Software-only Diverse Redundancy on GPUs for Autonomous Driving Platforms

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

Abstract—**Autonomous driving (AD) builds upon high-performance computing platforms including (1) general purpose CPUs as well as (2) specific accelerators, being GPUs one of the main representatives. Microcontrollers have reached ASIL-D compliance by implementing diverse redundancy with lockstep execution. However, ASIL-D compliant GPUs rely on either fully redundant lockstep GPUs (i.e. 2 GPUs), which doubles hardware costs, or fully redundant systems with a GPU and another accelerator, which virtually doubles design and validation/verification (V&V) costs. In this paper we analyze the degree of diversity achieved when implementing redundancy on a single GPU, showing that diverse redundancy is not achieved in many cases, and propose software strategies that guarantee achieving diverse redundancy for any kernel on systems using commercial off-the-shelf (COTS) GPUs, thus showing how to achieve ASIL-D compliance on a single COTS GPU in controlled scenarios.**

I. Introduction

Autonomous driving (AD) systems require integrating a number of high-performance computing (HPC) platforms in the car. While performance provided by many existing High-Performance Computing (HPC) platforms suffices to meet the computing requirements of AD systems – as confirmed by existing AD systems demonstrations [1] – it is unclear how these computing systems may meet the highest Automotive Safety Integrity Levels (ASIL) as dictated by ISO26262 [2]. Thorough validation and verification (V&V) processes must be followed to retrieve evidence on whether safety requirements are effectively met under the most stringent, yet plausible, circumstances [3]. The level of assurance needed depends on the actual ASIL of the target application.

Automotive systems usually have a safe state upon a failure, either by stopping the car or by resorting to the driver to manage unexpected situations. This allows computation-related components not to be certified at the highest ASIL (ASIL-D) since ASIL-D monitoring facilities are in place to detect failures timely and transfer the system to a safe state within the fault-tolerant time interval (FTTI), thus being failures in the computation components an availability concern, but not a safety concern. Hence, specific ASIL-D cores are used to run monitoring software, which requires specific strategies for lockstep execution to provide redundancy and diversity, thus avoiding common cause failures (CCFs), whereas other components intended to deliver higher performance (e.g. accelerators) require cheaper safety measures (if any). However, safe states may not exist any longer in the context of AD since it may be unacceptable to transfer the control to a hypothetic driver. Instead, the system must keep operating correctly upon a failure, which makes that computing components in ASIL-C/D functionalities must also reach ASIL-C/D.

Processor manufacturers have already released a number of products targeting AD systems, such as the RENESAS R-Car H3 [4] and the NVIDIA Xavier [5] platforms among others. Those platforms include a number of general purpose cores (e.g. ARM-based) paired with some accelerators, being the GPU a key actor to process huge amounts of sensed data in AD. So far, those platforms have been regarded as ASIL-B compliant and have been claimed to enable ASIL-C/D, but based on details available, this is achievable, for instance, by using redundant functionalities (e.g. based on GPUs and Deep Learning accelerators) [6]. Unfortunately, fully-redundant functionalities double or triple the design and V&V costs, which is highly undesirable. Hence, it becomes critically important enabling some form of lockstep in the GPU part of AD platforms to reach ASIL-C/D to avoid using fully-redundant functionalities. Moreover, such lockstep operation must occur on-chip, as in the case of general-purpose cores (e.g. Infineon AURIX processors [7]), for efficiency and cost reasons, since setting up two GPUs increases hardware costs and reliability concerns.

In this paper we tackle this challenge by enabling diverse redundancy on a single GPU with software-only means. In particular, the contributions of this work are as follows:

- A thorough analysis of the features of GPUs, with focus on an NVIDIA representative, and how they enable or limit diverse redundancy.
- An analysis of some compute intensive applications on the GPU identifying different kernel categories depending on whether software redundancy achieves also diversity and, if not, the cause impeding to achieve diversity.
- Software strategies to achieve diversity on those kernels failing to achieve it, so that diverse redundancy is achieved for any kernel size.

Overall, our approach enables ASIL-C/D compliance on GPUs without needing fully-redundant systems, thus containing design and V&V costs.

II. Background on ISO26262

Safety-related automotive functionalities are classified into different ASIL based on their functional safety risks, from A to D, where ASIL-D corresponds to the highest safety risk. The higher the ASIL of an item, the more stringent the safety measures needed to avoid hazardous situations. For instance, error detection (e.g. lockstep execution) and recovery (e.g. reset and restart) features may be required to preserve safety of an ASIL-D microcontroller. Apart from detecting and correcting faults, safety measures may also need to adhere to a specific Fault-tolerant time interval (FTTI), which determines the maximum time allowed since the fault occurs until either the affected functionality delivers a correct output or a safe state is reached. Exceeding the FTTI may lead to a hazardous situation.

