# Achieving Diverse Redundancy for GPU Kernels

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

**Abstract**—Autonomous driving requires high-performance computing devices including general-purpose CPUs as well as specific accelerators, with GPUs having a key role due to their flexibility. Safety-critical microcontrollers have achieved ASIL-D compliance by implementing diverse redundancy with lockstep execution on-chip. However, a GPU does not provide diverse redundancy natively, thus failing to reach ASIL-D, which could only be reached with fully redundant lockstepped GPUs (2 GPUs) or pairing a GPU with another accelerator. However, both options may be infeasible due to procurement costs, and additional power, space and reliability costs to accomodate two devices. In this work, we present a variety of solutions to enable diverse redundant execution using only one GPU by taking advantage of the already internal redundancy of GPUs. We provide two lowly-intrusive hardware solutions and a software-only solution, with the latter evaluated directly on a real platform. In the case of the software-only solution, kernel execution on the GPU may require tailoring some parameters. With that objective, we also propose an algorithm that performs such tailoring automatically to guarantee software-only diverse redundancy on GPUs. Overall, our solutions allow achieving ASIL-D with a single GPU either with software-only solutions on a Commercial off-the-shelf GPU, or in a more efficient manner by introducing minor changes in the GPU design.

## 1 Introduction

The advent of Autonomous Driving (AD) imposes the adoption of high-performance hardware in critical real-time embedded systems (CRTES) for the execution of object detection and tracking algorithms, as well as for driving decisions. However, hardware designs for CRTES must undergo a strict design, Verification and Validation (V&V) process to guarantee that failure risks are residual in accordance with automotive safety regulations (ISO26262 and ISO21448). This V&V process typically clashes with the design of high-performance hardware, where performance optimization prevails over controllability and observability requirements needed for V&V. Instead, safety requirements include detecting faults and preventing hazardous situations, and these requirements propagate to system items.

Graphical Processing Units (GPUs) are becoming a highly popular accelerator for AD. To respond to these opposing requirements, namely high performance, and functional safety, some hardware vendors have recently deployed specific products for the automotive market and, particularly, for AD applications. For instance, Renesas R-Car H3 [1] and NVIDIA Xavier [2] product families deliver a multicore CPU together with a GPU, where the latter is in charge of executing high-performance and parallel kernels so that the system can meet the stringent performance requirements of AD applications.

Functionalities in automotive systems are classified into different Automotive Safety Integrity Levels (ASILs) as dictated by ISO26262. ASILs are determined by the exposure, severity and controllability upon a failure of hazardous events. As long as an item inherits some safety requirements, its ASIL ranges between A to D, being D the highest level and A the lowest. If the item does not inherit any safety requirement, then it is regarded as Quality Managed (QM).

AD functionalities include braking, accelerating, and steering, so they are naturally classified as ASIL-D. To reach ASIL-D, any functionality needs to build upon some form of diverse redundancy to avoid Common Cause Failures (CCFs), which are those failures caused by a single fault (e.g., a fault affecting similarly redundant and non-diverse items). Diverse redundancy (e.g., dual-core lockstep execution in CPUs) is normally deployed so that faults are detected timely (i.e. with redundancy) even if they affect redundant components, since those components provide diversity by either being heterogeneous or by holding different internal state at any time. Error Detection and/or Correction Codes (EDC/ECC) such as SECDED (Single Error Correction/Double Error Detection) or Cyclic Redundant Check (CRC) are often used as a means to achieve diverse redundancy for storage and communication. Environment sensing achieves diverse redundancy by building on different technologies such as cameras, LiDAR, and radar. Finally, computation often builds upon staggered lockstep execution, deployed in automotive CPUs, such as for instance, the Infineon AURIX microcontroller family [3], where identical cores run the same software simultaneously with some staggering (i.e., time shift) across cores. Such a solution is beneficial for several reasons:

1) Cost: a single hardware and software design is needed since redundant cores and software executed are identical.
2) Performance and power: on-chip comparison can be performed.
3) Reliability and space: keeping activities on-chip avoids involving additional physical components.

In the case of GPUs, hardware manufacturers claim ASIL-B compliance and enabling ASIL-D compliance, where the latter can be achieved by either setting up two redundant ASIL-B GPUs with some form of diversity or by using diverse components such as the GPU and the NVDLA *(NVIDIA Deep Learning Accelerator)* accelerator as in the case of NVIDIA [2]. However, these solutions defeat at least 2 out of the 3 benefits indicated above. Using two GPUs reduces the performance and doubles the power while also reducing reliability. Instead, using two diverse components

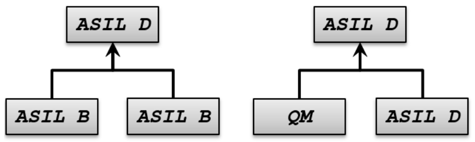


Fig. 1: Usual decomposition patterns for ASIL-D items.

increases the cost by requiring to design, not only two different ASIL-B hardware components but also two different ASIL-B software implementations to perform the same functionality, which ultimately also doubles the V&V costs.

Given the intrinsic internal redundancy of Commercial Off-The-Shelf (COTS) GPUs, this work proposes alternative solutions to reach ASIL-D building on a single COTS GPU, one by software-only means and the others with lowly-intrusive hardware modifications[1]. These solutions address successfully the three challenges presented above in terms of (i) cost, (ii) performance and power, and (iii) reliability and space by exploiting the intrinsic redundant nature of GPU designs.

The rest of the paper is organized as follows. Section 2 provides some background on automotive functional safety and GPU architecture. Section 3 introduces key concepts and strategies to achieve diverse redundancy on GPUs. Section 4 presents our two solutions based on hardware modifications. Section 5 presents our software-only solution. All solutions are evaluated and compared in Section 6. Related work is described in Section 7. Finally, Section 8 summarizes this work.

## 2 BACKGROUND

### 2.1 ASIL Decomposition in ISO26262

ASIL decomposition is often used for the highest integrity levels as an economically viable way to meet safety requirements by using appropriate combinations of lower ASIL items. This relates to the fact that reaching certain coverage levels and failure rates needed for the highest ASILs may impose excessive costs (or simply be unreachable) if a single item is used, whereas lower ASIL items are cheaper to design and verify. However, in order to meet safety requirements and avoid common cause failures (CCFs)[2], the two lower ASIL components used redundantly must be proven to be *sufficiently independent*. This imposes the use of some form of *diversity* across redundant items. For example, using two power supplies, one per each component, could avoid a failure due to a voltage droop.

ASIL-D compliant CPUs are available in the market, such as for instance, the Infineon AURIX family [3]. Those CPUs reach ASIL-D by using redundant identical cores that execute the same instruction stream, but with some staggering (i.e., one core is ahead of the other by few cycles). So that core's internal state differs and, upon a fault affecting both redundant cores, the impact in their state will differ and lead to different states, thus allowing to detect the errors before they become a failure. However, high-performance accelerators do not reach ASIL-D in general, and for GPUs in particular, which challenges their use in safety-related systems. Note that, as discussed before, some chip vendors, such as NVIDIA, already deploy GPUs for AD systems but failing to reach ASIL-D with a single GPU.

Automotive systems have been traditionally regarded as fail-safe, meaning that they have a safe state that, in general, consists of transferring the control to the driver. Hence, a valid ASIL decomposition pattern consists of using an ASIL-D microcontroller monitoring faults and transferring the system to a safe state whenever needed, and a QM high-performance computing unit, thus without safety requirements, see Figure 1 (right). However, such a fail-safe system is not fault-tolerant since, upon an error, the system transitions to the safe state and becomes unavailable. For instance, consider an ASIL-D system based on a QM accelerator (e.g., a GPU) in charge of the decision system of a car and a monitor regarded as ASIL-D, which monitors the accelerator execution (Figure 1 right). A failure in the accelerator, will be noticed by the monitor and will put the system in a safe state but will not carry out the task timely. However, the highest autonomy levels in AD must be fail-operational, meaning that they operate correctly at all times with no safe state available (e.g., the car may even lack a steering wheel).

Instead, fail-operational systems must build on a different ASIL decomposition, such as that shown in Figure 1 left, where ASIL-D is reached through the combination of diverse redundancy of lower ASIL (yet with some ASIL) items. For instance, a scheme with two GPUs with some sort of diversity would allow detecting erroneous outputs and, upon a fault, restart the task at least in the fault-free GPU to guarantee operation despite faults. This is analogous to the use of Dual-Core Lockstep (DCLS) for CPUs.

However, as explained before, using redundant devices is ill-advised due to a number of reasons related to cost, integration, and reliability issues (i.e., increasing likelihood of a physical fault with two devices and their interconnects w.r.t. a single device). Therefore, ASIL-D must be reached in a single system-on-chip (SoC), as it is done for CPUs, executing tasks redundantly in a single GPU with some form of control guaranteeing that redundancy is effectively diverse. Thus, in this work, we provide three complete solutions (for any kernel) to achieve some form of on-chip DCLS for GPUs, two of them by using lowly-intrusive hardware modifications and the latter by software-only means.

It is worth noting that this work focuses on fault detection mechanisms, so error handling is out of the scope of this paper, and additional mechanisms might be required to perform the error handling tasks. However, such concern is analogous to that of DCLS in the case of CPUs, so solutions based on restart or checkpointing apply.

### 2.2 GPU Architecture and Support for Diverse Redundancy

GPUs in general, and NVIDIA GPUs in particular, which are explicitly considered in this work, consist of a number of components (see Figure 2). First, a kernel scheduler which schedules thread blocks (groups of threads that may synchronize or communicate through shared memory) to Streaming Multiprocessors (SMs). Second, several SMs, where computation effectively occurs. Each SM has a set of local resources for storage, such as first level cache memories and register files, and computing cores (including load/store data units). SMs also share part of the cache hierarchy (e.g., a second level cache) and communication and memory interfaces.

1. This work combines, extends and compares our previous work in [4] on hardware solutions and [5], [6] on software solutions.

2. CCFs are those failures where a single fault affecting redundant elements leads to undetectable errors (e.g., both items deliver identical erroneous outputs), so that, despite redundancy, errors remain undetected and can become failures.

