need to pass from a staging area of non-paged memory. In other words, in a CPU to GPU transfer, memory needs to be copied first to this intermediate buffer using the CPU and therefore synchronously, before the DMA can kick in to perform the asynchronous transfer to the device. This results in additional memory and additional timing overhead in GPU transfers.

In order to avoid these overheads, the programmer can allocate non-paged CPU memory, also known as *pinned* memory or *paged-locked* memory using `cudaMallocHost`. However, this type of memory in the system is limited and its allocation is more expensive since it requires a user space to kernel space switch. This allows the use of fully asynchronous transfers using `cudaMemcpyAsync`.

There is also the option to allocate another type of *pinned* memory in the CPU side, which is also memory-mapped to the GPU, using `cudaHostAlloc` and specifying the flag `cudaHostAllocMapped`. This means that no explicit copies are required between the CPU and GPU, which gives the name *zero-copy*. Depending on the type of the GPU, this is implemented in a different way. In a discrete GPU, the copies are performed in a fine-grained manner using the DMA engines to transfer the data over the PCIe link. On the other hand, in embedded GPUs which share the same main memory with the CPU, the GPU directly accesses the same memory as the CPU. Of course, in both cases it is up to the programmer to ensure the consistency of the shared memory between CPU and GPU. This functionality is supported by a feature known as UVA (Unified Virtual Addressing), which allows both the CPU and the GPU to operate using the same virtual address.

Finally, CUDA also provides a feature called Unified Memory, which takes away the responsibility of transferring data between CPU and GPU. Unified Memory allows the programmer to do a single allocation for both CPU and GPU using `cudaMallocManaged`, which eliminates the need of memory copies and simplifies the code writing. Unified Memory is also implemented using the UVA feature, which allows the use of a single memory pointer accessible from both the host side and the device side. Data transfers between CPU and GPU are done internally by the CUDA runtime, which migrates memory pages on demand. Despite the increase in productivity, the performance of this feature heavily depends on the memory access patterns of each application, and it adds even more black-box behavior to the memory management.



(a) Traditional memory model **(b)** Zero-copy memory model

(c) Zero-copy with HW I/O coherency model **(d)** Unified memory model

Figure 2.2: CUDA memory models in embedded GPUs

Since the CPU cores and the integrated GPU share the same physical memory in embedded GPU platforms, different considerations must be taken into account to select the appropriate memory model to achieve better performance and efficient memory consumption of GPU applications. Figure 2.2 shows a visual representation of how the different CUDA memory models work in embedded GPU platforms.

When using the traditional memory model (Figure 2.2a), the memory is partitioned into two logical spaces, one for host allocations and the other one for device allocations. Before a kernel execution, the data is copied from the host logical space to the device logical space. Once the kernel execution finishes, the results are copied back from the device logical space to the host logical space. In this scenario, both the CPU and GPU caches are enabled, which can accelerate data transfers. Moreover, the data access synchronization between the CPU cores and the integrated GPU is guaranteed by design. However, in most of the cases, the use of caches is not enough to completely hide the data transfers overhead. Additionally, the memory consumption is not optimal, since both CPU and GPU allocations are served from the same physical memory.

The zero-copy memory model provides a more efficient approach. Since both CPU and GPU share the same physical memory, it allows the CPU to share pointers to pinned host allocations that the GPU can directly access without relying on DMA transfers over the PCIe bus. This eliminates the need for memory copies and reduces memory consumption compared to the traditional memory model. However, having a shared memory space requires the system to guarantee cache coherence between CPU and GPU. Since software-based cache coherency mechanisms can add extra overhead, some platforms, such as the NVIDIA embedded GPUs with compute capability less than 7.2, address the cache coherency problem by disabling the last level caches from both CPU and GPU (Figure 2.2b). This solution can negatively affect the performance of cache-dependent GPU applications. For this reason, NVIDIA recommends the use of zero-copy memory for smalls buffers, since the caching effect is negligible for those buffers. Zero-copy memory can also be beneficial in applications with large buffers if the memory access patterns does not rely on caches.

To reduce this limitation, most recent NVIDIA embedded platforms with compute capability 7.2 or higher implement a hardware-based I/O coherency mechanism (Figure 2.2c). I/O coherency is a feature that allows an I/O device such as a GPU to read the latest updates in the CPU caches. It removes the need to perform CPU cache management operations when the same physical memory is shared between CPU and GPU. However, the GPU cache management operations still need to be performed because the coherency is one way. For this reason, the GPU cache remains disabled when using zero-copy memory on these platforms.

In embedded GPU platforms, unified memory allocations are pointers to a single unified logical space, which can be accessed by both the CPU and the GPU (Figure 2.2d). However, the implementation of unified memory in embedded GPUs is quite different from that for discrete GPUs. Instead of having on-demand page migrations between host memory and device memory, the embedded implementation of unified memory is similar to the zero-copy memory model with single allocations for both CPU and GPU, reducing memory consumption. Nevertheless, unlike the zero-copy memory model, both CPU and GPU caches are enabled in the unified memory model, which requires the runtime system to perform software-based cache coherency maintenance operations. The overhead caused by these operations is higher in embedded platforms with compute capability less than 7.2, as they lack hardware I/O coherency. It is important to note that, according to NVIDIA, software-managed coherency is by nature non-deterministic and not recommended in a safe context. Therefore, in these applications, it is preferable to use the zero-copy memory model [25].

Memory Allocation in OpenCL

In OpenCL, memory allocations are handled via memory objects, using the `cl_mem` type. Generic memory allocations are known as *buffer objects* and are created using the `clCreateBuffer` function. This function receives a *flags* parameter which controls how the memory will be accessed by the device. By default, the memory allocated by this function belongs to the device, and the value of flags is `CL_MEM_READ_WRITE`, which indicates that the device can read and write in this region of memory. However, it also can be configured to be read-only using the `CL_MEM_READ_ONLY` flag or write-only using the `CL_MEM_WRITE_ONLY` flag.

The flags parameter can also be used to specify which kind of memory will be allocated. The `CL_MEM_USE_HOST_PTR` flag indicates that OpenCL should use the memory referenced by a host pointer passed to the function instead of allocating a new memory region. This means that the user is responsible of allocating this region of host memory before calling the `clCreateBuffer` function. According to the standard, OpenCL implementations can cache the contents of the host memory region in device memory.

The `CL_MEM_ALLOC_HOST_PTR` flag indicates OpenCL to create the allocation using host accessible memory. Usually this is the flag used to allocate pinned memory in OpenCL implementations. The `CL_MEM_ALLOC_HOST_PTR` flag can be used with the `CL_MEM_COPY_HOST_PTR` flag to initialize the contents of the new allocated memory with the values stored in a buffer referenced by a host pointer passed to the function. To read or write data to a buffer created using the `CL_MEM_ALLOC_HOST_PTR` flag, the user should use the `clEnqueueMapBuffer` function to map the memory region, operate on the buffer and then use the `clEnqueueUnmapMemObject` function to unmap the memory region.

The zero-copy behavior is not explicitly defined in the OpenCL standard. It is important to take into account that the OpenCL standard is just a set of abstract definitions, and each vendor is responsible for their implementation. For example, in the Intel implementation of OpenCL, the buffers created using the `CL_MEM_ALLOC_HOST_PTR` and `CL_MEM_USE_HOST_PTR` flags are by default zero-copy buffers [26], however, this may not be the case with the implementation of other vendors.

2.4 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 domain is still in its infancy with several open challenges which are currently addressed by the research community.

Several works in the literature address the real-time behavior of GPUs. As a complex hardware design with a black box non-preemptive behavior, GPU requires novel approaches for scheduling of real-time tasks [27][28][29], preemption [30][31][32], reduction of offloading overheads [33], characterization of contention [34][35][36][37] as well as the computation of worst case execution times [38]. Regarding time determinism in GPU applications, some authors have used the *persistent threads* model of programming, introduced in [39]. This model improves determinism by launching a GPU kernel only once, which remains running during the whole execution of the application. The threads of the persistent kernel spin-wait for work to execute, and the CPU application can send and receive data via memory transfers. This way, the persistent threads model bypasses the traditional launching mechanism. This model has been applied in works like [40] [41] and [42] to improve the performance and time determinism of GPU applications. It also has been applied in works like [43] and [44] for the implementation of alternative GPU schedulers. In [33] the persistent threads model is listed as one of the novel methodologies for achieving predictable behavior when launching GPU tasks.

Other works analyze the necessary properties which need to be taken into account when GPUs are used in the context of critical systems. For example, [45][46] reverse engineered non-obvious aspects of the GPU behavior which need to be taken into account when GPUs are used in real-time systems. Other authors address the compliance of GPUs regarding functional safety certification [47][48][49] by proposing the use of language subsets or the adaptation of safety standards. Regarding industrial applications, some works analyze the challenges of using GPUs in embedded systems [50][51][52], while others analyze the exploitation of GPUs parallelism when executing common control workloads [53][54] or advanced control techniques like predictive control [55] and reinforcement learning-based control [56].

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 [57]. 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.

3 Methodology and Experimental Setup

In this Chapter we describe the methodology used to develop and evaluate the proposals for the thesis, as well as the experimental setup for their evaluation.

3.1 Methodology

The approach we follow in order to develop the contributions of this thesis is based on the iterative methodology shown in Figure 3.1. We start by selecting each of the technologies under analysis in this thesis and then evaluate its properties regarding its suitability for implementing critical real-time systems. With this evaluation, we identify the limitations of the current technology and we propose improvements to overcome those limitations. After applying our proposals, we go through a cyclic refinement process returning to the evaluation phase to verify the impact of the applied improvements. In this process, we discard the solutions that do not have an acceptable impact in the problem and, if necessary, we suggest alternative approaches. Once we have confirmed that the proposed improvements are suitable for overcoming the limitations found in the evaluation phase, we develop a final implementation in the form of a reusable tool.

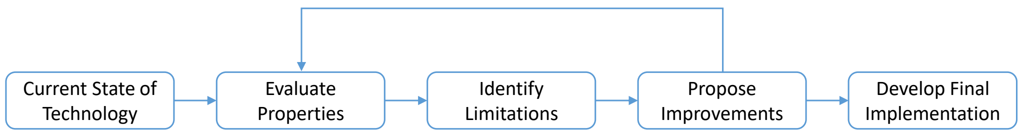


Figure 3.1: Iterative methodology for the development of the thesis

We repeat the same approach for the different contributions in this thesis. The methodology described in Figure 3.1 is a general overview of the process. Each contribution has specific characteristics and scenarios, which will be described thoroughly in the corresponding chapter. In the same way, we use a different experimental setup with a determined toolset for each contribution. The following sections provide a description of the main platforms and tools used to create the experimental setup of the different chapters. Then, on each chapter, we specify which of these platforms and tools are used.

3.2 Embedded GPU Platforms

The contributions presented in this thesis are evaluated in discrete and embedded GPUs from different vendors. However, due to its increasing use in autonomous systems research, we focus mainly on the NVIDIA Jetson family of embedded GPUs for most of our contributions. At the time of writing this thesis, the NVIDIA Jetson family is composed by the Jetson Nano, the Jetson TX2, the Jetson Xavier NX and the Jetson AGX Xavier platforms. Not all platforms are used for all our contributions, since they became available in different points during the evolution of the thesis.

Jetson Nano

The smallest platform of the NVIDIA Jetson family is the Jetson Nano. The architecture of the Jetson Nano SoC is composed by a quad-core 1.43 GHz ARM Cortex-A57 64-bit processor and an integrated NVIDIA Maxwell GPU, which has compute capability 5.3. The four Cortex-A57 cores share a 2 MB L2 cache. The Maxwell GPU contains a single SM with 128 CUDA cores that share a 256 KB L2 cache. The CPU cores and the integrated GPU share 4 GB of 1.6 GHz DRAM memory. Figure 3.2 shows the architecture of the NVIDIA Jetson Nano.

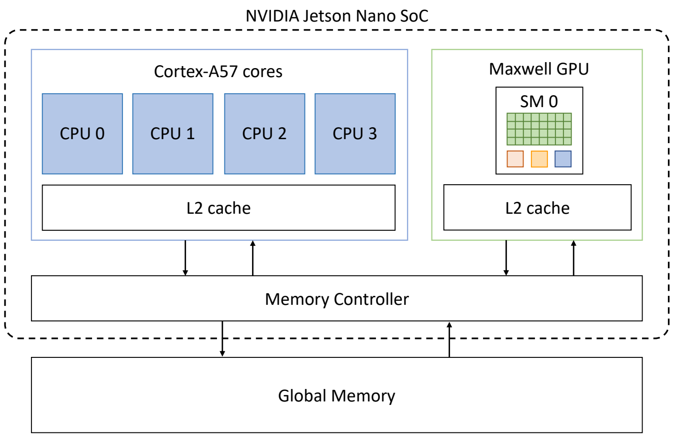


Figure 3.2: Architecture of the NVIDIA Jetson Nano

Jetson TX2

The Jetson TX2 platform incorporates a quad-core 2.0 GHz ARM Cortex-A57 64-bit processor, a dual-core 2.0 GHz superscalar 64-bit NVIDIA Denver processor, and an integrated NVIDIA Pascal GPU, which has compute capability 6.2. There are two 2 MB L2 caches, one shared by the four Cortex-A57 cores and the other one shared by the Denver cores. The Pascal GPU contains 256 CUDA cores divided into two SMs of 128 CUDA cores each. The SMs share a 512 KB L2 cache. The CPU cores and the integrated GPU share 8 GB of 1.866 GHz DRAM memory. Figure 3.3 shows the architecture of the NVIDIA Jetson TX2.

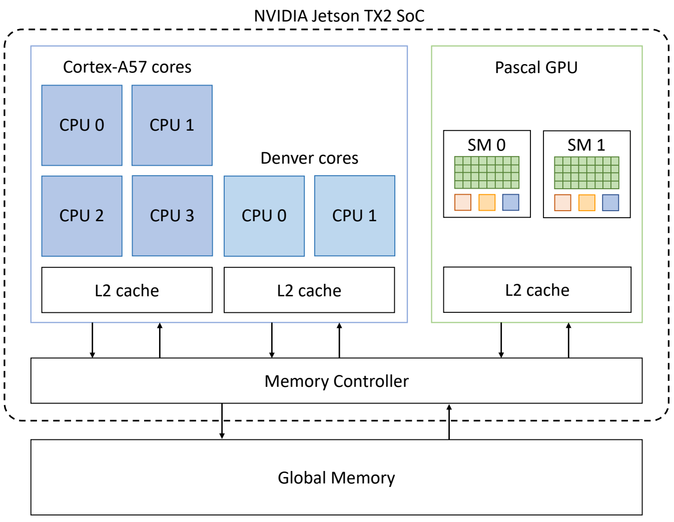


Figure 3.3: Architecture of the NVIDIA Jetson TX2

Jetson Xavier NX

The Jetson Xavier NX platform includes a 6-core 1.4 GHz NVIDIA Carmel 64-bit processor and an integrated NVIDIA Volta GPU, which has compute capability 7.2. The six Carmel cores share a 6 MB L2 cache and a 4 MB L3 cache. The Volta GPU contains 384 CUDA cores and 48 Tensor cores, divided into six SMs of 64 CUDA cores and 8 Tensor cores each. The SMs share a 512 KB L2 cache. The CPU cores and the integrated GPU share 8 GB of 1.866 GHz DRAM memory. Figure 3.4 shows the architecture of the NVIDIA Jetson Xavier NX.

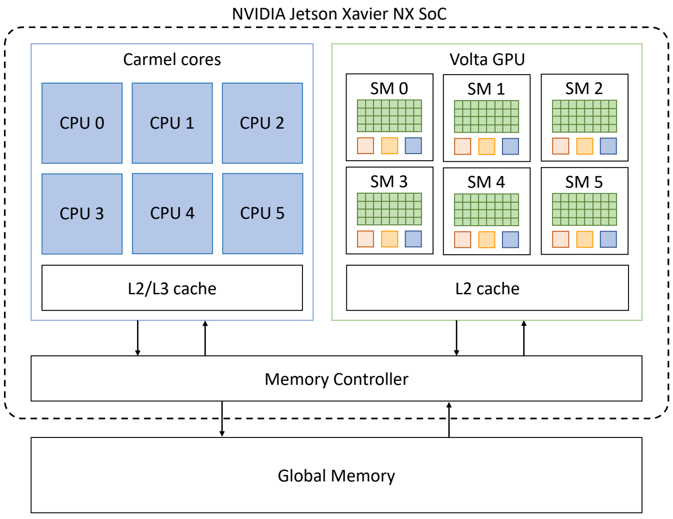


Figure 3.4: Architecture of the NVIDIA Jetson Xavier NX

Jetson AGX Xavier

The Jetson AGX Xavier platform is composed by an 8-core 2.26 GHz NVIDIA Carmel 64-bit processor and an integrated NVIDIA Volta GPU, which has compute capability 7.2. The eight Carmel cores share a 8 MB L2 cache and a 4 MB L3 cache. The Volta GPU contains 512 CUDA cores and 64 Tensor cores, divided into eight SMs of 64 CUDA cores and 8 Tensor cores each. The SMs share a 512 KB L2 cache. The CPU cores and the integrated GPU share 32 GB of 2.133 GHz DRAM memory. Figure 3.5 shows the architecture of the NVIDIA Jetson AGX Xavier.

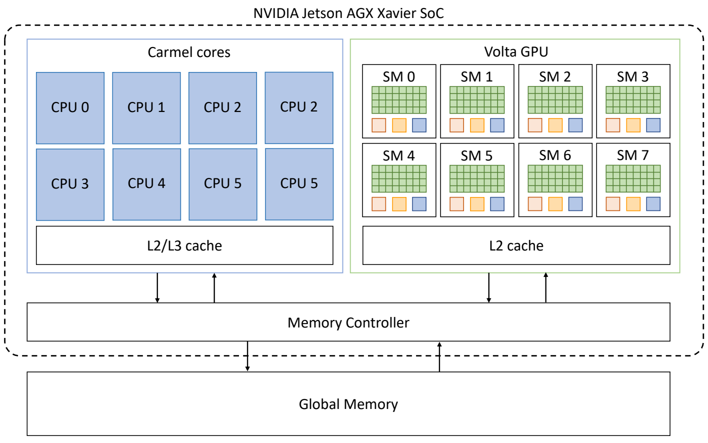


Figure 3.5: Architecture of the NVIDIA Jetson AGX Xavier

3.3 GPU Software Configuration

The board support package provided by NVIDIA for the boards of the Jetson family is the Jetson Linux for Tegra Driver Package (L4T) [58]. This package includes the Linux kernel, a bootloader, NVIDIA drivers, flashing utilities, a sample file system based on Ubuntu 18.04, and tools for GPU software development. During the development of this thesis, we used different versions of L4T up to version 32.5. Independently from the version used, for the contributions of this thesis, we focused on two different base setups for L4T.

Stock System Setup: In this setup we use an off-the-shelf version of L4T, using the standard packages provided by NVIDIA. We use this setup in scenarios where we want to characterize the standard behavior of the Jetson platforms.

Real-time System: For this setup, we compile the Linux kernel applying the PREEMPT-RT real-time patches. Then, we use the flashing tools provided by NVIDIA to install the customized kernel into the Jetson platforms. The PREEMPT-RT patches enable the full preemptive mode of the Linux kernel. We use this setup in scenarios where we want to reduce the latency introduced by the operating system and secondary tasks.

To avoid undesired interference while running experiments on the GPUs, and to improve the overall performance of the system, we apply the next configurations to both setups after installing the system in any of the Jetson platforms:

- Stop and disable the GNOME Display Manager (GDM): L4T is a full Linux distribution which includes the GNOME desktop environment. Since this service may use the GPU for graphics processing, it is important to disable it to avoid interference in experiments where we use the GPU.
- Change the default boot to text mode: By default, L4T boots in graphical mode, which can enable services that use the GPU and cause interference while running experiments. We configure the system to forcefully boot in text mode.
- Set the CPU frequency scaling to performance mode: In Linux, CPU frequency scaling enables the operating system to scale the CPU frequency to save power. Dynamically changing the CPU frequency can have a negative impact on performance. To avoid this, we configure the Linux scaling governor to remain always in performance mode.

3.4 GPU Benchmarks

For the evaluation of the functionality of some of our contributions, in this thesis, we use three benchmark suites that are widely used in the GPU research community: Rodinia [59], Parboil [60] and PolyBench-ACC [61].

The Rodinia benchmark suite was released by the University of Virginia to support the research in the field of parallel computing with accelerators. The suite contains 23 benchmarks of different domains, including medical imaging, bioinformatics, fluid dynamics, data mining and physical simulation. Each benchmark has CUDA, OpenMP and OpenCL implementations. Table 3.1 shows a list of the benchmarks included in the Rodinia benchmark suite.

Table 3.1: List of benchmarks in the Rodinia suite

Benchmark	Description
backprop	Back Propagation machine-learning algorithm
bfs	Breadth-First Search algorithm
b+tree	B+ Tree graph traversal
cfd	CFD solver for unstructured grids
dwt2d	Discrete Wavelet Transform
gaussian	Gaussian elimination for linear systems
heartwall	Tracking movement of a mouse heart in ultrasound images
hotspot	Estimation of processor temperature based on simulation
hotspot3D	Structured grid for physics simulation
huffman	Huffman lossless data compression
hybridsort	Hybrid Sort sorting algorithm
kmeans	K-means clustering algorithm for data-mining
lavaMD	Calculation of particle potential and relocation
leukocyte	Detection and tracking of leukocytes in video frames
lud	LU Decomposition for linear equations
mummergpu	High-throughput parallel pairwise local sequence alignment
myocyte	Model and simulation of the behavior of heart muscle cells
nn	K-nearest Neighbors from an unstructured data set
nw	Needleman-Wunsch method for DNA sequence alignments
particlefilter	Statistical estimator for tracking cells
pathfinder	Find a path with the smallest accumulated weights on 2-D grids
srad	Speckle Reducing Anisotropic Diffusion
streamcluster	Clustering of a stream of input points based on nearest centers

The Parboil benchmark suite was created by the IMPACT Research Group in the University of Illinois at Urbana-Champaign for studying the performance of throughput computing architecture and compilers. The suite is a collection of applications from different fields, including image processing, biomolecular simulation, fluid dynamics, and astronomy. It contains 11 benchmarks in total, with CUDA, OpenMP and OpenCL implementations. Table 3.2 shows a list of the benchmarks included in the Parboil benchmark suite.

Table 3.2: List of benchmarks in the Parboil suite

Benchmark	Description
bfs	Breadth-First Search algorithm
cutcp	Distance-Cutoff Coulombic Potential
histo	Saturating Histogram
lbm	Lattice-Boltzmann Method for Fluid Dynamics
mri-gridding	Magnetic Resonance Imaging - Gridding
mri-q	Magnetic Resonance Imaging - Q
sad	Sum of Absolute Differences
sgemm	Dense matrix-matrix multiply
spmv	Sparse-Matrix Dense-Vector Multiplication
stencil	3-D stencil operation
tpacf	Two Point Angular Correlation Function

The PolyBench-ACC benchmark suite was created in the University of Delaware as a version for accelerators of the PolyBench suite, which was created in the University of California at Los Angeles. The suite is a collection of applications for data mining, linear algebra and stencil computations. It contains 21 benchmarks in total, with implementations in CUDA, OpenCL, OpenMP, OpenACC and HMPP. Table 3.3 shows a list of the benchmarks included in the PolyBench-ACC benchmark suite.

3.5 Model-Based Design Frameworks

As stated in Section 2.2, model-based design is the preferred choice in industry for developing and testing control systems. Although in the market exist different MBD frameworks, in this thesis we focus in Mathworks MATLAB-Simulink, since it is one of the most widely used frameworks in industry. Moreover, currently MATLAB-Simulink is the only industry-ready MBD framework that support custom code generation for GPUs, through its GPU Coder toolbox [62]. GPU Coder is a MATLAB-Simulink toolbox oriented to the generation of optimized CUDA code for NVIDIA GPUs, with special focus on tasks related to deep learning, embedded vision and autonomous systems. During the development of this thesis, we have used different versions of these tools, however, we focus mainly in MATLAB-Simulink version 2021a and GPU Coder version 2.1.

Table 3.3: List of benchmarks in the PolyBench-ACC suite

Benchmark	Description
correlation	Correlation computation
covariance	Covariance computation
2mm	2 matrix multiplications
3mm	3 matrix multiplications
atax	Matrix transpose and vector multiplication
bicg	BiCG sub kernel of BiCGStab linear solver
doitgen	Multi-resolution analysis kernel
gemm	Matrix-multiply
gemver	Vector multiplication and matrix addition
gesummv	Scalar, vector and matrix multiplication
mvt	Matrix vector product and transpose
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
gramschmidt	Gram-Schmidt decomposition
lu	LU decomposition
adi	Alternating Direction Implicit solver
convolution-2d	Convolution with 2-D filter and data
convolution-3d	Convolution with 3-D filter and data
fdtd-2d	2-D Finite Different Time Domain kernel
jacobi-1d	1-D Jacobi stencil computation
jacobi-2d	2-D Jacobi stencil computation

3.6 Compiler Frameworks

One of the main parts of our work consists in the analysis and improvement of automatically generated GPU source code. For this task, we use Clang [63], which is part of the LLVM compiler infrastructure [64]. Clang provides a language front-end and a tooling infrastructure for languages in the C family (C, C++, Objective-C, OpenCL, CUDA) for the LLVM project. Clang is a project under active development, making its API evolve very fast. This can cause the tools based on older versions to become obsolete or incompatible with the newest versions of Clang. For this reason, we use the latest available version for our contributions, which at the time of writing this thesis is Clang 14.0.0.

Among the tools provided by the Clang framework, we use mainly LibTooling and LibASTMatchers. LibTooling is a library aimed to support the writing of standalone tools based on Clang. LibTooling provides mechanisms to ease the development of tools for syntax checking, automatic code formatting, code refactoring, and source-to-source transformation, among others. To do any of these tasks, Clang needs to process the source code of a target application and create an Abstract Syntax Tree (AST) that represents the original source code. The AST must represent all the possible complexity that can appear in the source code without losing any details, for which it can be complex and challenging to work with. To ease this task, LibASTMatchers provides a domain-specific language to locate and manipulate patterns in the AST.

3.7 Other Software Utilities

Given the black-box nature of current commercial GPU platforms, it has been necessary to use reverse engineering techniques to analyze their behavior and uncover properties that are not present in the official documentation. For this task, we use other software utilities that are present in any Linux distribution. First, we use the `strace` utility to capture the system calls generated by GPU applications and get information about the use of system resources. Then, we use the GNU Project Debugger `gdb` to get extra details about resources consumption and to automate the extraction of information through its `.gdbinit` automation script. Finally, we use the library-preloading features of the GNU Dynamic Linker `ld` to inject out analysis functions to GPU applications and get information about resources consumption at run-time.

4 Analysis of Dynamic Memory Allocation in GPUs

Critical real-time systems require strict resource provisioning in terms of memory and timing. The increasing demand for higher performance in these systems has led the industry to recently include GPUs. However, GPU software ecosystems are by their nature closed source, forcing system engineers to consider them as black boxes, complicating resource provisioning.

In this chapter, we expose for the first time the internal resource allocation mechanism of a GPU system. This way, we allow the accurate resource provisioning for a GPU-based critical real-time system. We start by demonstrating the basis of our methodology with a small motivational example. Next, we describe in detail our methodology to discover the properties of the memory allocator used in a GPU-based system. Subsequently, we present the implementation of GPU Memory Allocator Inspector (GMAI), a tool which allows to extract automatically the properties of the memory allocator of a GPU and allows to analyze the memory consumption of GPU applications written in both CUDA and OpenCL. Finally, we present our findings for a wide range of GPUs from different vendors and we use the information obtained from GMAI about the internals of the memory allocator to demonstrate the benefits of accurate resource provisioning with two case studies, showing that the actual memory consumption is significantly higher than the one requested by the software.

4.1 Motivational Example

We start our analysis by showing a motivational example which explains the need for understanding the internals of GPU memory allocation. We execute the instructions shown in Listing 4.1 on a Jetson TX2 platform, and we measure the execution time of the 6 GPU-related calls shown in the listing using `nvprof`, NVIDIA's profiler. For the memory allocations, we use two different types of dynamic memory that can be allocated with CUDA: pinned host memory and device memory.

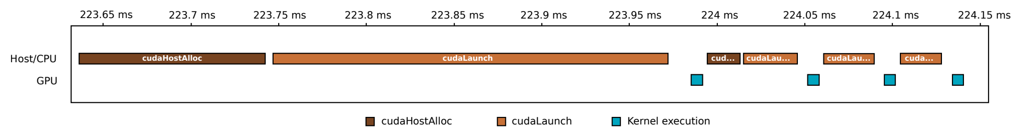


Figure 4.1: Execution times for GPU related calls shown in Listing 4.1 with same size, using pinned allocations.

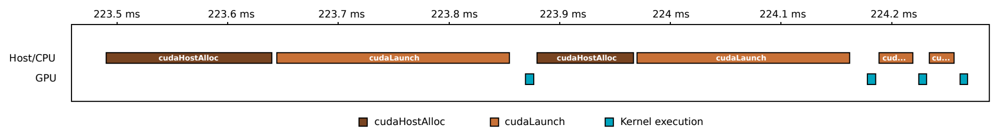


Figure 4.2: Execution times for GPU related calls shown in Listing 4.1 with different size, using pinned allocations.

```
Allocate X bytes;
Launch kernel;
Allocate Y bytes;
Launch kernel;
Launch kernel;
Launch kernel;
```

Listing 4.1: Motivational Example

In Figure 4.1 we show the results of running the example with two pinned memory allocations of the same size (1024 bytes). We notice that the first allocation takes considerable time, while the second one is shorter, and the same happens with the first and second kernel launches. The third and fourth kernel launches are used as a reference to compare the execution times. As shown in the figure, the execution time of the second kernel launch is very similar to the execution time of the third and fourth kernel launches.

However, when we allocate two chunks of memory with different sizes (1024 and 4096 bytes), we notice that both allocations take a similar long time, and the next kernel launch after each allocation takes longer than the third and fourth kernel launches (Figure 4.2). As we can see, the first allocation of a determined size takes longer than the next allocations of the same size. In addition, the execution time of a kernel kernel launch is longer when it is next to the first allocation of a determined size. We notice the same patterns when we use device memory instead of pinned memory, as shown in Figure 4.3 and Figure 4.4.

This observation indicates that the underlying memory allocator implemented in the closed source GPU runtime/driver manages each of the memory allocations

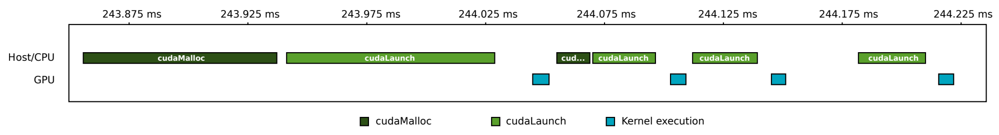


Figure 4.3: Execution times for GPU related calls shown in Listing 4.1 with same size, using device allocations.

