High-Integrity GPU Designs for Critical Real-Time Automotive Systems

Sergi Alcaide[‡,†], Leonidas Kosmidis[†], Carles Hernandez[†], Jaume Abella[†]
[‡] Universitat Politècnica de Catalunya (UPC) [†] Barcelona Supercomputing Center (BSC)

***Abstract*—Autonomous Driving (AD) imposes the use of high-performance hardware, such as GPUs, to perform object recognition and tracking in real-time. However, differently to the consumer electronics market, critical real-time AD functionalities require a high degree of resilience against faults, in line with the automotive ISO26262 functional safety standard requirements. ISO26262 imposes the use of some source of independent redundancy for the most critical functionalities so that a single fault cannot lead to a failure, being dual core lockstep (DCLS) with diversity the preferred choice for computing devices. Unfortunately, GPUs do not support diverse DCLS by construction, thus failing to meet ISO26262 requirements efficiently.**

In this paper we propose lightweight modifications to GPUs to enable diverse DCLS for critical real-time applications without diminishing their performance for non-critical applications. In particular, we show how enabling specific mechanisms for software-controlled kernel scheduling in the GPU, allows guaranteeing that redundant kernels can be executed in different resources so that a single fault cannot lead to a failure, as imposed by ISO26262. Our results on a GPU simulator and an NVIDIA GPU prove the viability of the approach and its effectiveness on high-performance GPU designs needed for AD systems.

I. Introduction

The advent of autonomous driving (AD) makes automotive industry embrace high-performance hardware such as accelerators to execute performance-hungry tasks (e.g. object recognition and tracking) timely. However, while those accelerators have been widely deployed in the consumer electronics market, where performance within given power and thermal envelopes is the main concern, critical real-time systems, such as those in AD, pose a set of different challenges related to functional safety. In particular, safety-related automotive systems (e.g. braking, steering, etc), which include most AD functionalities, need to meet specific requirements described in the ISO26262 functional safety standard [1] to be deployed in cars. Those requirements, which mostly relate to the ability of the system to detect faults and prevent hazardous situations, impose strict verification and validation (V&V) requirements on the system and its components thereof. Hence, high-performance accelerators deployed in cars for AD must adhere to those requirements, which needs to be conveniently proven.

In the context of ISO26262, functionalities are classified in different Automotive Safety Integrity Levels (ASIL) based on the type of hazard they can cause, their severity, their exposure and the controllability upon a failure. In particular, safety-related functionalities are ranked from ASIL-D (the highest integrity level) to ASIL-A (the lowest), being ASIL-D components involved in ASIL-D functionalities those subject to the strictest V&V processes.

GPUs are becoming the most popular accelerator for AD, and they are already included in specific AD commercial platforms, such as RENESAS R-Car H3 [2] and NVIDIA Xavier [3] platforms. Those platforms include general purpose microcontrollers (e.g. ARM or Infineon cores) proven ASIL-D capable, as well as high-performance accelerators whose adherence to ASIL-D must also be proven so that they can perform AD-related activities. In particular, as detailed in ISO26262, ASIL-D systems must not allow a single fault lead the system to a hazardous situation. Appropriate safety measures include some form of independent redundancy, thus ensuring that a single fault will not lead redundant elements to identical erroneous outputs. For instance, Error Detection and/or Correction Codes are often used for storage and communication interfaces, whereas Dual Core LockStep (DCLS) with some source of diversity (e.g. staggered execution) is used for computation elements so that a single fault affecting all redundant components (e.g. a voltage droop) does not cause them to fail identically. In the case of GPUs, they have already been proven ASIL-B compliant, but, to our knowledge, ASIL-D compliance has only been achieved by implementing expensive independent redundancy means. In particular, either full system replication or heterogeneous implementations are used. The former, for instance, performs object recognition based on cameras and LIDAR, with different software implementations and, potentially, different hardware support. The latter, for instance, performs object recognition based only on cameras, but software is implemented and deployed for different accelerators (e.g. a GPU and a Deep Neural Network – DNN – accelerator) [4]. In both cases, design and V&V costs are duplicated, which is against efficiency and costs. For instance, these approaches clash with that followed for ASIL-D microcontrollers, which build upon diverse DCLS. Hence, it is critically important enabling some form of diverse DCLS on GPUs so that ASIL-D compliance can be achieved without needing to design and certify multiple heterogeneous software and hardware components.

