Vector Extensions in COTS Processors to Increase Guaranteed Performance in Real-Time Systems

Roger Pujol[†,*], Josep Jorba[*], Hamid Tabani[*], Leonidas Kosmidis[*,†], Enrico Mezzetti[*], Jaume Abella[*], and Francisco Cazorla[*]

[†]Universitat Politècnica de Catalunya
[*]Barcelona Supercomputing Center

Abstract

The need for increased application performance in high-integrity systems like those in avionics is on the rise as software continues to implement more complex functionalities. The prevalent computing solution for future high-integrity embedded products are multi-processors systems-on-chip (MPSoC) processors. MPSoCs include CPU multicores that enable improving performance via thread-level parallelism. MPSoCs also include generic accelerators (GPUs) and application-specific accelerators. However, the *data processing approach* (DPA) required to exploit each of these underlying parallel hardware blocks carries several open challenges to enable the safe deployment in high-integrity domains. The main challenges include the qualification of its associated runtime system and the difficulties in analyzing programs deploying the DPA with out-of-the-box timing analysis and code coverage tools. In this work, we perform a thorough analysis of vector extensions (VExt) in current COTS processors for high-integrity systems. We show that VExt prevent many of the challenges arising with parallel programming models and GPUs. Unlike other DPAs, VExt require no runtime support, prevent by design race conditions that might arise with parallel programming models, and have minimum impact on the software ecosystem enabling the use of existing code coverage and timing analysis tools. We develop vectorized versions of neural network kernels and show that the NVIDIA Xavier VExt provide a reasonable increase in guaranteed application performance of up to 2.7x. Our analysis contends that VExt are the DPA approach with arguably the fastest path for adoption in high-integrity systems.

The final publication is available at ACM via http://dx.doi.org/10.1145/3561054

1 Introduction

Improving guaranteed and average performance of applications on high-performance embedded platforms is an emerging challenge across all real-time high-integrity domains. This is driven by the increasing use of software applications to control complex safety-related functionalities that involves handling large amounts of data and implementing complex artificial intelligence (AI) algorithms [12, 25, 28]. MPSoCs comprising GPUs and/or othe specific accelerators are the key hardware technology used to provide the required performance levels. At the software level, several DPAs exploit the computing capabilities of the underlying parallel hardware to provide the required increased performance.

Thread-level parallelism (on multicore processors) is a DPA that introduces disruptive changes in high-integrity software design, development and verification: multi-threaded applications can introduce race conditions and require changes to the consolidated timing analysis and code coverage tools to be supported [63, 45, 46, 17]. Furthermore, parallel programming models (e.g., OpenMP) require time to analyze and qualify their associated runtime system for thread management. In fact, data sharing among threads commonly builds on hardware-supported coherence, which is a major challenge for timing analysis [57, 23, 56].

GPUs are another DPA that brings its own certification challenges. Very little information is disclosed by GPU manufacturers about the internal functioning of the GPU's hardware or software [1], which makes GPU timing analysis an open challenge. Consequently, despite the numerous reverse engineering and characterization efforts that have studied GPU's internals [68, 2, 43, 11], no out-of-the-box timing analysis tool currently supports GPUs. Also, general-purpose computing languages such as CUDA, OpenCL, HIP [44], and Vulkan [13] on GPUs bring additional certification issues due to the use of pointers and dynamic memory allocation, violating guidelines for high-integrity systems such as MISRA-C. This hampers their certification [62]. On the other hand, certified graphics and vision APIs, e.g., OpenGL SC 2 and OpenVX SC, have been used for the acceleration of general-purpose computations on GPUs [9], but they have their own limitations: the timing aspects of these languages are largely ignored by their specifications [3], which compounds the limited information about the GPU hardware to derive tight WCET estimates for GPUs.

This work explores a complementary DPA to increase application performance: vector extensions (VExt). VExt implement vector/SIMD (single-instruction multiple-data) processing with which the same operation is performed independently and simultaneously on several pieces of data[1]. Modern processors natively support VExt via vector/SIMD instructions in the Instruction Set Architecture (ISA), like Arm's NEON VExt, including processors under evaluation for their use in high-integrity systems, e.g. the NXP's T2080 [20] in avionics or the NVIDIA's Xavier [41] in automotive. Furthermore, a wide range of time-critical

[1]In Sections 2 and 4 we respectively introduce and discuss the properties of two related technologies to VExt for high-integrity systems, namely, scalable vector extensions (scalable VExt) and vector co-processors (VCOP), whose main related works we describe in Section 6.

applications (such as Autonomous Driving) include several processes with various execution time granularities[50]: the processing data of some applications are too small to be efficiently computed with a GPU or other accelerators but are performance-critical amenable for parallelization. However, to our knowledge, while VExt incur no extra cost in hardware development and validation as they are already present in processors for real-time systems, they are not yet exploited in high-integrity systems. This is mainly due to the lack of understanding of certification implications of vectorized code and the qualification of the support tools required to that end. In order to cover this gap, we perform the following contributions:

1. We carry out a thorough analysis, via key relevant metrics for high-integrity systems, of the main limiting factors for the certification of software using other DPA and the necessary tool qualification (Section 3).

2. We show that VExt provide advantages in all these metrics (Section 4), including the following: (i) VExt require no runtime support to reduce qualification costs. (ii) VExt build on easy-to-qualify vector built-in types and compiler constructs, which simplify achieving MISRA-C compliance. (iii) VExt build on hardware no more complex than regular scalar hardware and (iv) prevent race conditions that can arise with parallel programming models, while (v) enabling the use of out-of-the-box code coverage tools, (vi) VExt are compatible with out-of-the-box measurement-based tools, and (vii) VExt present no major roadblock for their adoption by static timing analysis.

3. Using the Arm NEON VExt, we develop vectorized versions of two representative neural network kernels widely used for autonomous operation. We show how commercial and qualified timing analysis and code coverage tools can analyze these vectorized versions without requiring any change and reporting no compatibility issue. This allows us to collect software timing and coverage results, respectively. Results show that we achieve full code coverage and competitive performance improvements for the vectorized neural network kernels, up to 2.7x in the Jetson Xavier AGX, over non-vectorized versions (Section 5).

Overall, we contend that in the short term, VExt are the DPA with the fastest path for adoption in high-integrity systems and achieving good performance improvements. In the long term, VExt are compatible with any other DPA, though this depends on the time it takes the different paradigms to reach certification. As an illustrative example, we show how OpenMP and VExt can be combined to exploit SIMD and thread-level parallelism, reaching performance improvements of up to 12x over baseline scalar single-threaded versions of neural network kernels (Section 5).

The rest of this work is structured as follows: Section 2 provides some basic background on vector computing. Section 3 compares different DPA with

relevant metrics for high-integrity systems. In Section 4, we develop VExt advantages for real-time systems using those metrics. In Section 5, we assess the performance benefits of VExt for several representative neural network kernels. Section 6 and 7 present the related works and conclusions, respectively.

2 Background on Vector Extensions

2.1 Introduction

VExt (SIMD processing) perform the same operation on multiple data simultaneously. A scalar instruction acts on one piece of data at a time, while a single vector instruction can act on several pieces of data at the same time, as shown in Figure 1. Hence, VExt exploit data-level parallelism but not concurrency. VExt are consistently supported across processor generations by all major vendors, so they can be deemed as COTS consolidated technology. This is fundamental in high-integrity domains where industry seeks to exploit consolidated hardware technologies, as opposed to hardware features that can be removed in future processor generations due to the software's high development costs.

For instance, Intel included x86 processors support for Streaming SIMD Extensions (SSE) that was followed by Advanced Vector Extensions (AVX). Arm also introduced an advanced SIMD extension called NEON, which is a combined 64- and 128-bit data SIMD instruction set. Other chip vendors also provide SIMD vector extensions like IBM's AIX and NXP's Altivec. While each VExt architecture/implementation has its own instruction set, width and capabilities, they all have a common substrate, as shown next.

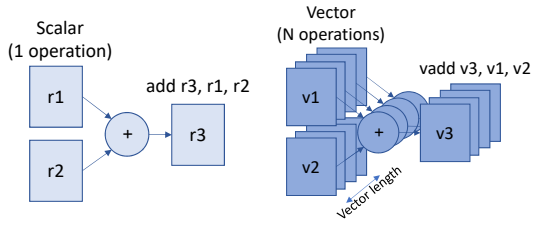


Figure 1: Example of a scalar operation vs. vector operation.

Vector extensions are exposed to software via vector instructions in the ISA that use architectural vector registers on which vector instructions operate. A vector instruction has the same format as any other scalar instruction comprising operation code, source register(s), and a destination register. These are software-visible vector architectural registers. Vector instructions operate on long operands that may contain different values of different sizes 8 bits, 16 bits, 32, and 64 bits and types (e.g., integer and floating-point). Vector load/store instructions operate at these different granularities.

2.2 Hardware Implementation

At the hardware level, vector instructions are fetched and decoded as regular scalar operations. In Figure 2, the instruction fetch stage comprises the instruction cache and MMU, and branch prediction logic which operates seamlessly on scalar and vector instructions. Instructions go to the instruction buffer that decouples the fetch logic from the rest of the pipeline. Instructions are then decoded and renamed to remove read-after-write and write-after-write dependencies. Next, they are dispatched to a different issue queue (IQ). In all these processes, there is no difference between scalar and vector operations.