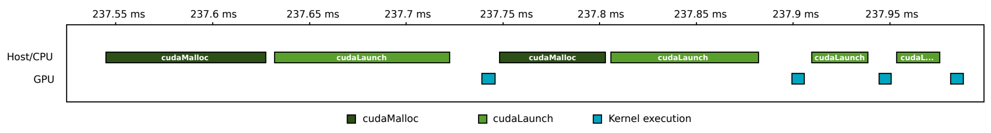


Figure 4.4: Execution times for GPU related calls shown in Listing 4.1 with different size, using device allocations.

of different sizes in a separate way. The question that is raised is following: *can we determine the internals of this memory allocator, so that we can know the exact system memory allocated and predict which of the GPU related calls are expected to take longer?* In the following sections we will introduce our methodology to discover the GPU memory allocator internals in both CUDA and OpenCL devices.

4.2 Background on Memory Allocators

A memory allocator provides memory to a program when requested and takes it back when the program frees it. It also keeps track of the regions of memory that have been assigned and the regions that are free to assign, using an auxiliary data structure. The main goal of an allocator is to do these tasks in the least possible amount of time, while at the same time minimizing memory waste [65].

Initially the memory allocator reserves a contiguous chunk of memory which is used as *pool*, to satisfy dynamic memory requests. When the pool is full, the allocator expands by reserving a new pool. Depending on whether the allocator is implemented in the operating system or at user space, the memory for its pool is reclaimed by using a predefined range of addresses or a preallocated memory region in the former case, or using the `break` or `mmap` system calls in the latter. Custom memory allocators can also use the standard C library calls such `malloc`.

A common challenge for a memory allocator is that programs may free the allocated memory in any order, creating holes between used *blocks*. Note that for efficient representation, block sizes are usually powers of two and they have a minimum *granularity*. The proliferation of small holes leads to the creation of unusable blocks of memory, a problem known as *fragmentation*.

Fragmentation leads to memory waste, incrementing the amount of memory used by the allocator. *External* fragmentation occurs when the available free blocks are too small for the requested size or when the allocator is unable to split bigger blocks to satisfy smaller requests. *Internal* fragmentation occurs when a block larger than needed is assigned, leaving wasted memory inside the block. To avoid fragmentation, techniques like *splitting* free blocks (to satisfy smaller requests) and *coalescing* free blocks (to create larger blocks) are used in conjunction with an allocation policy.

As stated in [65][66][67] there are different policies and mechanisms used by memory allocators to manage memory efficiently:

Sequential fits: memory allocators in this category are based in a single linear list to manage the free blocks of memory. A *best fit* allocator searches the smallest free block in the list large enough to satisfy a request. A *first fit* allocator searches from the beginning of the list and uses the first free block large enough to satisfy the request. A *next fit* allocator begins the search from the last used position. A *worst fit* allocator looks for the largest free block in the list.

Segregated free lists: such memory allocators use an array of free lists, having one list for each block size. When a program requests memory, the allocator uses the list with the smallest block size large enough to satisfy the request. The fit of the allocations is not always perfect because the available block sizes are limited, which causes some internal fragmentation. Some segregated free lists allocators use *size classes* to put together a range of sizes in the same list.

Buddy systems: these allocators allocate memory in fixed block sizes which are split in two parts (or coalesced together) repeatedly to obtain blocks of the requested size. A free block can only be merged with it's *buddy*, so coalescing usually is fast.

Indexed fits: some memory allocators, instead of searching sequentially in a free list, use a more complex indexing data structure like a tree or a hash table to keep track of unallocated blocks. The use of this type of indexed structures leads to faster searches and allocations.

Bitmapped fits: these allocators use a *bitmap* to keep a record of the used areas of the heap. A bitmap is a vector of one-bit flags where each bit represents a word in the heap. The search in a bitmap is slower than in an indexed structure, however, the memory consumption is lower because it does not need to store the size of the blocks.

4.3 Reverse Engineering GPU Memory Allocators

In this section we present our methodology to discover the internals of the GPU memory allocators. Note that we are after obtaining only the key parameters of the memory allocator which affect its memory consumption and timing behavior, but we are not after obtaining every single detail about its design ie. whether its free list is implemented using a list, tree or a bitmap, since such a task may not be entirely possible to achieve or at least not with a reasonable amount of effort. Furthermore, these details do not affect time and space resource provisioning in the same degree to the other parameters.

4.3.1 Reverse Engineering CUDA Memory Allocators

Without loss of generality, in this subsection we focus on the Jetson TX2, the same platform we used for the motivational example. In fact, as we show in Section 4.5, the same methodology is applicable to all NVIDIA GPUs we tried, ranging from old to bleeding edge GPU models. Moreover, since our methodology does not depend on CUDA, it can also be applied to non-NVIDIA GPUs programmed in OpenCL, as we explain in the next subsection.

Starting from the pinned memory scenario, we want to identify the basic design of the memory allocator which is used in order to allocate pinned memory in the CUDA runtime and driver. The fact that the allocation for different sizes results in significantly longer execution times for the first allocation means that the allocator follows a segregated free list design. Therefore, the next step is to identify its size classes as well as the pool size of each free list. In order to achieve our goal, we design carefully crafted memory allocation experiments and observe their behavior to extract the information we are after. In Section 4.4 we present a tool that fully automates our methodology and can be executed in any system featuring a GPU to extract its memory allocator properties.

Pool Size: In order to identify the pool size of each free list, we first create an experiment in which we allocate the minimum amount of memory as shown in Algorithm 1. Since pinned memory has to be requested from the operating system, a user space to kernel space transition based on a system call is required. We monitor

Algorithm 1: Pool size extraction

```
Output: pool_size
1 Allocate 1 byte of pinned memory
2 Capture mmap system call
3 Extract $len$ argument from mmap system call
4 $pool\_size \leftarrow len$
5 Free memory allocated
```

Algorithm 2: Granularity calculation

```
Input: pool_size
Output: granularity
1 Allocate 1 byte of pinned memory
2 $allocations \leftarrow 1$
3 while a new mmap is not generated do
4     Allocate 1 byte of pinned memory
5     $allocations \leftarrow allocations + 1$
6 end while
7 $granularity \leftarrow pool\_size/allocations$
8 Free memory allocated
```

the system calls of the executing process using the `strace` utility, which intercepts the system calls as well as their parameters.

We notice that the memory allocation call generates a `mmap` system call during the allocation process, whose second argument corresponds to the size of the memory pool for the list. In our reference platform, this size is 2 MB.

As a validation, running `strace` on the example of Listing 4.1 reveals a single `mmap` of 2 MB, only on the first allocation of each size. This means that, when the two allocations are for the same size, such as in Figure 4.1, only a pool of 2 MB is created to satisfy both allocations. However, when the allocations have different sizes, which is the case in Figure 4.2 a pool of 2 MB is created for each allocation, which explains the execution time of both memory allocations in that scenario.

Allocation Granularity: Once we know the memory pool size, we need to identify the minimum memory size which corresponds to a single entry within the free list. We achieve this by applying Algorithm 2. The idea is simple: we try to repeatedly allocate the minimum size, until the free list is expanded, by using a new memory pool, which is indicated by a `mmap` call in `strace`. In our reference platform, this happens after 4096 allocations, which means that each allocation reserved a 512 bytes entry within the free list.

Algorithm 3: Size classes extraction

```
Input: granularity
Output: size classes information
1  $inferior\_size \leftarrow granularity$
2  $superior\_size \leftarrow granularity$
3  $size\_class \leftarrow 0$
4  while not all classes extracted do
5      Allocate $inferior\_size$ bytes of pinned memory
6      $size\_class \leftarrow size\_class + 1$
7      while a new mmap is not generated do
8          $superior\_size \leftarrow superior\_size + granularity$
9          Allocate $superior\_size$ bytes of pinned memory
10         Free last allocation
11     end while
12     Save $size\_class$, $inferior\_size$ and $superior\_size - granularity$
13     $inferior\_size \leftarrow superior\_size$
14     Free memory allocated
15 end while
```

Size Classes: Knowing the size of each free list and the allocation granularity, we can focus on detecting how many free lists are kept by the allocator, each corresponding to a different size class. In Algorithm 3 we start creating allocations of increasing sizes, by using the granularity as an increment factor. If a new pool is not created (no new `mmap` is detected), we free the allocation and try the next size. This way we prevent the case that the existing pool used for the current size class is expanded and therefore generating a false positive `mmap`.

In this experiment, we also validate that the pool size and granularity obtained for the first size class using Algorithms 1 and 2 respectively, hold also for each of the other free lists corresponding to the rest of the size classes. However, this validation is not shown in Algorithm 3 for clarity. This is achieved by using the same algorithms, but instead of allocating 1 byte, we allocate the minimum size corresponding to the examined size class. We confirm that in all our experiments, these values are consistent among all the size classes for the examined systems described in Section 4.5.

Allocation Policy: Having obtained all the parameters of the memory allocator, it only remains to identify the policy used in a free list. For this reason, we created validation tests for each type of the four main policies: first fit, best fit, next fit and worst fit. Algorithm 4 shows one these tests checking for the best fit policy. We first create a number of allocations with a decreasing size corresponding to the entire

Algorithm 4: Best fit ascending test

```
   Input: inferior_size, superior_size
   Output: Determines if the policy used is best fit
 1 for $size = superior\_size$ to $inferior\_size$ do
 2     Allocate $size$ bytes of pinned memory
 3 end for
 4 foreach other_allocation do
 5     Store size of $other\_allocation$
 6     Free $other\_allocation$
 7 end foreach
 8 for $size = min\_stored\_size$ to $max\_stored\_size$ do
 9     Allocate $size$ bytes of pinned memory
10 end for
11 Check if all new allocations were assigned using best fit policy
12 Free memory allocated
```

range of allowed sizes for a given size class, so that all allocations are held in the same free list (lines 1-3). Since at this point the free list is empty, each allocation takes the next available free block, resulting in consecutive allocations in the list.

Next, we start freeing every other allocation, creating free blocks of decreasing size and keeping track of their size (lines 4-7). In the final step, we start allocating the same size of blocks that were released in the previous step, but in the reverse order (lines 8-10). That is, each new allocation best fits in the last block of the free list. If the allocator follows a best fit policy, it will result in allocating the same positions as the ones that were freed in the previous step. Otherwise, eg. if the allocator follows a first fit policy, then the allocations would be suboptimal, resulting in an expansion of the original pool.

In order to perform the validation, we use multiple measures. First we use `strace` to validate that there is no expansion of the pool during lines 8-10. Moreover, we keep track of the addresses returned by each and make sure that the new allocations correspond to their best locations, which were their old locations.

Note that the presented example is only one of the variations of the policy validation tests, which are not shown here because they are quite similar. In particular, we have versions which perform the allocations in reverse order, or applying the last step (lines 8-10) in random order, in order to check whether the policy instead of best fit follows a LIFO (Last-In First-Out, stack-like) policy. Another variation of this test uses allocations of the same size, in order to identify what is the allocation policy in the presence of multiple equal size blocks.

Coalescing: In this experiment we perform a series of allocations with arbitrary sizes which however can be rounded up to the same size in a given size class. Next, we create two neighboring free blocks in the middle of the free list. In the following, we allocate a single block with size equal to the addition of the free blocks and we check whether the allocator merges the two free blocks or creates a new allocation in the free list.

Splitting: This experiment is similar to the previous one, with the difference that only one block is freed in the free list. Then a smaller size block is allocated, to check whether the allocator splits the free block to serve the new allocation, or the new allocation takes place elsewhere in the free list.

Expansion Policy: For this experiment, we perform allocations for a given size class, until the pool is expanded one or multiple times. Then we check whether the pool is expanded when it is full – after allocating exactly the same size of allocations with the pool size – or earlier, when an occupancy threshold in the list is exceeded.

Pool Usage: For this experiment, we create multiple pools for a given size class. Then we free a block from the first pool, and perform a new allocation. This way we can check whether the allocation policy is applied across all the pools of the same size, or whether an alternative policy is applied eg. only to the last allocated pool.

Shrinking: Finally, we check whether the memory allocated for expanded memory pools is returned to the system. This is similar to the previous experiment. We perform allocations of the same size class until the memory pool is expanded several times and then we free all the allocations of a given memory pool. We validate whether the memory pool is returned to the system by observing a `munmap` after its last block is freed. Moreover we check whether only a certain memory pool is returned eg. only the last allocated or any of them.

Timing: The methodology we presented so far corresponded to the case of pinned memory and in particular with zero-copy. In this case, in addition to the `mmap` during memory allocation calls, we obtain also `ioctl` system calls during the kernel launches. These system calls are used in order to communicate with device drivers. We observe that in the first kernel execution after a new pool is created for a new size class, the kernel invocation has an extra `ioctl` call. We attribute the longer execution time of these kernels to this additional `ioctl`, which we speculate that is responsible for performing the memory mapping of the host pinned memory to the GPU's Memory Management Unit (MMU).

Device Memory Allocator: To observe the internals of the memory allocator used to allocate device memory we need a slightly different methodology. In particular, the device memory allocations do not require a user-to-kernel switch and therefore its parameters cannot be obtained using `strace`. However, we assume that the same allocator design used for pinned memory for CUDA is also used for device memory within CUDA, in order to reduce development and verification costs. As we comment in the Section 4.5, this assumption is fully validated. Since `strace` is not applicable in this case, the observation of the memory allocator's behavior is applied by instrumenting the code with `gdb` in order to obtain the API call parameters and the returned pointers to the allocated blocks. Also, the timing behavior is observed as previously, using NVIDIA's profiler. With these modifications, we validate that the device memory allocator has the same properties as the pinned memory allocator.

4.3.2 Reverse Engineering OpenCL Memory Allocators

As we have already mentioned, OpenCL follows as similar programming model with CUDA. In this subsection, we adapt the algorithms presented in the previous subsection to the memory allocation calls supported by OpenCL.

A significant difference between OpenCL and CUDA is that each vendor has its own implementation of OpenCL, which can result in different memory allocators for each vendor. For illustration purposes, we have selected a Mali-T860 GPU as OpenCL reference platform. Using this GPU we have applied our reverse engineering methodology trying to extract the same information we extracted from NVIDIA GPUs.

Pool Size: The same way we did with CUDA, we create an experiment in which we allocate the minimum amount of memory as shown in Algorithm 1, intercepting the generated system calls with the `strace` utility. In this case, the memory allocation also generates a `mmap` system call, whose second argument corresponds to the size of the memory pool for the list. In our OpenCL reference platform, this size is 256 KB.

Allocation Granularity: To identify the minimum memory size which corresponds to a single entry within a pool, we also apply Algorithm 2. We create a memory pool and then repeatedly allocate 1 byte of memory until a new `mmap` is generated, which indicates that a new pool has been created. In our OpenCL reference platform, this happens after 4096 allocations. Having a pool size of 256 KB, this means that each allocation reserved 64 bytes within the first memory pool.

Size Classes: Having the pool size and the allocation granularity within each pool, we focus on detecting whether the allocator uses size classes. In Algorithm 5 we start creating an allocation with the minimum size. Then, we create allocations of increasing sizes, by using the granularity as an increment factor, until we reach the pool size. If a new pool is not created (no new `mmap` is generated) it means that all possible sizes within a memory pool are compatible, so size classes are not used.

Algorithm 5: Use of size classes

```
Input: pool_size, granularity
Output: size_classes_used
1  $inferior\_size \leftarrow granularity$
2  $superior\_size \leftarrow granularity$
3  $size\_classes\_used \leftarrow false$
4  Allocate $inferior\_size$ bytes of pinned memory
5  while $superior\_size < pool\_size$ do
6      $superior\_size \leftarrow superior\_size + granularity$
7      Allocate $superior\_size$ bytes of pinned memory
8      if a new mmap is generated then
9          $size\_classes\_used \leftarrow true$
10         Free last allocation
11         break
12     end if
13     Free last allocation
14 end while
15 Free first allocation
```

Allocation Policy: To determine the allocation policy used by the allocator we created validation tests for each type of the four main allocation policies, in a similar way we did with CUDA. First we create a simple test doing several allocations of different sizes and releasing some of them in specific positions. After creating the free spaces, we create a new allocation and check which space is used. This way we deduce the allocation policy used.

To validate the deduced policy we use some tests similar to the one shown in Algorithm 4 or its random variant. The main difference between the CUDA implementation and the OpenCL implementation of these tests is that with CUDA we need to check different size classes.

Coalescing: For this experiment we perform a series of allocations with the same size of the internal granularity of a pool. Next, we free two neighboring allocations to create a continuous free space in the middle of the pool. Then, we perform a new allocation with size equal to the double of the granularity to check whether the allocator merges the free blocks or creates a new allocation in the pool.

Splitting: This experiment is similar to the previous one, with the difference that only one block is freed in the pool. Then a smaller size block is allocated, to check whether the allocator splits the free block or the new allocation takes place elsewhere in the pool.

Expansion Policy: For this experiment we perform several allocations with the same size of the granularity until a new pool is created. Then we check whether the new pool is created when the previous one is full or when an occupancy threshold in the pool is exceeded.

Pool Usage: To determine how the pools are used when there are several pools created, we perform a series of allocations until we have two full pools and a third one with free space. Next, we create a big free space in the first pool and a smaller one in the second pool. Then, we try to make an allocation with the size of the smaller free space to check if the allocator uses the lastly created pool or if it uses the best space available in the previous pools.

Shrinking: Finally, we check how the memory allocated for the pools is returned to the system. In a similar way we did with CUDA, we perform several allocations to create several pools of memory. Then, we free all the allocations and corresponding to a pool and check when is generated the corresponding `munmap` system call. This way we determine if the pools are returned to the system immediately after freeing its last block or if they are returned at the end of the program.

4.4 GMAI: GPU Memory Allocator Inspector

Based on the methodology defined in Section 4.3, we implemented GMAI (GPU Memory Allocator Inspector), which is a tool that can be executed in any system featuring a GPU and extract its memory allocator properties. GMAI consists of two parts: the first part is a set of scripts on which we implement the experiments we defined in Section 4.3 to extract the properties of a GPU memory allocator. The second part is a preload library which can be used to determine the real GPU memory consumption of GPU-based applications. Figure 4.5 shows the GMAI workflow. The source code of GMAI is available at [68].

GMAI can be used in two ways. In the first one we use reverse engineering techniques to extract the memory allocator properties of a target GPU. With this information we generate a configuration file which can be used to visualize the extracted properties. This information could later be used by an engineer to manually

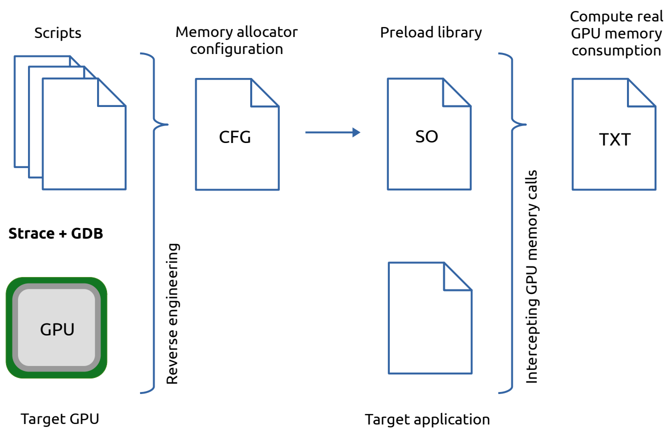

Scripts
Memory allocator configuration
Preload library
Compute real GPU memory consumption
CFG
SO
TXT
Strace + GDB
GPU
Reverse engineering
Intercepting GPU memory calls
Target GPU
Target application

Figure 4.5: GPU Memory Allocator Inspector (GMAI) workflow

analyze the GPU memory consumption of GPU-based applications. However, the second way of using this tool consists on a preload library which can be used to automatically compute the real GPU memory consumption of a target application based on the memory allocator properties stored in the configuration file.

For the implementation of GMAI we have used different debugging techniques. For the initial analysis of the system calls generated by a GPU application we have used the `strace` utility. This way we know what are the functions and parameters we have to look at to get the information we are after for. Using this information, we have used `gdb` to execute our experiments and extract the values we needed in each of them. Having a functional set of `gdb` commands to extract the information for all the experiments, we have automated the debugging process using the `gdbinit` file. This way we automatically generate the configuration file by executing the necessary `gdb` commands over our experiments.

For intercepting the GPU memory allocation calls of a GPU application, we have used the preloading technique, which is a feature of `ld`, the dynamic linker in UNIX-like systems. With preloading we can override arbitrary function calls in a program, by using the environment variable `LD_PRELOAD` to specify a library with wrappers for the functions we are interested in. With this technique we implemented our own versions of the GPU functions used to allocate memory in both CUDA and OpenCL. In our function wrappers we keep track of their parameters before calling the actual functions. This way we can intercept those functions from a GPU running application and use their parameters by combining them with the properties of the GPU memory allocator stored in the configuration file to compute the application's real GPU memory consumption. An important detail to take into account is that, even though our solution could be adapted to be used in systems where multiple CPUs

Table 4.1: Tested NVIDIA GPU Platforms

Device Name	Compute Capability	Runtime/ Driver	Kernel Version	GPU Type
GeForce 9300M GS	1.1	6.5	3.19.0	Discrete
Quadro FX 3700	1.1	6.5	3.12.9	Discrete
GeForce GTX 960M	5.0	10.0	4.15.0	Discrete
GeForce GTX 1050 Ti	6.1	9.2	4.15.0	Discrete
GeForce GTX 1080 Ti	6.1	9.2	4.15.0	Discrete
V100-PCIE-16GB7.0	7.0	10.0	4.15.0	Discrete
Tesla T4	7.5	10.0	4.15.0	Discrete
Tegra X1 (Nano)	5.3	10.0	4.9.140	Integrated
Tegra X2 (TX2)	6.2	9.0	4.4.38	Integrated
Xavier	7.2	10.0	4.9.108	Integrated

share the same GPU, or feature more than one GPU, in the current implementation we assume that only one GPU is being analyzed at a time.

4.5 Results

4.5.1 Obtained Properties of CUDA Allocators

In this subsection, we provide the results we have obtained using our methodology on a wide range of NVIDIA GPUs, ranging from very old products with compute capability 1.1 to the latest NVIDIA embedded SoCs Nano and Xavier, as shown in Table 4.1.

As explained in the previous section, we have implemented our methodology in the GMAI tool which automates completely the process. Once GMAI is executed, in a few seconds a report is generated with the information about the memory allocator. In the Listing 4.2 we can see the generated report about the NVIDIA TX2 platform, which we used in the discussion of the previous sections. In Listing 4.3 we can see the generated report about the NVIDIA GeForce GTX 1080Ti, which we use as a discrete GPU reference platform.

```
Device name: NVIDIA Tegra X2
Compute capability: 6.2
CUDA runtime version: 9.0
CUDA driver version: 9.0

Pool size: 2097152 bytes
Granularity: 512 bytes

Size classes
1: 1   to 2    blocks of 512 bytes [1      to 1024 bytes   ]
2: 3   to 8    blocks of 512 bytes [1025   to 4096 bytes   ]
3: 9   to 32   blocks of 512 bytes [4097   to 16384 bytes  ]
4: 33  to 128  blocks of 512 bytes [16385  to 65536 bytes  ]
5: 129 to 512  blocks of 512 bytes [65537  to 262144 bytes ]
6: 513 to 3583 blocks of 512 bytes [262145 to 1834496 bytes]
Larger allocations: mmap size next 4096 bytes multiple

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full
Pool usage: Last created
Shrinking support: Yes. Any pool deleted
```

Listing 4.2: NVIDIA TX2 memory allocator report

```
Device name: GeForce GTX 1080 Ti
Compute capability: 6.1
CUDA runtime version: 9.2
CUDA driver version: 9.2

Pool size: 2097152 bytes
Granularity: 512 bytes

Size classes
1: 1   to 2    blocks of 512 bytes [1      to 1024 bytes   ]
2: 3   to 8    blocks of 512 bytes [1025   to 4096 bytes   ]
3: 9   to 32   blocks of 512 bytes [4097   to 16384 bytes  ]
4: 33  to 128  blocks of 512 bytes [16385  to 65536 bytes  ]
5: 129 to 512  blocks of 512 bytes [65537  to 262144 bytes ]
6: 513 to 3583 blocks of 512 bytes [262145 to 1834496 bytes]
Larger allocations: mmap size next 4096 bytes multiple

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full
Pool usage: Last created
Shrinking support: Yes. Any pool deleted
```

Listing 4.3: GeForce GTX 1080 Ti memory allocator report

Table 4.2: Tested OpenCL GPU Platforms

Device Name	Vendor	Architecture	OpenCL Version	Kernel Version	GPU Type
Mali-T860	ARM	Midgard	1.2	4.4.154	Integrated
Mali-G72	ARM	Bifrost	2.0	4.9.78	Integrated
GeForce GTX 1050 Ti	NVIDIA	Pascal	1.2	4.15.0	Discrete
GeForce GTX 1080 Ti	NVIDIA	Pascal	1.2	4.15.0	Discrete

In both GPUs, we observe that the pool size is 2 MB and the minimum allocation granularity is 512 bytes. The allocator is using 6 size classes, with the last one ranging up to the pool size. Larger allocations are always rounded up to the next 4 KB multiple, which is the system's page size. The allocator is implementing a Segregated Free Lists Allocator with best fit policy. In the event of expansion, the allocator is keeping a stack of pools. Deallocations can happen to any of the pools, however new allocations are only allocated in the last created pool. Finally, the allocator frees the memory used by any pool when all its blocks are freed.

Regardless of the version of the driver or the hardware, we obtained exactly the same results with the NVIDIA GPUs GTX 1050 Ti, Tesla V100, Tesla T4 and Xavier. For the GPUs GeForce GTX 960M and TX1 Nano we also obtained identical results but with the pool size being 1 MB. For the older NVIDIA GPUs, Quadro FX 3700 and GeForce 9300M GS we obtained a pool size of 1 MB and a granularity of 256 bytes.

4.5.2 Obtained Properties of OpenCL Allocators

In this subsection, we provide the results we have obtained applying our methodology to some OpenCL compatible GPUs, which are shown in Table 4.2. Our main reference platforms for this subsection are the ARM Mali GPUs since they are the first non-NVIDIA GPUs we analyze. We include two OpenCL compatible NVIDIA GPUs for comparison purposes.

As we did with CUDA, we also automated our methodology to extract the information about the memory allocators of OpenCL GPUs by incorporating it in the GMAI tool. Listing 4.4 and Listing 4.5 show the generated report for Mali-T860 and Mali-G72 GPUs respectively.

We observe that the only difference between these ARM GPUs is the granularity of the memory allocator, which is 64 bytes in the Mali-T860 GPU and 128 bytes in the

```
Device name: Mali-T860
OpenCL driver version: 1.2

Pool size: 262144 bytes
Granularity: 64 bytes

Size classes: Not used
Large allocations: mmap size next 4096 bytes multiple

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full
Pool usage: Best available
Shrinking support: No. Pools deleted at the end
```

Listing 4.4: Mali-T860 OpenCL memory allocator report

```
Device name: Mali-G72
OpenCL driver version: 2.0

Pool size: 262144 bytes
Granularity: 128 bytes

Size classes: Not used
Large allocations: mmap size next 4096 bytes multiple

Allocator policy: Best fit
Coalescing support: Yes
Splitting support: Yes
Expansion policy: When full
Pool usage: Best available
Shrinking support: No. Pools deleted at the end
```

Listing 4.5: Mali-G72 OpenCL memory allocator report

Mali-G72 GPU. The pool size is 256 KB in both GPUs and the allocator does not use size classes. This is a significant difference between the NVIDIA memory allocators we examined in the previous subsection. We speculate that this decision is related to the smaller size of memory available in these platforms (4 GB) and therefore the memory allocator this way wastes less memory due to internal fragmentation.

Allocations larger than the pool size are rounded to the next 4 KB multiple, which is the page size. The allocator implements a best fit policy and supports coalescing and splitting of free blocks. The allocator expands creating new pools when there is no enough space on the previous pools. When there are multiple pools with free space, the allocator applies the best fit policy across the pools. However, even when the pools are created consecutively in memory, the allocator does not use a free region shared by two pools to satisfy the space required by a new allocation. Another difference we observed compared to the NVIDIA allocators, is that in NVIDIA GPUs only the last created pool is used for new allocations, even when there is free space in other pools. Finally, we found out that the OpenCL allocator on ARM GPUs does not free the memory used by a pool when all its blocks are freed, unlike the CUDA allocators. Instead, the memory is released at the end of the program. However, we observed that those regions of memory are reused for the creation of new pools.

We also tested GMAI on two OpenCL compatible NVIDIA GPUs (GTX 1050 Ti and GTX 1080 Ti) which we analyzed in the previous subsection. Repeating these experiments with OpenCL we got the same results shown in Listing 4.3. This means that the NVIDIA OpenCL implementation internally uses the same memory allocator used by CUDA. When doing these tests, we also observed that in Mali GPUs the memory is reserved when we create the corresponding `cl_mem` object. However, in the NVIDIA implementation, the memory pools are created until we map a region of a `cl_mem` object.

4.5.3 Exploiting the Knowledge of GPU Allocators in Automotive Case Studies' Resource Provisioning

The ultimate purpose of exposing the internals of the GPU allocators is this knowledge to be leveraged to compute precisely the amount of memory used by real-time applications. This will be essential when GPUs will be incorporated in avionics and automotive real-time operating systems. Moreover, in current general purpose operating systems, it allows to make sure that the system can safely accommodate the memory and timing requirements of the application, without the use of unpredictable swap memory.

In order to demonstrate these benefits, we apply our knowledge on two automotive case studies used in modern vehicles' environment perception: a sobel filter for edge detection and a pedestrian detection task [69]. The former, edge detection, is very common in both Advanced Driving Assistance Systems (ADAS) and autonomous driving for numerous tasks such as lane departure [70], sign [71] and car detection [72]. Pedestrian detection is also used for ADAS, eg. automated breaking as well as for autonomous driving.

As we described in Section 4.4 when we execute GMAI on a given platform, it generates a configuration file with the properties of the memory allocator. At

Table 4.3: GPU memory allocations in Edge Detection Task

Variable	Type	Size (bytes)
Input Image (640×480)	int8 RGB	921600
Filter Kernel (3×3)	int8	9
Output Image (640×480)	int8	307200
Total (bytes):		1228809

Table 4.4: GPU memory allocator usage in Edge Detection (NVIDIA TX2)

Variable	Size Class	Size (bytes)	Occupied 512b Blocks	Occupied Size (bytes)
Input Image	6	921600	1800	921600
Filter Kernel	1	9	1	512
Output Image	6	307200	600	307200
Total (bytes):				1229312

runtime, we execute the GPU program with the GMAI preload library which exposes the GPU memory allocation API calls. This way GMAI intercepts all memory requests and their sizes, and based on the configuration file, it provides details about the actual memory consumption of the allocator, which we present in the results of the two case studies next.

Edge Detection

The edge detection task explicitly allocates dynamic memory for the variables shown in Table 4.3, which are needed for the GPU implementation of the sobel filter. We notice that the input is a 3-component (RGB) image 640×480 and a 3×3 filter kernel, while the output is a single component 640×480 image, containing the detected edges. Without knowing the internals of the GPU memory allocator, when the task is executed on a platform with zero-copy pinned memory a system engineer might provision 1228809 bytes memory consumption.