This paper tackles this challenge by identifying the main requirements to enable ASIL-D compliance for Commercial Off-The-Shelf (COTS) GPUs, assessing to what extent they have the potential to meet ASIL-D requirements, and providing a set of lowly-intrusive modifications that allow adhering to ASIL-D requirements without diminishing their performance for non-safety-related functionalities. Those modifications allow GPU vendors to reuse their designs avoiding a significant increase of their Non-Recurring Expenses (NRE). In particular, we perform our analysis on an NVIDIA COTS GPU, implement modifications on a GPU simulator where we can assess both performance and ASIL-D compliance, and evaluate what the performance impact would be on the COTS GPU.

II. Background on ISO26262

A. ASIL decomposition

As explained before, safety-relevant components are classified from ASIL-D to ASIL-A. Additionally, non-safety-related components are regarded as QM (Quality Managed). In the context of ISO26262, the safety level is attached to systems based on their safety requirements and a hazard and risk

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

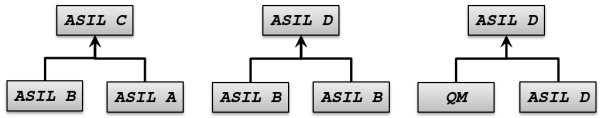

Fig. 1: Examples of ASIL decomposition.

analysis. Then, such safety level is propagated to the different components following some rules. For instance, the default rule consists of validating all components for the same ASIL as the higher level item where they are integrated. However, since increasingly higher ASIL have increasingly higher design and V&V costs, alternative approaches are followed based on what is usually referred to as *ASIL decomposition*.

Under a given ASIL, some specific diagnostic coverage must be achieved and some random failure rates are deemed as acceptable, being coverage and failure rates more stringent for the highest ASIL. Since reaching certain coverage levels and failure rates may impose excessive cost (or simply be unreachable), specific ASIL levels can be reached with the appropriate combination of lower ASIL components. This is illustrated with some examples in Figure 1. For instance, ASIL levels can be *added* as long as components provide independent redundancy, so two independent redundant ASIL-A and ASIL-B components allow reaching ASIL-C, or two ASIL-B ones allow reaching ASIL-D. The latter corresponds to the case of DCLS for cores in the microcontroller, which are individually certified for ASIL-B operation, and used redundantly for ASIL-D operation. How independent redundancy is achieved is detailed later in this section.

The rightmost example in the Figure corresponds to the case where the functionality of an item is split into several subitems, being one of them in charge of preserving functional safety for the whole item. This is a usual solution for items with a *safe state*, i.e. a state in which functional safety is not challenged. For instance, upon a failure of the steering lock system, the safe state would be unlock the steer wheel. In this case, monitor capabilities to detect malfunctioning of the system must remain at the corresponding ASIL, whereas the operation part of the system can be kept at QM – thus with no specific V&V requirements – as long as any failure in the operation part can be timely detected by the monitoring part to drive the system to the safe state within the fault-tolerant time interval (FTTI). In the context of AD, however, the latter example cannot be applied for most of the functionalities since safe states may not exist. While most systems related to braking and steering resort to some sort of driver intervention to manage potentially hazardous situations, for the highest autonomy levels in AD – levels 3 to 5 as described in J3016 standard [5] – control can only be transferred to the driver in some circumstances (levels 3 and 4) or simply can never be transferred (level 5). Hence, computation components, which consist of accelerators (e.g. GPUs) must be certified to reach ASIL-D. Therefore, similar solutions to those for microcontrollers (i.e. DCLS) must be used for GPUs.

B. Redundancy and Diversity

As explained before, ASIL-D has been reached for AD systems building upon coarse-grain ASIL decomposition such as replicating full systems or parts thereof. However, since ASIL decomposition imposes the use of *independent* redundancy to avoid Common Cause Faults (CCFs) to lead to a failure, approaches used so far consist of using fully heterogeneous hardware and/or software components [4], with large impact in design and V&V costs. In particular, items must be proven to be free of systematic hardware and software faults with diagnostic coverage levels in accordance to their corresponding ASIL. However, since random hardware faults cannot be avoided, whenever there can be a CCF for redundant elements, safety measures must be put in place to ensure that they are timely detected and corrected. For instance, faults due to voltage droops, crosstalk, etc., which may affect identically redundant elements, must not lead redundant elements to the same erroneous output so that faults can be detected timely and corrected (e.g. by resetting and restarting the system).

