UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

INCREASED RELIABILITY ON INTEL GPUS VIA SOFTWARE DIVERSE REDUNDANCY

NIKOLAOS ANDRIOTIS

Thesis supervisor: ALEJANDRO SERRANO-CASES (Barcelona Supercomputing Center)

Thesis co-supervisor: JAIME ABELLA FERRER

Tutor: LEONIDAS KOSMIDIS (Department of Computer Architecture)

Degree: Master Degree in Innovation and Research in Informatics (High Performance Computing)

Thesis report

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

26/06/2023

Acknowledgements

In the first place, I would like to thank my advisors, Alejandro Serrano Cases, Jaume Abella and Francisco Cazorla for their guidance and mentoring throughout the development of this Thesis.

I also want to thank Leonidas Kosmidis for introducing me to the CAOS team at BSC. Without him, I would not have discovered the amazing realm of GPU programming.

Moreover, I would like to acknowledge the BSC institution for financially supporting my Master studies, and also to the following institutions that have partially supported this work: 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).

Last but not least, I would like to thank my beautiful mother, my anxious father and my curious sister for their unconditional support in my life and studies.

To my beloved friends in Greece, this is not about me, it is about us!

Abstract

During the past decade, the industry revolutionized its processes by including Artificial Intelligence. Nowadays, this revolutionary process extends from the manufacturing industry to more critical sectors, such as the avionics, automotive, or health industry, where errors are unacceptable. One clear example of this process is the automotive industry, where the installation of Advanced Driver Assistance Systems (ADAS) is now a reality, and the aim is to achieve fully self-driving cars (SDCs) in the near future. This new emerging domain has been increasing the interest of researchers in ADAS and Autonomous Driving (AD) systems, as these domains require processing high volumes of data using complex algorithms (Deep Learning (DL)) at high frequency to meet highly tight time constraints (Real Time (RT)).

In this context, traditional computing quickly became a bottleneck due to CPUs being unable to handle such amount of data and process it on time. In contrast, high-performance graphics processing units (GPUs) have recently provided the required computing performance and partially fulfilled the timing constraints. Thus, electronic manufacturers continuously innovate to improve their devices' performance by introducing state of the art GPUs that are equipped with new accelerators as well as enhancing their GPUs in terms of performance and efficiency (i.e., performance per Watt). For instance, Nvidia introduced in 2017 Jetson AGX Xavier SoC, a GPU-based low power device designed mainly for accelerating machine learning applications, and focused on the automotive sector. However, AD or ADAS challenges are not only related to the performance or the timing constraints; another constraint to satisfy is safety. Critical systems, such as AD or ADAS, have to provide the correct outcome on their computation as people's life depends on them. In this sense, the AD sector has an additional constraint: functional safety. Functional safety problems have been long studied, and the only way to address them is through redundancy to identify or correct the erroneous outcome. Additionally, to ensure the highest safety levels, these systems introduce diversity to avoid redundant computation getting compromised at the same point and the errors going undetected (common cause faults (CCF)).

To ensure that the high-performance hardware used for AD is working as expected and that specific safety goals are met, specific hardware support is included to realize safety measures, and exhaustive verification and validation (V&V) processes are carried out. These verification processes are incredibly costly, especially when custom hardware is used, and the design and fabrication of such hardware is also an onerous task. As a result, the automotive industry tries to avoid these non-recurring costs by targeting widespread and cheap hardware, i.e. commercial off-the-shelf products (COTS). However, COTS devices present a drawback, manufacturers are reluctant to provide redundant hardware to end users due to the high costs, power consumption, and low-performance ratio. In addition, they jealously guard, in most cases, the implementation details, which limits the adoption of the industry that requires reliable computation. Therefore, the hardware limits the redundancy by design and thus extends the functional safety requirements beyond the boundaries of the hardware layers to the entire software stacks on such devices.

In this sense, researchers have to deal with the limitations of COTS solutions and build more affordable and promising software-based solutions, especially to realize diverse redundancy so that, even if a single fault leads to error all replicas, by being diverse, errors are also diverse and can be detected using comparison.

Thus, software-only diverse redundancy solutions have to be deployed on top of COTS solutions and deal with two main limitations: 1) computation needs to occur redundantly to enable error detection, and 2) redundancy must be guaranteed to occur with diversity to guarantee that, even if an error affects all replicas (e.g., affecting the clock or power networks), errors differ and can be detected, hence avoiding the so-called Common Cause Failures (CCFs). For instance, COTS GPUs lack of explicit hardware devoted for diverse redundancy; thus, software-based solutions are being developed, but most of the current implementations provide limited guarantees and have only been focused on NVIDIA brand.

In contrast, this thesis presents a software-only solution to enable diverse redundancy on Intel GPUs, achieving strong guarantees on the diversity provided for the first time. One key characteristic of this solution is that it is built on top of OpenCL, a hardware-agnostic programming language. This programming language allows it to be expanded using some special functions that the compiler handles, the so-called intrinsics. These functions are implementation-dependent and highly optimized, meaning integrators should provide them. For instance, the intrinsics used in this thesis allow identifying the hardware thread of the GPU where any given software executes, which allows performing smart tailoring of the workload geometry and allocation to specific computing elements inside the GPU. As a result, redundant threads are guaranteed to use physically diverse execution units, hence meeting diverse redundancy requirements with affordable performance overheads. The technique is based on the fact that it issues as many software threads as available HW threads in the GPU, then allocates half of them for executing one kernel and the other half for executing the redundant one. To reach the final diverse and redundant solution, several scenarios are developed to efficiently measure the impact of each step of our modifications to a normal OpenCL kernel execution. At first, only half of our available GPU resources are allocated, allowing one kernel to run wherever the scheduler decides. Then the scheduler is overridden and forced to use half of the resources, forcing only one independent part of the GPU to be used (in this way, the overhead for having a HW-thread aware work allocation is evaluated). Subsequently, duplicating the work (to mimic the two kernel execution) is applied, and lastly, both kernels are forced to be executed in independent parts of the GPU.

Contents

Chapter 1

Introduction

This Chapter focuses on the motivation behind the development and implementation of Software Diverse Redundancy, a proposed solution for addressing certain challenges. Furthermore, the contributions of this work are outlined, providing a clear understanding of the value and impact of the proposed solution. The organization of the subsequent Chapters is also briefly summarized, giving readers an overview of what to expect in the following Chapters.

Autonomous driving (AD) has emerged in the past few years as a promising technology [31]. But, even if it is not a new topic, it is still of great interest [38] to researchers and the industry, as new technologies emerge around it. It has the potential to revolutionize the way we move around, making transportation safer, more efficient, and more environmentally friendly. However, building a safe and reliable autonomous driving system is a challenging task that requires a lot of computational power and sophisticated algorithms. For instance, a key component of an AD system is object detection and tracking algorithms. These algorithms rely on processing high volumes of data at high frequency and require a lot of computing power.

Unlike traditional hardware-based solutions (e.g., general-purpose computing cores), which are a bottleneck due to their inability to handle enormous data volumes and deliver results on time, high-performance graphics processing units (GPUs) have been shown to provide the required workforce for these tasks. GPUs have numerous processing cores, meaning that they can perform many identical computations in parallel on different data at the same time due to their matricial computing nature. This behavior makes them ideal for tasks that require a lot of parallelism, such as massive image-processing tasks. In this context, GPUs quickly became of great interest for AI research and were recognized as an opportunity by the industry. A clear case in the industry of this adoption is the case of Nvidia, which spotted the market opportunity earlier than the competitors and brought the Nvidia Jetson series of embedded computing boards in 2014 [37]. This strategy led NVIDIA to immediately move the company's focus onto the automotive sector, being the first company to introduce a computing platform for use in cars. Now, the automotive industry is introducing this hardware into the car to develop a technology to improve and assist drivers with driving and parking operations. This technology, the so-called Advanced Driver Assistance Systems (ADAS), improves vehicle and road safety by providing a suitable and secure human-machine interaction by using sensors and cameras to react appropriately, allowing different levels of autonomous driving [36].

However, even when the hardware (HW) and software (SW) can outperform the computing requirements

effectively, AD is within a critical sector (automotive) in which people's lives depend on the decision taken by a computer. Due to this criticality, this sector has been bounded by another requirement - Functional safety, specified in ISO26262 [19] in the case of the automotive domain. As a result, the software used in the system, such as object detection and tracking, must meet the most stringent safety requirements since it controls functionalities such as steering, braking and accelerating. According to ISO26262, those functionalities have the highest integrity level, i.e., Automotive Safety Integrity Level (ASIL) D, and their implementation requires the use of safety measures such as diverse redundancy. Redundancy is usually addressed by means of multiple copies of the same software running in parallel, and diversity making their execution differ from each other in some way. The idea behind redundancy and diversity is to ensure that if one copy of the software fails, there is another copy to compare with and detect the error. In fact, these systems shall prevent the so-called Common Cause Failures (CCFs), i.e., failures experienced in redundant systems due to a single (shared) fault (e.g., a voltage drop) leading to identical errors.

Unfortunately, high-performance GPUs lack explicit hardware devoted for diverse redundancy, which makes it challenging to meet the safety requirements for autonomous driving. In contrast, state-of-the-art software-based solutions [4, 5] built on top of this unreliable hardware do not provide diversity guarantees needed to achieve the highest integrity and safety levels, and also present some drawbacks as they only focus on specific vendor implementation.

1.1 Contributions

This thesis shows how some of these problems can be overcome or addressed by providing:

1. A **software-only** mechanism **with strong guarantees** on the diverse redundancy.
2. **Fine-control** on the computation **hardware thread scheduling** on systems without such support.
3. An **easy-to-integrate** implementation by building on **prolog and epilog routines** to bound the original (unmodified) GPU kernel code.
4. **Reduced execution time overheads**, for usual kernels used in AD and ADAS systems.

Note that, while the proposed solution has been realized on Intel GPUs, building on some software available features for those GPUs, nothing precludes the adoption of our solution for other GPU families if they provide analogous features. In fact, our preliminary analysis of other GPUs from other GPU vendors (e.g., NVIDIA) shows that similar features exist in those devices, and hence, our solution could be ported to those other devices.