Edge detection allocations with CUDA: Table 4.4 shows the actual memory used by the memory allocator when the edge detection algorithm is executed on the NVIDIA TX2 platform with zero-copy allocations. We notice that we have allocations from two different size classes. This means that two memory pools are created, with 2 MB each. Each of these creations will increase the execution time of two memory allocation calls, the first ones corresponding to these size classes, as well as the execution time of the first kernel invocation following these allocations.

Therefore, the total memory consumption to be provisioned is 4 MB for this platform and configuration, which is 3.4$\times$ more than it was expected, due to internal fragmentation. The memory allocator, however, is only using a fraction of that. In the first free list, the 3$\times$3 kernel is occupying a single block of 512 bytes instead of 9 bytes due to the minimum block granularity, while in the other free list 1228800 bytes are occupied compared to the 2 MB of the pool, resulting in 58% free list occupancy.

On the other hand, in the NVIDIA Jetson Nano platform, each memory pool occupies 1 MB. However, the two images exceed the memory pool size for size class 6, requiring the memory pool to expand. Therefore the allocator uses 3 MB for its pools, which is 2.6$\times$ larger that the memory explicitly allocated by the application. In older NVIDIA GPUs like the GeForce 9300M GS, the figures are almost identical, with the difference of the block size of 256, which slightly changes the occupied size in the pool for the filter kernel. Table 4.5 shows a summary of this analysis.

Table 4.5: Real GPU memory usage in Edge Detection Task (NVIDIA GPUs)

Sample GPU	Pool size	Required pools	Real GPU memory used
TX2	2 MB	2	4 MB
Nano	1 MB	3	3 MB

If the application is configured to either use non-mapped pinned memory or pageable memory for the host side, and device memory for the GPU side, the above numbers are also correct. The only difference is that in these cases both CPU and GPU memory is used, which doubles the aggregate memory consumption[1].

Edge detection allocations with OpenCL: For the OpenCL scenario, Table 4.6 shows the memory used by the memory allocator in an ARM Mali-T860 GPU. In this platform, the memory is allocated in pools of 256 KB with a granularity of 64 bytes. However, when an allocation is larger than 256 KB, the pool size is the next multiple of 4 KB. This means that for the input image the allocator will create a pool of 925696 bytes, using 921600 bytes and leaving 4096 bytes free. For the filter, we only need a block of 64 bytes which can be allocated in the free space of the previously created pool. For the output image the allocator will create a pool of 311296 bytes, which is the next 4 KB multiple.

[1]In fact, in the case of CPU pageable memory, the memory consumption is closer to the explicitly allocated memory using `malloc`, since the GNU memory allocator [73] only uses 8 byte aligned blocks in 32-bit platforms and 16 byte aligned blocks in 64 bit ones and it does not use segregated lists. Moreover, the memory pool in CPU is lazily allocated, which means that the OS only reserves the pages of the heap which have been accessed. However, considering equal CPU and GPU memory consumption simplifies the CPU side memory analysis and provides a safe upper bound for a safety critical system in which lazy allocation is not used.

Table 4.6: GPU memory allocator usage in Edge Detection (ARM Mali-T860)

Variable	Size (bytes)	Occupied 64b Blocks	Occupied Size (bytes)
Input Image	921600	14400	921600
Filter Kernel	9	1	64
Output Image	307200	4800	307200
Total (bytes):			1228864

The total memory consumption to be provisioned is 1236992 bytes, which is 8183 bytes more than the memory explicitly allocated by the application. However, it only represents a 0.67% of increment. Using a Mali-G72 GPU the only difference is that for the filter the allocator will reserve a block of 128 bytes, since this is the granularity in this platform. However, the results will be the same because in this case the filter can also be allocated in the free space of the pool created for the input image.

Pedestrian Detection

This application is significantly more complex than the previous task and it is obtained from the open source implementation of the benchmark described in [69]. In addition to the input and output images, this task uses a complex dynamically allocated cascade classifier structure. This structure consists of numerous smaller dynamically allocated structures with sizes ranging from 32 bytes to 84 bytes arranged in arrays, requiring a total of 7534 dynamic memory allocations. The order in which the memory for these structures is allocated is shown in Listing 4.6.

The different dynamic GPU allocations of the application are summarized in Table 4.7. Without knowing the internals of the GPU allocator, a system engineer would provision 1484912 bytes, out of which 870512 correspond to the structure of the classifier.

Pedestrian detection allocations with CUDA: Table 4.8 shows the actual memory consumption within the memory allocator when the pedestrian detection algorithm is executed on the NVIDIA TX2 platform with zero-copy allocations. Again, we notice that the allocations are rounded up to 512 byte multiples, since this is the minimum allocation granularity in the allocator, which penalizes small allocations. In this task, 3 different size classes are used.

```
N_MAX_STAGES = 30;
N_MAX_CLASSIFIERS = 250;
Allocate sizeof(Struct_A) = 32 bytes;
Allocate sizeof(Struct_B) = 480 bytes;
for i = 1 to N_MAX_STAGES do
    Allocate sizeof(Struct_C) = 8000 bytes;
    for j = 1 to N_MAX_CLASSIFIERS do
        Allocate sizeof(Struct_D) = 84 bytes;
    end for
end for
Allocate 307200 bytes for input_image;
Allocate 307200 bytes for output_image;
```

Listing 4.6: Pseudocode of memory allocations in the pedestrian detection case study

Table 4.7: GPU memory allocations in Pedestrian Detection

Variable	Allocations	Individual Size (bytes)	Total Size (bytes)
Input Image (640×480)	1	307200	307200
Output Image (640×480)	1	307200	307200
Classifier			
Struct A	1	32	32
Struct B (30×16 array)	1	480	480
Struct C (250×32 array)	30	8000	240000
Struct D	7500	84	630000
Total:	7534		1484912

In platforms like the NVIDIA TX2 where the size of memory pools is 2 MB, a single pool is enough for size classes 3 and 6. However, for size class 1, the total size exceeds 2 MB, which requires the free list to expand to accommodate the total of 3841024 bytes required for this size class. Therefore, the allocator uses 8 MB in total, which is 5.6× more than the initially provisioned memory.

For platforms like the Nano with 1 MB pool size, again the size classes 3 and 6 can use a single pool, while the size class 1 requires 4 pools. Therefore, the total consumption of the allocator is 6 MB, 4.2× bigger than the memory explicitly requested by the application. Table 4.9 shows a summary of this analysis.

Pedestrian detection allocations with OpenCL: For the OpenCL scenario, Table 4.10 shows the memory used by the memory allocator in an ARM Mali-T860 GPU. As shown in Listing 4.4, this allocator does not use size classes, which means that the order in which the allocations are made will dictate the order in which the memory pools will be created and used. Listing 4.6 shows the order in which the allocations are made. As mentioned earlier, this allocator creates pools of 256 KB

Table 4.8: GPU memory allocator usage in Pedestrian Detection Task (NVIDIA TX2)

Variable	Size Class	Individual Size (bytes)	Occupied Size (bytes)	Allocations	Total Size (bytes)
Input Image	6	307200	307200	1	307200
Output Image	6	307200	307200	1	307200
Classifier					
Struct A	1	32	512	1	512
Struct B	1	480	512	1	512
Struct C	3	8000	8192	30	245760
Struct D	1	84	512	7500	3840000
Total:				7534	4701184

Table 4.9: Real GPU memory usage in Pedestrian Detection Task (NVIDIA GPUs)

Sample GPU	Pool size	Required pools	Real GPU memory used
TX2	2 MB	4	8 MB
Nano	1 MB	6	6 MB

with a granularity of 64 bytes. For Struct A, the allocator will create a new pool of 256 KB and will use only a block of 64 bytes. For Struct B the allocator will use 512 bytes of the previously created pool, leaving 261568 bytes free. On each iteration of the outer loop the allocator will allocate 8000 bytes for Struct C and 32000 bytes for Struct D. The free space in the first pool will be enough for the allocations of the first 6 iterations. To allocate the 960000 bytes required by the other 24 iterations, the allocator will create 4 extra pools of 256 KB. Finally, for each image, the allocator will create a pool of 311296 bytes.

The total memory consumption to be provisioned is 1933312 bytes, which is 1.3× larger than the memory explicitly allocated by the application. Using a Mali-G72 GPU, which has a granularity of 128 bytes, the difference is that the allocator will reserve 128 bytes for Struct A and 8064 bytes for each allocation of Struct C. However, even with these differences, 5 pools of 256 KB are enough to allocate all the structures. For this reason, the memory consumption will be the same.

We notice that in both case studies, the amount of extra memory allocated due to the internal fragmentation in the ARM OpenCL memory allocators is lower than the one in NVIDIA platforms when it is compared to the amount of memory requested by the programmer. This difference comes from the fact that the ARM implementations do not use size classes in their memory allocators and use smaller pool size and granularity. For this reason, we speculate that this has been a design choice due to the limited amount of memory present in these devices.

Table 4.10: GPU memory allocator usage in Pedestrian Detection Task (Mali-T860)

Variable	Individual Size (bytes)	Occupied Size (bytes)	Allocations	Total Size (bytes)
Input Image	307200	307200	1	307200
Output Image	307200	307200	1	307200
Classifier				
Struct A	32	64	1	64
Struct B	480	512	1	512
Struct C	8000	8000	30	240000
Struct D	84	128	7500	960000
Total:			7534	1814976

4.6 Related Work

In this section, we present some previous works in the literature similar to our work. We can categorize these works in articles related to resource allocation and reverse engineering techniques in GPUs and CPU memory allocators.

GPU Memory Allocators. Multi-core memory allocators like the one proposed by Berger et al. [74], have been shown not to scale well with many-core architectures like GPUs. For this reason, some authors have approached the GPU resource management topic by creating custom memory allocators suited for many-core architectures.

Huang et al. [75, 76] proposed *XMalloc*, a memory allocator based in two techniques: allocation coalescing (aggregation of memory allocation requests from SIMD-parallel threads to be handled by the CUDA allocator) and buffering of freed blocks for faster reuse using parallel queues. Results on a NVIDIA G480 GPU showed that *XMalloc* magnified the CUDA allocator throughput by a factor of 48.

Steinberger et al. [77] showed that traditional memory allocation strategies used by CPUs are not suited for the use on GPUs and proposed *ScatterAlloc*. This allocator reduces collisions by scattering memory requests using hashing. Experimental results showed that *ScatterAlloc* was about 100 times faster than the CUDA allocator and up to 10 times faster than *XMalloc*.

Widmer et al. [78] proposed *FDGMalloc*, which makes use of the SIMD parallelism present in GPUs to significantly speed-up the allocation of dynamic memory. The authors compared their implementation with the CUDA allocator and with *ScatterAlloc*, achieving a speed-up of several orders of magnitude.

A common characteristic in all these works is that they focus their analysis in comparing the performance of their allocators with the performance of the CUDA allocator, without trying to understand its internal structure or the way it works as we do in this chapter. Moreover, these works obtain their memory through the CUDA memory allocator, so they are still susceptible to the timing effects of its usage.

Reverse Engineering Works on GPUs. The black box nature of the GPUs has led to the creation of some research works oriented to the use of reverse engineering techniques to get information about their internal characteristics.

Wong et al. [79] developed a microbenchmark suite to measure various undisclosed characteristics of the processing elements and memory hierarchies of a NVIDIA GTX280 GPU. Their results validated some of the hardware characteristics publicly available and revealed some other undocumented hardware structures used for control flow and caching. Following a similar approach, Mei et al. [80] exposed previously unknown characteristics about the memory hierarchy of Fermi, Kepler and Maxwell NVIDIA GPUs.

Amert et al. [45] applied black-box experimentation to a NVIDIA TX2 GPU. Based on the results, they defined a set of rules describing the behavior of the NVIDIA TX2 scheduler. The same group later extended their work on software and disclosed a set of non-obvious pitfalls to avoid when using CUDA-enabled GPUs for safety-critical systems [46].

All these works are based in applying reverse engineering techniques to hardware or software of GPUs, however, none of them is oriented to get information about the memory allocation system and leverage it, which is the focus of our study.

Reverse Engineering Memory Allocators. Eventhough memory allocation is an extensively researched area, the only work to our knowledge related to reverse engineering memory allocators is the *MemBrush* tool, proposed by Chen et al. [81]. The purpose of *MemBrush* is to detect the API functions of custom memory allocators in stripped binaries. *MemBrush* has been used to improve other reverse engineering tools like *Howard* [82], which is used to extract data structures from C binaries without having any symbol tables.

To the best of our knowledge, our work is the first one oriented to extract information (real memory usage, size classes and allocation policy) about a closed source GPU memory allocator and to analyze the benefits of this information for real-time systems.

4.7 Summary

In this chapter, we presented a methodology and an automated tool to extract information about the internals of the GPU memory allocators. We applied our methodology to a wide range of GPUs from different vendors, supporting both CUDA and OpenCL. We identified that there is only a slight difference between different CUDA GPUs, in the amount of memory used internally as a pool and the granularity, in particular in older GPUs. On the other hand, we found out that OpenCL memory allocators do not use different size classes, but they serve all their memory allocations from a single size class.

We have presented GMAI (GPU Memory Allocation Inspector) which allows the extraction of the memory allocator properties in an automatic way and based on this information it enables the computation of the actual memory consumption of GPU applications.

We have applied GMAI in two automotive case studies, showing how a system engineer can be benefited by this information, in order to provision the correct amount of memory. In particular we have shown that the actual memory consumption of the memory allocator can be up to an order of magnitude higher than the amount requested by the application, by running our tests in several GPUs from various vendors.

5 Characterization of Dynamic Memory Usage in GPU Applications

In the previous chapter, we showed how dynamic memory allocation can have a negative impact on the execution time of GPU kernels due to the overhead added by the memory allocation system when creating new memory pools. We reverse-engineered the GPU memory allocator and extracted its internal properties to understand under which conditions memory pools are created and to know the real amount of memory used in their creation. Then we presented GMAI, a tool that automates the extraction of the memory allocator properties. GMAI provides a preload library that captures the GPU memory function calls of a target application and extracts data to calculate its actual GPU memory consumption based on the GPU memory allocator properties. Since dynamic memory allocation can have a negative impact on the execution time of GPU applications, we can further extend the functionality of the preload library to capture the behavior of all GPU dynamic memory operations and identify memory allocation patterns that can harm the execution time.

In this chapter, we propose extending GMAI to characterize the way dynamic memory is allocated and deallocated in real GPU applications. We present a memory characterization library that captures the dynamic memory operations of a target GPU application and extracts data to generate a memory characterization report and an interactive plot. Then, we use our library to characterize three popular GPU benchmark suites and we identify memory allocation patterns that could be modified to improve performance and memory consumption in embedded GPU platforms. It is worth noting that while the motivation of our work was initiated by its applicability in the safety critical domain, the contribution of this chapter extends beyond the domain of real-time and critical systems. Understanding the dynamic GPU memory allocation patterns and GPU memory usage of real-life applications is interesting for research in GPU programming in general such as in High Performance Computing or programming models. To our knowledge, this is the first work analyzing the memory behavior of GPU benchmarking suites, which is as important as their performance characterization, which has been studied extensively in the literature.

5.1 Design of the Memory Characterization Library

The main objective of this work is to extend the functionality provided by GMAI to characterize the use of dynamic memory in GPU applications. Our goal is to create a tool to generate the memory characterization of a GPU application showing the evolution of the different types of dynamic memory used in the application and the location of the related function calls in the source code. This allows users to observe the dynamic memory behavior of a target GPU application and trace back interesting function calls to their location in the source code. To implement this functionality, we need to capture all GPU-related dynamic memory calls and extract data related to the requested memory size, assigned addresses, and location of the function calls in the source code.

GPU applications can use different types of dynamic memory. For each type of memory, there are specific functions we need to capture. For device memory, we must keep track of `cudaMalloc` and `cudaFree` calls. For pinned memory, we must capture calls to `cudaMallocHost`, `cudaHostAlloc` and `cudaFreeHost`. In the host side, GPU applications can also use dynamic pageable memory, which usually is allocated with `malloc` and deallocated with `free`. However, the allocation of pageable memory can also be done with other functions such as `calloc`, so we must keep track of those function calls as well.

The use of pageable memory (i.e. dynamic memory allocated by `malloc`) in GPU applications is not exclusive for GPU tasks, so keeping track of all pageable memory operations can add noise and unnecessary overhead to our analysis. We are interested in tracking only the subset of pageable memory allocations that are used to transfer data between host and GPU. This way, the characterization could be used to identify possible memory optimizations for embedded GPU platforms, where a pageable memory allocation and a device memory allocation can be replaced by a single zero-copy or unified memory allocation. Then, besides capturing the pageable allocations, we must also keep track of the `cudaMemcpy` function calls to identify which pageable allocations are used for GPU data transfers.

To capture the target memory functions, we use a technique known in the literature as *function interposition*, which consists in replacing the target functions with user-defined wrapper functions. These wrapper functions can then be used to add extra functionality to the original functions, in our case, to extract information from the arguments. In UNIX-like systems, function interposition can be done at run time using the environment variable `LD_PRELOAD` [83] that we used in GMAI, at load time using the `--wrap` argument of the `ld` linker [84], or at compile time, replacing the target function calls at the source code level.

Since GMAI uses `LD_PRELOAD` to implement function interposition at run time, our first approach was to extend the GMAI preload library to add the memory characterization functionality. However, with this approach, the extraction of source code information is not optimal. To implement it, we must assume that the target application has been compiled with debugging information. Then, for each function call, we must extract the debugging symbols from the compiled binary. We created a prototype using this approach, and we found that the extraction of source information added much processing overhead on function calls. Moreover, while this approach works well with CUDA functions, for more generic functions like `malloc` it generates recursion problems since `malloc` is used internally by functions like `dlsym`, which we use inside the wrapper functions to invoke the real functions.

Due to the limitations of the `LD_PRELOAD` approach, we opted to create a new library implementing function interposition at compile time. With this approach, the target function calls are replaced in the source code using `#define` directives. One of the advantages of this approach is that the extraction of source code information is easier to implement. In C/C++, we have access to the `__FILE__` and `__LINE__` macros which expand to the full path of the current file and the current line number, respectively [85]. In our library, these macros are added as extra arguments into the wrapper functions to get the source information of every function call. Listing 5.1 shows some examples.

```
#define malloc(size) chr_malloc(size,__FILE__,__LINE__)
#define free(ptr) chr_free(ptr,__FILE__,__LINE__)
#define cudaMalloc(devPtr,size) chr_cudaMalloc(devPtr,size,__FILE__,__LINE__)
#define cudaFree(devPtr) chr_cudaFree(devPtr,__FILE__,__LINE__)
```

Listing 5.1: Examples of function interposition at compile time

We defined the replacements required for each target function in a header file of the memory characterization library. In order to replace the original function calls with the wrapper functions, this header file must be included in the source files of the target application. When the application is compiled, the memory characterization functionality is added to the function calls. Figure 5.1 shows the entire workflow of the GPU memory characterization library.

The memory characterization library uses as input the configuration file generated by GMAI. From this file, we get information such as the pool size and the size classes used by the GPU memory allocators. Using these values, we keep track of the

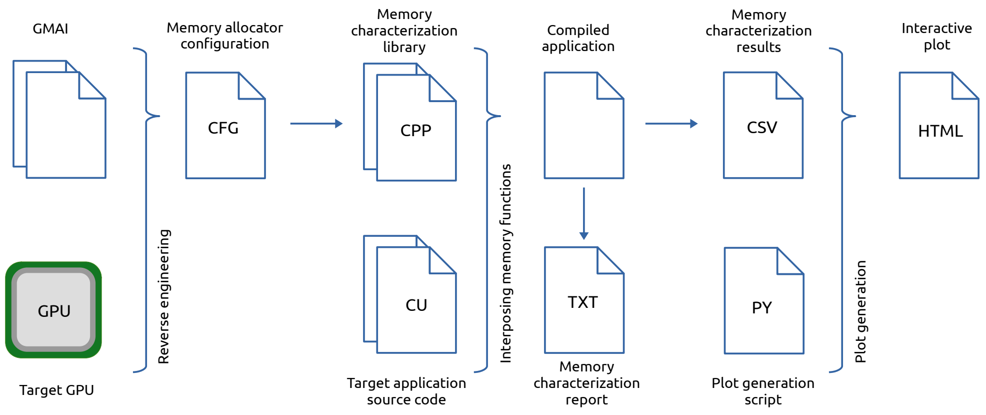

GMAI
Memory allocator configuration
Memory characterization library
Compiled application
Memory characterization results
Interactive plot
CFG
CPP
CSV
HTML
GPU
Target GPU
Reverse engineering
CU
Target application source code
Interposing memory functions
TXT
Memory characterization report
PY
Plot generation script
Plot generation

Figure 5.1: GPU memory characterization workflow

creation of memory pools and we calculate the real amount of device memory and pinned memory reserved by the CUDA allocators.

To create the memory characterization of a target application, first we must include the library header in the source files we want to track. This way we replace the target function calls at the source code level. Then, the target application must be compiled together with the memory characterization library to add the characterization functionality.

When the compiled application is executed, each wrapper function gathers information about its corresponding memory operation. At the end of the execution, the memory characterization library creates a report with information such as the maximum amount of dynamic memory used by the application, the number of memory pools created, the number of memory transfers, and the detected memory leaks. The library also creates a `csv` file with information about the use of the different types of memory in the target application.

For the final part of the workflow, we created a script that transforms the information stored in the `csv` file into an interactive memory characterization plot, like the one shown in Figure 5.2.

The interactive memory characterization plot shows the evolution of pageable host memory, pinned host memory, and device memory through the execution of the application. For device and pinned memory, it shows the values requested by the user as *Device memory (user)* and *Pinned memory (user)* respectively.It also shows the real amounts of memory reserved by the CUDA memory allocators as *Device memory (allocator)* and *Pinned memory (allocator)*, based on the information extracted with GMAI. At each point, the *Total memory* is calculated as the sum of pageable host

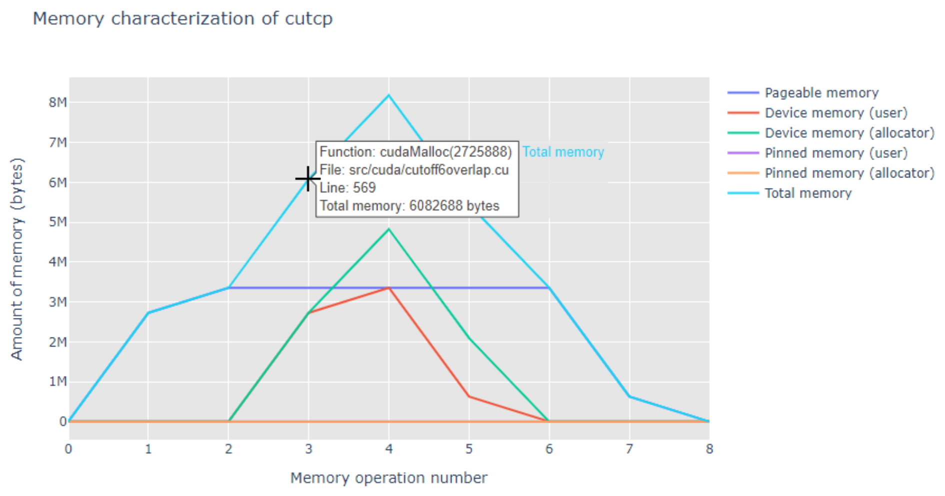


Figure 5.2: Example of an interactive memory characterization plot

memory plus the amount of device memory and pinned memory reserved by the CUDA allocators. For each memory operation, the plot shows the current state of all types of memory and the information of each function call. As an example, Figure 5.2 shows that the third memory operation of the *cutcp* benchmark was a `cudaMalloc` call to reserve 2725888 bytes of device memory, the function call is at line 569 of the cutoff6overlap.cu source file, and the total amount of GPU-related dynamic memory used at this point of the program execution is 6082688 bytes.

5.2 Evaluation

In this section, we provide the results of the memory characterization of the benchmarks of three popular GPU benchmark suites: Rodinia [59], Parboil [60] and PolyBench-ACC [61] performed using the tool we have developed. To carry out the evaluation, we compiled all the benchmarks and executed them on a Jetson TX2 platform.

5.2.1 Memory Characterization of Rodinia Benchmarks

The Rodinia benchmark suite contains 23 benchmarks with CUDA, OpenMP and OpenCL implementations. Due to compilation issues, we do not include the *mummergpu* benchmark in the evaluation. The *cfd*, *particlefilter* and *srad* benchmarks have multiple versions, but we consider that selecting one version of each of these benchmarks is enough for characterization purposes. For *cfd* we selected the *euler3d* version, for *particlefilter* we selected the *particlefilter_float* version, and for *srad* we selected the *srad_v1* version.

To carry out the evaluation, we compiled the CUDA version of the benchmarks including our memory characterization library in the compilation process. Then, we executed the resulting applications to generate the memory characterization reports and we generated the corresponding interactive memory characterization plots. Table 5.1 shows a summary of the information obtained in the memory characterization reports.

Table 5.1: Rodinia benchmarks characterization results

Benchmark	Max. memory used (bytes)	Pools created	H2D transfers	D2H transfers	Leaked (bytes)
backprop	20455632	3	5	3	0
bfs	79202120	5	18	13	2097152
b+tree	84762324	5	15	3	0
cfd	17094144	6	3	1	0
dwt2d	36700160	12	1	3	0
gaussian	2097296	1	3	3	0
heartwall	46035288	21	24	4	0
hotspot	7340032	2	2	1	2097152
hotspot3D	50331648	3	2	1	0
huffman	10487808	5	4	2	0
hybridsort	90218496	20	12	12	0
kmeans	205642200	3	5	2	680
lavaMD	16359296	3	4	1	0
leukocyte	3779072	6	32	7	0
lud	2359296	1	1	1	0
myocyte	3517200	1	7800	7800	1419600
nn	2268208	1	1	1	0
nw	83976204	2	2	1	0
particlefilter	2305544	1	6	3	8552
pathfinder	82097280	2	2	1	0
srad	9315952	4	5	201	0
streamcluster	147656756	524	3223	3222	2097152

As we can see in the results, there are some benchmarks with interesting characteristics. First, we detected memory leaks in the *bfs*, *hotspot*, *kmeans*, *myocyte*, *particlefilter* and *streamcluster* benchmarks. We validated these results by manually verifying the source lines of the memory leaks indicated in the reports. In *bfs*, the memory leak corresponds to 1 byte of device memory not being released, causing a whole pool of 2 MB to remain reserved. In *hotspot*, the memory leak is caused by two allocations of 1 MB of pageable host memory, which are not released in the end. In *kmeans*, the memory leak corresponds to a non-released pageable host allocation of 680 bytes. In *myocyte*, there is an allocation used to create an array of pointers, and then, for each position in the array, there is an allocation of 364 bytes of pageable host memory. In the end, the memory allocated for the array of pointers is released, but the memory reserved for each position of the array remains reserved. The corresponding code is executed several times, causing an accumulation of 1419600 bytes of leaked host memory. In *particlefilter*, the memory leaks are caused by two allocations of pageable host memory (8000 bytes and 552 bytes) that are not released in the end. Finally, in *streamcluster*, the memory leak is caused by a non-released allocation of 2 MB of pageable host memory.

Regarding memory transfers, Table 5.1 shows the number of host-to-device and device-to-host memory transfers (H2D and D2H respectively) for each benchmark. The benchmarks with the highest number of memory transfers are *myocyte*, *srad* and *streamcluster*. When executing applications like these benchmarks on embedded GPU platforms, replacing the pageable allocations and their corresponding device allocations with zero-copy allocations could help to improve performance, since all related memory transfers could be avoided.

In terms of memory consumption, Table 5.1 shows the peak of maximum dynamic memory used for each benchmark. The *kmeans* and *streamcluster* benchmarks are the ones with more memory consumption. In the case of *streamcluster*, the allocation activity is more intensive, creating device memory pools up to 524 times. The *maximum memory used* value is composed of different types of dynamic memory. To better understand its composition, Figure 5.3 shows the percentage corresponding to each type of memory in the maximum peak value for all the benchmarks.

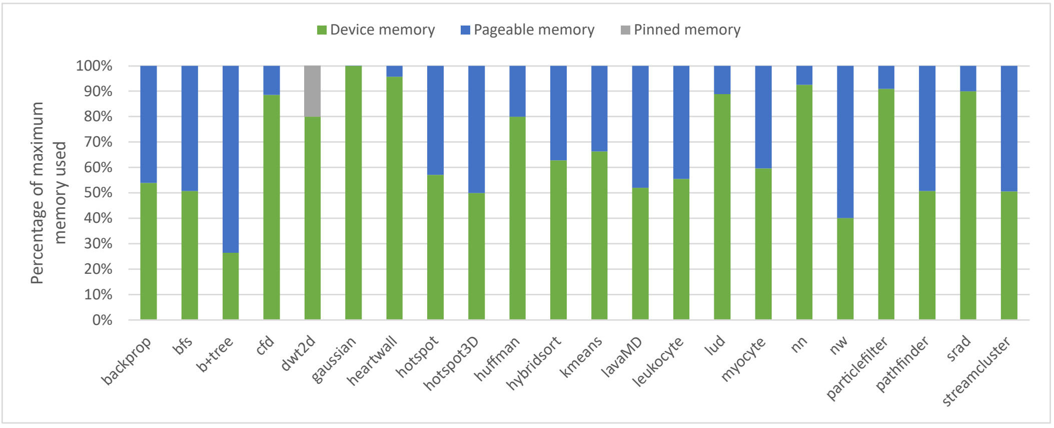


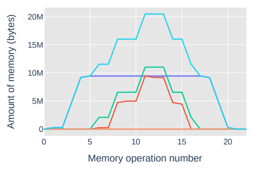
Figure 5.3: Rodinia benchmarks distribution of maximum total memory used

As we can see, in almost all the benchmarks the peak is composed only by pageable host memory and device memory. The only benchmark using pinned host memory is *dwt2d*. There are some benchmarks like *backprop*, *bfs*, *hotspot*, *hotspot3D*, *lavaMD*, *leukocyte*, *nw*, *pathfinder*, and *streamcluster* where the memory is almost evenly distributed between host memory and device memory. When executing applications like these benchmarks on embedded GPU platforms, replacing the pageable allocations and their corresponding device allocations with zero-copy allocations could help to reduce the memory consumption approximately to the half.

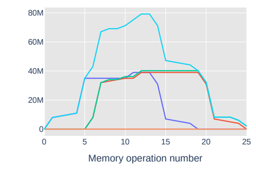
In applications like *cfd*, *dwt2d*, *gaussian*, *heartwall*, *huffman*, *lud*, *myocyte*, *nn*, *particlefilter* and *srad*, where the amount of device memory is significantly larger than the amount of host memory, the replacement with zero-copy allocations will not represent a big difference in memory consumption. However, in cases like *srad*, it could be still beneficial to make the replacement to avoid the high amount of memory transfers. It is important to note that, in *gaussian*, *lud*, *myocyte*, *nn* and *particlefilter*, the unbalance between host and device memory is due to the pool size used by the CUDA allocator when reserving device memory. For example, in the *gaussian* benchmark, the user allocates 144 bytes of host memory and 144 bytes of device memory. However, to allocate 144 bytes of device memory, the allocator reserves 2 MB, which is the pool size on the Jetson TX2 platform as shown in Listing 4.2. For this reason, the percentage of maximum used memory corresponding to device memory in the *gaussian* benchmark is 99.9%, as shown in Figure 5.3. In *myocyte* the unbalance may not be very evident because most of the reported pageable host memory corresponds to the accumulation of leaked memory.