As indicated in ISO26262, DCLS is an appropriate solution for redundancy, but diversity is also needed. Typically, it is implemented with some form of staggered execution so that, by executing the same software with some (sufficient) slack across redundant cores, a simultaneous identical transient fault will lead to different outputs for both cores if the fault causes an error. Thus, the error will be detected. For instance, this is the solution adopted by Infineon AURIX processors for automotive systems [6] as well as some ARM Cortex-R processors [7], [8].

ISO26262 does not provide explicit means to quantify diversity, which remains as an open challenge [9], and is typically assessed by safety experts. To the best of our knowledge, only solutions based on diverse lockstep operation have been deployed for computing devices. Hence, the diversity provided by this approach can be regarded as sufficient. Moreover, such as solution is typically applied at specific spheres of replication (SoR) so that physical redundancy is kept low. For instance, by using the core as SoR, main memory, communication interfaces and other non-computing elements do not need to be replicated and, instead, can rely on much lighter solutions to achieve diverse redundancy such as Error Correcting Codes (ECC) or Cyclic Redundancy Check (CRC).

In next sections we identify an appropriate SoR for GPUs and achieve diverse redundancy with low cost.

III. GPU Analysis and Strategy for Diverse Redundancy

This section introduces some concepts related to GPU design and operation, how those relate to the execution of kernels, and how diverse redundancy could be achieved on top of COTS GPUs.

A. GPU Design and Operation

This section provides some key concepts related to COTS GPUs relevant for our work. Note that the purpose is not providing a full description of GPU design and operation, but providing only those elements needed for our work due to space limitations. Since different components have different names across GPU vendors, we adhere to NVIDIA nomenclature for the sake of simplicity (and because NVIDIA is already targeting the automotive domain [3]), but concepts apply to virtually any COTS high-performance GPU.

Figure 2 shows a schematic of the main GPU components relevant for this discussion. First, the GPU has a number of Streaming Multiprocessors (SM), which we indicate as $SM1$ to SMn in the plot. Each one consists of a number of *execution* elements, which include CUDA cores (or simply cores), but also load/store units and complex cores. We group all of them within the concept of *cores* for the sake of this discussion. SMs also include a number of internal resources shared across cores such as instruction and data caches, on-chip shared memory, a register file, and an internal scheduler

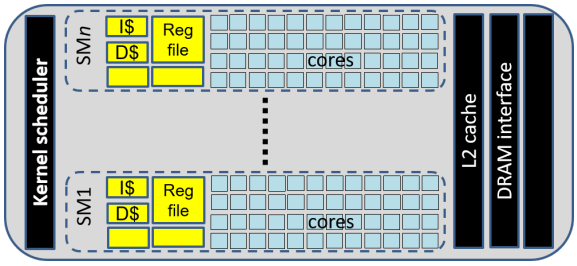

Fig. 2: GPU schematic.

among others. The GPU also includes a number of resources shared across SMs, such as a second level (L2) cache, DRAM interfaces, and other interfaces, and a kernel scheduler.

The kernel scheduler dispatches thread blocks of kernels to SMs. In particular, the CPU dispatches kernels to the GPU, each kernel consists of a number of thread blocks, and each thread block is bound to a SM for its entire execution, without possibility to migrate. However, different thread blocks from the same kernel can coexist on a SM provided that there are enough resources. For instance, if kernels $k1$ and $k2$ are dispatched to the GPU, where $k1$ has 3 thread blocks (tb_1^{k1}, tb_2^{k1}, tb_3^{k1}), $k2$ has 4 thread blocks (tb_1^{k2}, tb_2^{k2}, tb_3^{k2}, tb_4^{k2}), and our GPU has 2 SMs ($SM1$ and $SM2$), $SM1$ may execute tb_1^{k1}, tb_2^{k1}, tb_2^{k2}, tb_4^{k2} in a time-multiplexed manner but not necessarily completed with this order, and $SM2$ may therefore execute tb_1^{k2}, tb_3^{k1}, tb_3^{k2} also time multiplexed. Note that newer GPU architectures targeting the high-performance domain may have fewer limitations about executing different kernels in a single SM but, in general, how thread blocks are scheduled to SMs is an undisclosed feature, which, as discussed later, has prominent importance in our work.