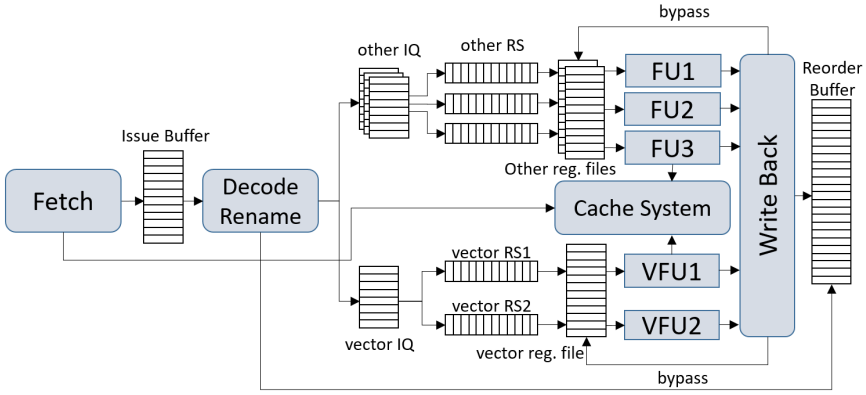


Figure 2: Reference Processor Pipeline with support for VExt

Processors supporting vector operations have a vector pipeline (see the bottom part in Figure 2) in the same way they have the integer, floating-point, and load/store pipeline (top part of Figure 2). The vector pipeline starts with the vector IQ, where instructions are held until their input operands are ready. When this happens, instructions are sent to the reservation stations (RS). In the vector pipeline, there can be several types of vector functional units (VFU) to execute simple vector integer operations: vector simple integer unit (VSIU) to execute short-latency instructions (addition, subtraction, maximum, minimum, comparisons, etc.), vector complex integer unit (VCIU) to execute complex and longer-latency operations (multiplication and multiplication/addition), vector floating-point unit (VFPU) to execute vector float operations. This matches the structure of the scalar integer and floating-point pipelines that have their own functional units (FU). For memory operations, vector load/store units (VLSU) execute vector accesses to cache/memory, similar to the scalar LSU unit. Note that in Figure 2 VLSU/LSU are shown as the regular VFU/FU accessing the cache system block. After execution, results are consolidated in the vector register file, which keeps vector registers (e.g., 32) plus the vector rename registers. Vector and scalar operations are kept in the reorder buffer to provide a safe save/recovery from interrupts, precise exception management and maintain

program execution order.

Overall, there are no major differences in the semantics of scalar and vector operations, besides, of course, that the latter operate on several pieces of data. Scalar and vector operations share part of the pipeline, notably the part related to the control flow, which simplifies timing analysis and code coverage of vector programs.

2.3 Scalable VExt and Vector Co-Processors (VCOP)

The nominal notion of VExt implies that the vector sizes exposed in the ISA define the size of the vector registers and VFU (e.g. ARM's NEON and x86's AVX).

In terms of Scalable VExt, recently Arm has introduced Scalable Vector Extension (SVE) [59] and RISC-V the Vector Extension [49] which provide a vector-length agnostic abstraction so that the data processed by vector instructions do not need to match the size of the vector register and the VFU. In that case, the execution of a vector instruction is not done at once, but it is 'chunked' according to the vector register size with vector chunks executed sequentially.

Vector co-processor (VCOP), unlike VExt that are integrated as an additional execution path within the data path of the main processor, are external to the processor and it is paired with it. To that end, the cpu processor sends vector decoded instructions to a buffer the VCOP which dequeues, executes them, and send the results back. Note that the RISC-V Vector Extension can be implemented both as SIMD as well as VCOP [49].

The main insight and conclusions we develop for VExt apply to both scalable VExt and VCOP (we further develop on this in Section 4.6). The main reason for our choice of VExt is that it is a technology which has existed for decades and that is now consolidated, with processors undergoing a certification process in avionics, like the NXP T2080 [51] and the Xilinx Zynq UltraScale+ [67] already featuring VExt in their CPUs.

3 Other DPAs

Next, we assess the readiness of several DPAs in high-integrity domains. The comparison in Table 1 covers relevant metrics for hardware support and software ecosystem around each approach, in line with high-integrity systems' certification guidance documents such as DO-178C [54] (software), CAST-32A [14] (multicore-related aspects), and DO330 [55] (tool qualification) for avionics; and ISO 26262 [27] for automotive and tool qualification. We use the following metrics:

(M1) compliance with the MISRA-C coding standard;

(M2) changes required to out of the box (OOB) timing analysis tools/techniques to analyze software programmed with the considered data processing paradigm;

(M3) changes required to OOB code coverage tools;

(M4) whether race conditions can arise;

(M5) whether a runtime software is required and

(M6) it is open, in case it is not certified;

(M7) the analyzability of required hardware to simplify deriving tight WCET estimates;

(M8) programming model required to achieve the acceleration;

(M9) the existence of widely-used libraries accelerated using the data processing approach that reduces development and certification cost if they are already used for autonomous systems.

More in detail, the relevance of these metrics lies on the fact that for embedded high-performance critical systems, software must adhere to safety standards and best industrial practices for design, validation and verification (V&V), in line with the usual V development model illustrated in Figure 3. For instance, (M5) and (M6) relate to the architectural design of the software part of the system, which will change depending on whether a runtime is used or not. (M1) relates to the coding guidelines when implementing software units, and (M8) and (M9) to performance-related implementation choices. (M7) is related to the hardware implementation in general, and to the hardware platform selection in particular. Finally, (M2), (M3) and (M4) relate to software and system testing activities intended to validate that the system design adheres to its specifications.

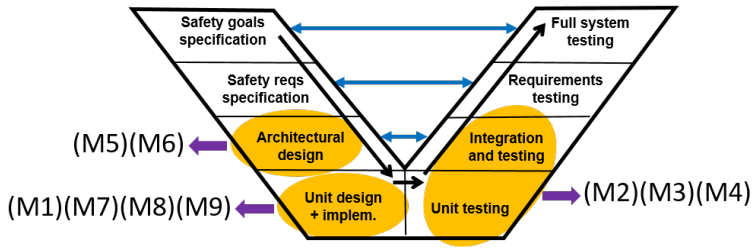


Figure 3: Metrics defined in this work in relation to the standard V system development model (design and verification on the left, implementation at the bottom and, testing/validation on the right) for systems with safety requirements like automotive [27].

3.1 Multicore Programming Models (OpenMP/pthreads/MTAPI)

For single-node shared-memory systems, OpenMP is the de facto programming model for high-performance computing. As such, it is also evaluated for high-integrity systems to exploit multicore capabilities. There are mainly two approaches for its adoption in high-integrity domains:

Table 1: Comparison of data processing methodologies using metrics of interest for high-integrity systems. '*' indicates that only some of the implementations provide this feature, but not in the general case.

		Data Processing Paradigm			
ID	**Feature**	OpenMP/pthreads/MTAPI	CUDA/HIP/OpenCL/Vulkan	OpenGLSC2/OpenVXC	Vector
M1	MISRA-C compliance	✗	✗	✓	✓
M2	OOB timing analysis tools	✗	✗	✗	✓
M3	OOB coverage tools	✗	✓*	✗	✓
M4	Avoids race conditions	✗	✓	✓	✓
M5	Needs no/simple runtime	✗	✗	✗	✓
M6	Open runtime (if any)	✓	✓*	✗	-
M7	Easy-to-predict hardware	✗	✗	✗	✓
M8	Long/Complex Code	✗	✓	✗	✓
M9	Commonly used libraries	✓	✓*	✗	✓

1. reusing the OpenMP specification and implementation for mainstream systems, i.e., as it is currently defined in its standard. However, this fails to cover many requirements for high-integrity systems [53, 36].

2. defining a high-integrity OpenMP specification. While the creation of an OpenMP standard for high-integrity systems has been going on for several years, there is no draft standard available. This is partially due to the fact that a high-integrity version is not going to be compatible with the core OpenMP specification [53]. For example, the core OpenMP standard does not define what should be the expected functional behavior when an error occurs during the execution of a parallel region [66]. While such features cannot be disregarded in the specification of the high-integrity version of the standard, their introduction will also cause compatibility issues. A compromise is still far from being achieved.

OpenMP's runtime system is responsible for thread management for the parallel processing (M5) and it is usually open-source, since it is an integral part of the GCC compiler suite (libgomp) (M6). For use in high-integrity systems, runtimes need to be *qualified* according to the applicable functional safety standard and support documents. Recently, there have been some research proposals in this direction [53, 36]. However, to our knowledge, the code of those solutions is neither publicly available nor qualified for use in the industry yet. In addition, none of the available runtimes are developed following safety standards or programming guidelines for high-integrity systems (M1), and therefore they are not qualified either.

Also, OpenMP programmers can indicate the number of threads for a program. However, this is taken simply as a recommendation and the particular OpenMP implementation is free to select larger values. This creates issues regarding the control of the OpenMP program's execution, which affects its verification (M4).

Although general-purpose profilers work properly with OpenMP programs, no out-of-the-box (OOB) qualified code coverage tool supports it yet (M3).

OpenMP programs are not MISRA-C compliant (M1) and since the OpenMP runtime is not written to comply with safety standards, it is hard to achieve 100% code coverage [53].

Parallel programming models, including OpenMP, exhibit fundamental differences with respect to sequential programming, mainly in the capability of concurrently sharing the same data across runtime entities. The necessary synchronization/communication primitives are typically designed for average performance and high throughput, at the expense of predictability and analysability [45, 46, 36]. This results in programs difficult to analyze with either static or dynamic approaches (M2). Also, as a shared memory programming model, OpenMP relies on hardware cache coherence between data caches and other local memories to simplify data sharing (M7). However, several works have demonstrated that cache coherence complicates the timing analysis of real-time systems. Consequently, several hardware mechanisms and real-time specific protocols have been proposed [23][29][58], but they have not been adopted in COTS processors yet.

On the positive side, OpenMP provides a good level of abstraction that does not require the programmer to use low-level device-specific features to obtain high performance (M8), reducing the possibility to get deadlocks or race conditions (M4) – frequent in alternative low-level multicore programming models such as pthreads and its lightweight counterpart, MTAPI, which has been explored by the European Space Agency for exploiting multicore processors in space [24]. Moreover, there are many available commonly-used libraries programmed with OpenMP (M9) such as ATLAS.