To analyze the behavior of Rodinia benchmarks in terms of how memory is allocated and deallocated, Figure 5.4 shows the memory characterization plots for all the benchmarks. In typical GPU applications, memory for different variables is allocated gradually, presenting an increasing behavior until reaching the peak of maximum memory used by the application. Then, these allocations are gradually released after use, presenting a decreasing behavior until the total memory used is zero. In this category we have benchmarks like *backprop*, *bfs*, *gaussian*, *hotspot*, *hotspot3D*, *kmeans*, *lavaMD*, *lud*, *nn*, *nw*, *particlefilter*, *pathfinder* and *srad*. This is the desired behavior for most GPU applications. We just need to make sure that all allocations are grouped at the beginning of the application and that all deallocations are done in the end. This way, we can avoid the creation of memory pools in the middle of the execution, which can have a negative timing impact when launching GPU kernels as we have demonstrated in the previous Chapter.

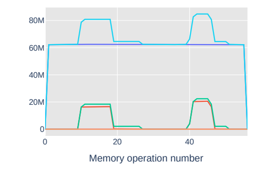
There are other applications like *b+tree*, *cfd* and *huffman* which have two or more peaks of memory allocation with different shapes. These peaks usually



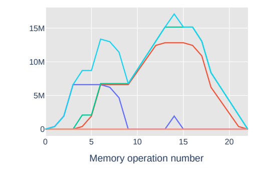
(a) backprop



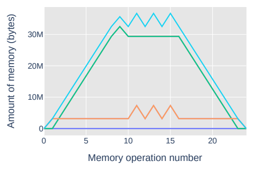
(b) bfs



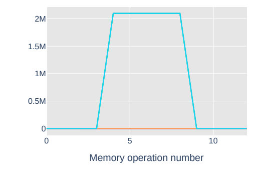
(c) b+tree



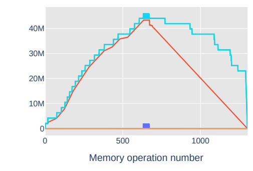
(d) cfd



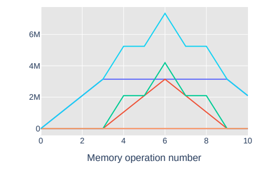
(e) dwt2d



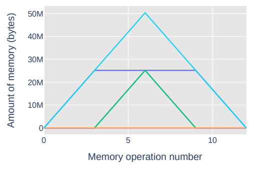
(f) gaussian



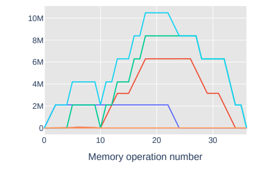
(g) heartwall



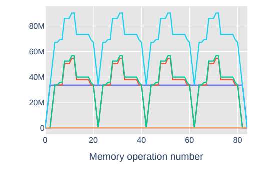
(h) hotspot



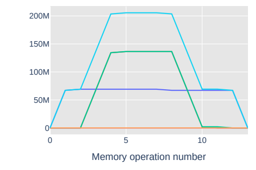
(i) hotspot3D



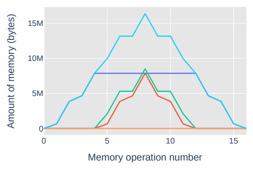
(j) huffman



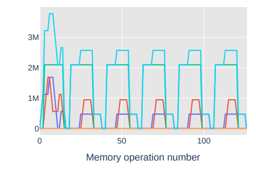
(k) hybridsort



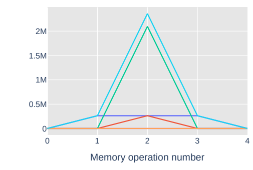
(l) kmeans



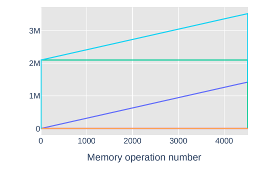
(m) lavaMD



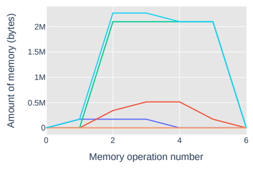
(n) leukocyte



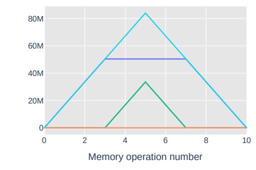
(o) lud



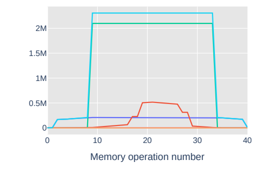
(p) myocyte



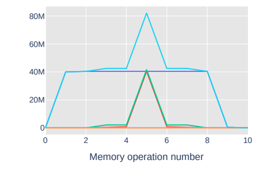
(q) nn



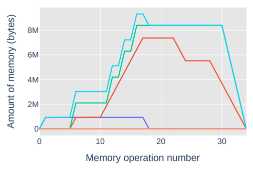
(r) nw



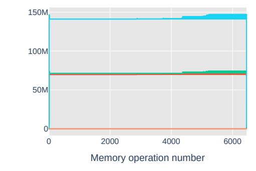
(s) particlefilter



(t) pathfinder



(u) srad



(v) streamcluster

Figure 5.4: Memory characterization of Rodinia benchmarks

correspond to different processing phases, which translates into the launching of different GPU kernels. In this scenario, there is a set of variables allocated at the beginning and deallocated at the end of each phase. Modifying the memory allocation patterns in applications like these can represent a trade-off between performance and memory consumption. Suppose we allocate all the memory at the beginning of the application, in that case, the creation of memory pools will not affect the intermediate kernel launches. However, all the memory needed for the different kernels will be reserved for the entire duration of the application.

Finally, there are applications like *dwt2d*, *heartwall*, *hybridsort*, *leukocyte*, *myocyte* and *streamcluster* which have multiple uniform memory allocation peaks. This behavior usually corresponds to iterative blocks of code with allocations and deallocations inside a loop. It is common in applications which execute the same GPU kernel several times, like applications processing images or frames in a video. In the context of this thesis, this behavior could also be present in control applications since control algorithms are executed iteratively. The problem with applications like these is not the iterative behavior, but to include the allocations and deallocations inside the processing loop. The reason is that for each allocation and deallocation the application needs to interact with the CUDA runtime, which adds processing overhead on each iteration. Moreover, some allocations can trigger the creation of new memory pools, which can add timing overhead on the execution time of the next GPU kernel launch. This behavior can be improved by taking the allocations and deallocations out of the processing loop and reusing the same variables for all the iterations, allocating memory before entering the loop and deallocating after the last iteration.

5.2.2 Memory Characterization of Parboil Benchmarks

The Parboil benchmark suite contains 11 benchmarks in total, with CUDA, OpenMP and OpenCL implementations. These benchmarks are a collection of throughput computing applications from different fields, including image processing, biomolecular simulation, fluid dynamics, and astronomy. We compiled the CUDA version of all the benchmarks including our memory characterization library in the compilation process. Then, we executed the resulting applications to generate the memory characterization information. Table 5.2 shows a summary of the information obtained in the memory characterization reports.

As we can see in the results, *bfs* is the only benchmark where we detected memory leaks. The memory leaks in this benchmark correspond to three device allocations of 4 bytes and two device allocations of 4000000 bytes which are not released in the end. To serve the three allocations of 4 bytes, the CUDA allocator creates a pool of 2 MB, which is the pool size in the reference platform. For each allocation of 4000000 bytes, the allocator reserves 4001792 bytes, which is the next multiple of the page size in the reference platform. Adding these values, we get a total of 10100736 bytes of device memory leaked.

Table 5.2: Parboil benchmarks characterization results

Benchmark	Max. memory used (bytes)	Pools created	H2D transfers	D2H transfers	Leaked (bytes)
bfs	137983504	7	14	10	10100736
cutcp	8179840	2	1	1	0
histo	272595200	7	1060	21	0
lbm	776601600	2	2	1	0
mri-gridding	599351296	15	1	4	0
mri-q	2146304	2	8	3	0
sad	19945776	2	1	1	0
sgemm	2097152	1	2	1	0
spmv	2146448	1	4	1	0
stencil	8388608	2	1	1	0
tpacf	2719556	1	1	1	0

Regarding memory transfers, *histo* is the benchmark with the highest activity, with a total of 1081 memory transfers between CPU and GPU. The rest of the benchmarks present a significantly lower number of memory transfers, being *bfs* in second place with only 24 memory transfers in total.

In terms of memory consumption, *lbm*, *mri-gridding*, *histo* and *bfs* are the benchmarks that use more dynamic memory. Figure 5.5 shows the distribution of the different types of dynamic memory at the point of maximum use for all the Parboil benchmarks.

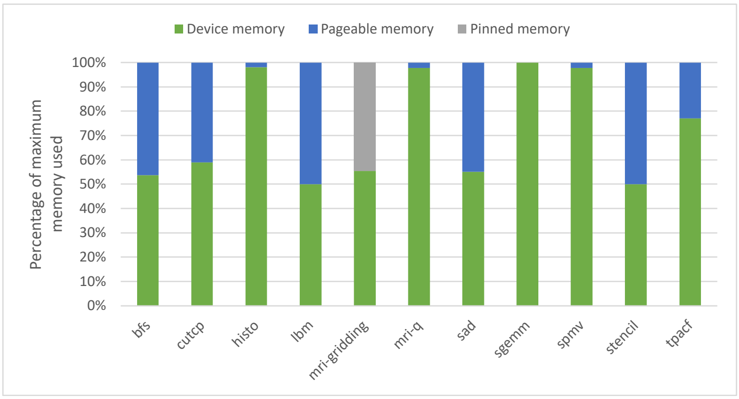
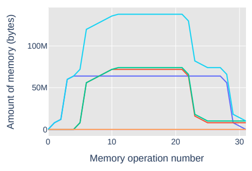


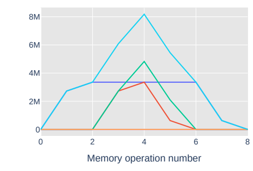
Figure 5.5: Parboil benchmarks distribution of maximum total memory used

Similar to what we observed in Rodinia, in Parboil, almost all the benchmarks use only pageable host memory and device memory. The only benchmark using pinned host memory is *mri-gridding*. In the *bfs*, *cutcp*, *lbm*, *sad* and *stencil* benchmarks, the memory is almost evenly distributed between host memory and device memory, which makes them good candidates for replacing their pageable and device allocations with zero-copy allocations to reduce the memory consumption in embedded GPU platforms. In applications like *histo*, *mri-q*, *sgemm* and *spmv* where the amount of device memory is significantly larger than the amount of host memory, replacing all the allocations with zero-copy allocations will not improve memory consumption. However, in benchmarks like *histo* with a high number of memory transfers between CPU and GPU, the use of zero-copy allocations could still improve the execution time.

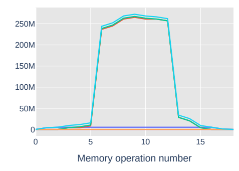
To analyze the behavior of Parboil benchmarks in terms of how memory is allocated and deallocated, Figure 5.6 shows the memory characterization plots for all the benchmarks. The *bfs*, *cutcp*, *histo*, *sad*, *sgemm*, *spmv*, *stencil* and *tpacf* benchmarks show a one-peak behavior, meaning that the dynamic memory is incrementally allocated, and then deallocated after use. The *lbm*, *mri-gridding* and *mri-q* benchmarks present two peaks of memory allocation, probably corresponding to two different processing phases in the GPU. None of the Parboil benchmarks present a behavior with multiple uniform memory allocation peaks, meaning that, if there are cyclic processing phases in any of the benchmarks, the allocations and deallocations are done correctly outside of the processing loop.



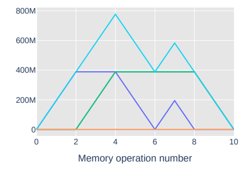
(a) bfs



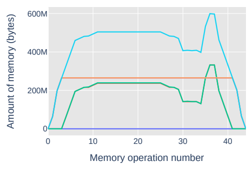
(b) cutcp



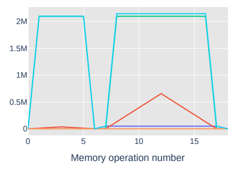
(c) histo



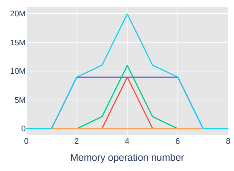
(d) lbm



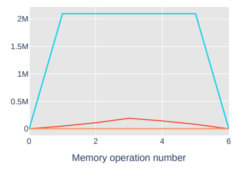
(e) mri-gridding



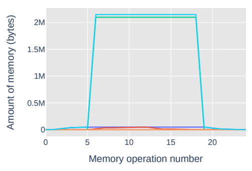
(f) mri-q



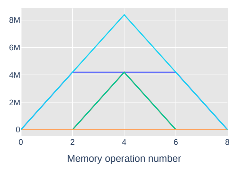
(g) sad



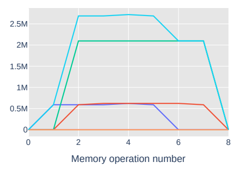
(h) sgemm



(i) spmv



(j) stencil



(k) tpacf

Figure 5.6: Memory characterization of Parboil benchmarks

5.2.3 Memory Characterization of PolyBench-ACC Benchmarks

The PolyBench-ACC benchmark suite contains 21 benchmarks in total, with implementations in CUDA, OpenCL, OpenMP, OpenACC and HMPP. We compiled the CUDA version of all the benchmarks including our memory characterization library in the compilation process. Then, we executed the resulting applications to generate the memory characterization information. Table 5.3 shows a summary of the information obtained in the memory characterization reports.

Table 5.3: PolyBench-ACC benchmarks characterization results

Benchmark	Max. memory used (bytes)	Pools created	H2D transfers	D2H transfers	Leaked (bytes)
correlation	85999616	3	5	1	0
covariance	85991424	3	3	1	0
2mm	46137344	5	5	1	0
3mm	16777216	4	7	1	0
atax	136380416	2	4	1	0
bicg	136413184	2	5	2	0
doitgen	35717120	3	3	1	0
gemm	8388608	2	3	1	0
gemver	136462336	2	9	1	0
gesummv	270598144	3	5	1	136314880
mvt	136413184	2	5	2	0
syr2k	29360128	3	3	1	0
syrk	20971520	2	2	1	0
gramschmidt	83886080	3	1	1	0
lu	50331648	1	1	1	0
adi	33554432	3	3	2	0
convolution-2d	268435456	2	1	1	0
convolution-3d	335544320	2	2	1	0
fdtd-2d	119539664	4	4	1	0
jacobi-1d	2162688	1	2	2	0
jacobi-2d	25165824	2	2	2	0

As we can see in the results, *gesummv* is the only benchmark on which we detected memory leaks. The memory leak can clearly be observed at the end of the memory characterization plot in Figure 5.8j, where the final values of the *Device memory (allocator)* and *Total memory* lines are not zero. We reviewed the memory characterization report and determined that the memory leak corresponds to three device memory allocations of 16384 bytes and two device memory allocations of 67108864 bytes that are not released at the end. The total 136314880 bytes of leaked device memory is composed by the addition of the two allocations of

67108864 bytes plus 2 MB, which is the size of the pool created by the CUDA allocator to serve the three allocations of 16384 bytes.

Regarding memory transfers between CPU and GPU, all the PolyBench-ACC benchmarks present a low number of memory transfers, being 11 the maximum number of total memory transfers (in *3mm* and *gemver* benchmarks). Likewise, the number of created memory pools is low in all the benchmarks.

In terms of memory consumption, Table 5.3 shows the maximum amount of dynamic memory used for each benchmark, having *gesummv*, *convolution-2d* and *convolution-3d* as the benchmarks with higher memory use. Figure 5.7 shows the percentages corresponding to each type of memory in the point of maximum memory use for all the benchmarks.

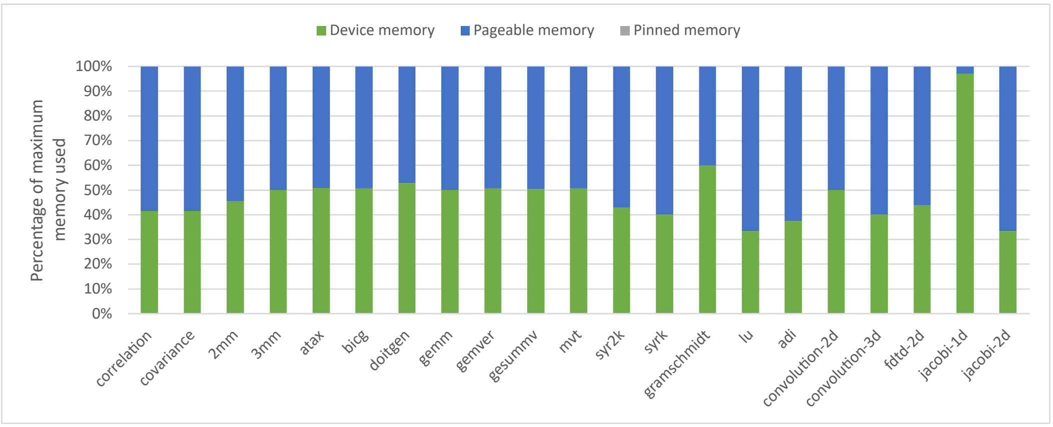
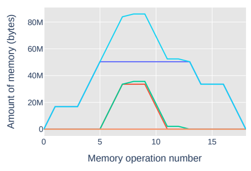


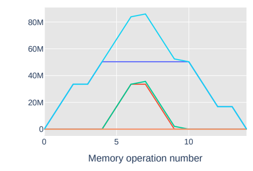
Figure 5.7: PolyBench-ACC benchmarks distribution of maximum total memory used

As shown in the Figure, in all the PolyBench-ACC benchmarks, the point of maximum memory use is composed only by pageable host memory and device memory. Furthermore, in almost all the benchmarks, the memory is evenly distributed between these two types of memory. As previously stated, when executing applications like these on embedded GPU platforms, the use of dynamic memory can be reduced approximately to the half if every pageable host allocation and its corresponding device allocation are replaced by a single zero-copy allocation. The only benchmark that could not benefit from zero-copy replacements is *jacobi-1d*, because, in this benchmark, the amount of device memory is significantly higher than the amount of pageable host memory.

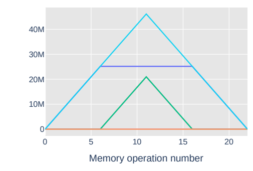
To analyze the behavior of PolyBench-ACC benchmarks in terms of how memory is allocated and deallocated, Figure 5.8 shows the memory characterization plots for all the benchmarks. As we can see in the Figure, some benchmarks have similar behavior with other benchmarks. For example, *correlation*, *covariance*, *3mm*, *doitgen*,



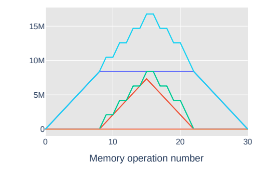
(a) correlation



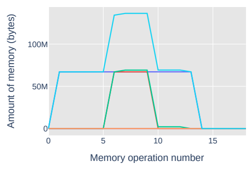
(b) covariance



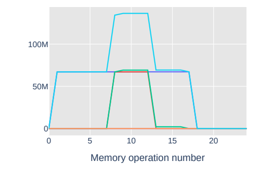
(c) 2mm



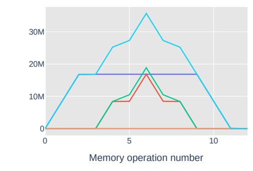
(d) 3mm



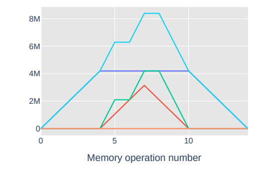
(e) atax



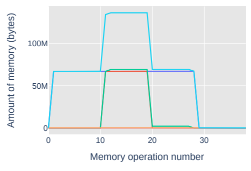
(f) bicg



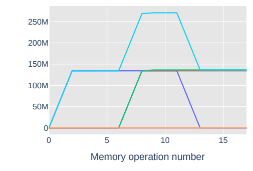
(g) doitgen



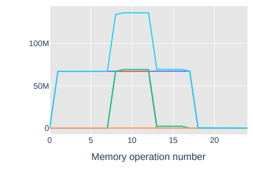
(h) gemm



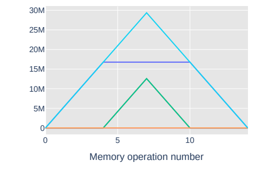
(i) gemver



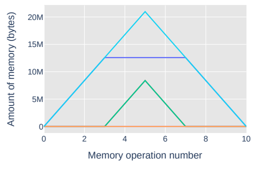
(j) gesummv



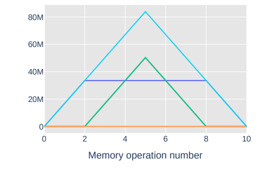
(k) mvt



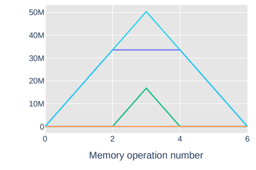
(l) syr2k



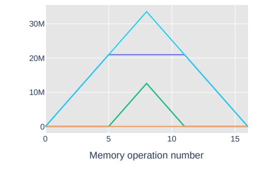
(m) syrk



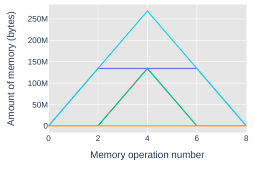
(n) gramschmidt



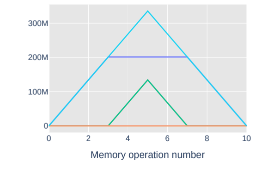
(o) lu



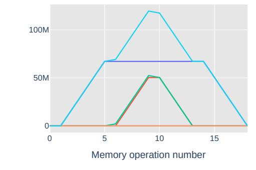
(p) adi



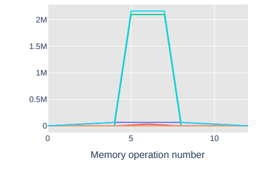
(q) convolution-2d



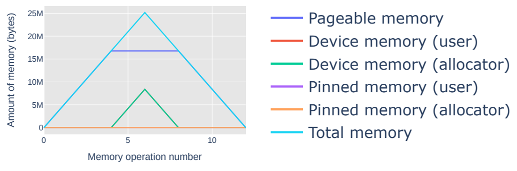
(r) convolution-3d



(s) fdtd-2d



(t) jacobi-1d



(u) jacobi-2d

Figure 5.8: Memory characterization of PolyBench-ACC benchmarks

gemm and *fdtd-2d* have similarities between them, *2mm*, *syr2k*, *syrk*, *gramschmidt*, *lu*, *adi*, *convolution-2d*, *convolution-3d* and *jacobi-2d* have similarities between them, and *atax*, *bicg*, *gemver*, *gesummv* and *mvt* have similarities between them. However, in general, all the benchmarks have the same behavior: the memory is gradually allocated until reaching the peak of maximum memory use and then it is gradually deallocated after use. In this case, what defines the differences in the shape of the characterization plots is the size of some allocations and the order on which host and device allocations are done.

5.3 Summary

In this chapter, we presented a library to characterize the use of dynamic memory in GPU applications. To evaluate our library, we applied it to three popular GPU benchmark suites that include GPU applications from different domains. As a result of the evaluation, we found that our library can be used to characterize the use of dynamic memory in GPU applications and identify memory allocation behaviors that can be modified to improve performance and memory consumption, especially when targeting embedded GPU platforms.

In particular, we have identified the following patterns: since GPU applications use different types of dynamic memory, it is a common mistake to forget to free some of the memory allocations, causing memory leaks. Most of the evaluated applications use the traditional CUDA memory model, on which the user needs to allocate pageable host memory and device memory, which is not optimal in embedded GPU platforms. Moreover, in some of these applications, the number of memory transfers between host and device is very high, causing unnecessary overhead. We also found applications with cyclic allocation/deallocation patterns, a behavior that can be modified to improve performance. In the next chapter, we take into account these results to propose a solution to improve the execution of legacy GPU applications on embedded GPU platforms.

6 Optimization of Dynamic Memory Use in Embedded GPU Platforms

Autonomous systems require high performance processing capabilities, which demand for powerful accelerators such as GPUs. However, using GPUs in critical systems presents several challenges, since GPU programming models rely on explicit dynamic memory management. Traditionally, dynamic memory allocation in such systems is restricted in certain controlled scenarios, which require programs to be rewritten, so that all the required memory is allocated in the beginning of the program and released at its end. However, as we have seen in Chapter 5, many GPU applications do not follow this approach. Moreover, as we have shown in Chapter 4, when dynamic GPU memory allocation is used, it is critical to compute the exact amount of memory used as well as to minimize it, in order to guarantee that it fits in the physical system memory.

In the previous chapter, we analyzed legacy GPU applications included as part of popular GPU benchmark suites, and we identified memory allocation patterns that could be modified to improve performance and memory consumption when targeting embedded GPU platforms. In this chapter, we present XeroZerox, a tool designed to tackle these memory allocation patterns. XeroZerox allows legacy GPU applications to be used in a critical setup without rewriting them while at the same time minimizing their memory consumption and memory management runtime overhead.

6.1 Design and Implementation

In this section, we present the design and implementation of XeroZerox. The main objective of XeroZerox is to minimize the memory consumption of legacy GPU applications when executed in embedded GPU platforms. Legacy GPU applications normally use the traditional CUDA memory model, in which the programmer must define and initialize a set of variables in the host side using dynamic pageable memory, and define an equivalent set of variables in the GPU side, using dynamic device memory. Furthermore, before a kernel execution, the programmer must explicitly copy the initial values from host to device, and after the kernel execution, copy the results back from device to host. As explained in Section 2.3.3, in embedded GPU platforms, both host and device allocations are served from the same physical memory. Therefore, when an application that uses the traditional GPU memory model is executed in an embedded GPU platform, its memory consumption is not optimal, and it performs unnecessary memory transfers.

XeroZerox has been designed to automatically detect the allocations of the traditional memory model and replace them with allocations served from a centralized memory pool. This memory pool is created using zero-copy or unified memory, and its size is calculated based on the amount of memory used by the target application. This way, XeroZerox reduces the memory consumption and eliminates the need for memory transfers. XeroZerox interacts with the CUDA runtime system at the beginning of the application to create the centralized memory pool and then at the end of the application to release the memory pool. All the allocations requested by the application are internally served by XeroZerox using the centralized memory pool. This way, XeroZerox minimizes the overhead produced when interacting with the CUDA runtime memory management system. Furthermore, since all the allocations are served from the centralized memory pool, XeroZerox avoids memory leaks by releasing the whole memory pool when the application finishes.

The functionality of XeroZerox is divided into two different libraries: the analysis library and the optimization library. Like the memory characterization library we presented in the previous chapter, both XeroZerox libraries are implemented using function interposition at compile time. In fact, the XeroZerox analysis library is an extended version of the memory characterization library. As shown in Figure 6.1, XeroZerox is applied in two different phases. In the first phase, the XeroZerox analysis library must be included in the compilation process of the target application. The analysis library contains wrapper functions for every key function required for dynamic memory management in CUDA applications. Each wrapper function gathers the information required for the analysis before calling the corresponding real function. When the resulting compiled application is executed, it generates a file we term *optimization profile* that is used in the next phase. The details about the analysis phase are discussed in Section 6.1.1.

In the second phase, the target application must be compiled again, including the XeroZerox optimization library in the compilation process. This library contains alternative function replacements for all functions used for dynamic GPU memory management in CUDA applications. The alternative functions implement optimized versions of the original functions. To work properly, these functions require the information included in the optimization profile generated in the previous phase. The details about the optimization phase are discussed in Section 6.1.2.

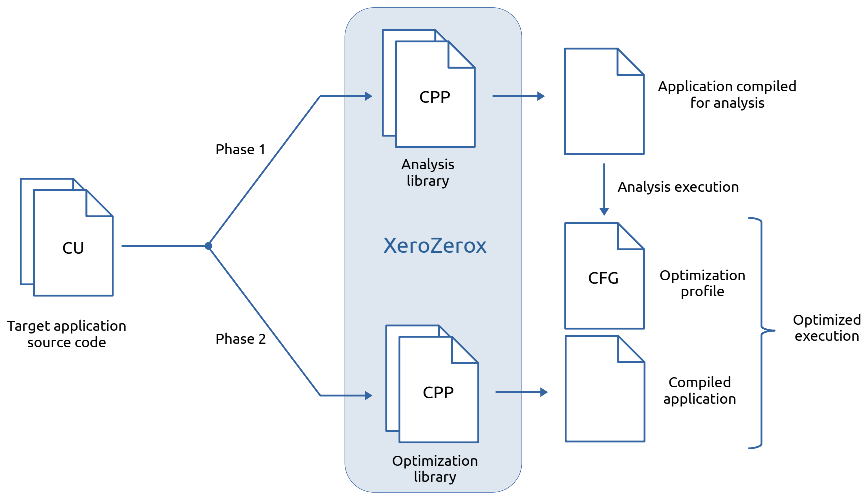
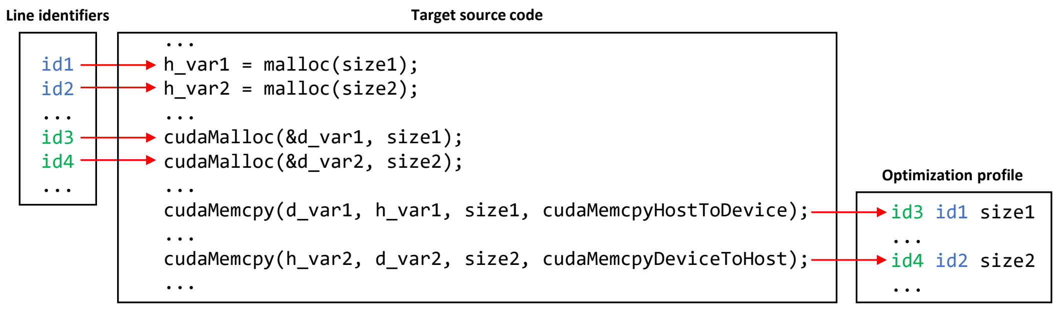


Figure 6.1: XeroZerox analysis and optimization workflow

6.1.1 XeroZerox Analysis Phase

The functionality required for the XeroZerox analysis phase is implemented in the analysis library. As a part of this library, we created a header file with `#define` directives to replace the target memory management functions with wrapper functions. To carry out the memory analysis, first, the header file must be included in the source code of the target application adding a simple `#include` directive. This way, the memory management functions are replaced at the source code level. Then, the application must be compiled, including the analysis library in the compilation process.