B. Redundancy and Diversity Elements

As discussed before, storage and communication components can be properly protected from CCFs by using ECC and/or CRC. In fact, some of those components are explicitly protected with those means in NVIDIA GPUS [10], including register files, SM cache memories, and shared L2 cache, which employ Single-Error Correction Double Error Detection (SECDED) codes.

Regarding cores, no explicit protection has been reported. However, we consider GPUs that have been shown compatible with ASIL-B ISO26262 requirements and thus, the failure rates and coverage of the cores and the corresponding safety mechanisms are in concordance with the requirements imposed by the certification standard. Additionally, GPUs are intrinsically redundant within an SM and across SMs. Therefore, it is possible executing the same computation twice in different cores at different times so that CCFs are avoided. In particular, CCFs related to defects of a hardware component can be avoided by executing the same computation redundantly in different cores. Transient CCFs related to faults affecting multiple components simultaneously (e.g. a voltage droop) can be avoided by performing redundant execution at different time instances.

Unfortunately, NVIDIA GPUs, as well as other families, do not provide means to control how thread blocks are scheduled across SMs or a thread block is scheduled within a SM. Even worse, scheduler policies are not even publicly described, which further defeats any attempt to exercise direct control on the execution in the GPU, thus challenging the ability to enforce diverse redundancy on GPUs.

Finally, to the best of our knowledge, the global kernel scheduler does not include any form of redundancy for fault detection.

The aim of this work is proposing the smallest modifications possible to COTS GPUs to enable diverse redundancy to prevent CCFs.

IV. Scheduling Strategy for Diverse and Redundant GPU Execution

As explained, execution on the cores (computing and load-/store units) need some form of strategy to reach diverse redundancy, and the global kernel scheduler needs also means to avoid CCFs. In this section we introduce first our software approach to achieve redundancy, we analyze to what extent diversity can be achieved, and then propose low-cost modifications on the GPU design to achieve fully diverse redundancy.

A. Kernel Redundancy

In our approach – in line with the existing AD platforms – we consider a system in which ASIL-D capable microcontrollers (e.g. DCLS) offload intensive computations to the GPU. Our strategy consists on executing kernels twice on the GPU, and comparing their outcomes in the DCLS cores of the CPU. In particular, a DCLS core (1) allocates memory on the GPU memory space for both redundant kernels, (2) transfers data physically (if needed), (3) launches the two redundant kernels, (4) collects results from both kernels back to the CPU, and (5) compares their outcomes in the DCLS cores. In this scheme, all actions performed on the DCLS cores are naturally protected against CCFs, as well as data communication and storage, which occur on ECC or CRC protected components[1].

We consider identical redundant kernels. In general, one could create different kernel grids so that thread blocks across redundant kernels differ to introduce some form of diversity. However, the lack of control on the global kernel scheduler and SM internal schedulers prevents from guaranteeing specific diversity levels in the execution in the general case. Therefore, in this work we do not study diverse kernel generation, which is part of our future work.

The process to dispatch kernels to the GPU is intrinsically serial, so redundant kernels arrive at different time instants at the GPU, which might bring some form of diversity. However, this does not guarantee that two redundant thread blocks (from redundant kernels) cannot arrive to different SMs at the same time and, therefore, execute the same operations simultaneously, thus being subject to some transient CCFs. With respect to permanent CCFs, ensuring diversity would require – as already done for functionally identical core replicas in ASIL-D DCLS processors – implementing some form of physical diversity at layout and/or floorplan levels. However, even when having this physical level diversity, redundant thread blocks across redundant kernels may end up executing on the same SM at different time instants, thus also being subject to some permanent CCFs. Overall, redundancy can be easily achieved, but further means are required to enforce diversity.

B. Redundant Kernel Execution Patterns

While the solutions we propose later in this section work for any kernel, depending on the kernel characteristics a given

[1]In this paper, to keep the focus on the GPU design we consider dual modular redundancy suffices to provide fail-operational capabilities (i.e. errors can be recovered within the FTTI) by, for instance, reexecuting upon an error detection. However, our approach could be seamlessly extended to other redundancy levels (e.g. triple modular redundancy).