Based on the work done in this Thesis, one paper has been published:

- **A Software-Only Approach to Enable Diverse Redundancy on Intel GPUs for Safety-Related Kernels** [7] Nikolaos Andriotis, Alejandro Serrano-Cases, Sergi Alcaide, Jaume Abella, Francisco J. Cazorla, Yang Peng, Andrea Baldovin, Michael Paulitsch, Vladimir Tsymbal. 2023. In Proceedings of the 38th Annual ACM Symposium on Applied Computing (SAC'23), Tallinn, Estonia, March 27 - March 31, 2023

1.2 Thesis Organization

The rest of this Thesis is organized as follows:

- Chapter 2 presents the background, which provides a context to the topic this thesis focuses on, and some references to related works in the topic field.
- Chapter 3 introduces the evaluation framework, encompassing the programming model and Intel GPU architecture.
- Chapter 4 delves into the proposed technique, providing a comprehensive explanation. To characterize and validate the solution, an illustrative example is employed, offering practical insights.
- Chapter 5 presents the experimental setup, encompassing the platform utilized and the benchmarks employed. Additionally, it outlines the various setups along with their corresponding code differences, which will be compared and utilized to gather the results for analysis.
- Chapter 6 provides an in-depth presentation of the results obtained from a representative benchmark, specifically matrix multiplication. It further includes the results from the remaining benchmarks and conducts a comparative analysis with previous NVIDIA solutions.
- Chapter 7 concludes the thesis by highlighting the primary findings discussed throughout the document. Additionally, it identifies promising avenues for future research, outlining areas of interest and potential directions for further investigation.

Chapter 2

Background

In this Chapter, the main concepts and keywords necessary to understand the proposed solution for achieving diverse software redundancy on GPUs are presented. Firstly, a definition of Artificial Intelligence (AI) is introduced, along with its importance in Safety-Critical Systems. Following that, details about ISO26262, which serves as the guideline for establishing the safety development process in the automotive industry are provided. Finally, the concepts of redundancy and diversity - the primary objectives of this thesis - are explained.

2.1 AI in Safety-Critical Systems

Nowadays, the term of **Artificial Intelligence (AI)** still is difficult to define. Even after more than two decades have passed, the VDE-AR-E 2842-61 standard, a general framework for the development of trustworthy solutions and trustworthy autonomous / cognitive systems, states that "there is no generally accepted definition of artificial intelligence" [34]. But it seems that there is a consensus around AI evolution from simple Neural Networks and Diffuse logic to more complex algorithms such as general model Deep-Neural-Networks [33], Recurrent-Neural-Networks [23], Spiking-Neural-Networks [12]; or more specific ones as GPT [27]. However, due to the variety of algorithms and models used in AI, "there is not even a consensus around what AI is" as Feldt's et al. work states [11]. Despite not reaching a broad consensus, some AI terminology is described in ISO/IEC 22989 as "a set of methods or automated entities that together build, optimize, and apply a model so that the system can, for a given set of predefined tasks, compute predictions, recommendations, or decisions" [20]. With this definition in mind, automotive manufacturers are investing in, and introducing AI technology to improve and assist drivers with driving and parking operations. This technology, the so-called Advanced Driver Assistance Systems (ADAS) and Autonomous Driving (AD) systems, shall improve vehicle and road safety and some examples include object detection and tracking for the system to make critical decisions. Thus, a number of systems in the automotive industry are considered **safety-critical systems (SCS)**, meaning that a system failure or malfunction could cause human or environmental harm, as well as loss or serious damage to equipment and property [35]. These systems require protection to guarantee some level of safety at all times and to lower the likelihood of failure occurrence. Since ADAS and AD make important decisions (i.e. control braking and steering), the software used inherits safety requirements, and hence needs to adhere to ISO26262.

2.2 ISO26262

The "Road vehicles - Functional safety" standard **ISO 26262** [19] provides specific regulation that involves automobiles, which presents a strict system development process as people's lives depend on the correct operation of those systems. The standard is an adaptation of the broader IEC-61508 [18] safety standard which dictates the use of electrical and electronic equipment in safety critical environments. Other safety-related standards, such as those for railway (e.g., EN50126/8 [9]), are also derived from IEC-61508, as can be seen in Figure 2.1. Some safety critical domains, such as avionics (e.g., DO178B/C [30]), have independent standards which, however, have significant commonalities with IEC-61508 despite being independent.

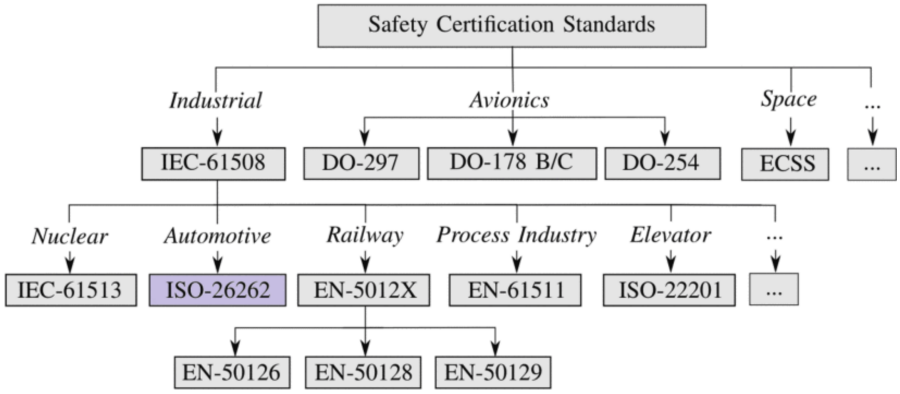


Figure 2.1: Safety standards in different application domains, taken from [1].

ISO26262 describes a development process that aims to mitigate potential risks by outlining the proper steps and methods to ensure the system design is correct by construction in accordance with its safety requirements. Additionally, it provides guidelines for thorough testing of the system's behavior. ISO26262 states that the functional safety hazards associated with safety-critical automotive capabilities are categorized into several Automotive Safety Integrity Levels (ASIL). ASILs are determined by the exposure, severity and controllability upon a failure of hazardous events. As long as an item inherits some safety requirements, its ASIL ranges between A to D, being D the highest level and A the lowest. If the item does not inherit any safety requirement, then it is regarded as Quality Managed (QM) as seen in Figure 2.2. Additionally, according to the aforementioned standard, all safety-related items (those with any ASIL) must go through a design, and V&V process in order to gather sufficient proof that they adequately meet their safety requirements.

Figure 2.2: ASIL Levels chart.

Depending on whether a failure or malfunction can be managed upon detection, we categorize SCS's as

fail-safe or **fail-operational**. The former, as long as it informs us, is considered safe even if it stops operating, but the latter needs to continue its operation regardless of a failure. This relates to the fact that fail-safe systems have a safe state where, despite being unavailable, the system is safe. For instance, proximity alarms provide assistance to the driver so that it can brake or change lane with increased safety. Hence, those systems inherit safety requirements since a failure to notify the presence of a vehicle could lead to an accident. However, in the event of a malfunction in any such system, it can be disabled and the driver will be notified. The driver can then take over the functions of the proximity alarms by personally assessing whether other vehicles could pose a challenge to any driving decision. In this case, the safe state consists of transferring full control to the driver. Safety requirements are inherited by the subsystem monitoring whether the proximity alarm system works well, and notifying the driver upon a failure. On the other hand fail-operational systems are those lacking a safe state, such as braking and steering functions in an AD car, which may even lack a steering wheel, and where, for instance, braking immediately is not safe (e.g., in the highest speed lane of the highway while driving at 120km/h).

Trying to reach specific low failure rates while complying with high-integrity requirements (e.g., ASIL-D in automotive systems) is generally expensive due to costs that come along (i.e., V&V costs). Hence, solutions based on **ASIL decomposition** are often used, where a component is decomposed into multiple ones following specific rules (e.g., redundant components producing different errors upon a fault to guarantee detection by comparison, or separating function implementation from safety monitoring across components). 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. The more relevant patterns in the context of automotive are illustrated in Figure 2.3. 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. In particular, two redundant components implement the same functionality as long as their potential failures are sufficiently independent. 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 Quality Managed (QM).

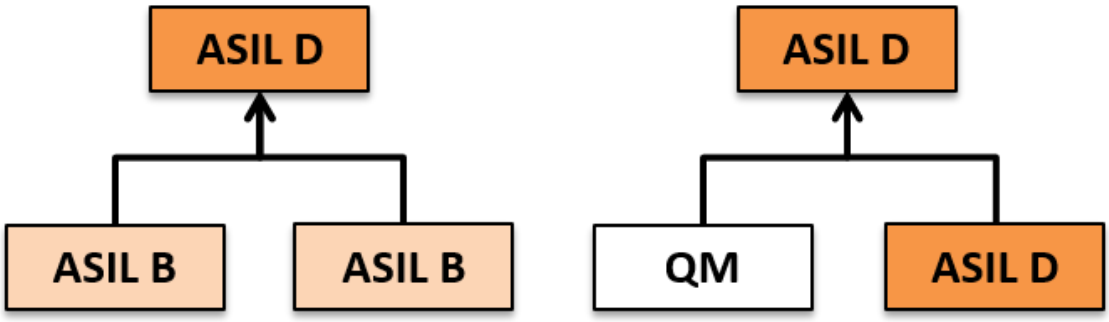


Figure 2.3: Usual decomposition patterns for ASIL-D items.

It is important to understand that since ADAS purpose is to assist the driver, the car can still operate even upon a failure, as long as it notifies the driver so he/she can take over. Hence, ADAS are generally considered fail-safe and one would normally assign the AI part of these systems as QM, with a regular non-AI system monitoring it. The monitor has to be ASIL-D because it must detect failures and notify the driver in a timely manner. On the other hand, when AI is used to achieve fully autonomous cars, there

may not even be a steering wheel, so there is no safe state, and hence the system must still drive even for stopping the car safely. Therefore, the AI part for object detection, trajectory prediction, etc. must remain operational despite faults, and hence it should be considered fail-operational. This is why the most relevant pattern of this work, 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*. In other words, upon a fault that could affect both at the same time (e.g., voltage droop, clock signal error), their behavior differs since, otherwise, redundancy would be useless.

