GPU Devices for Safety-Critical Systems: A Survey

Jon Perez-Cerrolaza†, Jaume Abella‡, Leonidas Kosmidis‡, Alejandro J. Calderon†, Francisco J. Cazorla‡, Jose Luis Flores†

†Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), Spain
‡Barcelona Supercomputing Center (BSC), Spain

Abstract

Graphics Processing Unit (GPU) devices and their associated software programming languages and frameworks can deliver the computing performance required to facilitate the development of next-generation high-performance safety-critical systems such as autonomous driving systems. However, the integration of complex, parallel and computationally demanding software functions with different safety-criticality levels on GPU devices with shared hardware resources contributes to several safety certification challenges. This survey categorizes and provides an overview of research contributions that address GPU devices' random hardware failures, systematic failures and independence of execution.

1 Introduction

High-performance computing devices such as GPUs and multi-core devices are increasingly considered for the development of safety-critical systems in multiple domains such as transportation (e.g., automotive [15, 203, 141, 13], railway [20], avionics [21, 27], space [105, 196, 9]) and industrial machinery (e.g., wind turbines, industrial control robots) [153, 11]. This trend answers the need for higher computational performance and higher on-chip integration of functions required to facilitate the development of next-generation safety-critical systems. For example, the automotive domain Autonomous Driving (AD) and Advanced Driver-Assistance Systems (ADAS) require unprecedented levels of computing performance to process computationally-demanding algorithms such as computer vision perception and Machine Learning (ML) algorithms [121, 141]. Moreover, there is also a cross-domain trend towards the integration of different safety criticality functions (a.k.a., mixed-criticality) in a reduced number of high-performance computing devices reducing hardware's solution size, weight, and power costs and potentially increasing the overall system reliability by a reduction of components, cables and connectors [153, 11].

A GPU, or graphics processing unit, is a computing device specialized in image processing acceleration and output display. However, as the provided high computing capacity enables the acceleration of computationally demanding algorithms and ML-based software, their usage and device specialization has evolved in the last decades from initial home applications (e.g., GPU for gaming) to new application domains such as safety-critical autonomous systems [196, 140]. Several technical challenges have been addressed as part of this evolution, such as reliability improvements and safety compliance [4, 15, 13, 140]. Also, during the last decade, the innovative nature of the semiconductor sector has developed GPU devices with steadily shrinking technologies down to the current 7 nm while continuously increasing the overall computing capacity.

Safety is defined as the "absence of catastrophic consequences on the user(s) and the environment" [22]. Safety-critical systems are programmable electronics systems whose failure can lead to catastrophic consequences, like environmental damage or human casualties (e.g., autonomous driving system accident). Thus, in several domains, strict certification processes must be followed for the development and certification of such systems. This implies high development costs where, as a rule of thumb, "the higher the safety integrity level, the higher the cost of safety certification" [11]. For example, the highest criticality industrial systems [92] must ensure an extremely low failure probability of up to 10^{-9} per hour of operation (approximately 114.155 years). This imposes strict technical requirements to mitigate to such low levels the probability of failure due to random hardware errors (e.g., memory

© 2022 Copyright held by the owner/author(s). 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 *ACM Computing Surveys*, vol. 55, issue 7, July 2023, article 147, pp 1–37.
http://dx.doi.org/10.1145/3549526

bit flip) and systematic errors (e.g., human, process and tool errors), and to achieve independence of execution among integrated functions. In this context, the development and certification of safety-critical systems based on Commercial Off-The-Shelf (COTS) GPU devices designed to maximize the average performance is a challenge [153, 11, 15, 141, 13].

This work provides a survey of GPU devices for the development of safety-critical systems with a categorization of selected research contributions that address GPU devices' random hardware failures (§4), systematic failures (§5), and independence of execution (§6). And Table 1 summarizes the threats, safety measures and techniques described in the corresponding Sections (§4, §5, §6). This survey targets both researchers and safety systems developers, providing a comprehensive view the state-of-the-art required to carry out a diligent development of safety-critical systems, whenever applicable safety standards have none or limited consideration of GPU devices [61, 153]. Regarding safety standards, both IEC 61508 [92] and ISO 26262 [95] are considered the reference functional safety standards in this survey.

The remainder of this survey is organized as follows. Section 2 describes basic concepts and terminology. Section 3 describes GPU device-specific features and challenges, along with the classification criteria and metrics used in this survey. Based on this, Sections 4, 5 and 6 categorize and provide an overview of research contributions that address GPU devices' random hardware failures (§4), systematic failures (§5), and independence of execution (§6). Section 7 summarizes the similarities and particular characteristics of other domains that do not consider IEC 61508 as a reference safety standard (avionics and space). Section 8 discusses links to related research topics and future trends to be considered in the development of GPU device-based safety-critical systems. Finally, Section 9 describes the overall conclusion and future research directions.

2 Background

This section summarizes basic concepts and terms used within the survey, such as GPU terminology, safety certification standards and fundamental safety technical requirements. The survey uses the NVIDIA CUDA terminology for GPUs, the dependable computing terminology defined by Avižienis et al. [22] and the functional safety terminology defined by IEC 61508-4 and ISO 26262-1 safety standards [92, 95]. We also integrate the terminology of several research fields described by referenced survey and review publications (e.g., Fault Injection (FI) techniques [28]).

2.1 Graphics Processing Unit (GPU)

A GPU device is a high-performance many-core processor with associated industry-standard software programming languages and Application Programming Interfaces (APIs) [141, 196]. Using NVIDIA CUDA terminology for convenience within the survey (see Table 2 for OpenCL terminology used by other semiconductor companies such as AMD), the GPU main processing units are the Streaming Multiprocessors (SMs). Each SM comprises several scalar cores and other resources such as registers, shared memories and schedulers. A scalar core is a pipelined Arithmetic Logic Unit (ALU) capable of executing integer and floating-point operations. The SMs execution management is performed by the host, composed of one or more CPU core(s).

Functions executed on the SMs are known as *kernels*. Before launching a kernel to the SMs, the user prepares the data to be processed using host memory, reserves the corresponding amount of GPU memory and triggers a copy operation from host to GPU memory. Once the SM kernel execution finishes the result is copied back to the host memory from the GPU memory. In *embedded GPU* devices, since global memory is shared between CPU cores and SMs, this may be done more efficiently by defining a single region of shared memory to avoid the memory copy operations. For that purpose, it is possible to use OpenCL Shared Virtual Memory (SVM) and CUDA Unified Virtual Memory (UVM) features, yet taking into consideration the need to manage memory consistency.

When launching a kernel to the SM, the user specifies a *grid* configuration with the number of *blocks* and threads per block to be executed. At the hardware level, the threads of a block are partitioned into *warps*, groups of 32 lockstep threads. The warp scheduler(s) of each SM fetch, decode and issue instructions to the execution pipelines. During the execution, resources like shared memory and registers are shared by the warps being scheduled.

GPU software programming is commonly based on standardized APIs such as Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL), Open Graphics Library (OpenGL) variants, Vulkan, Open Multi-Processing (OpenMP) [56] and Open Accelerators (OpenACC). Some of them are primarily built for graphics, such as OpenGL variants, while others for general-purpose computations (e.g., CUDA, OpenCL, OpenMP, OpenACC). Some of them support both graphics and compute, such as Vulkan. From those, OpenGL SC (Safety-Critical), is the only one of these APIs designed to comply with safety-critical systems' requirements [27].

Table 1: Categorization of the related works

	Example Threats	Safety Measures and Techniques	Sect.
Random HW failures	· PVT variability [143] · Device aging [143] · Transient faults (soft errors) [143] · Permanent faults [143] · Shared resources error propagation [195, 138, 140, 170, 51]	· Application-agnostic: - Detection with redundancy without diversity [163, 6, 191, 178, 199, 57, 73, 138, 99, 164, 34, 170, 120, 140, 77, 127, 139] and with diversity [13, 14, 8, 12] - Detection and/or correction with coding (e.g., ECC) and checkers [57, 120, 169, 139, 108, 124, 125, 140, 132] - Recovery with re-execution or checkpoints [71, 175, 180, 135, 115, 124] - Mitigation with shielding and reconfiguration [153, 127, 91, 193] · Application-dependent: - Detection and fault-tolerance algorithmic approaches [35, 138, 169, 170, 168, 195, 140, 49, 50, 110, 139, 156, 102, 202, 80, 204, 58, 76] - Fault-tolerance based on intrinsic application and/or input data characteristics [63, 78, 159, 158]	§4
Systematic failures	· Systematic faults (e.g., human, tools, process) · Software programming languages and frameworks not-compliant with safety standards [183, 15, 92, 95]	· Application-agnostic: - Software: Safety standard compliant programming languages and frameworks (e.g., OpenGL SC [23, 27]), 'safe language subsets' (e.g., Brook auto [183]) and tool qualification [184, 27] - Device hardware: Safety standard ASIC development requirements [92, 3] and development techniques for mass-produced devices (e.g., [103, 149])	§5
Indepen. of Execution	Temporal interference and spatial interference	Temporal independence and spatial independence	§6
· **Temporal Independence**	· Kernel execution time variability [141, 142, 46] · Host application(s) interactions with driver stack [141, 142, 16, 17, 59] · Memory contention [141, 142, 46, 96, 100] · Shared resources (e.g., shared memory) · Lack of information [46, 142, 16, 14], documentation inconsistencies [200, 144, 14] and partial understandability of temporal behavior [142, 100, 40, 144, 96, 18, 39]	· Application-agnostic: - Device hardware architecture and component-specific (e.g., memory [70, 43, 68, 101, 109, 104], cache [69, 87, 154, 96, 97, 194]) - Exclusive access policies [96, 97, 194] - Software virtualization [84, 114, 141, 41, 174, 112] - WCET analysis [48, 29, 30, 85, 33, 16, 90, 89, 88] - Execution time variability reduction and management [18, 41, 42, 67, 82, 84, 88, 142, 144, 200, 101, 70, 68, 109, 43, 141, 147, 47, 81, 24] - Temporal diagnostics [25, 153] · Application-dependent [176, 62]	§6.1
· **Spatial Independence**	· Shared resources [100, 46] · Cache/Memory coherency [153] · Parallel execution of kernels [200, 136] · Software memory transfer mechanisms [46] · Software frameworks and middlewares [96, 46]	· Application-agnostic: - Device hardware architecture, components [155, 128, 141] and memory coherency support - Exclusive access policies [112, 111] - Software virtualization [84, 114, 141, 41, 174, 112] - Memory transfer and memory coherency support - Spatial diagnostics [153]	§6.2
		Safety measures and techniques that can be used, adapted or extended from multi-core devices and FPGAs [127, 153, 143, 31, 75]	

Table 2: Summary of GPU terminology equivalence (CUDA, OpenCL)

Terminology	Terms					
CUDA	streaming multiprocessor	scalar core	grid	block	thread	warp
OpenCL	compute unit	processing element	NDRange	work-group	work-item	wavefront

2.2 Safety Certification Standards

IEC 61508 [92] generic safety standard is considered as a reference by several industrial and ground transportation domains such as industrial machinery (ISO 13849 [94]), robotics (ISO 10218 [93]), automotive (ISO 26262 [95]) and railway (EN 5012X [60]). For further details concerning domain-specific safety standards and certification processes, refer to Martinez et al. [122]. However, among all these safety standards, there is considerable variability in terminology, definitions and requirements. For example, in IEC 61508, the Safety Integrity Level (SIL) is defined with a discrete level range where number one is the lowest (SIL1) and four is the highest (SIL4), while in ISO 26262, the level is defined by the Automotive Safety Integrity Level (ASIL) with a range from 'A' the lowest (ASIL-A) to 'D' the highest (ASIL-D). Therefore, the survey considers IEC 61508 as a reference safety standard and takes into technical consideration previously listed industrial and ground transportation domains. Besides, the automotive

domain is a common target application for GPU devices and provides guidelines for applying the safety standard to semiconductors (ISO 26262-11). Thus, both IEC 61508 and ISO 26262 are considered reference safety standards in the survey. Nonetheless, several additional domain-specific safety standards do not consider IEC 61508 as a reference standard. Section 7 summarizes the specific characteristics of these domains (e.g., space).

Both reference safety standards require the adoption of *safety measures* (ISO 26262-1 §3.141) to avoid, detect, control and/or mitigate *random hardware failures* ("failure that can occur unpredictably during the lifetime of a hardware element and that follows a distribution probability" [95]) and *systematic failures* ("failure related in a deterministic way to a certain cause, that can only be eliminated by a change of the design or of the manufacturing process, operational procedures, documentation or other relevant factors" [95]). And safety standards already propose some example safety measures as recommended or not recommended for a given SIL (e.g., IEC 61508-3 Table A.2).

Finally, IEC 61508-4 (§3.2.11) classifies software off-line tools as class *T1* if it does not contribute to the executable software code (e.g., document editor), *T2* if it "supports the test or verification of the design or executable code" (e.g., test tool) and *T3* if it "generates outputs which can directly or indirectly contribute to the executable code" [92] (e.g., software compiler). And IEC 61508-3 (§7.4.4) describes tool qualification requirements for *T2* and *T3* classes. Besides, ISO 26262-8 (§11.4.5/6) defines the Tool Confidence Level (TCL) classification (*TCL1-3*), where *TCL3* is assigned if a tool error impact is severe and the error detection coverage is low (e.g., software compiler).

2.3 Fundamental Safety Technical Requirements

The development of IEC 61508 and/or ISO 26262 compliant safety-critical systems with devices that have on-chip redundancy (e.g., GPU devices) requires addressing at least the following fundamental safety technical requirements already defined for multi-core devices [153]:

- *Reliability of an item* (e.g., transistor, cache, device) is defined as the "ability to perform as required, without failure, for a given time interval, under given conditions" [2]. And the *reliability measure* is defined as the "probability of performing as required for the time interval (t1, t2), under given conditions" [2], which is quantified as the failure rate (λ) of a device, the number of failures per 10^9 device operation hours, measured in Failures In Time (FIT).
- *Diagnostic Coverage (DC)* stands for "the fraction of dangerous failures detected by automatic on-line diagnostic tests" [92]. It is expressed as coverage percentage and classified in ranges: low ($60\% \leq DC < 90\%$), medium ($90\% \leq DC < 99\%$) and high ($DC \geq 99\%$) [92].
- *Temporal independence* implies that "one element shall not cause another element to function incorrectly by taking too high a share of the available processor execution time, or by blocking execution of the other element by locking a shared resource of some kind" [92]. The reference standards [92, 95] recommend techniques such as *temporal predictability* (e.g., design time scheduling), temporal diagnostics and Worst-Case Execution Time (WCET) analysis.
- *Spatial independence* implies that "data used by one element shall not be changed by another element" [92]. For instance, exclusive access is a widely used technique to support *spatial independence* with shared resources [111, 112].

The survey uses both reliability definitions provided by the standard IEC 60050 [2] to support the reconciliation of the definition variability among research contributions that propose techniques to improve the reliability of items and improve reliability measures (e.g., [22, 153, 28]). However, neither IEC 61508 nor ISO 26262 define the term reliability, and internal references are scarce and mostly related to failure rate estimation. Nevertheless, as these standards state, the failure rate can be reduced (reliability increase) using, for example, control and mitigation techniques such as redundancy and fault tolerance (IEC 61508-4 §3.4.6, §3.6.3). Hence, Section 4 describes reliability research contributions that propose techniques to control and mitigate random hardware failures.

In addition to this, the Probability of dangerous Failure per Hour (PFH) is the average system failure frequency. The PFH depends, among other parameters, on both the device reliability (λ) and the DC [92]. The higher the reliability and/or the DC, the lower the PFH. The automotive domain defines different probability calculation formulas and methods (e.g., Probabilistic Metric for Random Hardware Failures (PMHF) in ISO 26262-5) that are also based at least on both parameters.

Finally, the integration of several software elements on a single device must ensure the *independence of execution*, so "that elements will not adversely interfere with each other's execution behavior such that a dangerous failure would occur" [92] (IEC 61508-3 Annex F). This requirement is equivalent to *Freedom from Interference (FFI)* [95] (ISO 26262-11 §5.4.2). And described *spatial independence* and *temporal independence* techniques are fundamental to achieve such freedom from interference and independence of execution (see Section 6).

