DOI 10.1109/DASC52595.2021.9594412

DO-178C Certification of General-Purpose GPU Software: Review of Existing Methods and Future Directions

Matina Maria Trompouki*
*Universitat Politècnica de Catalunya (UPC)

Leonidas Kosmidis†,*
†Barcelona Supercomputing Center (BSC)

***Abstract*—General-Purpose GPU software is considered for use in avionics to satisfy the increased computational requirements of future systems. Therefore, it needs to be certified following the DO-178C guidance as all airborne software. In this work, we review the existing methods in the literature, we analyse their advantages and disadvantages, and we discuss how they can be combined to obtain certification with lower effort and cost. Our focus is restricted on application-level software, under the premise that successful completion of verification of avionics graphics GPU software products has been demonstrated, so their GPU compiler has been considered acceptable for these already DO-178C certified products, or existing qualified GPU compilers exist. Finally, we discuss upcoming solutions for certified general purpose GPU computing.**

I. Introduction

Avionics software has been executed for decades on single-core computing platforms and its certification has been performed with the evolution of the DO-178 series of standards. However, the traditionally used single-core processors cannot satisfy any more the relentless need for higher performance which is required in order to offer advanced safety-related features in avionics. For this reason, the aviation industry explores the use of more complex hardware platforms such as multi-cores and Graphics Processing Units (GPUs).

While multi-cores can provide a moderate increase in the on-board computing capability of modern aircraft and can effectively replace single-core platforms, arguably they cannot enable the use of very demanding algorithms such as the ones required to implement higher degrees of autonomy. For such massively parallel workloads, GPUs are naturally a more appropriate choice. In fact, GPUs have been successfully employed in many different domains to accelerate general purpose computations. For example, the majority of supercomputers in the Top 500 list, relies on GPUs in order to achieve high performance processing of complex scientific computations. Moreover, GPUs are used in both desktop and hand-held devices to speed up the processing of generic computations ranging from physics simulations to Artificial Intelligence (AI) processing tasks. Finally, GPUs are also being adopted in safety critical industries such as the automotive [1] and the space sectors [2], to enable functionalities such as autonomous driving and advanced image processing assisted by AI.

However, the use of multi-core systems in avionics is simpler, thanks to the use of ARINC 653 compliant operating systems, which provide a common time and space partitioning abstraction regardless of whether the underlying platform is single-core or multi-core. Therefore, from the functional point of view, a multi-core avionics system can be programmed with the same single-core programming model employed in legacy systems. As a consequence, the "only" implication in terms of certification comes from the timing issues introduced by sharing hardware resources from the software instances running in parallel on the various CPU cores of the platform. For this reason, DO-178C is directly applicable to certify the functionality of multi-core software, while CAST-32A [3] is used in order to deal with the identification and mitigation of channels of interference in timing.

On the other hand, the use of GPUs in avionics for general purpose computations is more challenging from the certification perspective. Using GPUs in the aforementioned non safety-related or less stringent safety-critical domains relies on the use of programming models such as CUDA or OpenCL which do not meet the restrictions imposed by functional safety standards and coding guidelines for the development of safety-critical software such as MISRA-C [4], as it has been recently reported by [5]. In particular, these widely used general purpose GPU (GPGPU) programming models depend on features such as pointers, dynamic memory allocation and other dynamic features like on-the-fly code compilation.

While GPUs have been considered only recently for general purpose processing in avionics systems, they have been used in aircraft for their primary design purpose – graphics processing – for many years already. In particular, the glass cockpit of modern aircraft is equipped with several primary flight displays, multi-function displays, Heads-Up Displays (HUD) or head-mounted displays (HMD) [6] [7]. For example, in an Airbus A320 cockpit there are 4 LCD displays, in A350 6 very large displays and in A380 10 large displays while aircraft from other manufacturers include similar equipment. In addition, recent models feature also touch screens [8] [9]. These visual instruments are driven by avionics-grade GPUs [10] certified according to CAST-29 [11] certification recommendations.

On the software side, these GPUs work with graphics software stacks – GPU graphics APIs and device drivers – certified by FAA and EASA for the highest criticality level DAL-A according to DO-178C. Currently Khronos OpenGL SC 2.0 [12] graphics standard and its predecessor OpenGL SC 1.0.1 [13] are the only GPU-related software solutions known to be already certified in actual aircraft for visual processing in glass cockpit applications. These two OpenGL subsets are tailored for critical systems by lacking dynamic features which can result in runtime failures. Thanks to their

TABLE I
OVERVIEW OF EXISTING DO-178C CERTIFICATION-READY GENERAL PURPOSE COMPUTING SOLUTIONS

GPGPU Method	Flexibility	Certification Effort	Effort for Analysis with Automated Tools	Manual Code Review Effort	Lines of Code Required	Performance
OpenGL SC 1.0.1	+	+	+	++	++	+
OpenGL SC 2.0	++	++	+	+++	+++	++
Brook Auto / BRASIL	++	++	+	+	+	++

certification credit, known techniques for performing general-purpose computations with graphics can be used to accelerate computing algorithms.

In this paper, we first examine how general purpose computations can be achieved using graphics. Then we review the state-of-the-art options which exist for the implementation of general-purpose computations on GPUs so that they can achieve DO-178C certification building on the aforementioned certified graphics technologies and we discuss their advantages and disadvantages. Next we review the current work-in-progress regarding the next generation certified general-purpose computing solutions on modern GPUs and discuss future directions.

II. Background

A. Graphics APIs and Shading Languages

GPUs are special hardware units designed to accelerate graphics computations. In order to produce a 3D visual output, the scene is described using a set of vertices which are assembled in basic geometric primitives such as points, lines and triangles. Next, they are positioned in the 3-dimensional space using their Cartesian coordinates and carry additional colour information. In addition, the camera position in the 3D space is specified, together with a projection matrix which is used to convert the 3D information to a planar image.

