On the MC/DC Code Coverage of Vulkan SC GPU Code

Jaime Luis Martin Aleman* Antonio Agenjo* Sergio Carretero* Leonidas Kosmidis†,‡
*Airbus Defence and Space (ADS) †Universitat Politècnica de Catalunya (UPC) ‡Barcelona Supercomputing Center (BSC)

***Abstract*—Next generation avionics systems require high performance, which can be provided by graphics processing units (GPUs). The newly introduced API Vulkan SC, enables the development of safety critical GPU software with complex control flow, whose certification is subject to DO-178C certifiability objectives, such as MC/DC code coverage.**

In this paper we explain for the first time how MC/DC coverage can be applied in Vulkan SC code as well as the type of potential development errors which can arise in GPU programming. We show how GPU code can be converted in equivalent sequential CPU code and how both versions can achieve 100% MC/DC code coverage.

***Index Terms*—code coverage, MC/DC, graphics processing unit (GPU), Vulkan SC**

I. Introduction

Graphics Processing Units (GPUs) are ideal candidates for accelerating demanding general purpose computations to enable new features in upcoming aircraft [1], but first, a number of challenges must be overcome. This is because all airborne software, including the GPU one, must comply with the corresponding airworthiness safety requirements, therefore it must be developed according to the certification basis agreed with the competent authority.

Even when there is applicable regulation for the development of critical airborne systems (such as the DO-178C [2] for software, the DO-254 [3] for complex hardware, and even for embedded applications in parallel processing architectures, like EASA AMC 20-193 for multicores [4]), the only existing airborne regulation covering graphics processing units is related to their usage in display systems [5], but currently no certification basis exists for using GPUs in general purpose airborne software applications.

Certifiability objectives considered in the DO-178C standard could be applicable to GPUs [6], but, so far, compliance with these objectives has only been demonstrated for CPUs. When these objectives are analyzed for completion in GPU-based airborne software applications, test coverage of software structure, up to MC/DC, arises as a challenge, due to the lack of qualified tools providing the capabilities to perform this analysis and the need to define specific design strategies.

Very recently, in March 2022, the next generation GPU programming API targeting safety-critical domains, Vulkan SC [7], has been ratified, including its GPU programming languages GLSL and SPIR-V, for source code and byte-code programming respectively.

Although previous solutions based on safety-critical graphics APIs such as OpenGL SC 1.0.1 [8] and 2.0 [9] limited control flow structures in GPU code, Vulkan SCs GLSL allows arbitrary complex control flow in GPU software, similar to CPU code, allowing much more general purpose algorithms to be accelerated [6]. For this reason, MC/DC code coverage is required to be performed in the GLSL code. Moreover, unlike previous safety-critical APIs [6], Vulkan SCs GLSL permits multiple outputs per GPU thread and synchronization among the threads. This means that race conditions can be introduced when threads are writing in the same position, or stale data can be read in the absence of missing synchronization between reading and writing operations of the same or different threads. Finally, incorrect use of synchronization may result in problems at program execution including deadlocks.

In the following sections of this paper we explain in more detail these challenges and how they could be addressed by tool providers, using representative code examples.

It is worth noting that MC/DC coverage is barely addressed in the literature beyond single threaded code, not to mention the massively parallel execution model of GPUs. Therefore, since it is still an open problem, we do not intend to provide a definitive solution. The goal of our paper is to initiate and motivate future works in this area. Finally, we explain the potential data access and synchronization issues that can arise by the incorrect use of the synchronization in GLSL and we argue that MC/DC testing and static analysis methods could prevent the incorrect use of the aforementioned features.

To the best of our knowledge, this is the first work discussing this topic, not only for Vulkan SCs GLSL, but also for other GPU languages and we believe that our insights are very useful for future works in this direction.

The rest of the paper is organised as follows: Section II provides the necessary background on GPUs and code coverage. Section III presents our view on MC/DC code for both CPU and GPU parts of Vulkan SC code with concrete examples. Finally, Section IV summarises the conclusions of our paper.