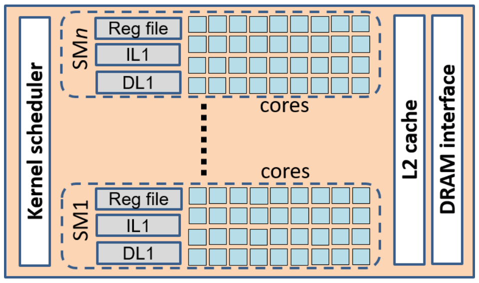


Fig. 2: Schematic of a usual GPU architecture.

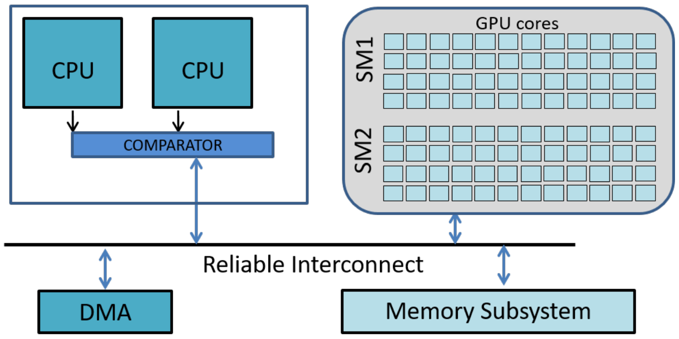


Fig. 3: Proposed Computing Platform architecture

As indicated before, storage and communication means are typically ECC, or CRC protected and, in fact, this is the case for NVIDIA GPUs. SMs are naturally redundant among them, so that this feature can be exploited for fault detection. Finally, the kernel scheduler needs explicit protection through replication. However, although adding this kind of redundancy brings some additional hardware cost, the vast majority of the GPU hardware is devoted to SMs and shared storage, which are implicitly or explicitly redundant. Hence, making kernel schedulers redundant is not foreseen as a challenging concern.

Regarding SMs, while they are redundant, it is mandatory to have some diversity so that redundant computations use different SMs and they do it at different time instants to avoid CCFs. Therefore, there are two requirements that must be fulfilled by redundant operations to guarantee diverse redundancy computation on a GPU.

1) They must not execute on the same hardware.
2) They must not execute synchronized. Instead, some form of staggered execution is needed.

## 3 Key concepts to achieve diverse redundancy in a GPU and system modeling

This section introduces relevant concepts that affect diverse redundant execution in the GPU. We also describe the system considered and the offloading process of the kernels. Finally, we provide a schematic of the software modifications made to enable diverse redundant execution.

### 3.1 GPU Concepts and Redundancy Granularity

In CUDA *(Compute Unified Device Architecture)*, the programming paradigm for NVIDIA GPUs, kernels and explicit memory transfers are both assigned to a *CUDA stream*. In each stream, kernels and memory transfers are executed serially in order of arrival (using a FIFO queue). As authors in [7] found, there is a FIFO queue per each stream, and only when kernels reach the head of their stream queue are enqueued in a FIFO *execution engine* (EE) queue. Only blocks of the kernel at the head of the EE queue are eligible to be assigned to SMs. Finally, a kernel only leaves the head of the EE queue (i.e., it is dispatched) when all its blocks are assigned to SMs[3]. A block can be only assigned to an SM when there are enough available resources to accommodate its execution eg. enough available registers, threads or shared memory. Therefore, when using different streams, two kernels can run concurrently as long as:

1) The first kernel can be fully dispatched (this will remove it from the head of the EE queue).
2) The second kernel can be fully dispatched while the first one is still executing (therefore there are enough remaining resources).

In addition, as pointed out in the CUDA programming guide [8], none of them can use the *default or NULL stream* if we want to achieve kernel concurrency since any CUDA command to the *NULL stream* creates an implicit synchronization, and so kernel serialization.

Redundant execution in a GPU can be achieved at different granularities, include replication at thread block level [9], [10], [11], [12], [13], [14], at thread level, and at instruction level [15]. The finer the granularity, the more synchronization required, such as inter-thread communications (e.g. using the shared memory). In the context of CCFs, if replication occurs inside an SM (intra-SM), there is no way to exercise control on whether redundant operations are executed in different hardware (e.g., ALUs). Therefore, we select kernel granularity for redundancy because kernels can be scheduled per SM, so each replicated kernel uses different SMs[4]. On the other hand, even if different hardware can be guaranteed for redundant operations (e.g., different SMs), there is no straightforward way to enforce staggering.

However, we discovered that, due to the GPU's kernel offloading processes performed by the CPU, an initial staggering is created automatically between the two kernel executions as evaluated later.

In particular, the CUDA runtime performs a number of sequential actions to offload a kernel to the GPU, which serializes the offloading of different kernels. GPUs from other vendors are expected to behave similarly since this is an effect due to the inherent interaction between the CPU and the GPU.

### 3.2 System considered

We consider a system – in line with existing AD platforms – in which an ASIL-D capable microcontroller offloads computation-intensive work onto the GPU. These computations are organized as kernels. For each kernel, a redundant copy is sent to the GPU serially using a different stream. Since the ASIL-D microcontroller launches the two redundant kernels serially (due to CUDA runtime serialization), both kernels arrive to the GPU scheduler at different times, which leads to blackan initial *staggered execution* inside the GPU. We regard the other elements outside the GPU safe by either

3. While this description uses explicit NVIDIA concepts, similar concepts hold for other GPU families such as, for instance, AMD ones.

4. We discuss this issue later in Section 5

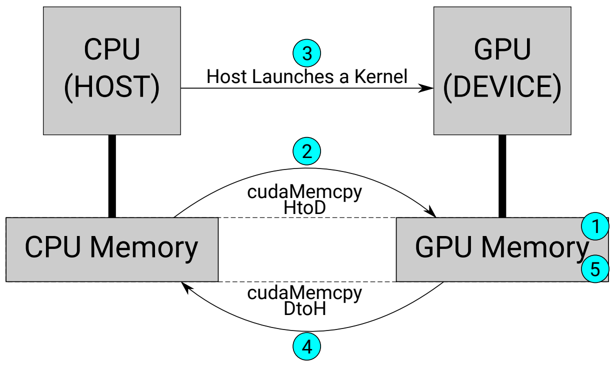


Fig. 4: Common CUDA Workflow

building on ECC (memory and storage), CRC (Interconnects), or DCLS (CPUs). In our system design, redundant kernels should execute on different GPU hardware with some staggering, and results are then sent back and compared in the ASIL-D microcontroller.

### 3.3 Offloading Process

The following steps are taken to offload computation onto the GPU:

- Offloading preparation process (① allocate memory and ② transfer data from host to the GPU device in Figure 4).
- ③ Kernel launching.
- ④ Collection of the results produced and ⑤ deallocating memory.

To prepare the execution of the kernel on the GPU, the code to be run must be sent to the GPU along with the input data, which must be transferred to the GPU memory. Typically, the GPU and the microcontroller use the same physical memory. However, each computing device has separate address spaces, and hence, independent memory mappings. Therefore, despite generally data are not physically copied, some bookkeeping and limited data transfer is needed (e.g. cache contents flushing to preserve memory consistency), which requires some time to be performed.

During the complete process, data transfers to/from memory and on-chip communication beyond the GPU use the same hardware components already used by the ASIL-D microcontroller, as shown in Figure 3. Hence, as explained before, their ECCs and CRCs already provide the required protection against errors. This implies that data transfers occurring in any of the steps of the kernel execution, namely kernel offloading, launching and result retrieval, occur on components delivering appropriate safety measures to reach ASIL-D compliance.

However, computation inside the GPU lacks appropriate support for safety compliance by default. Therefore, some sort of safety measures need to be deployed for the execution on the GPU, being those measures comparable to the ones of the microcontroller's DCLS cores. Appropriate safety levels can be achieved as follows (see Figure 5):

1) Set up two redundant and independent kernels.
2) Duplicate input data.
3) The GPU performs redundant computations on different components and with some staggering, therefore avoiding CCFs.
4) Data produced by redundant kernels is sent to the microcontroller where it will be compared.
5) The ASIL-D microcontroller performs the result comparison.

Note that, it would be possible avoiding the replication of read-only input data. However, then such data would not be replicated and some form of protection would be needed in the GPU (e.g. ECC) as a way to mitigate CCFs caused by errors in the non-replicated input data. Therefore, in this work we assume that all input data for redundant kernels is replicated, leaving the analysis of pros and cons of non-replicated read-only data for future work.

### 3.4 Redundant Kernel Execution Patterns

While our solutions work for any kernel, some kernel characteristics make a given solution more appealing than others, or even require methods to enable diverse redundant execution as in the case of our software-only solution (see section 5). Therefore, we classify kernels based on whether their execution lasts enough to overlap, and whether the amount of resources required is small enough to allow concurrent execution of redundant copies. We obtain the following three categories (see Figure 6):

- **Short kernels**. Short kernels last too little to overlap because the offloading process of the second kernel takes longer than the execution of the first one, so the first kernel completes its execution before the second starts.
- **Heavy kernels**. While heavy kernels run long enough to overlap, any such kernel needs so many GPU resources that precludes the other kernel from starting its execution until the first one finishes (or is close to finish). Therefore, overlap is tiny – if any – and occurs when the first kernel is about to finish and starts releasing GPU resources.
- **Friendly kernels**. Friendly kernels run concurrently in the GPU since their duration is long enough and the demanded GPU resources low enough.

Note that the classification of kernels depends on platform specific characteristics, such as for instance, the amount of GPU resources, which may make a kernel be heavy or friendly depending on whether those resources suffice to execute both redundant kernels concurrently. In the case of automotive applications, they typically have fixed input data size since input data always comes from the same sensor (e.g., images from a camera). This allows kernel classification to be performed a priori statically.

## 4 Hardware Solutions to Enable Diverse Redundancy on GPUs

In this section, we describe two HW solutions to achieve diverse redundant kernel execution on a GPU. Both of them consist of modifying the default kernel scheduler policy to ensure that redundant kernels are executed in different hardware and not at the exact same time.

The scheduling policies proposed, *SRRS* and *HALF*, target all types of kernels, regardless of how much they overlap with their redundant copies, i.e. no, little or large overlap.