2.3 Redundancy and Diversity

Some of the common requirements in safety standards are redundancy and diversity. **Redundancy** is defined in ISO26262 as *existence of means in addition to the means that would be sufficient for an element to perform a required function or to represent information.* Therefore, it can be seen as a synonym of replication (i.e. the two ASIL B components in Figure 2.3 running the same AI computation), at least to some degree. Redundancy then, is used in a system to ensure that results are correct by, for instance, performing the functionality multiple times and comparing them. Thus, it provides the system with more reliability which is to ensure that the system is performing correctly and inside the deadline expected. Redundancy is used in ISO26262 to decrease the risk of a failure upon a random hardware fault, which is a mandatory requirement for those systems targeting the highest integrity levels (ASIL-C/D). Redundancy can be achieved using different techniques. E.g. for storage we can use Error Correcting Codes (ECC) [10], which is particularly used on memory, or RAID [8]; for connections we can either use ECC or Cyclic Redundancy check (CRC) [28] and for computation, lockstep [13] is normally used. Lockstep execution, consists of having two or more components (e.g., processors) that perform the same work independently [21].

However, it is important to note that for the highest integrity levels, ISO26262 demands to have not only redundancy, but independent redundancy in order to avoid **Common Cause Failures** (CCF), which are failures experienced in redundant systems as a result of a single (shared) fault. This independent redundancy is commonly referred to as **diversity**. Although this property is difficult to quantify [2,24–26], achieving it gives the platform protection against a specific type of faults such as faults that can affect all the redundant parts of the platform (like crosstalk or a voltage droop). In this sense, diversity can be achieved by using two different hardware or software implementations of the same functionality. However, this approach virtually doubles design, and V&V costs. Moreover, using either heterogeneous software or hardware may degrade overall performance since the redundant execution of the task does not finish until the slowest replica finishes.

Thus, one of the common approaches to achieve diversity is to use staggered execution, where one of the replicas is ahead of the other in time, but both execute identical software on identical hardware. Then, if at some point a CCF appears, it will affect differently the two instances since they are performing different work at that instant. Therefore, the effect of the fault will be different in both copies and at the comparison the resulting error can be detected since the two redundant copies will have two different results. In this context, some works propose hardware changes [3], which unfortunately, cannot be applied to COTS products. However on COTS CPUs two of the most widely used software techniques are Triple Modular Redundancy (TMR) and Duplication with Comparison and Re-Execution (DWC-R), applied either temporal [29] or spatial [32]. On the other side, some research on NVIDIA GPUs provide some guarantees 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 this work.

Chapter 3

Evaluation Framework

In this Chapter, the main SW and HW technologies are introduced. Firstly, an overview of OpenCL, a hardware-agnostic programming language, is provided, summarizing its key features and functionalities. Additionally, an overview of an Intel GPU is presented, offering insights into its architecture and capabilities. These technologies play a crucial role in the assessment and evaluation process of the proposed solution.

3.1 OpenCL

OpenCL is an open industry standard created by Apple Inc. in 2008, and lately maintained by the Khronos Group [22], which is composed of significant CPU, GPU, and software manufacturers (including Apple, IBM, Nvidia, AMD, and Samsung) that now oversees its management. The main target of this standard is to create programs that run on a variety of heterogeneous computing devices, including CPUs, GPUs, and other processors. On September 30th, 2020, the most recent OpenCL specification (OpenCL v3.0 Finalized) was made public. In this thesis we utilized the OpenCL 2.0 specification's API. This is because the latest modifications to the standard are not important to achieve diverse redundancy.

The OpenCL framework provides a runtime system, libraries, and programming language that is an extension of the common C language (based on C99) that enables developers to create general-purpose, portable software, meaning that they can **write code once and execute it anywhere**. The OpenCL code is just-in-time compiled for the specific architecture during runtime, therefore the programmer does not need to worry about the target architecture as long as it supports OpenCL. This is not a widespread characteristic of other GPU programming languages as they have a software stack that is hardware-dependent, i.e. CUDA. Additionally, it offers a wide range of programming APIs that developers can use to query and identify the actual device capabilities and write effective code. Low-level hardware abstractions that are simple to use are also provided.

In order for OpenCL to achieve this portability across different architectures, three parts are defined, which are referred to as **models**. They consist of the platform, execution and memory model. The **platform model** describes how OpenCL understands the compute resources in a system to be topologically connected. The **execution model** captures how a program is initiated to operate in parallel in the different compute

resources available and finally, the memory hierarchy of the compute device is described by the **memory model**.

3.1.1 Platform Model

A platform is defined by the OpenCL specification as a **host** that is connected to numerous OpenCL **devices**, i.e., multi-core CPUs, GPUs, and additional processors like DSPs. Each of them is composed by several smaller components, the so-called **compute units (CU)**. For instance, individual cores in a multi-core CPU are just one example of the several computation units that make up a single device. However, compute units can be further decomposed into several different **processing elements (PE)**. The interaction between each of these components is shown in the Fig 3.1:

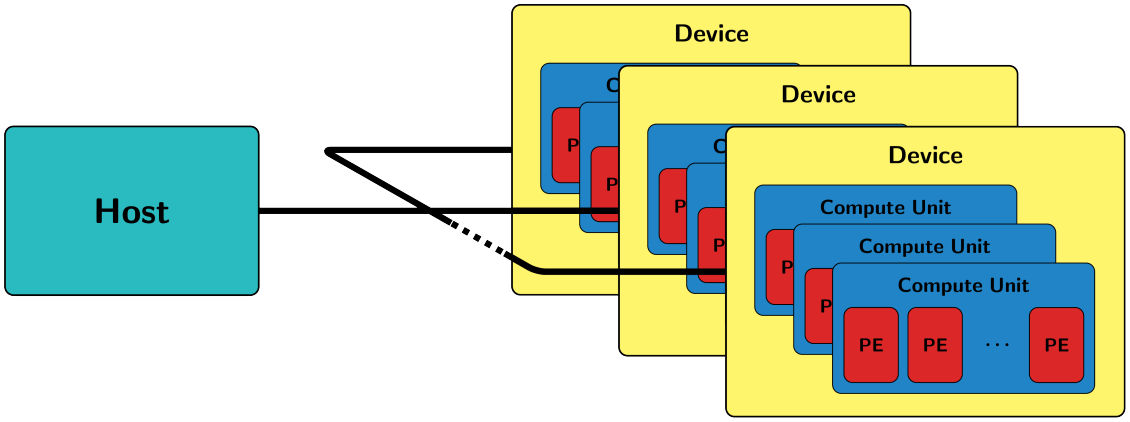


Figure 3.1: OpenCL platform model

Regarding the software side in the platform model, the application running on the **host**, which typically is a CPU, sends **commands** to the OpenCL device. For instance, some of the commands that are sent include kernel execution, reading from and writing to memory objects. The issued commands are scheduled onto the device after being queued up in a data structure known as the **command queue** which will be executed on a specific device. Commands sent to a command-queue are stored in-order but can be scheduled in-order or out-of-order.

3.1.2 Execution Model

The execution model captures how the OpenCL code will run on a device, the so-called **kernel**, and since all internal and external communication occurs through memory, kernel return types are always void. It is also important to note that everything that is required by the kernel to run, is managed by the host application, i.e. copying memory objects, setting kernel parameters, etc. An OpenCL **program** is then constructed with one or more kernels, as well as auxiliary functions required by them. To create a program the language used is called OpenCL-C, and features add-ons, such as memory space specifiers and extra keywords for designating a function as a kernel function. Programs are then compiled by the runtime's OpenCL compiler into executable binaries or stored for later loading.

A single kernel program will be initiated to operate in parallel across an N-dimensional data structure,

known as an **NDRange** using OpenCL's `clEnqueueNDRangeKernel` command. The parameters used to define the NDRange can be understood as a N-dimensional grid, for example the different dimensions in a two-dimensional image. In that case, each pixel computation will be assigned to a **work-item**, which will execute a copy of the kernel on a single processing element. All of the work-items execute the identical kernel code simultaneously, however the algorithm can change the exact path that is taken. A device with N compute units can therefore only perform N work items at once.

Several work-items are assigned to be executed on a single compute unit, and this collection of work items is what we refer to as a **work group**. The **global work size** and the **local work size** are two parameters related to work-groups that can be provided when a kernel is queued for execution. The former one describes the entire number of kernel instances or work items that will be launched for computation, whereas the latter describes the number of work items assigned to a single work group. Therefore, the number of work groups will always be equal to the global work size divided by the local work size. The OpenCL implementation will choose how to divide the global work items into the proper work groups if the local work size is not given. In case there are more work-groups than the available number of compute units, the work-groups will be scheduled one by one on the compute units. A compute unit will always execute work-items from one work-group concurrently before moving on to work-items from another work-group.

To be able to identify a work-item within a work-group, access the necessary memory address, and make the appropriate control decisions, **IDs** are given to the programmer. The initial global and local IDs in OpenCL 2.0 are always (0, 0, 0). Each work item has a distinct global ID that ranges from 0 to the global work group size minus one and represents a point in the index space. Workgroups are also given distinctive IDs in a similar manner. For a better understanding Figure 3.2 shows a graphical illustration of the NDRange.

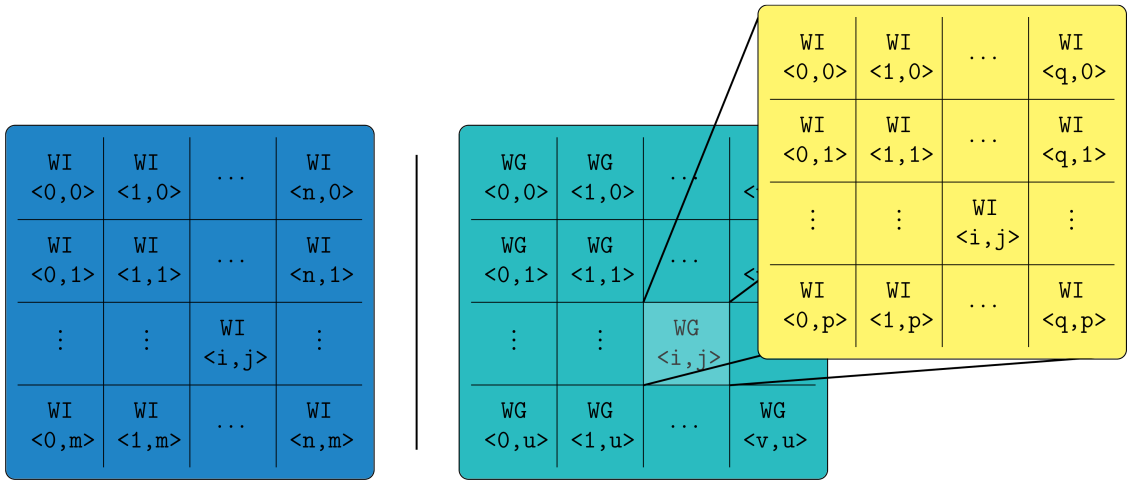


Figure 3.2: The hierarchical model used for creating an NDRange of work-items, grouped into work-groups.