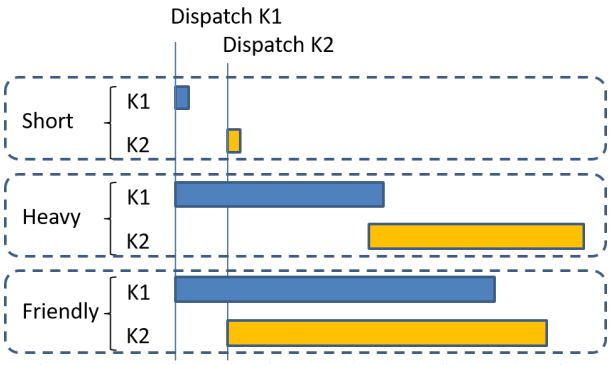

Fig. 3: Kernel categories based on their overlapping.

solution is potentially more appropriate than others. Thus, we first categorize kernels based on two criteria: whether they can potentially overlap their execution, and whether they use too many resources to prevent overlapping. This leads to three categories, also shown in Figure 3 for clarity:

- **Short kernels**. Those kernels execute too fast to overlap practically. In particular, by the time the second kernel is dispatched to the GPU, the first kernel has already finished its execution.
- **Heavy kernels**. Those kernels coexist in the GPU, but a single kernel uses too many resources to allow the other to start their execution. This makes that no overlapping occurs at all, or it is little, just at the end of the execution of one kernel, when it starts releasing resources so that the other can effectively start its execution.
- **Friendly kernels**. Those kernels coexist in the GPU and use limited resources so that both kernels can make progress concurrently.

As shown, different kernel types may have different degrees of overlapping (little-none or high). We propose two specific kernel scheduling policies in the GPU that allow achieving diverse redundancy in all cases: *SRRS* and *HALF* policies.

1) SRRS policy: SRRS stands for Start, Round-Robin and Serial policy. This policy requires that, first, we do not start the kernel execution until the GPU is idle; second, we can select the SM where the first thread block will be dispatched; third, SMs are allocated following a round-robin policy; fourth, kernel execution is fully serialized, thus delaying the start of the second (redundant) kernel until the first kernel has finished its execution; and fifth, no further kernel can be executed in the GPU until the second one also finishes.

By using *SRRS* with different starting SMs for both kernels, diversity is achieved naturally. The first kernel finds the GPU idle, starts in a particular SM (e.g. SMi) and allocates SMs in round-robin order starting from SMi until the kernel completes its execution, without any interference from any other kernel. The second kernel also finds the GPU idle, so in the same state as the first kernel, and starts its execution in SMj, where $i \neq j$. SMs are allocated also round-robin, but since the starting SM differs for both kernels and no interference occurs, each single thread block executes in different SMs across redundant kernels. Therefore, any single computation occurs in different kernels at different time instants, thus avoiding CCFs in the cores.

2) HALF policy: HALF policy builds upon allocating half of the SMs to one kernel and the other half to the other kernel. This naturally imposes the use of different SMs for each kernel. On the other hand, the fact that their starting times differ due to the serial dispatch of kernels to the GPU, enforces also that any given redundant computation occurs at different time instants. Note that kernels could interfere in the use of shared resources delaying each other. However, their requests can never occur at the same time because (i) either shared resources can process them in parallel, so no interference occurs and so no timing impact, or (ii) requests are serialized (at least partially) for a given shared resource, so that a given request arrives before for the first kernel than for the second, and the second can neither start nor finish simultaneously with the first one, thus preserving some slack across kernels. Therefore, any single computation occurs both, in different SMs and at different time instants, thus avoiding CCFs in the cores.

C. Diverse Redundancy in the Kernel Scheduler

Both policies, *SRRS* and *HALF*, schedule any given thread block from both kernels at different time instants and to different SMs. Therefore, any fault causing an improper execution of the kernels, may have several consequences: (1) execution occurs functionally correctly in different SMs to the ones intended, but still redundantly and with diversity. In this case, no failure occurs. (2) execution occurs functionally correctly in different SMs to the one intended, but failing to achieve diversity (e.g. the same computation occurs redundantly on the same SM). In this case, let us recall that ISO26262 requirements relate to the ability to avoid a single fault from causing a failure. Hence, upon a fault in the scheduler, we must assume that the remaining components are fault-free and hence, even if their execution is not diverse, no further fault is expected. (3) execution does not terminate or terminates with errors for at least one kernel (e.g. by skipping a thread block). In this latter case, the different behavior of both redundant kernels (each thread block is executed at different times in different SMs) makes that even if there is a physical fault in the scheduler, its behavior will differ across kernels, so evidence on diversity is enough to meet ISO26262 requirements.