3.2 General-Purpose/Compute GPUs (CUDA/HIP/OpenCL/Vulkan)

In order to enable the use of GPUs in high-integrity systems, two paths can be followed: through a compute API, such as CUDA, or a graphics/vision API. Compute APIs are the most widely used ones in general-purpose computing, since they are easier to use (M8) because race conditions are easy to manage (M4). The reason is that thanks to the single instruction, multiple threads (SIMT) programming model followed by GPU programming models, the same instruction is executed by all threads, and therefore race conditions can be simply avoided by using atomic operations on memory positions that are written by multiple threads and by the use of synchronization primitives among multiple threads when shared memory is used. However, this does not translate into easy (direct) adoption for high-integrity systems due to other blocking issues described below.

Compute APIs include the proprietary CUDA language and implementation from NVIDIA and the open-source language HIP from AMD, which has nearly identical syntax with CUDA. OpenCL has an open specification with both open and closed source implementations. This is also true for Vulkan, which is a low-level API appropriate for both computing and graphics. However, as a low-level API, Vulkan requires a significant amount of complex code to be written, even for a simple operation. This makes it difficult to master, error-prone, and

increases the cost of certification of the application code (M8).

An important limitation of GPU compute data processing paradigms is compliance with safety standards and programming guidelines like MISRA-C, as they depend on pointers and dynamic memory allocations (M1) [62]. In addition, their runtime systems are not developed with safety in mind (M5), and they are based on massive and complex code bases that hamper their tool qualification.

Also to our knowledge only one provider has a qualified tool for CUDA code coverage [64](M3). In addition, GPUs experience significant delays depending on the data access patterns of these 32/64 elements (absence of memory coalescing) or when threads take different execution paths (control divergence), complicating timing analysis (M7).

In addition, no commercial timing analysis suites have support for timing analysis of GPUs (M2) either. This is partly because GPU hardware significantly lacks available technical information compared to CPU hardware used in high-integrity systems. In particular, GPU implementation details are not available in sufficient detail neither at hardware nor at the software level. In fact, there are several works in the real-time literature that focus on reverse-engineering the behavior of GPUs [68, 2, 43, 11]. For this reason, GPUs are used mostly as black boxes, preventing the availability of open runtime systems. Even in the case that an open specification exists (eg. OpenCL, Vulkan) for a given GPU and an open-source driver/runtime, such as in the case of AMD GPUs, this does not necessarily mean that important hardware details are revealed. Therefore, the construction of an accurate model for static timing analysis is not possible nor scales well with the hardware complexity (M7). The use of measurement-based timing analysis methods is also challenging on GPUs since they have complex architectures, which result in complex event interactions, potentially leading to high variability, as well as the inability to control low-level sources of jitter without knowledge about the hardware.

On the other hand, all general-purpose GPU languages, except for Vulkan, include widely used libraries such as cuBLAS, clBLAS, and hipBLAS (M9). Moreover, some OpenCL and Vulkan implementations have open-source versions of their implementations for some GPUs, such as the ones from AMD (M6). However, a specific problem faced in the case of HIP is that the source code is evolving very fast (being still under active development) with the effect of increasing its maintenance cost [44].

3.3 Graphics/Vision APIs (OpenGL SC 2 / OpenVX SC)

In addition to compute APIs, GPUs are currently used in high-integrity systems by employing a safety-certified (SC) API like OpenGL SC 2, which focuses on graphics operations, or OpenVX SC, which is used for vision operations [9]. These certified APIs exist because GPUs are already used in high-integrity systems for a long time in order to perform visualization-related tasks. For example, GPUs drive screens within aircraft cockpits and display speed or engine rotations per minute on cars' dashboards. These GPUs are programmed with MISRA-C

compliant (M1), certified graphics or vision solutions for the highest criticality levels of high integrity systems, which are subsets of the larger general-purpose standards. As such, their runtime system is smaller than the fully-fledged one of the general-purpose standards (M5), but it is still required and needs to be qualified. Obviously, commercial implementations for these APIs are provided, therefore their implementations are proprietary (M6), and consequently, important information about their internals is not available. Also, and more importantly, these solutions only focus on graphics and visual processing tasks instead of general-purpose computations. Although this has some benefits, e.g., thanks to this property their processing model resembles a pipeline, in which memory positions can be only read or written by only one parallel element, and thus avoiding data races (M4), they do not natively offer the flexibility of general-purpose computation.

In the case of OpenGL SC 2, while it can be leveraged for general-purpose computations [9], it can do so at the cost of a highly complex and excessively large codebase (M8), which again increases the certification costs. The same happens with OpenVX SC, where complicated task graphs need to be constructed manually by the programmer [3]. Complex compiler/tool support is required for both the above solutions in terms of GPU tooling, not differently from compute APIs (M2). This is because GPUs are not standalone systems but accelerators that need to be programmed, so complex, low-level operations are needed in the driver/runtime system to program the GPU and manage data transfers (M8).

Despite the fact that both OpenGL SC 2 and OpenVX SC are designed to facilitate certification, their standards do not provision timing [3], which is an essential element of real-time systems. As a consequence, both methods face the same challenges we identified in the previous subsection with compute APIs for the computation of WCET (M2), since they both rely on 'cryptic', highly complex GPU hardware (M7) that is not supported by commercial timing analysis tools (M2).

Another issue of these solutions is that they do not support interoperability with commonly used libraries, e.g., for matrix operations or machine learning inference, which are frequently used for the development of autonomous systems (M9).

3.4 Putting It All Together

Overall, GPU-related DPAs are not yet generally suitable for certified applications and systems, and only restricted applications of those – and with non-negligible costs – are suitable for safety-related systems as of now. Therefore, they cannot be considered as drop-in replacements of scalar computation and VExt, so we leave them out of any comparison.

OpenMP is currently being assessed as a future approach to gain performance building on parallelization, both considering safety-related domains [35, 36], and the space domain with the support of the European Space Agency [38]. Hence, while this approach is not yet generally compatible with safety-related

systems, we assess its potential in the evaluation section considering both, the use of OpenMP on scalar code, and on code combining OpenMP with VExt.

4 Vector Extensions

This section assesses VExt adopting the same metrics used in the previous section to evaluate other data processing paradigms.

4.1 Context of Application

A large range of time-critical applications in the context of Advanced Driver Assistance Systems (ADAS) and autonomous driving, such as those related to object detection and navigation, are good candidates for vectorization rather than for other DPAs such as on GPUs. There are two reasons for that, (1) due to the certification challenges brought by GPUs and (2) because of the characteristics of the application.

Regarding certification, GPUs are highly efficient in executing massively parallel workloads with large data sets. However, they cannot be generally deployed in high-integrity systems, and only a few specific GPUs and runtimes have been successfully included in certified applications with high costs for their certification (see Sections 3.2 and 3.3).

Regarding the application characteristics, we observe the following:

- Some control applications often include parallel – not yet massive – computations too small to be deployed in a GPU due to the kernel preparation and launch costs, but still being performance-critical and amenable for parallelization (e.g., by means of vectorization). For instance, automotive radar applications [21, 61] work on small workloads (e.g., 8x16 and 16x16 matrices), and apply a number of algebraic operations on the data such as the generation of covariance matrices, Hessenberg matrices, P matrices, Eigenvalues, Eigenvectors, QR iterations and the like [19, 37]. Those operations consist of rather simple loops (with one or two nesting levels) intertwined with sequential code and executed within external loops. Hence, such code, generally in the form of matrix-matrix and matrix-vector operations with relatively small matrices and vectors, is highly amenable for vectorization but not friendly for deployment on a GPU.

- Also, autonomous driving frameworks like Apollo [6] use matrix-multiplication-based deep and recurrent neural networks in several stages like object detection, object tracker, etc. Each of those stages has different execution times and works with different input sizes [50].

This section shows that VExt tools and code do not present the qualification and certification issues of that for GPUs while exploiting with low-overhead the parallelism present in some control applications. VExt benefits are summarized in the last column of Table 1.

4.2 Timing analysis

Industrial-quality methods exist based on static and dynamic (i.e., measurement-based) analysis techniques or a combination thereof [65]. Timing analysis approaches have been prevalently deployed on traditional SISD (single instruction single data) models. While moving from the SISD to SIMD may have some implications on the specific analysis technique, we contend the same techniques are effectively applicable to vector operations (M2).

4.2.1 Measurement-Based Timing Analysis

The use of SIMD operations has practically no impact on pure measurement-based timing analysis approaches as these methods operate in a black-box manner, without any assumption on the underlying ISA [65]. Slightly different considerations apply to hybrid methods, where measurements at small granularity (e.g., basic blocks) are combined based on structural information on the program control flow graph (CFG) [18]. For this family of approaches, we maintain that the only issues that may arise are those occurring when vectorization is automatically produced by aggressive compiler optimizations (as in the static timing analysis case). This paper completely excludes this scenario by advocating for the explicit use of vector instructions. In Section 5.2.1 we show how the RapiTime commercial timing analysis tool [16] is able to provide timing analysis results for the vectorized code we have generated.

4.2.2 Static Timing Analysis

In static timing analysis techniques [65], vector operations need to be explicitly considered both in the low-level timing model and high-level hardware modeling phase.

In terms of the low-level timing analysis, vector operations are analogous to non-vector (scalar) operations, in the sense that their low-level hardware latencies are either constant or tightly bounded. As shown in Section 2, the operation of vector instructions in the front-end of the processor (fetch, decode, rename) is the same as scalar instructions, while in the back-end, SIMD vector instructions operate on all data loaded into a vector register in a single operation. That is, if the width of vector operations is 32 bytes (e.g., 8 floats), vector loads fetch 32 bytes at once, and vector add/mult/... operations are applied to the full register (e.g., 8 floats) at the same time (M7).