When the compiled application is executed, the wrapper functions gather information to create the optimization profile, as shown in Figure 6.2. The optimization profile is a file that contains information about the correspondence between host and device allocations, the sizes requested for each allocation, and the maximum amount of memory needed by the application. This information is required to carry out the optimization phase. To create the optimization profile, first, the host memory allocation wrappers (`malloc/calloc/cudaMallocHost`) and the `cudaMalloc` wrapper gather information about the allocation sizes requested by the application and assign an identifier to each allocation. Then, it is necessary to know which host allocation is related to which device allocation. For this purpose, we take advantage of the `cudaMemcpy` function. The `cudaMemcpy` wrapper identifies the correspondence between a host allocation and device allocation and creates the corresponding match in the optimization profile.



```
...
h_var1 = malloc(size1);
h_var2 = malloc(size2);
...
cudaMalloc(&d_var1, size1);
cudaMalloc(&d_var2, size2);
...
cudaMemcpy(d_var1, h_var1, size1, cudaMemcpyHostToDevice);
...
cudaMemcpy(h_var2, d_var2, size2, cudaMemcpyDeviceToHost);
...
```

```
id3 id1 size1
...
id4 id2 size2
...
```

Figure 6.2: Generation of the optimization profile

To calculate the maximum amount of memory needed by the application, the analysis library keeps track of the amounts of memory requested on each individual allocation using the `malloc`, `calloc`, `cudaMallocHost` and `cudaMalloc` wrappers. It also keeps track on how these allocations are released, using wrappers for the `free`, `cudaFreeHost` and `cudaFree` functions. With that information, the analysis library continuously updates the current amount of memory used and keeps track of the maximum reached value. At the end of the execution, the maximum registered value is stored in the optimization profile as the maximum amount of memory needed by the application.

6.1.2 XeroZerox Optimization Phase

The functionality required to apply the XeroZerox optimizations is implemented in the optimization library. For this library, we also created a header file with `#define` directives to replace the original memory management functions with alternative functions served by XeroZerox. To carry out the optimization, the header file must be included in the source code of the target application. This way, the memory management functions are replaced at the source code level. Then, the target application must be compiled together with the optimization library. In the compilation process, the optimization library must be configured to use either zero-copy memory or unified memory to create the centralized memory pool. When the resulting compiled application is executed, it uses the XeroZerox functions instead of the traditional memory model functions to serve the allocations using the centralized memory pool.

As shown in Figure 6.3, when using the traditional memory model in an embedded GPU platform, the memory is partitioned into one logical space for the host allocations and one logical space for the device allocations. The host allocations are served by a host allocator in the system, while the device allocations are served by an NVIDIA allocator. As previously stated, the memory consumption in this scenario is not optimal since both CPU and GPU allocations are served from the same physical memory. Also, it requires the application to perform memory transfers between both logical spaces, which can negatively impact performance. Furthermore, as we have shown in Chapter 4, the NVIDIA allocator can create multiple memory pools according to predefined size classes, which can negatively impact both memory consumption and performance.

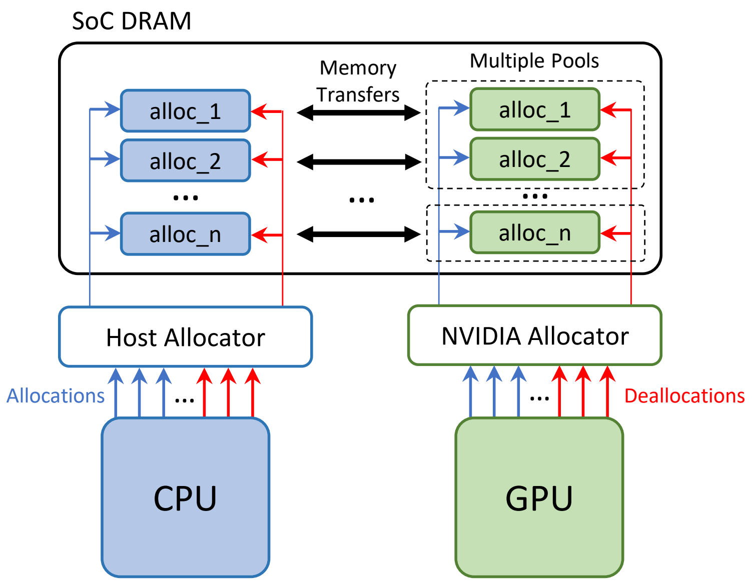


Figure 6.3: Traditional memory model

XeroZerox proposes an alternative memory model, which we show in Figure 6.4. When an application optimized with XeroZerox is executed, it first reads the optimization profile and loads the information required for optimization. The first value it reads is the maximum amount of memory needed, which is used as a reference to create the centralized memory pool. For this task, XeroZerox requests the NVIDIA allocator to reserve either zero-copy memory or unified memory, according to the memory type configured in the compilation process. It is important to note that the minimum amount of memory reserved for the centralized memory pool still depends on the NVIDIA allocator and the configured memory type. As explained in Chapter 4, for zero-copy memory, the minimum amount of memory reserved by the NVIDIA allocator is the pool size, which can be 1 MB or 2 MB depending on the platform. In the case of unified memory, the minimum amount of memory reserved by the NVIDIA allocator in the Jetson platforms is 4 KB, which is the page size. For sizes larger than the pool size in zero-copy, or the page size in unified memory, the amount of memory reserved by the NVIDIA allocator is the next multiple of the page size.

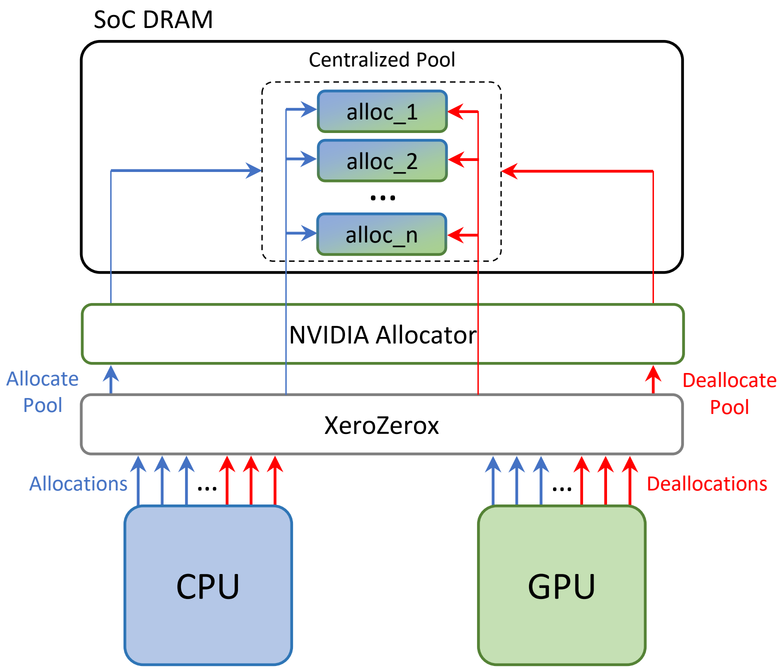


Figure 6.4: XeroZerox memory model

After creating the centralized memory pool, XeroZerox works as a sub-allocator, attending the allocation requests from the application based on the matches loaded from the optimization profile. Since each match represents a host allocation and a device allocation, XeroZerox only allocates memory for the first of the two allocation requests it receives. When it receives the second allocation request, it returns a pointer to the region of memory that has been already allocated for the first one. This way, XeroZerox transforms two matching host and device allocations into a single allocation served from the centralized memory pool. Similarly, XeroZerox deallocates a memory region only when it receives the two corresponding deallocation requests. At the end of the application execution, XeroZerox deallocates the centralized memory pool using the corresponding CUDA function.

It is worth noting that, while the zero-copy and unified memory models allow the reduction in memory consumption, the performance of the applications when using these memory models will depend on the memory access patterns of each application and the coherency mechanisms of the underlying platform, as explained in Section 2.3.3. Even when XeroZerox can improve the performance of an application by eliminating the need for memory transfers and reducing the interaction with the runtime system, this performance improvement can be overshadowed by the limitations of the zero-copy and unified memory models.

6.2 Evaluation

This section presents the results we obtained after applying the XeroZerox optimizations to the benchmarks of the Rodinia benchmark suite. To carry out the evaluation, we first compiled the CUDA version of the benchmarks, including the XeroZerox analysis library in the compilation process. Then, we executed the compiled benchmarks to obtain the optimization profiles. Finally, we compiled the benchmarks again using the XeroZerox optimization library, and generated one version optimized with zero-copy memory and one version optimized with unified memory for all the benchmarks. We used two different platforms from the NVIDIA Jetson family to perform the evaluation. The details of the selected platforms are provided in Table 6.1. We selected one platform with hardware I/O coherency and another without hardware I/O coherency to illustrate the different behaviors we can expect in the performance of the applications when using zero-copy and unified memory.

Table 6.1: Platforms used for XeroZerox evaluation

Jetson platform	GPU architecture	Compute capability	SMs	CUDA cores	HW I/O coherency	Pool size
TX2	Pascal	6.2	2	256	No	2 MB
Xavier NX	Volta	7.2	6	384	Yes	2 MB

6.2.1 Memory Consumption

To evaluate XeroZerox in terms of reduction in memory consumption, we executed the zero-copy and unified memory versions of the benchmarks in the selected embedded GPU platforms. Since the memory page size is the same in all Jetson platforms, and both selected platforms share the same pool size, the memory consumption results are identical for both platforms. While doing the evaluation, we realized that the *hybridsort* and the *kmeans* benchmarks were not compatible with XeroZerox. By design, XeroZerox works with legacy GPU applications on which the correspondence between host allocations and device allocations is one-to-one. If an application has one-to-many correspondences between host allocations and device allocations, applying XeroZerox would result in undesirable behavior. For this reason, these benchmarks are not included in the evaluation. Table 6.2 shows the results we obtained for the rest of the benchmarks.

Table 6.2: Memory consumption results for Rodinia benchmarks

Benchmark	Memory consumption (bytes)					
	Original		XeroZerox		Reduction %	
	Required	Allocated	ZC	UM	ZC	UM
backprop	9437460	20455632	9441280	9441280	53.85	53.85
bfs	38999881	79202120	39002112	39002112	50.76	50.76
b+tree	62377684	84762324	62377984	62377984	26.41	26.41
cfd	12824064	17094144	12824576	12824576	24.98	24.98
dwt2d	32505856	36700160	32505856	32505856	11.43	11.43
gaussian	144	2097296	2097152	4096	0.01	99.80
heartwall	43350996	46035288	43352064	43352064	5.83	5.83
hotspot	3145728	7340032	3145728	3145728	57.14	57.14
hotspot3D	25165824	50331648	25165824	25165824	50.00	50.00
huffman	6301704	10487808	6303744	6303744	39.89	39.89
lavaMD	7856000	16359296	7856128	7856128	51.98	51.98
leukocyte	1681920	3779072	2097152	1683456	44.51	55.45
lud	262144	2359296	2097152	262144	11.11	88.89
myocyte	812	3517200	2097152	4096	40.37	99.88
nn	513168	2268208	2097152	516096	7.54	77.25
nw	50380812	83976204	50384896	50384896	40.00	40.00
particlefilter	516392	2305544	2097152	520192	9.04	77.44
pathfinder	40400000	82097280	40402944	40402944	50.79	50.79
srad	7364992	9315952	7368704	7368704	20.90	20.90
streamcluster	72941620	147656756	72945664	72945664	50.60	50.60

In the table, the *Allocated* column shows the amount of memory allocated in the original Rodinia benchmarks using the traditional memory model. The *Required* column shows the maximum amount of memory required by each benchmark, which has been calculated in the XeroZerox analysis phase. The *XeroZerox* columns present the size of the centralized memory pool in both zero-copy (ZC) and unified memory (UM) versions of the benchmarks. Finally, the *Reduction %* columns present the percentage of reduction in memory consumption for both ZC and UM versions with respect to the memory allocated in the original benchmarks.

As shown in the table, the ZC and UM versions present an identical percentage of reduction in the cases where the required amount of memory is larger than the pool size, which in the selected platforms is 2 MB. This is an expected result since, in the Jetson platforms, all unified memory allocations and the zero-copy allocations larger than the pool size are reserved using the same rule: reserve the next multiple of 4 KB. On the other hand, in the benchmarks that require small amounts of memory, like *gaussian*, *lud*, *myocyte*, *nn* and *particlefilter*, the percentage of reduction of the UM version is higher than the percentage of the ZC version. This is due to the minimum pool size that can be allocated with each type of memory. For example, to serve the 144 bytes required for the *gaussian* benchmark, it is more efficient the minimum 4 KB that can be reserved with unified memory than the minimum 2 MB that can be reserved with zero-copy memory.

The most significant reduction in memory consumption is obtained in benchmarks like *backprop*, *bfs*, *hotspot*, *hotspot3D*, *lavaMD*, *leukocyte*, *nw*, *pathfinder* and *streamcluster*, where the percentage of reduction is around 50% with both ZC and UM versions. As shown in Figure 5.3, in these benchmarks, the memory used is almost evenly distributed between host memory and device memory, which makes possible the reduction in memory consumption to approximately the half. On the contrary, in benchmarks like *cfd*, *dwt2d*, *heartwall* and *srad*, where the memory is not evenly distributed, we get less than 25% of reduction in memory consumption.

6.2.2 Performance Evaluation

As explained in Section 2.3.3, the performance of applications using the zero-copy and the unified memory models depends on the memory access patterns of the application and the coherency mechanisms used in the underlying platform. Therefore, to evaluate the performance of the Rodinia benchmarks after applying the XeroZerox optimizations, we executed several times the original, the zero-copy, and the unified memory versions of the benchmarks on both embedded GPU platforms and registered their maximum execution times. The results are shown in Table 6.3.

As shown in the table, the performance of the *cfd*, *heartwall* and *streamcluster* benchmarks is severely affected when we use zero-copy memory in the Jetson TX2 platform. If we compare these results with the results obtained when we use unified memory in the same platform, we can see that the performance is not affected in the same way. This means that these applications are cache-dependent. In embedded NVIDIA platforms without hardware I/O coherency, like the Jetson TX2, the last level caches of both CPU and GPU are disabled when using zero-copy memory but remain enabled when using unified memory. In the case of the Jetson Xavier NX, there are no significant changes in the performance of the applications when using zero-copy or unified memory instead of the traditional memory model. Due to the presence of hardware I/O coherency in this platform, the performance of the benchmarks is no longer affected when using zero-copy memory.

Table 6.3: Maximum execution times of Rodinia benchmarks (s)

Benchmark	Jetson TX2			Jetson Xavier NX		
	Original	XeroZerox		Original	XeroZerox	
		ZC	UM		ZC	UM
backprop	0.199	0.282	0.214	0.223	0.222	0.227
bfs	3.811	4.225	3.846	3.624	3.655	3.058
b+tree	3.144	3.534	3.328	2.824	2.913	2.799
cfd	12.604	41.396	13.618	6.878	7.590	6.662
dwt2d	0.275	0.287	0.163	0.177	0.194	0.230
gaussian	0.087	0.083	0.083	0.129	0.129	0.131
heartwall	2.037	64.415	2.064	0.763	0.817	0.846
hotspot	0.691	0.776	0.736	0.788	0.671	0.686
hotspot3D	17.880	21.860	21.405	9.714	9.920	9.433
huffman	0.225	0.459	0.235	0.222	0.210	0.211
lavaMD	1.173	1.236	1.177	0.979	0.989	0.985
leukocyte	0.759	0.813	0.760	0.676	0.614	0.670
lud	0.115	0.130	0.121	0.153	0.164	0.160
myocyte	1.130	0.436	0.548	0.971	0.517	0.535
nn	0.120	0.173	0.135	0.198	0.190	0.184
nw	0.244	0.268	0.220	0.196	0.293	0.290
particlefilter	0.130	0.288	0.138	0.202	0.224	0.226
pathfinder	2.897	2.904	2.921	2.566	2.302	2.675
srad	0.307	0.982	0.334	0.299	0.319	0.322
streamcluster	15.553	49.666	14.371	9.001	7.785	10.306

It is worth noting that the performance of the *myocyte* benchmark is improved in both platforms when using either zero-copy or unified memory. As shown in Table 5.1, this benchmark performs 7800 host-to-device and 7800 device-to-host memory transfers, which are completely eliminated when using XeroZerox. This means that the amount of memory transfers in most of the benchmarks is not enough to obtain a significant performance improvement when using XeroZerox.

6.3 Summary

In this chapter, we presented XeroZerox, a tool designed to minimize the memory consumption and memory management overhead of legacy GPU applications when executed in embedded GPU platforms. To evaluate XeroZerox, we applied it to the benchmarks of the Rodinia benchmark suite and executed the resulting applications in two different embedded GPU platforms. The results show that XeroZerox can reduce to approximately 50% the memory consumption in applications where the use of memory is evenly distributed between host allocations and device allocations. In terms of performance, we provided results to show the impact of zero-copy and unified memory on the performance of the applications, depending on the coherency mechanisms used by the underlying platform. In the Jetson TX2, we observed that zero-copy memory could negatively affect the performance of cache-dependent applications due to the absence of hardware I/O coherency mechanisms. On the other hand, in the Jetson Xavier NX, which includes hardware I/O coherency, the performance of the benchmarks was not significantly affected. Finally, we observed that even when XeroZerox can improve the performance of an application by eliminating the memory transfers and reducing the interaction with the runtime system, the performance improvement can be overshadowed in applications with a small number of memory transfers.

7 Timing Characterization of Control Algorithms Executed on Embedded GPUs

As we have discussed in the Background Chapter, control systems are real-time systems that regulate the behavior of devices or equipment using feedback control loops. These control loops must be executed periodically, with sampling rates according to the needs of the system under control. An unexpectedly long execution time of a control algorithm could delay the action of the controller on the system under control, possibly causing an unwanted situation. For this reason, control systems usually have hard real-time constraints.

Most real-world control algorithms are simple enough to be executed in micro-controllers or embedded microprocessors. In this scenario, if the computational power of a single GPU core can be compared to the power of a microcontroller, the highly parallel architecture of embedded GPUs could be leveraged to implement scalable parallel control systems, replacing multiple microcontrollers. However, GPUs are known about their non-deterministic nature, which complicates their use for implementing real-time systems.

In this Chapter, we characterize the timing of control algorithms executed in embedded GPUs. We start by defining a case study for the parallel control of multiple motors. We introduce the theory needed to understand the control techniques used in this domain and we describe the corresponding control algorithms. Then, we define different scenarios for the configuration of embedded GPU platforms and show the timing behavior of the control algorithms on each scenario. Using this methodology, we identify which configurations and techniques help to improve real-time behavior. Finally, we propose a proof of concept implementation to validate the feasibility of the identified configurations by controlling an actual motor using an embedded GPU platform. We show that not only it is possible to meet real-time performance of control tasks using a platform based on an embedded GPU, but that the same platform can be used in order to perform multiple control tasks in parallel, without a slow down.

7.1 Case Study: Parallel Control of Permanent Magnet Synchronous Motors

A Permanent Magnet Synchronous Motor (PMSM) is a rotating electrical machine with 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. To apply currents through the windings, it needs external electronic commutation. For this purpose, a three-phase driving inverter topology is used. Figure 7.1 shows the inverter topology and the structure of a PMSM.

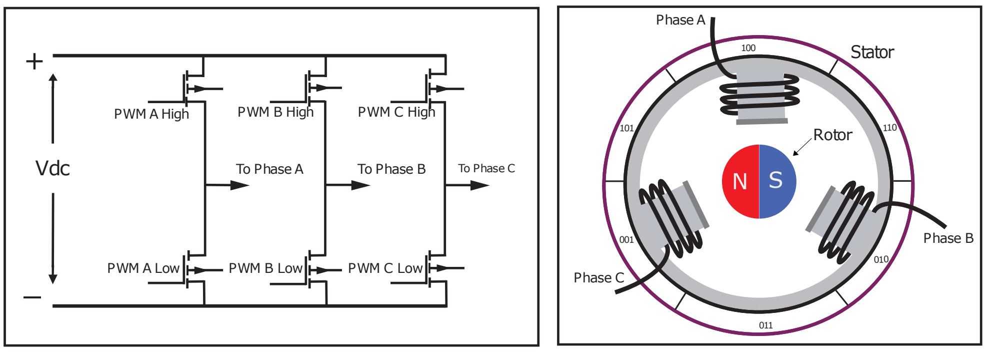


Figure 7.1: Driving inverter topology and PMSM structure [86]

The three stator coils of a PMSM are permanently energized with a sinusoidal current which is 120 degrees apart on each phase, which creates a rotating North/-South magnetic field. The maximum torque is produced when the magnetic vector of the rotor is at 90 degrees to the magnetic vector of the stator as shown in Figure 7.2. Therefore, in order to make the motor turn optimally, we need to know the angular position of the rotor in real time and then apply voltage on the three wires so that the magnetic field on the stator is 90 degrees apart. Having this information, a logical control solution would consist in reading continuously the rotor angle and applying the corresponding voltages to create the magnetic field 90 degrees apart. However, control based on the values of three sinusoidal currents can be complex and time consuming.

To simplify the control of PMSMs, a vector control technique known as Field Oriented Control (FOC) is used. The basic idea of the FOC technique is to decompose the stator currents into a flux-producing part (direct current I_d) and a torque-producing part (quadrature current I_q). Both components can be controlled separately after decomposition.

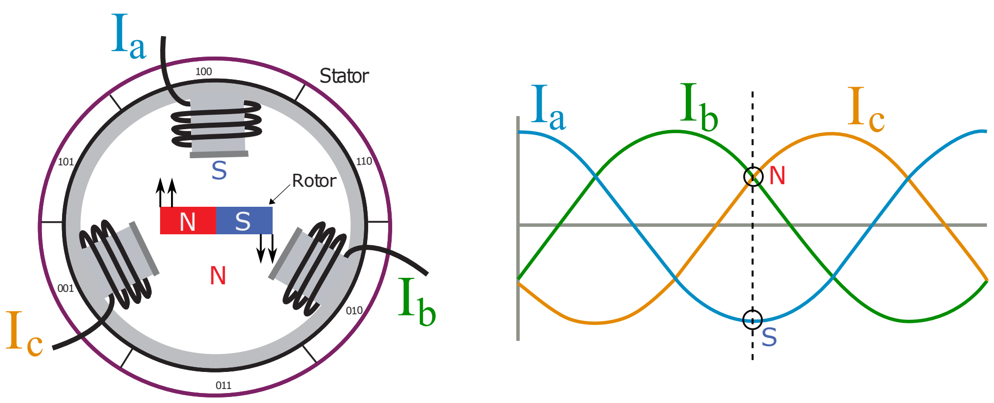


Figure 7.2: PMSM optimal torque angle

As shown in Figure 7.3, the direct current and the quadrature current are the ones causing the outward pull and the perpendicular pull respectively. The main goal of the FOC technique is to measure those two currents and adjust the phase of the voltages in order to bring the direct current I_d to zero, leaving only the I_q current to control the torque.

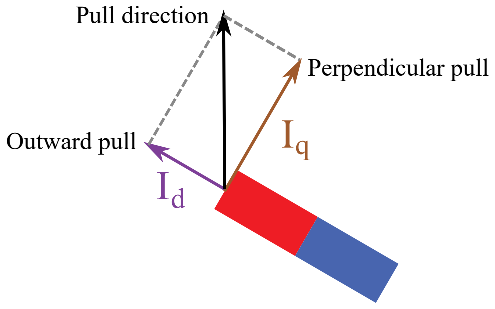


Figure 7.3: FOC currents and equivalent pulls

To convert the stator currents into direct and quadrature currents, the FOC algorithm first transforms three-phase quantities into equivalent two-phase quantities, in a stationary reference frame, by applying a *Clarke* transformation. Then, the two-phase quantities are transformed from the stationary reference frame to a rotating reference frame by applying a *Park* transformation. Figure 7.4 shows the transformations in the different reference frames. After the transformation, the regulation of torque and flux is done using Proportional-Integral (PI) controllers. The adjusted voltage values are then transformed into three-phase values by applying the inverse Park and inverse Clarke transformations.

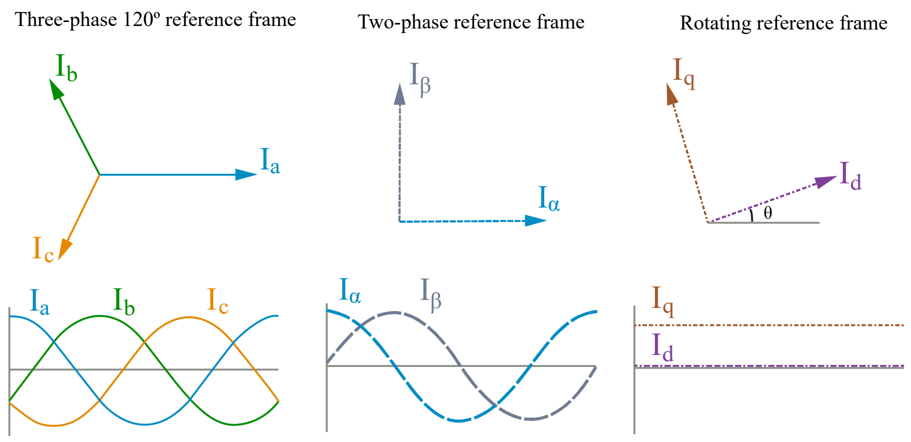


Figure 7.4: FOC transformations [86]

Figure 7.5 shows a diagram of a typical FOC process. The FOC algorithm is divided into two control loops that execute at different frequencies. The first control loop is the slower one and is a simple PI controller, which is used to control the velocity of the motor. Algorithm 6 shows the steps executed in the velocity control loop.

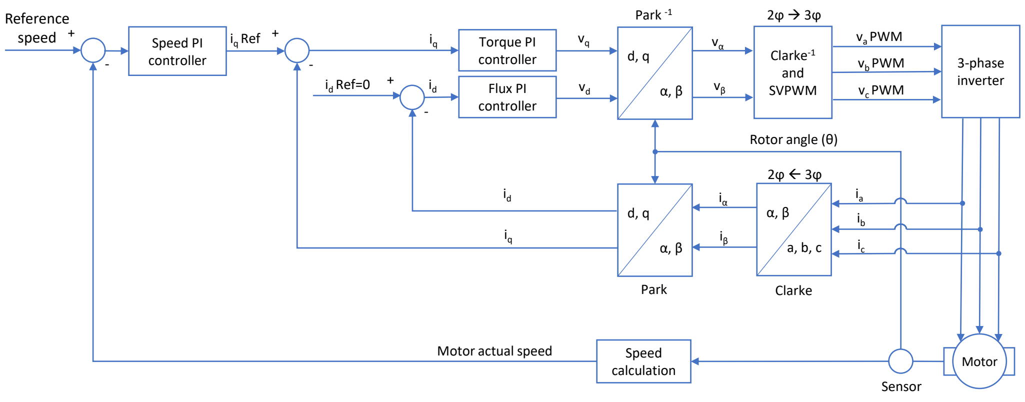


Figure 7.5: Field Oriented Control process

The output of this control loop is used as the reference value for the quadrature current (*reference_iq*) in the second control loop. The second control loop is more complex and runs at a higher frequency. This control loop performs the needed coordinate transforms of the currents to determine the time-invariant values of torque and flux of the motor. These values are then controlled using PI controllers. Algorithm 7 shows the steps executed in the current control loop.

First, the three-phase currents of the stator are converted to a stationary two-axis system by applying a Clarke transform. This stationary two-axis system is then

Algorithm 6: Velocity control algorithm

```
Input     : reference_speed, measured_speed
Output    : reference_iq
InOut     : accumulated_error
Parameter : kp, ki, dt

// PI controller for speed
$speed\_error \leftarrow reference\_speed - measured\_speed$
$accumulated\_error \leftarrow accumulated\_error + speed\_error * dt$
$reference\_iq \leftarrow kp * speed\_error + ki * accumulated\_error$
```

Algorithm 7: Current control algorithm

```
Input     : reference_iq, ia, ib, angle
Output    : va, vb, vc
InOut     : accumulated_error_id, accumulated_error_iq
Parameter : kp, ki, dt

// Clarke transform
$ialpha \leftarrow ia;$
$ibeta \leftarrow (1/sqrt(3)) * (ia + (2 * ib))$
// Park transform
$id \leftarrow cos(angle) * ialpha + sin(angle) * ibeta$
$iq \leftarrow cos(angle) * ibeta - sin(angle) * ialpha$
// PI controller for id
$id\_error \leftarrow -id$
$accumulated\_error\_id \leftarrow accumulated\_error\_id + id\_error * dt$
$vd \leftarrow kp * id\_error + ki * accumulated\_error\_id$
// PI controller for iq
$iq\_error \leftarrow reference\_iq - iq$
$accumulated\_error\_iq \leftarrow accumulated\_error\_iq + iq\_error * dt$
$vq \leftarrow kp * iq\_error + ki * accumulated\_error\_iq$
// Inverse Park transform
$valpha \leftarrow cos(angle) * vd - sin(angle) * vq$
$vbeta \leftarrow sin(angle) * vd + cos(angle) * vq$
// Inverse Clarke transform
$va \leftarrow valpha$
$vb \leftarrow (-valpha + (sqrt(3) * vbeta))/2$
$vc \leftarrow (-valpha - (sqrt(3) * vbeta))/2$
```

rotated to align with the rotor flux by applying a Park transform, using as reference the angle of the rotor measured in the last iteration of the control loop. The resulting values of this operation are the direct current (*id*) and the quadrature current (*iq*). The direct current controls the rotor magnetizing flux and the quadrature current controls the torque output of the motor. The purpose is to control the torque using as reference the last *reference_iq* value produced by the velocity controller. For this task, these values are fed to the corresponding PI controllers to get the values *vd* and *vq*, which are the voltage vectors that we need to apply to the motor. The final three-phase motor voltages (*va*, *vb* and *vc*) are generated by applying the inverse Park transform, and then applying the inverse Clarke transform combined with some space vector modulation techniques to derive the PWM signals required for the inverter switches.

The sampling rates required to execute both FOC algorithms depend on the switching frequency of the three-phase inverter and the physical characteristics of the motor. The period of the velocity control loop is usually around a few milliseconds, while the period of the current control loop is typically inferior to 100 microseconds. As a general rule, the velocity control loop should be executed once for every ten iterations of the current control loop. Therefore, for the timing evaluation in this chapter, we consider 1 millisecond and 100 microseconds as the maximum execution times acceptable for the velocity control loop and the current control loop, respectively.

7.2 Experiments Design

This section describes the different software configurations we used to run our experiments, the methodology we applied to measure the timing on each scenario, the embedded GPU platforms we selected, and the technologies we used to implement the proof of concept prototype.

7.2.1 Timing Characterization Scenarios

To observe the attainable time determinism when executing control algorithms on embedded GPUs, we created CUDA versions of algorithms 6 and 7 and executed them on different scenarios. On these CUDA implementations, we used the zero-copy memory model to avoid the overhead of memory transfers. The three scenarios we defined are described next.