During the *vertex processing* stage, the GPU is performing the required matrix calculations in order to convert the geometric primitives to their 2D projection on screen. Then, the triangles are *rasterized*, that is, converted to discrete pixels. In the next step, *fragment* or *pixel processing*, each pixel is *shaded*, obtaining its colour which is written in the framebuffer in order to be displayed on the screen or saved in an off-screen image for later display. The colour is computed using a combination of complex calculations of fixed or interpolated values from the vertex shading processing and of predefined images (known as *textures*) which can be either sampled at specific coordinates or interpolated.

The aforementioned graphics operations are described using a graphics *Application Programming Interface (API)*. Several graphics APIs exist which offer similar capabilities such as OpenGL and DirectX and very recently Vulkan. OpenGL is a portable, royalty-free family of graphics APIs with open specification defined by the Khronos Group, and has versions spanning desktop, embedded (ES) and safety-critical (SC) systems. DirectX on the other hand is a proprietary graphics API developed by Microsoft, while Vulkan, which we discuss more in the last part of this article, is a low-level software API defined by Khronos for both graphics and compute operations, which enables low-level access to the GPU hardware for maximum control over its execution.

Some GPU API versions feature programmable stages such as the vertex or fragment processing. In this case, programs can be written which will be executed for each vertex or each pixel of these stages, using a *shading* language. Similar to the graphics APIs, shading languages look much alike and have similar capabilities. OpenGL and Vulkan use the GLSL (GL shading language) family of shader languages while DirectX uses HLSL (High-Level shader language).

B. General Purpose Computations using Graphics

In order to perform general purpose computations with graphics APIs, the general purpose processing algorithm has to be converted in graphics terms. Several complex algorithms have been accelerated in this way, during the first years of GPGPU computing in the early 2000s, before the appearance of general purpose GPU languages such as CUDA or OpenCL. During that time, researchers have realised that the parallel computing capabilities of GPUs can outperform CPUs and they managed to leverage their power in this way. A comprehensive collection of the variety of techniques developed over the years can be found in [14].

In its simplest form, this procedure is implemented as follows. The input and output data of the algorithm are mapped to textures, which are represented by 2D arrays. Then the programmer draws a planar geometry parallel to the texture, which covers exactly the desired portion of the texture on which computation has to be performed e.g.. a rectangular region. The parallel computation which is invoked for each output position is described in a fragment shader or a fixed function specified by the graphics API. Finally, the graphics hardware takes care of the execution.

III. Certification-ready General-purpose computing using Graphics

In this Section we survey the 3 certification-ready solutions for general-purpose computations on top of graphics APIs. Table I summarises the characteristics and trade-offs of each method in terms of certification effort and performance, which are further analysed in the description of each solution.

A. OpenGL SC 1.0.1

OpenGL SC 1.0.1 [13] is the first Khronos' graphics standard which focuses on safety critical systems. Being a cut-down version of the OpenGL ES 1.0 specification, it offers only fixed function graphics processing. This means that the

GPU operations which can be performed for each vertex and for each output pixel are predefined and cannot be changed.

In terms of certification, this is very convenient since it only requires to certify the CPU API code according to DO-178C as any other airborne CPU software. Historically, programmable GPUs featured only this fixed functionality, because the functionalities exposed by the OpenGL 1.x API were implemented directly in hardware, and therefore could not be changed.

However, this is not the case in modern GPUs, which are fully programmable both in vertex and fragment processing, as well as in other stages of the graphics pipeline which are not used for general purpose computations and therefore their description is considered outside of the scope of this paper, such as geometry and tessellation. This means that a modern OpenGL SC 1.0.1 driver is restricting access to the programmable features of the GPU, providing only access to the functionality exposed by the OpenGL SC 1.0.1 specification. Internally, this functionality is implemented using software, i.e. vertex and fragment shaders, which have been written and validated by the GPU driver vendor. The fixed functionality provided by each of the API calls can be implemented with minimal shaders, consisting of just a few GPU instructions. Moreover, the outcome of these operations can be easily validated against the well known behaviour of older fixed functionality GPUs which implement these operations in hardware and the official Khronos OpenGL SC 1.0.1 Conformance suite.

Therefore, OpenGL SC 1.0.1 provides the easiest path to certification at the expense of limited functionality, since if some desired operations are not provided by the standard, they cannot be implemented. This reduces significantly the potential for general purpose computations on the GPU. However, there are still a few general purpose algorithms which can be accelerated this way. For example, basic image processing tasks can be accelerated. Other types of algorithms include sorting [15], cryptographic operations such as AES [16] and matrix operations such as matrix multiplication [17].

The biggest limitation in this case is that the OpenGL SC 1.0.1 standard only supports textures with 4 channels (Red, Green, Blue, Alpha) of 8-bit each. In the past this was a true hardware limitation of older GPUs, similar to the fixed function limitation that we mentioned earlier. However, all current GPUs do support at least 32-bit integer and floating point formats, with latest ones supporting even 64-bit ones. In fact, higher precision textures have been added in GPUs before full programmability was introduced, which means that there exist GPGPU techniques with fixed functionality relying on 24 or 32-bit texture support and therefore cannot be used on OpenGL SC 1.0.1. Therefore, the fact that this feature is not exposed by OpenGL SC drivers reduces the potential for general purpose acceleration, but it simplifies further the validation of the fixed functionality, allowing even exhaustive range tests for the GPU operations.

It is worth noting that although in terms of standards compliance, general purpose algorithms implemented in that way are trivial to be checked with existing code analysis tools for conventional avionics software, manual code reviews checking their functionality are more complicated. The reason is that the graphics hardware is used in non-obvious ways in order to achieve the desired functionality. Consequently, extensive documentation of the code is required when this method is used.