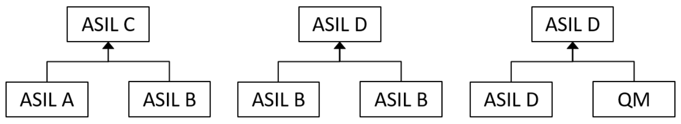


Fig. 1: Examples of ASIL decomposition.

A. *ASIL decomposition*

A given ASIL can be reached by using redundant items with lower ASIL if they are proven to be *sufficiently independent*. For instance, ASIL-C can be reached with ASIL-B and ASIL-A items (see examples in Figure 1). Such a solution is often used because lower ASIL items are cheaper to design and verify than higher ASIL ones. For instance, an ASIL-D CPU can be implemented with two ASIL-B processor cores operating in lockstep. To allow this decomposition, redundant cores must be proven to have independent behavior, i.e. a fault must not affect all cores in the same way. In particular, evidence is needed proving that a single fault (e.g. a voltage droop) does not lead redundant cores to the same failure, which might remain undetected. Hence, solutions such as *staggered execution* across cores are used to ensure that, upon a fault, cores perform distinct activities simultaneously and hence, faults cannot lead to identical errors in both cores, so that any error will be timely detected.

While redundancy with sufficient independence is needed in general to reach a higher ASIL by means of ASIL decomposition, it is explicitly requested in ISO26262 to reach ASIL-C/D.

Finally, ASIL decomposition is also used for cost reduction trading off availability. In particular, a component of a given ASIL (e.g. ASIL D) can be decomposed into multiple components, being at least one of them ASIL-D and the rest, lower ASIL or QM (Quality Managed), as shown in Figure 1. In this case, the ASIL-D component must be able to preserve safety despite failures of the other components. For instance, the computational part may be deemed as QM (no safety requirements), and monitoring capabilities as ASIL-D, being the ASIL-D component in charge of detecting failures in the QM component and transferring the system to a safe state within the FTTI. However, in the context of AD, a number of ASIL-D systems related to steering and braking become fail-operational so that a safe state does not exist and hence, their computational parts must remain as ASIL-D since correct operation is mandatory. In this case, ASIL decomposition can only be used to combine lower ASIL components providing sufficiently independent redundancy.For instance, if we combined an ASIL-D low-performance CPU with a QM GPU, being the latter in charge of running ASIL-D processes (e.g. object detection), on a GPU failure, the ASIL-D CPU would be able to detect it, but will not be able to guarantee safety due to the lack of a safe state (i.e. the car must keep taking driving decisions). Hence, the GPU must reach ASIL-D on its own.

B. *Redundancy, Diversity and Sphere of Replication*

Software faults and some hardware faults are regarded as systematic, and it must be proven that their failure risk is residual. However, random hardware faults cannot be avoided, and means are required to prevent them from causing hazards. Those faults can be caused by, for example, voltage droops, crosstalk, process variations, etc. In order to reach a given ASIL, it must be proven with appropriate diagnostic coverage and failure rate targets that any such single fault cannot lead the system to a hazard.

Diverse redundancy is used to avoid hazards due to a single random hardware fault (i.e. no CCF exists), since it allows proving that the effects of a single fault – if any – are different in the redundant copies so that the fault can be effectively and timely detected. For instance, Dual Core LockStep (DCLS) has been deemed as an appropriate solution in ISO26262. However, some random hardware faults (e.g. a voltage droop) could affect simultaneously and identically both cores. Hence, DCLS is often implemented with staggered execution so that relevant transient CCFs are avoided by construction (or their residual risk can be deemed as sufficiently low).

ISO26262 provides no explicit recommendations on how to assess whether diversity has been achieved to a sufficient degree, and quantifying to what extent two different implementations performing the same functionality are diverse is an open challenge [8]. Hence, diversity is typically assessed qualitatively by safety experts.

While different means exist to achieve redundancy and diversity, using diverse software or hardware designs may double design and verification costs due to having to build two different components for the same functionality. Hence, although DCLS execution also halves performance efficiency (the corresponding functionality is executed twice), it allows reusing the same design (e.g. the same core design) for the primary and the redundant paths (e.g. with staggered execution), thus containing design and verification costs.