### 4.1 SRRS policy

The Start, Round-Robin and Serial (*SRRS*) policy imposes the following 5 requirements: (1) kernel execution starts only when

```
//Input and Output data allocation on GPU
float *d_A, *d_C;


cudaMalloc(d_A, N*sizeof(float));cudaMalloc(d_C, N*sizeof
     (float));



cudaStream_t Streams[1];//Stream creation
cudaStreamCreate(&Streams[0]);


//Input data transfer to the GPU
cudaMemcpy(d_A, A, N*sizeof(float),
     cudaMemcpyHostToDevice);



//Kernel launch
kernel<<<NumBlocks, ThreadsPerBlock, 0, stream[0]>>>(d_A,
      d_C, N);



//Results transfer to the CPU
cudaMemcpy(C, d_C, N*sizeof(float),
     cudaMemcpyDeviceToHost);



//No comparison
```

(a) Original CUDA code

```
//Input and Output data allocation on GPU
float *d_A, *d_A_redundant;
float *d_C, *d_C_redundant;

cudaMalloc(d_A, N*sizeof(float)); cudaMalloc(
     d_A_redundant, N*sizeof(float));
cudaMalloc(d_C, N*sizeof(float)); cudaMalloc(
     d_C_redundant, N*sizeof(float));

cudaStream_t Streams[2];//Stream creation
cudaStreamCreate(&Streams[0]); cudaStreamCreate(&Streams
     [1]);

//Input and Replicated input data transfer to the GPU
cudaMemcpy(d_A, A, N*sizeof(float),
     cudaMemcpyHostToDevice);
cudaMemcpy(d_A_redundant, A, N*sizeof(float),
     cudaMemcpyHostToDevice);

//Redundant Kernel launch
kernel<<<NumBlocks, ThreadsPerBlock, 0, stream[0]>>>(d_A,
      d_C);
kernel<<<NumBlocks, ThreadsPerBlock, 0, stream[1]>>>(
     d_A_redundant, d_C_redundant);

//Results and Redundant result transfer to the CPU
cudaMemcpy(C, d_C, N*sizeof(float),
     cudaMemcpyDeviceToHost);
cudaMemcpy(C_redundant, d_C_redundant, N*sizeof(float),
     cudaMemcpyDeviceToHost);

//Comparison of C and C_redundant
```

(b) Applying Redundant Kernel Execution

Fig. 5: Original and modified CUDA code

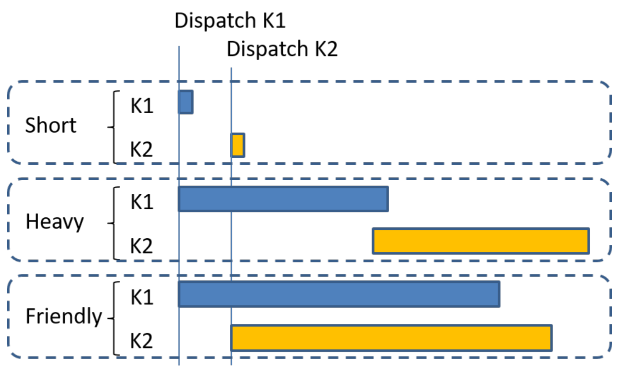


Fig. 6: Kernel categories based on their overlapping when launched with different streams. (K2 is the redundant kernel of K1)

the GPU is idle; (2) the SM for the first thread block of the first kernel is chosen (any SM is acceptable); (3) SM to thread block allocation follows a round-robin policy starting in the SM just chosen for the first thread block; (4) the second kernel does not start its execution until the first one is fully completed (complete kernel serialization) and uses a different starting SM; and (5) no new kernel is allowed to execute until the second (redundant) one completes its execution.

The operation of *SRRS* enforces diversity implicitly. The first kernel starts using the specific SM indicated (e.g., $SM_i$). Since the GPU is idle, round-robin arbitration policy allocates the following SMs to the thread blocks of this kernel in the corresponding round-robin order. The first kernel completes the execution and the GPU becomes idle again. Then, the second kernel starts using a different SM to the initial one of the first kernel, $SM_j$, where $i \neq j$. Since SMs are again allocated in round-robin and without interference from other kernels, thread block to SM allocation for the second kernel occurs analogously to that of the first kernel but systematically allocating different SMs across kernels to a given thread block. Regarding staggering, since kernel execution is serialized, staggering is guaranteed.

## 4.2 HALF policy

The *HALF* policy allocates half of the SMs to each kernel so that they use disjoint sets of SMs. This ensures diversity in terms of SMs used. Regarding time diversity, the serial kernel launching provides some initial staggering. If accesses to resources experience no contention, either because they are not shared or because they have enough bandwidth to serve requests from both kernels simultaneously, no contention occurs and hence, staggering is preserved. If, instead, kernels share a resource serializing requests, requests from the first kernel arrive before those of the second kernel due to the initial staggering. By serializing requests, those from the second kernel can only complete later and hence, some staggering is necessarily preserved. Therefore, both spatial and time diversity is guaranteed, hence avoiding CCFs.

## 4.3 Diverse Redundancy in the Kernel Scheduler

Both policies, *SRRS* and *HALF*, guarantee that both instances of any thread block execute on different SMs and with some staggering. Based on that, next we analyze the potential impact that any fault could have on the kernel scheduler:

1) Kernels are executed correctly (producing expected results) but in different SMs to those originally intended, however still preserving diverse redundancy. In this scenario there is no actual failure.

2) Kernels are executed correctly (producing expected results) but in different SMs to those originally intended. Moreover, diversity is lost, for instance, because the two redundant copies of at least one thread block execute on the same SM. ISO26262 calls for considering single faults from causing failures (e.g. due to a CCF), but assumes that multiple independent faults are too unlikely to occur in such a short period of time to be considered. Hence, if a fault impacts the scheduler leading to diversity loss, no other fault must be considered, and therefore execution is expected to complete correctly even without diversity.
3) At least one of the kernels produces erroneous results or simply does not terminate properly. Since different thread blocks of the redundant kernels are being scheduled at any point in time due to the staggering, a fault will not lead to the same erroneous decision for both redundant kernels, and hence, even if both of them may produce erroneous outputs, those will be naturally different so that the error will be detected, as needed to comply with ISO26262 requirements.

An additional consideration related to ISO26262 is the fact that a single independent fault must be considered subject to it being timely detected. This is particularly relevant for case (2) above, where the impact of a fault is only a decrease in the overall diversity, but not a functional failure. In that case, if permanent, the fault must be detected since, otherwise, multiple independent faults could be possible over a long period of time. Then, we could have that fault decreasing diversity, and another fault impacting results but escaping unnoticed due to the loss of diversity. To avoid this scenario, the kernel scheduler must pass tests periodically so that permanent faults are detected timely.

### 4.4 Appropriateness of the Scheduling Policies

Both, *SRRS* and *HALF*, allow guaranteeing diverse redundancy for any kernel, regardless of whether it is short, friendly or heavy. However, there is a preferable policy for each type of kernel.

- Short kernels have short duration, but may use most of the GPU resources during that short time. Since kernels do not overlap, *SRRS* is generally inoquous because it serializes already serial kernel executions. However, *HALF* may increase the execution time of those kernels by limiting the number of SMs to use. Still, such impact is low in absolute terms due to the short duration of this type of kernels.
- Heavy kernels are serialized, at least to a very large extent, due to lack of resources to run them simultaneously. Given that *SRRS* imposes full serialization, its impact can only be low. Instead, *HALF* may have large impact on performance. By decreasing the number of SMs used by the kernel, execution time may increase significantly. Moreover, despite restricting the number of SMs, kernels may still run with high serialization due to lack of other resources such as, for instance, registers.
- In the case of friendly kernels, they naturally use up to half of the resources of any type, thus allowing both kernels to execute simultaneously. In this context, *HALF* limits the number of SMs per kernel to half the ones available, which, indeed, is an equal or higher number of SMs than needed. Therefore, *HALF*'s impact is expected to be tiny – if any. However, *SRRS* fully serializes kernels that can run simultaneously, thus harming performance noticeably.

In summary, *SRRS* is better suited for heavy and short kernels, whereas the best policy for friendly kernels is *HALF*. The type of kernel can be determined during system development, prior to deployment. Hence, we can determine a priori what policy to use for each kernel during operation. Such policy selection can be performed during operation by updating the corresponding configuration register prior to executing each kernel, which is not different from other features that can already be configured dynamically, such as activating/deactivating prefetchers, changing branch predictor policies, and the like.

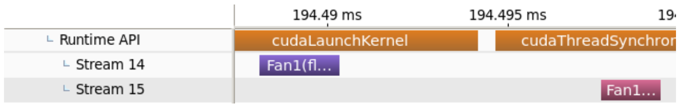
## 5 Software Solutions to Enable Diverse Redundancy on GPUs

This section describes a software-only solution to obtain diverse redundant kernel execution on a COTS GPU. While this solution is naturally more constrained than those allowing hardware modifications, it can be directly implemented and evaluated on top of a real COTS GPU rather than on a simulator.

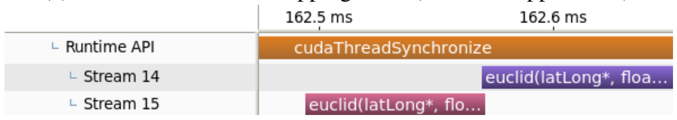
### 5.1 SM Sharing

Our software-only solution assumes that SMs cannot be shared across thread blocks from different kernels simultaneously. Instead, we assume that when a kernel starts running, it uses a number of SMs without interruptions (i.e., SMs are only released once they are not needed anymore), and during such period no other kernel can use those SMs. However, as shown in [16], the scheduling policy for some NVIDIA GPUs may allow, in some situations, sharing SMs across kernels if intra-SM resources allow it. Solutions avoiding this behavior relying on some hardware support (e.g., modifying the kernel scheduler) exist in the literature [17], and the ones presented in this paper would also solve this issue. However, our target in this section is to achieve diverse redundancy by software-only means. SM sharing across threads can be effectively avoided by building on *persistent threads* [18], [19], [20], where each SM can only be used exclusively by one kernel. Persistent threads bring some lack of flexibility since they impose a behavior similar to that of our HALF hardware solution, plus some overheads for the allocation and management of those threads (e.g., due to polling the GPU on the CPU side to detect the end of the kernel execution). Since, in general, SMs are rarely shared in practice, we avoid using persistent threads in our work and hence, we apply the same software architecture as for the hardware solutions, which includes: data replication, redundant kernel launching with a different stream, and result comparison at the CPU side.

