# Evaluation of the Parallel Computational Capabilities of Embedded Platforms for Critical Systems

**Author:**
Alvaro Jover-Alvarez

**Advisor:**
Dr. Leonidas Kosmidis

Department of Computer Architecture
Universitat Politècnica de Catalunya (UPC)
Barcelona Supercomputing Center (BSC)

High Performance Computing
Master in Innovation and Research in Informatics (MIRI)

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

Computer Architecture - Operating Systems Group (CAOS)
Barcelona Supercomputing Center (BSC)

October 21, 2021

# Acknowledgements

This work is funded by the European Commission's Horizon 2020 programme under the UP2DATE project (grant agreement 871465). It is also partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grants PID2019-107255GB, FJCI-2017-34095, TIN2015-65316-P and the HiPEAC Network of Excellence. This work has also received funding from the European Space Agency (ESA) under the GPUs for Space (GPU4S) Project, answer to the ESA ITT AO/1-9010/17/NL/AF.

This thesis wouldn't have been possible without my advisor Dr. Leonidas Kosmidis and his enormous support, numerous advices and help. I would like also to thank him for giving me the opportunity to participate in the GPU4S and the Horizon 2020 UP2DATE project.

I would like also to thank all the collaborators that contributed in one way or another to the project: ESA, IKERLAN, OFFIS, IAV, TTTech Auto, Marelli, CAF Signalling and Airbus Defence and Space.

I would also like to thank specially Leonidas Kosmidis and Ivan Rodriguez Ferrandez from the CAOS group and Alejandro J. Calderón from IKERLAN for their help in the development of the different publications that contributed to this project.

In closing, I would like to highlight the great support of my family and friends, for their constant support during this pandemic, without them this would not have been possible.

"*Happiness can be found even in the darkest of times, if one only remembers to turn on the light.*"

**Albus Dumbledore**

# Abstract

Modern critical systems need higher performance which cannot be delivered by the simple architectures used so far. Latest embedded architectures feature multi-cores and Graphics Processing Units (GPUs), which can be used to satisfy this need. In this thesis we parallelise relevant applications from multiple critical domains represented in the GPU4S benchmark suite, and perform a comparison of the parallel capabilities of candidate platforms for use in critical systems.

In particular, we port the open source GPU4S Bench benchmarking suite in the OpenMP programming model, and we benchmark the candidate embedded heterogeneous multi-core platforms of the H2020 UP2DATE project, NVIDIA TX2, NVIDIA Xavier and Xilinx Zynq Ultrascale+, in order to drive the selection of the research platform which will be used in the next phases of the project. Our result indicate that in terms of Central Processing Unit (CPU) and GPU performance, the NVIDIA Xavier is the highest performing platform.

# Contents

# List of Figures

# List of Tables

# Acronyms

**ADAS** Advanced Driver-Assistance Systems.

**AI** Artificial Intelligence.

**AMBA** Advanced Microcontroller Bus Architecture.

**API** Application Programming Interface.

**ASIC** application-specific integrated circuits.

**BSC** Barcelona Supercomputing Center.

**COTS** Commercial off-the-shelf.

**CPI** clock-per-instruction.

**CPU** Central Processing Unit.

**DLA** Deep Learning Accelerator.

**DMA** Direct Memory Access.

**DSP** Digital Signal Processing.

**ECC** Error-Correcting Code.

**EDF** Earliest deadline first.

**EMB$^2$** Embedded Multicore Building Blocks.

**ESA** European Space Agency.

**FPGA** Field-Programmable Gate Array.

**FPU** Floating-Point Unit.

**GLSL** OpenGL Shading Language.

**GPGPU** General-purpose computing on graphics processing units.

**GPU** Graphics Processing Unit.

**GPU4S** GPUs for Space.

**HPC** High Performance Computing.

**HSM** Hardware Security Module.

**IoT** Internet Of Things.

**ISA** Instruction Set Architecture.

**KSCAF** Khronos Safety Critical Advisory Forum.

**MAC** Multiply–accumulate.

**MCCPS** Mixed Criticality Cyber-Physical System.

**MIPS** Million Instructions Per Second.

**MMU** Memory Management Unit.

**MPU** Memory Protection Unit.

**MTAPI** Multicore Task Management API.

**NASA** National Aeronautics and Space Administration.

**NGMP** Next Generation Microprocessor.

**OBPMark** On-Board Processing Benchmarks.

**OTASU** Over-The-Air Software Updates.

**PMC** Performance Monitor Counter.

**pthreads** POSIX threads.

**PVA** Programmable Vision Accelerator.

**RAW** Read After Write.

**RTOS** Real-time operating system.

**SASE** Safety and Security.

**SMP** Symmetric Multiprocessing System.

**SoC** system on a chip.

**SoC** Systems-on-Chip.

**TDP** Thermal Design Power.

**VPU** Vector Processing Unit.

**WAR** Write After Read.

**WAW** Write After Write.

**WCET** Worst Case Execution Time.

# Chapter 1

# Introduction

## 1.1 Introduction and Motivation

Modern safety critical systems like the ones we can find in the automotive, railway, avionics and aerospace domains require high performance, in order to provide the computational power needed to meet the criteria for the implementation of modern advanced functionalities, such as increased autonomy.

Traditionally, the processors used in these domains were very simple, single-core processors without advanced architectural features, however the increasing need for performance, has led these industries to start considering more advanced architectures. Heterogeneous platforms are particularly attractive since multi-core processors alone cannot provide the level of performance which can be delivered by general purpose accelerators such as GPUs and Field-Programmable Gate Arrays (FPGAs).

This Master Thesis has been performed within the scope of two projects at the Barcelona Supercomputing Center (BSC):

- In the context of the GPU4S project, funded by the ESA, which studies the applicability of embedded GPUs in the space domain, we extended the open source GPU4S benchmarking suite by including OpenMP parallelisations for each of the space-relevant algorithms described in [79].
- In the context of the UP2DATE project, funded by the European Commission's Horizon 2020 programme, which studies the update features of mixed criticality, high performance platforms in safety-critical systems, we performed a performance benchmarking study of the initial selection of candidate platforms using the GPU4S benchmarking suite. This helped to identify the most performant platform, in order to guide the selection of platform which was considered in the project during the next phases.

## 1.2 Contributions

The main contributions of this Master Thesis consists on the OpenMP CPU parallelisation of the GPU4S test suite and a performance benchmarking study of several modern high-performance heterogeneous embedded platforms suitable for critical systems.

In this thesis we also provide an in-depth review of the current state-of-the-art hardware and programming models used today in various critical sectors.

We place all this effort in the context of the European Horizon2020 UP2DATE project, where part of the work was undertaken in its initial stage is to identify a suitable platform for its next phases. For this reason, we compare the performance capabilities of three potential next-generation candidates: NVIDIA Xavier, NVIDIA TX2 and Xilinx Zynq Ultrascale+ ZCU102.

In addition, we explain the importance, benefits and challenges of online updating a critical device, we carry a performance benchmarking evaluation of the three candidate platforms under test employing the GPU4S benchmark suite developed in the context of GPU4S project, and finally, we describe the rationale behind the selection of the baseline research devices for the UP2DATE project, which was published in [53].

## 1.3 Thesis Organisation

This Master Thesis is organised as follows: Chapter 2 reviews the current and preceding embedded hardware platforms and parallel compute methodologies for Critical Systems. Chapter 3 describes more in depth the UP2DATE project, including the platform selection requirements and the work done in the context of the GPU4S project to benchmark the candidate devices.

Then, Chapter 4 describes the experimental setup for the benchmarking of the candidate boards defined in the previous chapter, followed by Chapter 5, where we present and discuss the results of the experiments performed in the context of the UP2DATE project.

Chapter 6 draws the conclusions and presents future avenues for this work and finally, Chapter 7 includes a list of the publications produced in the context of this thesis.

# Chapter 2

# State of the Art

Parallel computing is the science of distributing computation throughout the different available resources present in a system or series of systems. Within this paradigm we find multiple programming models that define how certain workloads can be distributed and how this can be specified from the programmer point of view. Since the scope of this thesis encompasses embedded platforms, we are going to focus our attention only in those programming models that could be used in the context of an isolated single device. Thus, in Section 2.2 we will study some of the options available for CPU parallel computing and Section 2.3 elaborates about some of the platforms available for GPU computing.

## 2.1 Hardware platforms for Critical Systems

Embedded devices have been dominant over the past decades in many critical sectors, like avionics, automotive or railway. These devices should comply with a series of strict certification processes according to domain-specific safety standards, which makes the development and cost of the applications targeted to the domain very expensive.

The computational demands of modern applications and the coexistence of non-critical and critical software in a *mixed-criticality system* require higher performance that can only be achieved by means of multi-core devices and mono-core devices with higher frequency.

However, single-core platforms with higher frequency present a series of issues that make them non-eligible in several domains: increased sensibility to electromagnetic interference, low reliability of thermal dissipation fans and cooling systems volume and weight. Another very relevant issue is that the industry is moving towards Commercial off-the-shelf (COTS) multi-core devices, making single-core platforms obsolete in many domains except for micro-controllers or strictly real-time processors. For this reason, even very conservative markets (avionics) are considering the use of multi-core devices [21].

Nevertheless, the adoption of multi-core devices in critical sectors is not simple, as most safety certification standards target single-core architectures. The CAST-32A [22] certification guidance document for multicores in avionics was presented in 2016 to provide guidance for software planning, development and verification in multi-core devices. However, as [9] mention in their study, CAST-32A still presents a series of questions and challenges to resolve under current COTS multi-core systems.

In this section we will divide the current state of the art of critical embedded architectures in two categories: Section 2.1.1 references some of the most relevant mono-core and multi-core embedded boards used nowadays in different critical sectors, and Section 2.1.2, in which we will elaborate on some of the heterogeneous multi-core platforms targeting critical markets manufactured in the last few years.

### 2.1.1 Traditional architectures

Among all the current non-heterogeneous critical architectures we find a good amount of multi-core processors as chip manufacturers shifted to this paradigm due to the reasons exposed in Section 2.1. The automotive domain is one of the first safety-critical industries that adopted this new methodology by implementing specific multi-core hardware designs like the AURIX Tricore Multicontroller. These type of microcontrollers excell at delivering determinism for the target platform, but often lack the essential features to run full operating systems, as many of the candidate platforms disregard the use of Memory Management Units (MMUs).

Another relevant sector that is adopting the use of multi-core processors is the aerospace industry. Most of the critical and real-time tasks currently performed in this area are carried out with single-core processors, such as the LEON2, due to the high criticality and real-time nature of the applications (like a rocket thruster). However, tasks that require lower criticality but need performance, e.g. instrument control or data processing, are already performed with multi-core processors such as the LEON3 or LEON4.

#### 2.1.1.1 LEON

The LEON microprocessor family consists of a series of radiation-tolerant CPUs, designed originally by the ESA, that implements the SPARC V8 Instruction Set Architecture (ISA) developed by Sun Microsystems [24].

The selection of SPARC as the primary ISA for the LEON family was presented in the final report of the ERC32 program, where the main objective was to reuse an existing processor architecture to minimize both software and hardware development cost [78], which also adds to the other benefits of using this ISA [40]:

- Open architecture without patents or license fees [63].
- Well designed and documented.
- Easy to implement.
- Established software standard.
- Available reference design (Cypress 601).

The original objectives with the development of this processor family were to provide an open, portable and non-proprietary processor design capable to meet future requirements for performance, software compatibility and low system cost, while preserving functional safety by providing error detection and error handling mechanisms [12].

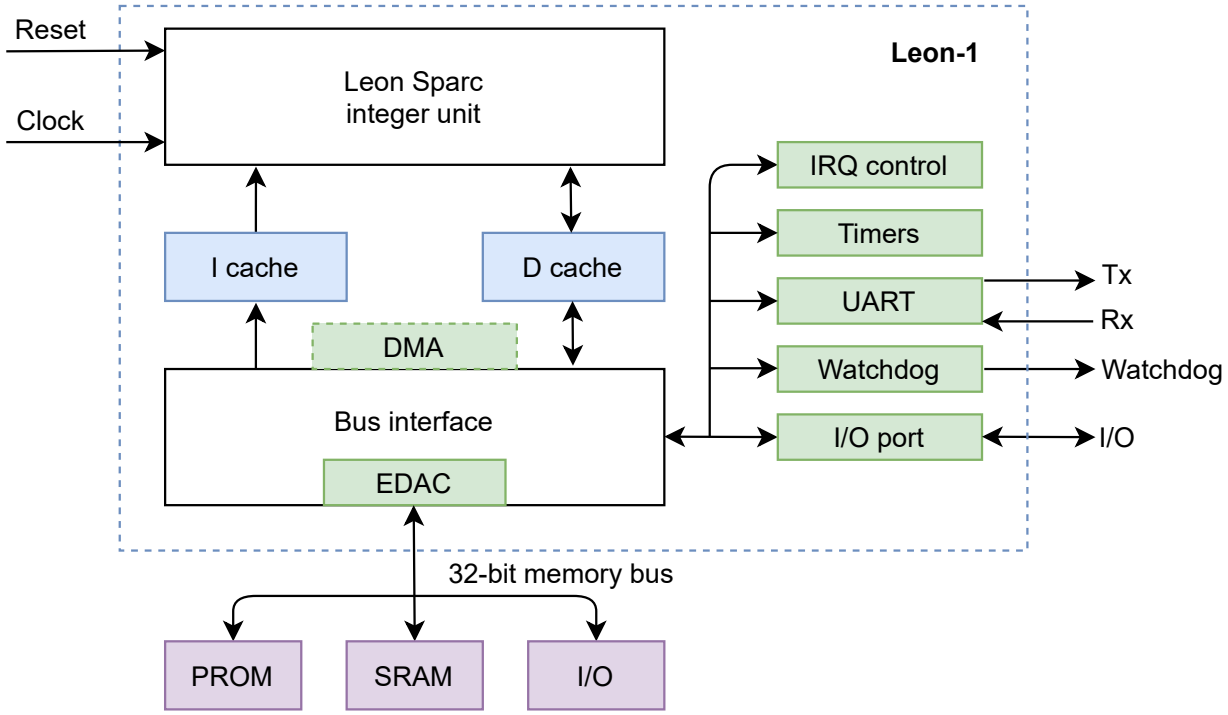



Figure 2.1: ESA's Free Leon-1 block diagram. Original figure credit: European Space Agency.

As we can see in Figure 2.1, the first Leon design included a Sparc-compatible integer unit, separated data and instruction caches, an interrupt controller, two 24-bit timers, two UARTs, a 16-bit I/O port, a power-down feature, write protection, a watchdog timer and a 32-bit memory bus with EDAC, PROM and SRAM support.

The next paragraphs will focus on two successor LEON designs, the LEON2-FT CPU and the LEON4 CPU, single-core and multi-core architectures respectively.

**LEON2** is the second synthesisable VHDL model from the LEON family developed by ESA. The experience with LEON1 lead to the development of the LEON2 core, which improved drastically the base design of the LEON core by including the following features:

- On-chip PCI and SDRAM controllers.
- Multi-set caches
- Hardware multiply and divide units.
- On-chip debug support.
- Advanced Microcontroller Bus Architecture (AMBA) buses.

In addition, LEON2 also improved the overall performance delivered by the original LEON design. Initially, LEON1 operated at 50 MHz with a clock-per-instruction (CPI) figure of 1.6, resulting in an average of 30 Million Instructions Per Second (MIPS); while LEON2 ran at 100 MHz with a CPI of 1.5, achieving an average performance of 70 MIPS. Most of these performance improvements came through the switch from the $0.35\mu m$ CMOS process in LEON1 to the $0.18\mu m$ in LEON2 [39].

Based on the LEON2 technology, ESA developed an extension of the core named LEON2-FT to include advanced fault-tolerance features to withstand arbitrary errors without loss of data, which is required for processors operating in harsh environments like space, where radiation can introduce errors.

The LEON2 (non-FT) model is no longer maintained. It is superceded by LEON2-FT, and the subsequently released LEON models (LEON3, LEON4) [7]. Among other satellites, this core was used in ESA's Intermediate eXperimental Vehicle (IXV) [28] and China's Chang'e-4 lander [5].

**LEON4** is the latest multi-core design released by Gaisler Research that improves the design of the preceding versions of the core. The Symmetric Multiprocessing System (SMP) support present in the LEON4 core comes from it's preceding version, LEON3, in which we could find the following new features according to its specification [33]:

- Advanced 7-stage pipeline.
- Multiply–accumulate (MAC) units.
- High-performance and fully pipelines IEEE-754 Floating-Point Unit (FPU)[1].
- Configurable caches.

[1] `https://www.gaisler.com/index.php/products/ipcores/ieee754fpu?task=view&id=138`

- Local instruction and data scratch pad RAM.
- SPARC Reference MMU (SRMMU) with configurable TLB.
- Up to 125 MHz in FPGA and 400 MHz on 0.13 $\mu m$ application-specific integrated circuits (ASIC) technologies.
- Large range of software tools: compilers, kernels, simulators and debug monitors.

The new iteration of the LEON family, LEON4, included the following new features: static branch prediction, optional L2 cache, 64-bit or 128-bit path to AMBA interface and higher performance (1.7 DMIPS/MHz as opposed to 1.4 DMIPS/MHz of LEON3) [36] [8].

One of the devices that benefit from the design of the LEON4 is the GR740 Quad-Core LEON4 SPARC V8 Processor (Fig. 2.2) [34], which was released by the fourth quarter of 2020. This device is part of the ESA roadmap for standard microprocessor components and is the first radiation hardened implementation of the ESA Next Generation Microprocessor (NGMP) system on a chip (SoC) architecture [6].

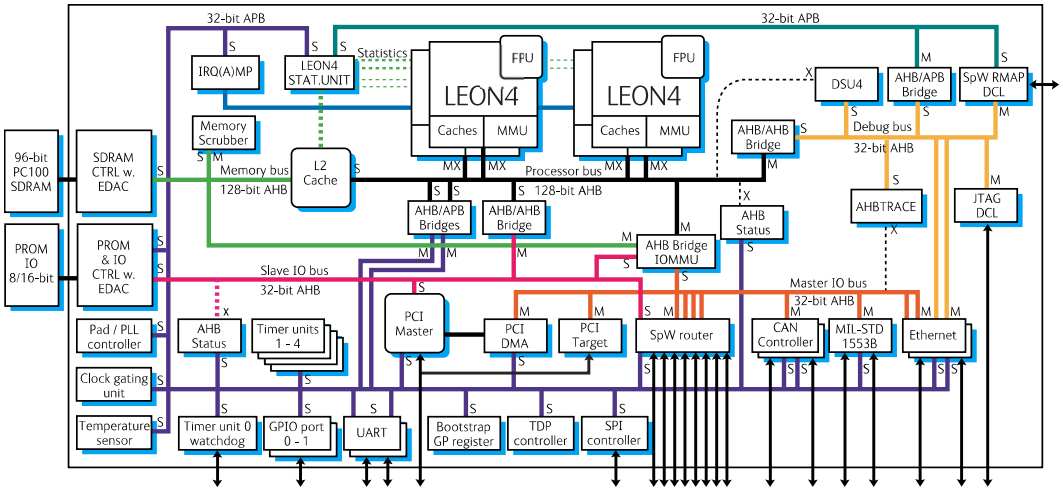



Figure 2.2: GR740 Quad-Core LEON4 Processor. Original figure credit: Cobham Gaisler.

The GR740 has been selected for multiple missions, including ESA's Copernicus and National Aeronautics and Space Administration (NASA)'s WFIRST [35].

The LEON family of processors is evolving nowadays presenting newer designs like the LEON5 [37] or the RISC-V based NOEL-V core [38], enhancing the original design with modern SoC technology while retaining the essence of the original LEON processor.

#### 2.1.1.2 AURIX Tricore Multicontroller TC397x

The 32-bit-Microcontroller Tricore family unites the elements of a RISC core and a Digital Signal Processing (DSP) in a single chip. These devices are designed to provide safety and security features for a wide range of automotive and industrial applications. One of the main features of the AURIX Tricore family is their high configurability, enabling the end user to choose between a wide range of memories, peripheral sets, temperatures and packaging options [50].

In terms of performance, the TC39x offers 6 cores running at 300 MHz. The cores are organised in 2 clusters, divided in 4 lock-stepped cores for functional safety and 2 non lock-stepped cores. Figure 2.3 shows the block diagram of the TX39x lead device:

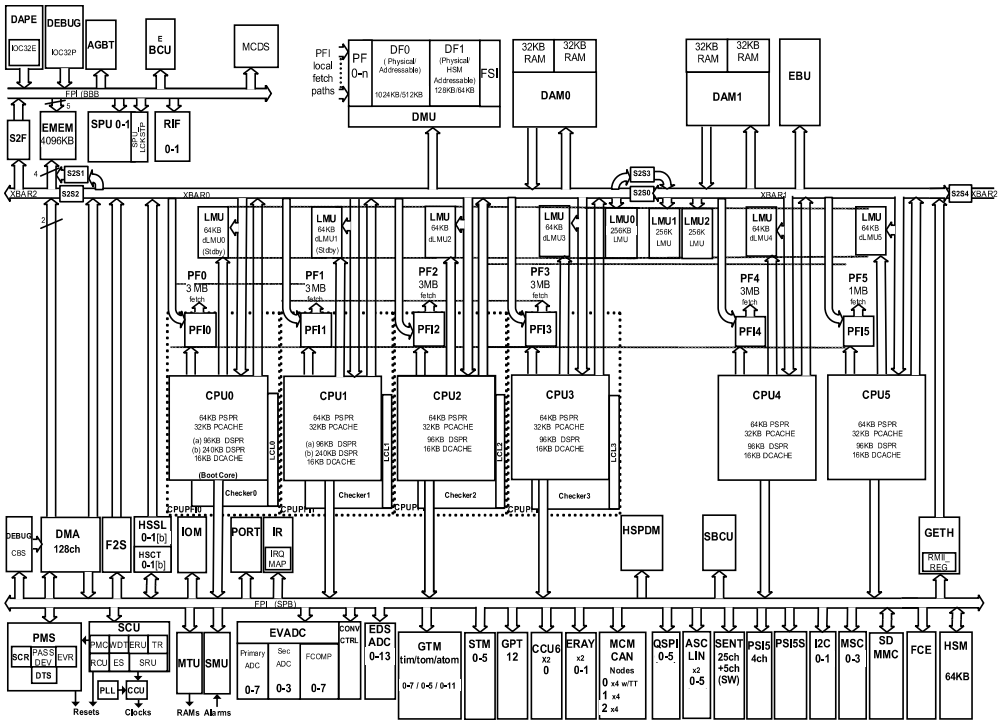



Figure 2.3: Block Diagram of TC39x. Original figure credit: infineon.

In addition to the dual lockstep functionality, all RAM memories included in the device are protected with Error-Correcting Code (ECC). Furthermore, it counts with a number of security features included in a eVita[2] compliant Hardware Security Module (HSM): secure boot process, true random number generator and cryptographic accelerators. Finally, the device also enables granular power management to control the power consumption of the different units present in the platform [51].

