A Software-Only Approach to Enable Diverse Redundancy on Intel GPUs for Safety-Related Kernels

Nikolaos Andriotis[‡,†], Alejandro Serrano[†], Sergi Alcaide[†], Jaume Abella[†], Francisco J. Cazorla[†]

Yang R. Peng[⋆], Andrea Baldovin[⋆], Michael Paulitsch[⋆], Vladimir Tsymbal[⋆]

[‡] Universitat Politècnica de Catalunya (UPC)
[†] Barcelona Supercomputing Center (BSC)

[⋆] Intel Corporation, Germany

***Abstract*—Autonomous Driving (AD) systems rely on object detection and tracking algorithms that require processing high volumes of data at high frequency. High-performance graphics processing units (GPUs) have been shown to provide the required computing performance. AD also carries functional safety requirements such as *diverse redundancy* for critical software tasks like object detection. This implies that software must be executed redundantly (in a single GPU for efficiency reasons), and with some form of diversity so that a single fault does not cause the same error in both redundant executions. Unfortunately, high-performance GPUs lack explicit hardware means for diverse redundancy and software-based solutions with limited guarantees have only been provided for NVIDIA GPUs. This paper presents a software-only solution to enable diverse redundancy on Intel GPUs achieving, for the first time, strong guarantees on the diversity provided. By smartly tailoring workload *geometry* and managing workload allocation to execution units with thread-level wrappers, we guarantee that redundant threads use physically diverse execution units, hence meeting diverse redundancy requirements with affordable performance overheads.**

***Index Terms*—Cache coherence, multicore real-time systems, contention**

I. Introduction

Safety-related functionalities in autonomous systems (e.g., autonomous cars and single-pilot planes) require increasing levels of computing performance for functionalities like object detection and tracking. Those functionalities generally build on matrix operations such as those in Deep Neural Networks [21], operate large matrices (e.g., images from high-resolution cameras), and have real-time constraints since decisions are taken while vehicles are in operation conditions. Hence, powerful accelerators such as GPUs are often the choice in industry to realize performance-demanding functionalities.

Safety-related functionalities must also meet a number of safety requirements, in line with domain-specific safety standards like ISO26262 for automotive systems [14]. For instance, in the context of Autonomous Driving (AD), functionalities like object detection and tracking inherit the highest integrity level – Automotive Safety Integrity Level (ASIL[1]) D – and therefore, must implement some form of *diverse redundancy* to avoid the so-called *Common Cause Failures* (CCFs), i.e. failures experienced in redundant systems due to a single (shared) fault. For instance, a CCF occurs when two redundant components (e.g., two identical cores) experience a fault affecting both of them simultaneously (e.g., a voltage drop), and such fault leads to identical errors (e.g., if both cores are fully synchronized and their state is identical). In the case of computation, this is generally addressed using Dual Core Lockstep (DCLS) for computing cores so that identical cores execute the same process with some cycles of staggering. Hence their state is never identical and any fault is expected to lead to different errors. In the worst case, the system can detect them and trigger appropriate mitigation. This is, for instance, the solution used by different Infineon AURIX microcontroller families [9].

Safety-related computation requiring GPUs (e.g., camera-based object detection) needs specific software solutions to prevent CCFs to occur since, to the best of our knowledge, commercial off-the-shelf (COTS) GPUs do not implement DCLS. Solutions with redundancy across multiple GPU devices increase costs and reliability concerns severely due to the increased number of physical components and off-chip communication to retrieve and compare results from both GPUs. In fact, in the context of CPUs, these concerns pushed for the adoption of on-chip DCLS as implemented by the aforementioned AURIX microcontroller.

To achieve some form of diverse redundancy support in a single GPU, some works propose hardware changes [3]. Unfortunately, those solutions cannot be applied to COTS GPUs. Other works provide some guarantees for NVIDIA GPUs by running redundant kernels concurrently and staggered by exploiting the way CUDA – the NVIDIA API to manage kernel execution – dispatches kernels onto the GPU [4], [5]. The latter solution leads to diversity while the execution of both redundant kernels overlaps since they use disjoint resources by construction, but no guarantee is given when no overlapping occurs and, by construction, part of the execution does not overlap. In particular, whenever one kernel finishes, the other one could use the same resources used by the former kernel, hence losing diversity. Moreover, the particular solution used in [4], [5] is NVIDIA specific (relies on CUDA) and cannot be exported to Intel GPUs, which are the target of our work.

This paper overcomes the limitations of existing software-only solutions for COTS GPUs, and presents a software-only solution to achieve diverse redundancy on Intel GPUs with strong guarantees on the diversity achieved. Our solution builds on a software layer that overrides the hardware

[1]ASIL ranges from D (highest integrity) to A (lowest integrity). There is an additional level called *Quality Managed* (QM) for components with no safety hazards, and hence no safety requirements.

published in Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23). Association for Computing Machinery, New York, NY, USA,
451–460, http://dx.doi.org/10.1145//3555776.3577610

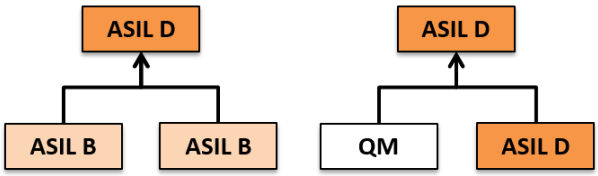

Fig. 1: Usual decomposition patterns for ASIL-D items.

scheduler decisions ***with no hardware changes***, and schedules redundant software threads onto hardware threads in different GPU "regions" (i.e., subslices). In particular, we bring the following contributions:

- A software-only mechanism providing strong guarantees on the diverse redundancy achieved on Intel GPUs by controlling the particular part of the computation carried out by each hardware thread.
- An easy-to-integrate implementation on any kernel by building on prolog and epilog routines to bound the original (unmodified) GPU kernel code.
- Reduced execution time overheads, around 9% on average for several matrix multiplications, w.r.t. the non-diverse redundant execution of GPU kernels.