3 GPU Device

The development of GPU-based safety-critical embedded systems requires a careful analysis and understanding of the underlying GPU device architecture with associated software programming languages and frameworks, taking into consideration the provided built-in safety measures (e.g., diagnostics, fault tolerance techniques), along with the execution independence risks that need to be mitigated (e.g., temporal interference). From a research perspective, there is a considerable research fragmentation linked to the different nature of threats and technical challenges that must be tackled. This section provides a common technical foundation based on which the following random hardware failures (§4), systematic failures (§5) and independence of execution (§6) Sections can categorize the research contributions. This Section provides an introduction (§3.1), describes a GPU taxonomy and metrics (§3.2), and summarizes GPU-specific features and challenges for both the device hardware (§3.3) and associated software programming languages and frameworks (§3.4).

3.1 Introduction

There is a cross-domain research interest in the potential applicability of GPU devices for the development of next-generation safety systems (e.g., automotive [176, 114, 170, 119, 182, 38, 79, 26, 171, 197, 15], railway [21, 20], avionics [27, 192, 126, 21], space [36, 105, 106, 107, 74, 116, 162, 196, 36, 9]). Both GPU and multi-core devices could potentially meet the computational and mixed-criticality integration requirements. However, while multi-cores ease programmability and flexibility, GPU devices and their associated programming languages and frameworks provide higher computing capacity [105] for applications characterized by high computational performance and parallelizable software algorithms, either for the safety function itself or for other integrated applications (mixed-criticality). Examples of computationally-demanding applications are graphics processing [192], computer vision perception [182], ML [118, 54], mathematics calculations and domain-specific computing-intensive functions (e.g., space [105, 162], automotive [79], railway [20]). Furthermore, GPU devices can provide very high ratios of computational performance per power consumption (W) [105], key in domains such as space where available power is limited [105, 162]. However, the most common functional safety software applications are sequential [79, 104]. Therefore, the effort required to parallelize and accelerate the execution of software portions versus the achievable gain in execution time (Ahmdahl's law [83]) and computing resource utilization [79], limits *discrete GPU* devices' competitiveness with respect to multi-core devices concurrent execution of sequential applications. Nonetheless, both *discrete* and *embedded GPU* devices with one or multiple CPU cores, can potentially combine both advantages and enable the integration of safety sequential applications (in CPU cores) and parallelizable software algorithms (in SMs) [12]. For example, Renesas R-Car H3 [5] and NVIDIA Xavier [4] devices include ARM cores suitable for the implementation of safety functions (e.g., lockstep mode) [13]. In this context, the selection of which functions/algorithms (or portions of them) are assigned to SMs or CPU cores becomes relevant, because some algorithms are amenable to SMs and other algorithms are not (e.g., space domain analysis [105, 106, 107]).

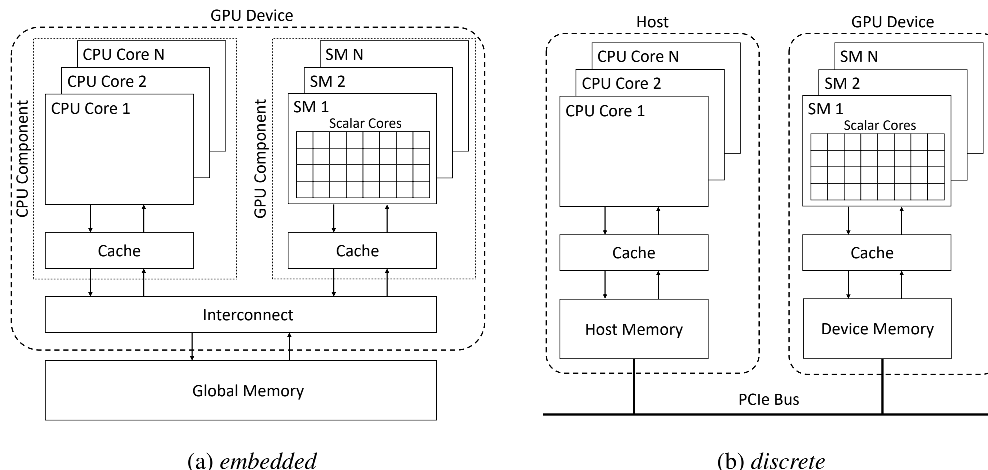
The integration of safety measures to manage random hardware failures (see Section 4) with a reasonable cost and shared memory management are key technical challenges that can limit the applicability of GPU devices for the development of safety-critical systems. Concerning shared memory management, in highly parallel computing architectures, the temporal and energy cost of arithmetic and logical operations can become negligible, compared with the cost of data transfers and memory accesses. For example, global memory operations can require two orders of magnitude more time than arithmetic and local memory operations [54]. So, device designers do not only need to optimize the parallel computational performance of the SMs, but especially the shared memory architecture (e.g., memory, cache(s)) and the interconnect components that perform data transfers. As shown in Figure 1a, in order to optimize maximum average performance, both the CPU cores and SMs share a common memory interconnect that arbitrates data access to a shared memory architecture composed with L2/L3 caches and shared memory components. In addition to this, in order to improve the maximum usage of resources and maximum computational performance of the GPU, several intermediate logic units (e.g., instruction dispatch units, Load Store Unit (LSU)) and hardware schedulers are required (e.g., warp schedulers).

3.2 Taxonomy and Metrics

This section describes the criteria used within the survey to classify GPU devices, safety software allocation scenarios, and GPU device reliability evaluation metrics.

3.2.1 GPU Device Types

Currently available *GPU devices* (or *devices* for short) go beyond the basic definition of graphics processing unit, and can be classified according to different taxonomies. This survey defines in Table 3 a GPU classification taxonomy



(a) *embedded* (b) *discrete*

Figure 1: Generic block-diagram architecture of *hard embedded and discrete GPUs*

that combines several sources [105, 141, 153, 200]. We use this taxonomy to classify and describe GPU devices and features including the computing specialization (*graphics*, *computing*), device type (*discrete*, *integrated* / *embedded*), implementation type (*hard*, *soft*) and safety compliance (*generic*, *safety*). For example, Xavier [4] is a *hard embedded safety computing GPU* device: a specialized ASIC device (*hard*), where the GPU component (SMs) is integrated in the same physical die with CPU cores (*embedded*), with a dedicated lock-step CPU compliant with ISO 26262 and IEC 61508 safety standards (*safety*), and natively supporting general-purpose parallel computing APIs (*computing*).

We use the term *GPU component* (or GPU for short) to refer to the SM(s) and associated hardware logic/blocks/registers/components, such as the shared cache, required to execute kernel(s) in the SM(s). If the GPU is implemented as an embedded element, see Figure 1 (left), the *GPU component* will be part of the Multi-Processor Systems-on-a-Chip (MPSoC) device along with the *CPU component* and other on-SoC components like I/O controllers. Instead, if the GPU is implemented as a stand-alone device, see Figure 1 (right), the terms *GPU component* and *GPU device* are the same. In the former case, *embedded GPU*, the integration can either be hard (ASIC), soft (e.g., FPGA) or combinations thereof (e.g., MPSoC with integrated FPGA). Finally, High-Level Synthesis (HLS) [131, 86] based soft-GPU solutions are considered outside the scope of the survey as they do not require the implementation of SM(s).

Table 3: GPU classification taxonomy

Feature	Type	Description
Computing Specialization	*Graphics*	GPUs capable of using only graphics APIs (e.g., OpenGL ES/SC).
	Computing	GPUs natively supporting general-purpose parallel computing (e.g., CUDA, OpenCL, OpenMP, OpenACC, Vulkan) and graphics APIs.
Device Type	*Discrete*	GPU implemented as a different physical module, connected to a host system through a connection interface such as PCIe. Usually, it contains its own DRAM memory.
	Integrated / Embedded	GPU implemented in the same physical module with CPU cores, i.e. die as a part of a Systems-on-a-Chip (SoC). Usually shares the same main memory with other computing components.
Implementation Type	*Hard*	GPUs provided as specialized ASIC or as part of a multi-core SoC / MPSoC device.
	Soft	GPUs provided as solutions to be integrated in FPGAs either as soft-core IP or HLS of a higher-level programming framework such as OpenCL.
Safety Compliance	*Generic*	General-purpose GPU (GPGPU) optimized for maximum average performance.
	Safety	GPU designed to comply with one or several safety standards.

3.2.2 GPU Safety Software Allocation Scenarios

A safety function is fail-safe if the safety function or the diagnostics can lead the system to a safe state if required (e.g., train stop). By contrast, it is fail-operational if, even in the presence of a failure, the developer must guarantee degraded or full system operation (e.g., autonomous driving). We describe techniques that safety engineers can use to design fail-safe or fail-operational safety-critical systems based on the defined architectural and fault tolerance strategies. For example, the fail-operational architecture for autonomous vehicles described by Yoshida [203] uses redundant and diverse hardware platforms that combine multi-core devices and a GPU device.

Regarding the allocation of safety software function(s) to a GPU device, we identify nine possible basic combinations as shown in Table 4:

- The combination where no safety application is executed in the GPU(s) and the CPU cores, is considered 'outside the scope' of the survey (safety-critical systems). See for example other generic surveys such as CPU-GPU heterogeneous computing techniques [130].
- The combinations where the GPU(s) execute one or more safety function(s) but not the CPU cores, are considered 'not feasible' because the CPU cores manage the execution of GPU(s). Thus, CPU cores should execute at least a portion of these safety function(s).
- The combinations where CPU cores execute one or more safety function(s) but not the GPU(s), are considered scenarios already covered by the survey on multi-core devices [153]. In this case, the GPU(s) are non-safety-related computing accelerator components.
- The combinations where CPU cores and GPU(s) execute one or more safety function(s) are considered within the technical scope of the survey.

3.2.3 GPU Device Reliability Evaluation Metrics

GPU device reliability evaluations are commonly performed by methods such as Fault Injection (FI) and vulnerability factor metric estimations [187, 170, 46, 51]. FI methods deliberately introduce errors to analyze the reliability and susceptibility of the device to such errors. Table 5 classifies FI methods and tools used to evaluate the reliability and resilience of GPU devices and components, using the FI taxonomy defined by Benso et al. [28]: *execution/simulation* and *software/hardware*. And the basic effects of a FI action are: *no effect* (an error is not induced, or implemented safety measures correct it), *functional interruption* (e.g., crash) and Silent Data Corruption (SDC) (the induced error corrupts data or the GPU internal state) [139].

Execution and software-based FI is performed with tools that inject software errors in a software application running on a device (e.g., SASSIFI [78]). Besides, simulation and software-based FI is executed with specialized GPU soft models where software errors are injected on a simulated/emulated device (e.g., FlexGrip [165]). Within the scope of the survey, the focus of these software-based FI techniques is not to inject software application errors (e.g., software bugs) but the (expected) software errors generated by random hardware errors. These FI techniques can be used to obtain probability-based metric estimates such as the Architectural Vulnerability Factor (AVF) and Program Vulnerability Factor (PVF). The AVF metric provides a probabilistic quantification of a transient hardware error leading to a computation error [187, 3] (ISO 26262-11 §4.6.1.8). That is, the fraction of transient hardware errors that lead to a computation error. Besides, the PVF quantifies software program(s) vulnerability metrics when executed on a given device. Moreover, additional metric specializations can also be defined, restricting the PVF analysis to particular portions of an algorithm [170, 46]. Finally, execution and hardware-based FI campaigns provide empirical failure rate measurements using specialized laboratories (e.g., neutron beam), extreme operating conditions (e.g., high temperature) or in-field testing in harsh environments.

The ISO 26262 defines two hardware architectural metrics, Single-Point Fault Metric (SPFM) and Latent Fault Metric (LFM), which measure the robustness (the higher, the better) of a given item to different types of faults (e.g., single-point, latent multi-point, residual faults) based on design (e.g., safe faults) and DC safety measures. And both AVF and PVF vulnerability factors can be used to estimate the fraction of safe failures for derating purposes, which potentially reduces the considered *residual failure* rate, reducing the PMHF and increasing the SPFM [95] (ISO 26262-5 Annex C).

3.3 Device Hardware - Components

The most relevant GPU device architectural components are described in Section 2.1 and Figure 1. Except for the SM, all described components are also common for multi-core devices: *processing units* (CPU cores, SMs), *memory*

Table 4: Basic allocation combinations of safety software function(s) in a *GPU* device

GPU(s)	CPU core(s)		
	Non-safety	**Safety**	**Mixed-Criticality**
Non-safety	Outside the scope	See multi-core devices survey [153]	
Safety	Not feasible	Scope of the survey: CPU core(s) and GPU(s) execute safety function(s)	
Mixed-Criticality	Not feasible		

Table 5: A classification of Fault Injection (FI) based reliability measurement/analysis

FI Type	GPU device	Tool	References
Execution and software-based FI	NVIDIA Tesla K20/K40, GeForce 900	SASSIFI tool	[160, 78, 159, 158, 170]
	NVIDIA Tesla K20, GeForce 900	LLFI fault injector variant	[117]
	NVIDIA Tesla C	GPU-Qin tool, *cuda-gdb*	[63, 64, 66, 65]
	NVIDIA Titan V	NVBitFI	[186]
Simulation and software-based FI	NVIDIA G80 (Soft-GPU)	FlexGrip model (VHDL)	[165, 163]
	NVIDIA Fermi (Soft-GPU)	Soft-GPU model (VHDL)	[133]
	AMD Radeon HD 7000 NVIDIA Quadro FX, GeForce 400	SIFI, based on Multi2Sim GUFI, based on GPGPU-Sim	[177, 187, 190, 189, 188]
Execution and hardware-based FI	NVIDIA Kepler, Tesla, GeForce 400, Tegra X1, Titan X	Neutron-beam	[157, 169, 161, 138, 140, 185, 72, 139]
	NVIDIA Tesla K20, Tesla C	Extreme operation	[55, 134]

Table 6: Classification summary of referenced *hard GPUs*: microarchitecture and CPU core type

Mfr.	Microarch.	CPU cores: Discrete GPUs (None), Embedded GPUs (e.g., ARM, x86)
AMD	Vega 8	x86 [36]
	Terascale	None [190, 189, 51]
	GCN	x86 [144, 80, 145, 27, 191]
ARM	Bifrost	None [105]; ARM [176]
	Midgard	None [74, 116, 40]
Intel	HD 530	x86 [46, 50]
NVIDIA	Tesla	None [40, 39, 26, 51, 74, 125, 124, 135, 199]
	Volta	None [119, 54, 100, 201]; ARM [12, 14, 34, 40, 47, 162, 15, 25, 32, 39, 47, 105, 142]
	Pascal	None [12, 14, 13, 16, 40, 169, 39, 96, 81, 120]; ARM [18, 9, 40, 42, 41, 25, 39, 109, 142, 200]
	Maxwell	None [40, 78, 117, 39, 50, 74, 116]; ARM [38, 40, 46, 79, 169, 39, 43, 69, 68, 70, 74, 116, 147, 146, 200]
	Kepler	None [74, 116, 35, 140, 170, 160, 78, 117, 157, 169, 134, 50, 49, 51, 97, 101, 167, 200, 139]; ARM [46, 43, 74, 85, 116, 200]
	Turing	None [40, 80, 142]
	Fermi	None [33, 30, 140, 63, 161, 138, 55, 50, 49, 51, 64, 66, 65, 72, 74, 135, 182, 139, 174, 6, 187, 161, 59, 67, 72]
Renesas	Power VR	ARM [203, 32, 105]
Vivante	GC2000	None [114]

(e.g., device memory, cache) and *data paths* (interconnection network and shared bus). Hence, both GPU and multi-core components share multiple common techniques [127, 143, 153]. In this survey we emphasize those particularly relevant for GPU devices.