Also, by defining vector instructions and adding them to the ISA, the compiler or the user can use them to explicitly identify which operations should be performed at the vector unit. The duration of the vector simple and complex, integer and floating-point units operations vary across architectures, but they have a fixed latency or experience limited jitter. The same applies to scalar

functional units. For instance, for the NXP T2080, the VSIU has single-cycle latency for most operations, the VCIU has three-cycle latency, and the VFPU has a four-cycle latency [20]. As another example, in the Arm Cortex-A57 [4] – used in a variety of devices and SoCs such as NVIDIA TX1 and TX2 [39, 40] – instructions such as `VMUL, VSLI, VMAX, VMIN`, etc. take 3-6 cycles. Jittery vector instructions – if any – would bring analogous concerns for timing analysis to their scalar counterparts. For instance, scalar divisions, either integer or floating point, have input-dependent variable latency in several processors, but the same considerations made by the static timing analysis tools for those scalar instructions would apply to their vector counterparts.

When considering higher-level timing modeling, in general, vector instructions do not introduce specific semantics affecting the program instruction and data flows. Similar to scalar load/store operations, the semantics of vector load/store operations require some minor adjustments to the data flow analysis and hardware models of pipelines and caches, for example. Few works have considered the implications between timing analysis and the SIMD model. The only relevant objection is about the use of (auto) vectorization due to aggressive compiler optimizations, compromising the effectiveness of data-flow analysis and the validity of flow-facts [32]. In fact, these objections are avoided by construction in the scenario considered in this work, as we advocate for the explicit use of the ISA vector extensions by the programmer or an automated code generator via *vector intrinsics* (Section 4.3).

Only a small subset of vector operations might exhibit more elaborated semantics than the ones that make them harder to analyze. These instructions are added for performance optimization reasons in mainstream systems. As an example, we find vector *gather* and *scatter* operations that generalize *load/store* operations. The *gather* instruction allows to populate a register with data coming from non-consecutive memory locations, either with a given stride among them or even different strides that can be handled by providing the vector instruction indexes to the data to fetch. The *scatter* operation simply does the opposite, i.e., scattering the contents of a register over the memory. It is similar to store operation, however, it can handle non-contiguous memory accesses. Vector *gather/scatter* are considered complex instructions, and usually, their latency is longer than other vector instructions due to their complexity. While these operations are useful, there is no functionality that cannot be developed without them, i.e., using only simple vector instructions. Avoiding the use of complex vector operations implies using other simpler vector operations instead, which can result in an increase in code size and execution time. On the other hand, however, it simplifies timing analysis. In the evaluation section, we show how representative neural network kernels used for autonomous operation in high-integrity systems can be implemented without using complex vector operations while obtaining significant performance improvements over non-vectorized versions of the same kernels.

Table 2: Arm NEON vector intrinsics and the vector ISA operations they are translated into by the compiler

Vector Intrinsic	Vector Operation	Description
`int32x4_t vaddq_s32 (int32x4_t a, int32x4_t b)`	`ADD Vd.4S,Vn.4S, Vm.4S`	Adds elements in the two source registers, places the results into a vector, and writes the vector to the destination register.
`int16x8_t vsubw_high_s8 (int16x8_t a, int8x16_t b)`	`SSUBW2 Vd.8H,Vn.8H, Vm.16B`	Subtracts each vector element in the lower or upper half of the second source register from the corresponding vector element in the first source register, places the result in a vector, and writes the vector to the destination register.
`int16x8_t vld1q_s16 (int16_t const * ptr)`	`LD1 Vt.8H,[Xn]`	Loads multiple single-element structures from memory and writes the result to one, two, three, or four registers.

4.3 Abstraction and MISRA-C compliance

Implementing a software function or parts thereof with VExt builds on two elements: built-in types and built-in functions (intrinsics). They provide abstractions that simplify the use of VExt and the certification of programs using VExt (M8).

Today, most compilers, including those that are open-source (e.g., GCC and LLVM), provide complete support for vector instructions. Vector intrinsics, or simply intrinsics, are function calls that provide a clear mapping to specific assembly vector instructions. Hence, the compiler can replace them with appropriate vector instructions in the ISA. Therefore, intrinsics provide direct access to the exact vector instructions the user needs. Table 2 shows a few examples (add, subtract and load) of Arm NEON vector intrinsics as well as the vector operations in the Arm64 ISA they translate to. It follows that no runtime support is needed for VExt (M5 and M6). In fact, intrinsics align to the MISRA-C approach to deal with assembly code (M1). In particular, MISRA-C requires that "assembly language shall be encapsulated and isolated" (MISRA C 2012, Directive 4.3).

Intrinsics can be as optimal as implementing code directly in assembly without requiring the programmer to deal with the burden of managing low-level details like register allocation. Overall, vector intrinsics provide the right balance between abstraction – so as to avoid shifting VExt management responsibility to the programmer increasing development and validation costs – and controllability – so as to avoid complex transformations between the source code and the object code or the actual code executed (M8).

Another advantage of VExt in terms of abstraction and reduction of development and validation costs is the existence of widely used libraries for several architectures, which are accelerated with vector instructions (M9). An example of such libraries are the BLAS (Basic Linear Algebra Subprograms)-compatible implementations ATLAS, OpenBLAS, and Intel MKL, to name a few, which im-

```c
void mul_add_baseline
    (const int N,
    const float X[],
    const float Y[],
    float Z[])
{
  int i;
  for(i = 0; i < N; ++i){
    Z[i] += Y[i] * X[i];
  }

}
```

```asm
mul_add_baseline:
 cmp    w0, 0
 ble    .L1
 mov    x4, 0
 .p2align 3
.L3:
 ldr    s0, [x2, x4, lsl 2]
 ldr    s2, [x1, x4, lsl 2]
 ldr    s1, [x3, x4, lsl 2]
 fmadd  s0, s0, s2, s1
 str    s0, [x3, x4, lsl 2]
 add    x4, x4, 1
 cmp    w0, w4
 bgt    .L3
.L1:
 ret
```

(a) Multiply-add Function in C (b) Compiler-generated assembly code

Figure 4: C and assembly code of a simple multiply-addition function.

plement operations between matrices and array vectors and that are frequently used in the development of autonomous driving software.

4.4 Code Coverage

Achieving structural code coverage requirements is a fundamental and recurring concern in the validation of high-integrity real-time systems. Domain-specific standards and regulations set specific coverage goals based on the criticality of the system or application under test [54]. Coverage objectives typically range in between baseline statement coverage and Modified Condition Decision Coverage (MC/DC). The vast majority of existing tools and techniques for code coverage build on widely-acknowledged coverage criteria that apply to sequential programs and do not straightforwardly apply to parallel programming and GPUs programming paradigms in reason of their specific execution models as well as memory model [10, 47]. Although few preliminary approaches for multi-threaded programming have been proposed [26], structural coverage on this class of programs tends to be overly complex and, still, not consolidated.

Conversely, the SIMD paradigm is not encumbered by the complexities of GPU and parallel programming in general. In particular, VExt preserve the single-threaded execution model and, hence, do not jeopardize the applicability of coverage criteria for single-threaded programming (M3). In Section 5.3, we show how a commercial and qualified code coverage tool supports the vectorized versions of our neural network kernels without reporting any compatibility issue.

4.5 Testing and race conditions

Vector extensions generally perform the same operation on multiple data simultaneously and synchronously for computing instructions. That is, they exploit data parallelism but not concurrency, preventing issues arising in parallel pro-

gramming models and GPUs, such as data races (M4) and barrier divergence, to name just a few.

In the case of memory load and store operations, read and write operations respectively are performed serially in a specific order, as if they were independent instructions of a single-threaded program. Therefore, while latencies may vary, the internal order of read and write operations in a vector load or store is preserved, hence avoiding race conditions by construction. As a result, all executions of a vector instruction will always trigger the only possible order among its individual operations: this simplifies testing drastically, since those approaches effectively deployed on sequential code can be used directly on vector code with the same advantages and limitations.

4.6 Scalable VExt and VCOP

Next we assess the different properties presented above for VExt for scalable VExt and VCOP. In terms of timing (Section 4.2), with scalable VExt a given instruction might be translated into multiple (micro) operations depending on the actual hardware implementation (e.g., 8 operations are required for a scalable vector of 1024 bits if the hardware vector pipeline supports 128 bits). From the programmers perspective, this means that the instruction might take longer where the hardware vector pipeline is smaller than the size of the scalable vector. With VCOP latencies can increase due to the off-loading of vector operations to the VCOP and the retrieval of data back to the main processor, yet their execution would be as predictable as regular scalar operations. The key differentiating element is whether VCOP is shared or not among the different cores of the MPSoC. If it is not, the same principles defined for the timing of VExt apply to VCOP. If it is shared, a new whole set of problems arise due to multicore contention [48].

MISRA-C compliance (Section 4.3), code coverage (Section 4.5), and race condition testing (Section 4.5) are not affected by scalable VExt and VCOP, as the abstraction provided to the programmer is the same, where the main difference w.r.t scalable VExt is the hardware implementation. For instance, if a scalable VExt instruction is executed, then all the chunks of the scalable VExt instructions are executed, so code coverage is maintained at the vector instruction as for VExt.

4.7 Illustrative Example

In order to consolidate some of the concepts presented in this section, we illustrate the use of vector intrinsics and the generated vector ISA operations. We focus on Arm NEON SIMD vector intrinsics [5] since those are the VExt implemented in our target platform. We start by showing the implementation of a simple vector (array) multiply-addition function in C in Figure 4 (a), which is similar to the `saxpy` function found in popular mathematical libraries such as BLAS and the generated assembly instructions without VExt using the GCC compiler in Figure 4 (b). In the latter case, we can see that the code does not

use vector instructions since all the used registers are `w0-w4` (32-bit), `x0-x4` (64-bit) and `s0-s2` (32-bit). Line 9 in the C code translates into lines 7-11 in the assembly code where we see the load operations for the elements of arrays X, Y, and Z (lines 7-9), the floating-point multiply-add operation (line 10) and the store to the array Z (line 11). The instructions in lines 7-11 are regular scalar instructions. Assembly lines 12-14 correspond to the increment of the loop counter and its conditional branch.

```
1  void mul_add_vector
2      (const int N,
3      const float X[],
4      const float Y[],
5      float Z[])
6  {
7   int i;
8   for(i = 0; i < N; i+=4) {
9    float32x4_t x = vld1q_f32(&X[i]);
10   float32x4_t y = vld1q_f32(&Y[i]);
11   float32x4_t z = vld1q_f32(&Z[i]);
12   z = vmlaq_f32(z, y, x);
13   vst1q_f32(&Z[i],z);
14  }
15 }
```

(a) C vectorized multiply-add

```
1  mul_add_vector:
2   cmp  w0, 0
3   ble  .L10
4   sub  w0, w0, #1
5   add  x4, x1, 16
6   lsr  w0, w0, 2
7   add  x0, x4, x0, uxtw 4
8   .p2align 3
9  .L12:
10  ldr  q1, [x1], 16
11  ldr  q2, [x2], 16
12  ldr  q0, [x3]
13  cmp  x1, x0
14  fmla v0.4s, v2.4s, v1.4s
15  str  q0, [x3], 16
16  bne  .L12
17 .L10:
18  ret
```

(b) Vector assembly

Figure 5: Vector implementation and compiler-generated assembly code of the multiply-addition function.