All these features and the compliance with ASIL-D ISO 26262 made this board a perfect candidate for the HORIZON2020 UP2DATE European project [53] [57], as part of the automotive use cases.

[2] `https://www.evita-project.org/`

### 2.1.2 Parallel and heterogeneous architectures for critical systems

As mentioned in the introduction to this Section, the current state of the art is populated by multi-core processors, so any new heterogeneous architecture today is also multi-core. This tendency has dominated the industry since the last two decades as single-core processors did not deliver the sought-after performance [42] that multi-core processors could deliver. Unlike traditional multi-cores which consist of a number of identical cores, heterogeneous architectures include more that a single type of processing elements, such as different types of (multi-core) processors, as well as accelerators like GPUs and FPGAs. In this subsection, we are going to review different heterogeneous architectures that are targeting critical systems.

#### 2.1.2.1 NVIDIA Jetson AGX Xavier

The NVIDIA Jetson AGX Xavier is one of the latest embedded systems released by NVIDIA. This board is focused on the development of certified autonomous machines driven by AI [29]. It also stands out for its high configurability to operate in different power modes (Table 2.1), which makes the device suitable for systems that require low power consumption [58].

| **Property** | **Mode** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **MAXN** | **10W** | **15W** | **30W** | **30W** | **30W** | **30W** | **15W*** |
| Power budget | *n/a* | *10W* | *15W* | *30W* | *30W* | *30W* | *30W* | *15W* |
| Mode ID | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* |
| Online CPU | *8* | *2* | *4* | *8* | *6* | *4* | *2* | *4* |
| CPU maximal frequency (MHz) | *2265,6* | *1200* | *1200* | *1200* | *1450* | *1780* | *2100* | *2188* |
| GPU TPC | *4* | *2* | *4* | *4* | *4* | *4* | *4* | *4* |
| GPU maximal frequency (MHz) | *1377* | *520* | *670* | *900* | *900* | *900* | *900* | *670* |
| DLA cores | *2* | *2* | *2* | *2* | *2* | *2* | *2* | *2* |
| DLA maximal frequency (MHz) | *1395.2* | *550* | *750* | *1050* | *1050* | *1050* | *1050* | *115.2* |
| PVA cores | *2* | *0* | *1* | *1* | *1* | *1* | *1* | *1* |
| PVA maximal frequency (MHz) | *1088* | *0* | *550* | *760* | *760* | *760* | *760* | *115.2* |
| Memory maximal frequency (MHz) | *2133* | *1066* | *1333* | *1600* | *1600* | *1600* | *1600* | *1333* |

Table 2.1: NVPModel clock configuration for Jetson AGX Xavier 16GB and 32GB

The board features eight customised 64-bit ARMv8.2 Carmel cores developed by NVIDIA, and

includes a Volta GPU with eight stream multiprocessors. It also contains eight tensor cores, an ASIC specialised in MAC operations capable to perform 64x FP16 MACs or 128x INT8 MACs per cycle [93].

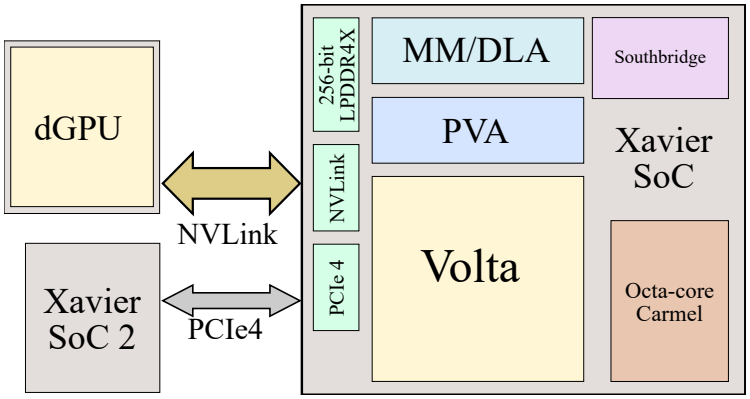



Figure 2.4: Nvidia Xavier block diagram. Original figure credit: Wikichip.

As we can see in Figure 2.4, the NVIDIA Xavier also comes with a Deep Learning Accelerator (DLA) - a physical implementation of the open source Nvidia NVDLA architecture - and two Programmable Vision Accelerators (PVAs) for processing computer vision, each driven by an ARM Cortex-R5 core with two dedicated Vector Processing Units (VPUs).

According to NVIDIA, the Xavier board has been designed with security and reliability in mind, supporting various standards such as ISO-26262 functional safety and ASIL Level C. To this end, the board also includes many resilience features, such as ECC-protected main memory as well as L1, L2 and L3 cache. Another relevant feature of the device in relation to critical systems is its monitoring capabilities, as the device has a series of core and uncore counters [73], as well as a specialized power monitoring system to be able to measure in depth any application running on the system [75].

#### 2.1.2.2 Xilinx Ultrascale+ ZCU102

The Xilinx Ultrascale+ is a general purpose heterogeneous, multi-processing platform targeted to create embedded applications. It comes with four ARM Cortex-A53 CPUs, two ARM Cortex-R5F real-time processors, and a Mali 400 MP2 GPU. The ZCU102, specifically, comes with high speed DDR4 SODIMM, FMC expansion ports, multi-gigabit per second serial transceivers, a variety of peripheral interfaces, and FPGA logic for user customized designs [95].

Regarding the real time capabilities of the board, the RT unit is set up to run in split mode:

- RPU-0: Is configured to run a Real-time operating system (RTOS).
- RPU-1: Is configured to run bare-metal.

This makes the device suitable for real time applications as it complies with safety and security standards.

Figure 2.5 displays a high-level block diagram of the device architecture and key building blocks inside the processing system and the programmable logic [96].

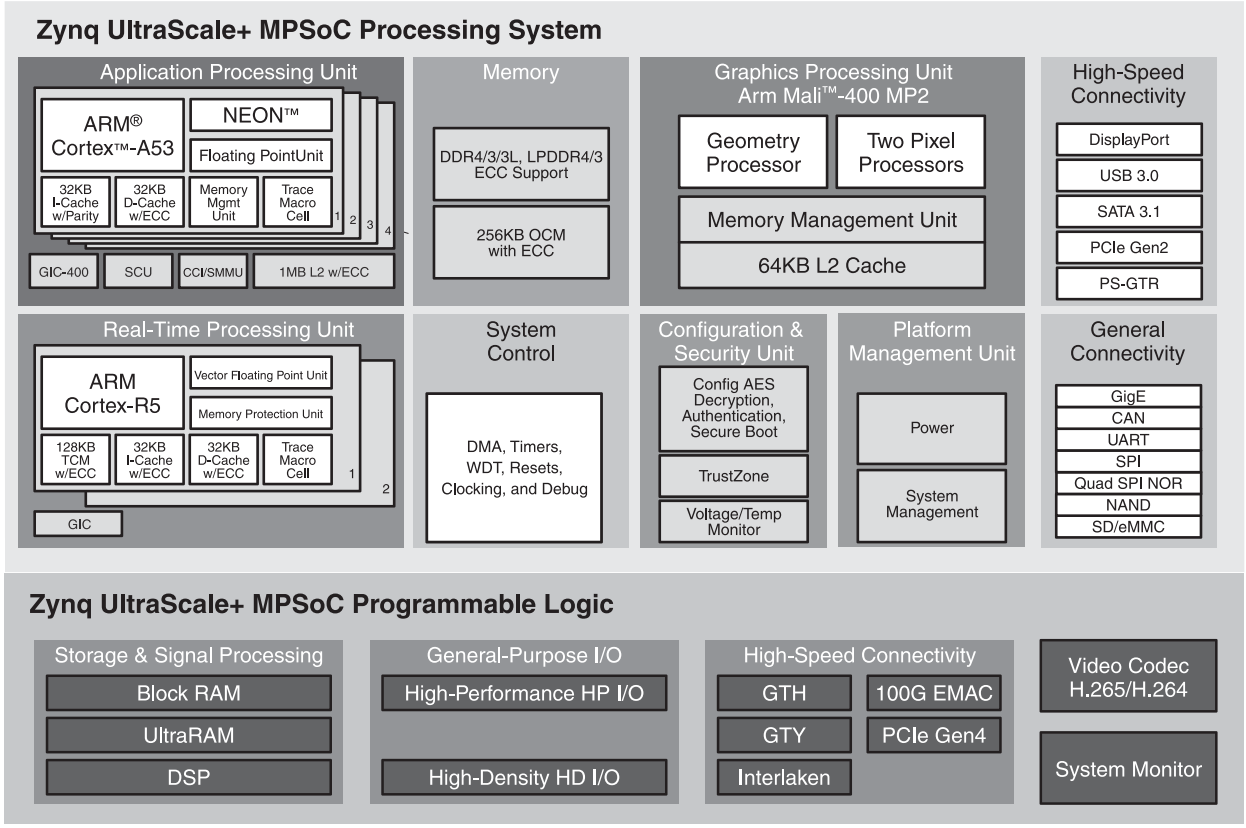



Figure 2.5: Zynq UltraScale+ MPSoC Block Diagram. Original figure credit: Xilinx.

Zynq has a dedicated documentation and framework to handle power in the architecture [94]. The SoC device is divided into four major power domains:

- Full power domain (FPD): Comprises the four ARM Cortex A-53 CPUs, as well as a number of peripherals used by them.
- Low power domain (LPD): Contains the R5 real-time processors, the platform management unit, the configuration security unit and the remaining on-chip peripherals.
- Programmable logic (PL) power domain: Contains the programmable logic.
- Battery-power domain: Contains the real-time clock as well as the battery-backed RAM.

These power domains can be altered through the power management Application Programming

Interface (API) by performing categorical operations listed in the following types: a) suspending and waking up processing units, b) slave device power management, such as memories and peripherals, c) direct-access and d) miscellaneous.

#### 2.1.2.3 GigaIPC AMD Ryzen V1605B

The *Ryzen Embedded V1000* processor family is one of the latest collection of embedded boards introduced by AMD. This collection features the Zen CPU technology and the Vega GPU technology in a Systems-on-Chip (SoC) solution oriented to High Performance Computing (HPC) and multimedia processing in embedded devices [10].

Specifically, the V1605B contains four CPU cores, each with 2 hardware threads and a Radeon Vega GPU counting with eight execution units. As for its configurability, the board does not provide any mechanism to switch between predefined power modes, however the manufacturer specifies a Thermal Design Power (TDP) between 12 and 25 Watts, obtaining a frequency up to 3.6 GHz.

In an effort to make its reconfigurability better, [80] has developed a custom solution to implement a number of power modes similar to the ones present in the NVIDIA Xavier platform. The details of some of the power modes present in the configuration script can be found in table 2.2:

| Power Mode | Number of CPUs | Threads per CPU | Memory Frequency (MHz) | GPU Frequency (MHz) |
|---|---|---|---|---|
| 12W | *2* | *1* | *400* | *200* |
| 15W | *4* | *1* | *1067* | *1100* |

Table 2.2: Custom configuration for the power modes of the V1605B (GigaIPC)

This embedded device supports up to 32 GiB of dual-channel DDR4-2400 memory and incorporates Radeon Vega 8 Graphics operating at up to 1.1 GHz.

However, official information regarding this embedded board is quite scarce, as the device, at the time of writing, does not have a technical reference manual. Therefore, some of the information that can be found is considered preliminary and may change by final release [92].

## 2.2 CPU Parallel Processing Methodologies

The most basic CPU workload distribution consists on converting a sequential job that runs in a single core to a parallel job that runs in multiple cores. We call this distribution strategy **multi-core parallelism**. In this type of parallelism, each core is in charge of processing part of the full computation present in a section of a problem. Thus, if a program is run with $n$ cores, then ideally it should be $n$ times faster than the sequential implementation of the same program:

$$T_{parallel} = \frac{T_{sequential}}{n} \tag{2.1}$$

Even if this speed-up is impossible to obtain, as Gene Amdahl and John L. Gustafson expose in their works [11] [47], multi-core parallelism has been employed historically to optimise problems.

### 2.2.1 General Purpose Multi-core Programming models

In this Section we are going to explore different multi-core programming environments. Section 2.2.1.1 showcases POSIX threads (pthreads), an execution model that provides low-level thread management programmability that enables creating and managing parallel tasks explicitly. However, this level of flexibility complicates the creation of parallel code, as the programmer must identify and handle complex specific details, such as critical sections or locks, which can lead to poor performance.

High level multi-core programming models have born to overcome these issues, offering a simpler way to approach parallelism. One of the most popular high level multi-core APIs is OpenMP, which supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, in many platforms. Section 2.2.1.2 provides some details about OpenMP and its specification.

Another relevant high level multi-core programming model is StarSs, developed at BSC, in which the user must identify individual pieces of code that can be expressed as tasks [77]. OmpSs represents the integration of StarSs into a single characteristic programming model, encompassing asynchronous parallelism and heterogeneity support in devices such as GPUs and FPGAs [31]. Section 2.2.1.3 exposes some of the features of OmpSs.

#### 2.2.1.1 pthreads

pthreads is a set of C language programming types and procedure calls that implements a standardized interface for the use of threads following the IEEE POSIX 1003.1c standard [1].

Before POSIX standard was created, different vendors had their own proprietary implementations of threads, which diverged significantly from each other, complicating the development of portable threaded applications. The POSIX standard emerged out of the need to unify and establish a common implementation of threads.

pthreads comprises a series of subroutines that can be classified in four categories [91]:

- **Thread management:** These routines enable the programmer to manage the thread attributes and context (creating, attaching, scheduling, etc.).
- **Mutexes:** These routines manage synchronisation (creating, destroying, locking and unlocking mutexes).

- **Condition variables:** These routines enable the programmer to define certain level of communication between threads that shared a mutex by specifying a series of conditions.
- **Read/Write synchronisation:** These routines manage read/write locks and barriers.

As we have seen in the previous list, pthreads is made of a series of low level functions that allows the user to manage threads explicitly. This implies that the programmer must control all write and read dependencies, as well as the synchronization between threads. This level of flexibility entails a certain degree of complexity such that applications that contain fine grained thread management become less readable. Following next, we present a minimal pthreads *VectorSum* example:

```
#include <pthread.h>

int A[1000], B[1000], C[1000];
int threads_id[4] = {0,1,2,3};

void* vectorSum (void* arg) {
    int *local_tid = (int*)arg;
    for(int i = (local_tid * 250); i < (local_tid + 1)*250; ++i) {
        C[i] = A[i] + B[i];
    }
    pthread_exit(0);
}

int main() {
    pthread_t tid[4];
    ...
    for(int i = 0; i < 4; ++i) {
        pthread_create(&tid[i], NULL, vectorSum, &threads_id[i]);
    }
    ...
    for(int i = 0; i < 4; ++i) {
        pthread_join(tid[i], NULL);
    }
    ...
    pthread_exit(NULL);
}
```

Listing 2.1: VectorSum example in pthreads

In the example above, we present the *VectorSum* problem, where we add two vectors with the same number of elements. For the sake of simplicity, we create 4 threads where each thread computes $\frac{1}{4}th$ of the array elements.

The global variable `threads_id` defines an identifier for each running thread. Each id is passed to the corresponding thread scope in the `pthread_create` call. Then, the function `VectorSum` performs the operation over 250 consecutive elements. Finally, `pthread_join` blocks the calling threads until the target thread terminates.

As we can see from the example above, it is required to engineer a customised solution based on the intended parallel strategy, therefore, advanced knowledge about multi-threading intrinsics is required to implement parallel efficient code.

In Listing 2.1, we studied a subset of the most relevant pthreads primitives, however due to the simplicity of the problem we have not employed more advanced functions to deal with mutexes and/or locking. The following example illustrates a synchronisation problem using pthreads:

```
#include <pthread.h>

int counter = 0;

void* kernel(void* arg) {
    counter += 1;
    printf("Task %d starts.\n", counter);
    for (unsigned long i = 0; i < (0xFFFFFFFF); i++);
    printf("Task %d ends.\n", counter);

    pthread_exit(0);
}

int main() {
    pthread_t tid[2];

    for(int i = 0; i < 2; ++i) {
        pthread_create(&tid[i], NULL, &kernel, NULL);
    }

    for(int i = 0; i < 2; ++i) {
        pthread_join(tid[i], NULL);
    }

    pthread_exit(NULL);
}
```

Listing 2.2: Synchronisation issue in pthreads

Running the code from Listing 2.2 results in the following output:

```
Task 1 starts.
Task 2 starts.
Task 2 ends.
Task 2 ends.
```

This is because `counter` is a global variable that a second thread modifies while the first thread processes the loop, as both threads call the function almost simultaneously. In order to sort this issue we need to make the second thread wait until the first thread is done with the computation.

`pthread_mutex_lock` is a function that can be used to lock a mutex object previously initialised in `pthread_mutex_init`:

```
#include <pthread.h>

int counter = 0;
pthread_mutex_t lock;

void* kernel(void* arg) {
    pthread_mutex_lock(&lock);

    counter += 1;
    printf("Task %d starts.\n", counter);
    for (unsigned long i = 0; i < (0xFFFFFFFF); i++);
    printf("Task %d ends.\n", counter);

    pthread_mutex_unlock(&lock);
    pthread_exit(0);
}

int main() {
    pthread_t tid[2];
    pthread_mutex_init(&lock, NULL)

    for(int i = 0; i < 2; ++i) {
        pthread_create(&tid[i], NULL, &kernel, NULL);
    }

    for(int i = 0; i < 2; ++i) {
        pthread_join(tid[i], NULL);
    }

    pthread_mutex_destroy(&lock);
    pthread_exit(NULL);
}
```

Listing 2.3: Mutex lock in pthreads

This time thread synchronization took place by the use of a lock:

```
Task 1 starts.
Task 1 ends.
Task 2 starts.
Task 2 ends.
```

The two examples exposed above, demonstrate at a very simplistic level the complexity implied to create parallel code in pthreads. This exposes the need of more simpler parallel methodologies that we'll study in the following Subsections.

#### 2.2.1.2 OpenMP

OpenMP follows the **fork-join model** in which a main thread forks a set number of sub-threads, subsequently dividing a task among them. The API also supports the **task-parallel model** since OpenMP 3.0, whereas the programmer requires to specify deferrable units of work called tasks, which by definition, are not bound to any specific thread [15].

OpenMP specification defines the different compiler parameters needed to compile an OpenMP program per platform and language[3]. The part of the program that is meant to run in parallel should be marked with certain OpenMP directives and clauses based on the intention of the programmer, accompanied by a very basic syntax we can find in the following code section:

```
#pragma omp <directive> <clauses>
```

Following next, we present a list some of the most relevant **directives** (also called constructs) supported in the environment[4]:

- `parallel`: This directive is employed to instantiate additional threads to perform the work scoped by the pragma in parallel.
- `for`: This directive divides the loop body among the available threads. The clauses of the directive specify how the division is made.
- `critical`: The scope under this directive ensures that the associated structured block will be executed by a single thread at a time.
- `atomic`: The atomic construct should be used when a specific storage location requires to be accessed atomically, meaning that only one thread can update it simultaneously.
- `barrier`: Forces the threads to wait until the remaining ones finish the previous work.
- `task`: The code within the task scope will be executed by a single thread. Tasks are run on demand based on their priority and dependencies.

As we pointed previously, OpenMP supports a number of clauses that allow the programmer to mutate the default behaviour of the constructs. Next, we present a list with some of the most popular **clauses** we can find in OpenMP:

[3] https://www.openmp.org/resources/openmp-compilers-tools/
[4] https://www.openmp.org/spec-html/5.0/openmp.html

- `num_threads(num)`: Specifies the number of threads to use within the scoped parallel region.
- `shared(`$var_0, ..., var_n$`)`: By using this clause, the data specified in the parallel region can be viewed and accessed by every thread.
- `private(`$var_0, ..., var_n$`)`: The variables under this clause will be copied per thread, meaning that each thread will have a local copy of the same variable. Private variables aren't initialised and their value isn't maintained outside the parallel region.
- `firstprivate(`$var_0, ..., var_n$`)`: Like `private` except the variable is initialised to its original value.
- `reduction(type:`$var_0, ..., var_n$`)`: Each thread privatises the listed variables in the clause to then update the original variable associated with each private copy by performing the operation (type) dictated in the reduction clause, avoiding any kind of data races.
- `schedule(type, size)`: Determines how the iterations of a loop are scheduled:
  - `static`: Contiguous iterations are distributed equally among all threads. The size variable determines how many contiguous iterations will be assigned to the same thread.
  - `dynamic`: Works like the static scheduling with the difference that if a thread completes the assigned iterations, it will continue to compute non-assigned iterations following a *first-come, first-served* order.
  - `guided`: Each thread is assigned to a contiguous large chunk of iterations. The size of the chunk will exponentially decrease with each allocation down to a set size.
- `depends((in|out|inout):`$var_0, ..., var_n$`)`: This is a `task` dependency clause that according the specified type, the tasks will be deferred accordingly to avoid Read After Write (RAW), Write After Read (WAR) and Write After Write (WAW) dependencies.

OpenMP also provides a series of functions to gather run-time information and configure the parallel environment. Some of the most relevant functions are listed below:

- `omp_get_max_threads()`: Returns an upper bound on the number of threads available to create new parallel regions without taking in consideration `num_threads`.
- `omp_get_num_threads()`: Returns the number of threads in the current scope.
- `omp_get_thread_num()`: Returns the thread id of the calling thread.
- `omp_set_num_threads(num)`: Sets the number of threads used for downstream parallel regions that do not contain the `num_threads` clause.
- `omp_get_wtime()`: Returns the elapsed time in seconds with double precision.