Processing Units As described in Sections 2.1 and 3.2.1, the SMs and the CPU cores are the two common and distinct processing units of a GPU device, which could also integrate additional device-specific accelerators (e.g., deep learning accelerator [4]). Table 6 summarizes the GPU microarchitectures analyzed or used (e.g., case study) by referenced research contributions to facilitate the search for microarchitecture-specific techniques, solutions, and examples. As explained in Section 2.1, CPU cores (with their local register files and cache memories) offload work on the GPU(s), monitor execution, and provide a user interface to use the GPU(s) [196]. Commonly, they are part of multi-core CPU components for performance reasons (e.g., [4]). Therefore, systems equipped with GPU devices inherit those multi-core device characteristics summarized in the survey by Perez et al. [153], in terms of reliability, DC, temporal and spatial independence.

Memory As described in Section 2.1 and shown in Figure 1, a GPU device contains several memory components such as device memory, cache(s) and external global memory. GPU devices also include local SRAM memories for predictable and low latency data sharing and communication across threads executing either concurrently or serially. Those SRAM memories are explicitly managed by software and hence, fall in the category of Scratchpad Memory (SPM). The extreme case corresponds to register files, whose size in GPU devices may be similar to the size of L1

caches in CPU cores. Therefore, while they are referred to as register files, in essence, they are not too different from an SPM.

Discrete GPU devices feature their own DRAM memory, which is distinct from the one used by the CPU cores. These two DRAM memories are generally non-coherent among the host (CPU cores) and GPU(s), thus needing the host to copy data into the GPU(s) memory region before starting computation and retrieving output data back to host memory when GPU(s) computation is complete. Such a memory management model avoids the need for managing data coherence across the host and the GPU(s). However, it may cause a non-negligible performance degradation. More recent *embedded GPUs* use a single DRAM memory shared with the CPU cores in the same SoC with hardware support for coherence [196]. Note that shared memory in embedded GPUs is different from OpenCL SVM and CUDA UVM [39, 40], which are programmability features to simplify GPU programming. By default, the programming model in both types of GPUs is the same and requires explicit copies. In fact, it is argued that unified memory should be avoided in real-time systems (e.g., [137] §3.1) due to its black box implementation [137, 39, 40]. However, when it is used, its implementation on embedded GPUs takes advantage of the physically shared DRAM.

In those systems where memory is not coherent across the CPU cores and the GPU(s) (e.g., *discrete GPU*), it is particularly critical to keep data local in the GPU(s) to minimize performance costs due to data transfers. Instead, in embedded GPUs while keeping data local is still preferable from a performance perspective, fetching data from memory is less costly. In both cases, GPU devices typically need to manage large amounts of data. Hence, memory is a key component with a significant impact on hardware random failures and independence of execution.

Data Paths Interconnection networks and shared buses for GPU devices, either internal or external to connect the CPU cores, GPU(s) and the shared memory, are generally subject to the same requirements as those for multi-core devices. Therefore, we refer the reader to a survey on multi-core devices [153] for further details regarding reliability, DC, temporal and spatial independence. In both cases, namely CPU cores and GPUs, this component is key to providing high average computing performance. Thus, silicon manufacturers tend to provide none or limited public information about the internal design and deployed safety measures to manage random hardware errors, temporal interference and spatial interference.

3.4 Device Software - Parallel Computing Languages and Frameworks

GPU devices support only a subset of the many parallel programming models and tools [56]. In fact, embedded GPU and safety GPU devices support even fewer options since the software stack of GPU devices is dominated by closed source drivers and libraries, in order to maintain the secrecy about the proprietary designs [200, 145, 39, 107]. Hence, the available programming models and supported operating systems for each GPU are limited to the ones supported by the vendor, which is frequently dictated by its customers. Some GPU manufacturers follow a more open approach, such as Broadcom, whose VideoCore IV GPU is the only embedded GPU with publicly available specifications. Others like AMD and Intel, despite that they do not provide information about the internals of their hardware designs, they do publicly provide the description of the hardware/software interface of their devices so that open source or safety-critical drivers can be developed. In fact, both of these companies offer open source device drivers for their GPUs, while AMD also supports a proprietary driver, which usually performs better than the open source one.

The primary function of GPU devices has been graphics acceleration, therefore graphics APIs such as OpenGL are used for this purpose. OpenGL has experienced a long evolution over the years, and has variants for serving the desktop and gaming domain (OpenGL), embedded devices (OpenGL ES) and safety-critical systems (OpenGL SC). Some low-end solutions such as ARM's Mali-400 found in MPSoCs used in safety-critical systems, such as the Xilinx Zynq Ultrascale+, only support graphics operations (*graphics GPUs*), using the OpenGL ES version. However, to our knowledge, there is no publicly available information of this API to be used in a safety-critical context. There are two reasons for this: the OpenGL ES contains features that are not appropriate for critical environments, such as dynamic allocation. Moreover, OpenGL ES implementations are closed source and only supported in general-purpose operating systems such as Linux and Android.

Some higher-end safety GPU devices, such as NVIDIA Xavier, support both desktop and embedded versions of OpenGL, but not the safety-critical one. In addition to these, the latest high-end GPU devices also support Vulkan, a low-level API that allows both, the execution of graphics and general-purpose compute workloads. Thanks to this, Vulkan provides almost full control over the GPU hardware, allowing for reduced time overhead of GPU code launches [47].

Besides graphics APIs, most of the latest GPU devices also natively support general-purpose computing APIs (for *computing GPUs*), which are widely used in high-performance computing, and are also investigated for use in real-time and critical systems. The most widely GPU programming model is CUDA, which is proprietary and supported

only by NVIDIA GPUs. Due to its closed source nature, CUDA presents several challenges for use in critical systems. For example, the authors of [200] have detected cases in which programs written in CUDA behave differently from the NVIDIA documentation. Similarly, Calderón et al. [39] showed that the memory allocated by the NVIDIA driver exceeds by far the requested memory by the application.

AMD GPU devices support a CUDA-compatible language called HIP, which has an open source implementation on top of AMD's open source GPU software stack (ROCm). Some authors [145, 107] argue that due to the more open approach of AMD in terms of software programming models, they present a good candidate for use in critical systems. However, frequent changes in the development of these APIs [145] complicate the use of such methods in practice.

Besides these pure GPU languages and APIs, there are also other higher-level ways to program GPUs. For example, OpenMP has recently added support for GPUs through its accelerator offloading mode. This allows to easily annotate a portion of an application source code with directives in order to be executed on a GPU, without significant changes in the application source code. In a similar way, OpenACC, which is an OpenMP-like language for GPUs, offers the same functionality, using similar syntax, but higher performance compared to OpenMP.

3.5 Software Application

The software application executed in a given GPU device can be generalized as a set of software applications to be executed in the CPU cores and GPU(s). In order to enable the mixed-criticality integration of software application(s) of different criticality, Burgio et al. [38] propose a software stack composed of virtualization technology (hypervisor), General Purpose Operating System (GPOS) (e.g., Linux) and domain-specific stacks (e.g., AUTOSAR). These solutions are also common for multi-core devices and are described as related research in Section 8.2.

4 Random Hardware Failures

Random hardware failures occur unpredictably. They cannot be (completely) avoided, but only controlled, mitigated and/or detected in order to comply with (extremely) low probability failure ranges and specific metric ranges defined by safety standards (e.g., ISO 26262 PMHF, SPFM, LFM; IEC 61508 PFH). This Section introduces random hardware errors (§4.1) and describes safety measures to control, mitigate and detect errors (§4.2). As introduced in Section 3.3, GPU and multi-core devices share multiple common architectural components and GPUs already integrate, adapt or extend safety measures outlined in multi-core and FPGA device surveys [31, 153, 127, 143] (e.g., Error-Correcting Code (ECC) for memory). Hence, this section focuses on those works and features particularly relevant for GPU devices.

4.1 Introduction

Advances in manufacturing processes, electronic design automation tools and CMOS downscaling technology enable the production of 7 nm devices. However, this downscaling gives rise to several threats for reliability such as *Process, Voltage and Temperature (PVT) variability*, *device aging*, *transient faults* (*soft errors*) and *permanent faults* [143]. Two opposite trends exist due to this technology scaling: decreased reliability due to the use of smaller devices and lower supply voltage, thus increasing susceptibility to particle strikes and small defects, and increased reliability due to requiring smaller area for any given circuit, thus decreasing the susceptible area. The resulting FIT rates, in general, are kept low by applying mitigation techniques at the process level (e.g., improved lithographic process), or deploying fault tolerance solutions (e.g., hardening, redundancy). Moreover, some types of advanced manufacturing technologies, such as Fully Depleted Silicon On Insulator (FD-SOI), are inherently immune to certain radiation effects such as latch-ups, which provides increased reliability for certain domains such as space, while products from different fabs or using different manufacturing processes have been observed to have different reliability properties [127]. However, whether all those solutions are applied to improve raw reliability in COTS GPU devices is privately known (e.g., manufacturers) but generally not publicly known.

The optimization towards the highest computational performance is a potential risk for safety-related software execution, since, as already analyzed by Oliveira and Wunderlich et al. [195, 138], the corruption of shared resources such as hardware tasks schedulers and memory caches can lead to the corruption of thread data and downstreaming thread data [140, 170, 51]. There are several reasons, which can also be extrapolated to safety-related application domains, that make *generic GPUs* more error-prone and therefore less reliable than multi-core devices [51]:

- The optimization for maximum average performance and massive parallelism, where (1) the execution depends on several unreliable computing and communication units with no integrated error detection

mechanism and undisclosed documentation for analysis, and (2) an error in a shared resource such as shared memory, scheduler and dispatcher can lead to downstreaming thread errors.

- The higher density of memory, higher number of computing units executing in parallel and a higher ratio of failures due to overheating compared to CPUs [51] can potentially lead to a higher number of errors and reduced reliability.

However, *hard embedded safety GPU* devices already address these issues, bringing them on par with safety-related multi-core devices. For example, the latest automotive devices designed to comply with ISO 26262 and optionally IEC 61508 safety standard, such as NVIDIA's Xavier [4, 34], Renesas's M3 or ARM's Mali G78AE, already contain features that improve their reliability, such as ECC protection or virtualization, which can also be used for time and space partitioning. Moreover, *embedded GPU* devices have some relevant differences compared to *discrete GPU* devices counterparts used in high-performance computing that vary their reliability concerns. For instance, among other factors, smaller area potentially contributes to lower exposure to cosmic radiation, and their power profile is lower, which means that the heat they generate is also lower and, in fact, similar to that of multi-core devices [105]. However, device-specific analysis are required as these factors are just some device-specific relevant factors, but not the only ones to be considered.

Nonetheless, to the best of our knowledge, there are neither consolidated comparative research studies for safety-related systems as opposed to supercomputers [51, 117], nor public reliability measurements provided by GPU device silicon manufacturers as opposed to multi-core device silicon manufacturers (e.g., [198]) that could enable a quantitative comparison of two diverse but functionally equivalent implementations in GPU and multi-core/FPGA devices. Besides, Table 5 (page 8) provides a classification summary of fault injection-based campaigns, from which only a subset focuses on AVF metric estimation or generic reliability and resilience evaluation of the GPU device not linked to a specific software application. For example, radiation sensitivity evaluation campaigns have been implemented to experimentally measure the sensitivity of GPU devices and mitigation provided by integrated ECCs. And the neutron sensitivity experiments described by Oliveira et al. [138] show that even with ECC protecting memory and registers, the overall reliability of the device is jeopardized by errors in the logic and scheduler. This is so because several key resources and components are not protected by such mechanisms (e.g., warp scheduler, thread scheduler, interconnect, instruction dispatch units, pipeline stages, ALUs, LSU), and even internal documentation required for analysis is not disclosed [140, 160, 120, 64].

As characterized by fault injection reliability assessments, vulnerability outcome probabilities are dependent on the executed software application and instruction groups, the way the software stresses the hardware resources, the type of memory usage (e.g., general-purpose registers files, local memory) and can have a time-varying behavior correlated with executed kernel(s) characteristics [160, 78, 190, 51]. For example, a matrix multiplication algorithm software executed (with the required hardware components) in different GPU devices can lead to a significant FIT values variability range, between approximately 150 (FX5600) to 250 (Radeon 7970) [190].

Besides, the extensive analyses conducted to understand GPU errors on large scale supercomputers [123, 179, 134, 51] have drawn relevant insights on the reliability of generic GPU devices and the importance of managing random hardware errors, e.g., GPU as predominant FIT contributor [123], high random hardware error rates and high error correction rates (e.g., 99.997%) [123].

Santos et al. [169] provide an empirical analysis of the soft error sensitivity of a neural network-based object detection library on three different NVIDIA GPU devices: Kepler K40, Maxwell Tegra X1 and Pascal Titan X. Authors observe that the Tegra X1 error rate halves that of the K40, which is attributed to the smaller area used by the former (20nm technology) compared to the latter (28nm technology). While area could justify error rate to halve, using smaller transistors could easily lead to increased soft error susceptibility, as discussed before. Therefore, although not public information is given, we expect some additional process or fault tolerance solutions to be in place for Tegra X1 w.r.t. K40. The authors also observe a 10x lower FIT rate for Titan X w.r.t. K40. A number of factors contribute to such reduction: (1) the use of 3-D transistors (FinFET technology) instead of standard CMOS, which provides 10x lower sensitivity per bit to particle strikes. (2) The Pascal architecture is particularly devised for neural network execution. Hence, it is expected that fewer resources are used, therefore further decreasing FIT rates. However, it is difficult to draw solid conclusions from the results of these analyzes, due to the multiple factors that can potentially affect the sensitivity to transient errors and because multiple factors are even not publicly known, e.g., microarchitecture details, different number and mixes of SRAM and flip-flop cells.

Finally, the deployment of *soft GPUs* (VHDL) on FPGAs can take advantage of the 'safe execution environment' that can be provided by modern FPGAs and SoCs/MPSoCs with integrated FPGA(s) [31]. For example, FPGAs use several hardening techniques and fault tolerance mechanisms for increasing reliability and enable their usage in harsh environments such as aerospace [31, 153].

4.2 Safety measures to control, mitigate and detect errors

Safety measures to control, mitigate and detect errors, are commonly implemented using hardware built-in safety mechanisms and complementary software-only techniques. As briefly explained in Section 3.1, integrating safety measures with a reasonable cost is a key technical challenge. The integration of hardware built-in safety mechanisms (e.g., diagnostics, fault tolerance, redundancy, hardening) can increase the silicon die area, power consumption and cost, and performance decrease [153]. For example, the deployment of GPU devices for harsh industrial environments such as automotive grade-0 (operating temperature range between –40ºC to 150ºC) and space [105], needs to consider a trade-off between performance and reliability. This occurs because the development of devices designed to operate in those environments needs to consider larger hardened transistors, wider wires, circuit design constraints, diagnostics, and fault-tolerance mechanisms that reduce the overall computing performance and increase the cost [15, 153, 127]. Moreover, in some cases, COTS devices such as GPUs cannot be designed with the same reliable nano-technologies used for other critical components for that domain (e.g., space, avionics). In this case, the industry takes advantage of the inherent variability of the produced chips, so the most reliable devices can be selected out of a given batch, after passing a predefined set of screening and stress tests (e.g., thermal, vibration). Additionally, another common approach to manage this trade-off is the implementation of software-based safety measures, using and extending existing hardware built-in safety mechanisms or implementing additional software-only safety measures.

GPU devices provide a rich set of computing hardware that could be potentially used to implement generic redundancy and diversity techniques such as spatial diversity, temporal diversity, computation unit layout diversity. For example, software redundancy is a common technique for reliability improvement that can be implemented with additional requirements such as cache sharing limitation among threads, which mitigate the effect of common shared resources corruption as identified in fault-injection campaigns [138]. Redundancy can also be combined with spatial diversity, such as blocks duplication and distribution in different SMs, to further reduce the probability of common cause failure and improve reliability [138]. In addition, diverse redundancy can be applied at kernel level, exploiting the scheduling properties of the GPU devices to ensure that blocks from redundant kernels are executed by different SMs [14]. Similarly, this idea can be applied in the design of new GPU device architectures [13]. However, the practical exploitation of these techniques is limited for developers due to the lack of information and software execution control due to 'black box' hardware components (e.g., ECC algorithm [139]) and 'software frameworks' that manage such resources [15] (e.g., resource allocation [40]). These 'black box' hardware and 'software frameworks' are key elements to provide competitive devices with maximum average performance, so information provided by silicon manufacturers is none or limited [98], and several research initiatives aim at providing an understanding of the implemented management [40, 18].