Figure 5 (a) shows the same example of multiply-addition function in C but using vector intrinsics. The generated assembly looks very similar with just a few key differences. Assembly lines 4-7 in the initialization are different because now the loop counter increases by 4 each time, instead of 1. This accounts for the fact that vector instructions operate on 4 data elements of type float instead of just 1 float as in the scalar case. Assembly lines 10-12 and 15 are the loads/stores as before, but this time the registers `q0-q2` are vector registers (128-bit long). This is one way vector instructions in the ISA are exposed to the software, namely, the same opcode as scalar instructions but on vector registers. Assembly line 14 with the `fmla` operation performs a vector floating-point multiply-add operation on 128-bit (4 floats) operands. That is because it uses the registers `v0.4s-v2.4s`, which are the same 128-bit long registers as `q0-q2`, but this time are treated as vectors of 4 single-precision elements of 32-bit.

As shown in Figure 4(b) and Figure 5(b), scalar and vector assembly instructions individually, and the complete code block globally, are pretty similar (e.g., `ldr`), and their main differences are in the use of different register type (e.g., floating-point registers for scalar instructions, and vector registers for vector instructions).

Overall, with this example we have illustrated the use of vector intrinsics (e.g., `vld1q_f32`, `vmlaq_f32` and `vst1q_f32`) and how they translate into vector operations, creating a simple mapping between source code and object code. Also, we have shown how scalar and vector code share the same control flow, hence simplifying timing analysis and code coverage.

5 Evaluation

In this section, we assess the execution time reductions achieved by deploying VExt in COTS MPSoC on a set of representative kernels for neural network computation. We also develop OpenMP, OpenMP+VExt, and GPU implementations for those kernels to provide a more solid comparison in terms of performance. Finally, we provide evidence that, unlike for the other FPA, existing commercial and qualified code coverage tools support VExt out of the box.

5.1 Experimental Setup

We conduct our experiments on an NVIDIA AGX Xavier platform, which includes an octa-core Arm v8 CPU [41]. Arm processors are widely used in various domains, from high-performance domains, such as artificial intelligence and automotive, to low-power edge devices such as IoT devices. Table 3 shows the specifications and features of the CPU in the NVIDIA AGX Xavier and its vector unit, which implements NEON VExt. Note that there exist very few implementations of the recent Arm SVE VExt, which we discuss in Section 6, but none of them in embedded processors used in real-time domains.

Table 3: CPU and Vector configurations of the Xavier SoC.

Hardware block	**Feature**	**Configuration**
CPU	Cores	8-core Arm v8
	L1 caches	128KB 4-way icache, 64KB 4way dcache
	L2 caches	2MB 16-way L2 per cluster of 2 cores
	L3 cache	4MB 16-way L3 shared
VExt	Vector Instruction	Arm v8.2 NEON unit
	Vector Length	128 bits
GPU	Micro-Architecture	Volta
	Streaming Multiprocessors (SM)	8
	Warp-width	32 threads
	CUDA Cores	512 (64 per SM)
DRAM		32 GB, 256-bit LPDDR4x, 137 GB/s

We have developed vectorized code for two well-known functions, which are widely used and of relevance in a wide set of high-integrity functions in increased-autonomy products [6, 42]. For each of them, we develop two variants to capture

the variable size and form-factor of some vectorizable control applications as described in Section 4.1.

- *General Matrix Multiplication* (GEMM) is a common algorithm in linear algebra, machine learning, statistics, and many other domains. This function is used not only in a variety of scientific algorithms and functions, but also in deep learning frameworks where the core operations are implemented building on the GEMM operation, as well as in radar applications for computations such as covariance matrices. We evaluate two versions of the GEMM kernel:
  - *GEMMa*, that uses asymmetric matrices as in the convolutional layers of neural networks. We used the following matrix dimensions $M \times N \times K = 64 \times 92000 \times 288$.
  - *GEMMs*, that uses symmetric matrices with dimensions $2048 \times 2048 \times 2048$.
- *YOLO Object Detection* [52]. You Only Look Once (YOLO) is a state-of-the-art, real-time object detection system. YOLO is exposed to representative images during training and test time, so it implicitly encodes contextual information about classes as well as their appearance. YOLO [52] learns generalizable representations of the objects so that, when trained on natural images and tested on artworks, the algorithm outperforms other top detection methods.

  We deploy different versions of YOLO: YOLOl is the default version available with high-resolution images, whereas YOLOs has been created to experiment with smaller footprints and thus, different use of the processor.

  YOLOl uses 608×608 data inputs and all layers of the YOLO's neural network, which packs 106 layers where 75 are convolutional. This results in a total of 7.03×10^{10} multiply-add instructions. Given its size, the computations take relatively long to be performed.

  We also implemented a small version (YOLOs) that uses 416×416 data inputs and a reduced set of 22 layers, where 16 of them are convolutional, which results in 2.78×10^9 multiply-add instructions. These changes lead to faster computations.

5.2 Performance Analysis

We evaluate the performance obtained with VExt comparing it with purely scalar versions of the code. For completeness, and given that OpenMP is currently being considered for future safety-related systems (see Section 3.4), we also evaluate OpenMP implementations, including combinations of VExt and OpenMP since they are orthogonal.

```
void gemm_vector(const int M, const int N, const int K,
        const float ALPHA, const float A[], const int lda,
        const float B[], const int ldb, const float BETA,
        float C[], const int ldc)
{
  int i,j,k;
  for(i = 0; i < M*N; i+=4){
    float32x4_t vec_C = vld1q_f32(&C[i]);
    vec_C = vmulq_n_f32(vec_C,BETA)
    vst1q_f32(&C[i],vec_C);
  }
  for(i = 0; i < M; ++i){
    register int i_a = i*lda, i_c = i*ldc;
    for(k = 0; k < K; k+=4){
      float32x4_t vec_A_alpha = vld1q_f32(&A[i_a+k]);
      vec_A_alpha = vmulq_n_f32(vec_A_alpha,ALPHA);
      register int k_b0 = k*ldb, k_b1 = k_b0+ldb;
      register int k_b2 = k_b1+ldb, k_b3 = k_b2+ldb;
      for(j = 0; j < N; j+=4){
        float32x4_t vec_C = vld1q_f32(&C[i_c+j]);
        float32x4_t vec_B = vld1q_f32(&B[k_b0+j]);
        vec_C = vmlaq_laneq_f32(vec_C,vec_B,vec_A_alpha,0);
        vec_B = vld1q_f32(&B[k_b1+j]);
        vec_C = vmlaq_laneq_f32(vec_C,vec_B,vec_A_alpha,1);
        vec_B = vld1q_f32(&B[k_b2+j]);
        vec_C = vmlaq_laneq_f32(vec_C,vec_B,vec_A_alpha,2);
        vec_B = vld1q_f32(&B[k_b3+j]);
        vec_C = vmlaq_laneq_f32(vec_C,vec_B,vec_A_alpha,3);
        vst1q_f32(&C[i_c+j],vec_C);
} } } }
```

Figure 6: GEMM NEON vectorized code.

5.2.1 NEON Results

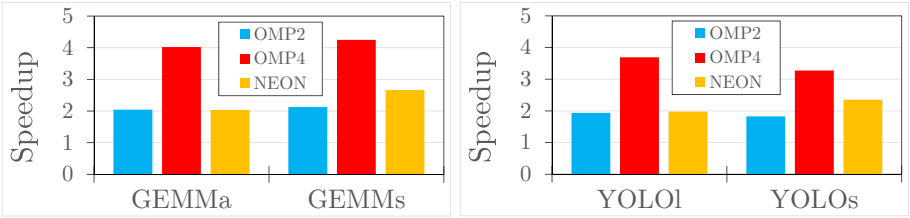
We have used RapiTime commercial timing analysis tool [16], which is qualified for both avionics and automotive use. We focus on the high-water mark execution time. Figure 7 shows the performance results of both VExt and OpenMP for both kernels. Since the maximum speedup we can get with the implementation of VExt in the target processor is 4 (VExt works with 128b registers, which translates into four 32b elements), the OpenMP implementation is executed for 2 (OMP2) and 4 (OMP4) cores for a fair comparison.

GEMM. For GEMMa, Figure 7a, the performance improvement achieved with NEON is 2.04x, therefore, almost equal to OMP2 (2.04x). OMP4 achieves 4.02x performance improvement. GEMMa's performance gains of NEON are limited because the innermost loop of the matrix multiplication (lines 19-30 in Figure 6) iterates over N, which is 92,000 for GEMMa. In this scenario, the data used by the algorithm does not fit in the L1 data cache (64KB): 92,000 elements times 2 matrices times 4 bytes per element results in a footprint of ≈719KB. This causes the benchmark not to fully exploit the potential of the NEON (vector) units. Instead, GEMMs iterates over 2048 elements only for the innermost loop, so it accesses 16KB in total for all matrices, reusing data for some matrices in the L1 cache. Hence, it can deliver higher gains than GEMMa (2.66x instead of 2.04x). GEMMs performance improvement is hence greater than that of OMP2 (2.13x).