While our solution has been realized on Intel GPUs building on some available features for those GPUs, nothing precludes the adoption of our solution for other GPU families if they provide analogous features.

The rest of the paper is organized as follows: Section II provides some background on automotive requirements relevant for our work and on Intel GPUs. Section III presents our approach to achieve diverse redundancy on Intel GPUs with strong guarantees. The effectiveness of our approach is evaluated in Section IV. Related work is discussed in Section V. Finally, Section VI presents the main conclusions of this work and discusses future work.

II. Background

A. The Need for Diverse Redundancy

Complying with high-integrity requirements (e.g., ASIL-D in automotive systems) is generally expensive due to the cost needed to reach specific low failure rates. Hence, solutions based on *ASIL decomposition* are often used, where a component is decomposed into multiple ones. Safety requirements become less stringent for individual components based on the specific decomposition pattern, whereas the overall integrity level reached by their composition is the target one.

Random hardware faults (e.g., particle strikes, voltage droops, etc.) cannot be avoided by design and appropriate safety measures must be included in the design to detect and manage those faults. A correct management may require either providing fault tolerance or reaching a safe state within a given time limit upon the detection of an error. The particular safety measure to implement and its particular realization relates to the integrity level (ASIL in automotive) decomposition pattern followed. The most popular ones in the particular context of automotive are illustrated in Figure 1. On the left, we have the case of a fail-operational ASIL-D component where safety cannot be managed separately of the functionality and hence, resulting components in the decomposition also inherit some ASIL. On the right, we have the case of fail-safe systems where a monitor can inherit the ASIL and preserve the overall safety of the component, whereas the functionality is relieved from any safety requirement, hence becoming QM. The relevant pattern in AD (our target) is the one in which an ASIL-D component is decomposed into two ASIL-B redundant components [3], [4]. Each component can have lower integrity requirements as long as they achieve a *sufficient degree of independence* (diversity), meaning that they are not subject to CCFs. This pattern is needed for the GPU in our work because it inherits safety requirements (i.e., it runs ASIL-D AD software components), and no safe state can be achieved since continuous service must be provided in AD systems (they are fail-operational). As explained before, this diverse redundancy is achieved on CPUs implementing some form of on-chip DCLS for computing components (e.g., Infineon AURIX microcontrollers [9]) due to effectiveness and cost reasons related to (i) designing and verifying a single core design, (ii) using the same software on both cores, and (iii) keeping redundancy on-chip to reduce procurement costs and reliability risks due to additional physical components.

In the context of GPUs, DCLS has not been realized yet to the best of our knowledge due to its associated complexity and cost. Hence, one can only resort to DCLS-less COTS GPUs, since deploying two GPU devices brings increased costs and reliability concerns. In fact, solutions preserving diverse redundancy on-chip (i.e. on a single GPU device) are strongly preferred, in line with solutions for CPUs [9]. Recent work has shown that single-GPU diverse redundancy is feasible for NVIDIA GPUs [4], [5]. Those software solutions build on sizing kernels so that they do not use more than half of the resources in the GPU, so they can be run redundantly and simultaneously. Further, redundant kernels naturally are expected to use disjoint computing resources based on the assumption that resources cannot be simultaneously allocated to multiple kernels. Staggering is achieved via the sequential start of the kernels from the CUDA runtime. However, the initial part of the *head* replica (i.e., the one starting execution first) until the *trail* replica (i.e., the one starting execution last) starts, as well as the final part of the trail replica after the head one finishes, execute in isolation and, potentially, could use computing resources later or formerly used by the analogous computation in the replica, hence challenging diversity. While this is not generally expected, it could potentially happen and impair the assumption of disjoint usage of compute resources.

In this work, we focus on Intel GPUs, which are becoming increasingly attractive for the automotive domain [17]. We leverage their advanced observability features, such as the ability to determine the particular hardware thread where each individual software thread runs. Building on that information, we propose a ***software-only*** mechanism that provides diverse redundancy and guarantees that strict diversity is achieved for all computation, hence avoiding the uncertainties of the aforementioned solution for NVIDIA GPUs by construction. Our approach can be generalized to GPUs other than Intel ones if those provide analogous support to the Intel intrinsics that allow any software thread identify the hardware thread where it is running.

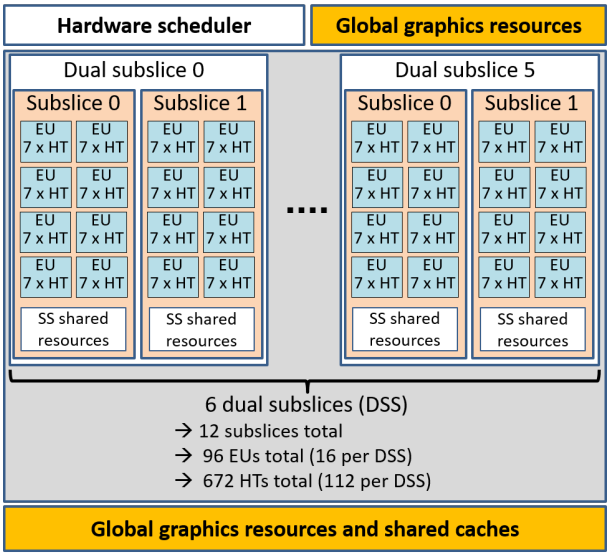

Fig. 2: Schematic of the geometry of the X^e_{LP} GPU used in this work [11].

B. The Intel X^e_{LP} GPU and Support for Diverse Redundancy

GPUs consist of several computing elements capable of performing a large number of regular computations in parallel with high throughput. Conceptually, a GPU architecture organizes the computing elements into groups and subgroups based on whether software threads can be scheduled simultaneously or independently and whether there are shared resources per group or sub-group.