Therefore, selecting and combining sufficient and cost-effective safety measures becomes a device and application-specific challenge. For example, given a software safety function that requires a medium DC and uses a set of (design-time identified) device hardware resources, the safety engineer needs to use available hardware built-in safety mechanisms that commonly provide low to medium DC (e.g., ECC for memory) and complementary software-only techniques to achieve a medium DC. In this context, this subsection describes hardware and software safety measures for most relevant GPU components (§4.2.1) and software-only safety measures (§4.2.2).

Moreover, due to the high variability of GPU devices and microarchitectures used in research contributions (see Table 6 on page 8), along with different experimental scenarios (e.g., case study, method, programming style [72]), it becomes difficult to compare proposed safety measures and select the appropriate ones without performing device and application-specific analyses. Besides, the potential adaptation to the safety-critical domain of generic high-performance computing domain research contributions is not straightforward, as it requires to comply with strict safety standards requirements and associated technical challenges. For example:

- *Depth and rigor*: Adapted safety measures need to meet the depth and rigor required by safety standards. For example, the naïve implementation of software Duplication with Comparison (DWC) of (just) the output results does not guarantee a medium DC, as at least sufficient and representative intermediate calculation results should also be compared to detect latent errors. And the device-specific benefits of spatial/temporal diversity should also be analyzed.
- *Not recommended* techniques: Whenever a safety measure is not-recommended, reasonable and sufficient justification should be provided if used. For example, *backward recovery* technique (e.g., checkpoint and restart) is considered not-recommended (e.g., IEC 61508-3 Table A.2) because hard temporal deadlines could be potentially missed in case of several consecutive backward recoveries.
- *Dynamic*: Dynamic and reconfigurable behavior are commonly considered not recommended due to the difficulty of providing sufficient evidence of correct and safe behavior under all possible scenarios. However, not all designs have to be static, and it is possible to use limited and design time defined dynamic behaviors.

- *Tool qualification*: Techniques that rely on off-line software tools that modify/instrument software source code need to consider tool qualification requirements (e.g., T3, TCL3).

4.2.1 Components

This subsection summarizes hardware built-in and software-based safety measures that control, mitigate and detect random hardware errors at SM, memory and data path components. As SMs also integrate memory and data path (sub)components, the described SM safety measures can be complemented with memory and data path specialized safety measures.

Processing Units Due to the limited public documentation, controllability and observability of SM architectures, few works target the reliability of SMs explicitly. In general, works attempt to improve SMs (and other components simultaneously) employing different software-based redundancy techniques for the whole SM, its cores only, or parts of those cores (e.g., pipeline registers [163]) using available underutilized resources in order to reduce the computing overhead [6, 191, 178, 199, 57, 73]. Some authors combine software redundancy with diversity to mitigate common cause failures by, for instance, making redundant threads execute with some staggering in different cores [13, 14, 8].

Besides, most analyses and solutions for error detection on SMs are also software-based, employing diverse and different approaches. For example, some works adapt common techniques such as redundancy to detect execution errors (e.g., DWC [99], instruction replication [120]) and *backward recovery*-based techniques [71, 175, 180, 135]. And other works propose specialized adaptations such as control flow and data flow checkers [132] and dynamic DWC to detect permanent errors [164].

As previously explained, due to the high variability of considered devices, microarchitectures and scenarios, the application of previously summarized diverse example techniques require a device and application-specific analysis with the required *depth and rigor*. Besides, techniques that rely on tools such as compilers (e.g., [191, 132]) shall consider *tool qualification* requirements, *dynamic* actions (e.g., [164, 6]) shall be safely managed and *backward recovery*-based techniques shall safely manage their generic *not recommendation* rationale.

Memory Memory concerns for GPU devices are shared for multi-core devices [127, 57]. In general, GPUs have higher memory performance demands than CPUs due to their increased computational capacity, which calls for larger amounts of data being transferred per time unit. Hence, reliability threats for multi-cores are further exacerbated for GPUs. Nowadays, ECC is a common built-in error detection (diagnostic) and recovery mechanism (fault tolerance) integrated in memories [120, 169], which can result in an effective reduction in the number of errors (e.g., 4x reduction [169]) should they occur while data is stored or whenever transmitted. And in order to improve reliability, new technologies have emerged (e.g., phase change memories, normally used for storage but also considered as main memory [113]) and advanced manufacturing technologies (e.g., 3D stack) [153, 127]. For example, Zhang et al. [193] have observed that 3D stacking brings a shielding effect where inner dies may get more than 90% of the particle strikes filtered. Hence, if GPU devices are integrated with 3D memory dies on-chip for performance reasons, soft error reductions may be obtained at the same time. Similar observations have been raised in multiple works that exploit heterogeneous exposure to particle strikes of the different layers due to shielding effects to trade-off between performance and reliability [127]. Note, however, that, despite soft error mitigation achieved with 3D stacking, such designs have some disadvantages related to their challenging heat dissipation, fabrication costs, and yield loss w.r.t. conventional 2D designs [53].

The reliability improvement of cache memories is crucial because most of the device area is devoted to cache SRAM memory, which is susceptible to transient faults [143]. Therefore, nowadays ECC is also commonly integrated in caches, e.g., in the recent NVIDIA Ampere architecture the first and second level caches are ECC protected [108]. However, it should be taken into account that increasing the cache size can lead to a higher soft error rate increase (superlinear) [127]. In addition to this, generic high-performance computing device techniques are also potentially applicable, such as 4-layer 3D stacked cache memory design to drastically reduce the innermost layer vulnerability to soft errors and usage of Spin-Transfer Torque RAM (STT-RAM) technology for caches [153, 127]. Finally, cache coherency diagnosis is briefly explained in Section 6.2.

Concerning DRAM-based *global memory*, some authors have studied the relevance of different types of faults, namely transient and permanent, in DRAM memory devices, concluding that permanent faults are more significant in memory [91]. While this study is not GPU-specific, it is particularly relevant for GPU devices due to their strong dependence on (abundant) data fetched (frequently) from memory. Hwang et al. [91] also show that decommissioning faulty pages is an effective solution with negligible cost in general to detect and control permanent errors. But this a *dynamic* behavior that needs to be defined at design time. Concerning transient errors, Maruyama et al. have seen that it was common having ECC protected DRAM memory for CPUs, but at that time, its use was not available for

GPU devices [124, 125]. Hence, they proposed a combination of software-based ECC diagnostic and checkpointing for GPU memory to tolerate faults. However neutron-beam FI experiments have shown that software-based ECC could lead to a significant reduction of SDC (e.g., one order of magnitude) but even higher functional interruption rates than unprotected executions (e.g., 55% increase for matrix-to-matrix multiplication) due to random hardware errors associated with the hardware resources required to execute the ECC software [139]. So, in the absence of a hardware built-in ECC mechanism, the usage of software-based ECC for safety-critical systems requires a careful device-specific analysis (*depth and rigor*).

Moreover, DDR memories (mainly DDR3 and DDR4) suffer the "row hammer" effect, in which memory cells may leak some of their charges to nearby cells when accessed. While these effects have been mitigated to some extent, they are expected to exacerbate as memory technology scales down. Therefore, GPU devices accessing large DDR memories (several gigabytes) at high frequency may suffer reliability problems due to the row hammer effect [196]. This effect, either as a random hardware failure where the memory content is unintentionally modified or as a result of an intentional security exploitation, can lead to memory content corruption (a memory error).

Data path As explained in Section 3.3, interconnection networks and shared buses are key to providing high average computing performance, and manufacturers tend to provide none or limited public information. Thus, error detection is usually performed with software-only techniques (§4.2.2) that do not diagnose the data path itself but the errors that can be generated [153] (indirect diagnosis).

4.2.2 Software-Only Techniques

Software-only techniques can be used to increase reliability whenever hardware support is not sufficient. Software techniques are either application-agnostic (e.g., software redundant execution [34, 12, 14, 170, 120, 140, 57, 191], multithreading-based soft error fault tolerance [77, 57, 191]), application-dependent (e.g., Algorithmic-Based Fault Tolerance (ABFT) [35, 138, 169, 170, 168, 195], application-specific [102]) and/or combinations of the previous [170]. Equivalent solutions can be generally used in both GPU and multi-core devices. But, a fundamental difference is that controllability in GPU devices is much lower than in multi-core devices, for instance, when implementing software-only diverse redundancy with strong guarantees on multi-core devices [12] and limited guarantees on GPU devices [14] due to the inability to monitor progress in the GPU once kernels are offloaded. Oliveira et al. [140] provide an extensive description of software mitigation techniques for non-safety-related uses of GPUs. In addition to this, there is a reported correlation between the measured reliability and the executed applications [63, 78, 159], and even the explored application input sizes and biased input values [158, 159] (e.g., 30% failure rate increase with input size change [159]).

ABFT techniques are algorithm-dependent and exploit algebraic properties and data relationships of the algorithms to be protected to provide error recovery. Hence, by performing data interpolation and/or adding some form of software redundancy, ABFT techniques allow correcting errors without using *not-recommended* techniques such as checkpoints. Instead, by operating results from error-free computations, erroneous data can be corrected. ABFT techniques can be used for reliability improvement [138, 49, 50, 110], since they can correct both computing and memory errors [49], with significant example detection and correction ratios of radiation-induced errors (e.g., 60% in Kepler devices and 50% in Pascal devices [169]) and reasonable performance overhead (e.g., 18% [139, 156]). Most of them target matrix-based operations being matrix multiplication (MxM) the most popular [58, 76, 169, 35, 110], with the definition of specialized ABFT techniques for MxM [169, 35]. Neural networks have also been the target of abundant research [204, 168, 169], but their core computation operation consists of MxM, thus being these research works in the same group of those targeting MxM. This is also the case for Santos et al., whose target is evaluating the vulnerability factor for kernels and layers in the object detection process [170]. Finally, neutron-beam FI experiments have shown that ABFT can efficiently detect and correct SDC errors (e.g., for MxM reduction of two orders of magnitude with 18% computation performance overhead) but can potentially lead to worse *functional interruption* ratios (e.g., 12% increase for MxM) due to the increased hardware resources and computational operations required to execute the ABFT [139]. Thus, the applicability of ABFT techniques is algorithm dependent and the estimation/measurement of the potential benefits require a software and GPU-dependent analysis (*depth and rigor*).

Application-dependent analyses and fault tolerance solutions are also abundant. However, these hardening strategies might require device, algorithm and application-specific implementations that limit their generic application [138]. For example, the Kernel Vulnerability Factor (KVF) analysis of specific algorithms can lead to the definition of selective hardening techniques for a given algorithm, such as partial duplicated execution of the Histogram of Oriented Gradients (HOG) algorithm [170]. This can also be extended to higher software layers with the Layer Vulnerability Factor (LVF) analysis of specific software algorithm implementations and definition of selective hardening techniques, such as partial duplicated execution of the You Only Look Once (YOLO) software library [170]. Finally, other generic approaches define methods to analyze the results of FI campaigns. For example, [102] defines a statistical method

to estimate the resilience of GPU applications based on extracted dynamic execution and error profiles. Nonetheless, as the results of these analyses guide design decisions, their adaptation to the safety-critical domain should lead to improved methods with sufficient *depth and rigor*.

Besides, other high-performance computing domain techniques combine tools (e.g., compiler) and software runtime frameworks for performance efficient error detection and correction (e.g., [115]). Their adaptation to the safety-critical domain should at least consider *tool qualification* requirements and safety standard requirements for the software runtime framework.

Concerning error detection techniques (diagnostics), software DWC is a common software technique [138, 127, 57, 178, 6], where the results of the duplicated execution are compared to detect errors. This safety measure has a higher computational performance overhead (e.g., +90% to +151% [139]) and should be used with the required *depth and rigor* for safety-critical systems. Besides, the implementation of DWC can make use of spatial and/or temporal diversity in order to achieve high SDC error reduction rates (e.g., two orders of magnitude) and varying functional interruption ratios (e.g., one order of magnitude improvement with time diversity) [139]. Thus, DWC is a generic safety measure with none or limited GPU device architectural dependencies that can potentially provide a medium/high DC with a high computing performance overhead.

Finally, off-line tool-based software application-specific diagnosis solutions are also common. For example, Yim et al. perform fault injection campaigns and propose the ad-hoc insertion of strategically allocated error detectors in the software application [202] and Henriksen proposes a compilation strategy to support array index bounds checking diagnostics for GPUs [80]. However, the adaptation of off-line tool-based techniques that modify/instrument the software source-code should consider *tool qualification* requirement challenges (e.g., class *T3*, *TCL3*).

5 Systematic Failures

Reference safety standards set requirements for the avoidance and control of systematic faults in the overall safety life cycle with specific requirements for the development of hardware (e.g., IEC 61508-2 §7.4.6, §7.4.7) and software (e.g., IEC 61508-3, ISO 26262-6 §7.4). This section categorizes and describes research contributions that target the avoidance and control of systematic faults in the device hardware (§5.1) and software parallel programming languages and frameworks (§5.2).

5.1 Device Hardware

The generic informative recommendations for avoiding systematic failures in the design and development of ASICs are described in IEC 61508-2 (Annex F), and ISO 26262-11 provides the guidelines that describe the application of ISO 26262 to semiconductors. This is a silicon engineering challenge that, as stated in IEC 61508-2, assumes that in mass-produced electronic integrated circuits (§7.4.6.1) “the likelihood of faults in such devices is minimized by stringent development procedures, rigorous testing and extensive experience of use with significant feedback from users”. And it is also a safety engineering challenge if the GPU, or a portion of it, is a safety compliant item. The research contributions that target this topic are, generally, target agnostic, meaning that they are not specific for GPUs and, instead, the same techniques can be used for any SoC. Those techniques fall, for example, in the broad area of presilicon verification, where solutions build on either formal methods [103] (e.g., theorem proving, equivalence checking) or on emulation/simulation [149] (e.g., for timing and power management). The former, formal methods, are particularly suitable when digital properties (e.g., assuming perfect circuits) need to be proven. Instead, the latter, simulation and emulation, are used to verify analog properties such as circuit timing and power related features.

Finally, the hardware safety engineer designs the GPU-based electronics in compliance with safety standard requirements (e.g., IEC 61508-2 §7.4.6, §7.4.7) and follows the guidelines provided by the GPU safety-manual (if applicable) and GPU technical manuals, with a conservative and worst-case scenario approach. Therefore, the designer uses conservative margins and guardbands, and avoids optimizations used in the high-performance computing domain such as voltage reduction [148].

5.2 Device Software - Parallel Computing Languages and Frameworks

All the aforementioned general-purpose GPU programming models described in Section 3.4, present challenges for use in critical systems, since they violate software design and implementation guidelines used in their software development and certification [183]. In particular, all these programming models require dynamic GPU memory allocation and pointers, whose use is discouraged by several safety-critical standards and language subsets for critical systems [15] (e.g., IEC 61508-3 Table B.1., ISO 26262-6 Table 6). For this reason, Trompouki et al. [183] proposed

the use of a subset of Brook [37], a CUDA-like language, predecessor of CUDA and OpenCL. This subset, Brook Auto, is shown to be appropriate for the development of safety-critical GPU applications [37]. It is implemented using an open source, source-to-source compiler that automatically generates OpenGL ES 2 code, allowing the use of general-purpose computation on every GPU, including graphics GPUs. Trompouki et al. [184] also covered the concept of tool qualification of a GPGPU compiler according to ISO 26262 and demonstrated its effectiveness with an avionics case study [27].

Graphics GPU devices are used for example in avionics navigation instruments safety-critical systems (see Section 7.1) or in digital dashboards in modern cars [114]. For this reason, a subset of OpenGL ES has been brought apart as OpenGL SC [23], which is designed to comply with critical systems requirements. For example, OpenGL SC removes all the dynamic functionality of its predecessor, which can lead to runtime failures such as dynamic memory allocation or dynamic compilation of GPU code. Proprietary, certified OpenGL SC drivers exist for avionics and automotive highest criticality levels. However they are available only on few products from few manufacturers, mainly driven by prior certification credit and/or the safety features provided by the particular products and the willingness of the manufacturer to provide information for the development of the safety-critical drivers.