Stock System Scenario: In this scenario, we execute the FOC algorithms using the stock system setup described in Section 3.3. We installed the stock L4T version 32.5 on the selected platforms, and we applied the typical configurations a user would apply to gain performance. The goal of this setup is to observe the timing behavior of control algorithms executed on a stock Jetson system configured for maximum performance. To run the experiments in this scenario, we applied the next configurations:

- Set NVIDIA power model to maximum performance: The Jetson boards have different power models which can be enabled using the `nvpmodel` tool. The power model defines the number of CPU cores enabled and the maximum frequencies allowed for the CPU and the GPU.
- Switch the CPU, GPU and memory to maximum frequencies: After setting the maximum allowed frequencies with `nvpmodel`, it is necessary to switch the system to the maximum frequencies. This is done setting the minimum frequencies equal to the maximum frequencies with the command `jetson_clocks`.
- Isolate a CPU core: The Linux kernel allows the isolation of CPU cores from the general scheduler to reserve them for specific applications. The isolation of core n is done by adding the `isolcpus=n` parameter on the `APPEND` section of the `/boot/extlinux/extlinux.conf` file.
- Attach the process to an isolated CPU core: Once we have isolated a CPU core, we can attach a process to run alone on it. We used the function `sched_setaffinity` to specify the core on which our experiments have to run.
- Lock the process in memory: Virtual memory allows the use of more memory than the one present in the system. However, memory swapping can negatively affect the deterministic behavior of a program. To avoid the delays that can be caused by memory swapping in Linux, we lock the process in memory using the function `mlockall(MCL_CURRENT | MCL_FUTURE)`.

Real-time System Scenario: For this scenario, we applied the real-time system setup described Section 3.3 to the selected embedded platforms. We installed L4T version 32.5 with the PREEMPT-RT patches applied to the Linux kernel, to enable the full preemptive mode. The goal of this scenario is to observe the gain in determinism we can get when using a real-time Linux kernel, compared to the stock system scenario. To run the experiments in this scenario, we applied the performance configurations we used in the stock system scenario, and also applied the next configurations:

- Assign a real-time priority to the process: We assigned to our experiments a real-time priority 98 and enabled the `SCHED_FIFO` scheduler using the function `sched_setscheduler`.

- Disable real-time throttle: Real-time throttle is a mechanism that limits the amount of CPU time given to processes with real-time priority. With this feature, processes with real-time priority can get interrupted every certain time to allow the execution of processes with lower priority. In our experiments, the process run alone in an isolated core, so it is safe to disable this feature using the command:

  ```
  echo -1 > /proc/sys/kernel/sched_rt_runtime_us
  ```

- Disable kernel watchdog: The kernel watchdog timer is used to detect and recover from software faults. It requires a regular timer interrupt which can cause jitter. We can trade lower error detection for better time determinism by disabling it:

  ```
  echo 0 > /proc/sys/kernel/watchdog
  ```

- Delay virtual memory statistics timer: This is used for collecting virtual memory statistics. We can reduce the amount of jitter caused by this feature, configuring a large interval in seconds:

  ```
  echo 1000 > /proc/sys/vm/stat_interval
  ```

- Increase flush time to disk: We configure the write-back of dirty memory pages to occur less often using the command:

  ```
  echo 1500 > /proc/sys/vm/dirty_writeback_centisecs
  ```

Persistent Kernel Scenario: Besides the configurations we can apply at the operating system level to improve the time determinism of GPU applications, we must also consider the possible improvements in the GPU programming model. It is known that the GPU programming model itself has some characteristics which can cause a non-deterministic behavior. For example, one of the sources of non-determinism in GPU applications is the interaction between the CPU and the GPU driver when launching GPU tasks [33]. As introduced in Subsection 2.4, some authors have used the persistent threads model of programming [39] to improve the time determinism when launching GPU kernels. In this scenario, we created a persistent threads version of the FOC algorithms to compare the level of determinism we can get using this model. To run the experiments, we used L4T 32.5 with a real-time Linux kernel and applied the same configurations we used in the real-time system scenario.

7.2.2 Timing Measurement

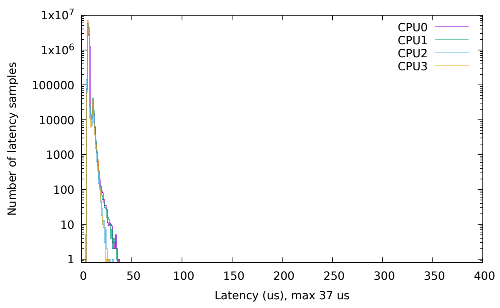
To measure the timing of each experiment in the different scenarios, we run 10 million iterations of the control algorithms, from which we get the maximum execution time, the average execution time, and the standard deviation. To get the execution time of each iteration of the control algorithm, we use the function `clock_gettime` with the `CLOCK_MONOTONIC` system clock before and after launching the GPU workload. To calculate the standard deviation on each experiment, we first calculate the variance using Welford's Method [87][88]. Welford's Method is an online algorithm for computing the sample mean and variance without storing a large number of values. This allows us to execute a large number of control iterations.

To measure the scalability of each scenario, we opted to run the control algorithm with 32, 64, 128, 256, 512, and 1024 threads for each set of configurations. The reason for using those limits is that 32 is the minimum number of threads executed in parallel on NVIDIA GPUs (because of the warp size), and 1024 threads is the maximum size of a block in the selected platforms. This way, we can launch experiments using only one block of threads, avoiding the overhead of any inter-block synchronization mechanism.

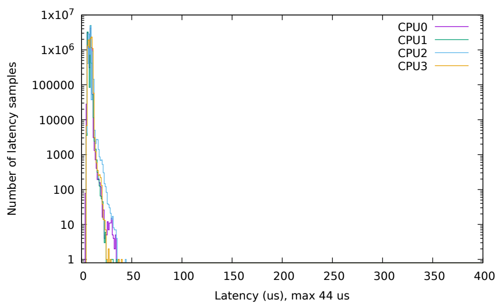
7.2.3 Systems Under Test

Initially, we planned to run the timing experiments on the four currently available platforms from the NVIDIA Jetson family, which are described in section 3.2. However, we realized that both Jetson Xavier platforms presented unexpectedly long results in some simple timing tests during the preparation and testing of the real-time system scenario. To know if the operating system was causing the latency, we executed the latency test recommended by the Open Source Automation Development Lab (OSADL) [89] on the four embedded GPU platforms, using a fresh installation of L4T 32.5 with the PREEMPT-RT patches applied to the Linux kernel. The latency test runs several `cyclictest` iterations and creates a latency plot showing a distribution curve of the obtained latency values for each CPU core. In addition, the latency plot shows the maximum latency value obtained during the test.

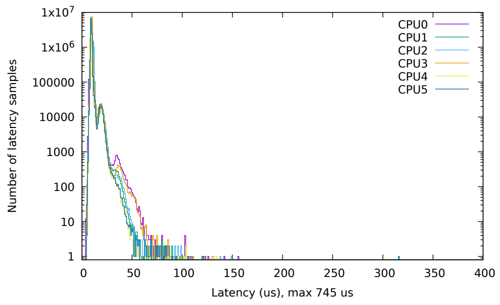
According to OSADL, currently, there is no standard way to determine the correct maximum latency of a system. However, following a rule of thumb, a system with a 1 GHz processor should get a maximum latency inferior to 100 microseconds. The results obtained in the four Jetson platforms are shown in Figure 7.6. As we can



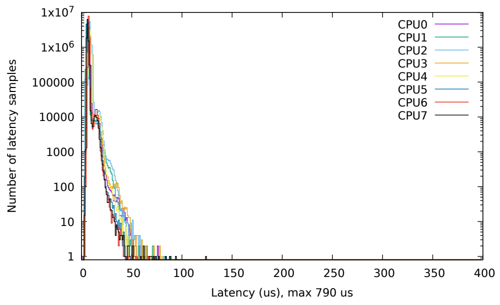
(a) Jetson Nano



(b) Jetson TX2



(c) Jetson Xavier NX



(d) Jetson AGX Xavier

Figure 7.6: Latency plots of Jetson platforms with real-time Linux

see in the results, we obtained a maximum latency of over 700 microseconds in both Jetson Xavier platforms. In comparison, in the Jetson Nano and Jetson TX2, the maximum latency is inferior to 50 microseconds. We tried the same test with previous versions of L4T, but the results were similar. At this point, we are not sure about the source of latency in those systems, but we concluded that currently, the PREEMPT-RT patches are not fully compatible with the Jetson Xavier platforms. For this reason, we selected the Jetson Nano and the Jetson TX2 to run the timing experiments.

7.2.4 Proof of Concept

To validate the feasibility of the previous configurations in a real-life scenario, we created a proof of concept prototype application to control a real motor using an embedded GPU platform. For this purpose, we first selected a baseline platform to establish a comparison point. Our approach involves implementing a version of the FOC algorithms into the baseline platform and then translating the implementation to an embedded GPU platform and comparing results. The GPU implementation should be based on the configurations with better timing behavior, according to the results of the scenarios presented in the first part of this section.

The first platform we considered as baseline is the Texas Instruments C2000 Delfino MCU F28379D LaunchPad development kit [90], which was designed specifically for the control of PMSMs. This development kit consists of a LAUNCHXL-F28379D Delfino board, a BOOSTXL-DRV8305EVM driver module, and a Teknic M-2310P-LN-04K motor. Figure 7.7 shows a block diagram of a PMSM control setup using the Delfino platform.

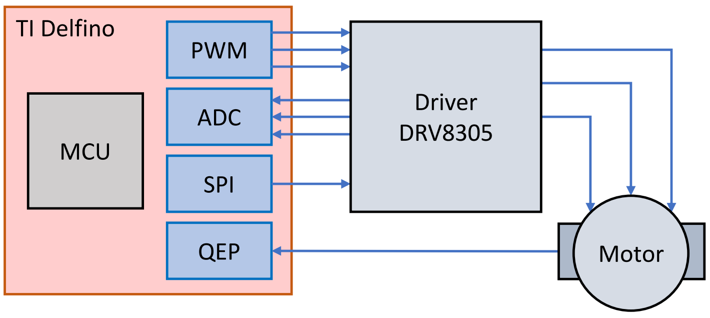


Figure 7.7: PMSM control using Delfino platform

After the first test with the Delfino platform, we realized that it works with optimized precompiled software modules to implement different parts of the control task, complicating the use of our implementation of the control algorithms. Moreover, it complicates the addition of the instrumentation code we need to extract the results of our experiments. For this reason, we decided to replace the Delfino board in our initial setup with a more open alternative platform.

To be able to communicate with the driver and motor included with the Delfino platform, the alternative board must include Pulse-Width Modulation (PWM) modules to generate complex pulse width waveforms with minimal microcontroller intervention, Analog to Digital Converter (ADC) modules to read and convert the values of the PMSM currents, a Serial Peripheral Interface (SPI) to configure the driver, and a Quadrature Encoder Pulse (QEP) module to read the values of the PMSM encoder and calculate the rotation angle. We selected the Teensy platform for this purpose, which fulfills all these requirements [91]. Moreover, Teensy is an open platform compatible with Arduino, which eases the development of our version of the FOC algorithms and their instrumentation. Figure 7.8 shows a block diagram of the baseline setup using the Teensy platform.

To implement the proof of concept setup, we selected the Jetson TX2 platform. Initially, we evaluated the possibility of connecting the DRV8305 driver and the motor directly to the Jetson TX2; however, the Jetson platforms lack ADC and QEP modules. The PWMs can be implemented via General Purpose Input Output (GPIO) pins, but this is not recommended since the signals might not be stable [92]. For

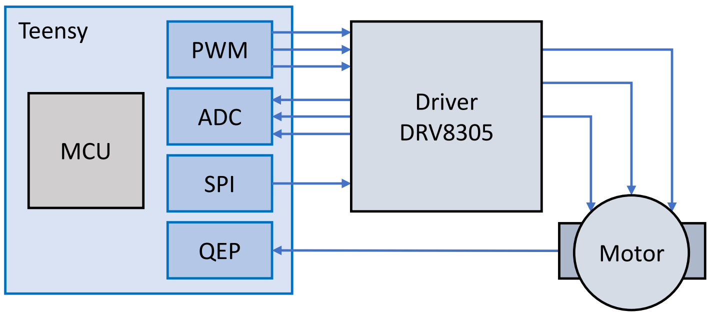


Figure 7.8: PMSM control using Teensy platform

these deficiencies, we decided to use the Teensy as an external board to provide the missing modules to the Jetson TX2. To communicate the two boards, we used the Inter-Integrated Circuit (I2C) bus, as shown in Figure 7.9. The Teensy acts as an interface between the Jetson TX2 and the driver/motor in this setup.

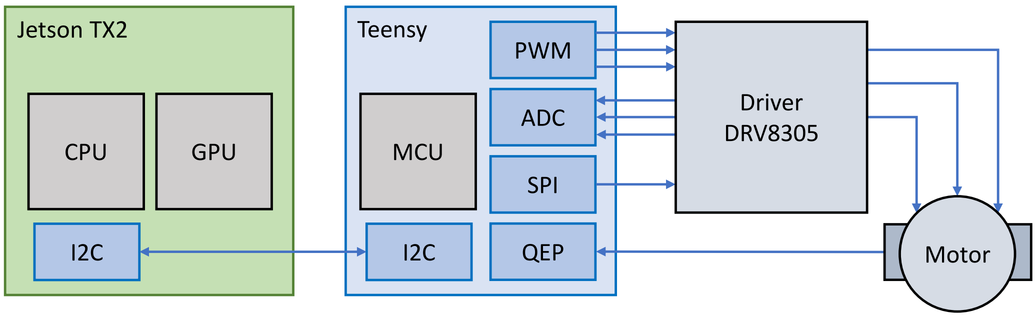


Figure 7.9: PMSM control using Jetson TX2 platform

We are aware that the I2C bus is a bottleneck since it may not be fast enough to comply with the timing required by the FOC algorithms. However, for a proof of concept, it may be enough. Therefore, to have a fair comparison point, we slow down the control frequency on the baseline setup to match the extra time the Jetson TX2 requires for the I2C communication. Then, we implement the FOC algorithms in the CPU of the Jetson TX2 to observe the effect of changing the control execution from one platform to the other one and adding the I2C bus to the setup. Lastly, we implement the GPU version of the control algorithms and compare the results with the previous ones.

7.3 Results

This section presents the timing results we obtained in the timing characterization scenarios and the control results we obtained with the different configurations of the proof of concept prototype.

7.3.1 Timing Characterization Scenarios

To evaluate the timing behavior of the FOC control algorithms on the previously defined scenarios, we executed timing experiments with different threads configurations on both selected Jetson platforms. We run 10 million control iterations on each experiment, 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).

Stock System Scenario

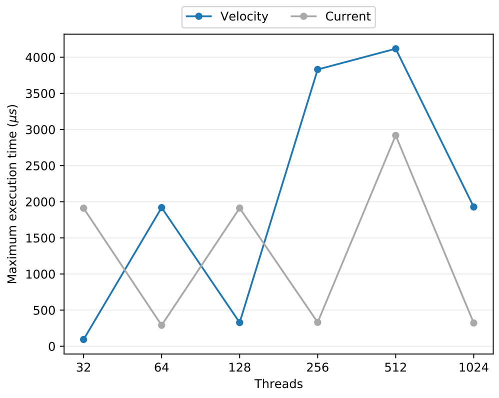
Table 7.1 shows the results of the timing experiments in the stock system scenario for both Jetson platforms. As shown in the table, the average execution time for both control algorithms in the Jetson Nano is approximately 30 microseconds. In the case of the Jetson TX2, the average execution time for both control algorithms is about 20 microseconds. While the average execution times are in an acceptable range, the maximum execution times are very high in some cases.

It is important to note that the high values of the maximum execution times are not directly related to the number of threads being executed. As shown in Figure 7.10, in both platforms, there are thread configurations on which the maximum execution time of one algorithm is very high and the maximum execution time of the other one is low.

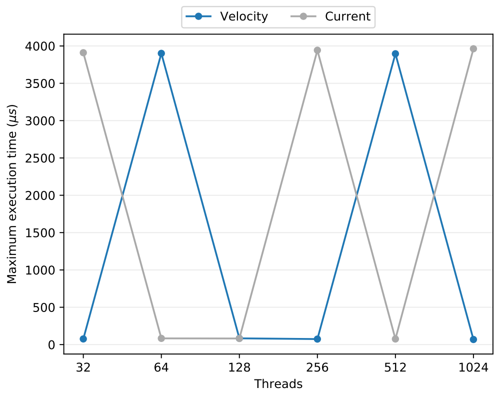
Since these experiments have been executed on an isolated CPU core, it is not likely that another user application causes the interference. However, in Linux, some system processes run with high priority, and they can interrupt user tasks even when running on isolated CPU cores. On the other hand, the source of latency may also be related to the interaction of the application with the GPU runtime system. In the following scenarios, we evaluate these cases.

Table 7.1: Execution times of FOC algorithms in stock system scenario (μs)

Platform	Threads	Velocity			Current		
		Avg	Max	S.D.	Avg	Max	S.D.
Jetson Nano	32	29.1	93.5	1.5	29.7	1911.0	1.6
	64	28.9	1918.6	1.8	30.0	290.2	1.3
	128	29.3	327.8	1.5	30.6	1912.1	1.5
	256	28.7	3830.5	2.6	30.3	330.4	1.3
	512	28.7	4117.6	2.4	30.4	2917.8	2.0
	1024	28.7	1927.8	1.9	32.3	321.6	1.2
Jetson TX2	32	18.9	75.9	0.9	21.2	3909.7	2.0
	64	19.2	3900.8	1.9	20.9	83.3	0.9
	128	19.0	83.8	0.8	20.8	81.8	0.9
	256	19.0	74.0	0.9	20.9	3942.8	2.0
	512	19.4	3895.7	1.9	21.4	75.2	0.9
	1024	19.0	68.4	0.9	21.8	3962.9	2.6



(a) Jetson Nano



(b) Jetson TX2

Figure 7.10: Maximum execution times of FOC algorithms in stock system scenario

In conclusion, using a stock L4T system is not appropriate to fulfill the timing requirements of a control application, even when it is configured for maximum performance and CPU isolation and memory locking techniques are used.

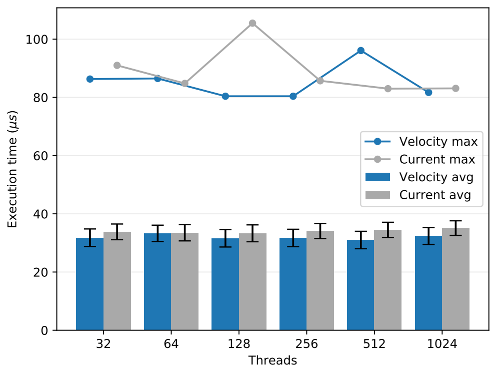
Real-time System Scenario

Table 7.2 shows the results of the timing experiments in the real-time system scenario for both Jetson platforms. As shown in the table, the maximum execution times are in an acceptable range for all the threads configurations in the Jetson TX2. However, in the Jetson Nano, there is one case where the maximum execution time of the current control algorithm is greater than 100 microseconds.

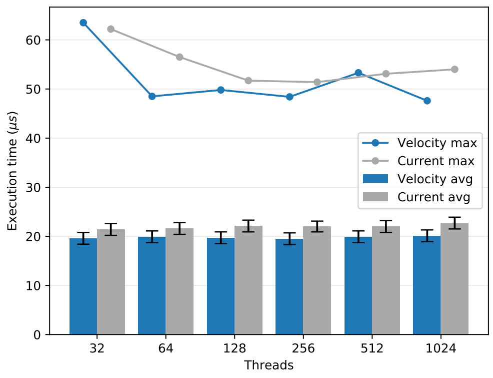
Table 7.2: Execution times of FOC algorithms in real-time system scenario (μs)

Platform	Threads	Velocity			Current		
		Avg	Max	S.D.	Avg	Max	S.D.
Jetson Nano	32	31.8	86.3	3.0	33.8	91.0	2.7
	64	33.3	86.5	2.8	33.5	84.8	2.8
	128	31.6	80.4	3.0	33.3	105.5	2.9
	256	31.7	80.4	3.0	34.1	85.7	2.6
	512	31.0	96.1	3.0	34.5	83.0	2.6
	1024	32.4	81.7	2.9	35.1	83.1	2.5
Jetson TX2	32	19.6	63.5	1.2	21.4	62.2	1.2
	64	19.9	48.5	1.2	21.6	56.5	1.2
	128	19.7	49.8	1.2	22.1	51.7	1.2
	256	19.5	48.4	1.2	22.0	51.4	1.1
	512	19.9	53.3	1.2	22.0	53.1	1.2
	1024	20.1	47.6	1.2	22.7	54.0	1.2

Even when the real-time configurations we used in this scenario dramatically improved the maximum execution times in all the cases, the timing behavior still has room for improvement. As shown in Figure 7.11, there is a big difference between the average execution time and the maximum execution time in all the cases. Moreover, the behavior of the maximum execution times does not seem to be affected by the number of threads being executed.



(a) Jetson Nano



(b) Jetson TX2

Figure 7.11: Execution times of FOC algorithms in real-time system scenario

In conclusion, using L4T with a real-time kernel and applying the configurations we used in this scenario can be enough to achieve the timing requirements of a control application. However, there is still a source of latency that causes the maximum execution times to be greater than expected. This latency could be related to the interaction of the application with the GPU runtime system when launching the GPU kernels. In the following scenario, we evaluate a workaround for this problem.

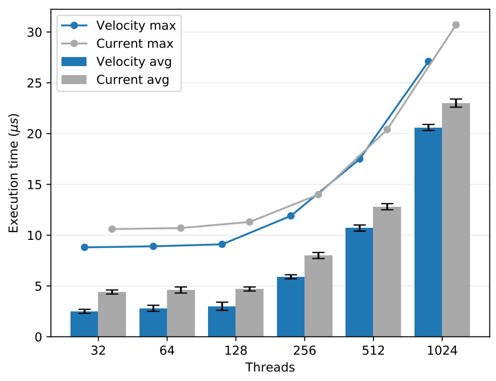
Persistent Kernel Scenario

In this scenario, we use the same operating system and configurations used in the real-time system scenario. However, in the implementation of the control algorithms, we replace the traditional kernel launch with a persistent kernel launch to avoid repeated interactions with the GPU runtime system when executing the control iterations. Table 7.3 shows the results of the experiments for both Jetson platforms. As shown in the table, the maximum execution times are in an acceptable range for all the threads configurations in both platforms. Moreover, the standard deviation indicates that most of the results are close to the average execution time in all cases.

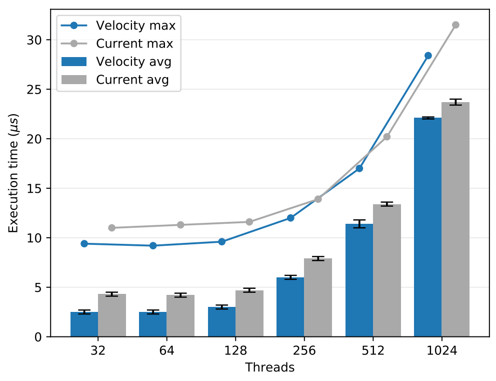
Table 7.3: Execution times of FOC algorithms in persistent kernel scenario (μs)

Platform	Threads	Velocity			Current		
		Avg	Max	S.D.	Avg	Max	S.D.
Jetson Nano	32	2.5	8.8	0.2	4.4	10.6	0.2
	64	2.8	8.9	0.3	4.6	10.7	0.3
	128	3.0	9.1	0.4	4.7	11.3	0.2
	256	5.9	11.9	0.2	8.0	14.0	0.3
	512	10.7	17.5	0.3	12.8	20.4	0.3
	1024	20.6	27.1	0.3	23.0	30.7	0.4
Jetson TX2	32	2.5	9.4	0.2	4.3	11.0	0.2
	64	2.5	9.2	0.2	4.2	11.3	0.2
	128	3.0	9.6	0.2	4.7	11.6	0.2
	256	6.0	12.0	0.2	7.9	13.9	0.2
	512	11.4	17.0	0.4	13.4	20.2	0.2
	1024	22.1	28.4	0.1	23.7	31.5	0.3

To evaluate the scalability of this approach, Figure 7.12 shows the execution times for all the threads configurations in both platforms. It is worth noting that the X-axis in these figures is not linear, which means that the increase in execution time is not exponential as it first looks. Each thread configuration is the double of the previous one. In fact, the scalability of this approach is very efficient. In both platforms, an increment of 32$\times$ in the number of threads (from 32 to 1024 threads) produces only an increment of approximately 3$\times$ in execution time.



(a) Jetson Nano



(b) Jetson TX2

Figure 7.12: Execution times of FOC algorithms in persistent kernel scenario

In conclusion, the configurations and techniques applied in this scenario significantly improve the time determinism of control applications. Furthermore, the scalability of the proposed approach is very efficient, which makes it applicable in systems with a high number of devices to control.

7.3.2 Proof of Concept

We created a C control program including both FOC algorithms to obtain the baseline results using the Teensy platform. In this program, the velocity control function is executed once for every ten executions of the current control function. To establish a fair comparison point, the period of these control functions is defined taking into account a realistic period achievable by the Jetson TX2 platform when using the I2C bus. For this purpose, we executed experiments between the Jetson TX2 and the Teensy, sending the control data and receiving the sensor values via I2C several times, measuring their execution times.

The maximum execution time obtained for I2C send is 105 microseconds and for I2C receive is 192 microseconds, which means that approximately 300 microseconds are required just for the I2C communication. Based on this information, we define a period of 350 microseconds for the current control function, meaning that the velocity control function will execute every 3500 microseconds approximately. Using these periods, we executed the control program on the Teensy platform, establishing a reference speed of 150 rad/s and registering the motor response for 30 seconds. As shown in Figure 7.13, the motor speed reaches a steady-state in approximately 5 seconds and then oscillates around the reference value. This response is not ideal, but considering the limitations of the system, we consider it an acceptable reference point.

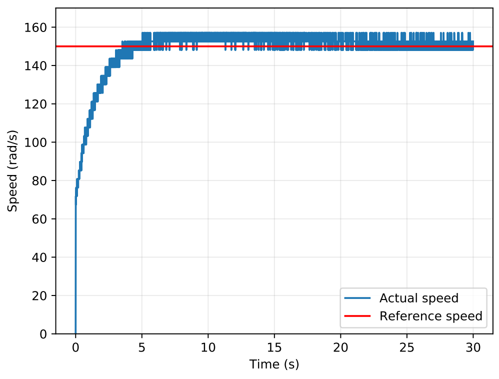


Figure 7.13: PMSM speed control using Teensy platform

After obtaining the Teensy baseline results, we created two control programs to be executed on the Jetson TX2. The first one is an adaptation of the original C program for the Teensy, which runs on the CPU of the Jetson TX2 and uses the I2C bus to communicate with the Teensy. The results of this version are considered baseline results for the Jetson TX2 platform. We executed this version with a control period of 350 microseconds on the Jetson TX2 using L4T 32.5 with real-time configurations. We set a reference speed of 150 rad/s and registered the motor response during 30 seconds. As shown in Figure 7.14, the motor speed reaches a steady-state in approximately 5 seconds, like in the baseline. However, the oscillations are more noticeable. Since we are using the same control algorithms, the same configurations for the PI controllers, and the same period we used in the baseline, the behavior should be similar. The main difference from the baseline setup is the use of the I2C bus to transfer control data. Therefore, we conclude that this behavior could be caused by unexpected delays in the I2C communication.

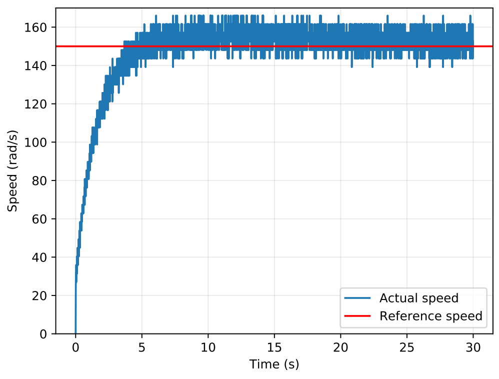
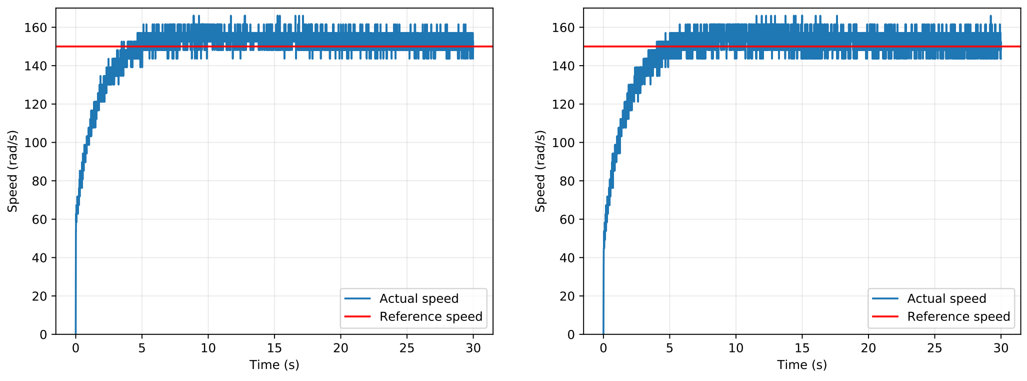


Figure 7.14: PMSM speed control using Jetson TX2 CPU

The second control program is a GPU version of the FOC algorithms implemented in a persistent kernel. In this version, both velocity and current algorithms are integrated inside the same persistent kernel, executing the velocity control every ten iterations. This way, we avoid the overhead of interacting with two persistent kernels. The host part of the CUDA program communicates with the Teensy via I2C to send and receive control data. Note that, since this is a GPU version, the control executes for at least 32 motors in parallel (because of the warp size), even when we are controlling only one real motor. For comparison purposes, we also execute the control for 1024 motors. The results are shown in Figure 7.15.



(a) Executing control for 32 motors (b) Executing control for 1024 motors

Figure 7.15: PMSM speed control using Jetson TX2 GPU

As shown in the figure, in both configurations, the motor speed reaches a steady-state in approximately 5 seconds, like in the previous cases. However, the motor speed noticeable oscillates more than in the baseline Teensy case, probably due to delays in the I2C communication. Nevertheless, even when the response of the system is not optimal, it is important to note that the GPU is able to control a real motor and get a response similar to the one obtained with the CPU version of the controller. Moreover, the GPU is able to maintain that behavior even when executing the control of up to 1024 motors in parallel.

In conclusion, existing embedded GPUs can support the timing requirements of control algorithms to implement parallel control systems, provided that the platform is correctly configured with a real-time Linux kernel and techniques like persistent threads are applied to minimize the overhead of interacting with the GPU runtime system. However, to be used in real parallel control scenarios, these platforms still lack the physical interfaces required to communicate with multiple external devices efficiently. For this reason, future embedded GPU platforms used in these scenarios, need to include the required physical interfaces.

7.4 Summary

In this chapter, we have characterized the timing of control algorithms executed in embedded GPU platforms. We have defined a case study for the parallel control of multiple motors, and we have executed the required control algorithms in three different scenarios. In the first scenario, we have executed the control algorithms in a stock L4T system configured for maximum performance, and we have concluded that the applied configurations are not enough to fulfill the timing requirements of control applications. In the second scenario, we have executed the control algorithms in a real-time L4T system, and we have observed that the timing behavior improves significantly. However, while the obtained behavior may be enough for some control applications, we have observed that it is still affected by the overhead of the interaction with the GPU runtime system. In the third scenario, we have transformed the control algorithms into persistent kernels to overcome this limitation. We have significantly improved the timing behavior with the proposed approach, and we have proved that its scalability is very efficient.