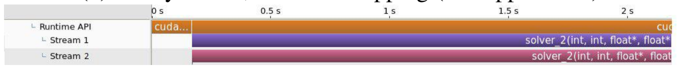
The types of kernels in our software-only solution are analogous to those of hardware solutions, but in this case, they can be observed on COTS GPUs with the NVIDIA Visual Profiler shown in Figure 7 for different Rodinia benchmarks [21]. The two bottom bars of each graph show the execution timespan for the redundant kernels, one in light red and the other in purple. Note that x-axis scale changes across graphs. In particular, it is $2.2\mu s$ for the first graph (short kernel), 0.1 ms for the second graph (heavy kernel), and 2,200ms for the third graph (friendly kernel). Like for the hardware approaches, solutions differ for each type of kernel:



(a) Short kernel, no overlapping at all (Gaussian application)



(b) Heavy kernel, small overlapping (NN application)


0 s
0.5 s
1 s
1.5 s
2 s
Runtime API
cuda...
Stream 1
solver_2(int, int, float*, float*
Stream 2
solver_2(int, int, float*, float

(c) Friendly kernel, large overlapping (Myocyte application)

Fig. 7: Timelines of redundant executions of Rodinia benchmarks [21] extracted using the NVIDIA Visual Profiler.

- *Short kernels*: These kernels can be run on the ASIL-D (lockstep) microcontroller due to their limited computation time. Using the ASIL-D microcontroller for their execution provides diverse redundancy as imposed by ISO26262. However, since execution time may raise noticeably in relative terms, whether the corresponding Fault Tolerant Time Interval (FTTI) is preserved needs to be assessed. We can expect the execution time increase to be affordable since short kernels must last very few $\mu$s to be short and do not overlap their execution on the GPU. Hence, even if their execution time increases by 1 or 2 orders of magnitude in relative terms, in absolute terms it should be comfortably below 1ms, which is a very short duration for critical tasks, whose usual duration is in the order of tens or hundreds of ms.
- *Friendly kernels*: In the case of friendly kernels, since they can execute concurrently, our software modifications are enough to execute them in different functional units (SMs). As explained before, due to the kernel launches' serializations, an initial staggering between the redundant executions is achieved naturally. Thus, obtaining a diverse redundant execution in a COTS GPU. Later, in the evaluation section, we will show evidence of this initial staggering. Note that, while such staggering is normally preserved due to the regular and highly deterministic execution of kernels on a GPU, it cannot be guaranteed a priori, and we can only assess it a posteriori to some extent.

  *Heavy kernels*: In the case of this type of kernels, redundant copies are serialized since each one requires more than half of at least one type of resources. By being serialized, redundant kernels may end up using the same components for redundant computations across kernels, thus challenging diversity. A simple solution for heavy kernels could be relegating them to execute on the ASIL-D microcontroller, as for short kernels. However, these kernels' execution time (heavy) can be arbitrarily large (e.g., tens or hundreds of milliseconds). Thus, slowdowns of 1 or 2 orders of magnitude would easily violate safety requirements for those systems. Instead, we introduce a new protocol to transform heavy kernels into friendly, thus, enabling them to be executed safely and timely, with diverse redundancy, on the GPU.

## 5.2 Heavy-to-Friendly Kernel Reshaping Protocol

Next we present our protocol to transform *heavy* kernels into *friendly* ones systematically. Our approach has been tailored to work with kernels not using shared memory for inter-thread communication since this is the type of *heavy* kernels found in our evaluation. If shared memory is used for inter-thread communication, then this would need to be managed manually. Extending our protocol to these scenarios is part of our future work. We first describe the operators on which our protocol builds, then the protocol itself, formal validation of its effectiveness, and finally, we discuss its complexity. In simple words, this protocol modifies each redundant kernel to fit in half of the SMs of a given GPU by reducing its parallelism and serializing some threads by combining them.

### 5.2.1 Thread Coarsening and Block Division Operators

Each of the redundant kernels uses a certain amount of resources (e.g., registers, shared memory). Whenever the requirements of the combined kernels in terms of resources exceed those available in the GPU (*heavy* kernels), the scheduler prevents them from achieving concurrent kernel execution between the head and shadow kernels. Our proposal is based on modifying these kernels' resource requirements, which may increase their execution time, but allows them to execute concurrently. Our protocol uses two techniques: *Thread Coarsening* and *Block Division*, which we introduce next.

**Thread coarsening:** Thread coarsening is the process of increasing the amount of work performed by each thread. This technique can cause some potential performance improvements: (1) Higher instruction-level parallelism (ILP) [22] by increasing the number of instructions per thread; (2) More efficient DRAM memory bandwidth utilization, by reducing the total number of memory-access instructions [23], for those data that would be otherwise fetched by more than one threads; and (3) Reduction in the number of computing instructions due to redundant computations across threads [24].

This technique, however, can also have several negative effects: (1) Reduction of the total amount of parallelism by reducing the number of threads, which can reduce the performance if there is not enough amount of work (threads) to keep the rest of the GPU busy; (2) Increase of the number of registers per thread; and (3) Worse memory access patterns since neighboring threads may end up accessing non-contiguous memory as stated in [25], which has detrimental effects on cache behavior.

Authors in [26] applied automatic thread Coarsening at compile time to GPU kernels to achieve a 1.3x speedup in a subset of Rodinia benchmarks. However, as mentioned before, thread coarsening may lead to increased cache pressure, thus resulting in performance degradation. Moreover, in the extreme degenerate case, thread coarsening would lead to sequential execution by a single thread, defeating the purpose of using parallel hardware such as GPUs. Therefore, while thread coarsening may produce some performance gains, in general, it is not used when the only concern is performance. However, our primary goal is not increasing performance but safety, by allowing redundant heavy threads to run concurrently (i.e., becoming friendly).

**Block division:** Block Division consists of splitting thread blocks into smaller ones, i.e., using fewer threads, while the total number of threads remains the same. This technique can be used when the

```
1: $\#SM_{available} \leftarrow \lfloor TotalSM/2 \rfloor$
2: $\#Register_{available} \leftarrow \lfloor Register_{SM}/2 \rfloor$
3: $\#SM_{Used} \leftarrow \#ThreadBlocks$
4: $\#Register_{UsedpBlock} \leftarrow Y$
5: while Kernels not concurrent do
6:     $TCF \leftarrow \lceil \frac{\#SM_{used}}{\#SM_{avail}} \rceil$
7:     $ApplyThreadCoarsening(TCF)$
8:     Recompute($\#SM_{Used}$), $1 \leq \#SM_{Used} \leq \#SM_{avail}$
9:     Recompute($\#Register_{UsedpBlock}$)
10:    if ($\#Register_{UsedpBlock} > \#Register_{available}$) then
11:        $BDF \leftarrow \lceil \frac{\#Register_{UsedpBlock}}{\#Register_{available}} \rceil$
12:        $ApplyBlockDivision(BDF)$
13:        Recompute($\#SM_{Used}$)
14:    else
15:        Done (Kernels concurrent)
16:    end if
17: end while
```

Fig. 8: Proposed protocol, where TCF = Thread Coarsening Factor and BDF = Block Division Factor

requirements per thread block exceed SM's resources, preventing the entire kernel from being executed.

This technique is particularly useful to reduce cache pressure and register requirements per thread block. Obviously, this is the remedy technique to use when thread coarsening leads to over-using some resources, and more particularly, registers over-use, since the lack of registers is the only limitation preventing the execution of a thread block. Other shared resources, such as cache space, can lead to lower performance if over-used but would not prevent a thread block's execution. Another limited resource is the shared memory, a scratchpad memory used to allow threads in the same block to communicate and reduce the DRAM bandwidth. We choose not to include it as part of our protocol since not using it will not impede the execution of a thread block, and can just slow down execution. As said before, performance is not our primary goal.

### 5.2.2 Protocol Step by Step

Figure 8 details the protocol followed to transform heavy kernels into friendly. It builds upon applying thread coarsening and block division iteratively until the resulting thread block does not exceed the number of registers available in an SM, and each kernel uses at most half of the resources available in the GPU. In particular, starting from a kernel whose thread blocks do not exceed the total number of registers available in an SM (so it is schedulable), but uses more than half of the registers of the entire GPU (so it prevents its shadow kernel from running concurrently), the protocol does the following: (1) merges threads so that repeated data fetches and computations can be removed, thus reducing the total number of registers required, although the number of registers per thread increases. (2) Divides thread blocks to decrease the total number of registers per block. Note that, as discussed later, this process necessarily decreases the number of registers per thread block in each iteration so that friendliness is achieved eventually.

The protocol first initializes the platform and kernel dependent variables and checks if the redundant kernels can already be executed concurrently or not (lines 1-5). If not, we compute the Thread Coarsening Factor (TCF) by dividing the current number of SMs used (= number of thread blocks) by the target number of SMs we want to use (up to half of those available in the GPU), as shown in line 6. Next, we apply thread Coarsening to reduce the number of threads by the factor computed (TCF), see line 7. We update the number of SMs used (line 8), which now will be less or equal to half of the SMs, and the number of registers required per thread block (line 9). At this point (line 10), if the number of registers per thread block is lower or equal to half of the SM's registers, kernels can execute concurrently (lines 14-15).

If the concurrent execution is not possible yet, then we require more registers per SM than allowed. Thus, Block Division must be used. First, we compute the Block Division Factor (BDF) based on the current register requirements and the registers available (line 11), and then we apply Block Division accordingly (line 12). This second step can increase the number of SMs used (line 13), which will require to perform again the thread coarsening technique. However, as shown next, the registers of the thread block are reduced with respect to the previous iteration, guaranteeing the convergence of the process.

### 5.2.3 Formal validation

We validate our protocol showing that a kernel can be made to fit in a single SM, so that if the number of SMs per kernel is higher, the protocol can converge faster.