So far, we have presented a subset of features of OpenMP that demonstrate the programmability of the API to create parallel programs. Nevertheless, an exhastive coverage of the full features of OpenMP are out of the scope of this thesis. Next, we present a parallel *VectorSum* example using OpenMP fork-join model:

```
void vectorSum (const double *A, const double *B, double *C) {
    #pragma omp parallel for
    for (unsigned i = 0; i < SIZE; ++i) {
        C[i] = A[i] + B[i];
    }
}
```

Besides the fork-join model we saw above, OpenMP also supports working with tasks, similar to pthreads, but handling the dependencies, locking and synchronisation automatically. The example below shows a demonstration of an iterative task decomposition with a task granularity of 1 iteration over the *VectorSum* problem:

```
void vectorSum (const double *A, const double *B, double *C) {
    for (unsigned i = 0; i < SIZE; ++i) {
        #pragma omp task firstprivate(i) shared(A, B, C)
        C[i] = A[i] + B[i];
    }
}
```

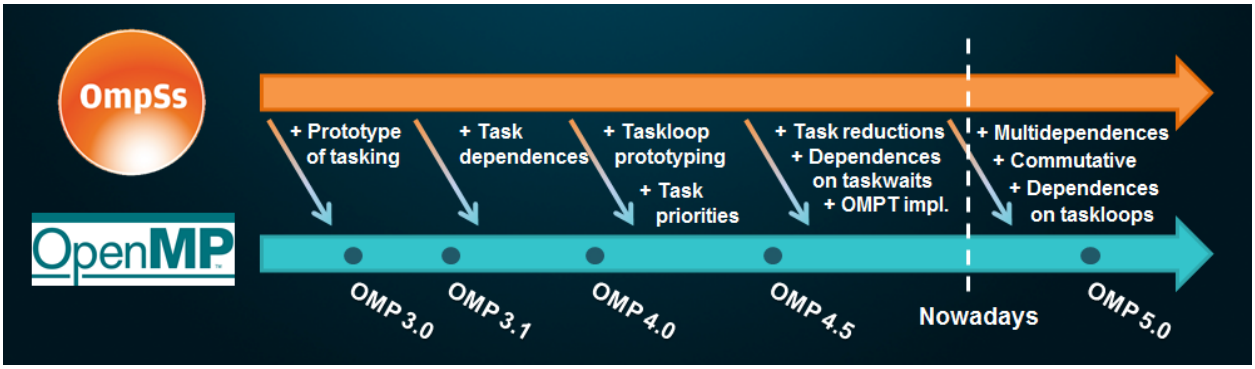
Usually, OpenMP provides parallelism only by annotating a sequential program with *pragmas*, which makes it ideal to work with, especially when working with legacy code, or maintaining code for multiple architectures. Thanks to this, it hardly introduces any noise in the source code due to the parallelism intrinsics and makes OpenMP a perfect candidate to simple and easy-to-read parallel code.

#### 2.2.1.3 OmpSs

OmpSs enables asynchronous parallelism by managing the data-dependencies between the different identified tasks in the running environment. One of the major advantages of OmpSs is that it supports heterogeneity by introducing the `target` construct, which allows the programmer to specify on which devices the scoped tasks should be executed. If no device clause is specified within this construct, OmpSs defaults to the SMP. Section 2.3.1.5 provides more information on the heterogeneity of OmpSs. One of the key differences with OpenMP, is that in OmpSs, the `task` construct can also be used in `void` typed function declarations, which makes every invocation of that function a task instantiation point.

As we mentioned in the introduction, OmpSs is an effort to implement the ideas from the StarSs programming model, and many of these ideas have been introduced also into OpenMP. In fact, OmpSs acts as a forerunner of several OpenMP features, showing their potential, and then they are brought in OpenMP committee for discussion, in which BSC is represented. The following Figure, extracted from the OmpSs specification[5], summarises the contributions to OpenMP:

[5] `https://pm.bsc.es/ftp/ompss/doc/spec/`




As shown above, OmpSs contributed to OpenMP with a prototype of asynchronous tasking, which was made available in OpenMP 3.0 (2008). Since then, OmpSs has been contributing to the OpenMP programming model in the context.

OmpSs parallelism is centered around asynchronous tasking, for that, the environment allows expressing data-dependences among them using different clauses in the `task` construct:

- `in`: The data specified must be available in the address space where the task is executed, the data will be read only.
- `out`: The data specified will be generated by the task in the address space where the task is executed.
- `inout`: The data specified must be available in the address space where the task runs, also, the data will be updated in the scoped code segment.

These dependence clauses shape OmpSs programming model, as we will use them in most of the constructs, the list of constructs relevant for multi-core programming is presented below:

- Task construct: The programmer can specify a task by adding this syntax to any structured block: `#pragma omp task [clauses]`. These are some of the supported clauses found in this construct:
  - `private, firstprivate, shared, reduction`: Perform the same operation we have seen in Section 2.2.1.2.
  - `depend(<type>:` $var_0, ..., var_n$`), <depend-type>(`$var_0, ..., var_n$`)`: This clause allows to infer task scheduling restrictions from the parameters it defines. The syntax of this clause includes the dependence type followed by its associated variables.
  - `tied`: The task gets bound to the thread that starts its execution. In case this task has paused its execution, only the same thread can resume it.
  - `if`: If the expression contained in the `if` clause is `false`, then the running task must wait until the newly created task completes its execution.

- Loop construct: When this type of construct is found, a task is created for each of the blocks in which the iteration space is divided. Unlike OpenMP, OmpSs only supports scheduling clauses in the loop construct. The syntax of the construct and the clauses corresponds with the syntax provided in OpenMP described in Section 2.2.1.2.
- Taskwait construct: Specifies a wait on the completion of all direct descendant tasks. It's defined by: `#pragma omp taskwait`. The valid clauses are the following:
  - `on`$(var_0, ..., var_n)$: Specifies to wait only for the subset of direct descendant tasks.
- Taskyield directive: Specifies that the ongoing task can be stopped and that the scheduler is allowed to schedule another task. The syntax is the following: `#pragma omp taskyield`.
- Atomic construct: Ensures that the enclosed expression gets executed atomically. Defined by: `#pragma omp atomic`.
- Critical construct: Ensures that only one thread at a time is executing the scoped region, other threads will wait at the start of the critical section (mutual exclusion). The syntax of the critical construct is the following: `#pragma omp critical`, followed by a block.

Following next, we display a matrix multiplication pseudo-code in OmpSs to demonstrate some of the language features[6]:

```
#pragma omp task in([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void mm(double  *A, double *B, double *C, long NB)
{
    for (int i = 0; i < NB; ++i)
        for (int j = 0; j < NB; ++j)
        {
            double tmp=C[(i*NB)+j];
            for (int k = 0; k < NB; ++k)
            {
                tmp+=A[(i*NB)+k]*B[k*NB+j];
            }
            C[(i*NB)+j]=tmp;
        }
}

void compute(int *A, int *B, int *C, unsigned long NB)
{
    for (unsigned i = 0; i < DIM; i++)
        for (unsigned j = 0; j < DIM; j++)
            for (unsigned k = 0; k < DIM; k++)
                mm(A[i][k], B[k][j], C[i][j], NB);

    #pragma omp taskwait
}
```

[6] https://github.com/bsc-pm/ompss-ee/blob/master/02-beginners/matmul/.config/matmul.c

As we have saw in the example above, OmpSs makes task programming intuitive thanks to the different dependency directives defined by the specification of the parallel model. Besides all the exposed features, OmpSs also supports heterogeneity that we will study in Section 2.3.1.5.

### 2.2.2 Parallel Processing in Critical Systems

As we saw in Section 2.1, critical markets are moving towards multi-core systems, mainly due to the current availability, price and performance of single-core systems. However, this need for adaptation is not free for the critical domain, as the software resources employed must meet a number of requirements to be certified. For this reason, multiple operating systems and programming models adapted to the critical domain began to emerge.

In this Section we are going cover some state of the art RTOS (Section 2.2.2.1), as well as ADA/SPARK, a programming language intended for the development of high integrity software (Section 2.2.2.2); and finally in Section 2.2.2.3 we will revise some of the most relevant multi-core programming models for critical systems to date.

#### 2.2.2.1 RTOS based Multicore Processing

A RTOS is a type of operative system scoped to real-time applications. These type of systems are characterised by their timing consistency and low variability when executing real-time applications.

In order to meet the specifications provided by real-time systems, RTOSs are required to accomplish the following criteria:

- Minimal interrupt latency: RTOSs should decrease the number of cycles required for a processor to respond an interrupt request.
- Minimal context switching latency: In a RTOS, the processor should take a reduced time to store the state of a process or thread so that it can be restored and resume the execution later.

This criteria was defined to favour predictability and low-latency, rather than amount of work done in a given period of time (throughput); as RTOSs are more frequently dedicated to a narrow set of applications rather than a wider specter of processes running simultaneously. These constraints tend to redefine the scheduling algorithms employed in these type of systems.

Thus, the design philosophy of a RTOS values the consistency, predictability and performance of the real time applications running in the OS.

In this Section we are to comment on the two most well known RTOSs, Zephyr and FreeRTOS.

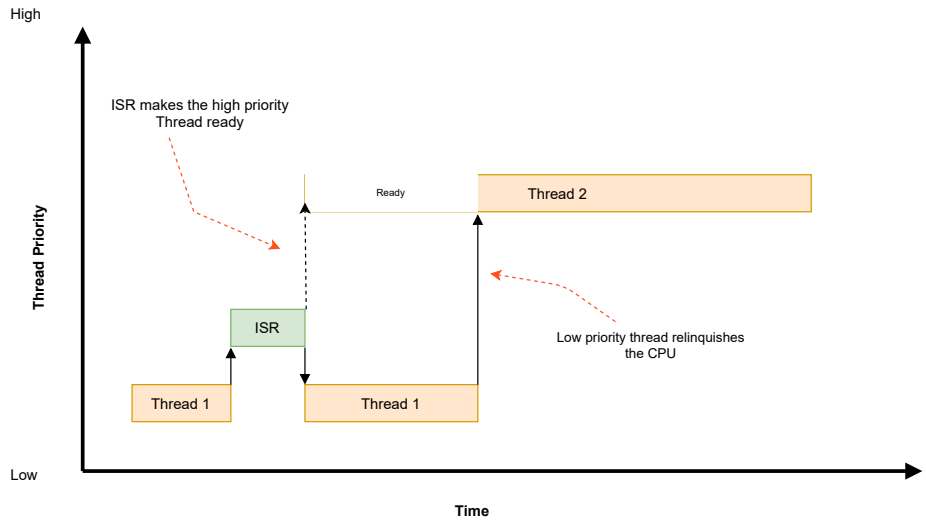
**Zephyr** is a scalable open-source RTOS optimised for resource-constrained devices and built targeting a secure environment [32]. Zephyr offers a series of features that distinguish it between other RTOSs, such as its multi-threading capabilities:

- Multi-threading Services: Zephyr kernel implements cooperative, priority-based, preemptive and non-preemptive threads. These multi-threading capabilities can be employed through the pthreads API.
- Interrupt Services: Enables compile-time registration of interrupt handlers.
- Memory Allocation Services: Zephyr supports dynamic allocation and freeing fixed or variable size memory blocks.
- Inter-thread Synchronisation Services: Support binary, counting and mutex semaphores.
- Inter-thread Data Passing Services: For byte streams, basic and enhanced message queues.
- Power Management Services: Supports tickless idle between others.

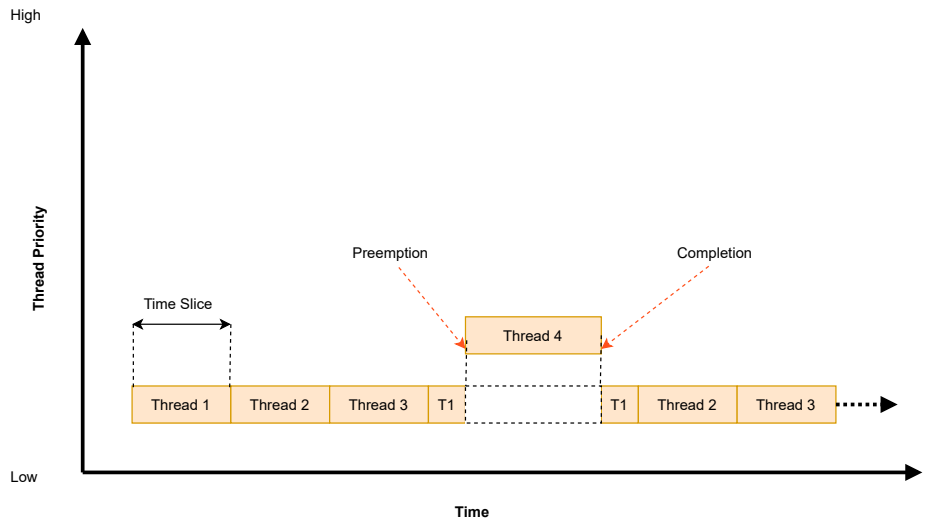
Zephyr also implements various scheduling features that allow an application's threads to share the CPU. In short, the scheduler selects the highest priority thread to be the current thread, and under conflicting situations the scheduler chooses the thread that has waited the most. Zephyr's code-base enables the end user to choose between different queuing strategies based on the needs of the target applications:

- Simple linked-list ready queue: The ready queue consists of an unordered list, with fast constant-time performance for single-threaded operations and very low code size.
- Red/black tree ready queue: The ready queue consists of a red/black tree, which offers slower insertion and removal operations but an increased scalability towards many thousands of threads. This scheduling policy is probably the worst candidate for the RTOS.
- Traditional multi-queue ready queue: In this queuing strategy the ready queue is implemented as an array of lists, one per priority. It slightly improves the code size over the simple linked-list policy and performs the computation in $\mathcal{O}(1)$ time in almost most of the circumstances. On the other hand, it has a notable memory footprint to store the array of lists. This algorithm is incompatible with several other features like deadline scheduling or SMP affinity.

Besides the offered queuing strategies, Zephyr's scheduler also implements time slicing policies to prevent thread starvation in its cooperative and preemptive scheduling modes (Figure 2.6).




(a) Cooperative time slicing: The cooperative thread halts its execution from time to time to permit other threads to execute.




(b) Preemptive time slicing: The scheduler divides time into a series of time slices, at the end of every slice, preemptive thread get yielded to execute other ready threads of the same priority.

Figure 2.6: Time Slicing policies (Images credit: The Zephyr Project).

Zephyr also implements Earliest deadline first (EDF), a scheduling algorithm employed in RTOSs, in which each time a scheduling event takes place, a priority queue is searched to find the process closest to its deadline in order to execute it next.

As we saw, Zephyr offers great flexibility in terms of scheduling choices favouring real time multi-threading. However, it also provides great features in terms of security, making this operating system suitable for use in critical systems. Some highlighted security features are the following: stack-overflow protection, device driver permission tracking, thread isolation, etc.

**FreeRTOS** is an open-sourced RTOS designed under specific size constraints to be able to run in micro-controllers, although its use is not limited to the intended scope. In short, FreeRTOS provides the following features:

- Multi-tasking: FreeRTOS structures each real time application as a set of independent tasks. The real time scheduler should decide which task from a running application should execute in a given point of time. The scheduler will swap tasks in and out as the application runs (context switch).

- Timing primitives and callbacks: FreeRTOS provides software timers that allow to execute a function later in time through a callback. The implementation of this feature in FreeRTOS doesn't consume processing time unless a timer expires, which is when the callback gets executed.

- Memory allocation and heap memory management: FreeRTOS provides a very simple API to create RTOS objects (tasks, queues, timers, semaphores...) using dynamically or statically allocated RAM if defined explicitly. `malloc` and `free` are not always ideal to allocate dynamic memory for a series of reasons: not thread safe, not time deterministic, often aren't available on embedded systems, etc. For that reason, FreeRTOS download includes five custom memory allocation implementations scoped for different purposes:
  - `heap_1`: The simplest, it doesn't permit freeing memory.
  - `heap_2`: Permits freeing memory but doesn't coalescence adjacent free blocks.
  - `heap_3`: Wraps the standard `malloc` and `free` for thread safety.
  - `heap_4`: Like heap_2, but coalescences adjacent free block to avoid fragmentation.
  - `heap_5`: As per heap_5 but spans the heap across multiple non-adjacent memory areas.

- Synchronisation primitives: FreeRTOS supports binary, counting, mutex and recursive mutex semaphores.

- Inter-task communication: Stream and message buffers are designed to serve a task to task and a interrupt to task communication. Unlike other RTOSs, FreeRTOS communication primitives are optimised for single reader-writer scenarios. Stream buffers pass a continuous stream of bytes, while message buffers work with variable sized discrete messages, employing the stream buffers for data transfer. An intrinsic feature among FreeRTOS objects is that the stream buffer assumes that there is only one task or interrupt that will write to the buffer, and only one task or interrupt that will read from the buffer. Thus, its a single-producer, single-consumer relationship.

  Another form of communication found in FreeRTOS are Task Notifications. A *direct to task notification* is an event sent explicitly to a task, rather than using an intermediary object. These types of messages are scoped to change the state of the target task explicitly, i.e.: blocking certain task until an arbitrary computation is completed, the task will only resume its activity after it receives another notification changing its state.

- Decoupled libraries: FreeRTOS also provides a collection of MIT licensed libraries available for use with the kernel. These libraries can complement the kernel with specific features any scoped application might need. We can find three types of libraries:
  - FreeRTOS+: Provides connectivity and utility functionality suitable to connect Internet Of Things (IoT) devices to the cloud.
  - AWS IoT Libraries: Like FreeRTOS+ but oriented to Amazon Web Services.
  - FreeRTOS Labs: Set of libraries that intend to improve the FreeRTOS environment including features such as pthreads support, TCP IPv6 or FAT file-system support.
- Power saving: FreeRTOS supports a tickless idle mode that stops the periodic tick interrupt when no application tasks are executing.

Regarding the scheduler, FreeRTOS supports two types of policies that can be selected simultaneously:

- Time Slicing Scheduling Policy (Round Robin): In this algorithm, all equal priority tasks get to run in equal portions of CPU time.
- Fixed Priority Preemptive Scheduling: In this algorithm, the scheduler selects tasks according to their priority. Low priority task will get executed only when there are no high priority tasks in the ready state.

FreeRTOS provides also Memory Protection Unit (MPU) support on certain ARMv7 and ARMv8 cores, to enable robustness and security by creating the concept of privileged and unprivileged mode, while also restricting access to resources such as RAM, executable code, peripherals and memory beyond the limit of the task's stack[7].

For a complete list of the features of this operating system, the reader can refer to the FreeRTOS reference manual [65].

#### 2.2.2.2 Ada/SPARK

Ada is a flexible concurrent and distributed object-oriented language focused towards real-time and embedded systems. It has been relevant in many different high-integrity areas (automotive, railway and avionic domains) thanks to a number of factors that make it qualify for its use in real-time systems [19]:

- Strong compile-time type checking.
- Native support for parallel programming.

[7] `https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html`

- Real-time scheduling compliance.
- Safe object-oriented programming approach.

In this Section we are going to focus on the real-time parallelism capabilities of Ada, for that we are going to display a very simplistic example of independent tasking:

```
type Vector is array (1..10_000) of Integer;
type Vector_Prt is access al lVector;
type Vector_Const is access constant Vector;

procedure Sumdif(A, B : Vector_Const;
                    Sum : Vector_Prt;
                    Diff : Vector_Prt);

procedure Sumdif(A, B : Vector_Const;
                    Sum : Vector_Prt;
                    Diff : Vector_Prt) is
    task typ Minus;
    task typePlus;
    M : Minus;
    P : Plus;

    task body Minus is
    begin
        for I in Vector'Range loop
            Diff(I) := A(I) - B(I);
        end loop;
    end Minus;

    task body Plus is
    begin
        for I in Vector'Range loop
            Sum(I) := A(I) + B(I);
        end loop;
    end Plus;
begin
    null;
end Sumdif;
```

In the Listing above, extracted from [19], `Sumdif` calculates the sum and difference of two large integer arrays. For that, the authors have set two tasks, so that the computation of the addition and the subtraction can happen concurrently. In this very simple example, the tasks terminate naturally and the procedure returns the values after all the tasks finish. With this example, we have learned that Ada follows a tasking parallelism approach, where the programmer is in charge of defining the tasks and managing their dependencies and synchronisation.

However, the example doesn't demonstrate the whole tasking capabilities of ADA. The language supports task communication through data sharing and synchronisation mechanisms [67], such as the following:

- Protected objects: Provides a secure mechanism to execute operations with mutually exclusive access to certain encapsulated data. These objects contain a public and a private part: the public part holds the procedures that interact with the encapsulated data, while the private part comprises the encapsulated variables that must be mutually excluded.
- Protected entries: The protected entries use a read/write lock mechanism to access data encapsulated within its protected object. To achieve object locking, each protected entry contains a Boolean property called `barrier`. A task calling a protected entry with a `False` barrier will be blocked until the barrier becomes `True`. Therefore, barriers get evaluated when an entry is called for the first time and are reevaluated upon the completion of an entry or the body of the procedure by some task. Protected entries can be employed to create many concurrent synchronisation mechanisms, like barriers, semaphores or broadcasts.
- Pragmas Atomic and Volatile: The pragma `atomic` instructs the compiler to perform no optimisations on a variable to ensure that it satisfies atomicity constraints (no temporary copies). The `volatile` pragma ensures that reads and writes to the named variable go directly to memory rather than to a possible copy that the compiler optimises into registers.

All these synchronisation mechanisms and much more can be found in Ada's technical reference manual [14].

As we mentioned in the introduction of this Section, the Ada compiler performs exhaustive type checking to generate executable code, which justifies the verbosity of the language as it leaves almost no room for end-user type errors, such as non-defined type conversions in C. For this reason, the industry has leaned recently towards C and C++ for high criticality domains, as these languages are more comfortable for the programmer. However, the adoption of these languages to many high-integrity areas makes the certification of parallelism a challenge, as modern parallel APIs do not adhere to the majority of certification guidelines required for use in the real-time domain (refer to Section 2.2.2.3).

To ease the certification of real-time systems, **SPARK** - a formally analyzable subset of Ada 2012 - and toolset that brings mathematics-based confidence to software verification can be used. SPARK is officially supported by AdaCore, a major compiler vendor for Ada. SPARK, as MISRA-C [71], MISRA-C++ [70] and other widely used safety-critical coding standards, present a series of rules that enable the certification of Ada programs for their use in critical systems. For example, pointers cannot be used in the context of a SPARK program and there is a restricted use of dynamic allocation.

Besides restricting the use of the language, one of the biggest advantages of SPARK is that it allows to ensure program correctness by providing contextual information like preconditions, post-conditions, loop invariants, assertions, etc. This analytic phase makes SPARK a suitable selection for very critical pieces of code, in which high-criticality and real time should be always ensured. SPARK comes with a set of tools to cover this purpose [66]:

- GNAT Compiler: The GNAT compiler ensures conformance with all the Ada semantic-syntax rules and generates executable code. The compiler can also generate machine code to check any assertions while the program runs.
- GNATprove: GNATprove is the verification tool for SPARK. It can be run in three different modes.
  - Check: Ensures that only the SPARK subset is used within the Ada program.
  - Flow: Performs an analysis that tracks the dependencies of data and subprograms.
  - Proof: Performs a formal verification of the SPARK program, so that code that may result in a run-time error (i.e.: divisions by zero) will be flagged.
- GNATtest: GNATtest is a tool based on AUnit that helps automate processes for developing and managing test cases needed for verifying software systems. Test cases for GNATtest may even be written directly in the Ada code. GNATtest also takes advantage of Ada 2012's contract-based programming features, including preconditions, postconditions and invariants.

However, proving all the software is very difficult and time consuming since the tools provided by SPARK require guidance and sometimes code refactoring is needed to complete the process, which makes development very slow and expensive. For that reason, there are different levels of assurance that can be targeted with SPARK (Stone, Bronze, Silver, Gold, Platinum) [4], that can be applied based on the software needs and project requirements. The cost of achieving higher SPARK level is increasing towards Platinum, however it can provide much more confidence on the correctness of the software.