A final important remark in the context of ISO26262 is the fact that we can assume that multiple faults cannot occur simultaneously as long as faults are timely detected. In particular, this means that faults of type (2), so with no functional impact but decreasing diversity, must be detected if related to a physical fault since, otherwise, a future fault on a core in a SM could lead to an undetected error, and thus to a failure. In order to avoid this behavior, the global kernel scheduler must undergo periodic tests so that physical faults do not become latent.

D. Appropriateness of the Scheduling Policies

Both scheduling policies, *SRRS* and *HALF*, achieve diverse redundancy for all kernel types. However, each kernel type is particularly suitable for one of them.

- Short kernels may potentially use many GPU resources during their (short) execution. Since their execution does not overlap at all, *SRRS* is expected to cause no performance degradation at all. Instead, *HALF* could increase kernel execution time, thus impacting performance.
- Heavy kernels need many GPU resources and have little or no overlap at all. Hence, *SRRS* may only slightly increase their execution time if they overlap a bit. Instead, *HALF* could easily increase execution time of a given kernel noticeably while not allowing the other one to start due to lack of resources.

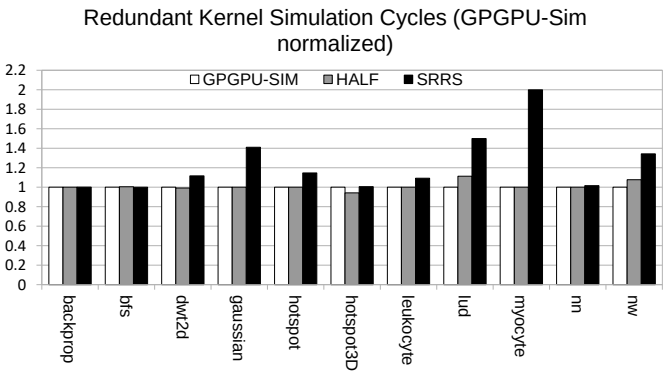

Fig. 4: Scheduler simulations using GPGPUSim

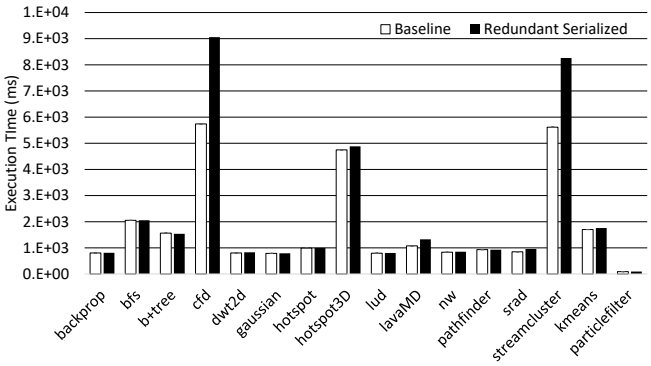

Fig. 5: SRRS implementation by serializing redundant kernels

- Friendly kernels can run concurrently, so using *SRRS* could cause significant execution time increase due to their serialization if a given kernel is unable to exploit all SMs efficiently. Instead, *HALF* grants each kernel half of the SMs, which is the amount of resources they would use if run concurrently without explicit control for the sake of diversity.

Overall, *SRRS* is the most convenient policy for short and heavy kernels, whereas *HALF* fits friendly kernels. Since kernel classification is performed during the analysis phase of the system, the particular policy to use for each one can be decided before system deployment, so that each kernel is executed with the most convenient policy during operation. Note that this implies that specific means are required to select the global kernel scheduler policy during operation, which we foresee as feasible since it is not different from other reconfigurations applied on high-performance components such as enabling/disabling prefetchers, changing fetch policies, and the like.

V. Evaluation

We have implemented the proposed scheduling policies *SRRS* and *HALF* in the latest version of GPGPUSim [11] (version 3.2.2). The GPU modelled with this simulator consists of 6 SMs. In order to evaluate our solutions, we use the Rodinia benchmark suite [12], [13]. We have modified Rodinia benchmarks by implementing redundant kernel execution and output comparison, as required to enable ISO26262 compliant redundant execution, and simulated them on GPGPUSim.