YOLO: For YOLOl we see in Figure 7b how NEON achieves slightly better performance results than OMP2 (around 2x). For YOLOs, NEON outperforms OMP2 reaching 2.35x improvements, and gets closer to OMP4, which respectively achieves 3.28x improvements. OMP's performance improvement reduces for YOLOs as OMP's overheads for thread management and synchronization at the beginning and the end of the execution have a relatively higher impact. VExt do not have that type of overheads. Moreover, VExt get higher performance for YOLOs because YOLOs is more CPU bounded as the data fits in the caches. YOLOl is more dominated by cache/memory and less by CPU transactions, hence inducing worse relative performance for VExt.

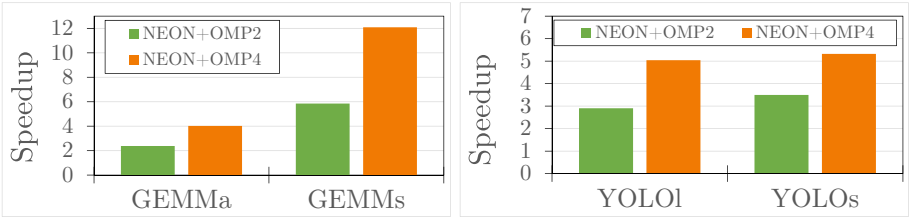
5.2.2 NEON+OMP Results

In the long term, VExt can be combined with other data processing paradigms when these other paradigms find their way to certification. In order to assess the potential benefits, we developed versions of our kernels using OpenMP and vectorization (VExt). For the OpenMP versions, we measure the time manually instead of using a qualified commercial timing analysis tool, as no such tool supports OpenMP (M2 in Section 3.1). The key question we address is whether both data processing paradigms attack the same sources of parallelism, and hence, their combination does not bring additional performance benefits, or



(a) GEMM (b) YOLO

Figure 7: Performance improvements achieved with VExt and OpenMP over a scalar single thread version.



(a) GEMM (b) YOLO

Figure 8: Performance improvements achieved with VExt+OpenMP over a scalar single thread version. Note the different scales in the Y axis in both charts.

instead, their combination effectively results in performance benefits higher than those obtained by each of them individually.

Figure 8 shows the performance improvements of OpenMP+VExt (NEON+OMP). We can see that in all four kernel variants, the benefits obtained by combining VExt (NEON) and OpenMP (OMP) are quite high for 2 and 4 cores reaching around 5x for YOLO and around 4x and 11x for GEMMa and GEMMs, respectively. Comparing this to the results of OMP without VExt, we see a significant improvement from VExt on top of OMP. In the best base (GEMMs) for 2 cores, the improvement goes from 2.1x with OMP2 to 5.9x with OMP2+NEON, and for 4 cores from 4.2x to 12.1x.

These improvements are not only bigger than those of OpenMP and VExt individually, but they are relatively close to the product of their individual performance improvements for each method. In Table 4 the first three result columns show the individual performance improvements. The following two columns present the theoretical improvements that could be achieved by multiplying NEON and OMPx speedups. The final 2 columns report the actual improvement we observed in our experiments. As it can be seen, the predicted improvements for NEON+OMPx are mostly aligned with those observed in our implementation, being GEMMa the only clear exception.

Note that the combined performance of NEON+OMPx is generally a bit

Table 4: Observed and predicted performance benefits

Kernel	OMP2	OMP4	NEON	Multiplicative		Observed	
				NEON+OMP2	NEON+OMP4	NEON+OMP2	NEON+EMP4
GEMMa	2.0	4.0	2.0	4.2	8.2	2.4	4.0
GEMMs	2.1	4.2	2.7	5.7	11.3	5.9	12.1
YOLOl	1.9	3.7	2.0	3.8	7.3	2.9	5.0
YOLOs	1.8	3.3	2.4	4.3	7.7	3.5	5.3

lower than the theoretical multiplicative case. This relates to the fact that data bandwidth requirements to load vectors increases, naturally, by a similar factor, but data does not fit in L1 data caches for the innermost loops of the kernels, so it needs to be fetched normally from the L2 cache. In the case of GEMMs, data fits in L1 for the innermost loop, as explained before, and hence its higher speedup. In the case of GEMMa, data does not fit, and hence its lower speedup. In the case of YOLOl and YOLOs, they combine operations with the innermost loop data fitting in L1 or fitting only in L2, so their speedup is in between that of GEMMa and GEMMs, being slightly higher for YOLOs due to its relatively smaller working sets. L2 cache cannot fully maintain such high bandwidth requested without some stalls in the access ports and queues, which leads to slightly suboptimal performance scalability.

5.2.3 GPU Results

For completeness reasons, we have developed and executed GPU versions of the two variants of both GEMM and YOLO. We executed them in the GPU of the NVIDIA's MPSoC whose main characteristics are listed in Table 3. The GPU has a warp size of 32 threads, which is equivalent to a vector width of 32. It features 8 SMs, each capable of executing 2 warps at the time, which provides a total CUDA core count of 512, ie. it is capable of executing 512 operations per GPU cycle. Figure 9 shows the speedup of GPU over other DPA: VExt executed in a single core using VExt, VExt+OpenMP (executed in 4 cores each with VExt enabled) and the baseline non-parallelized versions of the kernel (single core and no VExt). It is noted that GPU execution times have been captured manually, not via any timing analysis qualified tool (M2 in Section 3.2 and Section 3.3).

- With respect to the baseline, as expected, GPUs provide significant improvements that range from 60x to 257x. For the YOLOs, the overheads to move data from to the GPU reduce the performance benefits of GPUs w.r.t. YOLOl. Likewise, for GEMMa, the use of asymmetric matrices also causes a reduction of the benefits of the GPU w.r.t. GEMMs.
- Compared to VExt (NEON), and focusing on reduced-size kernels (representative of some application scenarios with limited parallel computation

needs) and non-symmetric problems (representative of some control applications), we see that GPU improvements over VExt are still significant 43x and 29x, yet much smaller than those for the baseline.

- When combining VExt and OpenMP (NEON+OpenMP), we see that GPU benefits, while still significant, reduce to 19x and 15x, respectively.

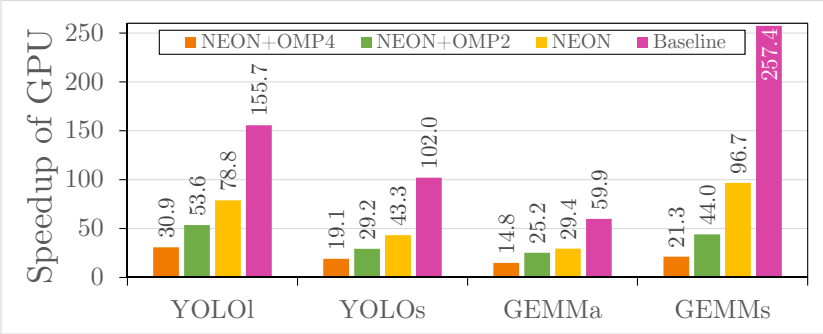


Figure 9: Performance improvements achieved with GPU over other DPA (VExt, and VExt+OpenMP).

These raw performance improvement results confirm the higher degree of parallelism exploitable by GPUs with respect to that VExt in current MPSoCs used in high-integrity systems like the NVIDIA's Xavier [41] in automotive and the Xilinx Zynq UltraScale+ [67] in avionics. Besides, in previous sections we have discussed the difficulties to certify GPU software [62] and qualify the associated tools [34] so that the GPU performance benefits shown can be safely exploited in high-integrity systems. VExt, which we show to address all certification and qualification challenges for their fast adoption in high-integrity systems, and their combination with OpenMP (which has a much shorter horizon to reach certification and qualification goals than GPUs) offer good performance improvements.

In our view, in the long term, VExt+OpenMP will co-exist in MPSoCs integrating multicore vector-enabled processors as well as GPUs; once GPUs can address all certification and qualification challenges. Each of these DPAs will cover different needs for parallelization and performance that future applications will have.

5.3 Code Coverage

We use the RapiCover commercial code coverage tool [15] on the code shown in Figure 6 to assess whether qualified code coverage tools support VExt. RapiCover fulfills tool qualification requirements for domains like avionics (DO-178C/D, DO-330), automotive (ISO 26262) and others. Note that we cannot make any comparison with the code coverage of other DPA as no qualified tool support them (M3 in Sections 3.1, 3.2, and 3.3).

Among the analysis templates offered by RapiCover, we choose "ISO 26262 HR ASIL D", which is the highest certification level provided by the tool for automotive. In Table 5 we show statement coverage and call coverage provided by RapiCover. For each of them we show the following metrics in absolute units: Req (Required coverage), Unk (Unknown, i.e. not instrumented or otherwise addressed), Adr (Missing coverage addressed by justification); Cov (Coverage achieved); and Total (Coverage achieved or addressed).

As it can be seen, all 33 statements (note that in Figure 6, line 6 does not count as a statement because it is only a declaration without definition, lines 13, 17, and 18 are double definitions count as 2 statements each, and the "for" loops in lines 7, 12, 14, and 19 counts as 3 statements each, one for the initialization, one for the condition, and one for the iterator update) in the code where analyzed and hence full statement coverage is achieved.

Regarding call coverage, we see the 15 'function' calls from our `gemm_vector` implementation (they correspond to the functions whose name is highlighted in blue and start with 'v' in Figure 6). These functions are, in turn, the vector intrinsics in the code. Table 5 shows that all 15 functions were called and hence we achieved full code coverage.

Overall, we conclude that VExt add no complexity to standard code coverage practice analysis and are supported by qualified tools out of the box. This emanates from the fact that VExt exploit data-level parallelism and not concurrency, which would cause changes in the control flow and hence, in all metrics related to coverage.

Table 5: RapiCover code coverage for vectorized GEMM in absolute units.