With all the above information, parallel execution of the kernel can be initiated and executed. The steps involved in kernel execution are as follows: 1) The device setup phase, were the platform is initialized and the available devices are listed to create different command queues for each device. 2) Buffer setup phase, were the creation of buffers in both the host and device happens and data is copied from the host memory to the device memory. 3) Kernel initialization phase, in which the kernel source code is loaded and the program object is created and built. 4) Execution phase, where the kernel arguments are set from the host side for the OpenCL kernel to execute and finally 5) the output and freeing phase were data is copied from

the device to the host and all allocated objects are deleted. All these steps are summarized in Figure 3.3.

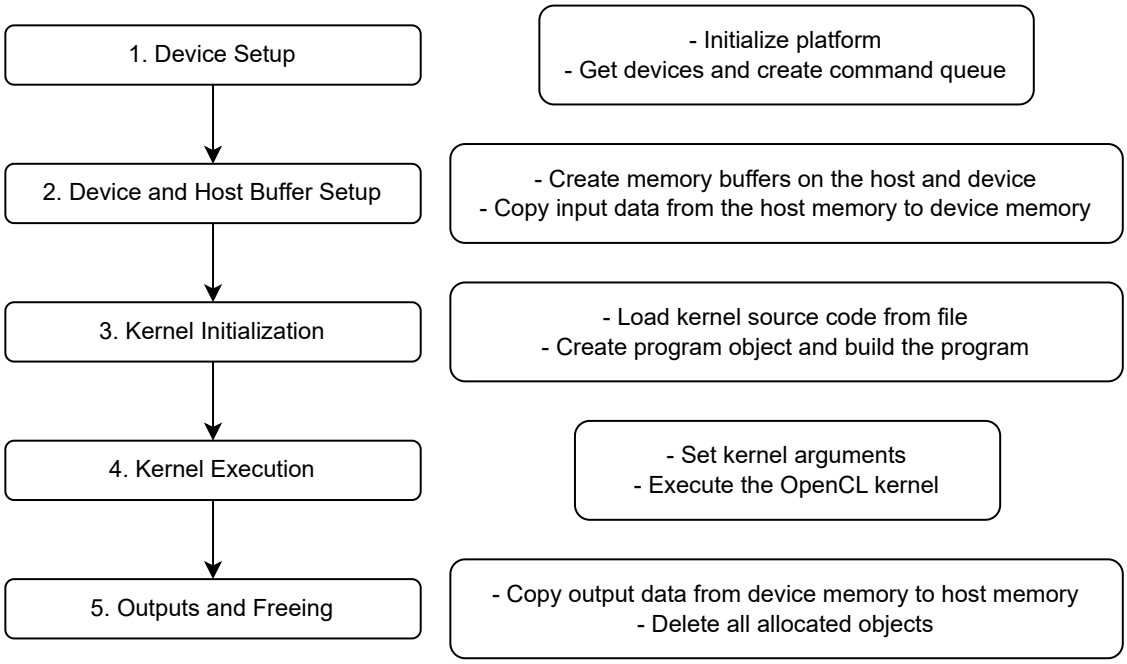


Figure 3.3: The steps involved in an OpenCL kernel execution.

3.1.3 Memory Model

The memory hierarchy for the compute device is defined within the memory model. OpenCL defines a four-level memory hierarchy consisting of global, constant, local and private memory. A visual representation of these levels and their scope is shown in Figure 3.4.

- *Global Memory*: All work-items have read-write access to this memory region (seen with dark orange color). Usually the input data for the work-items are written to this region by the host, and the computed output data is written there by the work-items.
- *Constant Memory*: This is a read-only global memory accessible to all work items (seen with a lighter orange color). The host part of the application allocates and initializes this memory region.
- *Local Memory*: This memory region is the local memory for a work-group (in cyan coloring). All the work-items in a work-group share this memory region. This memory allows work-items to communicate with each other within a work-group.
- *Private Memory*: This memory region (red) represents the local variables of the kernel instance. Each work-item has its own copy of the local variables and they are only visible to the work-item.

It is important to notice that not every device needs to implement each level of this hierarchy in hardware. In addition, consistency between the various levels in the hierarchy is relaxed, and only enforced by explicit synchronization constructs, notably barriers.

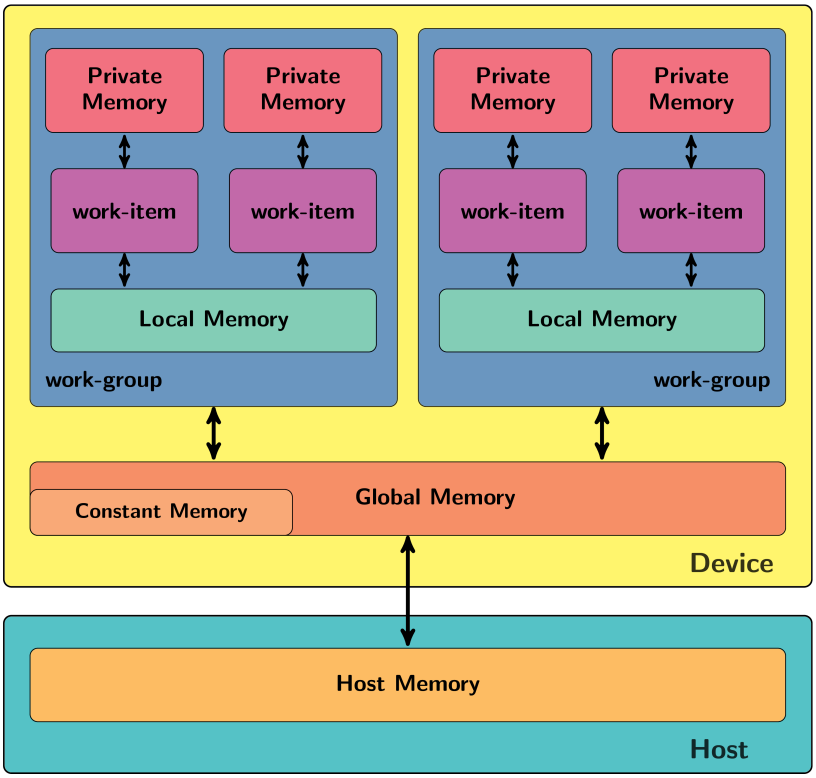


Figure 3.4: Memory regions and their scope in the OpenCL memory model.

3.2 Overview of Intel GPUs

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 sub-groups based on whether software threads can be scheduled simultaneously or independently and whether there are shared resources per group or sub-group.

A detailed block diagram provided by Intel in [15] can be seen in Figure 3.5. The foundational building block of Intel's GPU architecture is the **Execution Unit**, commonly abbreviated as EU. The architecture of an EU is a combination of simultaneous multi-threading (SMT) and fine-grained interleaved multi-threading (IMT). These EUs are compute processors that drive multiple issue, single instruction, multiple data arithmetic logic units (SIMD ALUs) pipelined across multiple threads, for high-throughput floating-point and integer compute. Each EU can execute a number of **Hardware Threads** (HTs) that share the ALUs in the EU. Arrays of EUs are then instantiated into a group called a **Subslice** (SS). Each subslice contains its own local thread dispatcher unit and its own supporting instruction caches. Each Subslice also includes a 3D texture sampler unit, a Media Sampler Unit and a dataport unit. SS are further organized into pairs called **Dual Subslices** (DSS) that include exactly 2 SS each.

Figure 3.5: Intel GPU detailed block diagram

Chapter 4

Specification and design of the solution (Methodology)

This Chapter presents the proposed solution's design to achieve diverse software redundancy on GPUs. First, the Rationale and the Context of this work is introduced. Then more details regarding the strategy that has been followed and the differences w.r.t the default process are presented. Finally, a step by step explanation of all the modifications and functions needed for a kernel to become diverse and redundant are shown, as well as 2 benchmarks for validating the chosen strategy.

As discussed in Chapter 3, Intel GPUs are composed of Subslices (`SS`), which do not share resources other than unique at slice-level granularity. That means that when redundant threads are executed across different `SS`, they will share only non-replicated components in the GPU, such as the L3 cache, the shared graphics resources, and the hardware scheduler. 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. Also, the hardware scheduler at the Slice level is likely not replicated, and hence, a potential source of CCFs. Despite that, by construction the proposed 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.

Executing redundant threads in different SS's, requires the kernel to be replicated so that it can be executed twice and the results can be compared upon completion. Each kernel replica spawns the exact same number of software threads that perform identical work. For shared caches and other shared components used for general-purpose computation, 2 conditions are introduced - $COND_{space}$ and $COND_{time}$ - that will need to hold in order to achieve diverse redundancy of the computing components.

4.1 Rationale

$\boldsymbol{COND_{space}}$ requires that replicated software (kernel) threads use different hardware computing resources. If they are free 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 mapping threads to different EUs within the same SS is allowed, 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.

With the GPU architecture used, since there are 2 SS per DSS, enforcing one of the replicated kernels to use SS0 of all DSS, whereas the other kernel is restricted to use SS1 of all DSS is enough for this condition to hold. In this way, by managing the SS identifier, i.e. SS0 or SS1, 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.

Regarding $\boldsymbol{COND_{time}}$, explicit control on the time dimension is not exercised. However, redundant threads use replicated data, so data fetched cannot be shared across redundant threads, which generate independent data load and store requests, which therefore are 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 [14], 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).

4.2 Context

The redundancy concept is going to be realized 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 this approach can be fully applied to the case of multiple kernels by applying it individually to each kernel and making them be fully serialized (if they are not already fully serialized).

Intra-kernel redundancy duplicating data (and computation) is generated by adding an additional dimension to the data used, so that the index for such dimension can only be ‘0’ or ‘1’. As shown later, this also allows

replicating the work cleanly without further modifications in the original code.

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

4.3 Strategy

The ideal scenario is to instruct the hardware scheduler on the particular SS to allocate the different threads in order 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, the hardware scheduler has freedom to map a software thread to any HT in any EU of any SS of any DSS and means of controlling that do not exist by default. This is illustrated in Figure 4.1 where both vkernels split into 36 software threads (18 for each vkernel). In the scope of this example, lets assume 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, it is shown 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.