#### 2.2.2.3 Multicore Programming models for critical systems

As we mentioned in Section 2.1, multi-core platforms are now common across several critical domains. This standardisation has allowed researchers to find solutions to exploit the parallelism present on these platforms, with the objective of providing the performance requirements of advanced critical systems, e.g. autonomous driving. However, most of the standard parallelisation programming models often present features that are against the principles of safety-critical systems: dynamic allocation, poor reliability, lack of resiliency mechanisms, etc. As a result, functional safety properties cannot be guaranteed. In this Section, we are going to explore parallel programming models and solutions that qualify for safety critical environments.

**OpenMP** is one of the most popular programming models to create parallel solutions, although due to the implications commented above, it is an unpractical candidate for the use on critical systems. Sara Royuela et al. propose a series of modifications to the specification, and a set of requirements for the compiler and run-time systems that makes OpenMP qualify for safety critical environments without compromising functional safety [81]. For that, the authors have identified and classified the features that can compromise functional safety in OpenMP:

- Unspecified Behavior: These can be observed in programs that do not follow the OpenMP specification, behaviors that are not defined by the specification (e.g., passing a negative number to `omp_set_num_threads`), or in various situations encountered at runtime or compile time, such as clauses that contain accesses out of the range of an array section.
- Deadlocks: OpenMP locking mechanism (`master`, `critical`, `barrier`, `omp_set_lock` and `omp_unset_lock`) can all produce deadlocks if not employed correctly. For instance, `omp_set_lock` and `omp_unset_lock` work in pairs, therefore a lock without a proper unlock, causes a deadlock situation. Synchronisation directives are also candidates for deadlocks, for example, nesting various `critical` constructs with the same name. The use of untied tasks can produce deadlocks due to the "non-restrictions" nature of the scheduler, these issues, however, would not happen with tied tasks.
- Race Conditions: A race condition can be defined as a conflict in a concurrent execution when multiple threads access the same data simultaneously and at least one of the threads issues a write, causing nondeterminism in the data that gets read, as only a subset of threads will retrieve the correct value (if any). Data races aren't acceptable in safety-critical systems, since they result in unpredictable behavior. The objective of a critical system is detecting these data races to prevent false positives.
- Cancellations: OpenMP cancellation constructs enable the threads to jump out from a region skipping part of its computation. These cancellations happen at cancellation points as they require to be synchronous. This introduces non-determinism due to certain intrinsics with the feature e.g. the behaviour of nested regions suitable of being canceled.
- Non-hazardous features to consider: Resiliency is crucial in safety-critical systems. However, OpenMP doesn't provide enough resilient measures to not compromise functional safety, for example, OpenMP doesn't specify how implementations should react if the user requests more threads than the ones present in the platform. Nesting is another complex issue for real-time systems, as the environment should create multiple parallel regions that can cause data locality issues and cause the oversubscribing of system resources in low memory budget situations.

To tackle these problems, the authors have defined a series of rules that define how OpenMP directives should be employed in order to comply with a functional safety critical paradigm. For instance, the directive `usage` is proposed in order to prevent illegal nesting; in this directive the user should specify the clauses and constructs employed in a parallel region, the compiler would then restrict certain combinations of clauses that fall under the `usage` context.

There are other types of rules that prohibit certain expressions to be used in specific contexts, for example: the clauses `cancel` and `cancellation point` should be only used in non-nested regions.

Data races are also in the list of relevant issues that should be sorted, for that, a new directive called `globals` is proposed. This directive defines which data is used within a given scope while it can be accessed concurrently from outside the scope producing a data-race. The directive would be accompanied by different clauses based on the nature of the data: `protected_read, read, protected_write, write`. `read` should be used when the global data is only read, and `write` when the data is also written, the `protected` versions simply imply atomicity. By doing that, you ensure that the run-time environment has information about how to protect the data.

```
#pragma omp usage any reduction(factorial)
#pragma omp globals write(factorial)
void fact(int N, int &factorial);

void fact(int N, int &factorial)
{
    factorial = 1;
    #pragma omp for reduction(*:factorial)
    for(int i = 2; i <= N; ++i)
    {
        factorial *= i;
    }
}
```

Listing 2.4: Function declaration and body using the extensions for safety-critical OpenMP.

An example of some of proposed the features mentioned above can be observed in Listing 2.4. In order to explore the full list of proposals by the authors please refer to the cited paper.

Another relevant area of interest towards a safety-critical OpenMP is the scheduler. Most safety-critical systems are based on static allocation strategies, while OpenMP implementations are based on dynamic schedulers. However, sub-optimal compliant static allocation approaches have been proposed to palliate this issue [68]. By implementing all of these, we can finally say that OpenMP can qualify for the use in safety critical systems.

**The EMB$^2$** is a multi-platform C/C++ library for the development of real-time parallel applications [82]. One of the key features of the library, is that, like OpenMP, relieves its users from the burden of synchronisation and thread management. This makes the development of reliable multi-core applications simpler.

The library provides basic parallel algorithms, concurrent data structures and skeletons for implementing stream processing applications (see Figure 2.7).

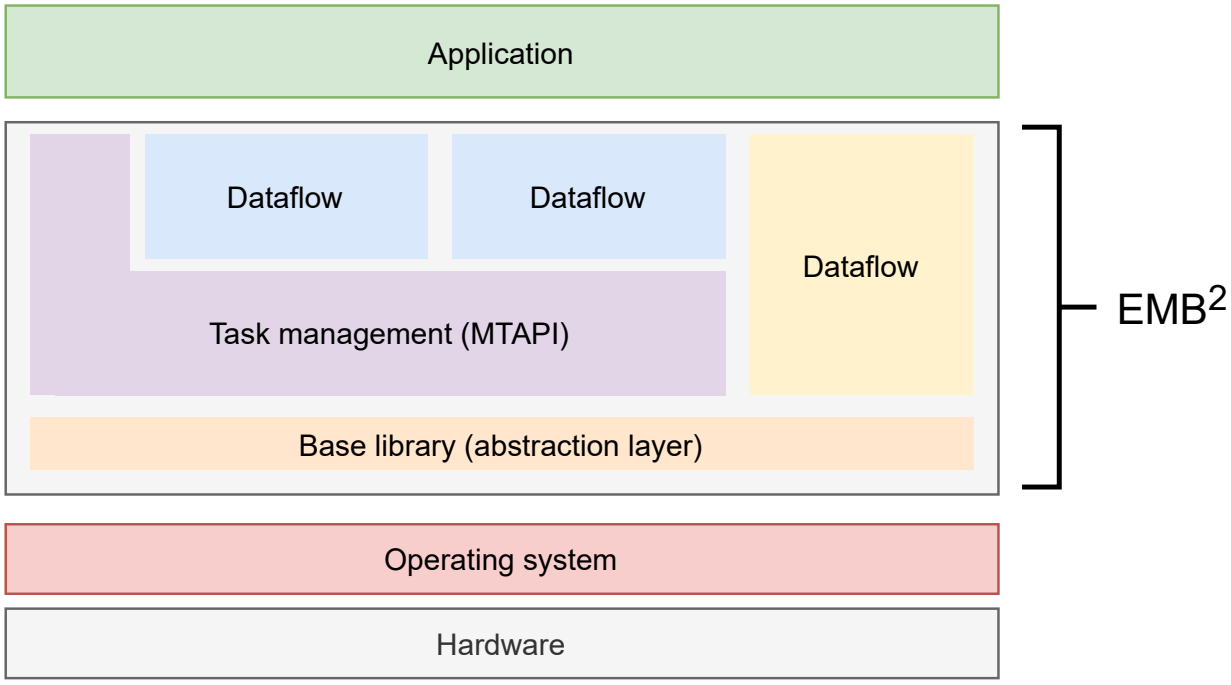



Figure 2.7: EMB$^2$ architecture. Original image credit: Siemens and Multicore Association.

EMB$^2$ employs Multicore Task Management API (MTAPI), a standardised programming interface for leveraging task parallelism in any kind of multi-core system [43]. The API provides a soft real-time compliant low-overhead scheduler to distribute fine-grained tasks among the available cores during run-time, accounting for task priorities and affinities, while offering scheduling strategies based on minimal latency or fairness.

MTAPI supports two types of programming models:

- Tasks: In this programming model, the developer decides where to deploy a task. The run-time system chooses the executing core based on certain predefined API attributes. MTAPI supports the following types of tasks:
  - Single tasks: When this type of task is started, the corresponding code is only executed once by the run-time environment.
  - Multi-instance tasks: When these tasks are started, the run-time environment executes the corresponding block multiple times in parallel.
  - Multiple-implementation tasks/load balancing: When starting these tasks, the MTAPI run-time system decides the executing resources to run the tasks based on the system load.
- Queues: MTAPI enables the user to employ queues to control the task scheduling policies. Users can define order and non-order preserving queues to determine the tasks' execution flow.

MTAPI can be implemented on bare metal, on top of an operating system, and also on top of a hypervisor. EMB$^2$ also supports intercommunication with CUDA and OpenCL employing MTAPI.The provided multi-platform support makes it ideal for the use on critical systems, although its API can produce very verbose programs on complex scenarios, as we can see below:

```
#define FIBONACCI_JOB 1

void fibonacciActionFunction(...) {
    /* cast arguments to the desired type */
    int n = *(int*)args;

    ...

    /* calculate */
    if (n < 2) {
        *result = n;
    }
    else {
        /* first recursive call spawned as task (x = fib(n - 1);) */
        int a = n - 1;
        int x;
        mtapi_task_hndl_t task = mtapi_task_start(..., fibJob, (void*)&a, sizeof(int),
     (void*)&x, sizeof(int), ...);
        /* second recursive call can be called directly (y = fib(n - 2);) */
        int b = n - 2;
        int y;
        fibonacciActionFunction(&b, sizeof(int), &y, sizeof(int), ...);
```

```
22          /* wait for completion */
23          mtapi_task_wait(task, MTAPI_INFINITE, &status);
24          /* add the two preceeding numbers */
25          *result = x + y;
26      }
27  }
28
29  static int fibonacci(int n) {
30      mtapi_status_t status;
31      mtapi_node_attributes_t node_attr;
32      mtapi_nodeattr_init(&node_attr, &status);
33      /* set node type to SMP */
34      mtapi_nodeattr_set(&node_attr, ..., &status);
35      /* initialize the node */
36      mtapi_info_t info;
37      mtapi_initialize(..., &node_attr, &info, &status);
38      /* create action */
39      mtapi_action_hndl_t fibAction;
40      fibAction = mtapi_action_create(FIBONACCI_JOB, (fibonacciActionFunction), ...);
41      /* get job */
42      mtapi_task_hndl_t task;
43      fibJob = mtapi_job_get(FIBONACCI_JOB, THIS_DOMAIN_ID, &status);
44      /* start task */
45      int result;
46      task = mtapi_task_start(..., fibJob, (void*)&n, sizeof(int), (void*)&result,
        sizeof(int), ...);
47      /* wait for task completion */
48      mtapi_task_wait(task, MTAPI_INFINITE, &status);
49      /* delete action */
50      mtapi_action_delete(fibAction, 100, &status);
51      /* finalize the node */
52      mtapi_finalize(&status);
53      return result;
54  }
```

Listing 2.5: Fibonacci sequence computed using MTAPI primitives.

Listing 2.5 showcases the Fibonacci sequence computed using MTAPI, in the example, we can find an increased number of lines of code compared with its sequential version. However, the `algorithms` library included in EMB$^2$ partially palliates this issue. Following next, we display a reduction example employing MTAPI's `algorithms` library:

```
1  vector<int> range(SIZE);
2  for (int i = 0; i < SIZE; i++) {
3      range[i] = i + 1;
4  }
5  using embb::algorithms::Reduce;
6  int sum = Reduce(range.begin(), range.end(), 0, std::plus<int>());
```

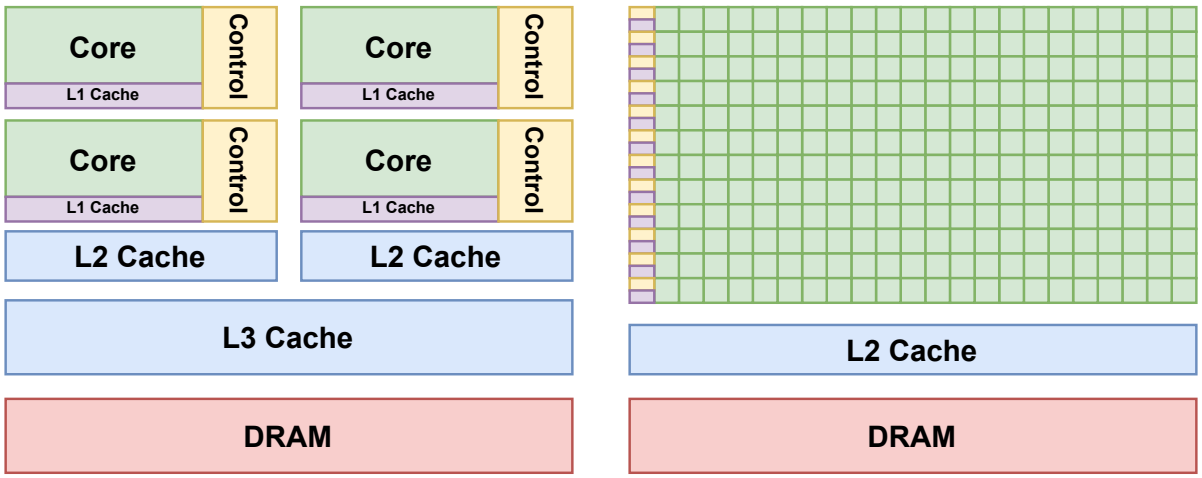
Listing 2.6: Vector reduction using MTAPI `Reduction` primitive.

In Listing 2.6 the `Reduce` primitive from the `algorithms` library of MTAPI is employed to add all the elements present in the `range` vector. This library compiles the most frequent primitives present in parallel applications, which reduces considerably the verbosity of MTAPI programs.

For further details about MTAPI, please refer to its reference card [13].

## 2.3 GPU Compute Processing Methodologies

GPU computing has become increasingly widespread over the last decade. The paradigm has been capable to demonstrate its ability to accelerate certain problems with results several orders of magnitude faster than CPUs at the same cost. This is due to the design nature of GPUs, in short, if we were to compare the purpose of a CPU with that of a GPU, we would say that multicore CPUs are capable of solving a few heavy problems simultaneously, while GPUs are capable of solving many simple problems at once.




(a) CPU architecture overview.

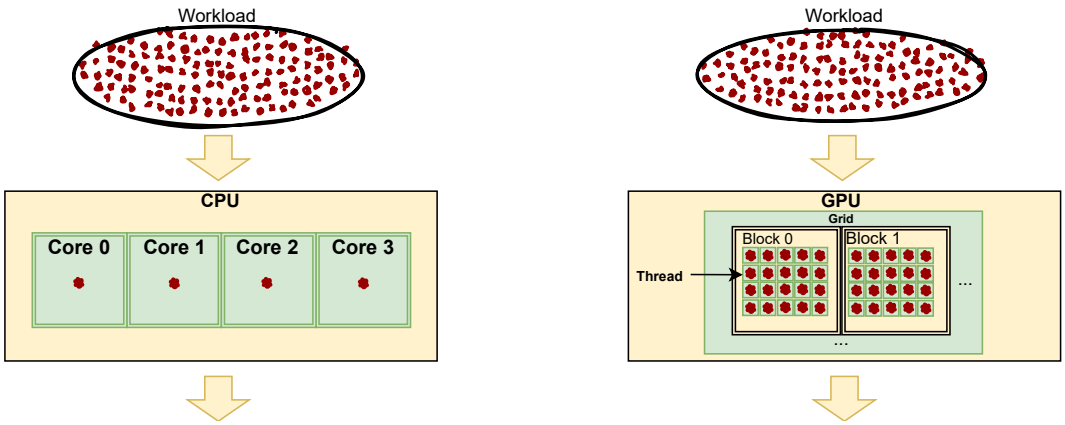
(b) GPU architecture overview.

Figure 2.8: CPU and GPU architecture comparison. Original images credit: NVIDIA.

In Figure 2.8 we can find the architectural differences between a CPU and a GPU [55]. This difference in capabilities exists because they are designed with different goals in mind:

- CPUs follow a **latency-oriented design**, in which the main objective is to execute, given a time frame, as many instructions as possible in a single thread; nevertheless, the number of cycles to process one instruction may vary depending on the case.
- GPUs on the other hand, follow a **throughput-oriented design**, where the goal is to minimise the total throughput of the system, rather than minimising the latencies of all the individual threads they dispose.

For that reason, we can find that GPUs are highly parallel devices, since they dispose a very high number of ”simple” threads capable to perform simpler operations. Figure 2.9 below displays an example of the level of parallelism between the GPU and the CPU.




(a) Multi-core CPU workload distribution.

(b) GPU workload distribution.

Figure 2.9: CPU and GPU workload distribution strategy.

Initially, the design of the GPU was driven solely by the need to handle graphics efficiently. However, with the emergence of the first GPUs, the scientific community became aware of the high level of parallelism that the device could offer. As a result, over the years, GPUs have adapted their design towards a more general purpose computing oriented architecture, and thus programming models oriented to GPU computing started to emerge. Section 2.3.1 revises the introduction of the GPU into the scientific computing community and shows some of the most relevant general purpose programming models on GPUs.

The critical system domain also started to get interested in GPU computing. Not only because of the throughput capabilities, but also by the fact that the use of accelerators enable offloading work from the CPU. However, it is well known that the critical markets (automotive, railway, avionics, etc.) require system safety certification, that is why streamlined certified APIs can significantly reduce certification costs. Section 2.3.2 describes some safety critical graphics APIs that emerged to cover these needs.

### 2.3.1 General purpose programming models

As we mentioned in Section 2.3, the use of GPUs to speedup problems has become popular since the beginning of the 21st century. Thus, a new paradigm was created, General-purpose computing on graphics processing units (GPGPU), which consists on using the GPU, typically employed for computer graphics, to perform computation traditionally handled by the CPU.

GPGPU became more relevant around 2003, specially after the inclusion of floating point color buffers and programmable shaders on graphics processors, which eased the port of CPU algorithms

to the graphics unit [69]. The inclusion of these features were encouraged by the video game industry to enable better effects and graphics in video games. However, the scientific computing community was already aware of some of the benefits that graphics computing could bring with the new hardware. Some of the early jobs in the domain demonstrated the capability of the graphics processor to solve traditional CPU problems, like the matrix multiplication algorithm presented in ACM in 2001 [62]. However, it wasn't until 2005, with the LU factorisation, that a GPU version of a program ran faster than a CPU optimised implementation [41].

These early contributions to the GPGPU paradigm required employing graphics primitives to be able to do compute in the GPU, which required to reformulate traditional CPU problems in these terms. This process was time consuming and complicated, making the development of GPU applications a tedious task. The two major APIs used to perform GPU compute were *OpenGL* and *DirectX*, which were specifically designed for rendering purposes.

The convoluted efforts needed to translate applications for GPGPU and the recent rise of popularity of GPUs in the computing community, resulted in the creation of high level APIs, that allowed developers to abstract from the underlying primitives present in graphics programming over more common high-performance computing concepts [23] [30]. Some of the high level abstractions that emerged in favor of GPU computing are the following: Section 2.3.1.1 explains CUDA's programming model. Section 2.3.1.2 reviews Apple/Khronos OpenCL graphics compute API. And finally, Section 2.3.1.3, 2.3.1.4 and 2.3.1.5 provide, respectively, some specification details about *OpenACC*, *OpenMP-GPU* and *OmpSs-GPU*.

#### 2.3.1.1 CUDA

CUDA is a GPGPU platform and programming model that employs the parallel compute engine in NVIDIA GPUs to enable developers to program readable compute code that runs in the graphics unit[8]. The following table illustrates, to date, the most used languages, APIs and directives-based approaches supported in CUDA:

| **GPU Computing Applications** | | | | | | |
|---|---|---|---|---|---|---|
| Libraries and Middleware | | | | | | |
| cuDNN<br>TensorRT | cuFFT<br>cuBLAS<br>cuRAND<br>cuSPARSE | CULA<br>MAGMA | Thrust<br>NPP | VSIPL<br>SVM<br>OpenCurrent | PhysX<br>OptiX<br>iRay | MATLAB<br>Mathematica |
| Programming Languages | | | | | | |
| C | C++ | Fortran | Java<br>Python<br>Wrappers | DirectCompute | Directives<br>(e.g. OpenACC) | |

[8] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Besides the wide programmability offered by the platform, CUDA is only supported in certain NVIDIA devices, ranging from the embedded to the data center domain[9].

CUDA programming model can be summarised in four main concepts:

- Kernels: A kernel is a user defined function, that, when called, gets executed N times in parallel by the N CUDA threads specified in the kernel call.
- Hierarchy of thread groups: Each CUDA thread is identified within a block with a three dimensional id vector. All the threads of a block are expected to reside on the same processor and share the memory resources of that core. Finally, blocks are contained in a three dimensional grid of thread-blocks. This hierarchy can be seen in Figure 2.9.
- Memory hierarchy: Each thread has private local memory. Each block has shared memory visible to all threads of the block. All threads can access to the same global memory.
- Heterogeneous programming: The CUDA threads execute on the GPU, a physically separated device from the main CPU host. Both the host and the device have their own memory space in DRAM, referred to as *host memory* and *device memory*. CUDA runtime provides an interface to transfer data between both memory spaces. A common CUDA program execution flow can be seen in Figure 2.10.

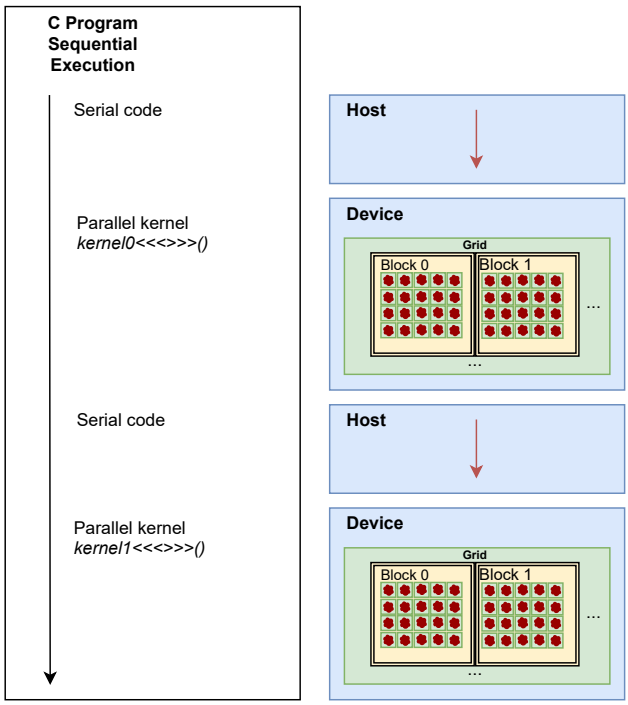



Figure 2.10: CUDA program execution flow. Original image credit: NVIDIA.