GEMM vector					
Analysis Type	**Req**	**Unk**	**Adr**	**Cov**	**Total**
Statements Coverage	33	0	0	33	33
Call Coverage	15	0	0	15	15

5.4 Discussion

As introduced in Section 4.1, certification challenges brought by other DPA and the characteristics of some application (many control applications include parallel regions that are small enough to be offloaded to a CPU) makes VExt an excellent DPA option. Scalable VExt and VCOP also provide the main properties achieved by VExt and hence bring the same benefits in terms of reduced certification effort. Besides its low-overhead, performance-improving, and minimal certification costs, we see that all chip vendors continue to provide support for and improve VExt.

Vector sizes have been continuously increasing across product generations, hence, increasing the maximum performance improving benefits of VExt. For instance, Intel proposed Advanced Vector Extensions (AVX) to extend SSE by

featuring a widened data path from 128 bits to 256 bits. More recently, Intel has proposed AVX-512, the 512-bit extensions to the 256-bit AVX for the x86 instruction set architecture. Likewise, Arm SVE supports wide vectors of up to 2048 bits and the RISC-V Vector extension up to 2^{16}. Although currently, most Arm processors with SVE have only 128-bit, this can increase in the future, effectively increasing the speedups obtained with SVE. RISC-V CPU implementations (not VCOP) with vector extensions use similar sizes. It is noted that the scalable VExt performance gain over scalar code is not proportional to the vector size but to the width of the vector pipeline. In this line, VCOP offer another good path to increase performance while containing certification costs.

6 Related Works

From the first computer implementing the SIMD concept in the '60s, the Illiac-IV [7], and some follow-up interest in the '70s, vector computers have not proliferated due to their traditional narrow application market (i.e., scientific computation). Later on, SIMD vector extensions widely reached the desktop systems with Intel's MMX extension to the x86 architecture in 1996. This trend kept growing, creating more powerful vector extensions (as mentioned at the end of Section5). For instance, in x86, Intel first introduced SSE, which then was improved to AVX, AVX2, and AVX512. Similarly, ARM started with a first vector extension NEON that expanded with SVE, while RISC-V introduced the scalable Vector Extension and is currently working a packed SIMD extension. Recently, instead, vector processing has been applied at a small scale in the form of vector co-processors and standalone vector accelerators with varying degrees of coupling with the cores [31, 30, 60].

Vector co-processors and accelerators offer a different set of trade-offs than vector extensions. First and foremost, VExt are often part of COTS scalar processors (supported by their ISAs), and are already present in processors used in the high-integrity domain and in multicore processors being evaluated for adoption in high-integrity systems. Conversely, co-processors and accelerators are often optional features of an MPSoC, typically providing richer functionality than VExt, and hence, may not be explicitly supported by the ISA depending on their degree of coupling. The most coupled incarnation of a vector co-processor may be integrated analogously to VExt with minor differences related only to its physical integration in the MPSoC, which may have negligible (if any) and constant additional latencies w.r.t. VExt. Hence, those co-processors may offer the same advantages as VExt. One such case is the Vicuna vector co-processor [49], which supports the RISC-V Vector Extension and has been devised to be particularly time predictable. Other co-processors not particularly devised to be time predictable [22, 33] may need further considerations. However, the main challenge for the use of vector co-processors and accelerators for time-critical applications is the fact that they are often conceived as external components to the core with the aim of sharing them across cores for resource efficiency [8], hence bringing concerns about timing interference across cores. Moreover, if

vector co-processors and accelerators are shared and/or are physically placed far from some cores in the MPSoC (or even in different chips), they may have higher latencies to start and complete computation by not being integrated into the core's pipeline.

Very Long Instruction Word (VLIW) have some commonalities with VExt, since VLIW processors execute instruction bundles all at once, hence processing data in parallel in a predetermined manner. However, unlike VExt, VLIW do not implement the SIMD concept and, instead, operate with heterogeneous instructions on heterogeneous data and are fully statically scheduled. VLIW processors further require the compiler to take full control of instruction scheduling.

The implications of parallel programming on consolidated approaches to software timing analysis have been increasingly considered to sustain the adoption of high-performance computing platforms in performance-eager high-integrity embedded systems [63, 46, 17]. From the perspective of timing analysability, the main focus has been placed on the role of synchronization/communication and critical sections [45, 46] or on the assumption of both a predictable hardware platform and a simple fork-join model to support composability across parallel tasks' timing in isolation [63, 17]. From the opposite perspective, other approaches have been focusing on the injection of a real-time dimension in consolidated (and more complex) parallel programming paradigms [36, 53]. Despite the progress made in the last decade, none of these approaches has been developed into a successful fully-fledged approach for the certification of embedded real-time high-integrity systems.

7 Conclusions

We performed a thorough analysis of the challenges for the use of several data processing approaches (DPAs) in high-integrity embedded domains. We showed that GPU-related DPAs are not yet generally suitable for certified applications. Also, while OpenMP is not yet generally compatible with certification and qualification requirements in high-integrity systems, it is currently assessed in some safety-related domains. We showed how vector extensions (VExt) address all major certification and qualification challenges of the aforementioned DPAs. In particular, vector intrinsics are a mechanism providing a good balance between control and abstraction, achieving the best of both controllability and performance, as required in embedded high-integrity systems. In terms of applicability, VExt provide low-overhead, performance improvements that fit the identified scenarios in which some, yet not massive, parallelism is needed. We provided experimental evidence that VExt are analyzable with commercial qualified out-of-the-box code coverage tool ad timing analysis tools. We also showed the performance benefits of vectorized matrix multiplication and object detection kernels on a Jetson Xavier AGX, and how VExt combines with OpenMP to reach even higher performance improvements. Overall, our analysis shows that VExt provide reasonable performance improvements, which will increase based on current prospects by different chip vendors on the size of vector sizes

in their chips, while offering a straight path for certification, removing the major stumbling blocks found in the road to certification by other DPA based on parallel programming models and GPUs.

Acknowledgements

This work has received funding from the the European Research Council (ERC) grant agreement No. 772773 (SuPerCom) and the Spanish Ministry of Science and Innovation (AEI/10.13039/501100011033) under grants PID2019-107255GB-C21 and IJC2020-045931-I.

References

[1] Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernández, Jaume Abella, and Francisco J. Cazorla. Safety-related challenges and opportunities for gpus in the automotive domain. *IEEE Micro*, 38(6):46–55, 2018.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU scheduling on the NVIDIA TX2: hidden details revealed. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pages 104–115, New York, NY, USA, 2017. IEEE Computer Society.

[3] Tanya Amert, Sergey Voronov, and James H. Anderson. Openvx and real-time certification: The troublesome history. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 312–325, New York, NY, USA, 2019. IEEE.

[4] Arm. Arm - Cortex-A57 Software Optimization Guide. `https://developer.arm.com/documentation/uan0015/b/`, 2020.

[5] Arm. Arm - Neon Intrinsics Reference. `https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics`, 2020.

[6] Baidu. Apollo, an open autonomous driving platform. `http://apollo.auto/`, 2019.

[7] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The ILLIAC IV computer. *IEEE Trans. Computers*, 17(8):746–757, 1968.

[8] Spiridon F. Beldianu and Sotirios G. Ziavras. Multicore-based vector coprocessor sharing for performance and energy gains. *ACM Trans. Embed. Comput. Syst.*, 13(2), sep 2013.

[9] Marc Benito, Matina Maria Trompouki, Leonidas Kosmidis, Juan David Garcia, Sergio Carretero, and Ken Wenger. Comparison of GPU computing

methodologies for safety-critical systems: An avionics case study. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 717–718, New York, NY, USA, 2021. IEEE.

[10] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for GPU kernels. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 113–132, New York, NY, USA, 2012. ACM.

[11] Alejandro J. Calderón, Leonidas Kosmidis, Carlos F. Nicolás, Francisco J. Cazorla, and Peio Onaindia. GMAI: understanding and exploiting the internals of GPU resource allocation in critical systems. *ACM Trans. Embed. Comput. Syst.*, 19(5):34:1–34:23, 2020.

[12] Carlos Hervás García. AI-4-GNC Airbus DS perspectives. In *14th ESA Workshop on Avionics, Data, Control and Software Systems (ADCSS2020)*, pages 1–12, Paris, France, 2020. European Space Agency (ESA).

[13] Roberto Cavicchioli, Nicola Capodieci, Marco Solieri, and Marko Bertogna. Novel methodologies for predictable cpu-to-gpu command offloading. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany*, volume 133 of *LIPIcs*, pages 22:1–22:22, Dagstuhl, Germany, 2019. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[14] Certification Authorities Software Team. *CAST-32A Multi-core Processors*, 2016.

[15] RAPITA Systems. A DANLAW Company. RapiCover. Low-overhead coverage analysis for critical software. `https://www.rapitasystems.com/products/rapicover`, 2019.

[16] RAPITA Systems. A DANLAW Company. RapiTime. In-depth execution time analysis for critical software. `https://www.rapitasystems.com/products/rapitime`, 2019.

[17] Steven Derrien, Isabelle Puaut, Panayiotis Alefragis, Marcus Bednara, Harald Bucher, Clément David, Yann Debray, Umut Durak, Imen Fassi, Christian Ferdinand, Damien Hardy, Angeliki Kritikakou, Gerard K. Rauwerda, Simon Reder, Martin Sicks, Timo Stripf, Kim Sunesen, Timon D. ter Braak, Nikolaos S. Voros, and Jürgen Becker. Wcet-aware parallelization of model-based applications for multi-cores: The ARGO approach. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 286–289, New York, NY, USA, 2017. IEEE.

[18] Boris Dreyer and Christian Hochberger. Non-intrusive online timing analysis of large embedded applications. In Sebastian Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis, WCET 2019, July 9, 2019, Stuttgart, Germany*, volume 72 of *OASICS*, pages 2:1–2:11, Dagstuhl, Germany, 2019. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[19] Alfonso Farina. Introduction to radar signal and data processing: the opportunity. Technical report, Selex Sistemi Integrati Rome (Italy), 2006.