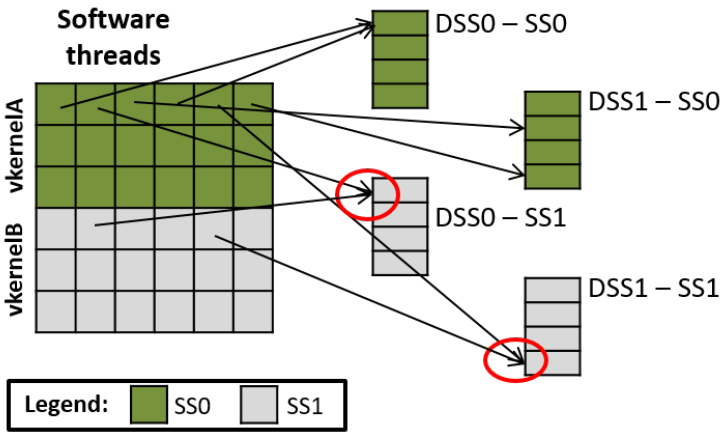


Figure 4.1: Example with work split arbitrarily, and mapping fully controlled by the hardware scheduler.

In order to exercise the control needed to override the work allocation performed by the hardware scheduler, an assumption was made that the hardware scheduler allocates all HTs in a (strict) 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 the test environment used, this assumption held in all the experiments for an idle state of the GPU, which is enforced prior to dispatching the kernels so that the assumption on the round-robin allocation of HTs is not violated. In the GPU, each HT will be allocated to exactly one software thread.

Control on how to make vkernelA and vkernelB run on S_0^{all} and S_1^{all}, respectively, is exercised as follows:

1. Set the number of software threads to match the number of HTs ($|HT|$).
2. Virtually split the work of each vkernel into $\frac{|HT|}{2}$ software threads with the aim of making each vkernel 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[1], 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 4.2, the first thing to do is 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 particular fraction of work to be carried out is selected using the HT, EU, SS and DSS identifiers. Overall, a bijective correspondence between software threads and HTs is achieved, making each HT execute its corresponding software thread performing a pre-decided fraction of the work, and moreover it is guaranteed that software threads of vkernelA only use HTs in S_0^{all}, whereas software threads of vkernelB only use HTs of S_1^{all}.

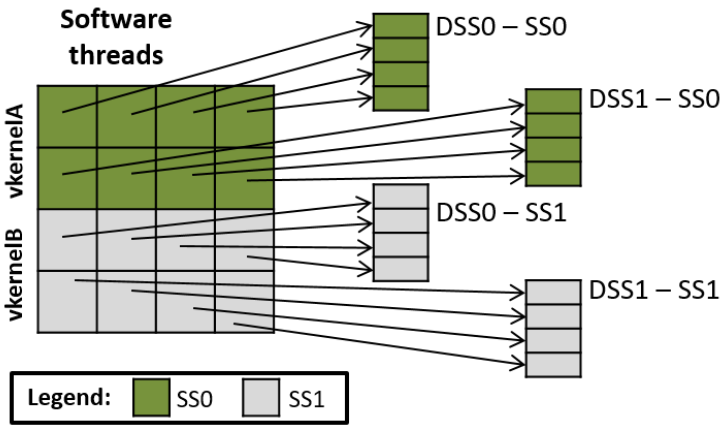


Figure 4.2: Example with work split as appropriately, and mapping overridden by our software strategy.

The overall process is summarized in Figure 4.3 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 the case of the proposed work, software threads select the work to carry out dynamically based on the HT where they are run, and hence, decisions on what HT performs what computation to enforce diverse redundancy are taken.

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

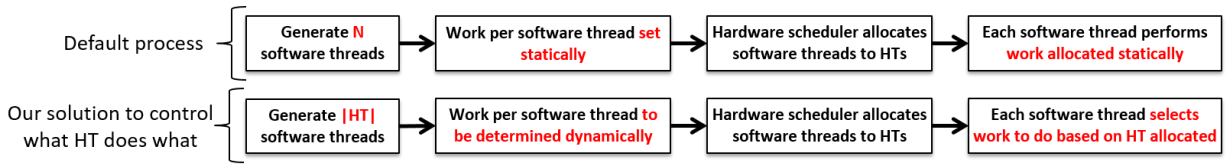


Figure 4.3: Summary of the default work split and scheduling process (top), and the proposed process to achieve diverse redundancy (bottom).

4.4 Strategy Realization and Integration

Three changes to the original code are required to implement the proposed strategy:

1. Creation of an additional dimension to the matrices, as described in Section 4.2 (Context), 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. 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. 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 [17].Such process has been tailored so that it is application independent and is it 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, the prolog is extended with additional functionality to record IDs and to initialize appropriate counters, and an epilog function is added 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 1024×1024 matrices).

Note that the presented solution requires no hardware change and it is a purely software-only solution realized on COTS Intel GPUs. All steps of the solution are GPU vendor agnostic except the intrinsics used in order to obtain the HT identifier where a given software thread is run. The particular intrinsics to use may change across GPU vendors and GPU models for a given vendor, and may not even exist for some GPU models or vendors, which would preclude integration onto those GPUs.

4.5 An illustrative Example

This Section details the application of the presented method to a specific example for illustration purposes. Step by step, the specific changes to be applied on the application code are shown, as well as the application independent routines and transformations used to achieve diverse redundancy.

Source code 4.1 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.

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

Source Code 4.1: **Original matrix multiplication CPU code**

The GPU version of this code is shown in 4.2.

```
__kernel void matrix_mult(const int size,
                          const __global float* A,
                          const __global float* B,
                          __global float* C)
{
  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;
  }
}
```

Source Code 4.2: **Original matrix multiplication GPU code**

It is important to recognize the call to `get_global_id(int)` which 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 1024×1024 dimensions, there will be $1,048,576$ software threads that will be scheduled by the hardware scheduler.

Source 4.3 shows the modified GPU code for the software threads.

```
__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;
    }
    FOOTER(i,j)
}
```

Source Code 4.3: **Modified matrix multiplication GPU code.**

As shown, modifications are trivial to apply:

- *(Only for debug purposes)* Add an additional variable in the declarations, `info`, but only for debugging purposes. This declaration would be dropped for a production version of this solution.
- *(Mandatory)* 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, `HARDTYPE` is called 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.

  ```
  #define HARDTYPE(x, NAME, SIZE)
  x * NAME = NAME + (intel_get_subslice_id()*SIZE);
  ```

- *(Mandatory)* Call the `HEADER` function (see the following code snippet), which computes the actual part of the work to carry out (i.e., the part of the result matrix to be computed by the actual software thread) 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 (i.e., the loops to traverse the corresponding part of the input data/matrices to produce the part of the result matrix allocated to the actual software thread).

```
1  #define HEADER(size1, size2)
2  dev_info DBG;
3  threadWorkID THD;
4  THD.devI = &DBG;
5  getHWResourcesINFO(&DBG);
6  getWorkitemsINFO(&THD, size1, size2);
7      for (THD.lRow=0; THD.lRow < THD.Mrow; THD.lRow++){
8          THD.eRow = THD.gRow + THD.lRow;
9          if (!(THD.eRow < THD.sDm1)) continue;
10         for (THD.lCol=0; THD.lCol < THD.Mcol; THD.lCol++){
11             THD.eCol = THD.gCol + THD.lCol;
12             if (!(THD.eCol < THD.sDm2)) continue;
```

Looking at the above snippet:

- Lines 2-5 take care of initiating the required objects and save information regarding where the software thread is executing.
- Line 6 with the call to `getWorkitemsINFO` will produce the values for the size of the block that the software thread will execute (globalRow and globalCol, gRow and gCol respectively).
- Lines 9 till the end will either skip or iterate over the computation that the thread has to execute (using gRow and gCol generated from line 6).

```
1  /**
2  * @brief Calculation of the workItem identification
3  */
4  void getWorkitemsINFO(threadWorkID* a, uint size1, uint size2){
5    sDm1 = size1;
6    sDm2 = size2;
7    eSkp = size1 * size2 * ssid;
8    Mrow = get_max_local_workitems(0, size1, size2);
9    Mcol = get_max_local_workitems(1, size1, size2);
10   //returns a unique number between 0-671
11   HWthID = tid + euid*7 + dssid*8*7 + (ssid * 8*7*6);
12   if (size1 <= 1) {
13       Mrow = 1;
14       gRow = 0;
15       gCol = (tid + euid*7 + dssid*8*7) * Mcol;
16   } else {
17       gRow = (tid) * Mrow;
18       gCol = (euid + dssid*8) * Mcol;
19   }
20 }
```

Taking a closer look on how `getWorkitemsINFO` function works:

- Lines 5 and 6 keep the dimensions of the problem.

- Line 7 determines the Skip that should be done in relation to the current operational SS.
- Lines 8 and 9 are related to the actual number of HW threads and will return the maximum dimensions of the assigned block of the thread.
- Line 11 is a formula that will return a unique identifier from 0 to the number of HW threads.
- Lines 12-end are responsible for producing gRow and gCol if we have a vector (size1=1) or a matrix case.

Below the details of *get_max_local_workitems*() are shown, the function responsible of returning the maximum amount of work-items in each dimension:

```
1  /**
2  * @brief For each dimension (0 or 1) compute the max_workitems to be
3  *        computed by a thread.
4  */
5  uint get_max_local_workitems(uint dim, uint size_dim1, uint size_dim2)
6  {
7  float TotalSize = size_dim1*size_dim2;
8  float numAvaibleThreads;
9  float DimSize;
10 if(size_dim1 <= 1) {
11  DimSize=1;
12  numAvaibleThreads = (dim==0)?1:NUMB_THREADS_VECTOR2;
13 } else {
14  numAvaibleThreads = (dim==0)?NUMB_THREADS_MATRIX:NUMB_THREADS_MATRIX2;
15  DimSize = (dim==0)?size_dim1:size_dim2;
16 }
17 float fsw = ((TotalSize/numAvaibleThreads)/DimSize);
18 return CEIL(fsw);
19 }
```

This function performs a CEIL over the total size of the problem divided by the number of available HW threads as seen in line 17.

- *(Mandatory)* Finally a call to the `FOOTER` function is made (see snippet below), all whose statements intend to store debug information back and would be dropped for a production version of this solution. The only lines of code truly mandatory are the braces closing the loops created in the HEADER.

```
1  #define FOOTER(...)
2  THD.thID = THD.eCol * THD.sDm1 + THD.eRow + THD.eSkp;
3  save_resources(info, DBG, THD,start, end);
4  }
5  }
```

Note that the only part of the `HEADER` and `FOOTER` calls that is application dependent is the number of parameters and their values, since as many parameters as dimensions of the matrices on which to iterate are needed, 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.