Finally, the OpenMP consortium has an OpenMP safety-critical working group, however, there is no published standard yet. Therefore, OpenMP solutions for critical systems exist only in research proposals [166, 173] and consider only CPUs. In fact, to our knowledge, there is neither support of OpenMP nor OpenACC for *embedded safety GPU* devices.

6 Independence of Execution

Independence of execution is required to avoid dangerous execution interference among integrated functions, which could lead to a dangerous failure. For example, in a shared resources device that executes several software applications, a safety software function may miss a critical deadline due to temporal interference among executed software functions/applications. This Section describes safety measures to achieve temporal independence (§6.1) and spatial independence (§6.2).

6.1 Temporal Independence

Olmedo et al. [141, 142] describe three major high-level sources of temporal interference, to be considered in the timing and WCET analysis of GPU device-based software application(s): (1) execution time of kernels taking into consideration technical aspects such as execution paths and branches; (2) CPU software application(s) interactions with the driver stack; and (3) temporal interference due to memory contention. For each of them, the detailed analysis of temporal interference sources becomes device and software application(s) specific and requires a deep and thorough technical analysis, often obscured by the lack of information, detected inconsistencies in provided documentation (e.g., [200, 144]) and partial understandability of the temporal behavior provided by 'reverse engineering' results (e.g., microbenchmark) [142, 100, 40, 144, 96, 18, 39]. For that purpose, additional detailed temporal interference sources should also be considered, e.g., (1) CUDA scheduling hierarchy, warp scheduling, built-in arbitration mechanisms [142]; (2) Linux temporal interference due to interrupts that lead to priority inversion [59], Linux kernel execution path variability [17]; (3) impact of memory copy operations among concurrently running jobs [142], register file bank conflicts, variable latency instruction write to register [100], and CUDA memory allocation [40].

Shared memory access is a major source of temporal interference among parallel software applications executing in both CPU cores, GPU(s) and among them. The use of common last level cache (e.g., L2) for the CPU cores and the GPU(s) can lead to undesired memory interference among parallel running software (e.g., self-eviction phenomena). In *discrete GPUs*, the parallel execution of several kernels can lead to private memory interference leading to execution delays, and the potentially uncontrollable conditional branching execution of a given kernel can delay the execution of other critical scheduled kernels. Besides, *embedded GPUs* inherit the previous interference sources amplified with the shared memory architecture between the CPU cores and GPU(s), based on a common memory interconnect, memory controller and the shared memory (e.g., DRAM).

As shared memory access strategies are key to maximize average performance, integrated arbitration mechanisms and components (e.g., memory interconnect, memory controller, caches) are often 'black box' components with proprietary and undisclosed documentation [46, 142]. The integrated strategies and components can potentially lead to a considerable negative impact on temporal predictability and WCET estimation, which in the absence of design documentation cannot usually be limited by pure analysis. Due to this, different research contributions aim at providing a qualitative analysis and characterization of the time impact of parallel shared memory access conflicts. For example, in the analysis provided by Cavichioli et al. [46] for Tegra K1 and X1 devices, the most relevant

memory contention components are identified (CPU cores shared L2 cache, memory interconnect, software-controlled coherency between CPU and GPU cores, private L2 caches and main memory controller access arbitration). As shown by their experimental results, the interference quantification depends on several device and software application(s) specific features such as the executed algorithms, considered parallel CPU and GPU activity (e.g., three competing CPU cores leads to 60-84% degradation), memory access pattern (e.g., random 3-12%, sequential 7-35%), unified or separated caches, memory transfer size (e.g., bigger or smaller than L1 cache size) and executed memory access functions (e.g., *memset* leads to 426% higher delay, *memcpy* 377%, UVM/CUDA kernel operations 226%). In the analysis described by Olmedo et al. [142], a lesson learned is that if a CUDA block is scheduled in the same group of SMs with another memory-bound kernel block, the CUDA block WCET can increase by an order of magnitude. Zia et al. [100] provide a microbenchmark of the Volta architecture, including global memory access latency measurements that are representative to understand the importance of shared memory management in the overall computing average performance. Authors show that the latency for the first access (L2 cache and Translation Look-aside Buffers (TLB) miss) is 1029 cycles, whereas the next data access (L1 cache hit latency) takes only 28 cycles. Other scenarios lead to 193 cycle latency for L1 cache miss and L2 cache hit, and 375 cycle latency for L1 cache miss and TLB hit. Thus, for a given safety-critical system, the temporal interference analysis, quantification and mitigation strategy becomes device and integrated software application(s) specific. Besides, custom-designed GPU devices (e.g., Tesla FSD [176]) can be tailored to specific software applications' execution and data-flow, which also imposes a shared memory access flow that limits the undesired temporal interference.

Concerning the software application, the design and implementation should consider using temporal independence techniques such as exclusive access policies and software virtualization, with WCET timing analysis and execution time variability reduction and management:

- Exclusive Access Policies: As for spatial independence, it is also recommended to establish exclusive access policies for safety function kernels. For example, the Multi-Process Service (MPS) middleware is a risk for temporal independence because, even with compute resource partitioning, shared-memory hierarchy conflicts among applications can lead to temporal interference such as a 10x read-write transactions slowdown [96]. For this reason, Jain et al. [96] provide a shared GPU cache partitioning solution based on cache coloring. Other partitioning schemes are also proposed by Janzen and Wu et al. [97, 194].
- Software Virtualization: As previously explained, software hypervisor-based virtualization can also be used to provide temporal independence [141] (see Section 8.2). For example, Capodieci et al. [41] describes the internals of the hypervisor-based solution found in NVIDIA's automotive platforms and a modification proposal to allow real-time scheduling of both compute and graphics GPU workloads in order to meet timing requirements.
- WCET: Many of the existing methods and tools for WCET and timing analysis used in classical single-core based safety-critical systems are not suitable for multi-core and GPU devices [32, 29, 7, 16], due to the device complexity and execution variability, complex shared resources temporal interference and limited available documentation [16]. The applicability of static methods is generally considered not feasible for GPU devices, but a technical option is the adaptation of methods such as Probabilistic Timing Analysis (PTA) [48, 29, 30], measurement-based execution time analysis [85] and hybrid methods thereof [33, 16]. Besides, several research contributions aim at analyzing component-specific contributions to the WCET estimates, e.g., L1 data cache [90], shared cache [89], warp scheduling [88].
- Execution Time Variability Reduction and Management: Execution time variability can be considerable with complex software applications. For example, the execution time variability of the Apollo AD software framework with Linux can be up to 21x (maximum) and 6.1x on average, with "highly variable timing behavior and arbitrary distributions" of internal software modules [16]. In order to reduce this variability and enable the real-time execution of algorithms, diverse techniques are proposed, e.g., scheduling [18, 41, 42, 67, 82, 84, 88, 142, 144, 200], memory scheduling and arbitration mechanisms [101, 70, 68, 109, 43, 141, 147], fine-grained CPU-GPU command offloading with Vulkan [47], and (device-specific) real-time neural networks software execution adaptation with a WCET performance model of the device [81]. Finally, additional case studies of interest that analyze the real-time characteristics of software applications implemented in GPUs and CUDA are: automotive sign detection application [146, 201] and power train application [79] that uses the GPU programming patterns described by Barbieri et al. [24].

Additionally, applications with energy, power consumption and thermal dissipation constraints (see Section 8.4) require the implementation of energy, power and thermal management techniques such as voltage scaling and Dynamic Voltage and Frequency Scaling (DVFS). As these techniques dynamically modify power supply voltages and clock frequencies, they can potentially change the computing capacity of the device and therefore the execution time of

software safety functions (temporal interference). See Fakih et al. [62] for further details and description of example applicable techniques.

Finally, even though reasonable measures are defined and implemented to avoid temporal interference, temporal diagnostics are required to check the fulfillment of safety time constraints, by means of techniques such as time watchdog, detecting unexpected interrupts and run-time monitoring (e.g., performance monitoring [25]) [153].

6.1.1 Component

Component specific safety measures can also be defined to safely manage temporal interference at component and device architecture level.

Processing Units As described by Alcaide et al. [14], the lack of public documentation, controllability and observability of SMs and the GPU device in general, limits significantly the achievement of some form of temporal independence. For example, the concurrent execution of kernels can lead to temporal interference (e.g., due to branch condition in one of the kernels).

Memory Techniques such as MemGuard memory controller, time-triggered scheduling and Predictable Execution Model (PREM) execution model [150, 109] apply to the case of GPU devices with adaptations such as GPUguard [70], SiT [43] and HePREM [68]. As a general rule, scheduling and memory-arbitration mechanisms [101, 70, 68, 109, 43], with associated experimental evaluations to verify them, can reduce temporal interference but by particularizing solutions for specific devices and software application(s).

Concerning caches, with the goal of reducing the temporal interference and improve temporal analysis, several GPU device-specific techniques are defined. For instance, minimization of self-eviction by prefetches managing [69], warp-based load/store reordering mechanism [87], L2 cache data locking [154], and L1 and shared data cache WCET impact estimation [90, 89].

Due to the fact that their data contents are known a priori, SPMs provide predictable latencies for accesses. This is relevant for GPUs where they are effectively used only if all threads finish more or less simultaneously. Hence, SPM, while posing the responsibility of managing their contents onto the end user, offer low and predictable latencies as opposed to cache memories. However, since they are still shared among multiple threads, but a lower number compared to the GPU L2 cache, they are also susceptible to contention from bank conflicts. For this reason, the accesses to the SPM have to be coordinated to minimize this contention [104].

6.2 Spatial Independence

Shared memory access techniques among the CPU cores and the GPU(s) are key for improving the average computational performance of a given GPU device. This shared memory is commonly managed either as a unified address space, partitioned or managed with unidirectional coherence between the CPU cores and the GPU(s) [155].

Several specialized components, such as Memory Management Unit (MMU), can be employed to underpin spatial independence in the GPU device. An MMU is a common component used to ensure exclusive access to shared addressable memories and peripherals [155, 128, 141]. The design of efficient MMU components [155] becomes a key challenge with the addition of required additional logic such as L1 cache data translation logic to prevent significant cache performance degradation (e.g., 20-50%) [155, 128]. In addition to this, cache/memory coherence management and diagnostics techniques shall be integrated. Previously described spatial independence mechanisms are commonly provided as silicon manufacturer-specific solutions with limited public information and documentation. Thus, spatial independence analysis requires device-specific analysis and configuration to ensure spatial independence among software functions executed in the CPU cores, GPU(s) and among them.

Device built-in hardware security technology (e.g., TrustZone) and virtualization technology required to deploy software hypervisors [114, 141, 84] can be utilized to underpin spatial independence among software partitions and shared resources. Moreover, the deployment of *soft GPUs* in FPGAs can make use of existing spatial isolation techniques [75].

Concerning the software application, the design and implementation should consider using spatial independence techniques such as:

- Exclusive Access Policies: Exclusive access policies for addressable components (shared resources) is a basic mitigation technique [111, 112]. As explained for SMs, safety function kernel(s) execution should be managed with exclusive access policies whenever feasible.

- Software Virtualization: Hypervisor based virtualization solutions can be used to provide spatial and temporal independence [84, 114, 141, 41, 174, 112] (see Section 8.2). A significant challenge faced by safety GPU devices with respect to virtualization, is due to the closed nature of their drivers. To allow the use of a GPU in a virtualized environment, a paravirtualized device driver is required. This means that the software should be aware that it is executed within a virtualized environment, to avoid unnecessary operations. However, since the source code of most of the GPU drivers is generally not available, such modifications are not possible. In certain cases, commercial real-time hypervisor-based operating systems offer support for specific GPU devices, in collaboration with the GPU vendors, e.g., SYSGO's PikeOS. In other cases, safety-critical GPU driver development companies, support certain GPU devices with specific RTOSes, e.g., CoreAVI with VxWorks.
- Memory Transfer and Coherency Mechanism: In addition to this, to avoid memory coherency errors, the usage of memory transfer mechanisms provided by programming languages and frameworks should be limited, e.g., CUDA UVM and OpenCL 2.0 SVM [46]. And whenever the device or the shared memory access technique does not provide required coherency mechanisms, ad-hoc software-controlled coherency is required to ensure spatial independence.

Finally, even though reasonable measures are defined and implemented to avoid spatial interference, spatial diagnostics are required to diagnose that data has not been modified by another element, using techniques such as cache and memory coherence diagnostic, periodic diagnostic of MMU and information redundancy (see for example multi-core device techniques [153]).

6.2.1 Component

Component specific safety measures can also be defined to safely manage temporal interference at component level. For example, the maximization of SMs computing capability utilization by allowing the concurrent execution of kernels from multiple processes [200, 136] is a potential risk for spatial independence, as the concurrent execution of other kernels could adversely interfere in the spatial (e.g., shared memory corruption) and temporal domain. Therefore, whenever feasible such computing resources should be managed with exclusive access policies for safety function kernels.

7 Domain-Specific Approaches

This section describes avionics and aerospace domain-specific approaches, where applicable safety standards do not consider IEC 61508 or ISO 26262 as reference safety standards. In both cases, due to small production volumes, the general strategy is to use available COTS GPU devices/IPs, and integrate software-based techniques and solutions to adapt them to their needs and safety standards requirements [106, 9, 36, 126, 27].

Nonetheless, both domains, and primarily the aerospace domain, must operate in harsh environments (e.g., radiation), with limited SWaP and heat dissipation capacity [106]. Thus, the selection of the appropriate GPU device/IP and system architecture (e.g., multi-core device and/or FPGA with GPU device) requires a detailed analysis and selection process [106, 9, 36].

7.1 Avionics Domain

Avionics safety-critical systems are subject to certification processes with domain-specific standards (e.g., DO-178C / EUROCAE ED-12C [1]). Nowadays, multi-core devices enable the potential evolution to the multi-Integrated Modular Avionics (IMA) architecture that supports the further integration of functions in a single computer [11]. Nonetheless, safety certification is a challenge and the updated ARINC 653 [19] imposes a hard restriction: single-core execution whenever a safety software partition is executed on a multi-core device. Besides, the CAST-32A [45] position paper provides some guidance by avionics certification authorities, but the practical applicability with COTS multi-core devices is limited due to temporal interference [10].

Due to this, the potential applicability of *computing GPU* devices and associated software programming languages and frameworks, for the development of avionics safety-critical systems implies even a higher technical challenge than for multi-core devices. Nonetheless, the high computational needs required for the development of next-generation avionics dependable and safety-critical systems [21] leads to analyze the potential applicability of *computing GPUs* by means of analysis and case studies [126, 27]. For example, collision avoidance demonstrator that combines image processing with inertial sensor data management, executed on a *GPU* as closed-loop control [126]; vision-based navigation with real-time graphics-based visualization and real-time image analysis calculations to estimate the rendez-vouz object location (OpenGL SC 2, Brook Auto/BRASIL) [27].

However, *graphics GPUs* have been used in critical systems for a long time for display tasks such as for navigation instruments and warning displays in modern aircraft [192, 27]. These displays are developed with avionics-grade *graphics GPUs* that comply with the certification guidelines already defined two decades ago for COTS graphical processors in airborne displays (CAST-29) [44, 27].

7.2 Space Domain

The increasing processing capacity required to handle massive data collection from sensors (e.g., optical sensing, infrared imaging) or critical autonomous operations (e.g., autonomous docking, exo-planetary missions) leads to the adoption of high-performance computing solutions such as GPU devices/IPs for in-orbit and deep space applications [36, 105, 106, 107, 74, 116, 162]. CoRoT was the first satellite with an integrated GPU for image compression [196]. Since then, several satellites have integrated GPUs [196] and the number is expected to increase as several analyzed space algorithms and software applications have been successfully ported to GPU devices [105, 106, 107].