II. Background

A. The GPU Programming Model

GPUs are accelerators, thus they are not standalone processors, but they need to be connected to a system with at least one CPU (central processing unit). GPUs can be either *discrete* or *integrated*. Discrete GPUs have their own DRAM memory which is distinct from the one of the host processor,

 DOI 10.1109/DASC55683.2022.9925766

and they are usually connected to the host system through a high performance bus system, such as AGP (Accelerated Graphics Port) in legacy systems or PCIe (Peripheral Component Interconnect Express) in modern computing systems. Integrated GPUs are implemented in the same die with the host processor and therefore share the same DRAM memory available to the system. Some manufacturers like AMD, refer to this combination of CPU and GPU as APU (Advanced Processing Unit).

Regardless of whether a GPU is integrated or discrete, and therefore whether it is sharing the same physical DRAM with the system CPU, the default programmer view includes two separate address spaces, one for the CPU and one for the GPU. Therefore, the programmer is responsible of allocating memory and explicitly transferring data between these two address spaces in order to program the GPU. This is achieved via various GPU programming APIs (eg. OpenGL, CUDA, OpenCL, Brook, Vulkan or their safety critical variants OpenGL SC [9], Brook Auto [10], Vulkan SC [7]), which all offer the same programmer view. It is worth noting that some of these APIs offer the possibility to use a single memory allocation in integrated GPUs, and therefore sharing the same physical memory and avoiding memory copies. However, for the rest of this paper we focus on the default case, which is shown in Figure 1 and we consider the Vulkan SC programming API.

Moreover, due to the accelerator programming model of GPUs, the programmer needs to provide two different types of code, one for the host (CPU) and one for the GPU (accelerator). First, there is the host code which is executed on the CPU and is responsible for the aforementioned allocations and transfers between the CPU and GPU. In addition, this code is used to initiate the offloading of a computation to the GPU, which is known as *kernel launch*. Depending on the GPU API, e.g. in Vulkan SC which is the focus of our paper, the kernel launch API call requires to specify the number of threads (also known as *work-items* or *invocations*) which will execute the computation and how they are organised in a *grid* (also known as *NDRange*) of threads. The grid can have up to 3 dimensions (x, y, z) and is composed by *work-groups*. In turn, each work-group can have also up to 3 dimensions (x, y, z) and consists of a group of threads. Finally, the CPU code can synchronise the execution between the CPU and the GPU, by waiting for the completion of the GPU operations such as the memory transfers and GPU computations, using barriers.

The GPU code is written in another language and it is organised in the form of functions known as *kernels*, which can be offloaded to the GPU. The kernel code is written following the SIMT (Single Instruction, Multiple Threads) approach, that is describing the computation to be performed by each thread of the GPU. In the particular case of Vulkan SC, the GPU kernels are written using a version of GLSL (OpenGL shading language), which is compiled to an intermediate representation called SPIR-V (Standard Portable Intermediate Representation). SPIR-V is finally converted to GPU executable code and can be loaded to the GPU using the Vulkan SC API.

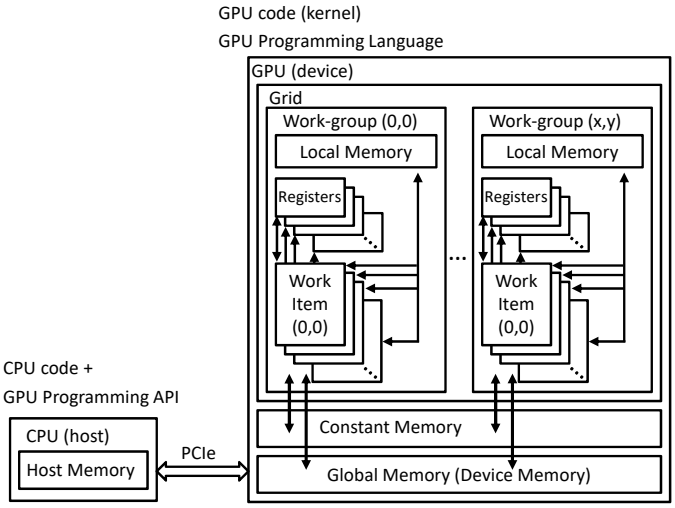

Fig. 1. Overview of the generic GPU programming model and the default programmer's view of the GPU program.

GPU programming languages, including GLSL, provide the programmer with multiple memory spaces within the GPU, as shown in the right part of Figure 1. In particular, the local variables of each kernel are private to each thread and they are allocated to GPU registers, while array thread memory operations access by default the device Global Memory, which resides in the DRAM and it is visible to all threads. Moreover, GPUs feature another type of fast, read-only memory, *Constant Memory*, which is also visible to all threads.