To validate the feasibility of the proposed approach in a real-life scenario, we have created a proof of concept prototype to control a real motor using an embedded GPU platform. We have concluded that, while existing embedded GPU platforms can provide the computational power and timing requirements of parallel control systems, they will need to obtain hardware support for the physical interfaces required to communicate with multiple external devices before they are adopted in this domain.

8 Assessment and Improvement of Model-Based Design for GPU-Accelerated Control Systems

As discussed in the Background Chapter, MBD is the preferred choice to attain by construction the safety and security requirements of critical systems. Recently, MBD tools have been enhanced with GPU code generation capabilities, which can be used to leverage the computational performance of these high-performance accelerators. So far in this Thesis we have seen potential future use cases of GPUs in the critical systems domain, particularly for control systems. Critical systems engineers are getting interested in GPU adoption, seeking to satisfy their ever increasing performance requirements. However, there is no analysis in the literature showing whether model-based GPU code generation tools are ready for the design of such systems.

In this chapter, we analyze the suitability of commercial MBD toolsets by designing and deploying a model-based parallel control case study on embedded GPU platforms, similar to the hand-developed one we presented in Chapter 7. We evaluate the generated code in functionality, resource consumption and scalability terms. Similar to our contribution in the timing characterization chapter, while our results show promising feasibility and scalability evidence, they also reveal shortcomings in resource consumption 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 embedded GPUs. Finally, we propose a source-to-source transformation tool that takes the GPU code generated by the MBD tools and automatically applies the improvements we suggest.

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

8.1.1 Preliminaries

The primary objective of this work is to assess the capabilities of MBD tools regarding GPU code support for the implementation of parallel real-time control systems. To carry out a comprehensive assessment, we opted to implement another case study from the control domain, similar to the hand-developed one we presented in Chapter 7. The main difference is that while in the previous case study we deployed our case study in a real platform, driving a real motor, in this chapter we follow a complete MBD approach, starting with model-in-the-loop simulation and we gradually refine it for execution on the GPU platform. However, the motor in this case is simulated in order to reduce the complexity of the evaluation, since the possibility of controlling a real motor has been already demonstrated.

We are interested specifically in the evaluation of parallel code generation capabilities of a GPU-compatible MBD toolchain 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 [62] and LabView with its GPU Analysis Toolkit [93]. 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 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 optimized 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-compatible MBD tool – towards the development of a real-time application, such as a control system.

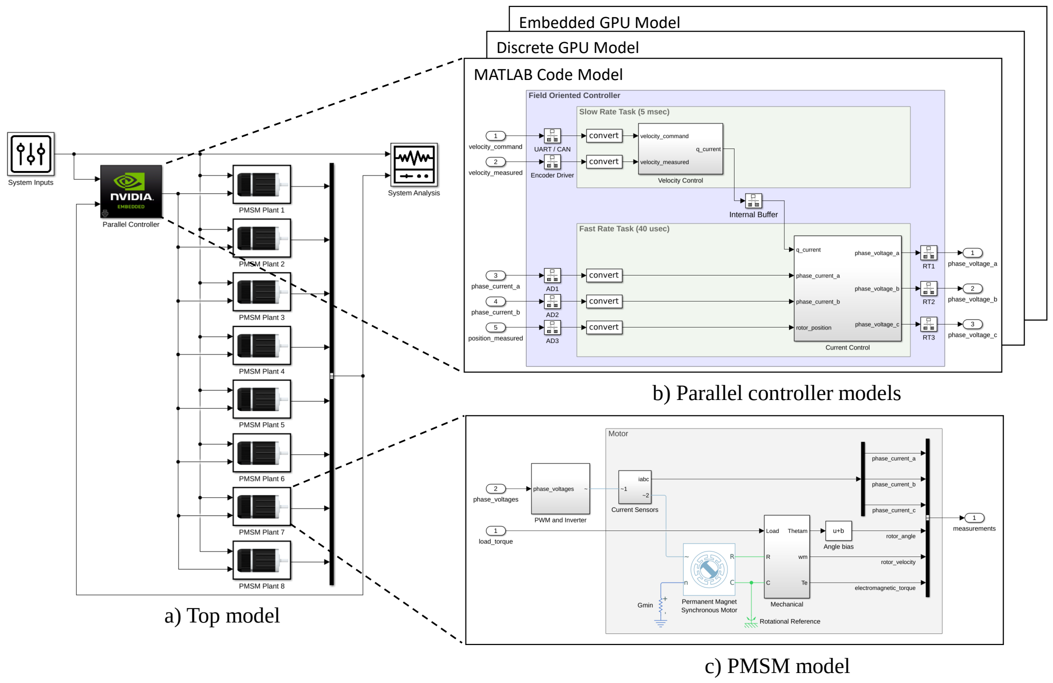


Figure 8.1: Simulink model of a parallel PMSM FOC controller

8.1.2 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 PMSMs, as shown in Figure 8.1a. 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 [90]. Note however that while our initial model which is described next uses 8 motors, in Section 8.3 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 system and controlled together. Moreover, this is the maximum number of threads per block supported by a single 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.

In our implementation, we follow the classic model-based development process with gradual refinement as introduced in Section 2.2. First, we start with a mathematical model validating its correct behavior 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 behavior 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 analyze its memory and timing properties.

In our top model shown in Figure 8.1a, each one of the 8 PMSM plants has the internal structure shown in Figure 8.1c. 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 PWM generation and a three-phase inverter circuit to simulate the commutation needed to produce rotation in the motor.

As shown in Section 7.1, PMSMs are controlled using the field oriented control technique, which is composed by a velocity control loop and a current control loop. Based on these control loops, we defined the structure of a field oriented controller subsystem, as shown in Figure 8.1b. In this configuration, the output of the velocity 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 8.1a, the Parallel Controller block is a Simulink *variant* subsystem on which we implemented three different models for the field oriented control structure, as shown in Figure 8.1b. The three models implemented in the Parallel Controller are discussed next.

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 6 and 7 with MATLAB code, 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 behavior 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 8.2.

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, at 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 [94]. 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 8.2.4. Then, in Section 8.3 we propose improvements for the generated code, which we implement and evaluate experimentally, showing their effectiveness.

8.2 Evaluation

8.2.1 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 equipped with an NVIDIA GeForce GTX 1650Ti discrete GPU. For the final embedded GIL evaluation, we used the same two NVIDIA Jetson platforms we used in the previous chapter. The details of each embedded platform are provided in Table 8.1. For the performance

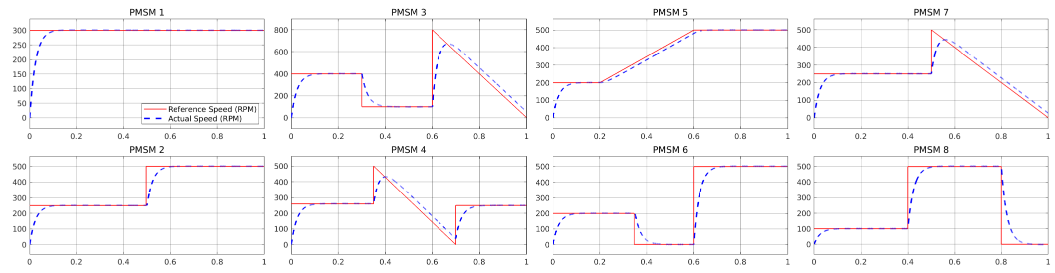


Figure 8.2: Reference and actual speeds of parallel PMSMs

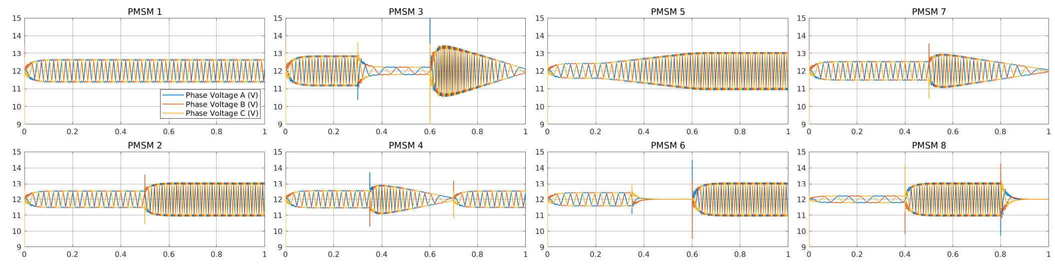


Figure 8.3: Phase voltages of parallel PMSMs

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 memory swapping disabled. To guarantee the maximum performance, `jetson_clocks` has been enabled with the maximum `nvpmodel` profile for each embedded GPU platform. In summary, we apply all the methods which according to our evaluation performed in Chapter 7 show an improvement in the GPU timing.

8.2.2 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 8.2 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

Table 8.1: 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

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 8.2 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 8.3 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 because they do not offer any additional value except confirming the functional equivalence of the MATLAB model and the generated CUDA code.

8.2.3 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 the parallel control algorithm is executed in the embedded GPU. As stated in Section 8.2.2 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 means that in cases such as our application where multiple CUDA kernels are used, their execution is serialized. 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.

8.2.4 Evaluation of Generated CUDA Code

Performance of generated CUDA code

In order to evaluate the actual performance of the generated CUDA 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 a resolution in nanoseconds.

GPU Coder can generate CUDA code which uses either the discrete or the unified memory models. As we discussed earlier in this Thesis, 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, it requires the driver and system software to manage coherence, and software managed coherence is by nature non-deterministic and not recommended in a safe context according to NVIDIA [25]. Regardless, we perform our evaluation with both memory models, since in Chapter 6 we have shown that unified memory can offer the potential of improvements of both timing and memory consumption, depending on the application and the target embedded GPU platform.

In the evaluation, we identified that GPU Coder generates `cudaMemcpy` calls to transfer data between CPU and GPU when using unified memory, which in this model are unnecessary even when using discrete GPUs, given that the CUDA runtime 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 model, removing these unnecessary `cudaMemcpy` calls.

Table 8.2 shows the execution times of the code generated for the discrete and unified memory models 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.

As stated in Section 7.1, the sampling rates needed for the FOC algorithms depend on the physical characteristics of the inverters and the motors. The period of the velocity control loop is usually around a few milliseconds, while the period of the current control loop is typically inferior to 100 microseconds. As shown in Table 8.2, 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.

Table 8.2: Execution time of generated CUDA code (μs)

Platform	Memory mode	Velocity			Current		
		Avg	Max	S.D.	Avg	Max	S.D.
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
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

Table 8.3: Profiling results for discrete memory model (μs)

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

Table 8.4: Profiling results for unified memory model (μs)

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

To better understand the maximum execution times, we executed some iterations of the control loops using the NVIDIA `nvprof` profiler. Tables 8.3 and 8.4 show the results reported by `nvprof` for the discrete and unified memory models respectively. Table 8.5 shows the profiling results for the fixed unified memory code.

In all 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

Table 8.5: Profiling results for unified memory mode fixed (μs)

Platform	Call name	Velocity			Current		
		Avg	Min	Max	Avg	Min	Max
Nano	Kernel execution	2.8	2.7	3.0	4.1	3.7	6.9
	cudaLaunchKernel	76.1	65.8	94.4	81.1	65.5	104.9
	cudaDeviceSynchronize	47.6	44.7	52.5	54.8	49.3	61.5
TX2	Kernel execution	2.4	2.4	2.6	3.5	3.4	3.7
	cudaLaunchKernel	52.0	42.8	72.2	60.4	51.3	80.5
	cudaDeviceSynchronize	28.1	24.9	31.8	32.4	30.2	39.4

time is spent on `cudaMemcpy` and kernel launch/synchronization calls. Therefore, if the time spent on these calls is reduced or eliminated, it is feasible to achieve the timings required for the control system. Based on this analysis, on Section 8.3 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 used GMAI to evaluate its memory consumption. 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 8.3. 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.

8.3 Improvement of Generated CUDA Code

As we have seen in the previous section, the generated GPU code by MATLAB/Simulink is suboptimal in terms of memory and timing for deployment in safety critical sys-

Table 8.6: Execution time of improved CUDA code versions (μs)

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

tems. For this reason, in this section we perform improvements and evaluate their impact. First we apply these improvements manually, and next we present the implementation of a source-to-source tool which performs the same modifications in an automated way.

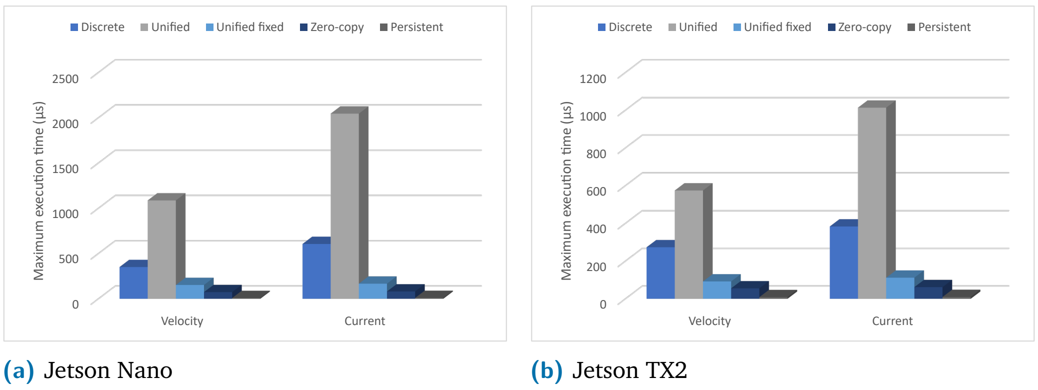
8.3.1 Manual Improvement of Generated CUDA Code

In embedded platforms where CPU and GPU share the same physical memory such as the ones we consider in our work, the memory copy overhead can be eliminated using the zero-copy memory model. 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 [39]. 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.

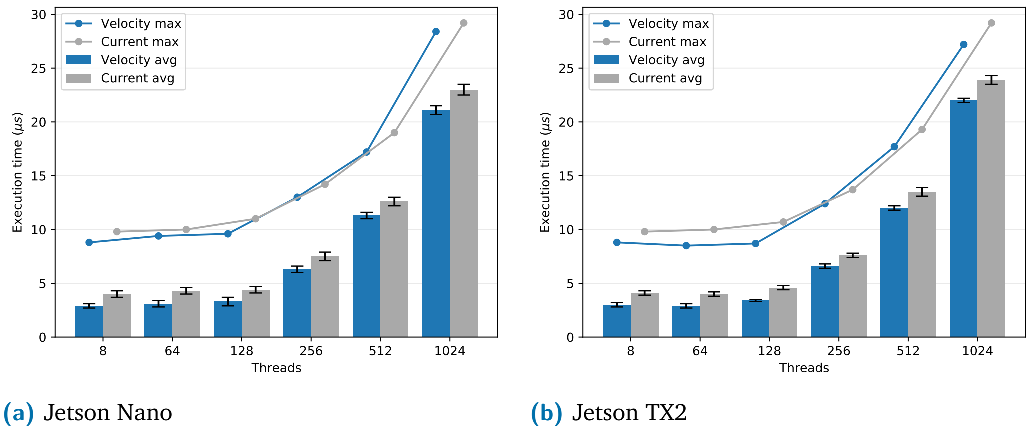
Based on these two approaches, we modified the generated code in two steps, creating two versions in order to evaluate the benefit obtained from each one. In the first step, we replaced the traditional memory allocations with zero-copy allocations to avoid using `cudaMemcpy` calls. In the second step, besides using zero-copy memory, we replaced also the kernel launch/synchronization with a persistent kernel launch. Table 8.6 shows the resulting execution times of the control algorithms with these improvements. Figure 8.4 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 significantly



(a) Jetson Nano (b) Jetson TX2

Figure 8.4: Maximum execution times of generated code and proposed improvements



(a) Jetson Nano (b) Jetson TX2

Figure 8.5: Performance scalability of improved CUDA code

faster on both platforms. This improvement by an order of magnitude can be beneficial for even tighter control scenarios.

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 8.5 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.

8.3.2 Automatic Improvement of Generated CUDA Code

After validating that our proposed modifications effectively improve the suitability of the generated CUDA code to be applied in control systems, we opted to implement a solution to apply these improvements into the generated source code automatically.

In terms of functionality, the two approaches provide identical results, therefore in this subsection we only focus on the implementation of our solution.

We created a source-to-source transformation tool using the LibTooling and LibASTMatchers libraries from the Clang framework. LibTooling provides mechanisms to support the development of standalone Clang tools. On the other hand, LibASTMatchers provides a domain-specific language to define *AST matchers*, which are predicates used to locate patterns in the Clang abstract syntax tree.

We applied the following methodology to create the tool: first, we used GPU Coder to generate CUDA code from a control function written in MATLAB code. Then, we identified the locations in the source code where the proposed improvements should be inserted. At each location, we identified which parts of the generated code were reusable and which modifications were needed to insert the improvements. Then, we defined AST matchers to find each location automatically, and we implemented the corresponding functions to apply the transformations to the source code. Finally, we put together all the AST matchers and the code transformation functions using a LibTooling structure to create a standalone Clang tool.

Our source-to-source transformation tool extends the GPU Coder workflow, as shown in Figure 8.6. First, GPU Coder takes as input a control algorithm written in MATLAB. Then, it generates CUDA code corresponding to the MATLAB function. In this process, GPU Coder generates extra files for initialization, termination, and data types management, like other Mathworks code generation tools. However, from this file structure, we only take the CUDA file where the control algorithm is implemented. Our tool takes this file as input, modifies the CUDA code to add the proposed improvements, and creates a *main* function to transform the CUDA source file into a standalone program.

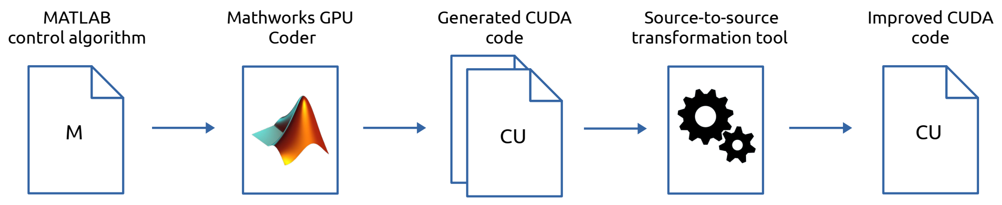


Figure 8.6: CUDA code generation and automatic improvement workflow

To illustrate the code transformation process, we will use as an example the MATLAB function for velocity control shown in Listing 8.1. First, we used GPU Coder to generate CUDA code from this function, and then, we used our tool to improve the generated code. The sections of interest of the generated CUDA code and the corresponding modifications applied by our source-to-source transformation tool are shown next.

```
function [q_current, v_mem_out] = velocityControl(v_command, v_measured, dt, v_kp,
↪ v_ki, v_mem_in)
    %#codegen
    coder.gpu.kernelfun();
    v_error = v_command - v_measured;
    v_mem_out = v_mem_in + (v_error * dt);
    q_current = (v_kp .* v_error) + (v_ki .* v_mem_out);
end
```

Listing 8.1: MATLAB function for velocity control

Kernel Prototype

The first section of interest in the generated CUDA code is the declaration of the kernel prototype, which is shown in Listing 8.2. This prototype corresponds to the GPU kernel that contains the code to control the velocity.

```
static __global__ void velocityControl_kernel1(const float v_ki[32], const float
↪ v_kp[32], const float dt, const float v_mem_in[32], const float
↪ v_measured[32], const float v_command[32], float q_current[32], float
↪ v_mem_out[32]);
```

Listing 8.2: Original kernel prototype

Since one of the proposed improvements is the transformation of the CUDA kernel into a persistent kernel, the only modification needed in this section is the insertion of two extra variables that are used to control the persistent kernel: `gpu_locked` and `terminated`. Listing 8.3 shows the resulting code after applying the source-to-source transformation tool.

```
static __global__ void velocityControl_kernel1(const float v_ki[32], const float
↪ v_kp[32], const float dt, const float v_mem_in[32], const float
↪ v_measured[32], const float v_command[32], float q_current[32], float
↪ v_mem_out[32], volatile int* gpu_locked, volatile int* terminated);
```

Listing 8.3: Transformed kernel prototype

```
 1  static __global__ __launch_bounds__(32, 1) void velocityControl_kernel1( const
    ↪  float v_ki[32], const float v_kp[32], const float dt, const float
    ↪  v_mem_in[32], const float v_measured[32], const float v_command[32], float
    ↪  q_current[32], float v_mem_out[32])
 2  {
 3      unsigned long threadId;
 4      int i;
 5      threadId = static_cast<unsigned long> (mwGetGlobalThreadIndexInXDimension());
 6      i = static_cast<int>(threadId);
 7      if (i < 32) {
 8          float f;
 9          float f1;
10          f = v_command[i] - v_measured[i];
11          f1 = v_mem_in[i] + f * dt;
12          v_mem_out[i] = f1;
13          f = v_kp[i] * f + v_ki[i] * f1;
14          q_current[i] = f;
15      }
16  }
```

Listing 8.4: Original kernel implementation

Kernel Implementation

The next section of interest in the generated CUDA code is the kernel implementation, which is shown in Listing 8.4. This section contains the CUDA code corresponding to the MATLAB velocity control algorithm shown in Listing 8.1. As with the kernel prototype, the modifications applied into this section of code are the ones needed for the implementation of a persistent kernel. Listing 8.5 shows the resulting code after applying the source-to-source transformation tool.

First, the variables `gpu_locked` and `terminated`, which are needed to control the persistent kernel, are inserted as extra arguments in the kernel header. Then, the calculation of the `threadId` variable in line 5 is replaced with a more straightforward calculation. We do this modification because the original calculation uses the `mwGetGlobalThreadIndexInXDimension` function, which is defined in an external Mathworks library. Finally, we surround the control algorithm with the persistent kernel structure: an infinite loop (line 8), the blocking code to wait for work and finalize the kernel (lines 9 to 11), and the control execution code to synchronize the persistent threads (lines 21 to 26).

```
static __global__ __launch_bounds__(32, 1) void velocityControl_kernel1( const float v_ki[32], const float v_kp[32], const float dt, const float v_mem_in[32], const float v_measured[32], const float v_command[32], float q_current[32], float v_mem_out[32], volatile int* gpu_locked, volatile int* terminated)
{
    unsigned long threadId;
    int i;
    threadId = blockDim.x * blockIdx.x + threadIdx.x;
    i = static_cast<int>(threadId);
    if (i < 32) {
        while (1) {
            while (*gpu_locked) { }
            if (*terminated)
                break;

            float f;
            float f1;
            f = v_command[i] - v_measured[i];
            f1 = v_mem_in[i] + f * dt;
            v_mem_out[i] = f1;
            f = v_kp[i] * f + v_ki[i] * f1;
            q_current[i] = f;

            if (threadId == 0) {
                __threadfence();
                __threadfence_system();
                *gpu_locked = 1;
            }
            __syncthreads();
        }
    }
}
```

Listing 8.5: Transformed kernel implementation

Main Function

On the host side, GPU Coder generates a control function with the same name as the MATLAB function used as input. This function is meant to be called from an external program sending as arguments the host variables, which have to be previously allocated. The function allocates device memory, transfers data from the host arguments to the device allocations, launches the kernel, copies the results from device memory to host memory, and finally frees the device allocations. Unfortunately, this means that all these operations must be executed on each control iteration, which adds unnecessary overhead. Our approach is to transform this function into the main function, where we enclose the control iterations into an inner loop to simplify their execution. In addition, we replace the traditional memory model with the zero-copy memory model and the kernel launch with a persistent kernel launch.

The source-to-source transformation tool uses the information provided in the header of the original control function, shown in Listing 8.6, and creates a main function with all the needed zero-copy allocations, as shown in Listing 8.7.

```
void velocityControl(const float v_command[32], const float v_measured[32], float
↪ dt, const float v_kp[32], const float v_ki[32], const float v_mem_in[32],
↪ float q_current[32], float v_mem_out[32])
```

Listing 8.6: Original control function

```
1   int main(int argc, char* argv[])
2   {
3       cudaSetDeviceFlags(cudaDeviceMapHost);
4
5       float* v_command;
6       float* v_measured;
7       float dt;
8       float* v_kp;
9       float* v_ki;
10      float* v_mem_in;
11      float* q_current;
12      float* v_mem_out;
13      volatile int* gpu_locked;
14      volatile int* terminated;
15
16      cudaHostAlloc((void**)&v_command, 32 * sizeof(float), cudaHostAllocMapped);
17      cudaHostAlloc((void**)&v_measured, 32 * sizeof(float), cudaHostAllocMapped);
18      cudaHostAlloc((void**)&v_kp, 32 * sizeof(float), cudaHostAllocMapped);
19      cudaHostAlloc((void**)&v_ki, 32 * sizeof(float), cudaHostAllocMapped);
20      cudaHostAlloc((void**)&v_mem_in, 32 * sizeof(float), cudaHostAllocMapped);
21      cudaHostAlloc((void**)&q_current, 32 * sizeof(float), cudaHostAllocMapped);
22      cudaHostAlloc((void**)&v_mem_out, 32 * sizeof(float), cudaHostAllocMapped);
23      cudaHostAlloc((void**)&gpu_locked, sizeof(int), cudaHostAllocMapped);
24      cudaHostAlloc((void**)&terminated, sizeof(int), cudaHostAllocMapped);
```

Listing 8.7: Main function created from original control function

First, the header of the function is replaced with the typical header of a main function (line 1). Then, for each array argument passed to the original function, the tool creates a pointer declaration (lines 5 to 12) and inserts the corresponding `cudaHostAlloc` call to allocate zero-copy memory (lines 16 to 22). The size and type of the original arrays are used to determine the amount of zero-copy memory to allocate. In this section, the tool also inserts the the call to enable the `cudaDeviceMapHost` flag needed to allocate zero-copy memory (line 3), and the declarations (lines 13 and 14) and memory allocations (lines 23 and 24) for the `gpu_locked` and `terminated` variables.

Device Pointers

In the original control function, GPU Coder generates arrays of pointers and `cudaMalloc` calls to allocate all the device variables, as shown in Listing 8.8.

```
    float(*gpu_q_current)[32];
    float(*gpu_v_ki)[32];
    float(*gpu_v_kp)[32];
    float(*gpu_v_command)[32];
    float(*gpu_v_measured)[32];
    float(*gpu_v_mem_in)[32];
    float(*gpu_v_mem_out)[32];
    if (!isInitialized_velocityControl) {
        velocityControl_initialize();
    }
    cudaMalloc(&gpu_v_mem_out, 128UL);
    cudaMalloc(&gpu_q_current, 128UL);
    cudaMalloc(&gpu_v_command, 128UL);
    cudaMalloc(&gpu_v_measured, 128UL);
    cudaMalloc(&gpu_v_mem_in, 128UL);
    cudaMalloc(&gpu_v_kp, 128UL);
    cudaMalloc(&gpu_v_ki, 128UL);
```

Listing 8.8: Original device pointers and allocations

In the zero-copy memory model, these device allocations are no longer needed. However, it is still necessary to define device pointers and link them to the previous zero-copy allocations shown in Listing 8.7, using the `cudaHostGetDevicePointer` function. The source-to-source transformation tool takes advantage of this code structure and transforms the arrays of pointers into single pointers, and the `cudaMalloc` calls into `cudaHostGetDevicePointer` calls, as shown in Listing 8.9.

```
    float* gpu_q_current;
    float* gpu_v_ki;
    float* gpu_v_kp;
    float* gpu_v_command;
    float* gpu_v_measured;
    float* gpu_v_mem_in;
    float* gpu_v_mem_out;

    cudaHostGetDevicePointer(&gpu_v_mem_out, v_mem_out, 0);
    cudaHostGetDevicePointer(&gpu_q_current, q_current, 0);
    cudaHostGetDevicePointer(&gpu_v_command, v_command, 0);
    cudaHostGetDevicePointer(&gpu_v_measured, v_measured, 0);
    cudaHostGetDevicePointer(&gpu_v_mem_in, v_mem_in, 0);
    cudaHostGetDevicePointer(&gpu_v_kp, v_kp, 0);
    cudaHostGetDevicePointer(&gpu_v_ki, v_ki, 0);
```

Listing 8.9: Transformed device pointers and assignments

Kernel Launch

The kernel launch generated by GPU Coder in the original control function has the typical kernel launch structure used when working with the traditional memory model, as shown in Listing 8.10. First, data is transferred from host memory to device memory using the cudaMemcpy function (lines 1 to 5). Then, the kernel is launched (line 7), and finally, the results are copied back from device memory to host memory (lines 9 and 10).

```
1     cudaMemcpy(*gpu_v_ki, v_ki, 128UL, cudaMemcpyHostToDevice);
2     cudaMemcpy(*gpu_v_kp, v_kp, 128UL, cudaMemcpyHostToDevice);
3     cudaMemcpy(*gpu_v_mem_in, v_mem_in, 128UL, cudaMemcpyHostToDevice);
4     cudaMemcpy(*gpu_v_measured, v_measured, 128UL, cudaMemcpyHostToDevice);
5     cudaMemcpy(*gpu_v_command, v_command, 128UL, cudaMemcpyHostToDevice);
6
7     velocityControl_kernel1<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(*gpu_v_ki,
      ↪  *gpu_v_kp, dt, *gpu_v_mem_in, *gpu_v_measured, *gpu_v_command,
      ↪  *gpu_q_current, *gpu_v_mem_out);
8
9     cudaMemcpy(q_current, *gpu_q_current, 128UL, cudaMemcpyDeviceToHost);
10    cudaMemcpy(v_mem_out, *gpu_v_mem_out, 128UL, cudaMemcpyDeviceToHost);
```

Listing 8.10: Original memory transfers and kernel launch

The source-to-source transformation tool takes this section of code and transforms it into a persistent kernel launch as shown in Listing 8.11. First, the tool removes all the cudaMemcpy calls since they are not needed in the zero-copy memory model. Then, the tool inserts the initialization of the terminated and gpu_locked variables (lines 1 and 2) to control the persistent kernel. The persistent kernel launch (line 3) is a slightly modified version of the original kernel launch. The arrays of pointers are replaced with single pointers, and the gpu_locked and terminated control variables are inserted as extra arguments. In addition to the persistent kernel launch, the tool inserts the main loop structure where the control iterations are executed (lines 4 to 9). The structure indicates where the user should add the code to modify the input values (line 5) and read the output values (line 8). In the middle, the control structure has instructions to indicate to the persistent kernel that there is new work to process (line 6) and to wait for the GPU to finish the execution of the current iteration (line 7). Finally, the tool inserts the instructions needed to terminate the persistent kernel when the execution exits from the main control loop (lines 10 to 12).

```
    *terminated = 0;
    *gpu_locked = 1;
    velocityControl_kernel1<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(gpu_v_ki,
    ↪   gpu_v_kp, dt, gpu_v_mem_in, gpu_v_measured, gpu_v_command, gpu_q_current,
    ↪   gpu_v_mem_out, gpu_locked, terminated);
    while (1) {
        /* Modify input values here*/
        *gpu_locked = 0;
        while (!(*gpu_locked)) { }
        /* Read output values here*/
    }
    *terminated = 1;
    *gpu_locked = 0;
    cudaDeviceSynchronize();
```

Listing 8.11: Transformed kernel launch and control loop

Memory Deallocations

The last part of the original control function contains the deallocation of the memory reserved for device variables, as shown in Listing 8.12. Since the original control function receives the host allocations as arguments, releasing them is not the responsibility of the control function. Instead, it is assumed that those allocations will be released in the external program that calls the control function.

```
    cudaFree(*gpu_v_ki);
    cudaFree(*gpu_v_kp);
    cudaFree(*gpu_v_mem_in);
    cudaFree(*gpu_v_measured);
    cudaFree(*gpu_v_command);
    cudaFree(*gpu_q_current);
    cudaFree(*gpu_v_mem_out);
```

Listing 8.12: Original device memory deallocations

The source-to-source transformation tool takes these device deallocations and transforms them into zero-copy deallocations using the `cudaFreeHost` function, as shown in Listing 8.13. It also inserts two extra calls to deallocate the two persistent kernel control variables (lines 8 and 9) and the `return` instruction to finalize the main function (line 11).

```
    cudaFreeHost(v_ki);
    cudaFreeHost(v_kp);
    cudaFreeHost(v_mem_in);
    cudaFreeHost(v_measured);
    cudaFreeHost(v_command);
    cudaFreeHost(q_current);
    cudaFreeHost(v_mem_out);
    cudaFreeHost((void*)gpu_locked);
    cudaFreeHost((void*)terminated);

    return 0;
```

Listing 8.13: Transformed zero-copy memory deallocations

8.4 Summary

In this chapter, 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 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 optimized 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.