[9] `https://developer.nvidia.com/cuda-gpus`

The CUDA runtime provides C and C++ functions that execute on the host to allocate device memory, transfer data and manage the execution flow of the multiple defined kernels. For this thesis we are going to employ a *VectorSum* example to review some of the most relevant features of the environment:

```
// Device code
__global__ void VectorSum(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    // Allocate and initialize host input vectors h_A and h_B
    ...

    // Allocate vectors in device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VectorSum<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Listing 2.7: VectorSum sample in CUDA.

In Listing 2.7 we find some representative functions of the CUDA runtime environment, which we describe below:

- `cudaMalloc(void** devPtr, size_t size)`: Allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory.
- `cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)`: Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy (device to host, host to device, device to device).
- `cudaFree(void* devPtr)`: Frees the memory space pointed to by devPtr, which must have been returned by a previous call to `cudaMalloc()`.

CUDA also introduces a number of C++ extensions such as the function execution space specifiers, which denote the execution and callable space of a given function:

- `__global__`: Specifies the given function as a kernel, executed on the device and callable from the host and device on some target platforms.
- `__device__`: Specifies that a function is executed on the device and callable only from the device.
- `__host__`: Specifies that a function is executed on the host and callable only from the host.

In the previous example, the `__global__` specifier in the *VectorSum* function qualifies it as a kernel, which can be executed from the host using the following expression:

```
Func<<<Dg, Db, Ns, S>>>(params)
```

Where the name of the function, number and type of the parameters should match the ones defined in the kernel definition.

Any call to a `__global__` function must specify the execution configuration for said call, which is what `Dg`, `Db`, `Ns` and `S` represent in the code snippet above:

- `Dg`: Number of blocks launched.
- `Db`: Number of threads per block.
- `Ns` (Optional, defaults to 0): Number of bytes per block to allocate for shared memory.
- `S` (Optional, defaults to 0): Specifies the associated stream with this kernel launch.

This summarises all the CUDA primitives found in Listing 2.7. A complete description of the runtime can be found in the CUDA API reference manual [74].

#### 2.3.1.2 OpenCL

OpenCL is and open standard parallel programming model targeted to heterogeneous systems. Unlike CUDA, OpenCL is cross-platform, which means that can be found in multiple devices from different vendors, such as AMD, NVIDIA or Xilinx. OpenCL targets diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms.

OpenCL defines *OpenCL C*, a C-like programming language used to write compute kernels. Besides that, it also defines an API to manage the device memory and launch the different kernels defined in the *OpenCL C* language. The standard is supported in the following languages and platforms:

| **GPU Computing Applications** | | | | | | |
|---|---|---|---|---|---|---|
| Libraries and Middleware | | | | | | |
| DeepCL | clFFT<br>clBLAS<br>clSpMV<br>libCL | ViennaCL<br>VOBLA<br>clMAGMA | clpp<br>M3 | VSI/Pro | Bullet Physics<br>ASL | MatCL<br>OpenCLLink |
| Programming Languages | | | | | | |
| Native support | | Third-party support | | | | |
| C | C++ | CLFORTRAN | | PyOpenCL<br>Perl Wrappers | | Java<br>.NET<br>Perl |

Table 2.3: List of some of the platforms, languages and libraries that support OpenCL acceleration. An updated list can be found in [52].

Similar to CUDA, OpenCL programming model can be divided into the following points:

- Kernels and Host program: OpenCL defines two distinct units of execution: a host program that executes on the host and the kernels that execute on one or more OpenCL devices.
- Hierarchy of work groups: In OpenCL, the kernels are where the ”work” of a given computation occurs. This work takes place through work-items that execute in groups. At the same time, multiple work-groups can execute in parallel.
- Memory hierarchy: The memory model defines how the values in memory are seen by the different units of execution. OpenCL leaves this responsibility to the programmer implementing: memory regions visible to the host and the devices that share a context, memory objects defined by OpenCL API, shared virtual memory between the host and the devices, and a consistency model to control atomicity and fencing in read-write conflict situations.
- Heterogeneous programming: The OpenCL API enables the host to interact with the device through a *command-queue* the following message types: kernel-enqueue, memory commands to transfer data between contexts, and synchronization commands (see Figure 2.11).

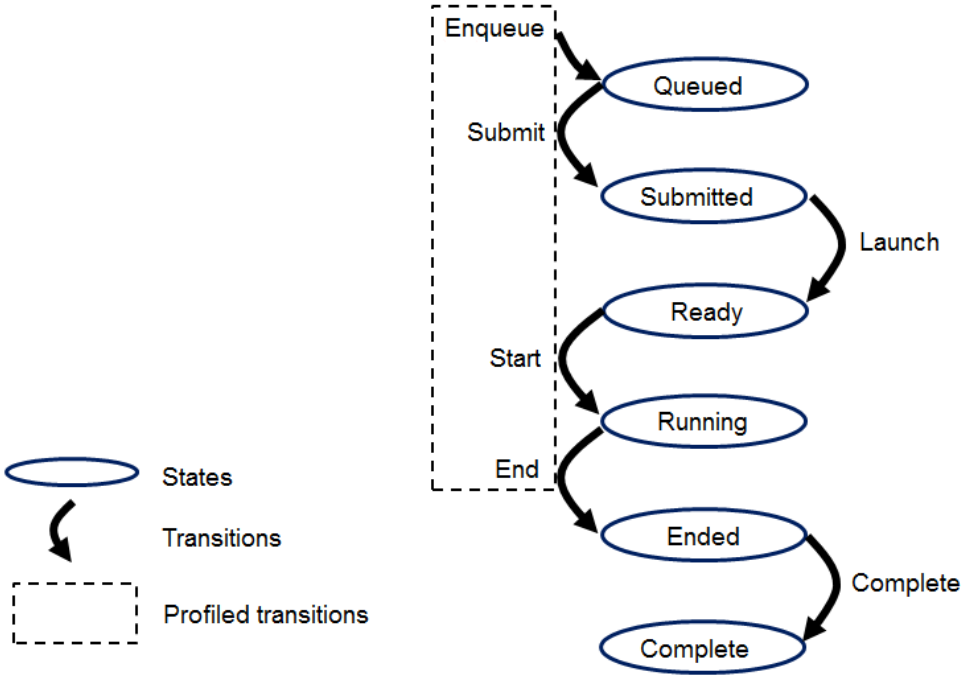



Figure 2.11: OpenCL program execution flow. Image credit: Khronos Group.

The previous Figure defines the execution states of a given kernel and the transitions between them, which is what defines the execution model of OpenCL. Following next we display a *VectorSum* example to review some of the features of the programming model of OpenCL:

```
// Device code
const char *kernelSource =                                                    "\n"\
"__kernel void VectorSum(__global int* A, __global int* B, __global int* C, int N)\n"\
"{                                                                            \n"\
"    int i = get_global_id(0);                                                \n"\
"    if (i < N)                                                               \n"\
"        C[i] = A[i] + B[i];                                                  \n"\
"}                                                                            \n";

// Host code
int main()
{
    // Allocate and initialize host vectors h_A, h_B, h_C
    ...

    // Create the compute kernel in our program
    cl_platform_id cpPlatform;
    cl_device_id device_id;
    clGetPlatformIDs(1, &cpPlatform, NULL);
    clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
    cl_context context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    cl_command_queue queue = clCreateCommandQueue(context, device_id, 0, &err);
```

```
23     cl_program program = clCreateProgramWithSource(context, 1, (const char **) &
       kernelSource, NULL, &err);
24     clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
25     cl_kernel kernel = clCreateKernel(program, "vecAdd", &err);
26
27     // Allocate vectors in device memory
28     float *d_A, *d_B, *d_C;
29     d_A = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
30     d_B = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
31     d_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);
32
33     // Copy vectors from host memory to device memory
34     clEnqueueWriteBuffer(queue, d_A, CL_TRUE, 0, bytes, h_A, 0, NULL, NULL);
35     clEnqueueWriteBuffer(queue, d_B, CL_TRUE, 0, bytes, h_B, 0, NULL, NULL);
36
37     // Set the arguments to our compute kernel
38     clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A);
39     clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_B);
40     clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_C);
41     clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
42
43     // Execute the kernel
44     int localSize = 64;
45     int globalSize = ceil(n/(float)localSize)*localSize;
46     clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, &localSize, 0, NULL,
       NULL);
47
48     // Wait for the command queue to get serviced before reading back results
49     clFinish(queue);
50
51     // Copy result from device memory to host memory
52     // h_C contains the result in host memory
53     clEnqueueReadBuffer(queue, d_C, CL_TRUE, 0, bytes, h_C, 0, NULL, NULL );
54
55     // Free device memory
56     clReleaseMemObject(d_a);
57     clReleaseMemObject(d_b);
58     clReleaseMemObject(d_c);
59
60     // Release rest of resources
61     clReleaseProgram(program);
62     clReleaseKernel(kernel);
63     clReleaseCommandQueue(queue);
64     clReleaseContext(context);
65
66     // Free host memory
67     ...
68 }
```

Listing 2.8: VectorSum sample in OpenCL.

By comparing the samples provided in Listing 2.7 and 2.8, we find that OpenCL is more lengthy and verbose than CUDA, partially caused by the *OpenCL C* factor. In the example above we saw different functions to build and manage the execution of the kernels written in *OpenCL C*. Next, we are going to provide a short description of each of the elements of the OpenCL API found in the above Listing:

- `cl_platform_id`: Identifies a platform, which consists of one host plus one or more compute devices, it can get retrieved with `clGetPlatformIDs`.
- `cl_device_id`: Identifies a compute device within a platform, can get retrieved with `clGetDeviceIDs`.
- `cl_context`: Represents an OpenCL context created with `clCreateContext`. A context is the environment within which the kernels execute and the domain in which synchronization and memory management get defined.
- `cl_command_queue`: Identifies a command queue and can be created using the `clCreateCommandQueue` function.
- `cl_program`: Indentifies a *OpenCL C* program. A source code should be loaded into the program object with the `clCreateProgramWithSource` function and then compiled with `clBuildProgram`.
- `cl_kernel`: Represents a compute kernel within the program object. The kernel should be specified with the `kernel` or `__kernel` keyword. Can be defined with `clCreateKernel`.

We can also find memory management directives:

- `clCreateBuffer(cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)`: Similar to an allocation, this function creates a buffer object of `size` bytes. `clCreateBuffer` returns the pointer to the device buffer data, which can be used for issuing commands onto the related command queue.
- `clEnqueueWriteBuffer(cl_command_queue command_queue, cl_mem buffer, cl_bool blocking_write, size_t offset, size_t cb, const void *ptr, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)`: Enqueues commands to write to a buffer object from host memory. `command_queue` refers to the command-queue in which the command will be issued. `buffer` refers to a valid buffer object present in the device. It writes `cb` bytes of data from the host buffer `ptr` to the device buffer `buffer`.
- `clReleaseMemObject(cl_mem memobj)`: Decreaments the memory object reference count by freeing `memobj`.

In *OpenCL C*, the `__kernel` or `kernel` specifier qualifies a function as a kernel. Kernels can be configured, executed and managed through the following functions:

- `clSetKernelArg(cl_kernel kernel, cl_uint arg_index, size_t arg_size, const void *arg_value)`: Used to set the argument value for a specific argument of a kernel.
- `clEnqueueNDRangeKernel(cl_command_queue command_queue, cl_kernel kernel, cl_uint work_dim, const size_t *global_work_offset, const size_t *global_work_size, const size_t *local_work_size, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)`: Enqueues a command to execute a kernel on a device.
- `clFinish(cl_command_queue command_queue)`: Blocks until all the queued OpenCL commands in the command-queue specified are issued to the associated device and have completed.

A further description of the OpenCL API can be found in the Khronos OpenCL Registry [44].

#### 2.3.1.3 OpenACC

OpenACC is a directive-based parallel programming model designed to ease the portability of code to a wide-variety of heterogeneous HPC platforms [76].

The execution model presented by OpenACC consists on using any parallel device present in the system, such as a GPU or a multi-core CPU, to accelerate problems simply using directives. OpenACC presents three levels of parallelism:

- *gang*: Coarse-grain parallelism. An accelerator can launch a number of *gangs*.
- *worker*: Each *gang* has one or more *workers*. Defines a fine-grained parallelism.
- *vector*: Represents the SIMD or vector operations within each *worker*.

OpenACC manages gang partitioning based on direct code analysis, where when the program reaches a loop, iterations are divided across gangs for parallel execution. The workload of each gang can be distributed among its associated workers based on the allocated resources. At the same time, it is also possible to define a finer level of parallelism with vector partitioning.

Since memory in an accelerator may be discrete from host memory, OpenACC handles all data movement between the host and the device using Direct Memory Access (DMA) transfers through the host thread. Similarly, the accelerator may not be able to read or write to the host memory.

Like in OpenMP and other directive based APIs, OpenACC employs macros to specify directives in all its supported languages: C, C++ and Fortran. For example, in C and C++ all OpenACC sections must be scoped with the `#pragma acc` keywords.

OpenACC provides heterogeneity by introducing the `device_type` clause. This clause determines in which devices the subsequent directives in the macro should execute, for example:

```
#pragma acc loop gang device_type(foo) worker
```

In this directive, `worker` is a device-specific clause that should be executed on `foo`, while the rest of the directive can be executed on any device, including `foo`. `device_type` may also contain an asterisk, to indicate that the subsequent clauses can be executed in all device types that are not named in any other `device_type` clause in that macro.

Following next, we are going to list a series of compute constructs that are going to determine how the scoped code in the directives is going to execute:

- `#pragma acc parallel` [*clause-list*]: One or more gangs of workers are creaed to execute the accelerator parallel region.
- `#pragma acc serial` [*clause-list*]: The behaviour is equivalent with the parallel construct except that it always executes with a single gang of a single worker with a vector length of one.
- `#pragma acc kernels` [*clause-list*]: The compiler will create a series of kernels for the scoped code based on the encountered nesting. These kernels will launch in order on the specified device.
- `#pragma acc loop` [*clause-list*]: When a loop construct is encountered, it is partitioned and executed on the target device following the type of parallelism defined in its clauses.

Each of these constructs support a series of clauses that can be consulted in OpenACC's specification [2], following next we list some of the most relevant ones:

- `if` [*condition*]: The region will execute on the current device only If the condition inside the `if` clause evaluates to true.
- `wait` [*wait-argument*]: When this clause appears in the directive, the current computing activity will wait for the completion of asynchronous operations.
- `reduction`(*op:var-list*): Specifies a reduction operator for one or more vars. For a further explanation on reductions, please refer to Section 2.2.1.2.

- `copyin`, `copyout`, `copy`: These clauses handle the data dependencies of the scoped regions. Variables within the `copyin` clause will be transferred to the specified device. Variables within the `copyout` clause will be transferred from the device to the host. Finally, variables within the `copy` clause will be copied and retrieved from the device once the computation is done.

OpenACC and other directive based APIs offer a simple solution to perform computation with minimal effort by simply annotating sequential code. To demonstrate this in OpenACC, we display a *VectorSum* example in Listing 2.9:

```
#pragma acc kernels copyin(A[0:size],B[0:size]), copyout(C[0:size])
for(int i=0; i<size; i++) {
    C[i] = A[i] + B[i];
}
```

Listing 2.9: VectorSum sample in OpenACC.

The above listing creates a kernel to perform the loop computation, in its clause we can see how it inputs and outputs the appropriate variables to the device. Therefore, we can find that one of the greatest features of OpenACC is creating GPU code adding barely a line of code.

However, this simplicity comes at the cost of a decreased performance if we compare the speedup between a program annotated with OpenACC pragmas with a program wrote directly in GPGPU low level code (CUDA, OpenCL) [49]. In addition, the macros can become very verbose when dealing with specific types of parallelisation and dependencies:

```
#pragma acc parallel loop present(row_offsets,cols,Acoefs,xcoefs,ycoefs) \
device_type(nvidia) vector_length(32) gang worker num_workers(32)
for(int i=0;i<num_rows;i++) {
    double sum=0;
    int row_start=row_offsets[i];
    int row_end=row_offsets[i+1];
    #pragma acc loop reduction(+:sum) device_type(nvidia) vector
    for(int j=row_start;j<row_end;j++) {
        unsigned int Acol=cols[j];
        double Acoef=Acoefs[j];
        double xcoef=xcoefs[Acol];
        sum+=Acoef*xcoef;
    }
    ycoefs[i]=sum;
}
```

Listing 2.10: Macro verbosity example in OpenACC.

As we can see in Listing 2.10, OpenACC allows the end user to create more advanced solutions to accelerate computation. However, most of the time these solutions require understanding the intrinsics of the target platform, so writing device specific code directly could be more beneficial and accessible to the programmer.

#### 2.3.1.4 OpenMP GPU

We saw in Section 2.2.1.2 how OpenMP can be used to accelerate problems in many-core systems. However, since version 4.0, the specification provided a set of directives to offload blocks of code to a device (GPU, FPGA, etc.).

For that, OpenMP introduced the `target` construct, which maps the specified variables to a device data environment and executes the construct on the device[10]:

```
#pragma omp target [clauses]
```

Following next, Listing 2.11 displays a matrix multiplication code using the `target` construct that runs in the GPU:

```
1  #define pA(i,j) (pA[((i)*N) + (j)])
2  #define pB(i,j) (pB[((i)*N) + (j)])
3  #define pC(i,j) (pC[((i)*N) + (j)])
4
5  ...
6
7  #pragma omp target data map (to: pA[0:N*N],pB[0:N*N]) \
8  map (tofrom: pC[0:N*N])
9  #pragma omp target teams distribute \
10 parallel for collapse(2) private(i,j,k)
11 for(i=0;i<N;i++) {
12     for(j=0;j<N;j++) {
13         for(k=0;k<N;k++) {
14             pC(i,j)+=pA(i,k)*pB(k,j);
15         }
16     }
17 }
```

Listing 2.11: Matrix multiplication offloaded to GPU using OpenMP pragma.

According to a study published by IBM [72], offloading the computation performed in the previous code to the GPU clearly gives performance benefits for bigger matrices. All of that, just adding barely two lines of code.

Another experiment performed by [64], shows how GPU offloading can be several orders of magnitude faster than a serial implementation for the LULESH benchmark [54]. In this case, the authors of this work employed a Pascal P100 CPU and two Power8 CPUs running at a frequency of 4GHz (a bar-chart comparison can be found in Figure 2.12).

[10] https://www.openmp.org/spec-html/5.0/openmpsu60.html

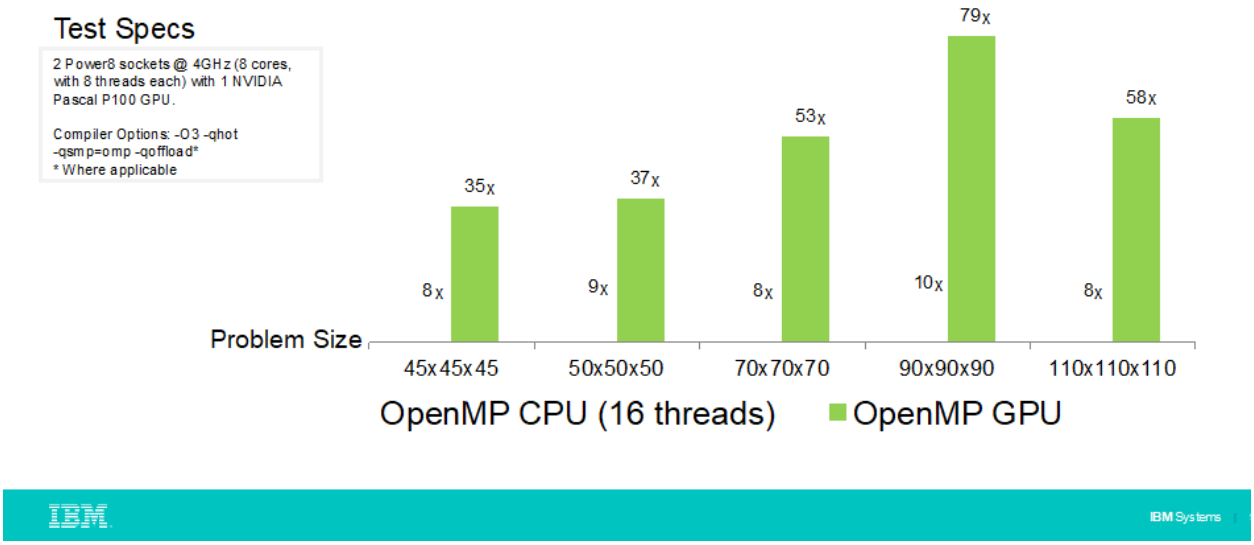



Figure 2.12: LULESH – Speedup Over Serial (Higher is Better). Image credit: [64].

However, besides the theoretical simplicity to benefit from the use of an accelerator in OpenMP, it is required that the compiler running in the target platform supports this feature. To date, not many built-in compilers support these offloading directives without explicit changes to their build scripts[11].

#### 2.3.1.5 OmpSs GPU

In Section 2.2.1.3, we studied the OmpSs programming model and we mentioned the heterogeneity aspect of it. In this Section we are going to explore the `target` construct, which brings heterogeneity to the platform.

OmpSs, provides heterogeneity by adding the *target* construct. The functionality of this construct is to specify the series of devices that can run a given element. As some of the other constructs studied previously, this construct can be applied over task constructs, function definitions or function headers. The syntax of the construct is the following:

```
#pragma omp target [clauses]
```

The valid clauses for the `target` construct are the following:

- `device(target-device)`: It allows the programmer to specify the devices that should be targeting the construct. OmpSs supports CUDA, OpenCL and SMP. If no device is provided then SMP is assumed.

[11] `https://kristerw.blogspot.com/2017/04/building-gcc-with-support-for-nvidia.html`

- `copy_in(list-of-variables)`: It specifies that a set of data should be transferred to the device before the associated code gets executed.
- `copy_out(list-of-variables)`: It specifies the set of data that should be transferred back to the host after the execution in the device.
- `copy_inout(list-of-variables)`: It specifies that the set of data should be transferred to the device before running the associated code block and then retrieved back to the host.
- `copy_deps`: If this clause is set, the dependence clauses will copy semantics (i.e., `in` will also be considered `copy_in` and `output` `copy_out`).
- `implements(function-name)`: The function provided in the clause is an alternate implementation of the associated code block for the target devices. OmpSs will ensure that the device is capable to run the alternate function.
- `ndrange(n, G1,…, Gn, L1,…,Ln)`: Sets the thread hierarchy used to run the associated kernel (or code block). In a two dimensional scenario, the `ndrange` clause contains five parameters: The first one is the number of dimensions of the kernel. The second and third one is the number of threads to be launched per dimension. And finally, the fourth and fifth ones are the group size per dimension.
- `shmem(size_t)`: Specifies the amount of memory the runtime will allocate for CUDA and OpenCL kernels.
- `file(file_name)`: For OpenCL kernels, specifies the file that contains the program object to load the kernel.

In the following we display a *VectorSum* example using OmpSs CUDA:

```
/* cuda-kernels.cu */
extern "C" {  // We specify extern "C" because we will call them from a C code

    __global__ void VectorSum(float* A, float* B, float* C, int N)
    {
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        if (i < N)
        {
            C[i] = A[i] + B[i];
        }
    }

} /* extern "C" */
```

Considering the above CUDA kernel, we can use it in an OmpSs using the `target` construct:

```
#pragma omp target device(cuda) copy_deps ndrange(1, N, 1)
#pragma omp task in(A[0 : N-1], B[0 : N-1]) out(C[0 : N-1])
__global__ void VectorSum(float* A, float* B, float* C, int N)

// Host code
int main()
{
    // Allocate and initialize host input vectors h_A and h_B
    ...