Each GPU consists of multiple *compute units (CUs)*, which are equivalent to the CPU cores in multi-cores. However, their main difference is that each CU supports the execution of a high number of threads which is implementation defined, and can reach up to thousands of threads.

CUs are executing threads which belong to the same *work-group* and which can potentially communicate through a fast on-chip memory called *Local Memory* and synchronise their execution using work-group barrier instructions. Local Memory is also known with different names depending on the GPU manufacturer, e.g. AMD uses the term *Local Data Store (LDS)* while the term *shared memory* is used by NVIDIA and other GPU vendors.

In the SIMT programming model, each GPU thread can query its thread identifier, either its local one i.e. within the work-group or the global one, i.e. in the grid, as well as the sizes of each dimension of the work-group and the grid. Local identifiers always range between 0 and the size of each dimension of the work-group, while global identifiers range between 0 and the total number of threads in each grid dimension. Usually, these identifiers are used in order to specify which memory positions each thread will access in the different GPU memory spaces, as well as to differentiate the execution of each thread if it is required by an algorithm.

CUs are scheduling work-items in groups of threads which are executed in lockstep and are called *wavefronts*. All threads within a wavefront, which usually consists of 64 threads in

AMD GPUs, follow exactly the same control flow. If during the execution of a control flow construct (e.g. `if-else`), the boolean condition of every thread within the wavefront does not evaluate in the same value, the execution of the work-group is serialised. This is known as *control divergence*. First the threads whose condition evaluate as true (and therefore take the `if` path) are executed, while the rest of the threads in the wavefront are disabled. When the execution of the wavefront reaches the `else` path, the condition is inverted and the threads of the `if` path are disabled, until the control flow of all threads of the work-group is joined again.

Thread divergence is only affecting the performance of the algorithm on a GPU since it reduces the computational efficiency of the GPU execution. However, from the programmer point of view, the functionality of each GPU thread is the same as if it was executed by a separate core or processor.

B. Code Coverage

Structural code coverage is one of the code testing methods which is required by DO-178C, the current certification standard in airborne software. The purpose of code coverage is to ensure that software testing has covered extensively complex software, such that potential hidden bugs are revealed.

Avionics code is primarily tested using a requirement-based testing method, in which the test cases which correspond to software requirements are tested, and the parts of the software which are exercised during this procedure are collected. In this way, it can be demonstrated that the software complies with its requirements. In addition to this, structural code coverage ensures that the requirements-based testing has exercised the software up to the level required by the applicable criteria depending on the software's DAL (Design Assurance Level).

Code coverage can be applied either at source code, object code or executable object code. However, DAL-A software requires that traceability to the source code is analysed in order to check that, if the compiler toolchain introduces any structure which is not directly traceable to the source code statements, the code coverage needs to be verified with additional means.

Depending on the DAL, different types of structural code coverage methods are required, as summarised in Table I. For the top 3 ones, each one is a subset of the following ones and therefore can be satisfied if the most strict coverage method is achieved. *Statement coverage* needs to verify whether all source code statements have been exercised at least once. *Decision coverage*, requires to test each conditional path in the software (i.e. each entry and exit point, as well as that every decision has taken all its possible outcomes at least once). *Modified Condition/ Decision Coverage (MC/DC)*, which is the main focus of this paper, requires in addition to decision coverage that every condition has also taken all possible outcomes at least once and that each condition in a decision has been shown to affect the decision outcome independently. This can be achieved either by varying a single condition while the rest of the conditions remain fixed, or at least only the conditions which affect the decision outcome remain fixed. Finally, DO-178C defines also another type of structural code coverage method related to data and control coupling. This method requires to cover the control flow and data dependence of a software module on other software modules.

TABLE I
APPLICABILITY OF STRUCTURAL CODE COVERAGE ACCORDING TO THE SOFTWARE'S DAL. ++ MEANS THAT IT HAS TO BE ACHIEVED WITH INDEPENDENCE, + THAT IT NEEDS TO BE ACHIEVED, WHILE BLANK MEANS THAT IT IS UP TO THE END USER TO DECIDE.