The most common reliability and diagnostic coverage techniques used in the space domain are several of the previously described techniques, or their adaptations, with software-based implementations that complement available hardware built-in techniques. For example, DWC [106], information redundancy (e.g., ECC, ABFT) [36], software resources triple redundancy (e.g., file system) [9], fault tolerant middleware [36] and software-based mitigation techniques [9]. However, a major difference is that these systems operate in harsh environments (e.g., high radiation) that require the adoption of improved and higher number of hardening, fault tolerance and diagnostic techniques.

8 Links to other research topics and future trends

This section describes several additional research topics and future semiconductor industry trends to be considered in the development of GPU device-based safety-critical systems, which have been either referenced in the survey (e.g., hypervisor) or considered relevant to provide a wider perspective.

8.1 Heterogeneous High-Performance Computing Platforms

Heterogeneous high-performance computing platforms can deliver the high computing capacity required by next-generation safety-critical systems. For example, multi-core devices [153], FPGAs [31], GPUs, domain-specific accelerators [54] (e.g., Deep Neural Network (DNN) accelerator), Network Processing Units (NPUs), Tensor Processing Units (TPUs) and neuromorphic computing [172]. However, the on-chip integration of complex and computationally demanding software functions of different criticality is a challenge to be addressed to enable the usage of such innovative platforms for the development of safety-critical systems. Furthermore, the development of next-generation systems leads to novel research problems such as portability of artificial intelligence designs across heterogeneous high-performance computing platforms, and the efficient design and testing for redundant software implemented and deployed on diverse platforms [13].

8.2 Software Stack (Hypervisor, GPOS, middleware)

As described in Section 3.5, Burgio et al. propose a software stack for safety and real-time computing GPU device-based systems [38] that combines several software technologies outside the primary scope of the present survey:

- Virtualization technology: Hypervisor technology enables the safe integration of software partitions of different criticality onto multi-core [112, 11] and GPU [141, 84, 41, 38] devices. For further details, see for example [84, 11].
- Linux GPOS: There are several research initiatives towards developing Linux-based safety-critical systems [17] that could be potentially adapted to GPUs. Besides, Linux is also increasingly considered for critical systems, such as in the Automotive Grade Linux. Furthermore, there are modifications for the Linux kernel (real-time patches) to improve Linux-based systems timing behavior.
- Middlewares and domain-specific standards such as AUTOSAR [38]: For example, Bruhn et al. [36] use a Linux-based system, with COTS ML libraries such as TensorFlow, and a software stack that comprises a fault-tolerant commercial middleware.

8.3 Security / Cybersecurity

Deep learning techniques' success is mainly attributed to the rise of GPUs and their applications in tackling complex problems. Moreover, newer models require increasingly extensive amounts of manually tagged training and testing

datasets, which involves high development costs. Thus, security issues such as intellectual property protection and verifiable computation models are essential for successfully integrating GPUs in critical learning tasks.

These issues have been a matter of different categories of attacks (see survey [129]), such as information extraction using binary leaking or covert-side channel attacks that take advantage of cache information, memory access patterns timing information, electromagnetic signals, or power. This set of techniques has been able to extract input data, architecture and parameters of the model. The second category of attacks is related to model corruption using software/hardware fault injection and hardware Trojans designed with specific payloads and triggering systems for precisely controlling the exact moment for disturbing the model. Thirdly, buffer overflow, denial of service and malware attacks by using targeted software. Finally, a subset of specific attacks focused on inserting fake devices [181] in collaborative environments.

In addition to these problems, there are some additional challenges for protecting GPUs, such as limitations of CPU-based security solutions for controlling or monitoring the execution of GPUs, lack of documentation, lack of data erasure and vulnerabilities in virtualization scenarios.

8.4 Energy, Power and Thermal Management

The latest evaluations of high-performance computing platforms (multi-core devices, SoC-FPGA, GPU) show that GPUs provide the highest computing performance and lowest performance / W ratio, but at the cost of the highest power dissipation [116, 167]. However, limited power dissipation requirement is also important to reduce the need for low-reliability external cooling systems and thermal dissipation fans [152]. For example, Tesla FSD *integrated computing GPU* device design considers power dissipation efficiency [176]. In addition to this, providing a competitive computing performance / W, with a reduced overall power and energy consumption is a requirement for a subset of heterogeneous safety-critical systems such as space applications, remote railway signaling systems and portable medical systems [62, 105, 143]. The reconciliation of power-management and low power techniques with safety-critical systems and associated safety standards, requires specific techniques and solutions as described by Fakih et al. [62].

8.5 Future Semiconductor Industry Trend

The semiconductor industry has entered a maturity stage where less chip manufacturers can afford the economic and technological investments required to produce shrinking devices. This maturity stage leads to an industrial consolidation of both companies (e.g., merging, acquisition) and devices [52]. See for example the limited diversity of silicon manufacturers in Tables 5 and 6. This trend leads to a scenario of:

- *Limited diversity* of major silicon manufacturers, GPU devices and device architectural solutions. For example, in Table 6 (page 8) and Table 5 (page 8), it is possible to identify a reduced diversity of silicon manufacturers and CPU cores for *integrated GPUs* (ARM, x86).
- *Maximum average computing performance devices* that target multiple domains and applications with a single device [52] (economies of scale), where the safety-critical niche is too small to impose technical requirements such as temporal independence. In addition to this, to continue scaling with Moore's law, the exploitation of Single Instruction Multiple Data (SIMD) data parallelism with GPUs, NPUs, accelerators and FPGAs will further exacerbate this trend, and the worst-case performance is expected to deteriorate further [32].
- *Short life-cycle* of devices with continuously evolving device architectures to provide competitive maximum average performance devices every few years. Moreover, it is also possible to identify in Table 6 a constant upgrade/evolution of SM microarchitectures. Nonetheless, this jeopardizes the long time support and architectural stability required by many safety-critical applications [153], to reduce the impact of updates and re-certification of safety-critical systems with a lifetime that can exceed 30 years (e.g., railway, avionics).
- *Limited public documentation and software* shared by manufacturers, to protect the intellectual property of crucial hardware and software elements that provide a competitive advantage to achieve maximum average computing performance, which are also a critical source of risks to fulfill the described safety technical requirements (e.g., temporal independence).

9 Conclusion and Future Research Directions

This survey summarizes and categorizes selected research contributions that address GPU devices' random hardware failures, systematic failures and independence of execution for the development of GPU-based safety-critical systems.

All in all, there is a considerable research effort to pave the way towards the development and certification of GPU-based safety-critical systems.

GPU devices, and more specifically *hard embedded safety GPU* devices, have rapidly evolved into new application domains such as safety-critical systems in recent years. However, the provided solutions are not yet as mature as multi-core devices for safety-critical applications and they extend or bring new technical challenges such as WCET analysis [153] and safe parallel software programming. In the absence of qualified/certified GPU devices (beyond CPU core(s)), limited availability and support of safety-compliant GPU programming languages and qualified tools, safety-compliant software frameworks and underlying operating systems (e.g., Linux), the safety system developer needs to analyze, define and combine several techniques and safety measures for the development of a GPU-based safety-critical system.

Table 1 summarizes the described safety measures and techniques to address random hardware failures, systematic failures and independence of execution challenges. Due to the different nature of the addressed threats, most techniques and safety measures are challenge-specific. The advantages and disadvantages of the techniques proposed in the research contributions cannot usually be generalized due to high dependencies on the considered GPU device, microarchitecture (see Table 6), the case study and its software application programming. Therefore as explained in the survey, selecting and combining sufficient and cost-effective safety measures usually requires a GPU device and software application-specific analysis, definition and selection.

Most research contributions focus on random hardware failures (control and mitigation) and temporal independence techniques. This is not a coincidence. In a shared resources COTS GPU developed for maximum average performance, many diagnostics and spatial independence techniques can be reused from techniques already defined for multi-core devices, while novel or adapted techniques are required to achieve required reliability and temporal independence. Besides, as the integration of built-in hardware techniques can lead to performance degradation and increase cost and power consumption, software-based techniques are a common approach to use or complement available built-in hardware techniques. On top of that, the adaptation of high-performance computing domain techniques to the safety-critical domain should consider safety-standard compliance requirements such as *tool qualification*, avoid or safely manage *dynamic* behavior and *not recommended* techniques, and apply techniques/methods with sufficient *depth and rigor*.

Next-generation safety-critical systems require high-performance computing devices such as GPUs, which provide high computational performance, software parallelization and low computing power / W ratios. In this context, the developers of GPU device-based safety-critical systems must meet two conflicting constraints. On the one hand, an innovative and rapidly evolving GPU semiconductor industry continues to develop higher computational devices with continuously shrinking technologies. On the other hand, the conservative nature of functional safety standards with limited consideration of GPU devices (see recent developments in ISO 26262-11). Hence, there is a clear need for research initiatives on both fronts, "how things can be done" with available COTS GPU devices and "how things should be done" with the definition of novel approaches.

- The reliability wall: The development of highly integrated and powerful computational devices such as GPUs with relatively large shared memories, hundreds of computational units and complex, obscure and proprietary logic (e.g., warp scheduler), can lead to lower reliability devices as opposed to more simple processor devices traditionally used. This is a significant limitation because safety-critical systems shall comply with a given probability of failure range defined by the SIL/ASIL of the integrated safety function(s). This probability of failure (e.g., PFH) is independent of the integrated devices. So, GPU-based safety-critical systems should also meet these value ranges defining the 'reliability wall' that could limit the potential amount of used GPU features. For example, current reliability measurements for even simple algorithms [190] indicate that this limitation could endanger the practical usability of such devices for safety-critical systems. Finally, there is a need to simultaneously address reliability at all abstraction levels [127], with 'software-based reliability' techniques that complement built-in hardware techniques.
- Temporal independence: Shared memory is the most relevant temporal predictability threat for temporal independence and it is expected to deteriorate further [32]. So, there is a need to tackle the temporal independence challenge with the definition of adapted/novel techniques and WCET analysis methods to achieve temporal predictability, along with programming models, languages and scheduling solutions. Moreover, there is also a need to define novel GPU architectures and components, with associated software programming languages and frameworks that support time predictability and independence.
- Diagnostic Coverage: Software (dual) replication-based diagnostic techniques are commonly not computationally and energy wise-efficient. So, there is a need to define additional efficient diagnostic techniques, either with built-in hardware and/or software. There is also a need to define efficient diagnosis strategies, which whenever feasible, perform diagnosis in the lowest abstraction layer to reduce the integration

complexity at higher abstraction layers (e.g., software application partition) and/or software level device-agnostic diagnostics that reduce dependency with the underlying device [151].

In addition to this, research contributions concerning *soft GPUs* are limited (e.g., simulation-based fault-injection case studies in Table 5). Besides, as described by Kosmidis et al. [107], most open source GPUs are incomplete, obsolete and lack documentation. Nonetheless, this is a research opportunity to develop *soft GPUs* and open source GPUs that could tackle "how things should be done", where the "RISC-V movement can create opportunities for commercially-friendly open source GPUs" [107].

Finally, there are multiple additional challenges to be addressed [32, 153]. For example, programming languages and frameworks should comply with the systematic capability defined by safety standards (e.g., a subset of programming language) with the availability of qualified tools to develop safe, parallel and real-time software. In addition to this, software product line and modular certification strategies should be defined to reduce the impact of 'short life-cycle' devices and integrated ad-hoc techniques for a given device and software application. For that purpose, the availability of GPU device adapted safety hypervisor solutions and safety compliant GPOSs such as Linux, which seamlessly integrate software frameworks and hardware resources among software partitions, would provide a foundation for developing reusable software partitions. Finally, GPU devices and previously described languages, frameworks and tools should also enable the development of energy and power-efficient systems with reduced SWaP.

Acknowledgements

This work has been partially supported by the European Research Council with Horizon 2020 (grant agreements No. 772773 and 871465), the Spanish Ministry of Science and Innovation under grant PID2019-107255GB, the HiPEAC Network of Excellence and the Basque Government under grant KK-2019-00035. The Spanish Ministry of Economy and Competitiveness has also partially supported Leonidas Kosmidis with a Juan de la Cierva Incorporación postdoctoral fellowship (FJCI-2020-045931-I). We would also like to thank reviewers for taking the time and effort necessary to review the manuscript. We sincerely appreciate all valuable comments and suggestions, which helped us to improve the quality of the manuscript.

References

[1] DO-178C / EUROCAE ED-12C - software considerations in airborne systems and equipment certification., 2011.

[2] IEC 60050-192: International electrotechnical vocabulary (IEV) - part 192: Dependability, 2015.

[3] ISO 26262-11: Road vehicles - functional safety - part 11: Guideline on application of ISO 26262 to semiconductors, 2018.

[4] NVIDIA Xavier series system-on-chip - technical reference manual. Report DP-09253-002, NVIDIA, 2020.

[5] R-Car H3 / M3 - user manual (rev.0.10). Report, Renesas, 2020.

[6] M. Abdel-Majeed, W. Dweik, H. Jeon, and M. Annavaram. Warped-RE: Low-cost error detection and correction in GPUs. In *45th Annu. IEEE/IFIP Int. Conf. on Dependable Syst. and Netw.*, pages 331–342, 2015.

[7] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE Int. Symp. on Ind. Embedded Syst. (SIES)*, 2015.

[8] V. Abhyankar. Performance-cost analysis of software implemented hardware fault tolerance techniques. Report, University of Wisconsin Madison, 2010.

[9] C. Adams, A. Spain, J. Parker, M. Hevert, J. Roach, and D. Cotten. Towards an integrated GPU accelerated SoC as a flight computer for small satellites. In *IEEE Aerosp. Conf.*, pages 1–7, 2019.

[10] I. Agirre, J. Abella, M. Azkarate-Askasua, and F. J. Cazorla. On the tailoring of CAST-32A certification guidance to real COTS multicore architectures. In *12th IEEE Int. Symp. on Ind. Embedded Syst. (SIES)*, pages 1–8, 2017.

[11] H. Ahmadian, R. Obermaisser, and J. Perez. *Distributed Real-Time Architecture for Mixed-Criticality Systems*. CRC Press, Taylor & Francis Incorporated, 2018.

[12] S. Alcaide et al. Software-only based diverse redundancy for ASIL-D automotive applications on embedded HPC platforms. In *IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Syst. (DFT)*, pages 1–4, 2020.

[13] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella. High-integrity GPU designs for critical real-time automotive systems. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, pages 824–829, 2019.

[14] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella. Software-only diverse redundancy on GPUs for autonomous driving platforms. In *IEEE 25th Int. Symp. on On-Line Testing and Robust System Des. (IOLTS)*, pages 90–96, 2019.

[15] S. Alcaide, L. Kosmidis, H. Tabani, C. Hernandez, J. Abella, and F. J. Cazorla. Safety-related challenges and opportunities for GPUs in the automotive domain. *IEEE Micro*, 38(6):46–55, 2018.

[16] M. Alcon et al. Timing of autonomous driving software: Problem analysis and prospects for future solutions. In *IEEE Real-Time and Embedded Technol. and Appl. Symp. (RTAS)*, pages 267–280, 2020.

[17] Imanol Allende, Nicholas Mc Guire, Jon Perez, Lisandro G. Monsalve, and Roman Obermaisser. Towards Linux based safety systems - a statistical approach for software execution path coverage. *J. of Syst. Architecture*, 116:102047, 2021.

[18] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *IEEE Real-Time Syst. Symp. (RTSS)*, pages 104–115, 2017.

[19] ARINC 653 - avionics application software standard interface, 2015.

[20] J. Athavale, A. Baldovin, and M. Paulitsch. Trends and functional safety certification strategies for advanced railway automation systems. In *IEEE Int. Rel. Physics Symp. (IRPS)*, pages 1–7, 2020.

[21] J. Athavale et al. AI and reliability trends in safety-critical autonomous systems on ground and air. In *50th Annu. IEEE/IFIP Int. Conf. on Dependable Syst. and Netw. Workshops (DSN-W)*, pages 74–77, 2020.

[22] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. In *IEEE Trans. on Dependable and Secure Comput.*, volume 1, pages 11–33, 2004.

[23] Nakhoon Baek and Kuinam J. Kim. Design and implementation of OpenGL SC 2.0 rendering pipeline. *Cluster Computing*, 22(1):931–936, 2019.