B. *OpenGL SC 2.0*

The OpenGL SC 2.0 standard added support for programmable shaders, enabling more advanced processing capabilities. Similar to the OpenGL SC 1.0.1, it is based on the embedded version of the OpenGL standard, in particular on the OpenGL ES 2.0, which introduced programmable shaders in the embedded GPUs. While OpenGL SC 1.0.1 featured some differences with its base standard, OpenGL SC 2.0 is a fully compatible subset of OpenGL ES 2.0. As with its safety-critical predecessor, the OpenGL SC 2.0 specification has removed all dynamic features which complicated certification, mainly the online compilation of vertex and fragment shaders.

In terms of the CPU graphics API, apart from some small differences in the API calls with the OpenGL SC 1.0 version, OpenGL SC 2.0 does not present any particular challenge with respect to certification. The main difference comes from the fact that now GPU code is explicitly provided in a programming language instead of being hidden behind fixed-functionality API calls. Both vertex and fragment shaders are written in the OpenGL ES Shading Language 1.0 [18] (GLSL ES 1.0), which is the same dialect used in OpenGL ES 2.0.

Interestingly, GLSL ES 1.0 is a very certification friendly subset of C. Unlike C, it is a strongly typed language, requiring explicit conversions between different types. Both static and dynamic recursion is disallowed and loop structures are well defined. Support for non-statically determined loop structures such as `while` and `do-while` loops is not mandatory to be supported, while `for` loops can only use a single, non-global loop variable which cannot be changed within the loop body. While array indexing is supported, both for matrices and textures, no pointers exist in the language. Function parameters need to be explicitly annotated as read-only inputs (`in`), output (`out`) and both input and output (`inout`) similar to Ada. GPU programs can have multiple read-only inputs but can have only a single, write-only output. These features make GLSL ES 1.0 very convenient to be checked with code analysis tools for regular safety-critical CPU code. Such checks are performed statically when the shaders are compiled and linked offline. At runtime, the program only needs to load the precompiled and verified GPU binaries.

Out-of-bounds array accesses do not result in memory violations i.e. memory corruption or exceptions, but depending on the texture configuration specified using the host OpenGL SC 2.0 API, may return the value of the first or the last element of the texture dimension (`CLAMP_TO_EDGE`) or simply wrap around over the texture values (`REPEAT`). This is frequently desired behaviour in graphics in order to avoid expensive checks over border conditions, as well as simplifying the written code, but it also acts in favour of safety when OpenGL SC 2.0 is used for general purpose computations.

As mentioned earlier, a vast body of works exists regarding the implementation of a multitude of general purpose algorithms, mainly using programmable shaders and extended precision textures [14]. However, the OpenGL SC 2.0 standard retains the same limitation with OpenGL 1.0.1 (and their ES counterparts), that only 4-component textures are permitted for GPU input and output despite the actual capabilities of the underlying hardware, which means that known GPGPU techniques cannot be directly used. A software solution to overcome this limitation has been developed quite recently, with the work presented in [19]. In that paper, mathematical transformations for conversion from and to any C supported format were introduced, enabling the use of any existing techniques for GPGPU computing over OpenGL SC 2.0.

In terms of flexibility, OpenGL SC 2.0 allows to implement almost any general purpose algorithm, with few limitations. First, the size of the input and the output of the program is limited by the maximum texture dimensions allowed by the GPU hardware. In case there is a need to process larger data sets, this has to be implemented by manually passing the data to and from the GPU in multiples of that size, and possibly process them in steps. Furthermore, since textures are natively represented as 2D arrays, the programmer needs to use manually implemented address translation tricks to convert single dimensional array accesses to 2D and vice versa. Moreover, atomic operations within the GPU shaders are not permitted, although they can be emulated with certain graphics tricks such as the blending feature which is exposed with a fixed-functionality API call. Similarly, the fragment processors which are mainly used for running general purpose computations are not capable of *scatter* i.e. writing in arbitrary positions in the output. However, this functionality can be implemented with address sorting [20] or vertex shaders and the blending functionality in order to handle atomic updates in the same output positions [21]. Also, the support for bitwise operators within OpenGL SC 2.0 shaders is limited, which makes it not suitable for algorithms which rely on operations of this kind.

Compared to OpenGL SC 1.0.1, OpenGL SC 2.0 can achieve not only higher functionality and flexibility for general purpose computing, but also higher performance. As we already mentioned, fixed function operations are usually implemented in software with few instructions. Multiple fixed function passes of an OpenGL SC 1.0.1 applications can be merged in a single fragment shader in OpenGL SC 2.0. This means that for the same computation, less GPU calls are needed, and therefore higher performance can be achieved. In terms of lines of code, although the amount of API calls is reduced in OpenGL SC 2.0, there is additional code written for the vertex and pixel shaders. As a consequence, the certification cost of OpenGL SC 2.0 code is higher compared to OpenGL SC 1.0.1, since now the user is responsible also for the verification of the GPU shaders.

Fortunately, the compliance checking of both CPU API calls and GPU code can be performed easily with automated tools. However, in terms of manual code review, understanding OpenGL SC 2.0 GPGPU code can be more complex than understanding OpenGL SC 1.0.1 GPGPU code. The reason is that not only the graphics hardware is used for non-obvious operations, but also the complexity of the code is increased due to the numeric format conversions and address translations, as well as when scatter and blending functionality are used for the implementation of atomic operations.

C. Brook Auto / BRASIL

The third alternative solution for general purpose computations on DO-178C certified graphics environments is Brook Auto [5] / BRASIL [22]. This solution is an evolution of the Brook GPU language [23], predecessor of CUDA. It defines a safety-critical GPGPU language subset with several recommended features by functional safety standards, allowing the certification of the application code as opposed to CUDA.