In the case of the Intel X^e_{LP} GPU considered in this work [12] (see Figure 2), computing elements are referred to as Execution Units (EUs). Each EU can execute up to 7 hardware threads (HTs) that share the arithmetic logic units (ALUs) in the EU. EUs are grouped into subslices (SS) so that each SS has 8 EUs (and hence $8x7 = 56$ HTs) sharing some SS-local components including an instruction cache and a thread scheduler. SS are organized into pairs called Dual Subslices (DSS) that include exactly 2 SS each (so $2x8x7 = 112$ HTs). The GPU includes one or several DSS in the Slice, which share some Slice-global shared resources for graphic processing as well as cache memories. The GPU used in this work has 6 DSS, with 2 SS per DSS, 8 EUs per SS, and 7 HTs per EU, as measured empirically with the appropriate GPU intrinsics [13], therefore with 672 HTs in the GPU. Note that the description of the X^e_{LP} architecture in the corresponding technical reference manuals [10], [12], as well as the actual GPU implementation used in this work, include a single Slice with all the aforementioned components. However, as we describe later, our solution works analogously independently of the number of Slices.

Since SS do not share resources other than unique resources at Slice-level granularity, when redundant threads are executed across different SS, they share only non-replicated components in the GPU such as the L3 cache, the shared graphics resources and the hardware scheduler. Regarding shared caches and other shared components used for general-purpose computation, we provide specific considerations in Section III-A. Regarding graphic-specific hardware sub-blocks of the GPU (e.g., pixel-related blocks), they are not used by general-purpose computing AD workloads, so preventing CCFs in those sub-blocks is unnecessary. Finally, the hardware scheduler at the Slice level is likely not replicated, and hence, a potential source of CCFs. Despite that, our approach mitigates a significant fraction of those failures by scheduling redundant software threads to different SS at different times. In any case, if replication is eventually physically added for the hardware scheduler, it will incur negligible hardware costs since most of the GPU area is devoted to EUs, caches, and graphics-specific resources.

Overall, in the scope of this work we address the diverse redundancy of the computing components, which will be achieved if the following two conditions hold:

1) $COND_{space}$. Redundant threads execute on different computing components (e.g., different SS).
2) $COND_{time}$. Redundant threads run with some staggering (i.e., they do not execute exactly at the same time).

III. Diverse Redundancy Approach

A. Rationale

Implementing software-based diverse redundancy requires the kernel to be replicated so that it can be executed twice and the results can be compared upon completion. Each replica of the kernel will spawn the exact same number of software threads that perform identical work.

1) $COND_{space}$: This condition requires that replicated software threads use different computing resources. If we allow them to be mapped to different HTs of a given EU, they could potentially share intra-EU and intra-SS resources, therefore with the risk of experiencing a CCF. Analogous reasoning applies if we map threads to different EUs within the same SS since they will share intra-SS resources. Hence, threads have to be mapped to different SS. Also note that, since SS do not share any resource within a DSS, it is irrelevant whether the SS where redundant threads are executed belong to the same DSS or not.

In our case, for simplicity of the implementation, since there are 2 SS per DSS, we enforce one of the replicated kernels to use SS0 of all DSS, whereas the other kernel is restricted to use SS1 of all DSS. In this way, by managing the SS identifier, i.e. SS0 or SS1, we can make replicated kernels use separate and symmetrical resources, which prevents CCFs while maximizing performance allocating homogeneous resources to (homogeneous) redundant kernels.

Redundant threads of replicated kernels use different (replicated) data. When redundant data is mapped to different sets of shared caches – due to memory alignment – the data is stored in diverse locations across kernels (i.e., a given datum and its replica are stored in cache lines in different cache sets). When both kernels map their data to the same cache set, since kernels are scheduled and run simultaneously, each thread will access a redundant copy of the data that is naturally located in different cache lines of that set, preventing a CCF.

2) $COND_{time}$: Regarding $COND_{time}$, our approach does not exercise explicit control on the time dimension. However, redundant threads use replicated data, so data fetched cannot be shared across redundant threads, which generate

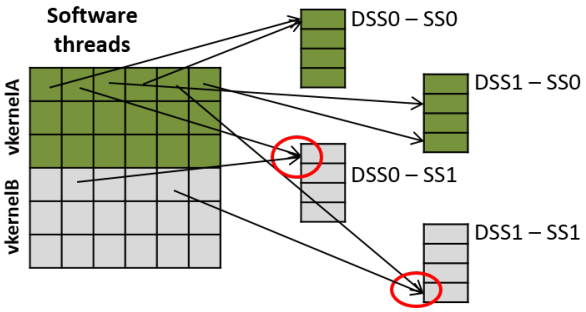

Fig. 3: Example with work split arbitrarily, and mapping fully controlled by the hardware scheduler.

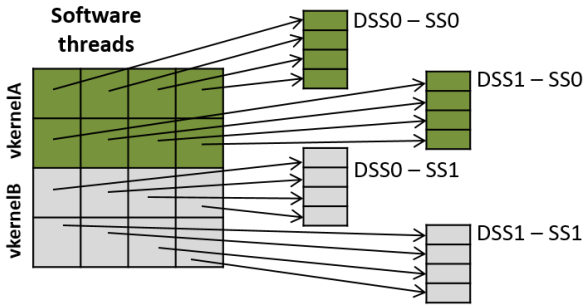

Fig. 4: Example with work split as appropriately, and mapping overridden by our software strategy.

independent data load and store requests, therefore naturally serialized in the access to shared caches or DRAM memory. Hence, while accesses may occur with limited staggering, some staggering exists and, as shown in commercial DCLS processors [9], 2-3 cycles of staggering suffice in general.

In line with previous work [4], [5], not all CCFs can be prevented with software only means, like those related to the use of unique hardware components in the GPU (e.g., thread scheduler, and decode logic of shared caches). Yet, due to the staggering across redundant threads, some diversity exists and it depends on the physical implementation of the GPU whether time diversity is enough to compensate the lack of space diversity in unique components. Also, by using replicated data, hence in different memory locations, addresses accessed by redundant threads differ, which brings an additional source of diversity particularly relevant for components where those addresses are managed (e.g., shared caches).

B. Context

We realize our redundancy concept within a single kernel, which factors out the effects of the serial kernel scheduling of the runtime [6], and additionally simplifies debugging and result analysis. Note, however, that our approach can be fully applied to the case of multiple kernels.