Structural Code Coverage Method	DAL-A	DAL-B	DAL-C	DAL-D
Statement	++	++	+	
Decision	++	++		
MC/DC	++			
Data and control coupling	++	++	+	

MC/DC is a well studied field in the avionics literature. Prior studies have analysed the effectiveness of MC/DC under various implementation schemes, showing that when inlining is enabled its bug revealing capability is maximised [11]. Although MC/DC achieves a good balance between software detection effectiveness and feasibility compared to more exhaustive coverage methods like *Multiple Condition Coverage (MCC)*, recent studies have shown that for languages with short circuit evaluation, MCC can also be feasible for software with decisions with up to 5 conditions [12].

There are several qualified code coverage tools for C and Ada which are widely used in the verification of avionics systems nowadays, however, to our knowledge none of them is currently supporting parallel programs such as GPU support for Vulkan SC, since current avionics software is inherently sequential.

III. MC/DC Coverage in Vulkan SC

A recent work on general purpose computations on GPUs [6] has argued that since Vulkan SC and GLSL are based on the C language, the existing code coverage tools for C can be extended to cover these dialects. In this paper we discuss in detail what are the implications of implementing MC/DC code coverage for GPU code, apart from the syntactic similarities of the GPU languages with sequential C code, and therefore what is the interpretation of MC/DC coverage in SIMT code. Moreover, we explain the potential issues which can arise in GPU software and can be detected using code coverage.

In order to support our discussion with code snippets, we are using an example of a frequently used GPU kernel (matrix multiplication) extracted from the GPU4S Bench [13] open source benchmarking suite. GPU4S Bench is an open source benchmarking suite which includes several application kernels which are common in several aerospace application domains, and which are implemented in multiple parallel programming models and with multiple versions, i.e. with a straightforward parallelisation, with an optimised version and with a vendor provided library when available. GPU4S Bench forms part of the European Space Agency (ESA) OBPMark (on-board processing benchmark) Kernels benchmarking suite [14] hosted at [15].

A. Vulkan SC Host API

As we described in Section II, Vulkan SC uses a set of API calls in order to manage the GPU execution. The code excerpt in Figure 2 shows an example of such code, which is used in order to launch a kernel to the GPU. Since Vulkan SC is a very low API, several steps are required for such an operation. Lines 1-4 setup a GPU *command buffer*, which in this example includes an one time kernel submission. Line 5 sets up the constant kernel arguments, i.e. the non-array parameters used in the call of the kernel function executed in the GPU. These arguments are stored in the GPU constant memory which we mentioned in Section II-A and in Figure 1. Lines 8-12 create a pipeline with the GPU operations which will be executed in the GPU, which in this example consists of a single GPU kernel launch. Line 13 computes the total number of GPU threads based on the sizes of the multiplied matrices as we discuss in the next subsection and line 14 specifies the kernel configuration, which is composed by `number_of_threads/256` work-groups. Lines 15-19 actually launch the kernel, and line 20 specifies a CPU barrier, in which the CPU remains idle until the GPU kernel finishes its execution.

It is worth noting that the Vulkan SC API calls are *asynchronous*, which means that the execution is returned immediately to the CPU, even if there is some memory transfer or parallel GPU computation taking place, such as between the lines 19 and 20. Despite this difference the (MC/DC) code coverage of the above code snippet can be computed on the CPU side with any existing qualified tool for C.

Regardless of the fact that some computation is performed at the same time on another computing element (i.e. in the GPU), the code statements which correspond to the CPU are executed one after the other, as in the case of conventional avionics code. Therefore, in terms of verifying the functionality of the host API code, it is enough to ensure the structural code coverage in the same way it is done in sequential code. In fact, we can argue that similar asynchronous operations already exist in avionics software when DMA (direct memory access) is used, or when a network message is transmitted over an AFDX (Avionics Full-Duplex Switched Ethernet) [16] or over another type of avionics network.

B. GLSL / GPU Code Coverage

GPU code in Vulkan SC can be specified either in GLSL source code or directly in SPIR-V. Since GLSL provides a more human friendly programming interface than the machine oriented bytecode of SPIR-V, in this paper we discuss code coverage at the source code level of this C-like language.