Redundancy can be applied at different granularities according to the sphere of replication (SoR). Choosing the right SoR depends on several tradeoffs like area overheads, redesign costs, fault detection time, and overall system costs. In the context of DCLS, the SoR is placed at the level of the CPU (core), as done for the AURIX processors. This requires including two replicas of the same core and compare their memory transactions, which requires roughly duplicating computational resources in the chip and being able to ensure that replicas can provide independent behavior. On the other hand, storage (memories, caches) and communication means (buses, crossbars) do not need to be fully replicated and can build upon Error Correction Codes (ECC) and Cyclic Redundancy Check (CRC) as a form of lightweight redundancy with diversity.

III. Cost-Effective ASIL-D Capable HPC Automotive Platforms

HPC ASIL-D capable platforms typically combine a low-performance microcontroller amenable for the automotive domain (i.e. ASIL-D capable) and an HPC accelerator delivering high computation throughput, but whose adherence to ISO26262 requirements is unknown, so its appropriate use for ASIL-C/D systems needs to be investigated. Without loss of generality, we consider an NVIDIA GPU accelerator, thus analogous to those in NVIDIA Drive and Xavier families for the automotive domain. However, the findings in this paper can easily be extrapolated to other products.

In this platform, the sequential (*control*) code is executed in the microcontroller in lockstep mode to achieve diverse redundancy, as needed for ASIL-D compliance. Instead, complex and parallel algorithms required for the continuous rendering of the surrounding environment (e.g. object detection and tracking) among other functionalities are offloaded to the

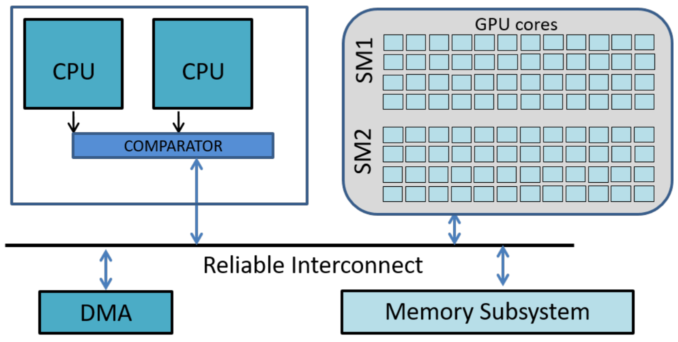


Fig. 2: Proposed Computing Platform architecture

GPU accelerator. Figure 2 shows a schematic of the proposed hardware platform.

In the rest of this paper, we investigate how to enable diverse redundancy on the GPU, analogous to that already had in the automotive microcontroller. In particular, we exploit the characteristics of kernel offloading and execution in accelerators (with special focus on a GPU) to reach the level of redundancy needed and appropriate diversity support for ASIL-D compliance.

Our proposal incurs lower design and V&V costs than having heterogeneous accelerators that would also require specific algorithm implementations, as has been recently proposed by NVIDIA [6]. Using multiple heterogeneous components (either at hardware or software level) requires designing, verifying and testing all those components, thus incurring high costs, which is against commercial interests. The other side of the coin is that achieving independent redundant execution with homogeneous components (i.e. the same software running in two identical hardware components or twice in the same hardware component) requires proving that the implemented diversity techniques suffice to protect the system against the relevant CCFs.

IV. Enabling ASIL-D GPU Operation

In this section we describe how to protect end-to-end the computations offloaded by the microcontroller onto the GPU. Our aim is to show that ASIL-B COTS GPUs can be used to reach ASIL-D without the need of using fully redundant systems. Due to space constraints, we assess dual modular redundancy; however, analogous reasoning can be applied in the case of triple modular redundancy.

A. *Offloading Process*

The computation offloading process from the microcontroller to the GPU has 3 steps: a) preparation of the offloading (① memory allocation and code/data transfer ② in Figure 4), b) kernel launching (③) and c) retrieval of the generated results (④ and ⑤ memory deallocation). The preparation step requires sending the code that has to be executed in the GPU and transferring the data from the microcontroller memory to the GPU memory. Although typically the microcontroller and the GPU share the same physical memory, each device retains its own mappings and separate address spaces. Hence, even though the data are not physically transferred, there is still a certain amount of bookkeeping and some data transferring required, e.g. due to cache flushing for consistency reasons, which makes this process not immediate.

The offloading process goes beyond the SoR of the microcontroller (e.g. the DCLS) and involves the memory and/or DMA controllers (see Figure 2). Memory data and on-chip communication during the execution phase occur on the same resources as those used by the ASIL-D microcontroller and hence, are protected by specific ECCs, as indicated before. Thus, data movements during the preparation of the offloading process, kernel launching and the retrieval are already protected by appropriate safety measures to comply with ASIL-D requirements. However, the redundant execution elements in the GPU used during the computation phase remain unprotected. To protect kernel execution, safety measures analogous to those in the microcontroller must be implemented during GPU operation, which can be achieved as follows (see Figure 3):

1) Two independent redundant kernels are set up.
2) Input data is duplicated.
3) Computation occurs in the GPU avoiding CCFs.
4) Output data from both redundant kernels is transferred back to the microcontroller for reliable comparison.
5) Comparison is performed in the ASIL-D compliant microcontroller (e.g. DCLS cores).