We have generated intra-kernel redundancy duplicating data (and computation) by adding an additional dimension to the data used, so that the index for such dimension can only be '0' or '1'. As we show later, this allows replicating also the work cleanly without further modifications in the original code.

For the sake of commodity, we refer to each of the two intra-kernel replicas as *virtual kernels* or *vkernels* for short since, as explained, we embed two such kernels (vkernelA and vkernelB) into a single kernel to ease result interpretation.

C. Overall Strategy

Ideally, we would like to instruct the hardware scheduler on what SS to allocate to different threads to guarantee that vkernelA runs only in SS with SS id 0 (S_0^{all} for short), whereas vkernelB runs only in S_1^{all}. However, we lack that control and the hardware scheduler has freedom to map a software thread to any HT in any EU of any SS of any DSS. This is illustrated in Figure 3 in which we have both vthreads split into 36 software threads (18 for each vthread). There are 2 DSS, each one with 2 SS, with each SS having 4 HTs. The organization of those HTs into EUs (e.g., 1 EU with 4 HTs, 2 EUs with 2 HTs each, etc.) is irrelevant for this example. As shown, since no control is applied, the software threads of a given vkernel (e.g., vkernelA) can be run on HTs of any SS. In the example, we can see how some software threads of vkernelA run in S_0^{all} and some others in S_1^{all}. The situation for vkernelB is analogous, with some redundant software threads (i.e., the same software thread in both vkernels) running in the same HT (or the same EU), hence using the same computing resources and hence, failing to avoid CCFs. This is illustrated in the figure with the red circles.

In order to exercise the control needed to override the work allocation performed by the hardware scheduler, we assume that the hardware scheduler allocates all HTs in a round-robin manner – if idle – so that, given N HTs, a particular HT_i is allocated again exactly after allocating other $N-1$ HTs assuming that they are all idle prior to allocation. In our test environment this assumption held in all our experiments. In our particular GPU with 672 HTs, this implies that, if the GPU is idle and we intend to run 672 software threads, each HT will be allocated to exactly one software thread.

We exercise control on how to make vkernelA and vkernelB run on S_0^{all} and S_1^{all}, respectively, as follows:

1) We set the number of software threads to match the number of HTs ($|HT|$) (672).
2) We virtually split the work of each vkernel into $\frac{|HT|}{2}$ software threads with the aim of making each vthread use half of the GPU computing resources.
3) Each software thread, upon execution, uses the HT, EU, SS and DSS identifiers to select the piece of work to execute. In particular, if $SS = 0$, work from vkernelA is performed. Else, if $SS = 1$, work from vkernelB is performed.

Hence the overall work is split into as homogeneous as possible execution "chunks", with each execution chunk mapped statically to a specific HT in the GPU[2], and such mapping occurs ensuring that all HTs in S_0^{all} perform together all work of vkernelA, and HTs in S_1^{all} do the same for vkernelB. As shown in Figure 4, we first enforce having as many software threads as HTs (16 in the example). Whenever a HT starts executing a software thread (e.g., the first HT in SS0 of DSS0), it performs the work allocated to that physical HT (e.g., the work in the first row and first column of vkernelA). The

[2]The particular software thread allocated to a given HT will perform the corresponding chunk of work mapped to the particular HT where it runs.

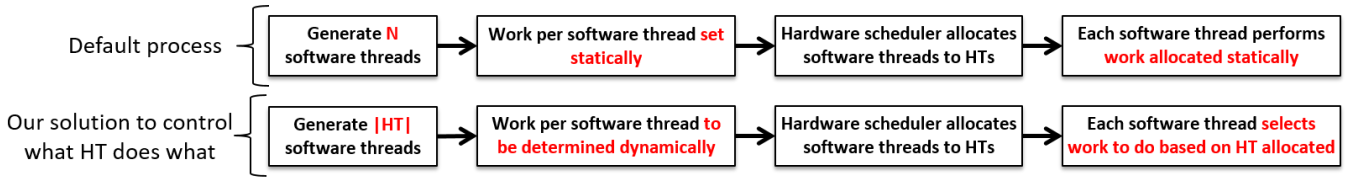

Fig. 5: Summary of the default work split and scheduling process (top), and our process to achieve diverse redundancy (bottom).

particular fraction of work to be carried out is selected using the HT, EU, SS and DSS identifiers. Overall, we achieve a bijective correspondence between software threads and HTs, we make each HT execute its corresponding software thread performing a pre-decided fraction of the work, and moreover we guarantee that software threads of vkernelA only use HTs in S_0^{all}, whereas software threads of vkernelB only use HTs of S_1^{all}.

The overall process is summarized in Figure 5 that shows how, in the default process, work is allocated to software threads statically, and then the hardware scheduler maps software threads – and hence work – to HTs. However, in our case, software threads select the work to carry out dynamically based on the HT where they are run, and hence, we take over decisions on what HT performs what computation to enforce diverse redundancy. The next subsection details how work is split into execution chunks and mapping across HTs is actually realized.

D. Strategy Realization and Integration

We introduce three changes to the original code to implement our strategy:

1) We create an additional dimension to the matrices, as described in Section III-B, to implement the virtual redundant kernels. Note that such modification relates to having redundant execution, not to the particular strategy proposed in this work to enforce diversity.
2) We set the number of software threads to $|HT|$. This is a trivial modification to apply in the CPU code where the kernel is launched.
3) We start each thread selecting the fraction of work to be carried out based on the actual DSS, SS, EU and HT ids of the HT where the software thread is run. Those ids are obtained using appropriate Intel GPU intrinsic commands [13]. We have tailored such process so that it is application independent and is encapsulated in a "prolog" routine call to be added in the user code before the actual execution of the software thread work.

For evaluation purposes, we have extended the prolog with additional functionality to record ids and to initialize appropriate counters, and have added an epilog function to allow retrieving results from those counters. That functionality is not really needed and could be dropped, although its impact in execution time is low in absolute terms, and completely negligible in relative terms for key workloads (e.g., matrix multiplications with $1024x1024$ matrices).