In the following subsections we are going to examine various elements related to GPU code coverage. First we reason about the conceptual meaning of MC/DC coverage in GPU source code and its equivalence to the MC/DC coverage of sequential CPU code. Next, we discuss additional points which can arise in GPU code and how they can be detected by MC/DC testing or static analysis.

```
1  VkCommandBufferBeginInfo beginInfo{};
2  beginInfo.sType=VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
3  beginInfo.flags=VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
4  vkBeginCommandBuffer(commandBuffer, &beginInfo);
5  pushConstants pC{.n = n, .m = m, .k = k};
6  vkCmdPushConstants(commandBuffer, pipelineLayout,
7     VK_SHADER_STAGE_COMPUTE_BIT,0,sizeof(pushConstants),&pC);
8  vkCmdBindPipeline(commandBuffer,
9        VK_PIPELINE_BIND_POINT_COMPUTE, pipeline);
10 vkCmdBindDescriptorSets(commandBuffer,
11       VK_PIPELINE_BIND_POINT_COMPUTE, pipelineLayout, 0, 1,
12        &descriptorSet, 0, nullptr);
13 uint_64t number_of_threads = m * n * k ;
14 vkCmdDispatch(commandBuffer, number_of_threads/256, 1, 1);
15 VkSubmitInfo submitInfo = {};
16 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
17 submitInfo.commandBufferCount = 1;
18 submitInfo.pCommandBuffers = &commandBuffer;
19 vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE);
20 vkQueueWaitIdle(queue);
```

Fig. 2. Example of Vulkan SC host code excerpt for launching a GPU kernel.

*1) **Code coverage concept in GPU Code**:* In Figure 3 we can see the naïve implementation of the single precision general matrix-matrix multiplication (sgemm) kernel in GLSL, which multiplies a matrix (m×n) with a matrix (n × k). With naïve implementation we refer to the straightforward parallelisation scheme, in which each thread is responsible for the computation of a single element of the output matrix, independently of the other threads.

Line 1 specifies the version of GLSL used in the kernel, and line 3 specifies the kernel configuration, which in this case uses a work-group size of 16×16. Recall that in the host code in Figure 2, the number of work-groups was specified as `number_of_threads` divided by 256, which is the number of threads within a work-group. Although the actual size of the work-group is not important for the correct functionality of the kernel, it is important that these two sizes match in order ensure the correct functionality of the GPU code. This creates a data coupling between the host Vulkan SC code (Figure 2, line 14) and the GLSL kernel code (Figure 3, line 3). Note however that the work-group size does have an impact in the kernel's performance and that it needs to be within the hardware limits of the target GPU, otherwise the kernel will not be possible to be executed.

Lines 5-14 specify the kernel array (*buffer*) and constant arguments. Again there is an additional data coupling of these lines with the corresponding Vulkan SC host code, e.g. Figure 2 line 5, as well as the part of the code which is responsible for the buffer creation and binding on the host side, which has been omitted from the code excerpt of Figure 2.

Lines 16-34 implement the actual executable code of the kernel. Notice that lines 17-18 obtain the position of the output matrix element computed by the current thread as a function of the global thread identifier, since the kernel is written in a SIMT programming model. These global thread identifiers are equivalent to the commented lines 20-23, which compute them as a function of the local thread identifiers, the thread work-group identifier and the work-group size.

In line 25, we ensure that the position of the current element

```
1  #version 450
2
3  layout(local_size_x=16, local_size_y=16, local_size_z=1) in;
4
5  layout(std430,set=0,binding=0) buffer inA { float a[]; };
6  layout(std430,set=0,binding=1) buffer inB { float b[]; };
7  layout(std430,set=0,binding=2) buffer outR {
8      float result[]; };
9
10 layout(push_constant) uniform pushConstants {
11     uint m;
12     uint n;
13     uint k;
14 };
15
16 void main(){
17     uint x = gl_GlobalInvocationID.x;
18     uint y = gl_GlobalInvocationID.y;
19     //Equivalent to:
20     //uint x = gl_WorkGroupID.x * gl_WorkGroupSize.x +
21                                     gl_LocalInvocationID.x;
22     //uint y = gl_WorkGroupID.y * gl_WorkGroupSize.y +
23                                     gl_LocalInvocationID.y;
24
25     if(x < k && y < m){
26
27         float r = 0.0;
28         for(uint i = 0; i < n; i++) {
29             r += a[y*n + i] * b[i*k + x];
30         }
31
32         result[y*k + x] = r;
33     }
34 }
```

Fig. 3. GLSL kernel implementation of naïve matrix multiplication.