In order to overcome these limitations, we evaluated the two aforementioned proposed solutions and we provided a source-to-source transformation tool, which can apply them automatically over the generated GPU code. To our knowledge, this is the first automated tool capable of transforming a GPU application to the *persistent threads* GPU programming model, which is beneficial not only for control systems but also for other GPU applications with real-time requirements.

Conclusions and Future Work 9

The real-time control systems industry is moving towards the consolidation of multiple computing systems into fewer and more powerful ones, aiming for a reduction in size, weight, and power. The increasing demand for higher performance in other critical domains like autonomous driving has led the industry to recently include embedded GPUs for the implementation of advanced functionalities. The highly parallel architecture of GPUs could also be leveraged in the control systems industry to develop more advanced, energy-efficient, and scalable control systems. However, the closed-source and non-deterministic nature of GPUs complicates the resource provisioning analysis required for the implementation of critical real-time systems. On the other hand, there is no indication of the integration of GPUs in the traditional development cycle of control systems, which is oriented to the use of a model-based design approach. Motivated by these challenges, in this thesis, we contributed to the state of the art of real-time control systems towards the adoption of embedded GPUs by providing tools to facilitate the resource provisioning analysis and the integration in the model-based design development cycle. In this chapter, we summarize the main contributions of our research and its impact, and we present open areas for future research work.

9.1 Summary of Contributions

This thesis contributed to the state of the art of real-time control systems by facilitating the adoption of embedded GPUs to implement future parallel control systems. We achieved this goal by providing methodologies and tools to ease the resource provisioning analysis of embedded GPUs and their integration in the model-based development cycle. In particular, this thesis provided the following contributions:

- The first contribution of this thesis defined a methodology to extract information about the internal properties of closed-source GPU memory allocators. Based on this methodology, we presented GMAI, a tool that automatically applies our reverse-engineering techniques and, based on the extracted information, enables the computation of the real amount of memory consumed by GPU applications. We applied GMAI to a wide range of GPUs of different vendors, showing its compatibility with both CUDA and OpenCL. In addition, we applied GMAI in two automotive case studies to show how it can be used to make an accurate resource provisioning analysis.

- With the purpose of extending the memory analysis functionality of GMAI, we analyzed the GPU memory behavior and allocation patterns of several GPU benchmarking suites, which can serve not only the safety critical domain that is the focus of this Thesis, but also the broader GPU research community. As part of this work, we created a library to characterize the use of dynamic memory in GPU applications. This library uses the information extracted with GMAI to keep track of the allocated memory and generates a visual representation that shows the evolution of the different types of dynamic memory in GPU applications. We applied our tool to three popular GPU benchmark suites that include GPU applications from different domains. As a result of the characterization, we identified memory allocation patterns that could be modified to improve memory consumption and performance.

- Based on the results obtained with the memory characterization of GPU benchmarking suites as representative of real GPU applications, we designed XeroZerox, a tool that minimizes the memory consumption and memory management overhead of GPU applications when executed on embedded GPU platforms. XeroZerox automatically changes memory allocation patterns of GPU applications to patterns that are favorable for embedded GPUs. We applied XeroZerox to the Rodinia benchmarks suite and showed that it can reduce the memory consumption to approximately 50% in legacy GPU applications executed in embedded GPUs. Furthermore, we showed that using a centralized memory model like zero-copy memory or unified memory impacts in different ways the performance of GPU applications, depending on the hardware coherency mechanisms present in the underlying platform, therefore creating a trade-off between memory consumption, predictability and performance, depending on the particular generation of embedded GPU platform which is used.

- After finishing the resource analysis in terms of memory, we focused our research to the timing domain. For this purpose, we performed a timing characterization of control algorithms executed on embedded GPU platforms. We presented a case study for the parallel control of multiple motors and executed the required control algorithms in three different scenarios. After the evaluation, we identified a set of configurations that allowed us to obtain an acceptable behavior in terms of timing and scalability. To validate the feasibility of the proposed approach in a real-life scenario, we created a proof of concept prototype to control a real motor using an embedded GPU platform. We concluded that existing embedded GPUs can provide the computational power and timing requirements to implement parallel control systems. However, we observed that they still require hardware support to communicate with multiple external devices before being adopted in the control domain.

- Finally, in the last contribution of this thesis, we analyzed the integration of embedded GPUs with Model-Based Design tools. First, we designed a parallel control system model using MATLAB-Simulink and validated the integration with GPU hardware for PIL/GIL testing. Then, we evaluated the GPU code generation capabilities and identified some inefficiencies in the generated GPU code. In particular, while the generated code is functional, its timing behavior is not optimal for control systems due to the use of memory copies and the overhead of the GPU kernel launches. Based on the results we obtained in the timing characterization analysis, we proposed and manually applied some improvements to the generated code. Finally, after validating that our proposed modifications effectively improved the suitability of the generated GPU code to be applied in control systems, we implemented a source-to-source transformation tool to automatically apply the improvements into the generated code. To our knowledge, this is the first automated tool capable of transforming a GPU application to the *persistent threads* GPU programming model, which is beneficial not only for control systems but also for other GPU applications with real-time requirements.

9.2 Impact

Until the beginning of this Thesis, reverse engineering of the black-box behavior of GPUs in the literature was only limited to their real-time properties. However, this thesis has opened the door to research about the reverse engineering of GPU black-box behavior in terms of memory management which is of equal importance for safety critical systems, and as we have showed it is closely related to their real-time behavior. The reason is that although dynamic memory allocation is prohibited or at least discouraged for use in safety-critical systems, it is an essential part required for the use of GPUs.

This Thesis has served as a starting point for a new PhD thesis topic which is currently ongoing and extends our work to cover extensively memory management in embedded GPUs and its interaction with GPU hardware. We believe that as work in this area progresses, more open problems will be identified and targeted by other PhD theses and research from other universities, research institutions and companies across all safety critical domains, which are working towards the adoption of GPUs in their products, such as in the automotive sector.

Moreover, our analysis of the memory allocation behavior of GPU applications as performed in 3 of the major benchmarking suites, has the potential to influence not only the safety-critical domain, but also the wider GPU research domain including HPC. In particular, understanding the memory usage patterns and their resource requirements can benefit not only application developers but also research in programming models and compilers.

Regarding the integration of GPUs in the model-based development cycle, our work has served as a basis for a project proposal on the integration in model-based design tools used for space systems, which is currently under evaluation. In this way, we can influence other domains where model-based design is extensively used and GPUs are considered for adoption. Again, this is only the beginning, since other project proposals will follow targeting other domains as well.

Apart from the theoretical contributions of this dissertation, during the development of this thesis we have developed and released as open source a series of tools that either enable reverse engineering of GPU memory allocations, or improve the memory behavior of GPU applications. This can have a significant impact on the continuation of this work by third parties, since it enables not only the replication of our observations, but also covering new platforms and extending them with new features. Moreover, our tools can provide important insights to GPU developers, both the ones targeting safety-critical applications as well as other domains such as high performance computing.

Our tools are currently used in the H2020 European Project UP2DATE, which uses embedded GPU platforms such as the ones used in this thesis. In particular, our tools are used for analysis and the optimization of the project use cases. This allows the dissemination of our work to a wider audience and an evaluation with larger applications.

Finally, another potential impact of this Thesis would be the adoption of our contributions in industry, such as the implementation our approaches within commercial or open source model-based design tools, GPU development and analysis tools and programming models, or even GPU vendors in their software stacks.

9.3 Future Work

This thesis can be extended in multiple directions which are planned for future work, while some of them are already on-going as described in the previous sub-section in the framework of a PhD Thesis and the UP2DATE project.

The first direction in which the thesis can be extended is beyond the control systems domain. In fact, already our contribution on the characterization of GPU benchmarking suites in terms of dynamic memory behavior open the door towards this path. Those analysis results, the proposed methodologies and tools presented in this thesis can be applied for the development and analysis of GPU-based real-time systems in other domains, possibly with modifications and extensions that address their special needs. As we already mentioned, this can be the design of aerospace and automotive systems, both of which used model-based design tools and consider the adoption of GPUs. Moreover, the high performance computing domain which extensively uses GPUs can benefit from our tools and methodologies, since memory management resource provisioning and behavior is important for performance reasons too.

In the real-time and safety critical domain, we plan also to cover new cases in which GPU code with real-time properties is required. In this direction, we are applying our tools within the UP2DATE project using the project's use case. Moreover, larger industrial case studies can be conducted and use of our tools with additional real GPU applications.

Moreover, we can apply our proposed reverse engineering techniques to other parts of the GPU software stack in order to obtain information to improve the development of open-source alternatives such as the Nouveau driver and covering new devices that will appear soon, such as upcoming embedded GPUs and GPU APIs like Vulkan SC targeting the automotive and other safety critical domains.

Finally, as a future work we will try to transfer the contributions of this thesis to industry, so that they are adopted for example in the code generation of model based tools, or in profilers and analysis tools for GPU codes. For this, we will exploit the large network of industrial collaborations we have in the institutions where the research of this thesis was conducted. Other potential adopters which we will target for technology transfer include companies working on embedded GPUs and safety critical domain, including compilers and development tools, GPU drivers and runtimes and real-time operating systems. In addition, we will work with industrial users developing GPU software for critical systems transferring our contributions in their applications.

Bibliography

[1] NVIDIA Corporation. *Self Driving Cars*. Accessed April 2019. URL: `https://www.nvidia.com/en-us/self-driving-cars` (cit. on p. 1).

[2] X. Yu, H. Wang, W. -. Feng, H. Gong, and G. Cao. "GPU-Based Iterative Medical CT Image Reconstructions". In: *Journal of Signal Processing Systems* 91.3-4 (2019), pp. 321–338 (cit. on p. 1).

[3] R. L. Davidson and C. P. Bridges. "Error Resilient GPU Accelerated Image Processing for Space Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 29.9 (2018), pp. 1990–2003 (cit. on p. 1).

[4] Samsung. *Exynos Auto V9*. Accessed August 2021. URL: `https://www.samsung.com/semiconductor/minisite/exynos/products/automotiveprocessor/exynos-auto-v9/` (cit. on p. 2).

[5] L. Kosmidis, C. Maxim, V. Jegu, F. Vatrinet, and F. J. Cazorla. "Industrial Experiences with Resource Management under Software Randomization in ARINC653 Avionics Environments". In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*. 2018 (cit. on p. 2).

[6] ARINC. *Avionics Application Software Standard Interface: ARINC Specification 653P1-3. Aeronautical Radio*. 2010 (cit. on p. 2).

[7] AUTOSAR. *AUTOSAR*. Accessed April 2019. URL: `https://www.autosar.org` (cit. on p. 2).

[8] Green Hills Software. *Integrity RTOS*. Accessed April 2019. 1996. URL: `https://www.ghs.com/products/rtos/integrity.html` (cit. on p. 2).

[9] R. Langner. "Stuxnet: Dissecting a Cyberwarfare Weapon". In: *IEEE Security and Privacy* 9.3 (2011), pp. 49–51 (cit. on p. 4).

[10] Symantec. *Dragonfly: Cyberespionage Attacks Against Energy Suppliers*. 2014 (cit. on p. 4).

[11] D. Mažeika and R. Butleris. "MBSEsec: Model-Based Systems Engineering Method for Creating Secure Systems". In: *Applied Sciences (Switzerland)* 10.7 (2020) (cit. on p. 4).

[12] R. Neisse, G. Steri, I. N. Fovino, and G. Baldini. "SecKit: A Model-based Security Toolkit for the Internet of Things". In: *Computers and Security* 54 (2015), pp. 60–76 (cit. on p. 4).

[13] N. Kekatos. "Formal Verification of Cyber-Physical Systems in the Industrial Model-Based Design Process". PhD thesis. Université Grenoble Alpes, 2018 (cit. on p. 4).

[14]R. Marinescu. “Model-Checking and Model-Based Testing of Automotive Embedded Systems: Starting from the System Architecture”. PhD thesis. Malardalen University, 2014 (cit. on p. 4).

[15]A. J. Calderón, L. Kosmidis, C. F. Nicolás, F. J. Cazorla, and P. Onaindia. “Understanding and Exploiting the Internals of GPU Resource Allocation for Critical Systems”. In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*. Vol. 2019-November. 2019 (cit. on p. 6).

[16]A. J. Calderón, L. Kosmidis, C. F. Nicolás, F. J. Cazorla, and P. Onaindia. “GMAI: Understanding and Exploiting the Internals of GPU Resource Allocation in Critical Systems”. In: *ACM Transactions on Embedded Computing Systems* 19.5 (2020) (cit. on p. 6).

[17]A. Jover-Alvarez, A. J. Calderón, I. Rodríguez, et al. “The UP2DATE Baseline Research Platforms”. In: *Proceedings -Design, Automation and Test in Europe, DATE*. Vol. 2021-February. 2021, pp. 1340–1343 (cit. on p. 7).

[18]A. J. Calderón, L. Kosmidis, C. F. Nicolás, J. de Lasala, and I. Larrañaga. *Assessing and Improving the Suitability of Model-Based Design for GPU-Accelerated Railway Control Systems*. Vol. 12800 LNCS. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2021, pp. 68–83 (cit. on p. 8).

[19]H. D. Kim, A. T. Perry, and P. J. Ansell. “A Review of Distributed Electric Propulsion Concepts for Air Vehicle Technology”. In: *2018 AIAA/IEEE Electric Aircraft Technologies Symposium, EATS 2018*. 2018 (cit. on p. 13).

[20]P. Schmollgruber, C. Döll, J. Hermetz, et al. “Multidisciplinary Exploration of DRAGON: an ONERA Hybrid Electric Distributed Propulsion Concept”. In: *AIAA Scitech 2019 Forum*. 2019 (cit. on p. 13).

[21]D. D. Gajski, A. Gerstlauer, S. Abdi, and G. Schirner. “Embedded System Design: Modeling, Synthesis and Verification”. In: Embedded System Design: Modeling, Synthesis and Verification. 2006, pp. 1–352 (cit. on p. 14).

[22]A. Sangiovanni-Vincentelli and G. Martin. “Platform-Based Design and Software Design Methodology for Embedded Systems”. In: *IEEE Design and Test of Computers* 18.6 (2001), pp. 23–33 (cit. on p. 14).

[23]A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi. “Benefits and Challenges for Platform-Based Design”. In: *Proceedings - Design Automation Conference*. 2004, pp. 409–414 (cit. on p. 14).

[24]J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. 1st edition. United Kingdom: Wrox Press Ltd., 2014 (cit. on p. 16).

[25]NVIDIA Corporation. *CUDA for Tegra*. Accessed September 2021. URL: `https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html` (cit. on pp. 26, 120).

[26] Intel Corporation. *Getting the Most from OpenCL 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel Processor Graphics*. Accessed October 2019. URL: `https://software.intel.com/en-us/articles/getting-the-most-from-opencl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics` (cit. on p. 27).

[27] G. A. Elliott and J. H. Anderson. "Real-World Constraints of GPUs in Real-Time Systems". In: *Proceedings - 1st International Workshop on Cyber-Physical Systems, Networks, and Applications, CPSNA 2011, Workshop Held During RTCSA 2011*. Vol. 2. 2011, pp. 48–54 (cit. on p. 27).

[28] G. A. Elliott, B. C. Ward, and J. H. Anderson. "GPUSync: A Framework for Real-Time GPU Management". In: *Proceedings - Real-Time Systems Symposium*. 2013, pp. 33–44 (cit. on p. 27).

[29] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru. "Deadline-Based Scheduling for GPU with Preemption Support". In: *Proceedings - Real-Time Systems Symposium*. Vol. 2018-December. 2019, pp. 119–130 (cit. on p. 27).

[30] H. Zhou, G. Tong, and C. Liu. "GPES: A Preemptive Execution System for GPGPU Computing". In: *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. Vol. 2015-May. 2015, pp. 87–97 (cit. on p. 27).

[31] G. Chen, Y. Zhao, X. Shen, and H. Zhou. "EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU". In: *ACM SIGPLAN Notices* 52.8 (2017), pp. 3–16 (cit. on p. 27).

[32] C. Hartmann and U. Margull. "GPUart - An Application-based Limited Preemptive GPU Real-Time Scheduler for Embedded Systems". In: *Journal of Systems Architecture* 97 (2019), pp. 304–319 (cit. on p. 27).

[33] R. Cavicchioli, N. Capodieci, M. Solieri, and M. Bertogna. "Novel Methodologies for Predictable CPU-to-GPU Command Offloading". In: *Leibniz International Proceedings in Informatics, LIPIcs*. Vol. 133. 2019 (cit. on pp. 27, 28, 100).

[34] R. Cavicchioli, N. Capodieci, and M. Bertogna. "Memory Interference Characterization between CPU Cores and Integrated GPUs in Mixed-Criticality Platforms". In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. 2017, pp. 1–10 (cit. on p. 27).

[35] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna. "SiGAMMA: Server Based Integrated GPU Arbitration Mechanism for Memory Accesses". In: *ACM International Conference Proceeding Series*. Vol. Part F131837. 2017, pp. 48–57 (cit. on p. 27).

[36] B. Forsberg, A. Marongiu, and L. Benini. "GPUguard: Towards Supporting a Predictable Execution Model for Heterogeneous SoC". In: *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*. 2017, pp. 318–321 (cit. on p. 27).

[37] W. Ali and H. Yun. "Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms". In: *Leibniz International Proceedings in Informatics, LIPIcs*. Vol. 106. 2018 (cit. on p. 27).

[38] K. Berezovskyi, K. Bletsas, and B. Andersson. “Makespan Computation for GPU Threads Running on a Single Streaming Multiprocessor”. In: *Proceedings - Euromicro Conference on Real-Time Systems*. 2012, pp. 277–286 (cit. on p. 27).

[39] K. Gupta, J. A. Stuart, and J. D. Owens. “A Study of Persistent Threads Style GPU Programming for GPGPU Workloads”. In: *2012 Innovative Parallel Computing, InPar 2012*. 2012 (cit. on pp. 27, 100, 123).

[40] N. Capodieci and P. Burgio. “Efficient Implementation of Genetic Algorithms on GP-GPU with Scheduled Persistent CUDA Threads”. In: *Proceedings - International Symposium on Parallel Architectures, Algorithms and Programming, PAAP*. Vol. 2016-January. 2016, pp. 6–12 (cit. on p. 28).

[41] A. Milluzzi and A. George. “Exploration of TMR Fault Masking with Persistent Threads on Tegra GPU SoCs”. In: *IEEE Aerospace Conference Proceedings*. 2017 (cit. on p. 28).

[42] Allen, T. *Improving Real-Time Performance with CUDA Persistent Threads (CuPer) on the Jetson TX2*. Tech. rep. Concurrent Real-Time, 2018. URL: `https://www.concurrent-rt.com/wp-content/uploads/2016/09/Improving-Real-Time-Performance-With-CUDA-Persistent-Threads.pdf` (cit. on p. 28).

[43] J. Anantpur and R. Govindarajan. “Taming Warp Divergence”. In: *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*. 2017, pp. 50–60 (cit. on p. 28).

[44] D. Troendle, T. Ta, and B. Jang. “A Specialized Concurrent Queue for Scheduling Irregular Workloads on GPUs”. In: *ACM International Conference Proceeding Series*. 2019 (cit. on p. 28).

[45] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. Donelson Smith. “GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed”. In: *Proceedings - Real-Time Systems Symposium*. Vol. 2018-January. 2018 (cit. on pp. 28, 63).

[46] M. Yang, N. Otterness, T. Amert, et al. “Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems”. In: *Leibniz International Proceedings in Informatics, LIPIcs*. Vol. 106. 2018 (cit. on pp. 28, 63).

[47] M. M. Trompouki and L. Kosmidis. “BRASIL: A High-Integrity GPGPU Toolchain for Automotive Systems”. In: *Proceedings - 2019 IEEE International Conference on Computer Design, ICCD 2019*. 2019, pp. 660–663 (cit. on p. 28).

[48] S. Alcaide, L. Kosmidis, H. Tabani, et al. “Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain”. In: *IEEE Micro* 38.6 (2018), pp. 46–54 (cit. on p. 28).

[49] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf. “Future Automotive Systems Design: Research Challenges and Opportunities”. In: *2018 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2018*. 2018 (cit. on p. 28).

[50] Y. Iwase, D. Abe, and T. Yakoh. “GPGPU Aided Method for Real-Time Systems”. In: *IEEE International Conference on Industrial Informatics (INDIN)*. 2012, pp. 841–845 (cit. on p. 28).

[51]D. Hallmans, M. Asberg, and T. Nolte. “Towards using the Graphics Processing Unit (GPU) for Embedded Systems”. In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. 2012 (cit. on p. 28).

[52]D. Hallmans, K. Sandström, M. Lindgren, and T. Nolte. “GPGPU for Industrial Control Systems”. In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. 2013 (cit. on p. 28).

[53]M. Lindgren, K. Sandström, T. Nolte, and D. Hallmans. “Applicability of Using Internal GPGPUs in Industrial Control Systems”. In: *19th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2014*. 2014 (cit. on p. 28).

[54]T. J. Maceina and G. Manduchi. “Assessment of General Purpose GPU Systems in Real-Time Control”. In: *IEEE Transactions on Nuclear Science* 64.6 (2017), pp. 1455–1460 (cit. on p. 28).

[55]M. Kozubik and P. Vaclavek. “Speed Control of PMSM with Finite Control Set Model Predictive Control Using General-purpose Computing on GPU”. In: *IECON Proceedings (Industrial Electronics Conference)*. Vol. 2020-October. 2020, pp. 379–383 (cit. on p. 28).

[56]A. Schmidt, F. Schellroth, M. Fischer, L. Allimant, and O. Riedel. “Reinforcement Learning Methods Based on GPU Accelerated Industrial Control Hardware”. In: *Neural Computing and Applications* 33.18 (2021), pp. 12191–12207 (cit. on p. 28).

[57]J. Huang. *Conversational AI, NVIDIA Jarvis*. Keynote part 5, NVIDIA Technology Conference (GTC). 2020 (cit. on p. 28).

[58]NVIDIA Corporation. *Jetson Linux*. Accessed January 2022. URL: `https://developer.nvidia.com/embedded/linux-tegra` (cit. on p. 33).

[59]S. Che, M. Boyer, J. Meng, et al. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009*. 2009, pp. 44–54 (cit. on pp. 34, 69).

[60]J. A. Stratton, C. Rodrigues, I. Sung, et al. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. IMPACT Technical Report, IMPACT-12-01. University of Illinois, at Urbana-Champaign, Mar. 2012 (cit. on pp. 34, 69).

[61]S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. “Auto-Tuning a High-Level Language Targeted to GPU Codes”. In: *2012 Innovative Parallel Computing, InPar 2012*. 2012 (cit. on pp. 34, 69).

[62]The MathWorks, Inc. *GPU Coder—Generate CUDA code for NVIDIA GPUs*. Accessed September 2021. URL: `https://www.mathworks.com/products/gpu-coder.html` (cit. on pp. 35, 114).

[63]The Clang Team. *Clang: a C language family frontend for LLVM*. Accessed January 2022. URL: `https://clang.llvm.org/` (cit. on p. 36).

[64]LLVM Developer Group. *The LLVM Compiler Infrastructure*. Accessed January 2022. URL: `https://llvm.org/` (cit. on p. 36).

[65] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. *Dynamic Storage Allocation: A Survey and Critical Review*. Vol. 986. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 1995, pp. 1–116 (cit. on pp. 41, 42).

[66] Y. Hasan and J. M. Chang. “A Tunable Hybrid Memory Allocator”. In: *Journal of Systems and Software* 79.8 (2006), pp. 1051–1063 (cit. on p. 42).

[67] V. Shah and A. Shah. *Proposed Memory Allocation Algorithm for NUMA-Based Soft Real-Time Operating System*. Vol. 814. Advances in Intelligent Systems and Computing. 2019, pp. 3–11 (cit. on p. 42).

[68] A. J. Calderón, L. Kosmidis, C. F. Nicolás, F. J. Cazorla, and P. Onaindia. *GMAI: GPU Memory Allocation Inspector*. URL: `https://github.com/ajcalderont/gmai` (cit. on p. 50).

[69] M. M. Trompouki, L. Kosmidis, and N. Navarro. “An Open Benchmark Implementation for Multi-CPU Multi-GPU Pedestrian Detection in Automotive Systems”. In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*. Vol. 2017-November. 2017, pp. 305–312 (cit. on pp. 56, 59).

[70] U. Ozgunalp. “Combination of the Symmetrical Local Threshold and the Sobel Edge Detector for Lane Feature Extraction”. In: *Proceedings - 9th International Conference on Computational Intelligence and Communication Networks, CICN 2017*. Vol. 2018-January. 2018, pp. 24–28 (cit. on p. 56).

[71] H. Vishwanathan, D. L. Peters, and J. Z. Zhang. “Traffic Sign Recognition in Autonomous Vehicles Using Edge Detection”. In: *ASME 2017 Dynamic Systems and Control Conference, DSCC 2017*. Vol. 1. 2017 (cit. on p. 56).

[72] R. Younis and N. Bastaki. “Accelerated Fog Removal from Real Images for Car Detection”. In: *2017 9th IEEE-GCC Conference and Exhibition, GCCCE 2017*. 2018 (cit. on p. 56).

[73] Free Software Foundation. *The GNU Allocator*. Accessed April 2019. 2019. URL: `%7Bhttps://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html%7D` (cit. on p. 58).

[74] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. “Hoard: A Scalable Memory Allocator for Multithreaded Applications”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*. 2000, pp. 117–128 (cit. on p. 62).

[75] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. Hwu. “XMalloc: A Scalable Lock-Free Dynamic Memory Allocator for Many-Core Machines”. In: *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*. 2010, pp. 1134–1139 (cit. on p. 62).

[76] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. Hwu. “Scalable SIMD-Parallel Memory Allocation for Many-Core Machines”. In: *Journal of Supercomputing* 64.3 (2013), pp. 1008–1020 (cit. on p. 62).

[77] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. “ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU”. In: *2012 Innovative Parallel Computing, InPar 2012*. 2012 (cit. on p. 62).

[78] S. Widmer, D. Wodniok, N. Weber, and M. Goesele. “Fast Dynamic Memory Allocator for Massively Parallel Architectures”. In: *ACM International Conference Proceeding Series*. 2013, pp. 120–126 (cit. on p. 62).

[79] H. Wong, M. -. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. “Demystifying GPU Microarchitecture through Microbenchmarking”. In: *ISPASS 2010 - IEEE International Symposium on Performance Analysis of Systems and Software*. 2010, pp. 235–246 (cit. on p. 63).

[80] X. Mei and X. Chu. “Dissecting GPU Memory Hierarchy through Microbenchmarking”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 72–86 (cit. on p. 63).

[81] X. Chen, A. Slowinska, and H. Bos. “Who Allocated My Memory? Detecting Custom Memory Allocators in C Binaries”. In: *Proceedings - Working Conference on Reverse Engineering, WCRE*. 2013, pp. 22–31 (cit. on p. 63).

[82] A. Slowinska, T. Stancescu, and H. Bos. “Howard: A Dynamic Excavator for Reverse Engineering Data Structures”. In: *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS’11)* (2011) (cit. on p. 63).

[83] M. Kerrisk. *Linux Programmer’s Manual - ld.so(8)*. Accessed August 2021. 2021. URL: `https://man7.org/linux/man-pages/man8/ld.so.8.html` (cit. on p. 66).

[84] S. Chamberlain. *Using LD, The GNU Linker*. Accessed August 2021. 1994. URL: `https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html` (cit. on p. 67).

[85] Free Software Foundation. *The C Preprocessor - Standard Predefined Macros*. Accessed August 2021. 2021. URL: `%7Bhttps://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html#Standard-Predefined-Macros%7D` (cit. on p. 67).

[86] Microsemi Corporation. *SmartFusion Field Oriented Control of Permanent Magnet Synchronous Motors Using HALL and Encoder*. Tech. rep. Microsemi Corporation, 2012. URL: `https://www.microsemi.com/document-portal/doc_view/130910-sf-foc-pmsm-using-hall-and-encoder-ug` (cit. on pp. 94, 96).

[87] B. P. Welford. “Note on a Method for Calculating Corrected Sums of Squares and Products”. In: *Technometrics* 4.3 (1962), pp. 419–420 (cit. on p. 101).

[88] T. F. Chan, G. H. Golub, and R. J. Leveque. “Statistical Computing: Algorithms for Computing the Sample Variance: Analysis and Recommendations”. In: *American Statistician* 37.3 (1983), pp. 242–247 (cit. on p. 101).

[89] Open Source Automation Development Lab (OSADL) eG. *HOWTO: Create a latency plot from cyclictest histogram data*. Accessed February 2022. URL: `https://www.osadl.org/Create-a-latency-plot-from-cyclictest-hi.bash-script-for-latency-plot.0.html` (cit. on p. 101).

[90] Texas Instruments, Inc. *C2000 Delfino MCU F28379D LaunchPad*. Accessed September 2021. URL: `https://www.ti.com/tool/LAUNCHXL-F28379D` (cit. on pp. 103, 115).

[91] SparkFun Electronics. *Teensy 3.6 Development Board*. Accessed February 2022. URL: `https://www.sparkfun.com/products/14057` (cit. on p. 103).

[92] Embedded Linux Community. *Jetson PWM*. Accessed February 2022. URL: `https://elinux.org/Jetson/PWM` (cit. on p. 103).

[93] National Instruments Corp. *GPU Analysis Toolkit*. Accessed September 2021. URL: `https://zone.ni.com/reference/en-XX/help/373575A-01/lvgpu/lvgpu/` (cit. on p. 114).

[94] The MathWorks, Inc. *MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms*. Accessed September 2021. URL: `https://www.mathworks.com/help/supportpkg/nvidia/index.html` (cit. on p. 117).