Let us consider a kernel containing $B$ blocks, where each block has $T$ threads and each thread uses $R$ registers. An SM of the GPU contains $S$ registers being $S \geq R$. Then, the number of registers used are the total number of threads ($B \cdot T$) multiplied by the registers per thread ($R$), so $B \cdot T \cdot R$

If $B \cdot T \cdot R < S$, all thread blocks can be allocated in a single SM. Otherwise, we would use thread Coarsening with a TCF ($\alpha$), where $\alpha \in \mathbb{N}$ and $\alpha \geq 2$, to reduce the total number of registers. Now the total registers used are: $\frac{B \cdot T \cdot Y}{\alpha}$ where $Y$ is the registers used per thread after applying Thread Coarsening.

Due to register reuse, $Y$ can be at most $\alpha \cdot R$ (worst case), but it will be typically lower. In fact, appropriate compilation constraints may make merged threads run purely sequentially, thus reusing the same output registers for each instruction, and thus ensuring that $Y < \alpha \cdot R$. Therefore:

$$\frac{B \cdot T \cdot Y}{\alpha} < \frac{B \cdot T \cdot (\alpha \cdot R)}{\alpha} = B \cdot T \cdot R$$

Since $Y < \alpha \cdot R$, in the first step, we reduce the total number of threads, but also the total number of registers used. Now let us apply block division by a factor $\beta$, where $\beta > 1$ and $\beta \in \mathbb{Z}$. Now, the total number of registers is:

$$\frac{\beta \cdot B \cdot T \cdot Y}{\beta \cdot \alpha} = \frac{B \cdot T \cdot Y}{\alpha}$$

As seen, Block division does not affect the total number of registers used, only the registers per block ($\frac{T}{\beta} \cdot Y$). Therefore, at each iteration of our loop, which includes both Thread Coarsening and Block division, we reduce the total register requirements. Considering that the kernel could be executed in the CPU, where the register file is smaller than the one of the SM, the extreme case of just using one thread would be valid. Thus, our protocol will always allow two kernels to be executed concurrently in a GPU with at least two SMs (one for each redundant kernel).

### 5.2.4 Complexity

The protocol is guaranteed to terminate, as explained above, since, in every iteration, we reduce the total number of threads of the kernels and, eventually, we will reach the case with one thread per kernel, which is always a functionally valid option. However, such a degenerate case may be far from being the optimal solution in terms of performance.

The number of iterations required to reach a solution where both threads can be executed concurrently depends on the particular kernel being considered. The factors affecting the number of iterations relate to (1) the requirements of the initial kernels (number of SMs used, the total number of threads, number of thread blocks, ...), and (2) the resources available (number of SMs, the maximum number of registers per block ...).

Generally, kernels with higher requirements per block will need more iterations since thread coarsening will easily make kernels exceed the resources per block limits, and will take more iterations to find a valid pair of values for TCF and BDF. To reduce the number of iterations, one could be more aggressive when calculating the two factors by selecting higher values for both factors. Once a valid solution is found, it would be a matter of checking intermediate values not evaluated to search for a likely better solution with a higher number of threads. Instead, kernels with small requirements will take fewer iterations to find a valid solution. In fact, as we see in the evaluation section, in most cases, one iteration of the loop is enough to achieve a valid solution.

Formally, after adjusting the number of SMs used to be at most half of those available in the GPU, the number of registers required per SM is:

$$\#Register_{UsedpBlock} = R \cdot \left\lceil \frac{\#SM_{used}}{\#SM_{avail}} \right\rceil$$

The worst case occurs when $R$ matches the total number of registers per SM, thus leading to the largest register per SM exceedance after thread coarsening. As explained before, in each iteration, the total number of registers per thread decreases at least by 1 given that $Y < \alpha \cdot R$. Thus, the worst case would be that to move from $\#register_{UsedpBlock}$ to $R$ in $\#Register_{UsedpBlock} - R$ iterations assuming that the number of SMs used is fixed. In practice, since the number of SMs that can be used to have a friendly kernel is between $\#SM_{avail}/2$ and $\#SM_{avail}/4$ SMs[5], the number of steps could almost double if the worst case occurs (i.e. if we end up using $\#SM_{avail}/4+1$ SMs), thus leading to up to $2 \cdot (\#Register_{UsedpBlock} - R)$ steps.

## 6 Evaluation

In this section, we evaluate the different proposals individually, and then we compare them on a common ground. We first describe the experimental setup for the hardware solutions and the software-only solution. Then, we provide results for the hardware solutions, and for the software-only solution. Finally, we compare side-by-side all solutions on a simulator.

### 6.1 Experimental Setup

In order to evaluate our proposal, we have adapted the Rodinia benchmark suite [21], [27] with the modifications explained before to enable software-based redundancy, as shown in the example in Figure 5. Since Rodinia benchmarks are representative of general GPU computations, we decided to add an additional benchmark which is more related to our target application. In the absence of a GPU benchmarking suite for automotive systems, we included the cifar_10_multiple complex application from the GPU4S Bench open source benchmarking suite [28], which targets GPU computations in another safety critical domain, space. This benchmark was developed in the GPU4Space project [29] funded by the European Space Agency (ESA) which it evaluating the applicability of GPUs in the space domain. This benchmark, implements a complex inference application used for computer vision problems similar to the one found in the perception system of an autonomous driving vehicle, to classify an image between 10 different categories and it operates over multiple images. In particular, the application consists of multiple neural network layers of various types eg. fully connected, convolution, max pooling etc, which are used together to form a neural network to perform an image classification task. As such, this benchmark contains significantly more kernel invocations compared to the Rodinia benchmarks.

For the hardware solutions, we use GPGPUSim [30] (version 3.2.2) modeling a COTS GPU, an NVIDIA 1050 Ti with 768 Pascal based CUDA cores grouped into 6 SMs, with 4GB GDDR5 memory. We use a Pascal-based NVIDIA COTS GPU, the same GPU micro-architecture used in the NVIDIA PX2 AutoChauffer product, found in modern high-end cars, and only available to affiliated NVIDIA automotive partners. For our software-only solution, we use a system that includes the same COTS GPU. In particular, we use a system with an AMD Ryzen 7 1800x CPU, an NVIDIA 1050 Ti GPU and 64GB of DDR4 memory.

### 6.2 Staggering Measurement Experiment on a COTS GPU

Diversity requires both, time and space diversity. This section evaluates the impact of serial kernel launching on staggering (i.e. on time diversity), whereas following sections focus on space diversity mainly. In particular, this section presents the results from an experiment measuring how serial kernel offloading favors staggering.

The kernel off-loading process executes the following routines: Configure Call, Kernel Setup Arguments, and CUDALaunch. In this experiment, we modified myocyte benchmark (part of the Rodinia Benchmark Suite) to make its kernels redundant and executed it 100 times. We used the NVIDIA profiler and obtained the results shown in Figure 9. The dark thick line corresponds to the time elapsed between the start time of both kernels, the original and redundant ones. Stacked bars show the individual contribution of each one of the kernel off-loading routines for the second kernel, which are executed serially on the CPU. There is some code in between those routines (CUDA calls), whose execution time covers the gap between the stacked bars and the total execution time (thick line). However, the NVIDIA profiler does not provide information about this non-CUDA code.

Kernels are launched on the GPU only after these CUDA calls and surrounding code are executed on the CPU. The dominant routine (CUDALaunch) takes around $6\mu s$ (if not more), and its execution time is independent on the characteristics of the kernel to be launched. This guarantees that there always be such staggering across kernels, although it will be typically higher due to the remaining code executed for off-loading purposes. Hence, the staggering across redundant kernels is guaranteed to exist. Since this behavior is not specific of this GPU but, instead, is intrinsic to the CPU-GPU relation, we can expect analogous behavior for different GPUs and even different runtimes (e.g. OpenCL).

5. Our mechanism decreases $\#SM_{used}$ by an integer factor to not exceed $\#SM_{avail}/2$. If the value obtained was not exceeding $\#SM_{avail}/4$, then the factor used could be doubled without exceeding $\#SM_{avail}/2$.

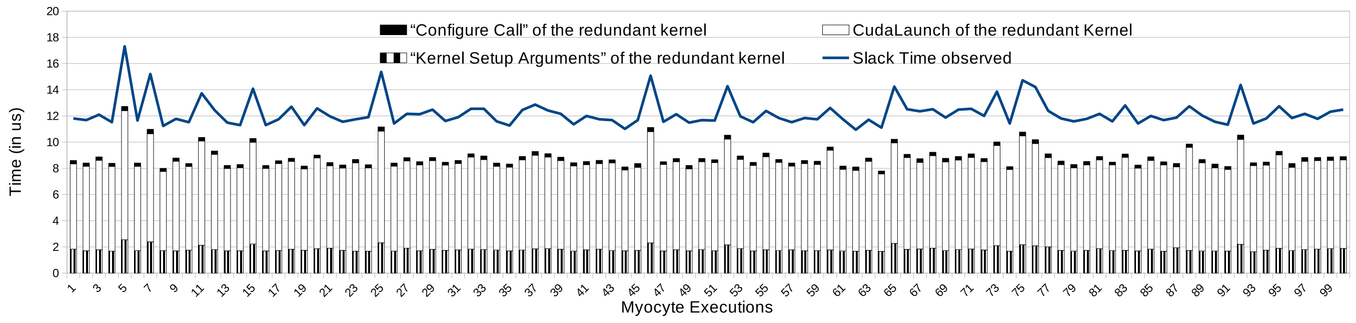


Fig. 9: Slack observed and subprocedures of the kernel launching for the consecutive executions of the Myocyte kernel