Note that our solution requires no hardware change and it is a purely software-only solution realized on COTS Intel GPUs.

```
matrix_multiplication(a, b, c, size){
  for i in (0,size):
    for j in (0,size):
      for k in (0,size):
        fc[i*size+j] = fc[i*size+j] + fa[i*size+k] * fb[k*size+j];
}
```

Fig. 6: Original matrix multiplication CPU code.

```
__kernel void matrix_mult
(
    const int size,
    const __global float* A,
    const __global float* B,
    __global float* C,
    __global struct HardwareThreadInfo* info
)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    if (i < size && j < size)
    {
        float acc = 0;
        for (unsigned int k = 0; k < size; ++k)
            acc += A[i*size+k] * B[k*size +j];
        C[i*size+j] =  acc;
    }
}
```

Fig. 7: Original matrix multiplication GPU code.

E. An illustrative example

This section details the application of our method to a specific example for illustration purposes. We show, step by step, the specific changes to be applied on the application code, as well as the application independent routines and transformations used to achieve diverse redundancy.

Figure 6 shows the CPU version of the original code of a matrix multiplication kernel. It consists of 3 nested loops where the innermost one computes one cell of the output matrix. The GPU version of this code is shown in Figure 7, where we see that the call `get_global_id(int)` is used to retrieve the indices for the two dimensions of the loop that have been parallelized into software threads. In this way, each software thread computes one cell of the output matrix, and there are as many software threads as cells has the output matrix. For instance, if such matrix has $1024x1024$ dimensions, there will be $1,048,576$ software threads that will be scheduled by the hardware scheduler.

Figure 8 shows the modified GPU code for the software threads. The code of the called routines is omitted due to space constraints. As shown, modifications are trivial to apply:

- *(Only for debug purposes)* We add an additional variable in the declarations, `info`, but only for debugging pur-

```
__kernel void matrix_mult
(
    const int size,
    const __global float* A,
    const __global float* B,
    __global float* C,
    __global struct HardwareThreadInfo* info
)
{
  {
    HARDTYPE(float, A, size*size)
    HARDTYPE(float, B, size*size)
    HARDTYPE(float, C, size*size)
    HEADER(size,size)
      //ORIGINAL CODE
      int i = get_global_id(0);
      int j = get_global_id(1);
      if (i < size && j < size)
      {
        float acc = 0;
        for (unsigned int k = 0; k < size; ++k)
             acc += A[i*size+k] * B[k*size +j];
        C[i*size+j] =  acc;
      }
      //END
    FOOTER(i,j)
  }
}
```

Fig. 8: Modified matrix multiplication GPU code.

poses. This declaration would be dropped for a production version of our solution.

- *(Mandatory)* We use the `HARDTYPE` function, which is in charge of selecting the part of the data to operate based on the specific SS where the software thread is allocated (obtained with the intrinsic call `intel_get_subslice_id()`). In particular, as explained before, each matrix is duplicated by adding an additional (first) dimension with 2 positions. The `HARDTYPE` routine sets the pointer of the matrix to the beginning of the matrix if the SS id is 0, or shifts it by the size of the original matrix (hence to the beginning of the second copy of the original matrix, as if the first dimension was set to 1) if the SS id is 1. Therefore, we call `HARDTYPE` for each of the matrices operated passing as parameters the data type, the pointer (name) of the matrix, and its size as the product of the size of its dimensions.
- *(Mandatory)* We call the `HEADER` function, which computes the actual part of the work to carry out based on the actual DSS, SS, EU and HT ids of the HT where the software thread has been allocated, and creates the wrapping loops to make the software thread execute its code as many times as needed for the corresponding output cells.
- *(Mandatory)* We finally call the `FOOTER` function, all whose statements intend to store debug information back and would be dropped for a production version of our solution. The only lines of code truly mandatory are the braces closing the loops.

Note that the only part of the `HEADER` and `FOOTER` calls that is application dependent is the number of parameters and their values, since we need as many parameters as dimensions of the matrices on which to iterate, and those parameters must be the dimension sizes. Different `HEADER` and `FOOTER` functions must be used if the number of dimensions on which to iterate differs (e.g., 1, 3, 4, etc. instead of 2), but their code is analogous to the one for two dimensions.

IV. Evaluation

This section presents the evaluation framework (platform, benchmarks and setups evaluated), detailed analysis of the results for matrix multiplication benchmarks, and summarized results for all the other benchmarks.

A. Evaluation Framework and Setups

Platform. We used an 11th Gen Intel(R) Core(TM) i7-1165G7 CPU at 2.80GHz with an Intel(R) Iris(R) Xe Graphics [0x9a49] GPU. Since it is deployed on a desktop with Linux, some services related to the display use the GPU periodically and cannot be disabled. Hence, some experiments are altered due to this since the scheduling assumption (i.e., round-robin allocation of HTs) is broken upon the interference of any other process in the GPU. Whenever this happens, our logs reflect that at least one HT has been allocate more than one software thread whereas at least one HT has been allocated none, and hence, results are discarded. Overall, we repeat experiments until achieving 6 runs per setup and matrix size without Linux interference, and report results using execution time averages.
Benchmarks. We build on kernels used in neural networks such as those implemented in autonomous driving frameworks, hence being deployed on GPUs and having strict functional safety requirements imposing the use of diverse redundancy. Note that, to have full control of the code executed, we do not use fully optimized APIs and, instead, use simple implementations of the benchmarks. The list is as follows:

- Matrix multiplication (MxM)[3]: we consider different square matrices of NxN rows and columns with N=672, 1024 and 1344 respectively. Sizes 672 and 1344 allow splitting work uniformly across HTs in the GPU (there are exactly 672), and hence, remove load imbalance effects. A different size of 1024 has also been used to account for those effects.
- Rectified Linear Unit activation function (RELU): Traverses a matrix of NxN setting each negative value to 0, and keeping non-negative values unmodified. The same dimensions sizes as for MxM are considered for consistency.
- Local Response Normalization (LRN): performed over a single matrix. The same dimensions sizes as for MxM are considered for consistency.
- Matrix (or 2D) convolution (Mconv): it performs the convolution of a 2D matrix computing each element of the 2D result matrix as a function of a 3x3 region of the input 2D matrix. The same dimensions sizes as for MxM are considered.
- Vector (or 1D) convolution (Vconv): it is applied on a 1D input and computes each element of the 1D result vector as a function of a region of the input 1D vector.