The steps above are performed on ECC/CRC protected communication means or on ASIL-D compliant microcontrollers, except GPU computation, which we analyze in the rest of this section. Note also that read-only input data could be used without replication. However, this would impose data protection in the GPU (e.g. ECC), which may not exist. Thus, for the sake of simplicity, we stick to fully redundant input data for the redundant kernels and leave considerations on non-replicated input data for future work.

B. *Exploiting GPUs Intrinsic Redundancy*

GPUs include abundant replicated resources such as, for instance, the elementary processing units – CUDA cores according to NVIDIA's terminology – that are in charge of the execution part of the GPU pipeline. As an example, the NVIDIA-Pascal GPU inside the Jetson TX2 (NVIDIA Driving platform) includes 256 CUDA cores divided into two Streaming Multiprocessors (SMs). SMs are also sub-divided into four components, each containing a single warp scheduler that is responsible for feeding 32 CUDA cores, 8 load/store units and 8 special function units. Thus, redundancy is intrinsic to the GPU architecture at multiple granularities: CUDA cores, subdivisions inside an SM, and SMs.

Several independent jobs can be executed concurrently in a GPU by using NVIDIA CUDA *streams*. A CUDA stream is a FIFO queue whose operations execute serially in the GPU. However, operations from kernels in different streams can overlap their executions. Kernels will execute, therefore, concurrently as long as there are enough resources for all of them. This, ultimately, enables redundant *simultaneous* execution, which helps achieving diversity since it may enforce the use of separate sets of resources by the two kernels, which diminishes the risk of CCFs by construction.

C. *Achieving Diverse Redundancy*

In the context of COTS GPUs, diverse (independent) redundancy can be achieved for two or more kernels if the following requirements are met:

Req1: Replicated computations do not use the same functional unit block (FUB) to execute the same code on the same data.