Brook Auto [5] shares many common features with GLSL ES 1.0 which is used in OpenGL SC 2.0 shaders. The reason is that Brook Auto generates GLSL ES 1.0 code, taking away the complexity introduced by the numerical conversions and address translations which are required for GPGPU computing on this safety critical API, as explained in the previous subsection. The Brook Auto subset complies with MISRA-C [4]. In addition to the strong typing and absence of recursion which are common with GLSL ES 1.0, Brook Auto enforces all loops to be statically upper bounded. In addition, similar to GLSL 1.0 the output of the kernel needs to be annotated with the `out` keyword. Therefore, Brook Auto can be statically analysed with existing tools used for avionics CPU code.

Brook Auto has an open-source source-to-source compiler implementation called BRASIL [22], which generates GLSL ES 1.0 code for the GPU shaders and OpenGL SC 2.0 code for the CPU API management. Therefore it also hides all the complexity related to the texture management, vertex setup, offline shader compilation and shader loading and setup during execution. As a consequence, Brook Auto resembles significantly the CUDA programming model, in which the programmer can just focus on writing the GPGPU code, invoke it similar to a CPU function call, and only take care of data movements from and to the GPU.

Brook Auto supports multiple compiler back-ends, including the generation of sequential, deterministic CPU code which can be used for debugging, GPU emulation and code coverage analysis at source code level, as well as for functional validation. In addition, it features a multi-core back-end which allows to execute the same code on multi-core architectures, as well as various GPU back-ends, which facilitates the development of the application code regardless of the target platform. A big benefit of this approach is that it allows the migration of the application code in a new target platform from a single Brook Auto / BRASIL code base, as we discuss in the next Section.

The tool qualification of the Brook Auto BRASIL compiler according to the ISO 26262 automotive standard for ASIL

D – the highest assurance level in automotive – has been studied in a recent academic work [22]. Although DO-178C and its Software Tool Qualification supporting document DO-330 [24] define more Tool Qualification Levels than ISO 26262, essentially the required evidence for the tool qualification of a tool such as a compiler used for the development of code of the highest criticality in both standards is very similar. In particular, BRASIL relies on extensive checks of the generated code and allows full source code traceability to facilitate additional manual inspection.

Thanks to its small codebase and compliance with safety-critical standards and language subsets (MISRA C, ISO 26262, OpenGL SC 2), Brook Auto/BRASIL can lower the DO-178C certification cost of avionics GPGPU software. In a recent demonstration with an avionics case study from Airbus Defence and Space, Madrid, Spain on top of an avionics-grade AMD E8860 GPU and a certified OpenGL SC 2.0 driver from CoreAVI, Brook Auto achieved a reduction of an order to magnitude in the amount of code and in the development time compared to a manual OpenGL SC 2.0 implementation [25]. More importantly, this result was achieved without prior knowledge of the language.

In terms of flexibility, the OpenGL SC 2.0 back-end of Brook Auto has identical capabilities and limitations with manually implemented OpenGL SC 2.0 GPGPU code. The only difference is that in its current version, Brook Auto lacks native support for scatter and atomic operations, which are frequently avoided in GPGPU computing since they result in lower performance compared to scatter-to-gather transformation used in GPUs. Despite that, the address sorting technique [20] which we mentioned earlier can be used. Brook+ [26], AMD's offering to compete with CUDA when it first appeared, already provided syntax for native scatter support. As a Brook subset, Brook Auto code is fully compatible with the Brook+ toolchain, so in order to retain this property, we are currently working on the implementation of scatter preserving compatibility with Brook+.

Regarding performance, [5] has shown that Brook Auto / BRASIL generated code can achieve between 50% and 90% of the performance of a small hand written and optimised matrix multiplication benchmark. Further comparison with a more complex avionics case study provided in [25] showed that Brook Auto / BRASIL code can achieve the same real-time performance with the handwritten OpenGL SC 2.0 version, meeting the 60 frames per second refresh rate requirement of an avionics display. Deeper investigation removing the requirement to limit performance to the display refresh late has shown that the performance difference between the two solutions is minimal, and only exhibited under the highly optimised CoreAVI GPU driver, while on the AMD GPU driver the two versions provided almost identical performance.

Overall, from the above comparison we can conclude that Brook Auto / BRASIL is a step ahead compared to OpenGL SC 2.0 in the development of avionics GPGPU software, since not only it can achieve DO-178C certification in the same way that OpenGL SC 2.0, but it can also reduce its cost. This is achieved by automating low-level, error prone tasks and by providing additional verification means. This allows to perform code reviews in smaller amounts of Brook Auto / BRASIL code focusing on the true functionality of the software, while the full generated OpenGL SC 2.0 and GLSL ES 1.0 code is also available for code review as in the case of manual OpenGL SC 2.0 GPGPU application development, and traceable back to Brook Auto / BRASIL code.

IV. Upcoming Certified GPGPU Methods

In the previous Section we examined the existing GPGPU methods which can achieve DO-178C certification in aircraft flying today. However, general-purpose GPU computing is highly desired in certified environments and industry is currently working in the development of new solutions that will bring new capabilities, such as the upcoming Vulkan SC standard by Khronos and other methods building on top of it. Table II summarises the certification and performance trade-offs of these solutions, which we discuss in detail in the following subsections.

A. Vulkan SC

Vulkan SC is a safety-critical subset of Khronos' Vulkan standard which is currently under definition. However, an early preview implementation of Vulkan SC is already available by CoreAVI, known as VkCore. Vulkan SC's low-level hardware access allows fine-grained hardware control beyond the one that is possible at the OpenGL SC 2.0 level, enabling high-performance and new functionalities, both in terms of graphics and compute. When it will be ratified by Khronos, it is going to be the first certified native general-purpose GPU computing solution. This means that general purpose computations will be specified directly by the programmer without the need to be mapped to any obscure graphics-related operations, similar to CUDA, OpenCL or Brook.