    VectorSum(h_A, h_B, h_C, N);

    #pragma omp taskwait

}
```

Listing 2.12: VectorSum sample in OmpSs CUDA.

As we can see in Listing 2.12, OmpSs abstracts the programmer from the memory management of CUDA or OpenCL, which makes the whole process of GPU programming less error prone.

### 2.3.2 Programming models for critical systems

As we have seen in the introduction of this Section, GPUs can provide an increased computational power due to their massively parallel architecture.

Modern automotive, railway and aerospace systems are exploring GPU solutions to meet their demand for computational power, however all the programming models employed in the critical domain are required to be amenable for software certification, thus ensuring functional safety. Low level programming models, such as those seen in Section 2.3.1, offer programmability on these devices, however, their software certification goes against safety standards, such as MISRA C [71].

In addition, CUDA and OpenCL as well as similar programming models, are not supported on many low-end devices, which also justifies the urgent need to find alternative solutions for high-criticality systems. In this Section we explore solutions based on safety critical graphics APIs, specifically we will cover OpenGL SC 2 in Section 2.3.2.1, then Section 2.3.2.2 introduces Brook Auto, followed by Section 2.3.2.3, in which we will explore Vulkan SC.

#### 2.3.2.1 OpenGL SC 2

OpenGL SC 2.0 is a Safety Critical graphics API oriented towards safety critical markets, including railway, avionics and automotive, between others. The specification of the API is complementary to the design guidelines defined by the Khronos Safety Critical Advisory Forum (KSCAF), which provides interop API standards for safety critical systems.

OpenGL SC 2.0 is in essence a subset of OpenGL ES 2.0 [45] functionality, aided to the safety critical domain. The API provides OpenGL Shading Language (GLSL) shader programmability complying with stringent safety standards for avionics and automotive systems, including FAA DO-178C, EASA ED-12C Level A, and ISO 26262. The fact that OpenGL SC 2.0 is a subset of OpenGL ES 2.0 makes that all the programs created in OpenGL SC 2.0 can be run in OpenGL ES 2.0, but not the other way around. However, the offered compatibility makes the API definition and header not to be MISRA-C compliant.

OpenGL ES 2.0 capabilities are supported by any current embedded or desktop GPU, meaning that OpenGL SC 2.0 can be functionally emulated on top of any device that supports OpenGL ES 2.0 by simply including OpenGL SC 2.0 Khronos headers. This makes the development of OpenGL SC 2.0 applications possible in every single GPU platform.

The wide availability and support of the standard in most existing GPU devices makes OpenGL ES 2.0 a perfect candidate to create portable applications capable to run in any system with a GPU, including those that do not support GPGPU oriented APIs, such as CUDA or OpenCL. Thus, OpenGL shading capabilities can shape with extra effort a universal general purpose GPU programming model which might provide benefits towards the standardisation of parallel code on GPU devices. However, OpenGL ES 2.0 presents a series of challenges in order to perform general purpose computations [86], following next we present the most representative ones:

- OpenGL ES 2.0 doesn't support compute shaders, therefore the GPGPU computations should to be done either in the vertex or the fragment shader (or both). The required parameters are passed through to the fragment shader as OpenGL `varyings`.
- OpenGL ES 2.0 only supports single byte format for texture values (RGBA). Thus, a mechanism to represent 32-bit floating point and other numerical formats is needed.
- Similar to the previous point, it is also required a mechanism to encode the shader computing output in a format other than normalised byte values.
- A fragment shader cannot output more than one array, therefore if we need to return multiple arrays, we would need multiple fragment shaders.
- Atomic counters aren't natively supported in OpenGL ES 2.0.

OpenGL ES 2.0 GPGPU workflow requires the programmer to understand the different rendering stages of the API to create simple kernels. The framework can result verbose and difficult to use as it doesn't provide any high level abstraction layer to develop GPGPU applications. For example: algorithms that require certain kind of atomicity can become difficult to read and maintain, since the programmer must handle the atomicity in multiple rendering passes explicitly.

Following next, we display a *VectorSum* kernel using a wrapped OpenGL SC 2.0 version:

```
precision highp float;
varying vec2 v_texCoord;
uniform sampler2D s_texture;
uniform sampler2D s_texture2;

...

void main()
{
    highp float reconstructed_A;
    highp float reconstructed_B;
    highp float sum_result;

    reconstruct(reconstructed_A, s_texture, v_texCoord);
    reconstruct(reconstructed_B, s_texture2, v_texCoord);
    sum_result = reconstructed_A + reconstructed_B;
    encode_output(sum_result);
}
```

Listing 2.13: VectorSum sample in OpenGL SC 2.0.

In the example above, the texture data (`s_texture`) is reconstructed onto the desired numerical format (`reconstructed_A`) to perform the computation. Then, the output must be encoded back to a GPU readable format. Once the result is available, we must read the pixels from the resulting texture using `glReadPixels` to obtain the result in the CPU. Finally, the output pixel data must be reconstructed to the desired format to get the correct result:

```
#define reconstruct_from_normalized_bytes(reconstructed, bytes)\
{\
    float tmp = floor(256.0*bytes[0] - (bytes[0]/255.0));\
    reconstructed = tmp;\
    tmp = floor(256.0*bytes[1] - (bytes[1]/255.0))*256.0;\
    reconstructed += tmp;\
    tmp = floor(256.0*bytes[2] - (bytes[2]/255.0))*256.0*256.0;\
    reconstructed += tmp;\
    tmp = floor(256.0*bytes[3] - (bytes[3]/255.0))*256.0*256.0*256.0;\
    reconstructed += tmp;\
    if(bytes[3] > 0.5) reconstructed -= 4294967296.0;\
}
```

Listing 2.14: Reconstructing an `int` given an array of 4 bytes.

With this, we demonstrated that OpenGL SC 2.0 can support different numeric formats employing the appropriate reconstructions. However, defining said functionality can become complex as it is dependent on the precision of the target platform. Modern GPGPU programming models ease this task, however they are not always available in the target platform.

#### 2.3.2.2 Brook Auto

Brook Auto [87] is an open source high level GPGPU programming language that extends C to include data-parallel constructs in order to employ the GPU as a streaming co-processor. Brook Auto specification consists of a well-defined subset of functionalities from the CUDA predecessor, Brook GPU [18], which are tailored to the automotive domain. The language is amenable to software certification and portable across every embedded GPU of the sector, thanks to the multiple run-time back-ends supported: CPU, OpenGL ES 2.0, DirectX 9, OpenGL, AMD CTM.

In terms of software certification, several other popular GPGPU languages, such as CUDA or OpenCL, are found to be non-compliant with the ISO26262 automotive standard certification. Among those violated constraints we find: a) restricted use of pointers, b) no dynamic memory allocation, c) static verification of program properties, d) resilience to faults and e) fault propagation. Brook Auto, however, guarantees ISO26252 compliance as it defines a series of rules over Brook to assure this purpose:

- Brook doesn't support pointers: Instead, it uses *streams* to process data in a kernel.
- Brook won't let the GPU threads to access beyond the allocated memory.
- Brook enforces upper-bounds to the loop constructs in the kernels, so that the maximum trip count can be deduced.
- Brook doesn't support recursion.
- Brook restricts the number of inputs and outputs to the ones supported in the target platform to avoid emulation cost.

Brook Auto compiler consists of a modified version of the Brook compiler to enforce the ISO26262 compliant subset. In [89] the authors have performed an academic assessment of the tool qualification of this compiler, named BRASIL, according to ISO26262, which provides assurance that the compiler can be used for the development of safety critical systems. To our knowledge, BRASIL is currently the first and only available qualifiable GPGPU compiler for high intergrity systems. In Listing 2.15 we present a Brook Auto *VectorSum* example:

```
kernel void vectorSum(float a<>, float b<>, float c<>) {
    c = a + b;
}

int main(void) {
    float a_h[100], b_h[100], c_h[100];
    float a_d<100>, b_d<100>, c_d<100>;

    streamRead(a_d, a_h);
```

```
10     streamRead(b_d, b_h);
11     vectorSum(a_d, b_d, c_d);
12     streamWrite(c_d, c_h);
13 }
```

Listing 2.15: Brook Auto *VectorSum* example.

As we can see from the above listing, the language syntax is very similar to that of CUDA, as Brook greatly influenced the design of CUDA [88]. This gives the benefit of a simpler portability between CUDA and Brook programs, while also providing wide support to platforms where other GPGPU languages aren't supported.

Brook Auto handles all intrinsic details pertaining to the selected back-end and platform, such as the numeric formats. This enhances productivity as the source-to-source compiler ensures proper code generation by relieving the programmer of the responsibility of managing textures and shader pipelines.

Regarding performance, Brook Auto delivers an increased speedup over its CPU back-end. Its developers also ran a series of benchmarks to evaluate the language. One representative example is *sgemm*, where Brook Auto back-end delivers between 50 and 90% of the performance exhibited by the handwritten application depending on the input size. According to the authors: "This difference in performance comes from the runtime overhead of Brook, and it is consistent with the overhead of the original Brook implementation over the desktop version of OpenGL". However, this decreased performance comes with the benefit of a better complexity and productivity: The Brook version was written in less than 2 hours with 70 lines of code, while the OpenGL ES 2 was written and optimized in more than a year, and contains 1500 lines of code. Similar results were presented with an avionics case study from Airbus Defence and Space, on an avionics-grade GPU and a certified OpenGL SC 2.0 driver provided by CoreAVI in [16].

For all these reasons, Brook Auto is a very suitable selection for GPGPU computing in embedded platforms that require compliance with the safety critical market.

#### 2.3.2.3 Vulkan SC

Vulkan SC is a safety critical API focused to parallel processing in modern graphic accelerators. The standard provides multithreading and low-level GPU access capabilities to enable finer-grained control in the application with the goal of providing performance gains [46]. Nevertheless, this leaves the responsibility of selecting the appropriate configuration in the target device in the hands of the programmer.

Although Vulkan SC is still under development, CoreAVI has developed VkCore®SC [26], a compute driver aligned with the Vulkan SC API from the Khronos Group. The driver is designed with the goal of achieving high performance and flexibility while offering certification options for its use

in critical systems (RTCA DO-178C/EUROCAE ED-12C, DAL A, ISO 26262, etc.). VkCore®SC presents a series of features suitable for safety critical systems, some of which are the following:

- GPU shader programming and graphics compute: VkCore employs an offline GLSL compiler to convert the shader source programs (Vertex, Fragment and Geometry) into Vulkan pipeline objects used at runtime.
- Multi-core partitioning: The computation can be distributed throughout the cores present in the target platform.
- GPU virtualization and hypervisor RTOS capabilities to support mixed criticality systems: VkCore SC includes a virtualisation manager to enable multiple graphics rendering partitions to drive single, or multiple GPUs. It's architecture supports single or multi-threaded applications in multiple address spaces as well as sharing a single GPU by applications residing in different Guest OSs.

Regarding the programmability of Vulkan, it can result verbose if we simply want to perform GPGPU compute, as the programmer should setup the environment, bindings, pipelines, and commands more appropriate for shader computing. However, unlike OpenGL ES 2.0, the compute shader code is much more readable and allows the use of thread identifiers. Following next, I show a *VectorSum* example running in a compute shader in Vulkan:

```
layout (std430, set=0, binding=0) buffer inA { int a[]; };
layout (std430, set=0, binding=1) buffer inB { int b[]; };
layout (std430, set=0, binding=2) buffer outR { int result[]; };