Req2: Functionally identical computation units (e.g. CUDA cores) produce different error manifestations in the presence of a single fault affecting several of those computation units.

```
//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. 3: Original and modified CUDA code

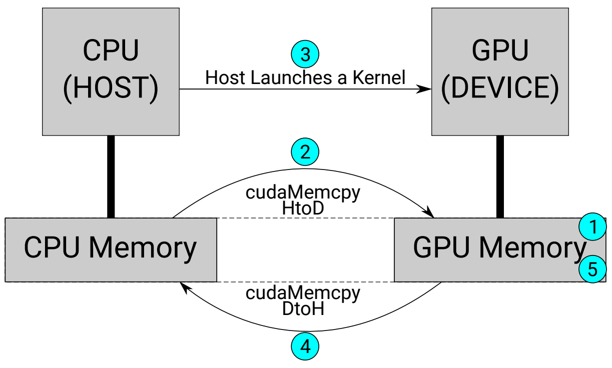


Fig. 4: Common CUDA Workflow

Req3: Unique resources (e.g. non-replicated buses or interfaces) implement intrinsic diverse redundancy (e.g. ECCs).

Req1 guarantees that permanent faults escaping testing or due to aging cannot cause the same error in the replicated executions. Achieving *Req1* requires having a kernel scheduling policy controlling which FUBs (SMs in NVIDIA GPUs) each kernel will use. This is currently done by the kernel scheduler whose policy is often unknown since some GPU vendors, such as NVIDIA, do not release this information [9]. In our evaluation, we show empirically that *Req1* is naturally achieved for those kernels that can run simultaneously. In general, we classify kernels into three categories regarding their potential concurrency:

- *Short* kernels: Kernels that are too small to run concurrently so that, by the time the second kernel is issued, the first one has already completed its execution.
- *Heavy* kernels: Kernels that use more than half of the shared resources of the GPU. Thus, two heavy kernels cannot fit at the same time on the GPU.
- *Friendly* kernels: Kernels that can run concurrently.

Note that kernel classification is platform and data-size dependent, so, for instance, a heavy kernel on a particular GPU could be a friendly kernel on another GPU with more resources. Instead, most automotive applications have a fixed data size since the input data comes always from the same sensor (e.g. images from a camera). This reduces the data size variability for most of the kernels. Short and heavy kernels challenge the achievement of diversity.

Solution for short kernels. Short kernels may be executed directly in the ASIL-D (lockstep) microcontroller, since they do not demand huge computation power. Executing them in the ASIL-D microcontroller guarantees *Req1*, although it must be assessed whether their likely larger execution time in the microcontroller still adheres to the corresponding FTTI. In general, this is the case since short kernels need to take at most few μs of execution in the GPU not to overlap, and even if they run 100x slower in the CPU, they will stay typically below 1ms only, which is a very low latency for functions executing typically every few tens of ms at most.

Solution for heavy kernels. Redundant copies for heavy kernels are executed sequentially due to lack of resources to run them concurrently, potentially using the same resources for the same computations of both copies, thus defeating diversity. However, heavy kernels run a number of threads in parallel. By reorganizing computations, some parallel threads can be serialized (e.g. splitting the kernel into multiple kernels or simply rearranging threads) so that the amount of resources is reduced sufficiently to allow two redundant copies to run concurrently, thus becoming the heavy kernel one or several friendly kernels. Such a solution is always feasible due to the nature of thread execution on GPUs because coherence across parallel threads is not controlled, so any sequential order of the operations across threads is semantically correct. Hence, by

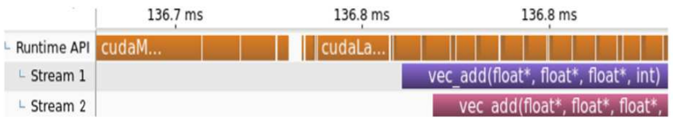


Fig. 5: Staggered kernel execution of a vector addition, obtained using NVIDIA's Visual Profiler

serializing parallel threads in any way semantics are preserved.

Solution for friendly kernels. Finally, in the case of friendly kernels, we can use CUDA Streams and launch each replica with a different CUDA Stream. Since SMs can only execute threads from the same CUDA Stream, the two redundant executions will use different SMs (FUB), thus guaranteeing *Req1*.

Req2 guarantees that a single fault does not lead to a CCF despite redundant computations occur on different resources. Hence, *Req2* imposes the use of some additional form of diversity. For instance, a usual technique to achieve diversity in functionally identical computational units is using staggered lockstep execution, so that redundant computations, apart from being performed on physically different resources, are also performed at different times. Interestingly, staggered execution, which we discuss in detail in the next subsection, can be implicitly achieved with COTS GPUs, and this provides them with protection against some of the most relevant sources of CCFs such as high-voltage pulses or voltage droops. Providing independent redundancy beyond the aforementioned CCFs could also be achieved using some of the techniques (e.g. layout diversity) already employed to achieve diversity in front of the most relevant CCFs in ASIL-D lockstep processors [7], [10]. However, this is not yet provided in COTS GPUs.