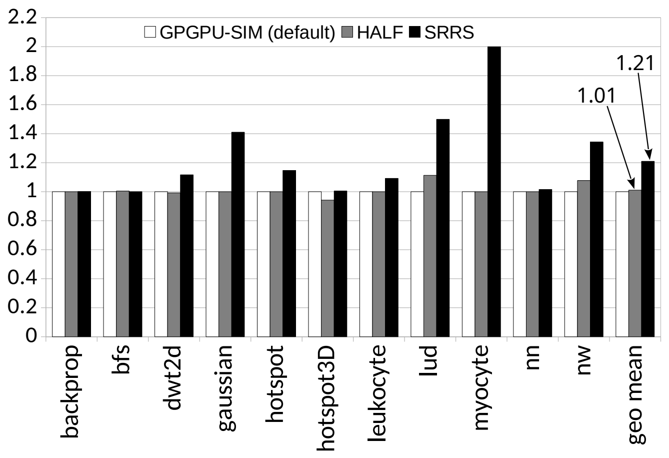


Fig. 10: Scheduler simulations using GPGPUSim

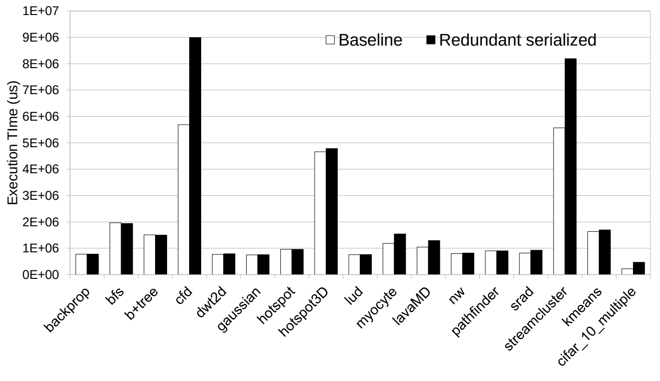


Fig. 11: SRRS implementation by serializing redundant kernels

## 6.3 HW Solution - Simulation Results

To evaluate *SRRS* and *HALF*, we have introduced modifications in GPGPUSim's [30] scheduling policy to assign SMs to as dictated by the proposed HW solutions. For instance, in the case of *HALF*, we build on the default GPGPUSim scheduling policy but limiting each kernel to use just half of the available SMs. We compare the performance of the default GPGPUSim scheduling policy using all SMs (6) without any specific limitation against that of *SRRS* and *HALF* policies.

Figure 10 shows results for the different scheduling policies w.r.t. those of the default GPGPUSim policy executing the applications modified accordingly to perform redundant kernel execution.[6] Given that most benchmarks have friendly kernels and experiments on GPGPUSim with the full benchmarks are very costly, we evaluate all short and heavy benchmarks, but only a subset of the friendly kernels for which no further insight is obtained by running more of them. Results show that the performance of *SRRS* and *HALF* is very close to that of the default scheduler, with the only exception of *myocyte* benchmark with *SRRS*. In particular, slowdowns are negligible for 9 out of the 11 benchmarks evaluated in the case of *HALF*, and the highest slowdown is only 10% for *lud*. In the case of *SRRS*, slowdowns are higher due to the serialization imposed by this policy, reaching 99% for *myocyte*. Since most of the kernels are friendly, they naturally require less than half of the SMs. Therefore, *HALF* is virtually inoquous and performance penalties are tiny. However, *SRRS* is particularly detrimental for friendly kernels since those kernels could naturally overlap their execution and, instead, *SRRS* fully serializes them. On the other hand, *SRRS* removes interference across kernels (e.g. in shared caches). However, the detrimental impact of serialization is generally much higher than the gains due to avoiding inter-kernel contention. Note that, instead, performance degradation for some kernels, such as *backprop*, with *SRRS* policy is negligible given that they are short and hence, their natural execution is already serialized.

6. GPGPUSim v. 3.2.2 requires applications to be compiled using cuda 4.0, which is the 2011 version. Although we have been able to compile all the applications, some of them (e.g. cifar_10) are not able to finish the execution on the simulator, even with the default scheduler, probably because of some not supported functionalities. However, all applications run without problems on the real GPU device, the NVIDIA 1050 Ti.

## 6.4 HW Solution - COTS Results

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 kernel's 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 does not provide control over the SMs used by a kernel.

Figure 11 compares the end to end execution time of the benchmarks with redundant kernels serialized and without kernel redundancy. By running on a real platform, we could afford to run

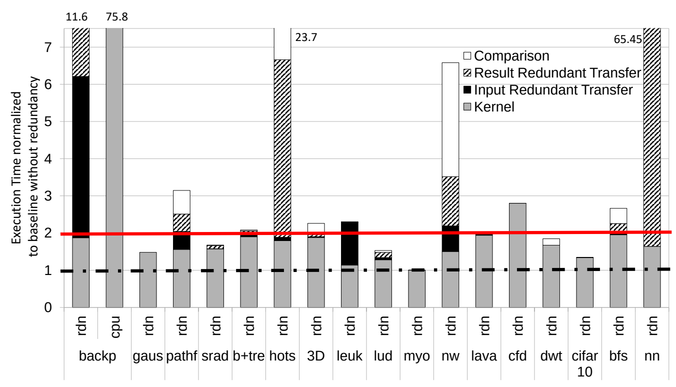


Fig. 12: Redundant execution times characterization for the Rodinia benchmark suite. Backprop and gaussian are short kernels; *nn* and *bfs* are heavy kernels; and the rest are friendly kernels.

all benchmarks timely. The bars in the plot show the minimum execution time out of 100 executions for each version. As shown in the plot, the redundant serialized execution 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 10.
2) The contribution of the kernel execution to the total execution time of the benchmark is relatively low in general.
3) The cost of sending input and output data twice and comparing the kernels' outputs 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 on 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.

## 6.5 SW-only Solution - COTS Results

Since the software-only solution does not require any hardware modification, we can directly evaluate it on the COTS platform. We have characterized the different parts of the redundant execution process in the Rodinia benchmarks as can be seen in Figure 12[7]. Execution times are normalized per each benchmark where "1" corresponds to the original benchmark's normalized execution time without redundancy. More precisely, we have characterized the execution into kernel execution, the redundant transfers, and the comparison phase, thus, showing all the overheads created by our strategy. The backprop benchmark also includes a CPU-only version, which we also included (2nd leftmost bar), as discussed next.

7. In this experiment we changed the inputs of the bfs benchmark in order to obtain another *heavy* kernel for the later evaluation of the heavy-to-friendly kernel transformation protocol.

| Default (heavy) configuration | | | | |
|---|---|---|---|---|
| Benchmark | Thread Blocks | Threads x Block | Total Threads | Registers Block |
| bfs | [1954 , 1] | [512 , 1] | 1,000,448 | K1:8,192, K2:7,680 |
| nn | [2560 , 1] | [256 , 1] | 655,360 | 3,840 |
| Final (friendly) configuration | | | | |
| bfs | [3 , 1] | [512 , 1] | 1,536 | K1:8,192 , K2:7,680 |
| nn | [2 , 1] | [256 , 1] | 768 | 3,840 |

TABLE 1: Default configuration of the applications that produces *heavy* kernels on the NVIDIA GTX 1050 Ti.

**Short Kernels (backprop and gaussian):** The backprop benchmark (the leftmost one) is a short kernel. As shown, the redundant version of this benchmark leads to an execution time above 2x the execution time without redundancy since redundant threads do not overlap. Due to the short duration of the kernel, the relative impact of needing redundant data transfers and having to compare results is huge w.r.t. GPU execution without redundancy. The CPU version of this benchmark, which is included in the benchmark suite, has an execution time 23x higher than the one for the GPU version. Such slowdown, despite huge in relative terms, is low in absolute terms, and hence, affordable. In the case of *gaussian*, the slowdown due to running it redundantly is below 2x since the execution time includes both, kernel launching in the CPU and kernel execution in the GPU. Hence, while kernel execution of both redundant copies does not overlap, the kernel launching of the second kernel overlaps with the kernel execution of the first one.

**Friendly Kernels:** The overlap of these kernels is large, and thus, the overall execution time to run both redundant kernels is far below 2x the execution time of the non-redundant kernel. Redundant execution for friendly kernels causes small overheads, most of which relate to the comparison of the results and to the data transfers. Those overheads could be reduced by performing comparisons redundantly in the GPU to reduce the amount of data to be transferred back to the CPU, and to parallelize the comparison. However, while this would be possible, it has not been explored explicitly in this work.

**Heavy Kernels (nn and bfs):** Since the redundant kernels for these benchmarks barely overlap, the impact of running them redundantly is above 2x in terms of execution time. However, the protocol introduced before allows converting these heavy kernels into friendly ones. Their friendly versions are evaluated in the next section.

## 6.6 Heavy-to-friendly Protocol Evaluation

As seen, most of the benchmarks turned out to be either friendly or short, and we only observed one *heavy* benchmark `nn`) for the 1050 Ti. To test the protocol with more workloads, we have modified the input variables of the benchmark `bfs` to make it also *heavy*.

The default grid configurations of the two applications are shown at the Table 1 (top rows). We obtain all the information shown in the Table through Nvidia's profiler nvprof.

`Bfs` contains two kernels with the same grid and block configuration, one using 16 registers per thread and the other 15. Instead, `nn` contains only one kernel that uses 15 registers per thread. Both applications use a high number of thread blocks, which results in a big TCF, 653 for `bfs` and 854 for `nn`. We obtained these factors by dividing the number of thread blocks by

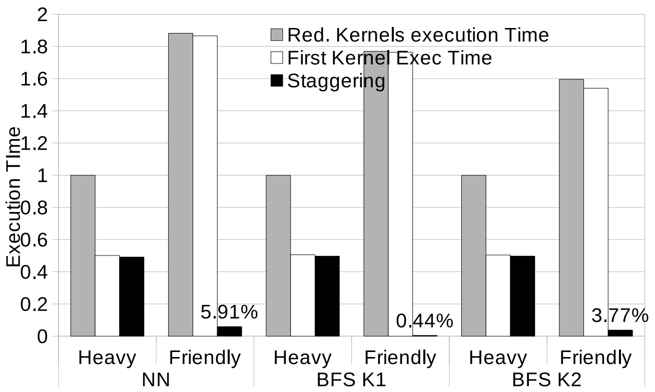


Fig. 13: Execution times for the total redundant execution, the first kernel launched and staggering w.r.t. total kernel redundant execution (*heavy*) of each benchmark.

half of the GPU's SMs (thus, 3 SMs for each redundant kernel), line 6 in Figure 8.