As it is the case of other solutions which enable GPU programmability, Vulkan SC consists of two parts: a CPU-oriented API and a GPU programming language. The CPU API allows the configuration of the GPU in order to perform graphics or compute operations. In terms of graphics processing, Vulkan SC can achieve the same functionality as OpenGL SC 1.0.1 or OpenGL SC 2.0, as well as to extend them with arbitrary new features, fully exploiting the hardware capabilities of GPUs. However, due to the low-level nature of Vulkan SC and the fine-grained control that it offers, the same functionality is achieved with many more API calls. Note that this is not necessarily negative. A similar increase has been experienced between OpenGL SC 1.0.1 and OpenGL SC 2.0, since additional capabilities are offered.

In a similar way, Vulkan can enable general purpose compute processing. Again, Vulkan SC can implement the same functionality offered by general purpose GPU languages such as CUDA, OpenCL or Brook. However, the same functionality is offered again with significantly more API calls. In terms of the lines of code required for compute among different solutions, CUDA and Brook require the least amount of code.

TABLE II
OVERVIEW OF UPCOMING GENERAL PURPOSE COMPUTING SOLUTIONS FOR DO-178C CERTIFICATION

GPGPU Method	Flexibility	Certification Effort	Effort for Analysis with Automated Tools	Manual Code Review Effort	Lines of Code Required	Performance
Vulkan SC	+++	+++	+	+++	+++	+++
ComputeCore	+	-	-	-	-	+++
Brook Auto / BRASIL (Vulkan SC)	++	++	+	+	+	++

For each CUDA or Brook API call, OpenCL requires multiple API calls, since it provides finer-grained control [27]. The same relationship exists between OpenCL and Vulkan, each API call of which corresponds to multiple API calls of Vulkan.

Regarding the GPU programming language, Vulkan uses the OpenGL Shading Language (GLSL) similar to OpenGL SC 2.0, both for graphics and compute shaders. The main difference is that Vulkan supports the latest GLSL version which is constantly updated with new features. Each new GLSL version is a superset of the previous one and the programmer can specify the GLSL version required by a given shader, which means that even OpenGL SC 2.0 vertex and fragment shaders written in GLSL ES 1.0 can be used directly with Vulkan. Vulkan GLSL shaders have similar properties with the GLSL ES 1.0 ones analysed in the previous Section, which makes them certification friendly, including the absence of pointers. Therefore they can be analysed by existing tools used for conventional CPU avionics code.

In the case of compute shaders, Vulkan exposes additional features such as bitwise operations, access to on-chip memory shared by multiple GPU threads executing in the same GPU shader core, synchronisation among threads and atomic operations. Moreover, in addition to 2D texture accesses, Vulkan allows accessing GPU memory in a flat, 1D fashion as it is the case in CPU code. This removes the restrictions of read-only inputs and single-output restrictions of OpenGL SC 2.0, however when this feature is used, the additional safety of out-of-bounds accesses provided by the texture hardware is no longer guaranteed. In addition, a unique property of compute shaders (also known as *kernels*) is that the programmer has full control over how many GPU threads will be used for the computation and how they will be organised in groups which will be executed in the same GPU shader core.

As in the case of the existing safety critical GPU standards, Vulkan SC is also going to be compatible with the full Vulkan standard, removing dynamic features which can hinder certification. Among these features, the concept of offline GPU shader compilation similar to OpenGL SC 2.0 is going to be preserved. More concretely, Vulkan GLSL code is compiled to SPIR-V bytecode which is portable among different GPU and driver vendors and is translated offline to the target GPU binary code. When the GPU application starts executing, the GPU binary code is loaded and executed using the corresponding Vulkan SC API calls.

Due to the fact that Vulkan SC is a very low-level API, it offers maximum flexibility in the implementation of any GPU operation available by the GPU hardware and high performance. This comes at the expense of higher complexity and amount of code, mainly at API level, but also at the shader level, since the new functionalities allow writing more complex shader code. Consequently, manually written Vulkan SC graphics code will have increased certification cost compared to OpenGL SC 2.0 graphics code. However, graphics code for avionics Human Machine Interfaces (HMIs) is rarely manually written. Instead, qualified code generators are used such as SCADE's Display or DISTI's GL Studio, which facilitate certification regardless of the graphics solution.

On the other hand, manually written Vulkan SC compute code can have a similar certification cost with GPGPU techniques over OpenGL SC 2.0, because despite the higher number of lines of code, the general purpose functionality is more obvious for manual code reviews. Therefore, compared to other general purpose GPU programming languages, Vulkan SC will require higher development and certification effort considering that the certification cost depends on the code size.

However, Vulkan is not intended to be used directly by the application developers, with very few exceptions in which the application requires low-level access to the GPU hardware features which cannot be exploited in another way. Instead, Vulkan is a building block that can be used for building higher level abstractions. For example, we can see a similar trend in the gaming industry, which was one the main drivers of the Vulkan development, aiming to satisfy the need of gaming developers to have full control over the GPU operations in order to achieve high performance. No game title is developed in Vulkan, despite its portability. Instead, Vulkan is used to build highly optimised game engines, which are subsequently used by application developers. Moreover, GPU driver vendors use Vulkan as a basis for their OpenGL based software stacks, which in the past were directly implemented on top of the hardware. Other companies such as Google, use Vulkan in the ANGLE project as a translation layer for various graphics APIs [28] in order to support multiple platforms for the Chromium browser and the Android operating system.