4.6 Strategy Validation

In this Section, with 2 simple benchmarks we are going to visualize and validate how the proposed solution assigns blocks to software threads as well as which block is calculated by which thread. To do so the first benchmark is called *cell_index_is_value* and it is assigning the index of the cell we operate as a value. Reference to previous code snippets will be also presented to validate that everything is working as expected. The benchmark code is the following:

```
__kernel void cell_index_is_value(const int size, __global float* C,
                                  __global struct HardwareThreadInfo* info)
{
    HARDTYPE(float, C, size*size)
    HEADER(size,size)
    //ORIG CODE
    int i = get_global_id(0);
    int j = get_global_id(1);
    if (i < size && j < size)
    {
        C[i*size+j] += i*size+j;
    }
    FOOTER(i,j)
}
```

Running this benchmark for a 672×672 size matrix will calculate in total $672 \times 672 \times 2 = 903,168$ cells since the computation is done twice in different subslices. With the previously mentioned logic, function *get_max_local_workitems*() will return $Mrow = 96, Mcol = 14$, meaning that each thread will be responsible for computing blocks of size $96 \times 14 = 1344$ cells. Now lets see a visualization on how the technique works in that case:

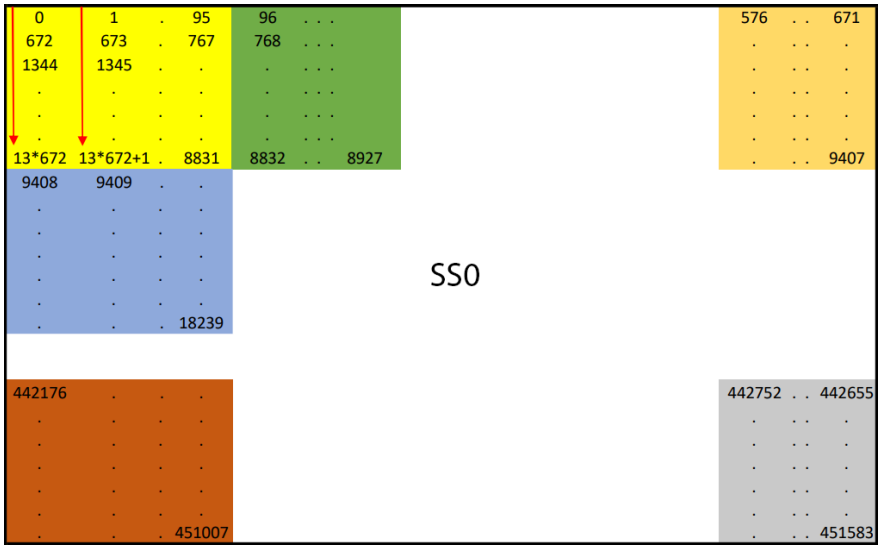


Figure 4.4: Cell work is done by columns in each block

From Figure 4.4 we can see that the technique will do the work by columns meaning that thread 0 will start

by calculating cell 0 then cell 672 and so on until cell 13*672 and then move to cell 1, cell 673 and so on.

Another benchmark developed to validate which block is done by which thread is called *which_hw_is_computing_block*. This benchmark's purpose is to save the HW thread ID that computed a specific cell. The code of the benchmark is shown in the following snippet:

```
__kernel void which_hw_is_computing_block(const int size,
                        __global float* C,
                        __global struct HardwareThreadInfo* info)
{
    HARDTYPE(float, C, size*size)
    HEADER(size,size)
    //ORIG CODE
    int i = get_global_id(0);
    int j = get_global_id(1);
    if (i < size && j < size)
    {
        C[i*size+j] = THD.HWthID%336;
    }
    FOOTER(i,j)
}
```

Visualizing the above benchmark we get the following Figure 4.5:

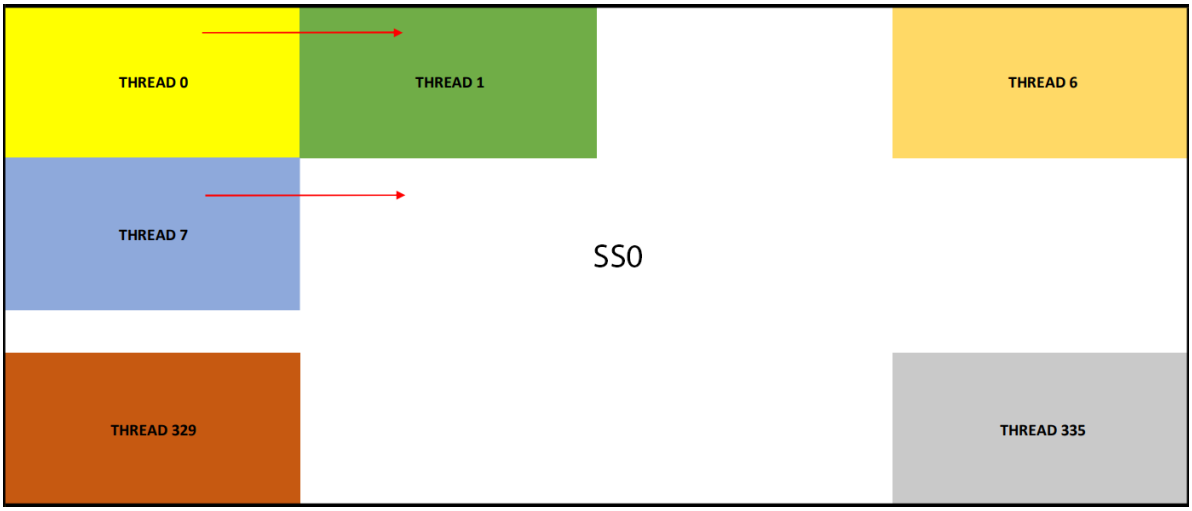


Figure 4.5: Block mapping to threads

With these 2 benchmarks we validate the proposed technique. We have seen where each block of cells is assigned in the GPU, and how each block gets computed.

Chapter 5

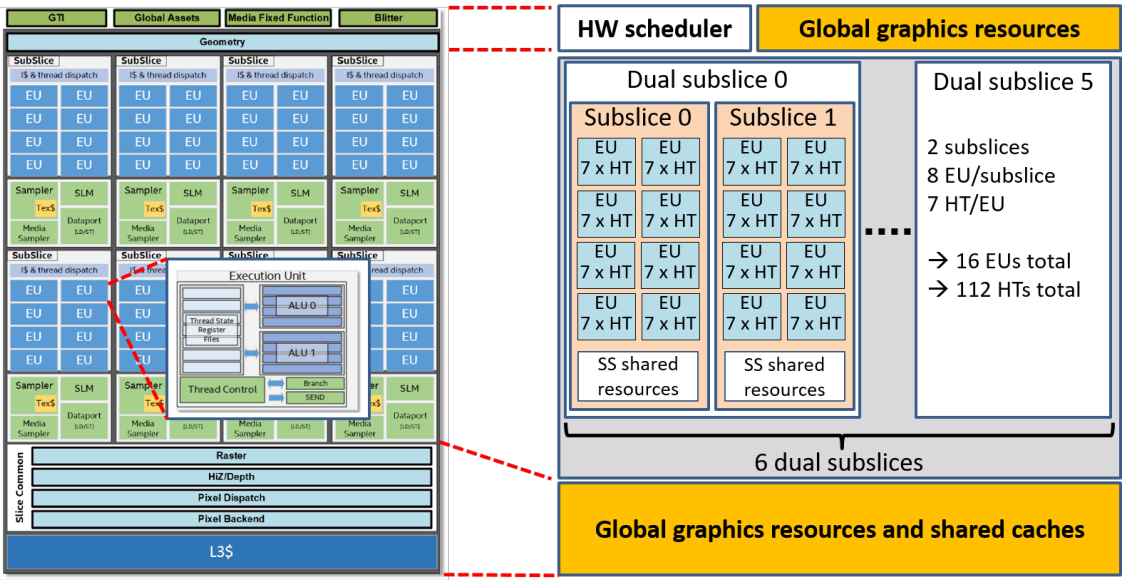
Experimental Setup

In this Chapter, the experimental setup used is comprehensively explained. Initially, the platform and benchmarks utilized in the study are presented. Subsequently, six distinct scenarios are introduced, which will be applied to the benchmarks for comparison purposes. The Chapter further summarizes the differences among these scenarios, providing a detailed explanation of their unique characteristics and implications.

5.1 Platform

In this work a 11th Gen Intel(R) Core(TM) i7-1165G7 CPU at 2.80GHz with an Intel(R) Iris(R) Xe Graphics [0x9a49] GPU is used. Going in more detail, the Intel GPU used 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 [17], therefore with 672 HTs in total (see Figure 5.1). Note that the description of the X^e_{LP} architecture in the corresponding technical reference manuals [15, 16], and the actual GPU implementation used in this work, include a single Slice with all the aforementioned components.

Since it is an embedded GPU, it is used for some services related to the display regardless of the system software used (e.g., Ubuntu, FreeBSD, Windows). Therefore, despite the disabling of as many interrupts as possible to minimize GPU interference, it is not possible to remove it completely in this setup. 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, obtained logs reflect that at least one HT has been allocated with more than one software thread whereas at least one HT has been allocated none, and hence, results are discarded. Overall, the experiments are repeated until achieving 6 runs per setup and matrix size without Linux interference, and report results using execution time averages.



(a) Generic Gen11 GPU with detailed components

(b) Geometry of the Gen12 GPU used in this paper

Figure 5.1: Intel GPU used for this thesis (right side)

5.2 Benchmarks

To evaluate the proposed technique, kernels used in neural networks such as those implemented in autonomous driving frameworks were selected. Hence this kind of kernels, if deployed on GPUs, shall have strict functional safety requirements imposing the use of diverse redundancy. Note that, to have full control of the code executed, the fully optimized APIs are not used and, instead, simple implementations of the benchmarks are chosen. The list is as follows:

- Matrix multiplication (MxM)[1]. Different square matrices are considered, 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 N×N 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) 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) applied on a 1D input to compute each element of the 1D result

[1]Note that "M×M" is used to refer to the benchmark and "N" or "N×N" to refer to the size of the dimensions of the matrices.

vector as a function of a region of the input 1D vector. 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) 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.