To facilitate the application of the protocol, we have adapted the code to enable Thread Coarsening and Block Division as follows:

1) Add a new parameter to the kernel function, the TCF.
2) Apply Thread Coarsening to the kernel code. In order to facilitate programming, we add an outer loop that iterates TCF times. However, while this automates the application of Thread Coarsening, this solution may not benefit from some optimizations. For example, memory instructions from originally different threads, accessing the same data, will not be performed closely because of the loop's body, which may lose potential cache hits. In order to improve performance, we recommend using the compiler technique in [26] whenever possible, although it makes less straightforward applying Thread Coarsening.
3) Modify the kernel launching, in the CPU code, to launch the kernel with the grid according to the Thread Coarsening and Block Division factors.

Using the calculated TCF, the benchmarks were launched. As expected, the execution of the kernels finished correctly, and the execution of the redundant kernels overlapped. In particular, we measured the execution time of the first launched kernel (white bar), the total kernel execution time (from the starting of the first until the completion of the second, grey bar), and the staggering time between them at the launching (black bar). Results for the two benchmarks are shown in Figure 13 normalized w.r.t. the execution time of the redundant kernels w.r.t. their baseline (*heavy*) state. The results shown for each benchmark are the average of 500 executions in the same COTS GPU used before, an NVIDIA GTX 1050 Ti.

As shown, in all heavy configurations of both benchmarks, the first kernel takes half of the total execution time, matching with the staggering time, meaning that both redundant kernels take a very similar amount of time to execute and are fully serialized. In the case of the friendly versions of the benchmarks, we observe that the first and total execution times nearly match, thus meaning that both redundant kernels finish virtually at the same time. Also, the fact that the staggering time is tiny indicates that both of them start almost simultaneously, thus overlapping their execution completely, with just some little staggering.

Note, however, that making kernels friendly impacts total execution time, which grows by a factor of 1.9x for `nn` and 1.7x and 1.6x for `bfs` kernels. While this effect is undesirable, it is the price to pay to guarantee diverse redundancy on a COTS GPU without explicit lockstep support and only by software means. Such performance loss relates to (1) worse cache access patterns that lead to an increased miss rate, and thus less efficient DRAM bandwidth utilization, whose access latency cannot be effectively hidden, and (2) the loss of parallelism since we reduce the number of parallel threads per kernel. For the sake of completeness, Table 1 (bottom rows) shows the final kernel configurations for both benchmarks after applying the protocol to make them friendly.

## 6.7 Fault Injection Campaign

To test the fault tolerance capabilities of our solutions, we have used the NVBitFi [31] a framework which is built on top of NVidia Binary Instrumentation Tool (NVBit) [32] that performs error injection campaigns for GPU application resilience evaluation. NVBitFI injects errors into the destination register values of a dynamic thread-instruction by instrumenting instructions after they are executed, only one injection is done per run. A dynamic instruction is selected randomly from all dynamic kernels of a program for error injection. This tool allows 4 different instructions to inject errors: writes to general purpose registers, single precision and double precision floating point instructions, and load instructions. Additionally, the tool supports four different fault models: one bit-flip in a register, bit-flips in two adjacent registers, random value in one register, and zero out the value in a register.

For the injections, we have selected one of the Rodinia benchmarks (backprop). This application contains two kernels which we will see later that matters when classifying the injection outcomes. For each fault model available and instruction type we have performed 10k injections on the baseline application and 10k for the application with our redundant kernel software approach. Since this application only uses single precision floats, double precision injections have been discarded[8]. For each execution, the output is analyzed and compared against a golden output (an output from an error-free execution).

Results from the fault injection can be observed in Figure 14. For each fault model, we can see three pairs of columns, each corresponding to a different instruction type targeted by the fault injection tool. From left to right, floating point instructions (32 bits), load instructions, instructions that write into general purpose registers[9]. For each pair, we have the baseline application on the left (*BAS*) and the application with our software-only redundant strategy on the right (*RED*).

Results follow similar behavior for all fault models. Baseline applications reported no Silent Data Corruption (SDC) for floating point injections or load instructions. Instead, injections on general purpose registers experienced a slight percentage of SDCs undetected from 10.7% to 24% on the baseline approach which are translated into SDC detected by our fault detection mechanism (comparison of kernel results). Detected Uncorrectable Errors (DUEs) only have a significant importance for general purpose registers and have similar values for the baseline and redundant application. The tool detects them by using the *dmesg* command, but the application could also detect them if the appropriate CUDA error handling procedures are called (e.g. *cudaGetLastError()*). In

9. 64-bit floating point instructions are not used by the application targeted, so we did not perform a fault injection campaign aiming at them.

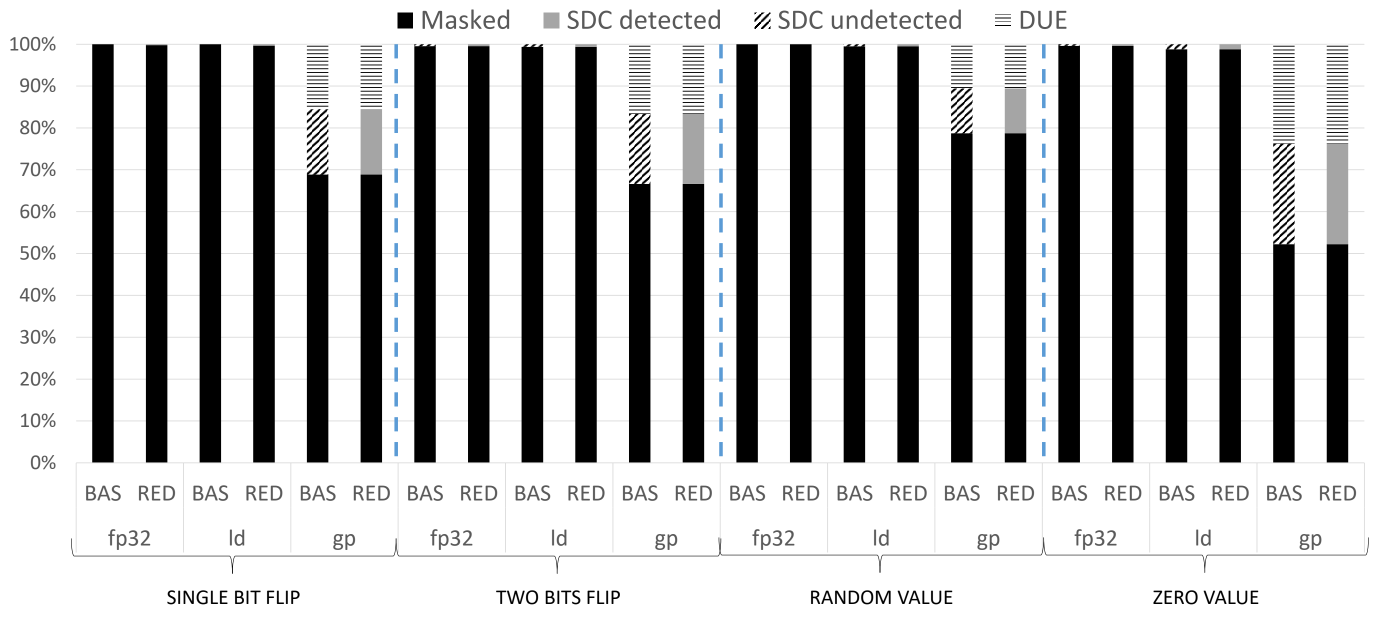


Fig. 14: Fault injection results for each fault model. **Masked**: Output of the execution was correct. **SDC detected**: An error was found by the detection mechanism an reflected in the output of the application. **SDC undetected**: Mismatch in the output was found which was not detected. **DUE**: A detected error prevented to finish the execution.

summary, we observe our redundant diversity scheme provides protection against single point faults.

### 6.8 HW and SW Solutions side by side

Last but not least, we show the execution times of all the solutions side by side. Since hardware solutions could not be integrated in a COTS platform since the kernel scheduler cannot be modified (at least to mimic HALF timing behavior), we perform this evaluation on GPGPUSim. In this experiment, we show our solutions' execution time, the baseline non-redundant version, and the simple redundant version, which only guarantees diversity for the *friendly* kernels. Results are shown in Figure 15.

Note that there is not a SW-based bar for backprop. Backprop is a *short* kernel, so this software-only solution is to be executed only on the CPU, but the simulator only models the GPU cycles. For this reason, we have not considered the backprop results when calculating the geometric mean, which are the rightmost bars labeled as GEO, for any of the solutions.

Generally, the HALF approach is the one that suits best, since most of these kernels are *friendly* and serializing them (SRRS) ends up in a longer execution time. The software-only solution also fits well when dealing with *friendly* kernels since the simple redundant version is applied. For heavy kernels (nn and bfs), we see that nn execution time is increased up to 2.6x whereas bfs obtains 1.8x, similar results to the ones tested on the real platform. Bfs takes advantage of the coalesced memory accesses and that fewer threads are competing to access memory. With this, the software solution obtains the best performance in this particular workload.

## 7 Related Work

Some authors assessed the effectiveness of FPGA, ASIC, and GPU designs for AD applications [33]. The suitability of GPU

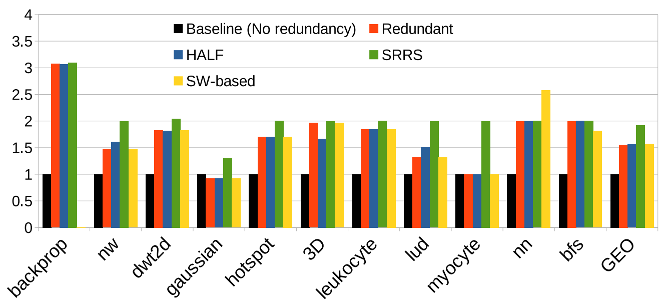


Fig. 15: Simulator Cycles of all the solutions

utilization in the context of safety-critical applications from the point of view of real-time performance has been assessed in several works [34], [7].

Redesigning GPUs, based on the reliability required for ASIL-D certification, has been regarded as too costly. Therefore, commercial platforms such as RENESAS R-Car H3 [1], and NVIDIA Xavier [35] targeting the automotive domain, include a general purpose high-integrity microcontroller together with a COTS GPU. Thus, in order to achieve ASIL-D fail-operational capabilities, these platforms rely on diverse software implementations of complex algorithms or fully redundant SoCs, which comes at the expense of drastically increasing the design and V&V costs in the former case, and the hardware cost and reliability concerns in the latter case.