TABLE II
MC/DC COVERAGE OF LINE 20 OF THE GPU CODE OF FIGURE 3

Test Case	$x < k$	$y < m$	result	$x < k$ independence	$y < m$ independence
1	T	T	T	3	
2	T	F	F	1	
3	F	-	F		1

TABLE III
MC/DC COVERAGE OF LINE 23 OF THE GPU CODE OF FIGURE 3

Test Case	$i < n$	result	$i < n$ independence
4	T	T	5
5	F	F	4

```
1  for( uint x = 0; x < number_of_threads_x; x++)
2  {
3      for( uint y = 0; y < number_of_threads_y; y++)
4      {
5          if(x < k && y < m){
6              float r = 0.0;
7              for(uint i = 0; i < n; i++) {
8                  r += a[y*n + i] * b[i*k + x];
9              }
10
11             result[y*k + x] = r;
12         }
13     }
14 }
```

Fig. 4. Equivalent sequential C code of the naïve matrix multiplication.

TABLE IV
MC/DC COVERAGE OF LINE 1 OF THE CPU CODE OF FIGURE 4

Test Case	$x < no\_of\_threads\_x$	result	$x < no\_of\_threads\_x$ independence
1	T	T	2
2	F	F	1

TABLE V
MC/DC COVERAGE OF LINE 3 OF THE CPU CODE OF FIGURE 4

Test Case	$y < no\_of\_threads\_y$	result	$y < no\_of\_threads\_y$ independence
3	T	T	4
4	F	F	3

TABLE VI
MC/DC COVERAGE OF LINE 5 OF THE CPU CODE OF FIGURE 4

Test Case	$x < k$	$y < m$	result	$x < k$ independence	$y < m$ independence
5	T	T	T	7	
6	T	F	F	5	
7	F	-	F		5

TABLE VII
MC/DC COVERAGE OF LINE 7 OF THE CPU CODE OF FIGURE 4

Test Case	$i < n$	result	$i < n$ independence
8	T	T	9
9	F	F	8

is within the matrix limits. In the loop of lines 28-30 we iterate over the entire line of matrix a and column of matrix b, and we compute their inner product. Finally, in line 32 we store the output to the result matrix.

Since the actual executable code of the kernel is in lines 16-34, this is the part of the GPU code that is subject to the MC/DC code coverage analysis. As we have explained, the kernel code is executed for each of the threads of the kernel which are in total: $number\_of\_threads = m \times n \times k$. By applying the concept of coverage in SIMT source code in the same way as it is applied in sequential CPU code, the meaning of coverage is interpreted as follows.

For statement coverage, checking whether each source code statement is executed at least once, it means that it needs to be executed by at least one of kernel threads.

For decision coverage, the entry and exit points of the kernel are exercised by default when a kernel is invoked, since its code is executed by all of its threads. In a similar way, since each thread is executing exactly the same code, it is enough to test that every decision has taken all its possible outcomes at least once, but not necessarily in the same thread. This means, that at least one thread needs to exercise each decision of the kernel.

The same concept applies for MC/DC, so every condition in the kernel needs to be tested with all possible outcomes at least once and that each condition in a decision need to have been shown to affect the decision outcome independently. This needs to be exercised by at least one thread of the kernel, not

each thread of the kernel.

Apart from the fact that each thread of the kernel is executing the same code, another way to reason about why it is enough to make sure that each condition and decision are exercised by at least one thread is to consider the equivalent sequential code. To that end, we introduce the concept of *single thread equivalence* of SIMT code. Figure 4 shows the equivalent sequential matrix multiplication code in C. The code is identical to the kernel code with the difference that it is executed within two nested loops, taking all possible values of x and y, which represent the number of threads in which the computation is partitioned in the x and y dimensions. Recall that according to the kernel configuration in Figure 2, line 14, the total number of threads in the x direction is $number\_of\_threads$ and in y direction is 1. Therefore, the thread identifiers in the GPU kernel are equivalent to the loop counters of the CPU version.