[24] D. Barbieri, V. Cardellini, and S. Filippone. SIMPL: A pattern language for writing efficient kernels on GPGPU. In *IEEE/ACM 1st Int. Workshop on Softw. Eng. for High Performance Comput. in Sci.*, pages 38–45, 2015.

[25] Javier Barrera et al. On the reliability of hardware event monitors in MPSoCs for critical domains. In *Proc. of the 35th Annu. ACM Symp. on Applied Comput.*, page 580–589, 2020.

[26] S. Bauer, S. Köhler, K. Doll, and U. Brunsmann. FPGA-GPU architecture for kernel SVM pedestrian detection. In *IEEE Comput. Soc. Conf. on Comput. Vision and Pattern Recognition - Workshops*, pages 61–68, 2010.

[27] M. Benito et al. Comparison of GPU computing methodologies for safety-critical systems: An avionics case study. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, 2021.

[28] Alfredo Benso and P. Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Springer, 2003.

[29] Kostiantyn Berezovskyi, Fabrice Guet, Luca Santinelli, Konstantinos Bletsas, and Eduardo Tovar. Measurement-based probabilistic timing analysis for graphics processor units. Architecture of Comput. Syst. (ARCS), pages 223–236. Springer Int. Publishing, 2016.

[30] Kostiantyn Berezovskyi, Luca Santinelli, Konstantinos Bletsas, and Eduardo Tovar. WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *Proc. of the 22nd Int. Conf. on Real-Time Netw. and Syst.*, page 279–288, 2014.

[31] C. Bernardeschi, L. Cassano, and A. Domenici. SRAM-based FPGA systems for safety-critical applications: A survey on design standards and proposed methodologies. *J. Comput. Sci. Technol.*, 30(2):373–390, 2015.

[32] Marko Bertogna. A view on future challenges for the real-time community (key note). In *27th Int. Conf. on Real-Time Netw. and Syst. (RTNS)*, 2019.

[33] A. Betts and A. Donaldson. Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *25th Euromicro Conf. on Real-Time Syst.*, pages 193–202, 2013.

[34] Richard Bramley. Functional safety and the GPU. In *GPU Technol. Conf.* NVIDIA, 2017.

[35] C. Braun et al. A-ABFT: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units. In *44th Annu. IEEE/IFIP Int. Conf. on Dependable Syst. and Netw.*, pages 443–454, 2014.

[36] Fredrik C. Bruhn, Nandinbaatar Tsog, Fabian Kunkel, Oskar Flordal, and Ian Troxel. Enabling radiation tolerant heterogeneous GPU-based onboard data processing in space. *CEAS Space Journal*, 2020.

[37] 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, 2004.

[38] P. Burgio et al. A software stack for next-generation automotive systems on many-core heterogeneous platforms. In *Euromicro Conf. on Digit. System Des. (DSD)*, pages 55–59, 2016.

[39] A. J. Calderón et al. Understanding and exploiting the internals of GPU resource allocation for critical systems. In *IEEE/ACM Int. Conf. on Comput.-Aided Des. (ICCAD)*, pages 1–8, 2019.

[40] Alejandro J. Calderón et al. GMAI: Understanding and exploiting the internals of GPU resource allocation in critical systems. *ACM Trans. Embed. Comput. Syst.*, 19(5):Article 34, 2020.

[41] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru. Deadline-based scheduling for GPU with preemption support. In *IEEE Real-Time Syst. Symp. (RTSS)*, pages 119–130, 2018.

[42] Nicola Capodieci, Roberto Cavicchioli, and Marko Bertogna. NVIDIA GPU scheduling details in virtualized environments: work-in-progress. In *Proc. of the Int. Conf. on Embedded Softw.* IEEE Press, 2017.

[43] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. SiGAMMA: server based integrated GPU arbitration mechanism for memory accesses. In *Proc. of the 25th Int. Conf. on Real-Time Netw. and Syst.*, page 48–57, 2017.

[44] CAST. Use of COTS graphical processors (CGP) in airborne display systems (CAST-29). Report, 1997.

[45] CAST. Multi-core Processors - Position Paper CAST-32A. Report, 2016.

[46] R. Cavicchioli et al. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *IEEE Int. Conf. on Emerging Tech. and Factory Automat. (ETFA)*, pages 1–10, 2017.

[47] Roberto Cavicchioli et al. Novel methodologies for predictable CPU-to-GPU command offloading. In *31st Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2019.

[48] 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):Article 14, 2019.

[49] J. Chen, S. Li, and Z. Chen. GPU-ABFT: Optimizing algorithm-based fault tolerance for heterogeneous systems with GPUs. In *IEEE Int. Conf. on Netw., Architecture and Storage (NAS)*, pages 1–2, 2016.

[50] Jieyang Chen. *Fault Tolerant and Energy Efficient One-Sided Matrix Decompositions on Heterogeneous Systems with GPUs*. Thesis, 2019.

[51] Nevin Cini and Gulay Yalcin. A methodology for comparing the reliability of GPU-based and CPU-based HPCs. *ACM Comput. Surv.*, 53(1), 2020.

[52] G. Corradi. Tools, architectures and trends on industrial all programmable heterogeneous MPSoC (keynote). In *29th Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2017.

[53] David Cuesta. *Thermal Aware Design Techniques for Multiprocessor Architectures in Three Dimensions*. Thesis, 2014.

[54] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, 2020.

[55] D. Defour and E. Petit. GPUburn: A system to test and mitigate GPU hardware failures. In *Int. Conf. on Embedded Comput. Syst.: Architectures, Modeling, and Simulation (SAMOS)*, pages 263–270, 2013.

[56] J. Diaz, C. Muñoz-Caro, and A. Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Trans. on Parallel and Distrib. Syst.*, 23(8):1369–1386, 2012.

[57] M. Dimitrov, M. Mantor, and H. Zhou. Understanding software approaches for GPGPU reliability. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, pages 94–104, 2009.

[58] C. Ding, C. Karlsson, H. Liu, T. Davies, and Z. Chen. Matrix multiplication on GPUs with on-line fault tolerance. In *IEEE 9th Int. Symp. on Parallel and Distrib. Processing with Appl.*, pages 311–317, 2011.

[59] G. A. Elliott and J. H. Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *24th Euromicro Conf. on Real-Time Syst.*, pages 267–276, 2012.

[60] EN50128 - railway applications: Communication, signalling and processing systems - software for railway control and protection systems, 2011.

[61] Meinhard Erben, Wolf Günther, Tobias Sedlmeier, Dieter Lederer, and Klaus-Jürgen Amsler. Legal aspects of safety designed software development, especially under european law. In *3rd Eur. Embedded Real Time Softw. (ERTS)*, page 6, 2006.

[62] Maher Fakih et al. SAFEPOWER project: Architecture for safe and power-efficient mixed-criticality systems. *Microprocess. Microsyst.*, 52(C):89–105, 2017.

[63] B. Fang et al. GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications. In *IEEE Int. Symp. on Performance Anal. of Syst. and Softw. (ISPASS)*, pages 221–230, 2014.

[64] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. A systematic methodology for evaluating the error resilience of GPGPU applications. *IEEE Trans. on Parallel and Distrib. Syst.*, 27(12):3397–3411, 2016.

[65] B. Fang, J. Wei, K. Pattabiraman, and M. Ripeanu. Evaluating error resiliency of GPGPU applications. In *SC Companion: High Performance Comput., Netw. Storage and Anal.*, pages 1502–1503, 2012.

[66] Bo Fang, J. Wei, K. Pattabiraman, and M. Ripeanu. Towards building error resilient GPGPU applications. In *High Performance Comput., Netw. Storage and Anal. (SC)*, 2012.

[67] Tobias Fleig. *Fair scheduling of GPU computation time in virtualized environments*. Master thesis, 2016.

[68] B. Forsberg, L. Benini, and A. Marongiu. HePREM: Enabling predictable GPU execution on heterogeneous SoC. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, pages 539–544, 2018.

[69] B. Forsberg, L. Benini, and A. Marongiu. Taming data caches for predictable execution on GPU-based SoCs. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, pages 650–653, 2019.

[70] B. Forsberg, A. Marongiu, and L. Benini. GPUguard: Towards supporting a predictable execution model for heterogeneous SoC. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, pages 318–321, 2017.

[71] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman. CRUM: Checkpoint-restart support for CUDA's unified memory. In *Int. Conf. on Cluster Comput. (CLUSTER)*, 2018.

[72] L. B. Gomez et al. GPGPUs: How to combine high computational power with high reliability. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, pages 1–9, 2014.

[73] M. M. Goncalves, I. P. Lamb, P. Rech, R. M. Brum, and J. R. Azambuja. Improving selective fault tolerance in GPU register files by relaxing application accuracy. *IEEE Trans. on Nuclear Sci.*, 67(7):1573–1580, 2020.

[74] D. Gonzalez-Arjona and G. Furano. High-performance avionics solution for advanced and complex GNC systems for ADR (HIPNOS). In *TEC-ED & TEC-SW Final Presentation Days at ESA*.

[75] E. Grade, A. Hayek, and J. Börcsök. Implementation of a fault-tolerant system using safety-related Xilinx tools conforming to the standard IEC 61508. In *Int. Conf, on Syst. Reliability and Science (ICSRS)*, pages 78–83, 2016.

[76] J. A. Gunnels, D. S. Katz, E. S. Quintana-Orti, and R. A. Van de Gejin. Fault-tolerant high-performance matrix multiplication: theory and practice. In *Int. Conf. on Dependable Syst. and Netw. (DSN)*, pages 47–56, 2001.

[77] M. Gupta and others. Compiler techniques to reduce the synchronization overhead of GPU redundant multithreading. In *54th ACM/EDAC/IEEE Des. Automat. Conf. (DAC)*, pages 1–6, 2017.

[78] S. K. S. Hari et al. SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation. In *IEEE Int. Symp. on Performance Anal. of Syst. and Softw. (ISPASS)*, pages 249–258, 2017.

[79] C. Hartmann, R. Mader, L. Michel, C. Ebert, and U. Margull. Massive parallelization of real-world automotive real-time software by GPGPU. In *30th Int. Conf. on Architecture of Comput. Syst. (ARCS)*, pages 1–8, 2017.

[80] Troels Henriksen. Bounds checking on GPU. In *Int. Symp. on High-level Parallel Programming and Appl. (HLPP)*, 2020.

[81] S. Heo, S. Cho, Y. Kim, and H. Kim. Real-time object detection system with multi-path neural networks. In *IEEE Real-Time and Embedded Technol. and Appl. Symp. (RTAS)*, pages 174–187, 2020.

[82] Chapel Hill. *Real-time scheduling for GPUs with applications in advanced automotive systems*. Thesis, 2015.

[83] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Comput.*, 41(7):33–38, 2008.

[84] Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. GPU virtualization and scheduling methods: A comprehensive survey. *ACM Comput. Surv.*, 50(3), 2017.

[85] A. Horga, S. Chattopadhyay, P. Eles, and Z. Peng. Measurement based execution time analysis of GPGPU programs via SE+GA. In *21st Euromicro Conf. on Digit. System Des. (DSD)*, pages 30–37, 2018.

[86] Lan Huang, Da-Lin Li, Kang-Ping Wang, Teng Gao, and Adriano Tavares. A survey on performance optimization of high-level synthesis tools. *J. of Comput. Sci. and Technol.*, 35(3):697–720, 2020.

[87] Y. Huangfu and W. Zhang. Warp-based load/store reordering to improve GPU data cache time predictability and performance. In *IEEE 19th Int. Symp. on Real-Time Distrib. Comput. (ISORC)*, pages 166–173, 2016.

[88] Y. Huangfu and W. Zhang. Static WCET analysis of GPUs with predictable warp scheduling. In *IEEE 20th Int. Symp. on Real-Time Distrib. Comput. (ISORC)*, pages 101–108, 2017.

[89] Y. Huangfu and W. Zhang. WCET analysis of the shared data cache in integrated CPU-GPU architectures. In *IEEE High Performance Extreme Comput. Conf. (HPEC)*, pages 1–7, 2017.

[90] Y. Huangfu and W. Zhang. WCET analysis of GPU L1 data caches. In *IEEE High Performance extreme Comput. Conf. (HPEC)*, pages 1–7, 2018.

[91] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. *SIGPLAN Not.*, 47(4):111–122, 2012.

[92] IEC 61508(-1/7): Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[93] ISO 10218-1/2:2011 - robots and robotic devices - safety requirements for industrial robots, 2011.

[94] ISO 13849-(1/2): Safety of machinery - safety-related parts of control systems, 2015.

[95] ISO 26262(-1/11) road vehicles - functional safety, 2018.

[96] S. Jain, I. Baek, S. Wang, and R. Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *Proc. of the IEEE Real-Time and Embedded Technol. and Appl. Symp., (RTAS)*, pages 29–41, 2019.

[97] J. Janzen, D. Black-Schaffer, and A. Hugo. Partitioning GPUs for Improved Scalability. In *Proc. - Symp. on Comput. Architecture and High Performance Comput.*, pages 42–49, 2016.

[98] X. Jean, M. Gatti, G. Berthon, and M. Fumey. The use of multicore processors in airborne systems (EASA 2011.C31). Report, EASA, Thales Avionics, 2011.

[99] H. Jeon and M. Annavaram. Warped-DMR: Light-weight error detection for GPGPU. In *45th Annu. IEEE/ACM Int. Symp. on Microarchitecture*, pages 37–47, 2012.

[100] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. Report, CoRR, 2018.

[101] Adwait Jog et al. Anatomy of GPU memory system for multi-application execution. In *Proc. of the Int. Symp. on Memory Syst.*, page 223–234, 2015.

[102] C. Kalra, F. Previlon, X. Li, N. Rubin, and D. Kaeli. PRISM: Predicting resilience of GPU applications using statistical methods. In *Int. Conf. for High Performance Comput., Netw., Storage and Anal.*, pages 866–879.

[103] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, apr 1999.

[104] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2016.

[105] L. Kosmidis et al. GPU4S: Embedded GPUs in space. In *22nd Euromicro Conf. on Digit. System Des. (DSD)*, pages 399–405, 2019.

[106] Leonidas Kosmidis et al. GPU4S: Embedded GPUs in space - latest project updates. *Microprocessors and Microsystems*, 77:103143, 2020.

[107] Leonidas Kosmidis et al. GPU4S: Major project outcomes, lessons learnt and way forward. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, 2021.

[108] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. NVIDIA ampere architecture in-depth, 2020. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

[109] Flavio Kreiliger, Joel Matejka, Michal Sojka, and Zdenek Hanzálek. Experiments for predictable execution of GPU kernels. *OSPERT*, page 23, 2019.

[110] Huang Kuang-Hua and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. on Comput.*, C-33(6):518–528, 1984.

[111] A. Larrucea, J. Perez, and R. Obermaisser. A modular safety case for an IEC 61508 compliant generic COTS processor. In *IEEE Int. Conf. on Comput. and Inform. Technol.; Ubiquitous Comput. and Commun.; Dependable, Autonomic and Secure Comput.; Pervasive Intell. and Comput. (CIT/IUCC/DASC/PICom)*, pages 1788–1795, 2015.

[112] Asier Larrucea, Jon Perez, Irune Agirre, Vicent Brocal, and Roman Obermaisser. A modular safety case for an IEC-61508 compliant generic hypervisor. In *Euromicro Conference on Digital System Design (DSD)*, page 4. CPS, 2015.

[113] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13, jun 2009.

[114] C. Lee et al. VADI: GPU virtualization for an automotive platform. *IEEE Trans. on Ind. Inform.*, 12(1):277–290, 2016.

[115] J. Leng et al. Asymmetric resilience: Exploiting task-level idempotency for transient error recovery in accelerator-based systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 44–57.

[116] George Lentaris et al. High-performance embedded computing in space: Evaluation of platforms for vision-based navigation. *J. of Aerosp. Inf. Syst.*, 15(4):178–192, 2018.

[117] G. Li, K. Pattabiraman, C. Cher, and P. Bose. Understanding error propagation in GPGPU applications. In *Proc. of the Int. Conf. for High Performance Comput., Netw., Storage and Anal. (SC)*, pages 240–251, 2016.