In order to evaluate the proposed solution, 6 different scenarios are developed, which were applied to the aforementioned benchmarks. The scenarios are chosen naturally to reach the final diverse and redundant proposal. In Figure 5.2 you can see a schematic of these scenarios. Below that, each setup is briefly summarized.

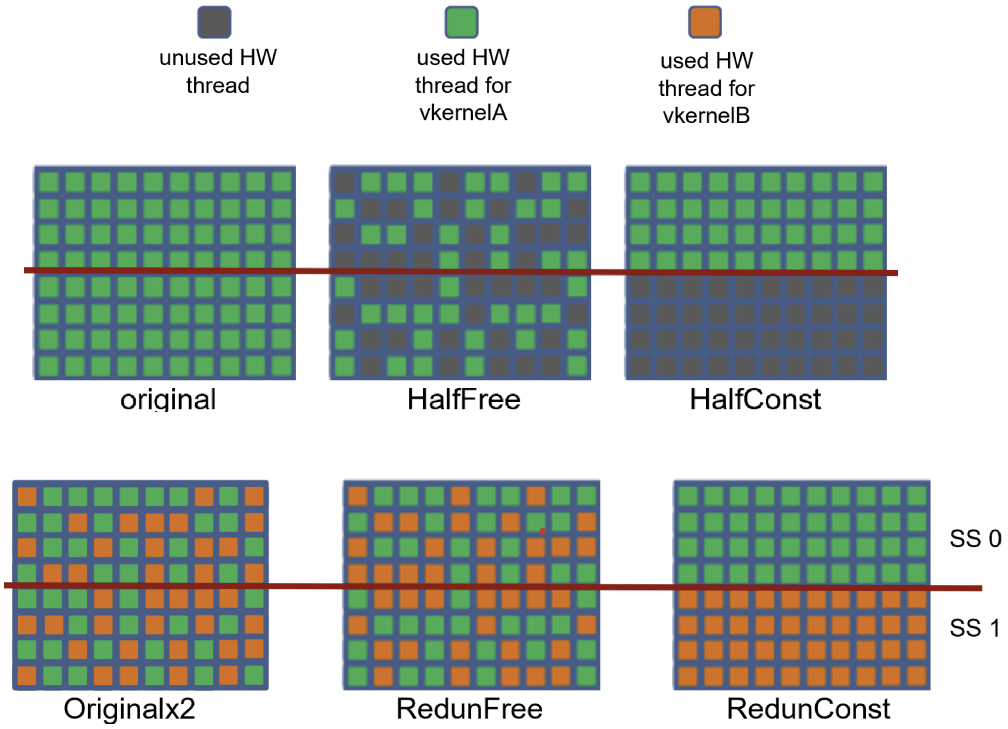


Figure 5.2: The different scenarios that we evaluate. From *original* (top left) to the proposed solution *RedunConst* (bottom right).

- *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 a software scheme to run only in HTs whose SS id is 1. Therefore, the kernel is constrained 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 the proposed solution. It is analogous to *RedunFree*, but software threads are controlled by a 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, diverse redundancy is achieved.

Those setups allow the comparison of *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 cause 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 the comparison of *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.

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

In order to develop all the different scenarios, critical parts of the code have to change. As explained in Chapter 4, the most important functions that will enable a specific execution strategy are the `HARDTYPE`, and the `HEADER`. `HARDTYPE` is only doing something if we actually compute 2 kernels (virtual kernels as stated before) meaning that it is activated only for Originalx2, RedunFree and RedunConst setups. The `HEADER` function will always change a bit, since each setup has a different computation for which part of the actual work to assign to a software thread. More specifically the change will be in the `getWorkitemsINFO` function of the `HEADER`. Finally for each setup we have different dimensions for the NDRange (The grid of threads essentially). In the following Table 5.1 you can see the important changes for the different setups.

Taking a closer look on this Table, the first thing to note is that `HARDTYPE` column is empty for HalfFree and HalfConst, since in these setups there is only 1 kernel executing.

Setups	NDRange	HARDTYPE	getWorkItemsInfo
Original	[size, size]	-	-
Originalx2	[size, size, 2]	use the third dimension in order to point on which vkernel we are operating $[0:1] \cdot size^2$	use only the eSkp to use the third dimension for the calculation eSkp = $[0:1] \cdot size^2$;
HalfFree	$[48, 7 \cdot 2]$	-	gRow = [0:13]; gCol = [0:47]; eSkp = 0; gRow *= Mrow; gCol *= Mcol;
HalfConst	$[48, 7 \cdot 2]$	-	eSkp = $size^2 \cdot ss\_id$; gRow = $(t\_id) \cdot$ Mrow + eSkp; gCol = $(eu\_id + dss\_id \cdot 8)$·Mcol + eSkp;
RedunFree	[48, 7, 2]	use the third dimension in order to point on which vkernel we are operating	gRow = [0:6]; gCol = [0:47]; eSkp = $[0:1] \cdot size^2$; gRow *= Mrow; gCol *= Mcol;

Table 5.1: Summary of changes for each setup

For the Original version, as expected there is only an NDRange grid of size*size and the rest of the functions do nothing in order to let the kernel execute as it would normally do with the OpenCL backend.

For the Originalx2, a 3D NDrange is needed so the 3rd dimension can be used as a guide for the second kernel. This will allow the use of the 3rd dimension in the `HARDTYPE` function to stride the input and/or output arrays. In more detail since the 3rd dimension is 2, we will get for each thread either a 0 or 1 when we call `get_global_id(2)`. So as it is shown in the Table, whenever `get_global_id(2)` returns 0 we operate on vkernelA, and when it returns 1, we operate on vkernelB.

Moving to HalfFree and HalfConst as it can be seen, `HARDTYPE` is not activated. This is because in both of these setups we do not utilize redundancy (only one kernel instead of vkernelA and vkernelB). The NDRange now is a 2D grid of exactly 672 software threads for both setups, meaning that we move on spawning the same amount of SW threads as available HW threads to observe the results. Note that from the loop logic in the `HEADER` function (remember Chapter 4), from these 672 threads spawned only half of them (336) will actually perform work and the rest will be skipped (eRow and eCol will exceed the problem dimensions). The main difference between them is on work assignment. HalfFree is free to compute a cell in any SS since the intrinsics are not used. HalfConst however, will use *ss_id, t_id, eu_id, dss_id* in order to only perform unique work on SS 0.

Finally RedunFree changes are shown. NDRange is a 3D grid of a size of 672, and the third dimension will be used for creating the virtual kernels and achieve redundancy. `HARDTYPE` will use that dimension to point on which vkernel we are working on. Here the work allocation is similar to HalfFree, with the difference that with the allowed values of gRow, all SW threads (672) will perform work.

RedunConst, which is the proposed technique is omitted from the Table since it is already explained extensively in the previous Chapter.

Chapter 6

Results

This Chapter presents the results taken with the proposed technique. First an in-depth analysis of the MxM results is done since of its global importance. Then the results on the rest of the benchmarks are shown and explained. As a conclusion, a theoretical comparison with the NVIDIA-specific solution is mentioned.

For the sake of simplicity, a detailed evaluation of the MxM benchmark is performed, while a broader set of benchmarks discussing only those effects differing from the MxM case will be later discussed. Emphasis is given since MxM already exposes most of the relevant scenarios.

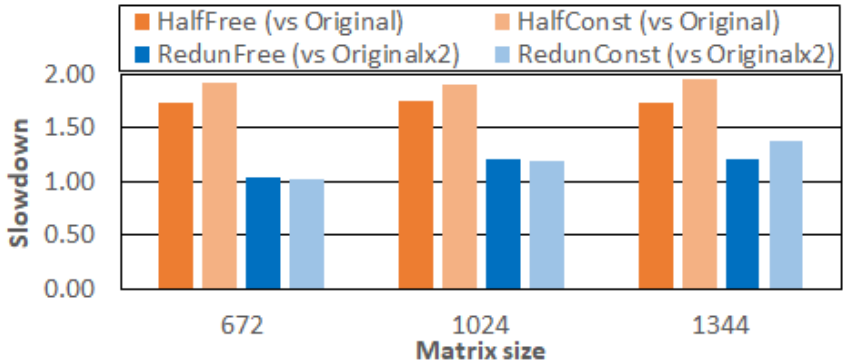


Figure 6.1: Slowdowns for the MxM for all *Half* and *Redun* setups and matrix sizes considered w.r.t. *Original* and *Originalx2*.

Figure 6.1 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 6.1 also shows that the slowdown of *HalfFree* is around 1.75× across all matrix sizes, hence below the expected 2×. 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*, it should be noted 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* the 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* 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 6.2 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 2× due to performing twice the same amount of work, is quite close to that ratio in practice across matrix sizes (between 2.03× and 2.06×).

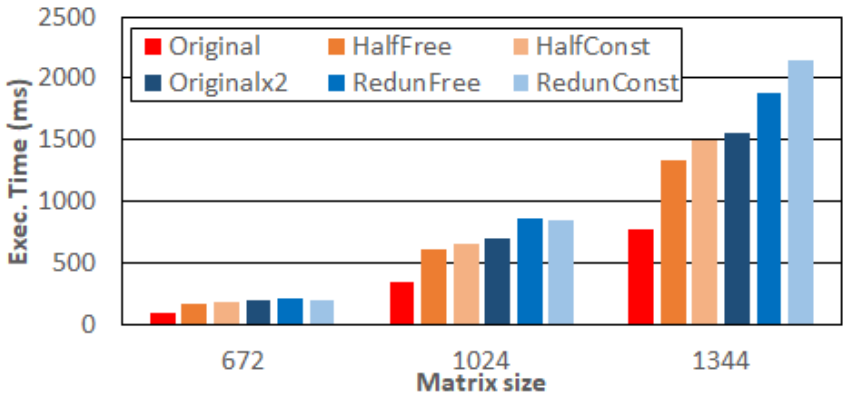


Figure 6.2: Execution times (in millions of cycles) for the MxM for all setups and matrix sizes considered.

6.1 Other Benchmarks Results

Figure 6.3 shows *RedunConst* w.r.t. *Original* for all the other benchmarks and three different sizes of the problem. The expected slowdown should be a bit above 2× due to the following reasons:

(A) the 2× 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. 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 2× slowdown expected due to (A) above, with negligible impact due to (B) and (C) (between 1% and 8%).

LRN, Mconv, and NN also experience a 2× 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 2.93×, 2.49× and 2.46×, being highly stable across matrix sizes.