Req3 guarantees that diverse redundancy is achieved in unique resources. In general, those resources include interconnects and interfaces, where data and control signals transmitted can be properly protected with ECCs or CRCs. In the case of storage, either it is also ECC-protected or contents are stored redundantly. Finally, combinational logic is usually hard to protect with any form of ECC. For instance, this is the case of the thread scheduler. However, by making redundant kernels run simultaneously in different SMs, each kernel uses different thread schedulers. Moreover, the SM scheduler, used to send thread blocks to SMs, also operates with some degree of diversity since the same thread block in redundant copies is dispatched to different SMs. However, diverse redundancy requires physical replication in general. As detailed before, this is practically in place for computation resources in a GPU, but other components, such as the kernel scheduler, may lack such support. In general, whether unique resources adhere to specific ASIL requirements cannot be assessed by industrial users due to the lack of detailed documentation, and observability or controllability means. Thus, whether the failure probability of those components can be deemed as residual risk cannot be assessed directly with software only approaches. Still, our work shows that most GPU resources can be leveraged by software means to ensure diverse redundancy.

D. Implementing Staggered Kernel Execution

As explained, all kernels have been made friendly or made to run in ASIL-D CPUs (short kernels). However, we must guarantee staggered execution for friendly kernels. The kernel invocation performs the offloading of the application. Data sets transfer, explicitly initiated by the programmer, must be performed for both kernels before starting any of them to minimize the risk of having short kernels. In other words, the slack between the initiation of both kernels is kept as low as possible. Kernels have small constant and implicit parameters for each kernel launch. Those are set by a configuration call, which performs the arguments passing, followed by a `CUDALaunch` operation performed by the CUDA Runtime for each kernel called. The CUDA Runtime performs all those operations serially, since it is executed on the CPU, thus serializing the launch of the two kernels with some delay (*slack time*) between the two concurrent executions, as illustrated in Figure 5. The figure, generated with the NVIDIA Visual Profiler, shows the serialization of the CUDA calls (top yellow bar) and how the kernel copies start with some slack in-between (bottom purple and light red bars). Therefore, staggered execution start is guaranteed by construction. Note that, although identical kernels are expected to progress almost identically – thus preserving the staggering, this cannot be guaranteed in general due to the lack of controllability had in COTS GPUs. We leave for future work the analysis of how staggering is preserved during the execution. In the rest of the paper, we assume that such staggering had at the start is preserved during the rest of the execution.

V. Experimental Validation

A. Experimental Setup

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. We use an NVIDIA GeForce GTX 1050 Ti as an example of a COTS GPU. It contains 768 Pascal based CUDA cores grouped into 6 SMs and has a 4GB GDDR5 memory. Note, however, that our analysis holds directly for other NVIDIA GPUs, and can be extrapolated to GPUs from other vendors.

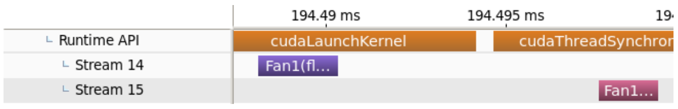
For benchmarking, due to the lack of AD or ADAS (advanced driver-assistance systems) benchmarks, we selected a widely used benchmark suite in the GPGPU domain, the Rodinia Benchmark Suite [11], [12][1]. For a more comprehensive evaluation, we are currently in the process of generating benchmarks for the GPU parts of the Apollo AD framework [14], which is a cumbersome process due to the high degree of coupling of the framework and the difficulties to replace closed-source libraries by open-source ones.

In our evaluation, we implemented kernel redundancy manually. Part of our future work is creating an automatic framework to generate diverse redundant kernels.

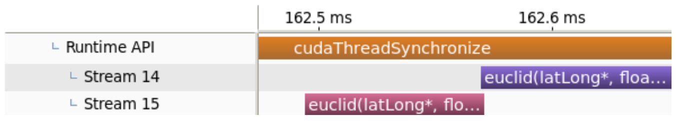
B. Slack Measurements Results

We measured the slack time of redundant kernel executions and analyzed its relationship with the offloading preparation of the CUDA Runtime calls, discussed in Section IV-D, namely the configuration call, the arguments setup and the CUDALaunch. We execute 100 times an application from Rodinia Benchmark Suite (myocyte) modified to use redundant kernel execution and collect measurements using the NVIDIA Profiler.

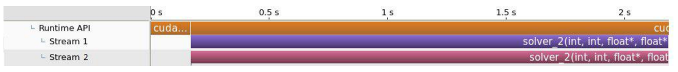
[1]The EEMBC (Embedded Microprocessor Benchmark Consortium) released just some months ago ADASMark [13], an ADAS Benchmark suite that would be highly relevant for our study. While we have performed all the necessary actions since its release to get access to it, access has not been yet granted, so we could not evaluate it. However, some information about the pipeline is publicly available and some Rodinia benchmarks have similarities with ADASMark (e.g. those for image processing and pattern recognition).



(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
Stream 1
Stream 2
cuda...
solver_2(int, int, float*, float*
solver_2(int, int, float*, float

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

Fig. 6: Timelines of redundant executions extracted using the NVIDIA Visual Profiler.

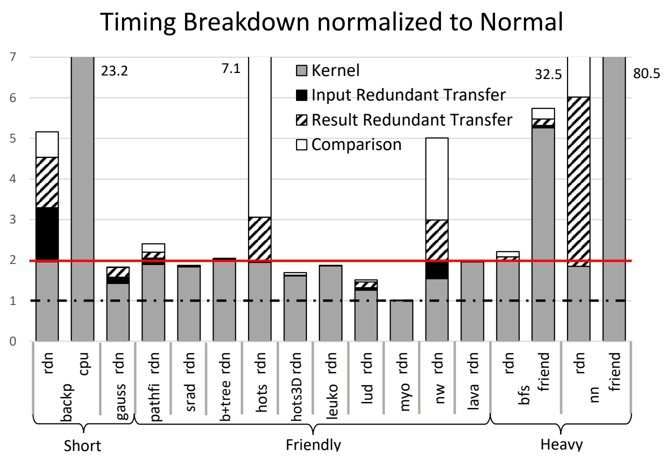


Fig. 7: Redundant (rdn) execution results for the different classes of kernels.

We observed that the slack time highly correlates with the accumulated cost of the offloading preparation elements. For this reason, there is always a minimum slack time of a few microseconds, which has been measured in our case to 9μs, and could reach up to 18. NVIDIA acknowledged this effect as launch performance or kernel latency [15].