[118] Guanpeng Li et al. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proc. of the Int. Conf. for High Performance Comput., Netw., Storage and Anal.*, 2018.

[119] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang. Resiliency of automotive object detection networks on GPU architectures. In *IEEE Int. Test Conf. (ITC)*, pages 1–9, 2019.

[120] A. Mahmoud et al. Optimizing software-directed instruction replication for GPU error detection. In *Int. Conf. for High Performance Comput., Netw., Storage and Anal. (SC)*, pages 842–854, 2018.

[121] Riccardo Mariani. Challenges in AI/ML for safety critical systems (key note). In *32nd IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Syst. (DFT)*. NVIDIA, 2019.

[122] I. Martinez et al. *Safety Certification of Mixed-Criticality Systems*. CRC Press, 2018.

[123] C. Di Martino et al. Lessons learned from the analysis of system failures at petascale: The case of Blue Waters. In *44th IEEE/IFIP Int. Conf. on Dependable Syst. and Networks*, pages 610–621, 2014.

[124] N. Maruyama, A. Nukada, and S. Matsuoka. A high-performance fault-tolerant software framework for memory on commodity GPUs. In *IEEE Int. Symp. on Parallel & Distrib. Processing (IPDPS)*, pages 1–12, 2010.

[125] Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. Software-based ECC for GPUs. Report, 2009.

[126] L. Mejias, J. Lai, J. J. Ford, and P. O' Shea. Demonstration of closed-loop airborne sense-and-avoid using machine vision. *IEEE Aerosp. and Electron. Syst. Mag.*, 27(4):4–7, 2012.

[127] S. Mittal and J. S. Vetter. A survey of techniques for modeling and improving reliability of computing systems. *IEEE Trans. on Parallel and Distrib. Systems*, 27(4):1226–1238, 2016.

[128] Sparsh Mittal. A survey of techniques for architecting TLBs. *Concurrency and Computation Practice and Experience*, 29(10), 2017.

[129] Sparsh Mittal, S. B. Abhinaya, Manish Reddy, and Irfan Ali. A survey of techniques for improving security of GPUs. *J. of Hardware and Syst. Secur.*, 2(3):266–285, 2018.

[130] Sparsh Mittal and Jeffrey S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), 2015.

[131] R. Nane et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. on Comput.-Aided Des. of Integrated Circuits and Syst.*, 35(10):1591–1604, 2016.

[132] R. Nathan and D. J. Sorin. Argus-G: Comprehensive, low-cost error detection for GPGPU cores. *IEEE Comput. Architecture Letters*, 14(1):13–16, 2015.

[133] W. Nedel, F. Kastensmidt, and J. R. Azambuja. Implementation and experimental evaluation of a CUDA core under single event effects. In *15th Latin American Test Workshop (LATW)*, pages 1–4, 2014.

[134] B. Nie et al. Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities. In *IEEE 25th Int. Symp. on Modeling, Anal., and Simulation of Comput. and Telecommun. Syst. (MASCOTS)*, pages 22–31, 2017.

[135] A. Nukada, H. Takizawa, and S. Matsuoka. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. In *IEEE Int. Symp. on Parallel and Distrib. Processing Workshops and Phd Forum*, pages 104–113, 2011.

[136] NVIDIA. Multi-Process Service. Report, NVIDIA, 2020.

[137] NVIDIA Corporation. CUDA for Tegra. https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html.

[138] D. A. G. Oliveira et al. Modern GPUs radiation sensitivity evaluation and mitigation through duplication with comparison. *IEEE Trans. on Nuclear Sci.*, 61(6):3115–3122, 2014.

[139] D. A. G. Oliveira, P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro. GPGPUs ECC efficiency and efficacy. In *IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Syst. (DFT)*, pages 209–215, 2014.

[140] D. A. G. Gonçalves de Oliveira, L. L. Pilla, T. Santini, and P. Rech. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. *IEEE Trans. on Comput.*, 65(3):791–804, 2016.

[141] I. S. Olmedo, N. Capodieci, and R. Cavicchioli. A perspective on safety and real-time issues for GPU accelerated ADAS. In *44th Annu. Conf. of the IEEE Ind. Electron. Soc. (IECON)*, pages 4071–4077, 2018.

[142] I. S. Olmedo et al. Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *IEEE Real-Time and Embedded Technol. and Appl. Symp. (RTAS)*, pages 213–225, 2020.

[143] M. Ottavi, D. Gizopoulos, and S. Pontarelli. *Dependable Multicore Architectures at Nanoscale*. Springer, 2018.

[144] Nathan Otterness and James Anderson. Exploring AMD GPU scheduling details by experimenting with "worst practices". In *Int. Conf. on Real-Time Netw. and Syst. (RTNS)*, 2021.

[145] Nathan Otterness and James H. Anderson. AMD GPUs as an alternative to NVIDIA for supporting real-time workloads. In *32nd Euromicro Conf. on Real-Time Syst. (ECRTS)*, pages 10:1–10:23, 2020.

[146] Nathan Otterness et al. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *IEEE Real-Time and Embedded Technol. and Appl. Symp. (RTAS)*, pages 353–364, 2017.

[147] Nathan Otterness, Ming Yang, Tanya Amert, James Anderson, and F Donelson Smith. Inferring the scheduling policies of an embedded CUDA GPU. In *Workshop on Operating Syst. Platforms for Embedded Real Time Syst. Appl. (OSPERT)*, 2017.

[148] G. Papadimitriou et al. Exceeding conservative limits: A consolidated analysis on modern hardware margins. *IEEE Transactions on Device and Materials Reliability*, 20(2):341–350, 2020.

[149] Savita Patil et al. Survey of memory, timing, and power management verification methods for multi-core processors. In *IEEE 10th Annu. Inf. Tech., Electron. and Mobile Communication Conf. (IEMCON)*, pages 0110–0119, 2019.

[150] R. Pellizzoni et al. A predictable execution model for COTS-based embedded systems. In *17th IEEE Real-Time and Embedded Technol. and Appl. Symp.*, pages 269–279, 2011.

[151] J. Perez et al. A safety certification strategy for IEC-61508 compliant industrial mixed-criticality systems based on multicore partitioning. In *17th Euromicro Conf. on Digit. Syst. Des. (DSD)*, pages 394–400, 2014.

[152] J. Perez et al. *A safety concept for an IEC 61508 compliant fail-safe wind power mixed-criticality embedded system based on multi-core partitioning*, volume 9111 of *Lecture Notes in Comput. Sci.* Springer Int. Publishing, 2015.

[153] 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), 2020.

[154] J. Picchi and W. Zhang. Impact of L2 cache locking on GPU performance. In *SoutheastCon*, pages 1–4, 2015.

[155] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. In *Proc. of the 19th Int. Conf. on Architectural support for programming languages and operating Syst.*, page 743–758, 2014.

[156] L. L. Pilla, P. Rech, F. Silvestri, C. Frost, P. O. A. Navaux, M. S. Reorda, and L. Carro. Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. *IEEE Trans. on Nuclear Sci.*, 61(4):1874–1880, 2014.

[157] F. G. Previlon et al. Combining architectural fault-injection and neutron beam testing approaches toward better understanding of GPU soft-error resilience. In *IEEE 60th Int. Midwest Symp. on Circuits and Syst.*, pages 898–901, 2017.

[158] F. G. Previlon et al. Evaluating the impact of execution parameters on program vulnerability in GPU applications. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, pages 809–814, 2018.

[159] F. G. Previlon et al. A comprehensive evaluation of the effects of input data on the resilience of GPU applications. In *IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Syst. (DFT)*, pages 1–6, 2019.

[160] F. G. Previlon et al. Characterizing and exploiting soft error vulnerability phase behavior in GPU applications. *IEEE Trans. on Dependable and Secure Comput.*, pages 1–1, 2020.

[161] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro. Impact of GPUs parallelism management on safety-critical and HPC applications reliability. In *44th Annu. IEEE/IFIP Int. Conf. on Dependable Systems and Netw.*, pages 455–466, 2014.

[162] I. Rodriguez et al. An on-board algorithm implementation on an embedded GPU: A space case study. In *Des., Automat. & Test in Europe Conf. & Exhibition (DATE)*, pages 1718–1719, 2020.

[163] J. E. Rodriguez Condia et al. Analyzing the sensitivity of GPU pipeline registers to single events upsets. In *IEEE Comput. Soc. Annu. Symp. on VLSI (ISVLSI)*, pages 380–385, 2020.

[164] J. E. Rodriguez Condia et al. A dynamic hardware redundancy mechanism for the in-field fault detection in cores of GPGPUs. In *23rd Int. Symp. on Des. and Diagnostics of Electron. Circuits & Syst. (DDECS)*, pages 1–6, 2020.

[165] Josie E. Rodriguez Condia, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone. FlexGripPlus: An improved GPGPU model to support reliability analysis. *Microelectronics Reliability*, 109:113660, 2020.

[166] Sara Royuela, Alejandro Duran, Maria A. Serrano, Eduardo Quiñones, and Xavier Martorell. *A Functional Safety OpenMP for Critical Real-Time Embedded Systems*. Springer Int. Publishing, 2017.

[167] M. Salim, A. O. Akkirman, M. Hidayetoglu, and L. Gurel. Comparative benchmarking: matrix multiplication on a multicore coprocessor and a GPU. In *Computational Electromagnetics Int. Workshop (CEM)*, pages 1–2, 2015.

[168] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech. Analyzing and increasing the reliability of convolutional neural networks on GPUs. *IEEE Trans. on Rel.*, 68(2):663–677, 2019.

[169] F. Fernandes Santos et al. Evaluation and mitigation of soft-errors in neural network-based object detection in three GPU architectures. In *47th Annu. IEEE/IFIP Int. Conf. on Dependable Syst. and Netw. Workshops (DSN-W)*, pages 169–176, 2017.

[170] Fernando Fernandes dos Santos, Luigi Carro, and Paolo Rech. Kernel and layer vulnerability factor to evaluate object detection reliability in GPUs. *IET Comput. & Digit. Techniques*, 2019.

[171] Bülent Sari. *Fail-operational Safety Architecture for ADAS/AD Systems and a Model-driven Approach for Dependent Failure Analysis*. Springer Vieweg, 2020.

[172] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. A survey of neuromorphic computing and neural networks in hardware, 2017.

[173] Maria A. Serrano, Sara Royuela, and Eduardo Quiñones. *Towards an OpenMP Specification for Critical Real-Time Syst.* Springer Int. Publishing, 2018.

[174] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. GPUvm: why not virtualizing GPUs at the hypervisor? In *USENIX Annu. Technical Conf.*, page 109–120. USENIX Association, 2014.

[175] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A checkpoint/restart tool for CUDA applications. In *Int. Conf. on Parallel and Distrib. Comput., Appl. and Technologies*, pages 408–413, 2009.

[176] E. Talpes et al. Compute solution for Tesla's full self-driving computer. *IEEE Micro*, 40(2):25–35, 2020.

[177] J. Tan, N. Goswami, T. Li, and X. Fu. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *IEEE Int. Symp. on Workload Characterization (IISWC)*, pages 226–235, 2011.

[178] Jingweijia Tan and Xin Fu. RISE: improving the streaming processors reliability against soft errors in GPGPUs. In *Proc. of the 21st Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, page 191–200, 2012.

[179] D. Tiwari et al. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *IEEE 21st Int. Symp. on High Performance Comput, Architecture (HPCA)*, pages 331–342, 2015.

[180] Nguyen Toan, Tatsuo Nomura, Hideyuki Jitsumoto, Naoya Maruyama, Toshio Endo, and Satoshi Matsuoka. MPI-CUDA applications checkpointing. Report, IPSJ SIG Technical Report, 2010.

[181] Assia Tria and Hamid Choukri. Invasive Attacks. *Encyclopedia of Cryptography and Security, 2*, pages Pages 623–629, 2011.

[182] M. M. Trompouki et al. An open benchmark implementation for multi-CPU multi-GPU pedestrian detection in automotive systems. In *IEEE/ACM Int. Conf. on Comput.-Aided Des. (ICCAD)*, pages 305–312, 2017.

[183] Matina Maria Trompouki and Leonidas Kosmidis. Brook auto: high-level certification-friendly programming for GPU-powered automotive systems. In *Proc. of the 55th Annu. Des. Automat. Conf.*, 2018.

[184] Matina Maria Trompouki and Leonidas Kosmidis. BRASIL: A high-integrity GPGPU toolchain for automotive systems. In *37th IEEE Int. Conf. on Comput. Des. (ICCD)*, pages 660–663. IEEE, 2019.

[185] I. Troxel, J. Schaefer, M. Gruber, and P. Gauvin. Summary of radiation test data for several GPUs. Report, 2019.

[186] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler. NVBitFI: Dynamic fault injection for GPUs. In *51st Annu. IEEE/IFIP Int. Conf. on Dependable Syst. and Netw. (DSN)*, pages 284–291.

[187] S. Tselonis and D. Gizopoulos. GUFI: A framework for GPUs reliability assessment. In *IEEE Int. Symp. on Performance Anal. of Syst. and Softw. (ISPASS)*, pages 90–100, 2016.

[188] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A simulation framework for CPU-GPU computing. In *21st Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344, 2012.

[189] A. Vallero, D. Gizopoulos, and S. Di Carlo. SIFI: AMD southern islands GPU microarchitectural level fault injector. In *IEEE 23rd Int. Symp. on On-Line Testing and Robust System Des. (IOLTS)*, pages 138–144, 2017.

[190] A. Vallero, S. Tselonis, D. Gizopoulos, and S. Di Carlo. Multi-faceted microarchitecture level reliability characterization for NVIDIA and AMD GPUs. In *IEEE 36th VLSI Test Symp. (VTS)*, pages 1–6, 2018.

[191] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *Int. Symp. on Comput. Architecture (ISCA)*, pages 73–84, 2014.

[192] B. X. Wang. Detecting hazardously misleading information on safety-critical displays. In *IEEE/AIAA 38th Digit. Avionics Syst. Conf. (DASC)*, pages 1–5, 2019.

[193] Zhang Wangyuan and Li Tao. Microarchitecture soft error vulnerability characterization and mitigation under 3D integration technology. In *41st IEEE/ACM Int. Symp. on Microarchitecture*, pages 435–446, 2008.

[194] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proc. of the Int. Conf. on Supercomputing*, volume 2015-June, pages 119–130, 2015.

[195] H. Wunderlich, C. Braun, and S. Halder. Efficacy and efficiency of algorithm-based fault-tolerance on GPUs. In *IEEE 19th Int. On-Line Testing Symp. (IOLTS)*, pages 240–243.

[196] Edward Wyrwas. Body of knowledge for graphics processing units (GPUs). Report, NASA, 2018.

[197] G. Xie, Y. Li, Y. Han, Y. Xie, G. Zeng, and R. Li. Recent advances and future trends for automotive functional safety design methodologies. *IEEE Trans. on Ind. Inform.*, 16(9):5629–5642, 2020.

[198] Xilinx. Device reliability report (UG116) - second half 2020. Report, Xilinx, 2020.

[199] Xin-Hai Xu, Xue-Jun Yang, Jing-Ling Xue, Yu-Fei Lin, and Yi-Song Lin. PartialRC: A partial recomputing method for efficient fault recovery on GPGPUs. *J. of Comput. Sci. and Technol.*, 27(2):240–255, 2012.

[200] M. Yang et al. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *30th Euromicro Conf. on Real-Time Systems (ECRTS)*, volume 106, pages 20:1–20:21, 2018.

[201] Ming Yang et al. Making OpenVX really "real time". In *IEEE Real-Time Syst. Symp. RTSS*, pages 80–93, 2018.

[202] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *IEEE Int. Parallel & Distrib. Processing Symp.*, pages 287–300, 2011.

[203] Junko Yoshida. *Unveiled: BMW's Scalable AV Architecture*. TechOnline, IEEE, 2020.

[204] Jin Zhang and Jingyue Li. Testing and verification of neural-network-based safety-critical control software: A systematic literature review. *Inf. and Softw. Technol.*, 123:106296, 2020.