For implementing *SRRS* and *HALF* in GPGPUSim [11], we have modified the default scheduling policy according to the requirements that each proposal imposes to the way the available SMs have to be assigned. For *HALF*, we use the default scheduling policy implemented in the GPGPUSim and restrict each kernel execution to 3 dedicated SMs. We compare the performance of *SRRS* and *HALF* with the one obtained with the default GPGPUSim scheduler that can allocate all GPU SMs (6) to the kernels without any constraint.

A. Simulation Results

Results of the simulated time *only for the kernel execution* for each policy can be seen in Figure 4. Due to the costly executions on the simulator, we evaluated a subset of the benchmarks. In particular, we inspected them and identified that most of them include friendly kernels. Thus, running additional experiments does not provide further insights. In general, the performance overheads of the proposed scheduler policies are not very high in comparison with the default scheduler (except for *myocyte*). In particular, *HALF* policy performance overheads are negligible for 9 out of the 11 benchmarks analyzed and only 10% in the worst-case (*lud*). The results for the *SRRS* policy are slightly worse due to the extra overheads that this policy incurs to perform the serialization. For *SRRS*, performance overheads can be up to 99%. In general, kernels are friendly or short and, if they are short, they also require at most half of the SMs. Hence, by restricting them to use half of the SMs with *HALF*, performance penalty is in general very low. Instead, serialization imposed by *SRRS* increases their execution time. The only exception are *bfs* and *backprop*, which have very short kernels requiring more than half of the resources. Hence, serialization imposed by *SRRS* is innocuous, whereas limiting the number of SMs with *HALF* increases execution time. However, since kernel execution time is much lower than the execution time of the CUDA commands to launch the kernels, the relative impact of such increase is tiny.

B. COTS GPU Results

Finally, in order to assess the suitability of the proposed redundant execution in a real environment and understand the impact of redundant execution w.r.t. non-redundant execution, we have mimicked the implementation of *SRRS* on a COTS GPU. To do so, we serialize the redundant kernels execution using the CUDA call *cudaDeviceSynchronize()* that prevents the execution of further operations until all previous operations on the GPU have been completed. While such a solution does not enforce diversity due to the lack of control of the particular SMs used, it causes the same timing behavior. Note that mimicking *HALF* is not possible on the COTS GPU, since CUDA doesn't provide control over the SMs used by a kernel.

Figure 5 compares the execution time of end-to-end executions of the benchmarks with the redundant serialized and no redundant kernels of the rodinia benchmarks. By running on a real platform, we could afford running all benchmarks timely. In particular, we use a system with an AMD Ryzen 7 1800x CPU, a GTX 1050 Ti GPU which has the same number of SMs as the simulated platform, and 64GB of DDR4 memory.

The bars in the plot show the result of averaging out 100 executions. As shown in the plot, the redundant execution of the kernels does not incur significant performance degradation for the workloads analyzed. In fact, for all the benchmarks but two (*cfd* and *streamcluster*) the impact of redundant execution is negligible. The main reasons for such behavior are as follows: (1) the impact of *SRRS* is, in general, low

as shown in Figure 4; (2) the contribution of the kernel execution to the total execution time of the benchmark is relatively low in general; and (3) the cost of sending input and output data twice, and comparing the outputs of the kernels in the CPU is also very low in relative terms. In the case of *cfd* and *streamcluster*, the two only notable exceptions to this behavior, we note that serialization imposed by *SRRS* has a relatively significant impact for the execution of the kernels, and execution time of the benchmarks is largely dominated by the kernel execution. The latter also makes that the relative contribution of duplicating input data, transferring back output data to the CPU twice and comparing outputs is non-negligible, thus contributing to the execution time increase w.r.t. the non-redundant version of the benchmark.

VI. Related Work

Lockstep processors like the AURIX [6] or the ARM Triple-core lockstep [7] implement a number of diversity techniques such as staggered execution, and layout and floorplan diversity to ensure the robustness of the systems in the presence of CCFs. Unfortunately, simple dual-core or triple-core lockstep processors do not suffice to meet computational and safety requirements of AD, which calls for more powerful ASIL-D capable computing platforms. In that respect, GPUs have already been positioned in the automotive systems market [2], [14] as a suitable computing platform for AD. In fact, NVIDIA has recently announced the first functionally safe autonomous driving platform [4]. However, to achieve ASIL-D fail-operational capabilities this platform relies on diverse software implementations of complex software algorithms running on the CPU, the CUDA GPU, a deep-learning accelerator and a programmable vision accelerator, which comes at the expense of drastically increasing design and V&V costs.