C. *Applying Staggered Kernel Execution*

In order to assess the effectiveness of GPUs to meet *Req1*, we have selected one representative case for each type of behavior explained: one short, one heavy and one friendly kernel. We can observe the concurrency of the three applications in the snapshots of the timelines in Figure 6 extracted from the NVIDIA Visual Profiler, which is the kind of tool an industrial user would have at hand. In particular, the timeline of the execution of each redundant kernel is shown in the two bottom bars of each plot, one in purple and another in light red. Note that time scales change across plots, being around 2.2μs for the short kernel, 0.1 ms for the heavy one and 2,200ms for the friendly one. As shown, each copy of the short kernel, which should be better executed in the CPU, runs for less than 5μs. Hence, although its execution may take much longer on a CPU (typically around 10x-100x), such time is still very low in absolute terms (lower than typical response times of physical actuators) and, additionally, using the CPU would avoid the overhead caused by GPU configuration and data transfer.

This effect is better shown with the detailed results presented in Figure 7, grouped per kernel classes, where “1” corresponds to the normalized execution time of the original benchmark without redundancy.

Short Kernels: In the leftmost part, we have the backprop benchmark, which belongs to the short kernel class. We observe that the redundant kernel execution takes longer than twice of the non-redundant version, since there is no overlapping between the replicas. Note also that, due to the small duration of the kernel, the relative overhead of the GPU redundancy is very large due to redundant transfers and comparison. However, when we execute the CPU version of the kernel on the CPU, it takes 7 times longer, which is affordable for this type of tiny kernels. This is the only short kernel that comes along a CPU version of the code. For *gaussian*, execution time does not double w.r.t. normal execution because kernel duration includes kernel launching (CPU) and kernel execution (GPU), and these steps overlap across kernels (first kernel execution with second kernel launch) as shown in Figure 6a.

Friendly Kernels: These kernels overlap significantly. Thus, their redundant kernel execution is smaller than twice the original version. In general, the overheads of the redundant execution are small. High overheads for some kernels relate mostly to large data transfers and result comparison. Hence, while not explored in this work, the cost of those operations can be mitigated by issuing redundant comparisons onto the GPU, so that results do not need to be transferred back for one of the kernels, and comparison is parallelized.

Heavy Kernels: For these kernels, the cost of their redundant kernel execution is above 2x due to the inability to achieve overlapping. As explained before, these kernels can be modified to become friendly. In particular, for bfs and nn we increased the amount of work performed by each thread and changed the kernel grid organization of threads in blocks. This change allows kernels to overlap, thus becoming friendly. However, it comes at the expense of longer execution times. In particular, bfs and nn execution times grow by around 2.5x w.r.t. their heavy redundant version. Note that in the case of nn, its redundant version takes much longer than its non-redundant version (32.5x) due to (1) the doubled data transfer, which moves from fitting in L2 caches to exceeding cache space, and due to the effect of then comparing this large set of data. Part of our future work consists on studying more flexible redundant and friendly kernel generation with lower slowdowns. For instance, part of the overheads may be mitigated by performing result comparison in the GPU, as discussed for friendly kernels.

D. *Result Comparison*

In order to guarantee that kernel executions on the GPU are correct, a comparison must be done between the two results in the lockstep CPU. Such comparison could be parallelized and performed (redundantly) on the GPU. However, our results show that comparison time is, below 1% for most of the kernels, thus not making worth the effort of porting the comparison to the GPU for most of them.

Differently to CPUs, which typically implement the IEEE 754 floating point (FP) standard, GPUs may not fully adhere to specific standards, or may simply schedule work so that FP operations of the redundant kernels occur in different order [16]. This may lead to different rounding choices which, ultimately, cause fault-free results be (slightly) different in practice. Hence, when implementing the result comparison in the CPU, we had to provide some flexibility to tolerate minor deviations.

VI. Related Work

ASIL-D capable processors like the Infineon AURIX [7] and the ST Microelectronics SPC56XL70 [10] deployed in current cars implement DCLS. DCLS may not suffice for some fail-operational ASIL-D systems with tight FTTI [17]. To improve the reaction time in case of an error detection, several works have proposed mechanisms to achieve timely error detection [18] and recovery by means of low-latency checkpointing and roll-back recovery [19]. However, computational power requirements of AD systems greatly exceed the ones of current ASIL-D applications and thus, more powerful – yet safe – computing platforms are needed to realize AD systems [20].