We expect a similar situation in the avionics industry. In fact, CoreAVI already provides an implementation of their OpenGL SC 1.0.1 and OpenGL SC 2.0 drivers on top of Vulkan SC, which are known as VkCore GL SC 1 and VkCore GL SC 2. Therefore, we are argue that Vulkan SC will be used with higher-level abstractions to lower the certification cost of future avionics GPGPU software. Two solutions in this direction are vendor-provided libraries and high-level GPU languages such as Brook Auto / BRASIL, which we discuss in the following subsections.

B. ComputeCore

A solution which can minimise certification cost of GPGPU avionics code is the use of precompiled, optimized and certified GPU implementations provided by safety critical driver vendors or experts. This is the solution offered currently by CoreAVI with the ComputeCore product, which offers common algorithm implementations such as matrix operations, Fast Fourier Transforms (FFTs) or image filters, executed on top of CoreAVI's Vulkan SC driver. Conceptually, this approach is very similar to the shaders which nowadays implement the fixed functionality of OpenGL SC 1.0.1 within GPU drivers, as we described in the previous Section. However, the big difference in this case is that ComputeCore algorithms are not trivial, but they are computationally intensive algorithms which are commonly used in multiple different applications. Moreover, their use is significantly easier, since they only consist of a GPU accelerated function call, compared to the complex set up of graphics APIs. This reduces the effort required for manual code reviews and automated code analysis to the same level followed for regular avionics CPU code.

In general, the performance of GPGPU software depends on the particular implementation choices taken during code development. The same functionality can be achieved in multiple, different ways, some of which can provide better average performance than others, but can make the execution time non-deterministic, affecting its worst case execution time. For example, we have discussed earlier that modern GPUs are capable of performing scatter operations and atomic operations. However, the same functionality can be achieved at algorithmic level by converting the processing pattern of an algorithm from scatter to gather [20]. The performance and determinism of different implementations can differ vastly [29] [30], so it is up to the programmer to make the best choice. On the other hand, GPGPU libraries such as ComputeCore already incorporate such decisions in their design. Moreover, since the implementation is provided by the GPU driver implementers who have access to much more hardware-related information for the target GPU, they can achieve better performance and determinism. Furthermore, the vendors can provide complete certification evidence for these libraries, as they do also for their GPU driver.

As a consequence, this method minimizes the certification cost at the expense of reduced flexibility, e.g. it provides only a finite number of algorithm implementations. Therefore, if there is a need for a new type of e.g. image filter which is not included in the library, it has to be requested to be implemented by the vendor – provided that it is general enough so that other clients can also benefit from such development or as a part of a special agreement between the client and the library vendor. Alternatively, the new algorithm has to be implemented by the application user, as a regular GPGPU algorithm, similar to the rest of their CPU application code. However, in that case the end user will be responsible for the certification of GPGPU code.

C. Brook Auto / BRASIL Vulkan SC back-end

Another orthogonal solution which enables full programmability is the use of a higher-level GPU programming language like Brook Auto / BRASIL, which we introduced in the previous Section and can reduce the certification cost of GPGPU code. Thanks to the retargetable nature of the Brook Auto / BRASIL source-to-source compiler, Vulkan SC code can be easily generated for the verbose API calls, as well as GLSL code for the GPU kernels. In this way, GPGPU programmability can be achieved, leveraging the performance benefits and full control offered by Vulkan SC compared to the existing OpenGL SC 2.0 back-end. At the same time, BRASIL's tool qualification methodology can be reused for the new Vulkan SC back-end, as well as its traceability to source code.

Currently we are working in the development of a Vulkan SC back-end within the BRASIL compiler. Similar to the OpenGL SC 2.0 back-end, we preserve compatibility with AMD's Brook+ [26] Brook dialect for the kernels, in order to allow access to the new features of GPU shaders which are exposed with the use of Vulkan SC. In particular, from the Brook Auto syntax point of view, the scatter implementation is identical to the one we are implementing on the OpenGL SC 2.0 back-end, however its implementation is more straightforward on top of Vulkan SC. In addition, shared memory access, bit-wise operation support and the organisation of the GPU threads in groups are also included, as well as the possibility to access regular arrays in addition to textures. In that case, both read and write access to an input array are supported and multiple outputs from a single GPU kernel. For read-only inputs and write-only outputs we also provide the possibility to use the texture functionality as in the case of OpenGL SC 2.0, in order to leverage the additional memory safety protection offered by the GPU texture hardware. Unlike manual Vulkan implementation, using this feature in Brook Auto / BRASIL is completely transparent to the programmer.

The only functionality not present in the original Brook+ specification which is exposed by Vulkan SC is the one of atomic operations. For this feature we selected to implement the same syntax with GLSL used by Vulkan SC, in order to ease the transition of Vulkan SC programmers to Brook Auto.

Apart from the reduction of the amount of source code required for GPGPU development, the second most important benefit of Brook Auto / BRASIL compared to the other methods is code reuse. Brook Auto application code developed for the OpenGL SC 2.0 back-end can be executed without changes on the Vulkan SC back-end. This can reduce significantly the re-certification cost of an application on top of a new platform. Manual code review of the original Brook Auto code does not need to be repeated, while the certification artefacts used for the OpenGL SC 2.0 version can be reused. Manual code inspection only needs to be performed on the generated Vulkan SC code, which is traceable to the Brook Auto version. This is also true for any other GPU or other accelerator technology which might appear in the future and can obtain a Brook

Auto / BRASIL back-end, including multi-cores. Brook Auto / BRASIL will be able to provide an easy way to migrate legacy GPGPU code to new platforms, while retaining low cost of re-certification. Therefore, it is a good choice for long term investment in this programming method.