[3]Note that we use "MxM" to refer to the benchmark and "*N*" or "*NxN*" to refer to the size of the dimensions of the matrices.

Dimensions used for the vector are N^2 so that its size matches that of the data for MxM (despite being 1D instead of 2D).

- Matrix multiplication transposed (MxMtrans): this benchmark is analogous to MxM, but instead of accessing one input matrix by rows and the other by columns, both are accessed by rows to maximize spatial locality when fetching input data.
- Nearest Neighbor (NN) is a non-parametric supervised learning method used for classification and regression. In our case, it is used to find the closest neighbor (based on the Euclidean distance) in a data set. As for the other benchmarks we use as input the matrix sizes used by MxM.
- Stencil 3D (Stencil) is a numerical data processing solution where an element of a matrix is updated as a function of some of its neighbors (including itself). In our case, we compute it as a function of the immediate neighbor elements in the three dimensions, as well as itself, and use the same overall data size as for MxM.

Setups. We consider 6 different scenarios:

- *Original*: the original kernel is run on the GPU matching each software thread to the computation of one cell of the result matrix.
- *Originalx2*: two (*Original*) (virtual) kernels are embedded into the kernel. Software threads are analogous to those of *original*.
- *HalfFree*: the *Original* kernel is split into 336 software threads (as many as half of the HTs in the GPU). Those software threads are let to run wherever the hardware scheduler spawns them.
- *HalfConst*: analogous to *HalfFree*, but software threads are controlled by our software scheme to run only in HTs whose SS id is 1. Therefore, we constraint the kernel to use a specific half of the computing resources of the GPU.
- *RedunFree*: the *Originalx2* kernel is split into 672 software threads (as many as HTs in the GPU). Those software threads are let to run wherever the hardware scheduler spawns them.
- *RedunConst*: this one is our proposed solution. It is analogous to *RedunFree*, but software threads are controlled by our software scheme to run one of the virtual kernels only in HTs whose SS id is 0, and the other virtual kernel in HTs whose SS id is 1. Therefore, we have diverse redundancy.

Note that those setups allow us comparing *Original* against *HalfFree* and *HalfConst* expecting a $\approx 2x$ slowdown due to using half of the computing resources, if computing resources are the performance bottleneck. *HalfFree* is expected to get its execution time doubled due to using half of the HTs, and then an additional impact in performance (either positive or negative) due to the change in terms of software threads imposed to control the amount of HTs used. Then, *HalfConst* is expected to bring some additional performance loss (likely low) w.r.t. *HalfFree* due to enforcing the use of specific HTs for the computation instead of using the first HTs allocated by the hardware scheduler.

Analogously, those setups allow us to compare *Originalx2* against *RedunFree* and *RedunConst* expecting no relevant slowdown. *RedunFree* will experience some performance impact (either positive or negative) w.r.t. *Originalx2* due to constraining its number of software threads. Then, *RedunConst* is expected to bring some additional performance loss (likely low) w.r.t. *RedunFree* due to enforcing the use of specific HTs for the computation instead of using the HTs as allocated by the hardware scheduler.

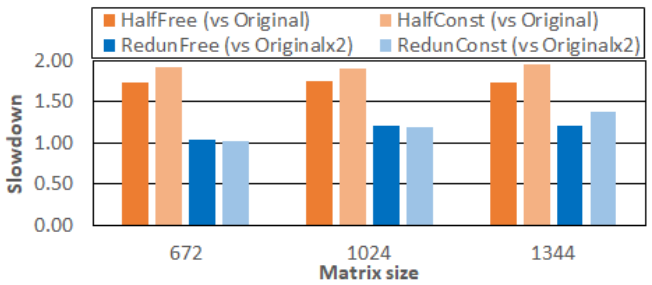

Fig. 9: Slowdowns for the MxM for all *Half* and *Redun* setups and matrix sizes considered w.r.t. *Original* and *Originalx2* setups respectively.

Finally, we can compare each of the pairs *Original* vs *Originalx2*, *HalfFree* vs *RedunFree*, and *HalfConst* vs *RedunConst* to understand the impact of doubling the workload. Note that such impact is caused by using redundancy, but has nothing to do with our mechanism itself. Such overhead relates to the increased pressure on the computing resources, shared caches, and memory bandwidth.

B. MxM Results

We first provide a detailed evaluation of the MxM, and then we analyze a broader set of benchmarks discussing only those effects differing from the MxM case. We do so because MxM already exposes most of the relevant scenarios.

Figure 9 shows the slowdown of the 4 non-original setups w.r.t. the corresponding original setup in each case, as explained before and indicated in the figure.

Figure 9 shows that the slowdown of *HalfFree* is around 1.75x across all matrix sizes, hence below the expected 2x. This occurs because, in the *Original* setup, there is some degree of bandwidth saturation to access L3 cache or main memory. Hence, despite all HTs are allowed to be used in the *Original* setup, their real utilization is below 100%, and therefore, when reducing HT utilization down to 50% for *HalfFree* (only half of the HTs are used), execution time does not double because the real HT utilization does not halve (e.g., moving from an 87.5% utilization to 50% could cause a 1.75 execution time increase).

Regarding *HalfConst*, we note that its slowdown is typically around 10% higher than that of *HalfFree*. This is the cost of constraining what HTs to use.