void main() {
    const uint i = gl_GlobalInvocationID.x;
    result[i] = a[i] + b[i];
}
```

Listing 2.16: Vulkan SC *VectorSum* example.

As we can see in the Listing, Vulkan SC compute code is recognisable as it's very similar to what we find in other GPGPU languages as CUDA or OpenCL. However, the verbose aspect of the API is in the initialisation, setup and memory allocation part, which consumed around 500 lines of code for this simplistic application. Similar results were reported also in [90] with the avionics application used in [16]. In order to overcome this and allow for higher productivity and lower certification effort, the authors mention that they are working on a Vulkan SC backed for Brook Auto/BRASIL which is currently under development, which can bring its benefits to Vulkan SC, too. Finally, one of the major advantages of Vulkan SC over OpenGL ES 2.0, is that the later does not explicitly support graphics compute, which makes Vulkan SC more appropriate for GPGPU code.

# Chapter 3

# The UP2DATE Project

In this chapter we provide the necessary context related to the UP2DATE H2020 project, to which this thesis contributed for the selection of the baseline hardware platform. Section 3.1 describes the motivations and objectives of the UP2DATE project. Following next, Section 3.2 exposes the project specifications in regards to the employed hardware, and finally, Section 3.3 describes the GPU4S benchmark suite, which was employed and extended to measure the performance of the proposed heterogeneous and multi-core candidate platforms.

## 3.1 Overview

In addition to the core requirements of safety-critical systems which we covered in the previous chapter, there is a recent interest in the safety critical market to adapt Over-The-Air Software Updates (OTASU) technologies on their embedded systems. This feature is increasingly used in the automotive sector and expands to other critical systems, too. It gives vendors the ability to enhance and add functionality to the software layer running on their devices without the need for explicit manual maintenance, which benefits a number of industries, as manual labour can sometimes become time-consuming, costly and in some cases dangerous:

- Time-consuming: Some software updates may require manual maintenance, which means that the device needs specific preparations before the operator can work on it.
- Costly: It is common that during an update, the system becomes unavailable. This is a problem for several domains, as it halts benefits while the device remains inoperable (i.e. planes, industrial machinery, etc.).
- Dangerous: Some software updates need to be performed in hazardous conditions due to the location of the device to update. By automating the procedure we remove any dangerous factor that can harm the operator while updating the device.

All these reasons motivate the industry to adopt automatic updates for the majority of their critical and non-critical devices.

However, OTASU presents a series of challenges in the safety-critical domain [3]:

- **Safety:** Current safety standards are not designed to cover OTASU, thus, ensuring functional safety after an update is challenging as there are no procedures that can guarantee this.
- **Security:** The fact that OTASU relies on online updates makes cybersecurity crucial, as fraudulent software can compromise the functional safety of the device to update.
- **Availability:** It is common that during an upgrade, the system becomes unavailable, as this activity may involve manual labour and a complete re-evaluation of the critical components which need to ensure functional integrity. For this reason, if we consider OTASU as a solution, we must ensure compliance with the Safety and Security (SASE) requirements before completing an online update.
- **Performance:** As we mentioned in Section 2.1, current software demands higher performance, so the state-of-the-art mixed-criticality devices consist of hardware platforms based on multi-core processors and accelerators capable to satisfy the performance requirements of modern needs. This demand of performance opens new challenges that have a negative impact in system safety and security.

The mission of the Horizon 2020 UP2DATE project is to address these issues by working on new software update solutions focused towards heterogeneous high-performance Mixed Criticality Cyber-Physical System (MCCPS). For this, UP2DATE proposes adopting a design-by-contract approach [17] to enable modular updates without affecting the SASE properties of the updated system. To this end, UP2DATE defines two types of contracts, **horizontal**: which covers the dependencies between connected software components; and **vertical**: which ensures that the updated software complies with the SASE requirements within the hardware platform.

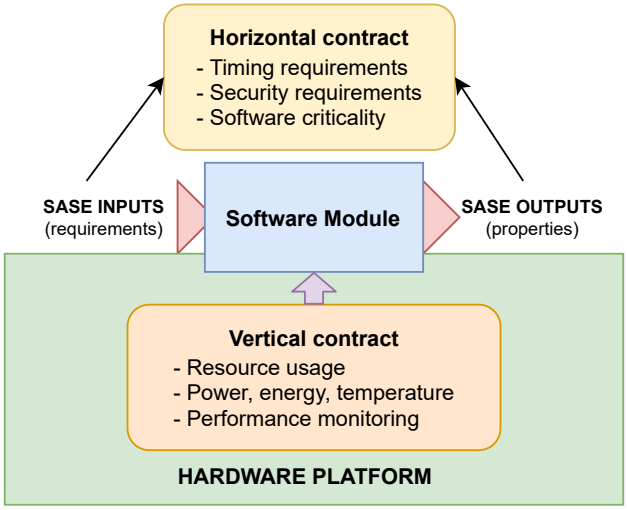



Figure 3.1: UP2DATE SASE contracts.

The novelty introduced at UP2DATE's design-by-contract approach, is that the SASE constraints and the hardware resource allocation requirements become an integral element of the contracts, as we can see in Figure 3.1.

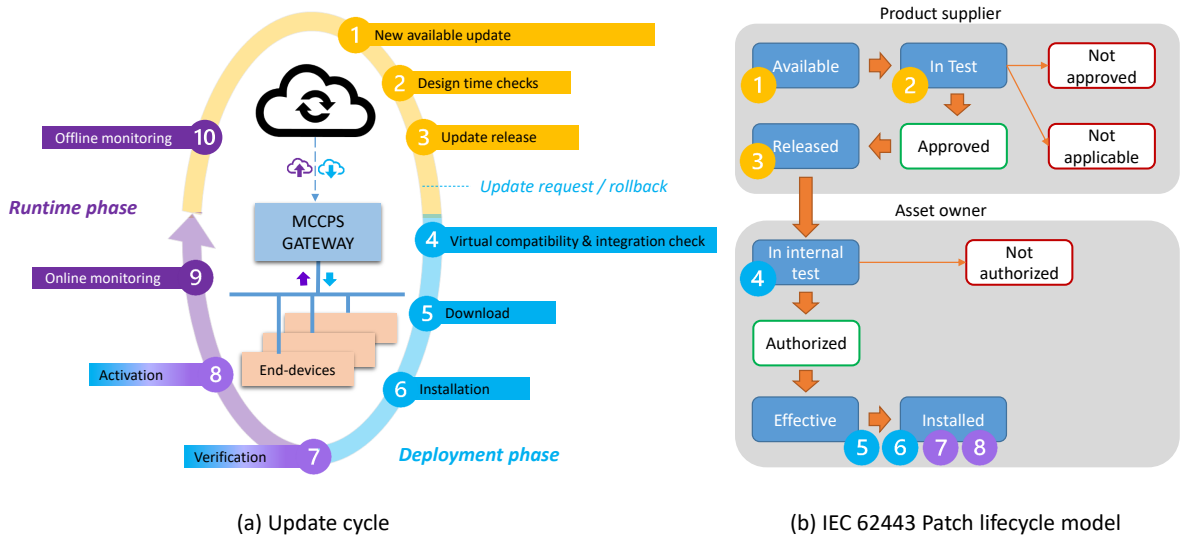



Figure 3.2: UP2DATE software update cycle.

All of this is comprised in the UP2DATE middleware, which is responsible of the entire update cycle, starting with the download of the OTASU from an external server, and finishing by ensuring compliance with the SASE contracts through different monitoring services. The high level workflow of the described mechanism is showcased in Figure 3.2.

## 3.2 Platform requirements

The rationale for the selection of the candidate UP2DATE platforms is based on the requirements described in the project proposal (summarised in Section 3.2.1) and on an exploration of the state of the art carried out in the early stages of the project (briefly introduced in Section 3.2.2).

### 3.2.1 Project proposal requirements

The project summary exposes a series of concepts that align with the requirement set from the ICT-01-2019 call for proposals [25]: *UP2DATE will focus on heterogeneous, high performance platforms on which we will develop observability and controllability solutions which will support on-line updates preserving safety and security of mixed-criticality workloads.*

In the following, we revise these requirements and explain their impact on the platform selection:

1. **Heterogeneous platforms:** Hardware platforms containing both, a CPU and an accelerator, can provide high performance at low power consumption. These solutions are becoming trending within the embedded and safety critical domain as throughput based solutions are becoming more popular due to the speed-up benefits they bring.
2. **High-Performance:** Currently, only accelerators like GPUs and FPGAs can guarantee the very high performance requirements of modern complex functionalities in critical systems, such as autonomous driving. This makes accelerators crucial for any throughput based task, and therefore influenced greatly its presence on the project.
3. **Observability:** Being able to monitor the platform state through Performance Monitor Counters (PMCs) or any other metric is key as monitoring is an essential part for the SASE requirements present in many applications.
4. **Controllabilty:** The target platform should be capable to support resource configurability to guarantee compliance with task requirements (i.e. contention).
5. **Online updates:** The platform should be able to replace and update software employing OTASU's technology, and the device should offer the possibility to do it while functioning.
6. **Safety and Security:** The candidate platforms should comply with SASE standards.
7. **Mixed Criticality:** The systems should support the co-existence of both, high and low criticality tasks, while preserving functional safety and security.
8. **Platform characterisation and benchmarking:** The selection of the platform should be based on the results granted by the execution of benchmarks and the characterisation of the device. This thesis particularly contributes to this goal. Chapter 4 and 5 will expose further details about this requirement.

### 3.2.2 Explored requirements

This Section describes identified requirements in the initial phases of the project:

1. **Space partitioning:** The target platform must offer mechanisms to isolate the memory of the software entities. Thus, any malfunctions or security failures will not have an impact on the rest of the system. Two features that enable space partitioning are MMUs and MPUs.
2. **Time partitioning:** The target platform must provide mechanisms to allocate timing budgets to each software entity, so that the Worst Case Execution Time (WCET) is lower than its deadline, which contributes to the SASE properties of the system.

3. **Software Randomisation:** This mechanism [27] [56] [61] provides a way to compute the WCET of critical tasks employing a technique called Probabilistic Timing Analysis [20]. Candidate platforms should therefore offer the possibility to implement software randomisation methods.

4. **Virtualisation:** The candidate platform must support some form of virtualisation solution to allow the coexistence of multiple operating systems running on the same host. This solution facilitates the implementation of the partitioning concept revised above.

From this list of requirements, Chapter 4 describes the devices that conform with the specification. But first, we describe in Section 3.3 the benchmark suite that is employed to evaluate all candidate devices and thus select the research platform.

## 3.3 The GPU4S Benchmark Suite

As we described in Section 3.2, our candidate platforms are high-performance heterogeneous embedded devices, so to evaluate them we need a benchmark suite capable of measuring the performance of the accelerators and CPUs present in each platform. However, most state-of-the-art benchmark suites do not meet this criteria.

As a solution, we decided to use GPU4S Bench, an open source benchmark suite developed at BSC for Space On-board Processing Systems [79] in the context of the GPU4S project [59] funded by the ESA. This suite contains many representative algorithms relevant for various safety-critical domains, with a focus on GPU evaluation, offering both CUDA and OpenCL implementations, as well as CPU reference code for each algorithm.

In GPU4S Bench each algorithm includes two or three variants depending on the literature:

- **Naïve:** Represents the most straightforward way to implement the algorithm.
- **Hand-optimised:** Employs different state-of-the-art techniques to further improve the performance of the benchmark.
- **Vendor-provided:** When available, GPU4S Bench includes existing implementations of the algorithm present in widely used libraries, such as *cuBLAS*[1] or *cuFFT*[2]

Since GPU4S Bench was designed for the GPU4S project, it was focused only embedded GPUs and as a consequnece it did not support multicore processors. For that reason we ported and optimised the whole suite in OpenMP in order to make a more accurate and fair comparison of the CPU capabilities of each platform, as described in Section 3.3.1.

[1] `https://docs.nvidia.com/cuda/cublas/index.html`
[2] `https://docs.nvidia.com/cuda/cufft/index.html`

| **Benchmark** | **Multi-core CPU** | **GPU** |
|---|---|---|
| Matrix Multiplication | OpenBLAS | cuBLAS |
| Fast Fourier Transform | FFTW | cuFFT |
| 2D Matrix Convolution | OpenMP hand-optimised | CUDA hand-optimised |
| Finite Impulse Response Filter | OpenMP hand-optimised | CUDA hand-optimised |
| Local Response Normalisation | OpenMP hand-optimised | CUDA hand-optimised |
| Max Pooling | OpenMP hand-optimised | CUDA hand-optimised |
| Rectified Linear Unit (ReLU) | OpenMP hand-optimised | CUDA hand-optimised |
| SoftMax | OpenMP hand-optimised | CUDA hand-optimised |
| CIFAR10 | OpenMP hand-optimised | CUDA hand-optimised |

Table 3.1: GPU4S Benchmarks used for the evaluation of the candidate platforms performed in this thesis. The versions used in the Multi-core CPU column, were implemented in the context of this thesis.

### 3.3.1 Porting GPU4S Bench to OpenMP

In order to extend the set of benchmarks with our OpenMP implementations, we decided to follow the same standard proposed in GPU4S Bench, which consists of two or three implementations per algorithm as discussed in Section 3.3.

All new OpenMP implementations were validated using the verification system included in the suite, which compares, with certain error tolerance, the output of our parallel OpenMP implementation against the verified output from the sequential reference code, included in every algorithm from the suite.

To continue the style of the benchmarking suite, we have followed the same approach and included vendor-provided library implementations wherever available, which is the case of the following two benchmarks:

- Matrix multiplication: *ATLAS* and *OpenBLAS*[3].
- Fast Fourier Transform: *FFTW*[4].

Table 3.1 displays the list of benchmarks we employed in the UP2DATE project to evaluate the candidate platforms. Note that in addition to algorithmic building blocks which cover multiple aerospace domains (observation/vision, telecommunication, machine learning), a complex inference application is included as well, for solving a classification task with 10 classes like CIFAR-10.

[3] `https://www.openblas.net/`
[4] `http://www.fftw.org/`

# Chapter 4

# Experimental Setup

Based on the requirement list exposed in Section 3.2, this Chapter presents and analyses the list of candidate platforms compliant with the exposed specification for the selection of the baseline research platform.

## 4.1 Candidate platforms

One of the main requirements limiting the list of embedded platforms to choose from is the heterogeneity factor, which implies that we must select an embedded device with a GPU or an FPGA.

For the GPU platforms, we find two family of products capable of safety certification:

- **NVIDIA Jetson:** Is a series of low-power embedded boards from NVIDIA designed for accelerating machine learning applications.
- **Renesas R-Car:** Is a series of embedded high-end SoCs for the automotive industry, specially focusing Advanced Driver-Assistance Systems (ADAS).

In a premature analysis, we found that the Jetson family offers higher theoretical performance, not only compared to the R-Car series, but to the majority of the embedded market. For this reason, we opted to benchmark two of the latests heterogeneous embedded boards from NVIDIA: the NVIDIA Jetson TX2 and the NVIDIA Jetson AGX Xavier.

As for the FPGA-based research platform, we selected the Xilinx Zynq Ultrascale+ [95], which is also capable of achieving functional safety certification.

All the candidate devices and toolchain will be described in detail in the following Sections.

### 4.1.1 NVIDIA Jetson TX2

This embedded platform contains four ARM A57 CPUs and two NVIDIA Denver cores, making it an hybrid architecture. It also counts with a Pascal-based GPU which supports ECC in its industrial version (TX2i)[1]. As for the software toolchain, we have used a clean installation of the NVIDIA Jetpack[2] 4.3, which comes with the following core features:

- Linux Kernel version: 4.9.140 / L4T 32.3.1
- Ubuntu version: Ubuntu18.04 LTS aarch64
- CUDA version: 10.0
- gcc version: 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)

In terms of power configurability, the device supports 5 defined performance modes:

| **Property** | **Mode** | | | | |
|---|---|---|---|---|---|
| | **MAXN** | **MAX-Q** | **MAX-P** | **MAX-P*** | **MAX-P** |
| Power budget | n/a | 7.5W | 15W | 15W | 15W |
| Mode ID | 0 | 1 | 2 | 3 | 4 |
| Online A57 CPU | 4 | 4 | 4 | 4 | 1 |
| Online D15 CPU | 2 | 0 | 2 | 0 | 1 |
| A57 CPU maximal frequency (MHz) | 2000 | 1200 | 1400 | 2000 | 345 |
| D15 CPU maximal frequency (MHz) | 2000 | n/a | 1400 | n/a | 2000 |
| GPU maximal frequency (MHz) | 1300 | 850 | 1122 | 1122 | 1122 |
| Memory maximal frequency (MHz) | 1866 | 1331 | 1600 | 1600 | 1600 |

Table 4.1: Manufacturer's Performance modes for Nvidia's TX2

The performance mode we have selected for the TX2 board uses its four ARM A57 CPUs running at 2000 MHz, and its pascal-based GPU running at 1122 MHz, all of that under a budget of 15 Watts. We decided to use the **MAX-P*** performance mode to explore the full capabilities of the multi-core parallelism of the platform. Moreover, for maximum performance in parallel workloads we need good load balancing, which is easier to achieve with homogeneous cores that have exactly the same performance. That's why we consider that mixing the Denver cores with the A57 cores for the benchmarking evaluation is not a good idea.

For that reason, all the multi-core devices under test will be evaluated with four cores running at their documented maximum theoretical frequency.

[1] `https://developer.nvidia.com/embedded/jetson-tx2i`
[2] `https://developer.nvidia.com/embedded/jetpack`

### 4.1.2 NVIDIA Jetson AGX Xavier

The NVIDIA Jetson AGX Xavier has a very similar software environment with the Jetson TX2, since it is its successor. As we explained in Section 2.1.2.1, the board features eight ARMv8.2 Carmel cores, and includes Volta GPU. It also includes ASICs and accelerators targeted towards Artificial Intelligence (AI).

In terms of functional safety, the ASIL-C certified version of the board (NVIDIA DRIVE Xavier) includes many resilience features, such as fault detection or redundancy [48], which makes this device suitable for the specification of the project (see Fig. 4.1).

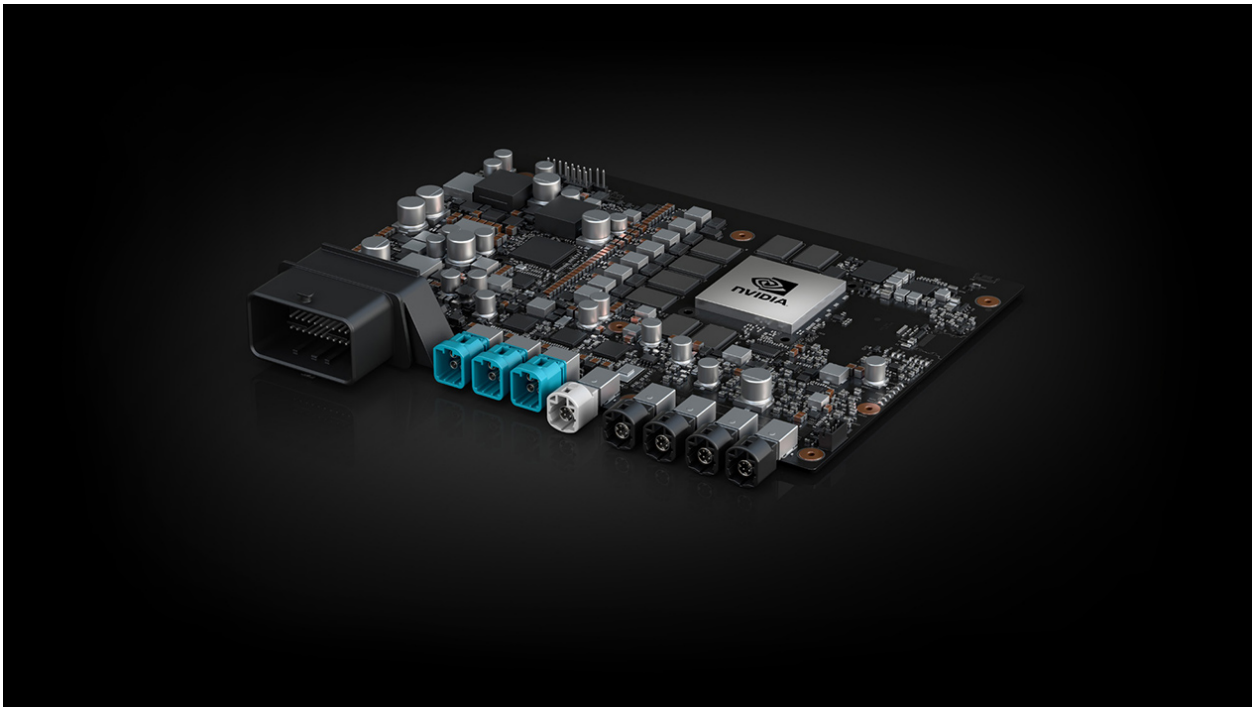



Figure 4.1: NVIDIA DRIVE AGX Xavier. Image courtesy of NVIDIA.

Another great advantage of the NVIDIA Xavier is its power configurability, similar to the Jetson TX2, we can find multiple power modes already pre-configured that will guarantee a series of properties on the board, as we saw in table 2.1.

The board features seven performance modes, ranging from different numbers of computing resources and frequencies, for which Nvidia ensures certain TDP.

To perform the benchmarking evaluation we have selected the **power mode 2**, which uses four of the eight Carmel cores present in the board, and the Volta GPU running at a reduced frequency. It is relevant to note that the power mode selected in the AGX Xavier is equivalent to that selected in the Jetson TX2.

### 4.1.3 Xilinx Zynq Ultrascale+ ZCU102

This heterogeneous device is a many-core platform containing both, a FPGA and a non-compute graphics card. As mentioned in Section 2.1.2.2, the device comes with four ARM Cortex-A53 CPUs, two ARM Cortex-R5F real-time processors, and a Mali 400 MP2 GPU (see Fig. 4.2). As for the chosen configuration, we assume that the cores are operating at their maximum frequencies of 1.5GHz for the ARM Cortex-A53 cores and 600MHz for the R5 cores as outlined in the Xilinx manual. Note that we are only benchmarking the multi-core capabilities of the platform, since the FPGA requires different programming models which require effort beyond the scope of the UP2DATE project. Similarly, the graphics-only capable GPU is not benchmarked either, since this would require porting the GPU4S Benchmarking suite to another programming model, too, Brook Auto. These activities are left as a future work, and should be taken into account when comparing the computational capabilities of the full platform.

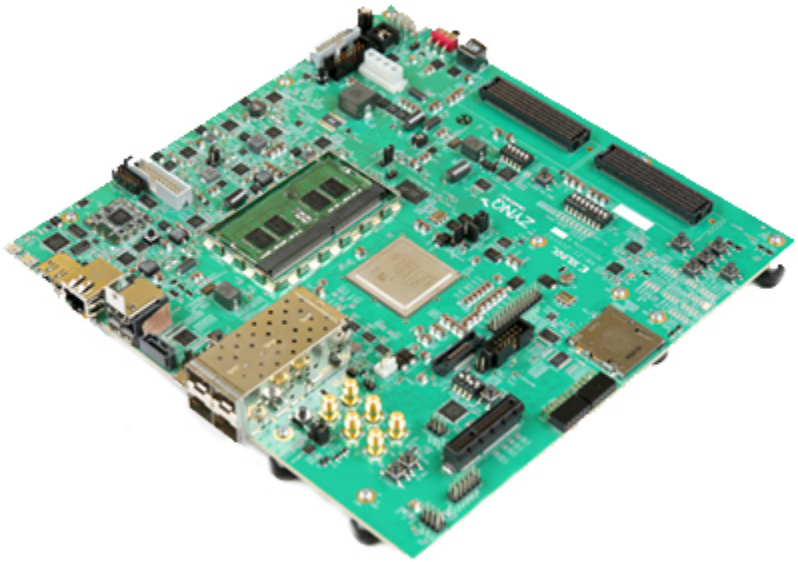


Figure 4.2: Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. Image courtesy of Xilinx.

Nevertheless, we were not able to configure the board to run multicore experiments, as the experiments were performed during the lock-down period working remotely. This limited the possibility of running experiments only to bare metal configurations, since booting in a linux environment (Petalinux), requires physical access in order to boot the device from an SDcard. For this reason, we perform the comparison of the Xilinx board with the rest of the platforms only in single core mode.

The configurability of this device isn't equivalent to that found in the two previous NVIDIA platforms, and due to the resource requirements we had to run the experiments using the default configuration of the board.

To perform the measurements, we accessed the low-level performance counters using a combination of C and assembly. For each benchmark, we collect the number of instructions executed, as well as the number of cycles. We then use the nominal frequency of each processor to convert these measurements into real elapsed time.

# Chapter 5

# Experimentation

In this Chapter we report and analyse the results of the benchmarking experiments we performed for the selection of the UP2DATE research platform. For that, we divide the experiments in four different sections: Section 5.1 reports the single core speed-up results; Section 5.2 elaborates on the multi-core benchmarking; next, Section 5.3 exposes the GPU results for both NVIDIA platforms; and finally, Section 5.4 wraps this study comparing the relative performance of the GPU over the CPU for all the heterogeneous devices.

## 5.1 Single Core Performance comparison

Figure 5.1 compares the single core performance of the NVIDIA platforms against the performance of the ARM-A53 CPU of the Zynq Ultrascale+ ZCU102. We omit the results of the real-time R5 cores of the Zynq platform, due to their low-performance nature compared to the other high performance processing elements. Note that since in Zynq we are running in bare metal, we cannot use the library versions for the Matrix Multiplication and the Fast Fourier Transform. For this reason, all the reported results are with a sequential, handwritten implementation which is identical for all the platforms. On the TX2, we are running the benchmarks on both of its cores, the Denver and the ARM A57. In general we see two trends: First, NVIDIA's custom designed cores perform better than the stock ARM CPUs. Second, the NVIDIA platforms are faster than the Xilinx one. Specifically, we see that Xavier's Carmel CPU is faster than the Zynq core in all benchmarks, ranging from 2.5x to 9x.