A recent work assesses the effectiveness of FPGA, ASIC and GPU designs for AD applications [15]. While the conclusion is that each solution provides a different power/performance tradeoff and hence, the best platform may change across AD applications, automotive-specific GPU platforms have already been released, as indicated before. Some authors show that GPU performance can be improved with some hardware modifications [16], [17], [18]. The suitability of using GPUs in the context of safety-critical applications from the point of view of real-time performance has been recently assessed in several works [19], [20]. These works show that current GPU architectures include features that limit the timing analyzability of the software running on top of these platforms challenging the timing verification step. However, to the best of our knowledge, no previous work provides a solution to enable diverse redundancy by construction in GPUs.

In this paper, we show how relatively simple modifications in the scheduling policies of COTS GPUs can facilitate achieving ASIL-D requirements without the need of using fully redundant systems and thus, significantly containing V&V costs.

VII. Conclusions

The use of GPUs for highly-critical autonomous driving (AD) software poses a number of functional safety requirements on the design and utilization of GPUs. While existing AD-specific GPUs already meet some of those requirements efficiently, redundant diversity – needed for ASIL-D software – is not reached efficiently and can only be reached by deploying heterogeneous software implementations and/or computing platforms, which jeopardizes cost and efficiency.

This paper proposes minor modifications of the scheduling policies of GPUs that allow guaranteeing by construction diverse redundancy, thus reaching ASIL-D compliance efficiently without the need of increasing design and/or procurement costs. In particular, we show how the explicit control of the SMs used for a given kernel together with the serialization of redundant execution in some cases allows achieving diverse redundancy with low cost w.r.t. uncontrolled redundancy.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P and the HiPEAC Network of Excellence. Jaume Abella has been partially supported by the MINECO under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717. Carles Hernandez is jointly funded by the MINECO and FEDER funds through grant TIN2014-60404-JIN.

References

[1] International Standards Organization, *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[2] "RENESAS R-Car H3," https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html.
[3] D. Shapiro, "Introducing Xavier, the NVIDIA AI Supercomputer for the Future of Autonomous Transportation," *NVIDIA blog*, 2016. [Online]. Available: https://blogs.nvidia.com/blog/2016/09/28/xavier/
[4] NVIDIA, "NVIDIA Announces World's First Functionally Safe AI Self-Driving Platform," https://nvidianews.nvidia.com/news/nvidia-announces-worlds-first-functionally-safe-ai-self-driving-platform.
[5] SAE International, *J3016: Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*, 2014.
[6] Infineon, "AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations," http://www.infineon.com/cms/en/about-infineon/press/press-releases/2012/INFATV201205-040.html.
[7] X. Iturbe et al., "Addressing Functional Safety Challenges in Autonomous Vehicles with the Arm Triple Core Lock-Step (TCLS) Architecture," *IEEE Design and Test*, vol. PP, no. 99, pp. 1–1, 2018.
[8] B. Venu et al., "A Fail-Functional Automotive CPU Subsystem Architecture for Mitigating Single Point of Failures," in *IEEE International Workshop on Automotive Reliability and Test*, 2017.
[9] S. Alcaide et al., "DIMP: A low-Cost Diversity Metric based on circuit Path analysis," in *DAC*, 2017.
[10] NVIDIA, "Fermi. NVIDIA's Next Generation CUDA Compute Architecture. White paper," 2009.
[11] A. Bakhoda et al., "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[12] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[13] ——, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," *IISWC*, 2010.
[14] TESLA, "Full Self-Driving Hardware on All Cars," https://www.tesla.com/autopilot.
[15] S.-C. Lin et al., "The architectural implications of autonomous driving: Constraints and acceleration," in *ASPLOS*, 2018.
[16] H. Dai et al., "Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls," in *HPCA*, 2018.
[17] J. T. Adriaens et al., "The case for GPGPU spatial multitasking," in *HPCA*, 2012.
[18] P. Aguilera et al., "Fair share: Allocation of GPU resources for both performance and fairness," in *ICCD*, 2014.
[19] M. Yang et al., "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," in *ECRTS*, 2018.
[20] T. Amert et al., "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *RTSS*, 2017.