Comparing *RedunFree* and *RedunConst* to *Originalx2* we see that slowdowns are generally around 1x, as expected. While some performance variations are observed, they generally relate to workload imbalance, which becomes more visible for larger matrices. Such variations make slowdowns increase for larger matrices. Note that, in some cases, *RedunFree*

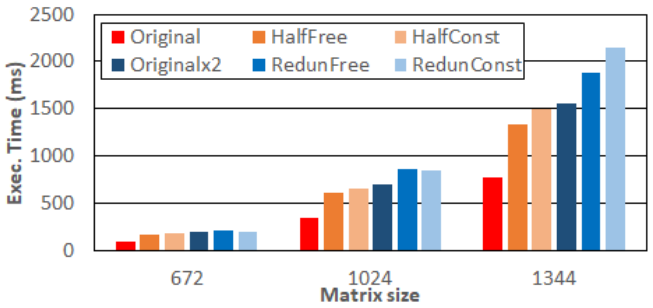

Fig. 10: Execution times (in millions of cycles) for the MxM for all setups and matrix sizes considered.

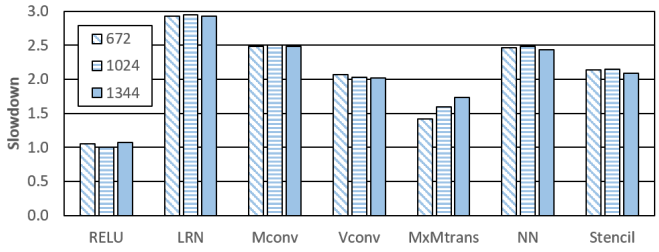

Fig. 11: Slowdown of *RedunConst* w.r.t. *Original*.

slowdown may be slightly higher than *RedunConst* one. This relates to unfortunate performance imbalance since software threads execution time is lower for *RedunFree* on average, but higher for its maximum.

Figure 10 includes the absolute execution times for completeness to ease the analysis of the data by the reader, and also validate that *Originalx2* slowdown w.r.t. *Original*, which should be around 2x due to performing twice the same amount of work, is quite close to that ratio in practice across matrix sizes (between 2.03x and 2.06x).

C. Comparison with NVIDIA-specific Solution

Note that, while the solution proposed to achieve diversity on NVIDIA GPUs cannot be directly applied on Intel ones, we can approximate what its expected performance would be. As discussed before, NVIDIA specific solutions build on the idea of decreasing the computing resources needed of the kernel under analysis to match (or not exceed) half of the computing resources available in the GPU [4]. Then, replicated kernels are launched simultaneously with the minimum inter-kernel launch delay so that they run simultaneously but staggered. However, no explicit control is exercised on the specific computing resources that the threads from each kernel use, which are determined by the hardware scheduler. Hence, the NVIDIA solution performance can be approximated with *Originalx2* and *RedunFree* by not constraining how software threads are mapped to HTs, and letting the hardware scheduler controlling such mapping. Each of those two configurations corresponds to different number of software threads, but preserving the idea behind the NVIDIA solution: each replica uses around half of the computing resources. As shown, performance for our solution, *RedunConst*, is comparable to the one that would be obtained with the NVIDIA solution, but providing stronger diversity guarantees.

As explained before, our solution provides stronger guarantees than that existing for NVIDIA GPUs. Our solution can be applied to other GPU families that provide analogous support to that of Intel ones so as to allow any software thread identify the particular hardware thread where it is running.

D. Other Benchmarks Results

Figure 11 shows *RedunConst* w.r.t. *Original* for all the other benchmarks for three different sizes of the problems. As explained before, the expected slowdown should be generally a bit above 2x due to the following reasons: (A) the 2x amount of computation performed; (B) extra contention arising in the access to shared resources that were not saturated with the original load; and (C) the work imbalance brought by the static allocation of work to HTs performed by *RedunConst*.

Vconv, and Stencil. We note that the expected behavior is observed for Vconv and Stencil being such slowdown quite stable across the three problem sizes considered, namely 672, 1000 and 1344. In particular, Vconv and Stencil exhibit the 2x slowdown expected due to (A) above, with negligible impact due to (B) and (C) (between 1% and 8%).

LRN, Mconv, and NN. These three benchmarks also experience a 2x slowdown due to (A). However, the additional slowdown due to (C) is significant for the three of them, and the slowdown due to (B) is also significant for LRN. Overall, slowdowns for LRN, Mconv, and NN are around 2.93x, 2.49x and 2.46x, being highly stable across matrix sizes.

The remaining benchmarks (RELU and MxMtrans) show, instead, lower slowdowns that we analyze case by case.

RELU. In the case of RELU, included due to its relevance in the context of neural networks, the amount of computation performed is tiny. Hence, by generating as many software threads as computed elements for *Original*, most of the execution time corresponds to overheads to create, schedule and terminate software threads. Since *RedunConst* creates only one software thread per HT, such overhead decreases drastically and performance gains offset by far the cost of doubling the computation. In this particular case, fewer and coarser software threads for *Original* should be used to increase efficiency. Nevertheless, we included this particular work split of RELU to illustrate a larger variety of scenarios.

MxMtrans. MxMtrans triggers specific data access patterns that lead to improved performance for *RedunConst* w.r.t. *Original*, which mitigates partially the 2x slowdown caused due to (A). In particular, each cell of the result matrix of MxMtrans is obtained by traversing one row of each one of the input matrices, whose footprint is much smaller than that of MxM. Hence, MxMtrans exploits spatial locality for both input matrices, and such data requires limited cache space. In the case of *Original*, since software threads are scheduled to HTs without cache locality in mind, no relevant reuse occurs across software threads sharing SS. However, while not on purpose, *RedunConst* often schedules software threads reusing each others' data in the same SS. Hence, this increases cache reuse w.r.t. *Original*, and leads to a slowdown clearly below 2x. Note also that, as the matrix sizes increase, the slowdown approaches 2x since the volume of data per SS is higher and

cache capacity limits data reuse across software threads.

In fact, the fine-grain control that our solution provides on what fraction of the work is performed by each HT could be exploited to favor cache locality. Hence, those applications performing some data reuse could be the target of performance optimizations by tuning what part of the work is performed by each HT. Exploiting such opportunities in a general manner is left for future work.