Moreover, if we compare the two NVIDIA platforms, we notice that the Carmel CPU of the Xavier is faster than the ARM-57 of the TX2 in all the benchmarks. The same happens for the Denver cores of the TX2, however it has similar performance with the Carmel cores in several benchmarks. We believe that this similarity is due to the fat that both cores were designed by NVIDIA, and the Carmel design is an evolution of Denver's.

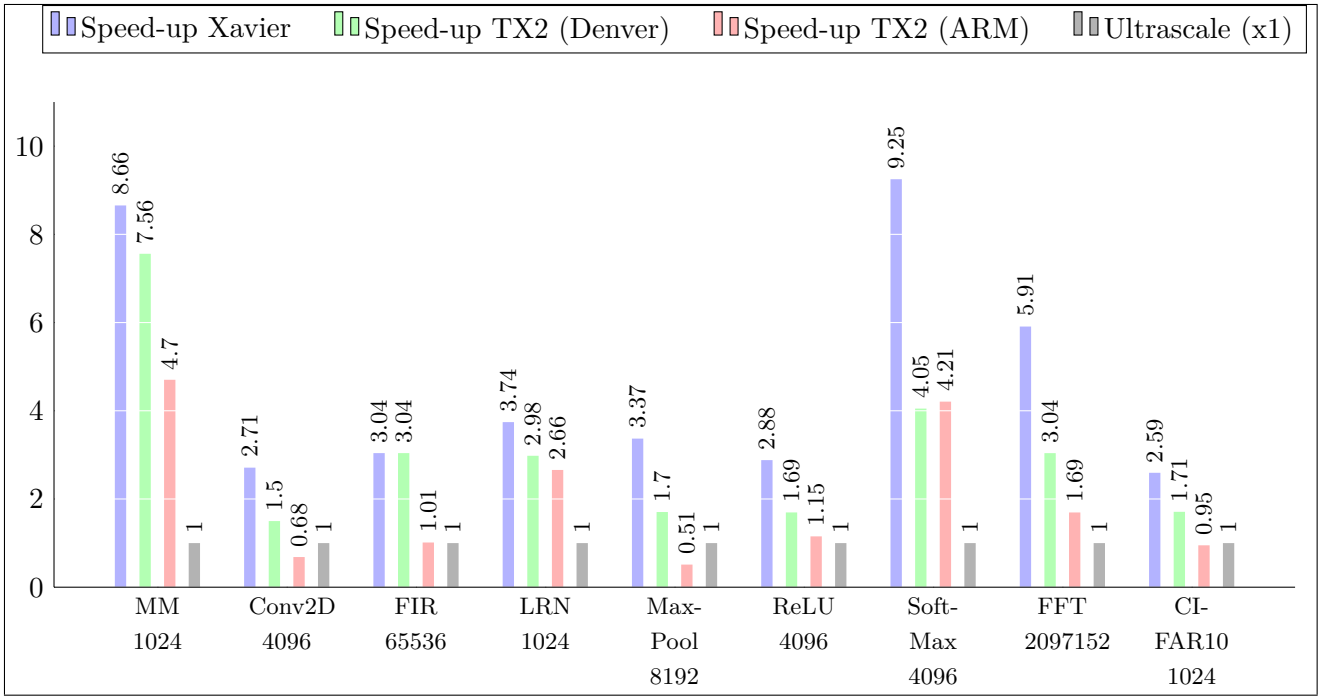



Figure 5.1: Single core performance comparison between the four CPUs found in the three candidate platforms, relative to the performance of Zynq Ultrascale+.

Regarding the stock ARM CPU designs, the newer ARM A57 of the TX2, which is based on an out-of-order microarchitecture performs significantly better in some benchmarks, however there are also several ones in which it is outperformed by the in-order A53 of the Zynq, but in a smaller extent.

## 5.2 Multi-Core Performance comparison

In the following, we compare the multi-core performance between the NVIDIA AGX Xavier and the NVIDIA TX2. For this, we will use the OpenMP version of our benchmarks and libraries where available.

Figure 5.2 shows the relative multicore performance of the two NVIDIA platforms, using the TX2 as a baseline. We used the same benchmarks with the same input sizes as in Section 5.1, but in the cases where the execution time wasn't high enough for a proper analysis we have evaluated an additional larger input set (Matrix Multiplication and Convolution 2D).

The general trend is that the CPU efficiency of the NVIDIA Xavier in the 15W power mode is significantly higher than that of the NVIDIA TX2 board using the same power mode in about half of the cases, while in the remaining algorithms the performance is slightly slower than that of the TX2.

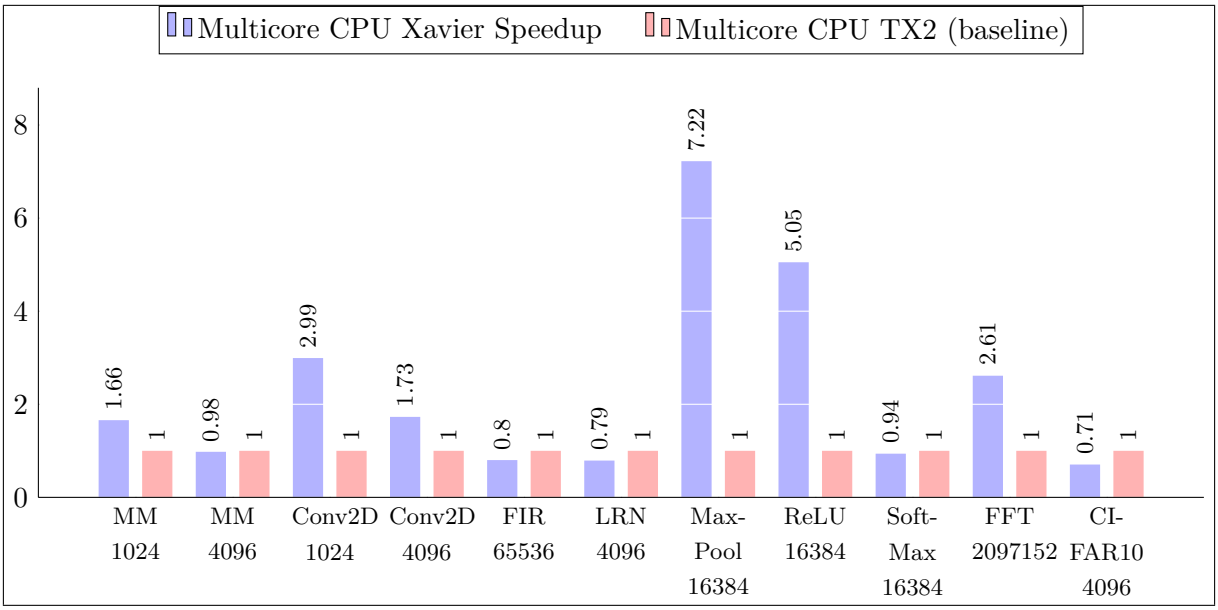



Figure 5.2: Relative Multicore performance comparison between of Xavier and TX2.

We also observe two cases (Matrix Multiplication and Convolution 2D) where increasing the benchmark input size reduced the performance advantage of the Xavier platform over the TX2.

## 5.3 GPU Performance comparison

Next we compare the GPU performance capabilities of the TX2 and Xavier. Figure 5.3 shows the relative performance of the Xavier's GPU against the TX2 under the same power budget.

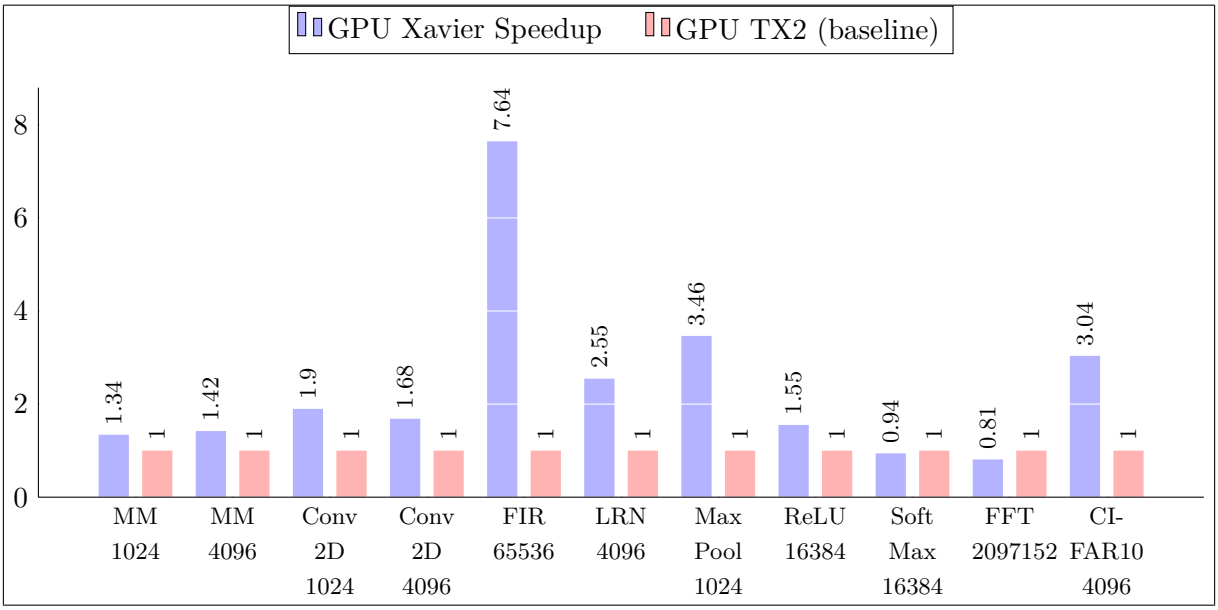



Figure 5.3: Relative GPU performance of Xavier over the TX2.

In terms of GPU performance the results are clearly in favour of the Xavier, since in all but two cases, the performance of the NVIDIA AGX Xavier is higher, and only in two cases it is slightly slower than the NVIDIA TX2.

By looking at Figure 5.2 and 5.3, we can conclude that the NVIDIA AGX Xavier is more energy efficient than the TX2. In addition, the AGX Xavier features 8 cores and supports higher performance modes running at 30 Watts, as we have seen in Table 2.1. Therefore, it is safe to assume that the device can provide higher throughput to support modern demanding applications.

## 5.4 CPU to GPU comparison

One of the strongest requirements for our candidate platforms was heterogeneity, since industry employs such solutions for the implementation of applications which require very high performance, such as autonomous driving. For this reason, in this Section we validate this hypothesis by comparing the relative performance of the GPU over the CPU of each heterogeneous platform evaluated in the project, the NVIDIA AGX Xavier and the NVIDIA TX2.

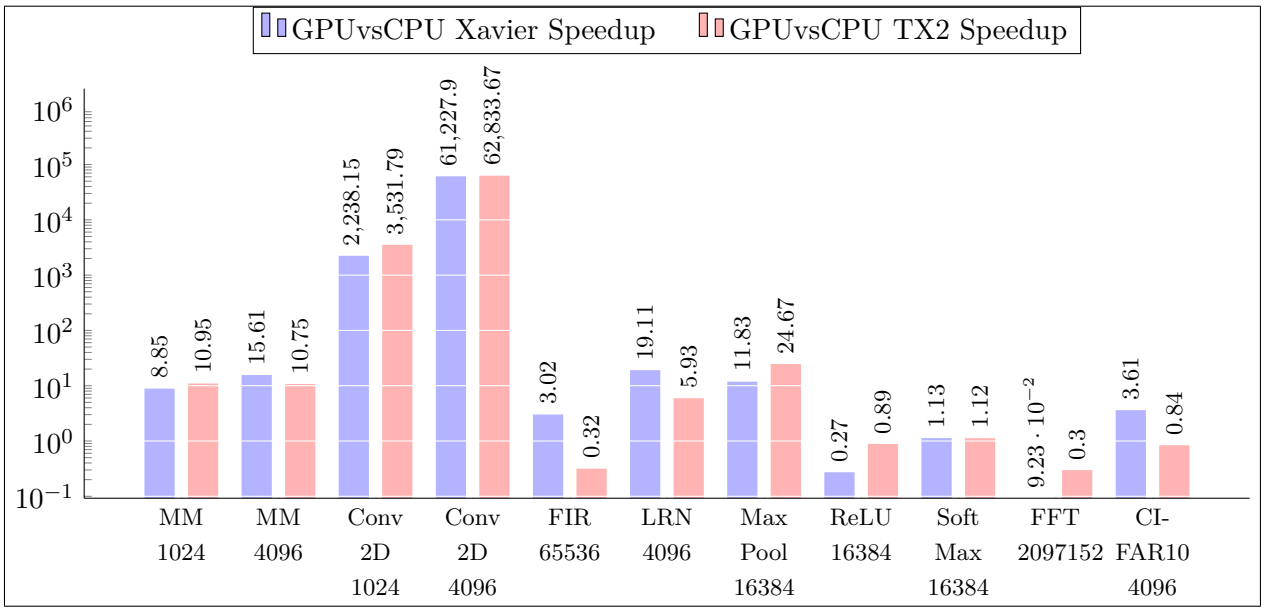



Figure 5.4: Relative GPU performance over the CPU in the same SoC.

Figure 5.4 shows the results of the relative performance of the GPU compared to the 4 core multicore CPU performance obtained using OpenMP of each of the considered NVIDIA platforms. The results of both platforms follow the same trend. Generally, the GPU performs better than the CPU in most of the algorithms. Specifically, there are benchmarks in which the GPU is several orders of magnitude faster than the CPU, most notably in the case of the 2D Convolution. Benchmarks that benefit from the GPU have high arithmetic complexity, which means that for each piece of data fetched from memory, multiple arithmetic operations are performed.

However, in the case of the FFT, the CPU library implementation (FFTW) is highly optimised with respect to the CPU memory hierarchy, yielding a 10x performance advantage versus the GPU. Nevertheless, we should mention that to gain performance benefits from this same algorithm on the GPU we must consider scenarios where FFTs are performed repeatedly or over multiple signals at the same time.

Other benchmarks with low arithmetic intensity, such as Relu, may not benefit from the GPU, due to the low arithmetic nature of the algorithm. However, other kernels also used in neural network, such as Matrix Multiplication or Max-Pooling obtain speedups ranging from 10% to 25x when performed on the GPU.

It is therefore clear that the use of the GPU is a major performance enhancer even on an embedded platform with a power consumption of 15W. Thus confirming our decision to invest in heterogeneous architectures.

# Chapter 6

# Conclusions and Future Work

After reviewing the results from benchmarking the different candidate platforms, we can conclude that the NVIDIA Xavier board is overall the most powerful platform we tested for several factors:

- **Power efficiency**: The NVIDIA Xavier provides more performance than the NVIDIA TX2 running at the same power budget.
- **Higher degree of configurability**: The NVIDIA Xavier supports vendor-verified power modes configured by NVIDIA, and can provide higher power modes (30 W) due to its higher TDP.
- **Higher degree of multi-core parallelism**: The NVIDIA Xavier provides 8 homogeneous cores, while the NVIDIA TX2 provides 6.
- **Main memory**: NVIDIA Xavier has a larger memory (32GB) then the NVIDIA TX2 (8GB).

For these reasons, it is safe to assume that the NVIDIA Xavier can deliver much higher overall performance compared with the rest of the platforms, for the type of workloads we consider to use in the project, since we are not planning to develop hardware accelerators in the FPGA in UP2DATE. Therefore, it will be the prioritised GPU research platform for the UP2DATE project (Chapter 3), followed by the Xilinx Zynq Ultrascale+ ZCU102 as our FPGA candidate, in case that some already implemented accelerators are deployed on it.

In regard to the possible future research lines that we will develop to complement this work, we plan to port more benchmarks, such as the On-Board Processing Benchmarks (OBPMark) benchmarking suite [85], with AI, compression and encryption algorithms representative of current and future space software and other critical domains. In particular, OBPMark is another open source benchmarking suite, again defined and implemented by ESA and BSC, which has been created as a response to the lack of an open, common benchmarking infrastructure for devices used in space [85]. Unlike GPU4S Bench which mainly consists of algorithmic building blocks

which allow covering multiple domains, OBPMark contains more complex applications, which are reusing the GPU4S Bench algorithmic building block implementations in order to provide optimised implementations for multiple devices, such as GPUs and multi-core processors thanks to the work of this Thesis.

Moreover, we plan to add additional parallel processing methodologies and models such as Brook Auto/BRASIL which will allow us to benchmark the Mali-400 GPU of the Xilinx platform and programming models targeting FPGAs such as hardware description languages like Verilog, VHDL and high-level synthesis solutions like OpenCL and OmpSs@FPGA.

# Chapter 7

# Publications

The main contributions of this Thesis have been published and presented in the following peer-reviewed conference and workshop publications:

**Alvaro Jover-Alvarez**, Alejandro J. Calderón, Iván Rodriguez, Leonidas Kosmidis, Kazi Asifuzzaman, Patrick Uven, Kim Grüttner, Tomaso Poggi, Irune Agirre. *The UP2DATE Baseline Research Platforms*, Design, Automation and Test in Europe Conference and Exhibition (DATE) 2021 [53]

Leonidas Kosmidis, Ivan Rodriguez Ferrandez, **Alvaro Jover-Alvarez**, Sergi Alcaide, Jérôme Lachaize, Olivier Notebaert, Antoine Certain, David Steenari. *GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward* Design, Automation and Test in Europe Conference and Exhibition (DATE) 2021 [59]

Leonidas Kosmidis, Iván Rodriguez-Ferrandez, **Alvaro Jover-Alvarez**, Sergi Alcaide, Jérôme Lachaize, Olivier Notebaert, Antoine Certain and David Steenari. *GPU4S (GPUs for Space): Are we there yet?*, ESA/CNES/DLR European Workshop on On-Board Data Processing (OBDP) 2021 [60].

David Steenari, Leonidas Kosmidis, Ivan Rodriguez-Ferrandez, **Alvaro Jover-Alvarez** and Kyra Förster. *OBPMark (On-Board Processing Benchmarks) – Open Source Computational Performance Benchmarks for Space Applications*, ESA/CNES/DLR European Workshop on On-Board Data Processing (OBDP) 2021 [85].

In addition to the main contributions, the work of this thesis was also presented in the following poster sessions:

- **Alvaro Jover-Alvarez** and Leonidas Kosmidis. *Evaluation of the Computational Capabilities of High-Performance Embedded Platforms for Safety Critical Systems.* International Summer School on Advanced Computer Architecture and Compilation for High-performance Embedded Systems (ACACES), Virtual Poster Session July 2020.

- **Alvaro Jover-Alvarez** and Leonidas Kosmidis. *Evaluation of the Computational Capabilities of High-Performance Heterogeneous Embedded Platforms for Safety Critical Systems.* ACM Student Research Competition at the International Conference in Computer Aided Design (ICCAD) 2020, Graduate Category

Moreover, we are currently working towards a survey paper based on the contents of the State-of-the-Art Section (Section 2).

Finally, the source code contributions of the thesis have been released as part of the open source benchmarking suites GPU4S and OBPMark, which are co-hosted at [84] [83].

# Bibliography


[1] IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)). *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages 1–3874, 2008.

[2] *The OpenACC Application Programming Interface. Version 3.1.* OpenACC-Standard.org, 2020.

[3] *UP2DATE: Safe and secure over-the-air software updates on high-performance mixed-criticality systems*. Zenodo, August 2020. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in DSD 2020. https://doi.org/10.1109/DSD51259.2020.00063.

[4] AdaCore. SPARK PRO.

[5] The European Space Agency. Chang'e-4 lander.

[6] The European Space Agency. Gr740: The esa next generation microprocessor (ngmp).

[7] The European Space Agency. Microprocessors, leon2 / leon2-ft.

[8] The European Space Agency. Microprocessors, LEON4 / LEON4-FT.

[9] Irune Agirre, Jaume Abella, Mikel Azkarate-Askasua, and Francisco J. Cazorla. On the tailoring of CAST-32A certification guidance to real COTS multicore architectures. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8, 2017.

[10] AMD. AMD Ryzen Embedded serie V1000.

[11] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[12] Jan Andersson, Jiri Gaisler, and Roland Weigand. Next Generation MultiPurpose Microprocessor. In L. Ouwehand, editor, *DASIA 2010 - Data Systems In Aerospace*, volume 682 of *ESA Special Publication*, page 8, August 2010.

[13] The Multicore Association. Mtapi reference card.

[14] Ada Conformity Assessment Authority. Ada Reference Manual, 2016.

[15] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Ernesto Su, Priya Unnikrishnan, and Guansong Zhang. A Proposal for Task Parallelism in OpenMP. In Barbara Chapman, Weiming Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era*, pages 1–12, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[16] Marc Benito, Matina Maria Trompouki, Leonidas Kosmidis, Juan David Garcia, Sergio Carretero, and Ken Wenger. Comparison of GPU Computing Methodologies for Safety-Critical Systems: An Avionics Case Study. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021.

[17] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim Guldstrand Larsen. Contracts for Systems Design: Methodology and Application cases. Research Report RR-8760, Inria Rennes Bretagne Atlantique ; INRIA, July 2015.

[18] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[19] Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007.

[20] Francisco J. Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. Probabilistic Worst-Case Timing Analysis: Taxonomy and Comprehensive Survey. *ACM Comput. Surv.*, 52(1), February 2019.

[21] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-Core Devices for Safety-Critical Systems: A Survey. *ACM Comput. Surv.*, 53(4), August 2020.

[22] Certification Authorities Software Team (CAST). Multi-core Processors, November 2016. `https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/media/cast-32A.pdf`.

[23] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008. General-Purpose Processing using Graphics Processing Units.

[24] Peter Clarke. European Space Agency launches free Sparc-like core. 2000.

[25] European Commission. ICT-01-2019, Computing technologies and engineering methods for cyber-physical systems of systems.

[26] CoreAVI. Vulkan SC Graphics and Compute.

[27] Fabrice Cros, Leonidas Kosmidis, Franck Wartel, David Morales, Jaume Abella, Ian Broster, and Francisco J. Cazorla. Dynamic Software Randomisation: Lessons Learned From an Aerospace Case Study. In *DATE*, 2017.

[28] Space Daily. Leon: the space chip that europe built.

[29] Michael Ditty, Ashish Karandikar, and David Reed. Nvidia's Xavier SoC. In *Hot chips: a symposium on high performance chips*, 2018.

[30] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012. APPLICATION ACCELERATORS IN HPC.

[31] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES. *Parallel Processing Letters*, 21(02):173–193, 2011.

[32] Linux Foundation and Wind River Systems. Zephyr.

[33] Aeroflex Gaisler. Leon3, multiprocessing cpu core.

[34] Cobham Gaisler. Gr740 quad-core leon4 sparc v8 processor.

[35] Cobham Gaisler. GR740 Qualification Results.

[36] Cobham Gaisler. Leon4 processor.

[37] Cobham Gaisler. LEON5 Processor.

[38] Cobham Gaisler. Noel-v processor.

[39] Jiri Gaisler. Preparations for next-generation SPARC processor. 01 2003.

[40] Jiri Gaisler. Fault-tolerant and radiation-hardened SPARC processors. 2007.

[41] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 3–3, 2005.

[42] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[43] Urs Gleim and Markus Levy. MTAPI: Parallel Programming for Embedded Multicore Systems.

[44] Khronos Group. Khronos OpenCL Registry.

[45] Khronos Group. OpenGL ES Overview.

[46] Khronos Group. Vulkan SC Overview.

[47] John L. Gustafson. Reevaluating Amdahl's Law.

[48] Gary Hicok. NVIDIA Xavier Achieves Industry First with Expert Safety Assessment.

[49] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 136–143, 2013.

[50] infineon. 32-bit AURIX TriCore Microcontroller.

[51] infineon. AURIX™ TC3xx User Manual Part-1.

[52] IWOCL and SYCLcon. OpenCL Libraries and Toolkits, 2021.

[53] Alvaro Jover-Alvarez, Alejandro J. Calderon, Ivan Rodriguez, Leonidas Kosmidis, Kazi Asifuzzaman, Patrick Uven, Kim Grüttner, Tomaso Poggi, and Irune Agirre. The UP2DATE Baseline Research Platforms. In *Proceedings of the Design, Automation & Test in Europe (DATE)*, 02 2021.

[54] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 Updates and Changes. Technical Report LLNL-TR-641973, August 2013.

[55] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach*, page 4. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.

[56] Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, Franck Wartel, Glenn Farrall, and Francisco J. Cazorla. Containing Timing-Related Certification Cost in Automotive Systems Deploying Complex Hardware. In *DAC. (Best Paper Award)*, 2014.

[57] Leonidas Kosmidis, Alvaro Jover-Alvarez, Kazi Asifuzzaman, Francisco J. Cazorla, Kim Gruettner, Patrick Uven, Tomaso Poggi, Jan Loewe, and Prajakta Garibdas. D2.2 Baseline Definition, 2020.

[58] Leonidas Kosmidis, Jérôme Lachaize, Jaume Abella, Olivier Notebaert, Francisco J Cazorla, and David Steenari. Gpu4s: Embedded gpus in space. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 399–405. IEEE, 2019.

[59] Leonidas Kosmidis, Iván Rodriguez, Álvaro Jover, Sergi Alcaide, Jérôme Lachaize, Olivier Notebaert, Antoine Certain, and David Steenari. GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021.

[60] Leonidas Kosmidis, Iván Rodriguez-Ferrandez, Alvaro Jover-Alvarez, Sergi Alcaide, Jérôme Lachaize, Olivier Notebaert, Antoine Certain, and David Steenari. GPU4S (GPUs for Space): Are we there yet? *On-Board Data Processing 2021*, Juny 2021.

[61] Leonidas Kosmidis, Roberto Vargas, David Morales, Eduardo Quiñones, Jaume Abella, and Francisco J. Cazorla. TASA: Toolchain Agnostic Software Randomisation for Critical Real-Time Systems. In *ICCAD*, 2016.

[62] E. Scott Larsen and David McAllister. Fast Matrix Multiplies Using Graphics Hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, page 55, New York, NY, USA, 2001. Association for Computing Machinery.

[63] leox.org. FAQ: Collection of frequently asked questions from the leon_sparc mailing list.

[64] Kelvin Li. OpenMP Accelerator Support for GPUs.

[65] Real Time Engineers Ltd. FreeRTOS.

[66] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015.

[67] John W. McCormick, Frank Singhoff, and Jerome Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, USA, 2011.

[68] Alessandra Melani, Maria A. Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quiñones, and Giorgio Buttazzo. A static scheduling approach to enable safety-critical OpenMP applications. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 659–665, 2017.

[69] Adam Moravanszky. Dense Matrix Algebra on the GPU. 2003.

[70] Motor Industry Software Reliability Association . *MISRA C++:2008 Guidelines for the use of the C++ language in critical systems*. 2008.

[71] Motor Industry Software Reliability Association. *MISRA-C:2012. Guidelines for the Use of the C Language in Critical Systems*. 2013.

[72] Aditya Nitsure, Himanshu Shrivastava, and Pidad Dsouza. GPU programming made easy with OpenMP on IBM POWER.

[73] NVIDA. TECHNICAL REFERENCE MANUAL NVIDIA Xavier Series System-on-Chip, 2020.

[74] NVIDIA. CUDA Runtime API, 2021.

[75] NVIDIA. Jetson AGX Xavier Power Rails, 2021.

[76] OpenACC-standard.org. OpenACC.

[77] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical Task-Based Programming With StarSs. *The International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[78] Mikael Ramström, Jonas Höglund, Björn Enoksson, Ritva Svenningsson, Mats Steinert, and Torbjörn Hult. FINAL REPORT, May 1997.

[79] Iván Rodriguez, Leonidas Kosmidis, Jérôme Lachaize, Olivier Notebaert, and David Steenari. Gpu4s bench: Design and implementation of an open gpu benchmarking suite for space on-board processing. Technical Report UPC-DAC-RR-CAP-2019-1, Universitat Politecnica de Catalunya. `https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html`.

[80] Iván Rodriguez, Leonidas Kosmidis, Olivier Notebaert, Francisco J. Cazorla, and David Steenari. An On-board Algorithm Implementation on an Embedded GPU: A Space Case Study. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1718–1719, 2020.

[81] Sara Royuela, Alejandro Duran, Maria A. Serrano, Eduardo Quiñones, and Xavier Martorell. A Functional Safety OpenMP* for Critical Real-Time Embedded Systems. In Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller, editors, *Scaling OpenMP for Exascale Performance and Portability*, pages 231–245, Cham, 2017. Springer International Publishing.

[82] Tobias Schuele. Embedded Multicore Building Blocks: Parallel Programming Made Easy. *Embedded World*, 2015.

[83] David Steenari, Leonidas Kosmidis, Alvaro Jover-Alvarez, and Ivan Rodriguez-Ferrandez. OBPMark (On-Board Processing Benchmarks) GitHub, 2021. `https://github.com/OBPMark`.

[84] David Steenari, Leonidas Kosmidis, Alvaro Jover-Alvarez, and Ivan Rodriguez-Ferrandez. OBPMark (On-Board Processing Benchmarks) Website, 2021. `https://obpmark.github.io/`.

[85] David Steenari, Leonidas Kosmidis, Ivan Rodriguez-Ferrandez, Alvaro Jover-Alvarez, and Kyra Förster. OBPMark (On-Board Processing Benchmarks) – Open Source Computational Performance Benchmarks for Space Applications. *On-Board Data Processing 2021*, Juny 2021.

[86] Matina Maria Trompouki and Leonidas Kosmidis. Towards general purpose computations on low-end mobile GPUs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 539–542, 2016.

[87] Matina Maria Trompouki and Leonidas Kosmidis. Brook Auto: High-Level Certification-Friendly Programming for GPU-powered Automotive Systems. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[88] Matina Maria Trompouki and Leonidas Kosmidis. Brook GLES Pi: Democratising Accelerator Programming. In *Proceedings of the Conference on High-Performance Graphics*, HPG '18, New York, NY, USA, 2018. Association for Computing Machinery.

[89] Matina Maria Trompouki and Leonidas Kosmidis. BRASIL: A High-Integrity GPGPU Toolchain for Automotive Systems. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 660–663, 2019.

[90] Matina Maria Trompouki and Leonidas Kosmidis. DO-178C Certification of General-Purpose GPU Software: Review of Existing Methods and Future Directions. In *Digital Avionics Systems Conference*, 2021.

[91] W. Wang. POSIX threads programming. 2005.

[92] WikiChip. Ryzen Embedded V1605B - AMD.

[93] WikiChip. NVIDIA Xavier Block diagram, 2021.

[94] Xilinx. Zynq Power Management Framework User Guide For Zynq UltraScale+ MPSoC Devices.

[95] Xilinx. Zynq UltraScale+ MPSoC.

[96] Xilinx. Zynq UltraScale+ MPSoC Base Targeted Reference Design.