Previous works use spatial partitioning to improve multi-tasking performance on a single GPU [36], [37] by exploring the scheduling per SM. Instead, Wu et al. [38] use a method that enables program-level spatial scheduling on the GPU by using SM-centric program transformations, which allow executing kernels in the desired SMs. Pai et al. [39] focus on enabling better multi-application concurrency by modifying the GPU runtime to avoid serialization of memory transfers and kernel executions. They also developed the idea of *elastic kernels*, by modifying the logical threads to avoid underutilization of the GPU resources and improving the concurrency of multiple kernels. Jain et al. [40] use a software-only technique to partition the GPU in order to execute multiple kernels without interference. Although redundant kernels are not evaluated, computing and also memory partitioning, by using memory coloring, could be employed in our work if we would like to replicate the input data of the redundant kernels. However, none of those solutions provides redundancy per se, neither provides any means to guarantee diversity since those solutions do not target critical real-time systems.

Some works have been performed in the high-performance domain targeting reliability by creating RMT (Redundant Multi Threading) in a GPU [41] or using automatic compiler transformations to transform GPU kernels into redundantly threaded versions [42]. However, none of those solutions guarantees diversity, as needed for ASIL-D automotive systems. Overall, our work is the first attempt to deliver diverse redundancy on GPUs, as needed to reach ASIL-D requirements in automotive systems.

## 8 Conclusions

The use of GPUs for highly-critical autonomous driving (AD) software poses a number of functional safety requirements for GPUs' design and utilization. In this work, we propose to exploit the intrinsic redundancy inside GPUs to achieve diverse redundancy, as needed for ASIL-D software components. With this idea, we present multiple solutions to achieve it, either by software-only means or by introducing minimal hardware modifications only in the kernel scheduler.

The software-only solution requires an early inspection of the kernel and its behavior on the desired platform since only *friendly* kernels are guaranteed to be executed in a diverse redundant manner. To solve this, we also presented a protocol to transform any *heavy* kernel into a *friendly* one based on the specification of the COTS GPU targeted. Smaller kernels (*short*) can be executed directly on the safe CPU side. Thus, this work delivers a full software-only solution for any given kernel.

Instead, both hardware solutions proposed in this work, namely SRRS and HALF, achieve diverse redundancy for any given kernel without additional software transformations. However, we show that it is convenient performing the same kernel classification as for software solutions in order to select the hardware solution that better suits each kernel in terms of execution time. In general, *friendly* kernels are executed faster when using HALF rather than SRRS, whereas *heavy* kernels may run faster with SRRS.

We have evaluated each solution alone and finally compared them side by side on a GPU simulator. As expected, hardware solutions generally offer lower execution times, but software solutions can instead be used right away in COTS GPUs.

## References

[1] "RENESAS R-Car H3," https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html.

[2] 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/

[3] 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.

[4] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, "High-integrity gpu designs for critical real-time automotive systems," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 824–829.

[5] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, "Software-only diverse redundancy on gpus for autonomous driving platforms," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 90–96.

[6] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, "Software-only triple diverse redundancy on gpus for autonomous driving platforms," in *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, 2020, pp. 82–88.

[7] T. Amert et al., "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *RTSS*, 2017.

[8] NVIDIA, "CUDA C PROGRAMMING GUIDE," 2019, https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[9] H. Jeon and M. Annavaram, "Warped-DMR: Light-weight error detection for GPGPU," in *Proceedings - 2012 IEEE/ACM 45th International Symposium on Microarchitecture, MICRO 2012*, 2012.

[10] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for GPGPU reliability," 2009.

[11] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for GPU error detection," in *Proceedings - International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018*, 2019.

[12] S. D. Carlo, G. Gambardella, I. Martella, P. Prinetto, D. Rolfo, P. Trotta, and P. Di Torino, "An improved fault mitigation strategy for CUDA Fermi GPUs An improved fault mitigation strategy for CUDA Fermi GPUs," Tech. Rep., 2014.

[13] J. E. Rodriguez Condia, P. Narducci, M. S. Reorda, and L. Sterpone, "A dynamic hardware redundancy mechanism for the in-field fault detection in cores of GPGPUs," in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2020.

[14] J. Fu, Q. Yang, R. Poss, C. R. Jesshope, and C. Zhang, "On-demand thread-level fault detection in a concurrent programming environment," in *Proceedings - 2013 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2013*, 2013.

[15] M. B. Sullivan, S. K. S. Hari, B. Zimmer, T. Tsai, and S. W. Keckler, "SwapCodes: Error codes for hardware-software cooperative GPU pipeline error detection," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2018.

[16] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh, "Constructing and characterizing covert channels on gpgpus," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[17] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[18] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[19] T. Allen, "Improving Real-Time Performance with CUDA Persistent Threads (CuPer) on the Jetson TX2," Concurrent Real-Time, Tech. Rep., March 2018, https://www.concurrent-rt.com/wp-content/uploads/2016/09/Improving-Real-Time-Performance-With-CUDA-Persistent-Threads.pdf.

[20] N. Capodieci and P. Burgio, "Efficient Implementation of Genetic Algorithms on GP-GPU with Scheduled Persistent CUDA Threads," in *Proceedings - International Symposium on Parallel Architectures, Algorithms and Programming, PAAP*, 2016.

[21] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[22] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC*, 2008.

[23] Y. Yang et al., "A unified optimizing compiler framework for different gpgpu architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 2, 2012.

[24] A. Magni et al., "A large-scale cross-architecture evaluation of thread-coarsening," in *SC*, 2013.

[25] B. Merry, "Faster gpu-based convolutional gridding via thread coarsening," *Astronomy and Computing*, vol. 16, 05 2016.

[26] N. Stawinoga and T. Field, "Predictable thread coarsening," *ACM Trans. Arhcit. Code Optim.*, June 2018.

[27] S. Che et al., "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," *IISWC*, 2010.

[28] I. Rodriguez, L. Kosmidis, J. Lachaize, O. Notebaert, and D. Steenari, “Gpu4s bench: Design and implementation of an open gpu benchmarking suite for space on-board processing,” *Report*, 2019. [Online]. Available: https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html

[29] L. Kosmidis, I. Rodriguez, A. Jover, S. Alcaide, J. Lachaize, J. Abella, O. Notebaert, F. Cazorla, and D. Steenari, “Gpu4s: Embedded gpus in space - latest project updates,” *Microprocessors and Microsystems*, vol. 77, p. 103143, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933120303100

[30] A. Bakhoda et al., “Analyzing CUDA workloads using a detailed GPU simulator,” in *ISPASS*, 2009.

[31] NVLabs, “Nvbitfi: An architecture-level fault injection tool for gpu application resilience evaluations,” https://github.com/NVlabs/nvbitfi, 2020.

[32] O. Villa, M. Stephenson, D. W. Nellans, and S. Keckler, “Nvbit: A dynamic binary instrumentation framework for nvidia gpus,” *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[33] S.-C. Lin et al., “The architectural implications of autonomous driving: Constraints and acceleration,” in *ASPLOS*, 2018.

[34] M. Yang et al., “Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems,” in *ECRTS*, 2018.

[35] 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, 2018.

[36] N. K. J. Adriaens, K. Compton and M. Schulte, “The case for GPGPU spatial multitasking,” in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2012.

[37] D. B. J. Janzen and A. Hugo, “Partitioning GPUs for Improved Scalability,” in *Proceedings - Symposium on Computer Architecture and High Performance Computing*, 2016.

[38] B. Wu et al., “Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations,” in *Proceedings of the International Conference on Supercomputing*, 2015.

[39] M. T. S. Pai and R. Govindarajan, *Improving GPGPU Concurrency with Elastic Kernels*, 2013.

[40] S. Jain et al., “Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs,” in *RTAS*, 2019.

[41] M. Dimitrov, “Understanding software approaches for gpgpu reliability,” in *GPGPU Workshop*, 2009.

[42] J. Wadden et al., “Real-world design and evaluation of compiler-managed gpu redundant multithreading,” in *ISCA*, 2014.

## About the Authors

**Sergi Alcaide** received his bachelor’s and master degrees from Universitat Politècnica de Catalunya, Barcelona, Spain, in 2016 and 2018, respectively. Currently he is in the third year of his PhD in the CAOS (Computer Architecture and Operative Systems) group in the Barcelona Supercomputing Center. His main research interests includes fault-detection mechanisms to preserve functional safety in safety critical systems and GPUs.

**Dr. Leonidas Kosmidis** is a Senior Researcher at the Barcelona Supercomputing Center (BSC). He is the recipient of the RISC-V Educator of the Year 2019 Award from the RISC-V Foundation. He holds a PhD and a MSc in Computer Architecture from Universitat Politécnica de Catalunya (UPC) and a BSc in Computer Science from University of Crete. In 2013 he interned at the Media Processing Department of ARM Holdings at Cambridge. His research interests lie in the intersection of accelerators and critical systems. Dr. Kosmidis is the co-coordinator of the GPU4S (GPU for Space) ESA funded project.

**Dr. Carles Hernandez** is a senior Researcher at the Universitat Politècnica de València. Previously from 2012 to 2018 he was senior researcher at the CAOS group from Barcelona Supercomputing Center. In 2012 he worked as intern at the IP verification group at Intel Mobile Communications Munich. His area of expertise includes on-chip interconnects, processor design, real-time aware hardware design, and reliability. He is currently co-advising 5 PhD students. Dr. Hernandez is the project coordinator of the H2020 SELENE project on high-performance computing for safety-related applications.

**Dr. Jaume Abella** is a senior PhD. Researcher in the CAOS group at BSC. He worked at the Intel Barcelona Research Center (2005-2009). Since 2009 Jaume is the BSC Principal Investigator on a number of projects related to hardware and low-level software for safety-relates space, avionics and automotive systems such as ECSEL FRACTAL, H2020 DeRISC, H2020 SELENE, H2020 RECIPE, H2020 SAFURE, and ARTEMIS VeTeSS. He has authored +15 patents and +180 papers in top conferences and journals.