Obviously, Brook Auto code using features only available on the Vulkan SC back-end (e.g. shared memory) is not portable to an older OpenGL SC 2.0 target, however this is not expected to happen, since migration happens only towards newer hardware and software platforms.

On the other hand, new developments in Brook Auto / BRASIL using the Vulkan SC back-end have comparable certification effort with the Brook Auto / BRASIL OpenGL SC 2.0 back-end. The reason is that in terms of manual code reviews of Brook Auto code, the same amount of code is used regardless of the back-end used, while the generated Vulkan SC code even though is larger than the generated OpenGL SC 2.0 code, it is more straightforward regarding its functionality. Moreover, in terms of code compliance checks with automated tools, the amount of effort is low in both back-ends, either for the Brook Auto or the generated code, and it's further facilitated by the certification friendly features of Brook Auto and its implementation.

Since this is still work in progress, we do not have any performance numbers yet, but we expect higher performance than the one obtained on the OpenGL SC 2.0 back-end, thanks to the opportunities for lower level access in the GPU features offered by Vulkan SC. Moreover, based on the overhead analysis of our Brook Auto / BRASIL OpenGL SC 2.0 back-end presented in [25], under extreme conditions we expect a similar performance difference as the one observed with the highly optimised CoreAVI OpenGL SC 2.0 driver compared to a manually written Vulkan SC application. However, a comparison between the amount of code required for the implementation of the same avionics application in Vulkan and in Brook Auto [31] shows that Vulkan required 25$\times$ more lines of code for the same functionality and 12 times more development time. Therefore, a slight loss of performance compared to the handwritten Vulkan SC implementation can be acceptable given the additional benefits provided by Brook Auto / BRASIL.

D. Discussion

We believe that all three upcoming solutions for certified GPGPU software are complementary and orthogonal between them, and we foresee all of them used for different cases. The ComputeCore library approach will be the first one to be adopted, since it lowers significantly the bar to entry in terms of GPGPU code certification and provides high performance. With the demand for general purpose computations on avionics GPUs to be increased in the following years, we expect the need for more complex functionality to be implemented, requiring more flexibility. In that case, the end users will need to develop their own GPGPU applications.

Brook Auto / BRASIL and possibly other certification-friendly high-level languages which may appear, will be preferred by most application developers of safety critical GPGPU software, since they provide a familiar and productive GPGPU programming model similar to the widely used CUDA and OpenCL. At the same time they provide a significant reduction in the certification cost of the GPGPU code according to DO-178C thanks its certification friendly features, reduction of the amount of manually written code and support for tool qualification.

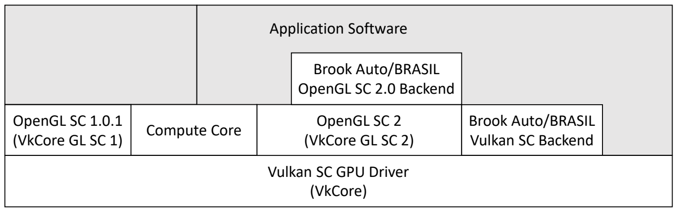
Application Software
Brook Auto/BRASIL OpenGL SC 2.0 Backend
OpenGL SC 1.0.1 (VkCore GL SC 1)
Compute Core
OpenGL SC 2 (VkCore GL SC 2)
Brook Auto/BRASIL Vulkan SC Backend
Vulkan SC GPU Driver (VkCore)

Fig. 1. Overview and possible combinations of upcoming certified GPGPU solutions. Note that the use of OpenGL SC 1.0.1 and OpenGL SC 2.0 is mutually exclusive.

On the other hand, expert safety-critical application developers with specific performance or other needs who require explicit control over the GPU use, will select to program directly in Vulkan SC. These will be users who can afford this and will be committed to invest significantly in this new technology.

Finally, we consider also the possibility for combined solutions as shown in Figure 1. For example, it won't be uncommon to use ComputeCore for the GPGPU for a universally used algorithm such as matrix multiplication, but implement also other GPGPU operations which are not available in ComputeCore using Vulkan SC or Brook Auto / BRASIL's Vulkan SC back-end. In a similar way, if an application uses both graphics and compute, it might use ComputeCore or Brook Auto for the GPGPU part of the application and Vulkan SC or OpenGL SC 2.0 implementation on top of Vulkan SC for the graphics part.

V. Conclusion

In this paper we described the existing state-of-the-art graphics-based GPGPU methods, which can achieve DO-178C certification today. These include the implementation of general purpose computations on top of Khronos' safety critical GPU APIs OpenGL SC 1.0.1 and OpenGL SC 2.0, as well as the certification-friendly CUDA-like GPGPU language Brook Auto and its qualifiable source-to-source BRASIL compiler, which generates OpenGL SC 2.0 code. We discussed their trade-offs in terms of performance, flexibility, productivity and certification effort, identifying Brook Auto / BRASIL as the best of the two worlds.

In addition, we discussed the new generation GPGPU solutions for safety-critical systems which are currently under definition, their relation with the existing certified approaches and their trade-offs. These solutions are mainly centred around Khronos' upcoming safety-critical graphics and compute API, Vulkan SC, and two complementary and orthogonal solutions on top of it. A library of precompiled and certified common algorithms implementation such as CoreAVI's ComputeCore and a Vulkan SC back-end for Brook Auto / BRASIL. ComputeCore will provide the fastest and easier path towards the certification of Vulkan SC based GPGPU computing, whereas Brook Auto / BRASIL or similar technologies will facilitate custom GPGPU developments and will reduce the certification cost of such a low-level GPU API. In special cases where full GPU control is required and the additional effort can be managed, Vulkan SC will be used directly. Finally, the possibility of using a combination of these solutions is also very probable, since the end user can mix and match them based on their needs and their trade-offs.