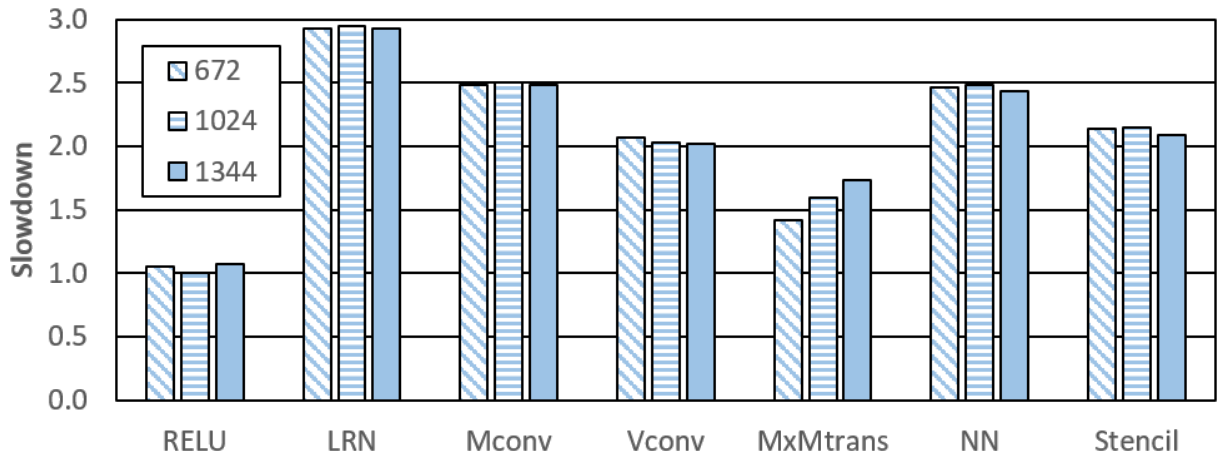


Figure 6.3: Slowdown of *RedunConst* w.r.t. *Original*.

The remaining benchmarks (RELU and MxMtrans) show, instead, lower slowdowns and an analysis is done 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, this particular work split of RELU is used 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 2× 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 2×. Note also that, as the matrix sizes increase, the slowdown approaches 2× 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 the proposed 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.

6.2 Comparison with NVIDIA-specific Solution

Note that, while the solution proposed to achieve diversity on GPUs from NVIDIA 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 the proposed solution, *RedunConst*, is comparable to the one that would be obtained with the NVIDIA solution, but providing stronger diversity guarantees.

As explained before, this solution provides stronger guarantees than those existing for NVIDIA GPUs and can be applied to other GPU families that provide analogous support to that of Intel ones.

In particular, to enable the use of the solution, the target GPU should provide appropriate support (e.g., intrinsics) that allow a software thread identify the hardware thread it has been allocated to, and use such hardware thread identifier to select the appropriate work to do by the software thread. Such mapping between hardware threads and work to do should be performed statically a priori, as done in our realization for Intel GPUs. If the target GPU does not provide support to determine the actual hardware thread where a software thread is running, portability is not possible.

Chapter 7

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. Despite this hardware limitation, 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. In contrast, this work focuses on Intel GPUs, which are becoming increasingly attractive in the automotive domain. Their advanced observability features are leveraged, like the ability to determine the particular hardware thread where each individual software thread runs. This feature elicits the creation of a ***software-only*** mechanism that provides diverse redundancy and guarantees that strict diversity is achieved for all computations, hence avoiding the uncertainties of the aforementioned solution for NVIDIA GPUs by construction. This approach can be generalized to GPUs other than Intel ones if those provide analogous support that allows any software thread to identify the hardware thread where it is running.

With this software-only solution Intel GPUs can be used 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. Results show that performance costs are low (e.g., around 9% for the ubiquitous matrix multiplication). Moreover, the proposed 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.

While this solution proves being effective, it relies on the assumption that the hardware scheduler allocates HTs in a strict round-robin fashion, and on the ability to split the kernel in exactly as many software threads as HTs. Part of future work consists of removing those constraints by enabling diverse redundancy in less ideal scheduling scenarios with no strict round-robin, and for kernels split into a number of hardware threads potentially smaller or larger (even much larger) than the number of HTs in the GPU. Additionally, this solution can be extended and evaluated on different GPU vendors, since it is implemented with a hardware-agnostic language (OpenCL). Also, an advanced recovery mechanism can be implemented consisting of either re-executing the redundant kernels or just adding another level of redundancy, which implies computing 3 copies at the same time if the HW resources, performance, and power constraints allows it.

Bibliography

[1] Irune Agirre, Francisco J. Cazorla, Jaume Abella, Carles Hernández, Enrico Mezzetti, Mikel Azkarate-askatsua, and Tullio Vardanega. Fitting Software Execution-Time Exceedance into a Residual Random Fault in ISO-26262. *IEEE Transactions on Reliability*, 67:1314–1327, 2018.

[2] Sergi Alcaide, Carles Hernandez, Antoni Roca, and Jaume Abella. DIMP: A Low-Cost Diversity Metric Based on Circuit Path Analysis. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. ACM, June 2017.

[3] S. Alcaide et al. High-Integrity GPU Designs for Critical Real-Time Automotive Systems. In Jürgen Teich and Franco Fummi, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pages 824–829. IEEE, 2019.

[4] S. Alcaide et al. Software-only Diverse Redundancy on GPUs for Autonomous Driving Platforms. In Dimitris Gizopoulos, Dan Alexandrescu, Panagiota Papavramidou, and Michail Maniatakos, editors, *25th IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS 2019, Rhodes, Greece, July 1-3, 2019*, pages 90–96. IEEE, 2019.

[5] S. Alcaide et al. Software-Only Triple Diverse Redundancy on GPUs for Autonomous Driving Platforms. In *50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020 - Supplemental Volume*, pages 82–88. IEEE, 2020.

[6] S. Alcaide et al. Achieving Diverse Redundancy for GPU Kernels. *IEEE Trans. Emerg. Top. Comput.*, 10(2):618–634, 2022.

[7] Nikolaos Andriotis, Alejandro Serrano-Cases, Sergi Alcaide, Jaume Abella, Francisco J. Cazorla, Yang Peng, Andrea Baldovin, Michael Paulitsch, and Vladimir Tsymbal. A Software-Only Approach to Enable Diverse Redundancy on Intel GPUs for Safety-Related Kernels. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, page 451–460, New York, NY, USA, 2023. Association for Computing Machinery.

[8] Rajkumar Buyya, Toni Cortes, and Hai Jin. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 2–14. IEEE, 2002.

[9] CENELEC. EN50126. Railway Applications: The Specification and Demonstration of Dependability, Reliability, Availability, Maintainability and Safety (RAMS), 2012.

[10] C. L. Chen and M. Y. Hsiao. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[11] Robert Feldt, Francisco G. de Oliveira Neto, and Richard Torkar. Ways of Applying Artificial Intelligence in Software Engineering. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, RAISE '18, page 35–41, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Spiking Neural Networks. *International journal of neural systems*, 19 4:295–308, 2009.

[13] Carles Hernandez and Jaume Abella. Timely Error Detection for Effective Recovery in Light-Lockstep Automotive Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1718–1729, Nov 2015.

[14] 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.

[15] 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.

[16] 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.

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

[18] International Electrotechnical Commision. IEC 61508:2010, 2010.

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

[20] ISO. *ISO/IEC DIS 22989: Information technology - Artificial Intelligence - Artificial Intelligence Concepts and Terminology (draft)*, 2021.

[21] Xabier Iturbe, Balaji Venu, Juergen Jagst, Emre Ozer, Peter Harrod, Chris Turner, and John Penton. Addressing Functional Safety Challenges in Autonomous Vehicles with the Arm TCL S Architecture. *IEEE Design & Test*, 35(3):7–14, 2018.

[22] Khronos OpenCL Working Group. *The OpenCL^{TM} Specification*, 10 2020.

[23] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Honza Cernocký, and Sanjeev Khudanpur. Recurrent Neural Network based Language Model. In *Interspeech*, 2010.

[24] Subhasish Mitra, Nirmal R. Saxena, and Edward J. McCluskey. Techniques for Estimation of Design Diversity for Combinational Logic Circuits. In *DSN*, 2001.

[25] Subhasish Mitra, Nirmal R. Saxena, and Edward J. McCluskey. A Design Diversity Metric and Analysis of Redundant Systems. *IEEE Trans. Comput.*, 51(5):498–510, may 2002.

[26] Subhasish Mitra, Nirmal R. Saxena, and Edward J. McCluskey. Efficient Design Diversity Estimation for Combinational Circuits. *IEEE Trans. Comput.*, 53(11):1483–1492, nov 2004.

[27] OpenAI. Gpt-4 technical report, 2023.

[28] W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[29] Heather Quinn, Zachary Baker, Tom Fairbanks, Justin L. Tripp, and George Duran. Robust Duplication With Comparison Methods in Microcontrollers. *IEEE Transactions on Nuclear Science*, 64(1):338–345, 2017.

[30] RTCA and EUROCAE. DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification, 1992.

[31] Saeed Asadi Bagloee, Madjid Tavana, Mohsen Asadi, and Tracey Oliver. Autonomous Vehicles: Challenges, Opportunities, and Future Implications for Transportation Policies. *Journal of Modern Transportation*, 2016.

[32] A. Serrano-Cases, A. Martínez-Álvarez, R. Possamai Bastos, and S. Cuenca-Asensi. Bare-Metal Redundant Multi-Threading on Multicore SoCs under Neutron Irradiation. *IEEE Transactions on Nuclear Science*, pages 1–1, 2023.

[33] Christian Szegedy, Alexander Toshev, and D. Erhan. Deep Neural Networks for Object Detection. In *NIPS*, 2013.

[34] VDE. *VDE-AR-E 2842-61: Development and Trustworthiness of Autonomous/Cognitive Systems*, 2021.

[35] Wikipedia contributors. Safety-critical system — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Safety-critical_system&oldid=1112933768, 2022. [Online; accessed 13-April-2023].

[36] Wikipedia contributors. Advanced driver-assistance system — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Advanced_driver-assistance_system&oldid=1148251318, 2023. [Online; accessed 13-April-2023].

[37] Wikipedia contributors. Nvidia jetson — Wikipedia, the free encyclopedia, 2023. [Online; accessed 12-June-2023].

[38] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*, 8:58443–58469, 2020.