[20] Freescale semicondutor. QorIQ T2080 Reference Manual, 2016. Also supports T2081. Doc. No.: T2080RM. Rev. 3, 11/2016.

[21] Jonah Gamba. *Automotive Radar Applications*, pages 123–142. Springer Singapore, Singapore, 2020.

[22] Yi Ge, Yoshimasa Takebe, Masahiko Toichi, Makoto Mouri, Makiko Ito, Yoshio Hirose, and Hiromasa Takahashi. A vector coprocessor architecture for embedded systems. In *2011 International SoC Design Conference*, pages 195–198, New York, NY, USA, 2011. IEEE.

[23] Mohamed Hassan, Anirudh M. Kaushik, and Hiren D. Patel. Predictable cache coherence for multi-core real-time systems. In Gabriel Parmer, editor, *2017 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburg, PA, USA, April 18-21, 2017*, pages 235–246, New York, NY, USA, 2017. IEEE Computer Society.

[24] Daniel Hellström and Fabrice Cros. Rtems smp final report: Development environment for future leon multi-core. Technical report, European Space Agency (ESA), Paris, France, 2015.

[25] Martin Hofmann, Florian Neukart, and Thomas Bäck. Artificial intelligence and data science in the automotive industry. *CoRR*, abs/1709.01989:1–22, 2017.

[26] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing concurrent programs to achieve high synchronization coverage. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 210–220, New York, NY, USA, 2012. ACM.

[27] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[28] Chris W. Johnson. The increasing risks of risk assessment: On the rise of artificial intelligence and non-determinism in safety-critical systems. In *the 26th Safety-Critical Systems Symposium*, page 15, York, UK, 2018. Safety-Critical Systems Club York, UK., SCSC on Amazon / CreateSpace.

[29] Anirudh M. Kaushik, Paulos Tegegn, Zhuanhao Wu, and Hiren D. Patel. CARP: A data communication mechanism for multi-core mixed-criticality systems. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 419–432, New York, NY, USA, 2019. IEEE.

[30] Mario Kovač, Philippe Notton, Daniel Hofman, and Josip Knezović. How europe is preparing its core solution for exascale machines and a global, sovereign, advanced computing platform. *Mathematical and Computational Applications*, 25(3):1–8, 2020.

[31] Christoforos E. Kozyrakis and David A. Patterson. Overcoming the limitations of conventional vector processors. In Allan Gottlieb and Kai Li, editors, *30th International Symposium on Computer Architecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA*, pages 399–409, New York, NY, USA, 2003. IEEE Computer Society.

[32] Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing flow information for tighter WCET estimation: Application to vectorization. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*, pages 217–226, New York, NY, USA, 2015. IEEE Computer Society.

[33] Yuan Lin, Nadev Baron, Hyunseok Lee, Scott Mahlke, and Trevor Mudge. A programmable vector coprocessor architecture for wireless applications. In *3rd Workshop on Application Specific Processors*, pages 103–110, New York, NY, USA, 2004. ACM.

[34] Matina Maria Trompouki and Leonidas Kosmidis. Do-178c certification of general-purpose gpu software: Review of existing methods and future directions. In *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, pages 1–9, New York, NY, USA, 2021. IEEE.

[35] Adrian Munera, Sara Royuela, Germán Llort, Estanislao Mercadal, Franck Wartel, and Eduardo Quiñones. Experiences on the characterization of parallel applications in embedded systems with extrae/paraver. In José Nelson Amaral, Lizy Kurian John, and Xipeng Shen, editors, *ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17-20, 2020*, pages 53:1–53:11, New York, NY, USA, 2020. ACM.

[36] Adrian Munera, Sara Royuela, and Eduardo Quiñones. Towards a qualifiable openmp framework for embedded systems. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 903–908, New York, NY, USA, 2020. IEEE.

[37] Netlib.org. EISPACK. `http://www.netlib.org/eispack/`, 2021.

[38] Eduardo Qui nones and Franck Wartel. Extrae: an OpenMP-compatible performance monitoring tool for the gr740. In *GR740 User Day (at ESTEC/ESA)*, pages 1–20, Paris, France, 2019. European Space Agency (ESA).

[39] NVIDIA. NVIDIA - Jetson TX1 Module. `https://developer.nvidia.com/embedded/jetson-tx1`, 2016.

[40] NVIDIA. NVIDIA - Jetson TX2 Module. `https://developer.nvidia.com/embedded/jetson-tx2`, 2017.

[41] NVIDIA. Technical Reference Manual. Xavier Series SoC. DP-09253-002. Version 1.1. Technical report, NVIDIA, 2018.

[42] NVIDIA. NVIDIA DRIVE PX. Scalable supercomputer for autonomous driving. `http://www.nvidia.com/object/drive-px.html`, 2021.

[43] Ignacio Sanudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko Bertogna. Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*, pages 213–225, New York, NY, USA, 2020. IEEE.

[44] Nathan Otterness and James H. Anderson. AMD gpus as an alternative to NVIDIA for supporting real-time workloads. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPIcs*, pages 10:1–10:23, Dagstuhl, Germany, 2020. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[45] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET analysis of real-time parallel applications. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, July 9, 2013, Paris, France*, volume 30 of *OASICS*, pages 11–20, Dagstuhl, Germany, 2013. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[46] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Minimizing the cost of synchronisations in the WCET of real-time parallel programs. In Henk Corporaal and Sander Stuijk, editors, *17th International Workshop on Software and Compilers for Embedded Systems, SCOPES '14, Sankt Goar, Germany, June 10-11, 2014*, pages 98–107, New York, NY, USA, 2014. ACM.

[47] Chao Peng. On the correctness of GPU programs. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 443–447, New York, NY, USA, 2019. ACM.

[48] Jon Pérez-Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 53(4):79:1–79:38, 2020.

[49] Michael Platzer and Peter Puschner. Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[50] Roger Pujol, Hamid Tabani, Leonidas Kosmidis, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla. Generating and Exploiting Deep Learning Variants to Increase Heterogeneous Resource Utilization in the NVIDIA Xavier. In Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[51] David Radack, Harold G. Tiedeman, and Paul Parkinson. Civil Certification of Multi-core Processing Systems in Commercial Avionics, 2018.

[52] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767:1–6, 2018.

[53] Sara Royuela, Alejandro Duran, Maria A. Serrano, Eduardo Quiñones, and Xavier Martorell. A functional safety openmp ˆ* for critical real-time embedded systems. In Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller, editors, *Scaling OpenMP for Exascale Performance and Portability - 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20-22, 2017, Proceedings*, volume 10468 of *Lecture Notes in Computer Science*, pages 231–245, New York, NY, USA, 2017. Springer.

[54] RTCA and EUROCAE. *DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification.* RTCA and EUROCAE, 2011.

[55] RTCA and EUROCAE. *RTCA DO-330 - Software Tool Qualification Considerations.* RTCA and EUROCAE, 2011.

[56] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. On how to identify cache coherence: Case of the NXP qoriq T4240. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPIcs*, pages 13:1–13:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[57] Nivedita Sritharan, Anirudh M. Kaushik, Mohamed Hassan, and Hiren D. Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 433–445, New York, NY, USA, 2019. IEEE.

[58] Nivedita Sritharan, Anirudh M. Kaushik, Mohamed Hassan, and Hiren D. Patel. Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 433–445, New York, NY, USA, 2019. IEEE.

[59] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[60] Hideki Sugimoto and Koji Adachi. Vector compliance testing for risc-v. In *RISC-V Global Forum*, pages 1–35, Zurich, Switzerland, September 2020. RISC-V International.

[61] Lee Teschler. The basics of automotive radar, 2019. `https://www.designworldonline.com/the-basics-of-automotive-radar/`.

[62] Matina Maria Trompouki and Leonidas Kosmidis. Brook auto: high-level certification-friendly programming for gpu-powered automotive systems. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 100:1–100:6, New York, NY, USA, 2018. ACM.

[63] Theo Ungerer, Christian Bradatsch, Martin Frieb, Florian Kluge, Jörg Mische, Alexander Stegmeier, Ralf Jahr, Mike Gerdes, Pavel G. Zaykov, Lucie Matusova, Zai Jian Jia Li, Zlatko Petrov, Bert Böddeker, Sebastian Kehr, Hans Regler, Andreas Hugl, Christine Rochange, Haluk Ozaktas, Hugues Cassé, Armelle Bonenfant, Pascal Sainrat, Nick Lay, David George, Ian Broster, Eduardo Quiñones, Milos Panic, Jaume Abella, Carles Hernández, Francisco J. Cazorla, Sascha Uhrig, Mathias Rohde, and Arthur Pyka. Parallelizing industrial hard real-time applications for the parmerasa multicore. *ACM Trans. Embed. Comput. Syst.*, 15(3):53:1–53:27, 2016.

[64] VECTOR. Coffee with Vector: Code Coverage for CUDA Code using VectorCAST/QA. `https://www.vector.com/es/es/eventos/global-de-en/webinar-recordings/2021/coffee-with-vector-code-coverage-for-cuda-code-using-vectorcastqa/`, 2021.

[65] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand,

Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

[66] Michael Wong, Michael Klemm, Alejandro Duran, Tim Mattson, Grant Haab, Bronis R. de Supinski, and Andrey Churbanov. Towards an error model for openmp. In Mitsuhisa Sato, Toshihiro Hanawa, Matthias S. Müller, Barbara M. Chapman, and Bronis R. de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, 6th Internationan Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan, June 14-16, 2010, Proceedings*, volume 6132 of *Lecture Notes in Computer Science*, pages 70–82, New York, NY, USA, 2010. Springer.

[67] Xilinx. Rockwell Collins Uses Zynq UltraScale+ RFSoC Devices in Revolutionizing How Arrays are Produced and Fielded: Powered by Xilinx, 2019.

[68] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. Avoiding pitfalls when using NVIDIA gpus for real-time tasks in autonomous systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPIcs*, pages 20:1–20:21, Dagstuhl, Germany, 2018. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.