NVIDIA has recently announced the first functionally safe autonomous driving platform [6], which includes support to achieve fail-operational capabilities by allowing complex software algorithms run on the CPU, the CUDA GPU, a deep-learning accelerator and a programmable vision accelerator to enhance redundancy and diversity. According to the announcement, ASIL-D rating is achieved by an NVIDIA DRIVE Xavier GPU and an ASIL-D rated safety microcontroller with appropriate safety logic. However, to the best of our knowledge, ASIL-D compliance for functionalities requiring high performance can only be achieved with diverse software implementations, which ultimately increase drastically design and V&V costs.

Some authors evaluate the use of GPU, FPGA and ASIC designs for AD applications, showing that each design provides a different performance and power tradeoff, so the best hardware platform may change across different AD applications [21]. However, GPUs have already been suited to automotive systems, which provides GPUs with an advantage w.r.t. other hardware platforms [1], [4].

While redundancy is a well known reliability measure to combat random (independent) faults, such as radiation, either by means of time-redundancy [22], [23], space-redundancy [24], [25] or both indistinctly [26], none of those works considers the case of CCFs, which may lead the system to failure despite redundancy. Differently to those works, in this paper we consider specifically CCFs, which are the faults of relevance for the highest ASIL in automotive, and show how CCFs can be avoided – and to what extent – by enforcing diverse redundancy.

VII. Conclusions

In this work we analyze how COTS GPUs can be used to provide diverse redundancy by means of qualitative and quantitative assessments, reaching the following findings:

① GPUs offer the degree of physical redundancy needed to enable some form of loose lockstep execution. In particular, plenty of redundant computation units are in place, and storage and communication means could be protected analogously to those in the microcontroller (e.g. with ECC and CRC).

② Some kernels can be executed redundantly and simultaneously in a staggered manner in a GPU, thus achieving diverse redundancy naturally, whereas others cannot be executed simultaneously due to being either too short or too resource demanding.

③ For those kernels failing to achieve diversity, appropriate software transformations allow them achieve it, either by reducing the amount of resources used simultaneously, or by executing them on the microcontroller if they are too short.

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.

References

[1] TESLA, "Full Self-Driving Hardware on All Cars," https://www.tesla.com/autopilot.

[2] International Standards Organization, *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[3] J. E. et al, "Analysis and RTL Correlation of Instruction Set Simulators for Automotive Microcontroller Robustness Verification," in *DAC*, 2015.

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

[5] 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/

[6] NVIDIA, "NVIDIA Announces World's First Functionally Safe AI Self-Driving Platform," 2018, https://nvidianews.nvidia.com/news/nvidia-announces-worlds-first-functionally-safe-ai-self-driving-platform.

[7] Infineon, "AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations," 2012, http://www.infineon.com/cms/en/about-infineon/press/press-releases/2012/INFATV201205-040.html.

[8] S. Alcaide et al., "DIMP: A low-Cost Diversity Metric based on circuit Path analysis," in *DAC*, 2017.

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

[10] STMicroelectronics, "32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications," 2014.

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

[12] ——, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," *IISWC*, 2010.

[13] EEMBC, "The ADASMark$^{\text{TM}}$ Benchmark," https://www.eembc.org/adasmark/index.php.

[14] "Apollo, an open autonomous driving platform." http://apollo.auto/, 2018.

[15] NVIDIA, "NVIDIA CUDA Toolkit 10.0.130," 2018, https://docs.nvidia.com/pdf/CUDA_Toolkit_Release_Notes.pdf.

[16] N. Whitehead and A. Fit-Florea, "Precision & performance: floating point and ieee 754 compliance for nvidia gpus," *rn (A+ B)*, vol. 21, 01 2011.

[17] 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. 35, no. 3, pp. 1–1, 2018.

[18] C. Hernandez and J. Abella, "Timely Error Detection for Effective Recovery in Light-Lockstep Automotive Systems," *IEEE TCAD*, vol. 34, no. 11, 2015.

[19] ——, "Low-cost checkpointing in automotive safety-relevant systems," in *DATE*, 2015.

[20] ARM, "ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade," 2015, https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php.

[21] S. Lin et al., "The architectural implications of autonomous driving: Constraints and acceleration," in *ASPLOS*, 2018.

[22] A. Mahmoud et al., "Optimizing software-directed instruction replication for gpu error detection," in *SC*, 2018.

[23] M. B. Sullivan et al., "Swapcodes: Error codes for hardware-software cooperative gpu pipeline error detection," in *MICRO*, 2018.

[24] D. A. G. Oliveira et al., "Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison," *IEEE Transactions on Nuclear Science*, vol. 61, no. 6, 2014.

[25] M. Dimitrov et al., "Understanding software approaches for gpgpu reliability," in *GPGPU*, 2009.

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