V. Related Work

The effectiveness of GPUs, FPGAs and ASIC designs in the context of autonomous driving applications has been analyzed in [18]. Other works focus exclusively on GPUs and evaluate their real-time performance capabilities for safety-relevant applications [24], [7].

Automotive platforms such as the NVIDIA Xavier [19] and RENESAS R-Car H3 [1] build upon COTS GPUs governed by automotive microcontrollers. ASIL-D compliance for fail-operational applications on those platforms requires the use of diverse software implementations of the algorithms deployed on the GPU, which leads to duplicated design and V&V costs, or fully redundant SoCs, which imposes high procurement costs and reliability concerns due to having additional inter-connected physical components.

A number of works, mostly focusing on NVIDIA GPUs, aim at characterizing or improving performance characteristics relevant for our work, yet they neither provide redundancy per se nor diversity since their target is not safety-critical systems. However, some of those solutions could be used to achieve redundancy on NVIDIA GPUs, yet without full diversity. In particular, some works explore scheduling approaches and program transformations to partition some GPU resources on NVIDIA GPUs [2], [16], [23], [15] or manage resource concurrency [20].

Some authors provide redundancy schemes on GPUs [8], [22], yet without providing diversity, as needed for safety-critical systems. So far only Alcaide et al. have developed solutions to achieve diverse redundancy on (NVIDIA) GPUs, either with hardware support [3] or with software-only solutions [4], [5]. As discussed before, software-only solutions are the only choice that can be used on COTS GPUs, but do not provide strong diversity guarantees during those periods when only one of the replicas is running since such replica could use the same resources used by the other replica, hence losing diversity. Moreover, they are NVIDIA specific. Overall, this paper is the first work providing diverse redundancy on Intel GPUs, as required for ASIL-D automotive applications, with strong guarantees even in those periods where only one of the kernels is still running by explicitly controlling what computing resources are used by each kernel.

VI. Conclusions and Future Work

COTS GPUs are becoming increasingly popular in automotive systems to implement AD functionalities. However, they do not provide explicit support for diverse redundancy, as needed for ASIL-D applications. So far, solutions for NVIDIA COTS GPUs have been proposed, yet with some caveats related to the limited diversity guarantees achieved when only one of the redundant kernels is running.

This paper proposes a new software-only solution to achieve diverse redundancy on Intel GPUs, hence enabling their use for ASIL-D automotive applications, by explicitly controlling the computing resources used by each computation overriding the hardware scheduler, yet with a software-only solution, hence also eliminating the caveats existing for previous work for NVIDIA GPUs. Our results show that performance costs are low (e.g., around 9% for the ubiquitous matrix multiplication) Moreover, our solution is easy to integrate on legacy software by virtue of its modular design, which only requires inserting specific calls while keeping original code unaltered.

VII. Acknowledgments

This work has been partially supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GB-C21/AEI/ 10.13039/501100011033, and by the project AUTOtech.agil of the German Federal Ministry of Education and Research (support code 01IS22088I).

References

[1] RENESAS R-Car H3. https://www.renesas.com/us/en/products/automotive-products/automotive-system-chips-socs/r-car-h3-m3-starter-kit, 2021.

[2] J. Adriaens et al. The case for GPGPU spatial multitasking. In *HPCA*, 2012.

[3] S. Alcaide et al. High-Integrity GPU Designs for Critical Real-Time Automotive Systems. In *DATE*, 2019.

[4] S. Alcaide et al. Software-only Diverse Redundancy on GPUs for Autonomous Driving Platforms. In *IOLTS*, 2019.

[5] S. Alcaide et al. Software-Only Triple Diverse Redundancy on GPUs for Autonomous Driving Platforms. In *DSN-Supplemental Volume*, 2020.

[6] S. Alcaide et al. Achieving Diverse Redundancy for GPU Kernels. *IEEE Transactions on Emerging Topics in Computing*, 10(2):618–634, 2022.

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

[8] M. Dimitrov. Understanding Software Approaches for GPGPU Reliability. In *GPGPU Workshop*, 2009.

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

[10] Intel Corporation. Intel Processor Graphics Gen11 Architecture. Version 1.0, 2019. https://www.intel.com/content/dam/develop/external/us/en/documents/the-architecture-of-intel-processor-graphics-gen11-r1new.pdf.

[11] Intel Corporation. Architecture Day, 2020. https://newsroom.intel.com/wp-content/uploads/sites/11/2020/08/Intel-Architecture-Day-2020-Presentation-Slides.pdf.

[12] Intel Corporation. Intel Iris Xe MAX Graphics Open Source. Programmer's Reference Manual. For the 2020 Discrete GPU formerly named "DG1". Volume 4: Configurations. Version 1.0, 2021. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-dg1-vol04-configurations.pdf.

[13] Intel Corporation. OpenCL(TM) Built-In Intrinsics, 2021. https://github.com/intel/pti-gpu/blob/master/chapters/binary_instrumentation/OpenCLBuiltIn.md.

[14] International Standards Organization. *ISO/DIS 26262. Road Vehicles – Functional Safety. Second edition*, 2018.

[15] S. Jain et al. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *RTAS*, 2019.

[16] J. Janzen et al. Partitioning GPUs for Improved Scalability. In *SBAC-PAD*, 2016.

[17] Chiyoung Lee, Se-Won Kim, and Chuck Yoo. Vadi: Gpu virtualization for an automotive platform. *IEEE Transactions on Industrial Informatics*, 12(1):277–290, 2016.

[18] Shih-Chieh Lin et al. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *ASPLOS*, 2018.

[19] 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.

[20] M. Thazhuthaveetil S. Pai and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *ASPLOS*, 2013.
[21] H. Tabani et al. A Cross-Layer Review of Deep Learning Frameworks to Ease Their Optimization and Reuse. In *ISORC*, 2020.
[22] J. Wadden et al. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *ISCA*, 2014.
[23] B. Wu et al. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *ICS*, 2015.
[24] M. Yang et al. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *ECRTS*, 2018.