Acknowledgments

This work was performed within the Airbus TANIA-GPU Project ADS (E/200) in collaboration with the project partners Airbus Defence and Space, Madrid, Spain and CoreAVI, Canada. It was also partially supported by the European Space Agency (ESA) through the GPU4S (GPU for Space) activity, the Spanish Ministry of Economy and Competitiveness under grants PID2019-107255GB and FJCI-2017-34095 (Spanish State Research Agency / http://dx.doi.org/10.13039/501100011033) and the HiPEAC Network of Excellence.

References

[1] S. Alcaide, L. Kosmidis, H. Tabani, C. Hernandez, J. Abella, and F. J. Cazorla, "Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain," *IEEE Micro*, vol. 38, no. 6, pp. 46–55, 2018.

[2] L. Kosmidis, I. Rodriguez, A. Jover-Alvarez, S. Alcaide, J. Lachaize, O. Notebaert, A. Certain, and D. Steenari, "GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021.

[3] Certification Authorities Software Team (CAST), "Multi-core Processors," November 2016, https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/media/cast-32A.pdf.

[4] Motor Industry Software Reliability Association, *MISRA-C:2012. Guidelines for the Use of the C Language in Critical Systems*, 2013.

[5] M. M. Trompouki and L. Kosmidis, "Brook Auto: High-level Certification-friendly Programming for GPU-powered Automotive Systems," in *Design Automation Conference (DAC)*, 2018.

[6] J. L. Tchon and T. J. Barnidge, "Review of the evolution of display technologies for next-generation aircraft," in *Display Technologies and Applications for Defense, Security, and Avionics IX; and Head- and Helmet-Mounted Displays XX*, vol. 9470, International Society for Optics and Photonics. SPIE, 2015, pp. 85 – 93.

[7] J. M. Ernst, L. Ebrecht, and S. Schmerwitz, "Virtual cockpit instruments displayed on head-worn displays – capabilities for future cockpit design," in *38th Digital Avionics Systems Conference (DASC)*, 2019.

[8] Airbus. (2019) Airbus begins deliveries of first A350 XWBs with touchscreen cockpit displays option to customers. [Online]. Available: https://www.airbus.com/newsroom/press-releases/en/2019/12/airbus-begins-deliveries-of-first-a350s-with-touchscreen-cockpit-displays-option-to-customers.html

[9] Boeing. (2016) Touchscreens come to 777x flight deck. [Online]. Available: https://www.boeing.com/features/2016/07/777x-touchscreen-07-16.page

[10] D. Joncas, "COTS GPU Selection Considerations for Mil-Aero Electronics," *Engineers' Guide to Military & Aerospace*, pp. 36–39, 2012.

[11] CAST-29, *Use of COTS Graphical Processors (CGP) in Airborne Display Systems*. Certification Authorities Software Team (CAST), 1997.

[12] Khronos Group, *OpenGL SC 2.0.0 (Full Specification)*, 2016.

[13] ——, *OpenGL SC Safety-Critical Profile Specification, Version 1.0.1*, 2009.

[14] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Eurographics 2005*.

[15] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 2005, pp. 611–622.

[16] D. L. Cook, J. Ioannidis, A. Keromytis, and J. Luck, "CryptoGraphics: Secret Key Cryptography Using Graphics Cards," in *RSA Conference, Cryptographer's Track (CT-RSA)*, 2005, pp. 334–350.

[17] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC)*, 2001, p. 55.

[18] Khronos Group, *The OpenGL ES Shading Language V1.0*, 2009.

[19] M. M. Trompouki and L. Kosmidis, "Towards General Purpose Computations on Low-end Mobile GPUs," in *2016 Conference on Design, Automation & Test in Europe (DATE)*, 2016.

[20] I. Buck, "Taking the plunge into GPU computing," in *GPU Gems 2*. Addison Wesley, 2005, pp. 509–519.

[21] T. Scheuermann and J. Hensley, "Efficient Histogram Generation Using Scattering on GPUs," in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D)*, 2007, pp. 33–37.

[22] M. M. Trompouki and L. Kosmidis, "BRASIL: A High-Integrity GPGPU Toolchain for Automotive Systems," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 660–663.

[23] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777–786, 2004.

[24] RTCA and EUROCAE, *DO-330 / ED-215, Software Tool Qualification Considerations*, 2011.

[25] M. Benito, M. M. Trompouki, L. Kosmidis, J. D. Garcia, S. Carretero, and K. Wenger, "Comparison of GPU Computing Methodologies for Safety-Critical Systems: An Avionics Case Study," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2021.

[26] AMD, "AMD Brook+ Subversion Repository," 2009, https://sourceforge.net/projects/brookplus/.

[27] I. Rodriguez, L. Kosmidis, J. Lachaize, O. Notebaert, and D. Steenari, "GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing," Universitat Politecnica de Catalunya, Tech. Rep. UPC-DAC-RR-CAP-2019-1, https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html.

[28] J. Madill, "ANGLE: OpenGL on Vulkan," in *Vulkan Developer Day*, Montréal, Canada, 2018.

[29] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient Gather and Scatter Operations on Graphics Processors," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*, 2007.

[30] J. Gómez-Luna, J. González-Linares, J. Benavides, and N. Guil, "An optimized approach to histogram computation on GPU," *Machine Vision and Applications*, vol. 24, pp. 899–908, 07 2013.

[31] M. Benito, M. M. Trompouki, L. Kosmidis, J. D. Garcia, S. Carretero, and K. Wenger, "Evaluation of Graphics-based General Purpose Computation Solutions for Safety Critical Systems: An Avionics Case Study," *Poster presented at the Conference on High-Performance Graphics (HPG)*, 2020. [Online]. Available: https://www.highperformancegraphics.org/posters20/01_benito_SCS.pdf