Since in the sequential version during MC/DC coverage we are not interested in which loop iteration each decision/condition was exercised, but only about the fact it was exercised at least once, naturally in the GPU version we are only interested whether each decision/condition was exercised by any thread at least once. This observation reduces significantly the effort and the amount of evidence required in order to collect coverage data for GPU code, because it keeps the required memory overhead for the collection of MC/DC information in the same range as in the one used by existing qualified code coverage tools.

Next, we examine the equivalence of MC/DC between the GPU and CPU code. In Figure 3, we have to cover two conditions. The condition of the `if` statement in line 25, and the loop condition in line 28. Table II and Table III show the test cases required to achieve MC/DC coverage for both conditions, following the method presented in [17]. The first column corresponds to the identifier of the test case, while the column marked as "independence", corresponds to the test case which provides decision independence with respect to that condition, as required by MC/DC.

Notice that due to the short circuit evaluation of the $\&\&$ (logical and) operator in GLSL (and in any other C-based language), test case 3 does not need to consider the value of the second condition. For the loop construct in line 28, Table III shows that it is trivial to achieve MC/DC coverage of the loop condition, as it requires only two test cases, one in which the loop is executed and one that it is not [18].

For comparison, Tables IV, V, VI and VII show the test cases required to achieve 100% MC/DC coverage in the sequential CPU code. We observe that the CPU and the GPU test cases only differ in the 4 additional trivial test cases, which correspond to the two additional loops which iterate over all thread identifiers. Therefore, the CPU test cases required to achieve MC/DC code are a superset of the GPU ones. In other words, the same test cases that achieve 100% MC/DC for the CPU code, can be used in order to achieve 100% MC/DC code in the GPU code.

In the general case, every naïve kernel parallelisation of sequential CPU code can be transformed to its equivalent sequential version by iterating over the grid dimensions, and the MC/DC test cases for the sequential version can be used to achieve 100% MC/DC coverage in the GPU version as well.

*2) **Code Coverage for complex parallelisation**:* In the previous subsection we considered the case where each kernel thread writes to a single memory position and operates independently of the other threads, which is usually found in naïve GPU implementations. However, such an implementation can be suboptimal in terms of performance as in the case of the matrix multiplication example we have examined. Moreover, some algorithms such as histograms, require to write in arbitrary output positions not determined by the thread identifier, a computation pattern known as *scatter* [6]. In that case, multiple potential programming issues can arise in GPU implementations, which we examine next.

Figure 5 shows an optimised implementation of the matrix multiplication kernel using tiling. *Tiling* [19] (also known as *blocking* [20]) is a technique in which a computation is performed in smaller chunks of data called *tiles*, in order to exploit data locality. The implementation in our example uses a 16×16 tile size. For this reason, the implementation features two 16×16 tile buffers in local memory (lines 18-19), in which we store the intermediate values fetched from each matrix used in the computation. Each thread of the kernel is again responsible of computing a single element of the output, however, all threads in the work-group are reusing the values fetched in the local memory. In fact, each thread in the work-group is responsible to load one element of each tile (lines 33-34) in every iteration. This creates an additional data coupling over the naïve implementation, since the working group size needs also to match the tile sizes in the local memory.

The loop in line 32 ensures that the processing is performed in steps of the size of the tile. After each thread in a work-group is loading the corresponding values in the two tiles, all threads of the work-group are waiting on a barrier (line 35) in order to ensure that the subsequent reads from the local memory (line 38) will see the correct values. This is required because each thread is reading values which were stored in the local memory by other threads. The loop in line 37 computes the inner product between the corresponding row and column of the two tiles in the local memory, and the barrier in line 40 ensures that all threads in the working group have computed their inner product and therefore consumed the local memory data, before proceeding with loading the next tile in the following loop iteration. Finally, when all tiles have been processed, the output element is written in the result matrix (line 42). In terms of MC/DC coverage, the tiled implementation has just an additional loop construct compared to the naïve one.

Figure 6 shows the equivalent tiled implementation for a CPU. In order to convert the SIMT model of the GPU kernel, we follow the same approach as before, by executing the code of the kernel as many times as the kernel work-groups for each grid dimension, using two nested loops (lines 1-4).

The difference in this case is for the parts of the kernel

```
1  #version 450
2  #define BLOCK_SIZE 16
3
4  layout(local_size_x=BLOCK_SIZE, local_size_y=BLOCK_SIZE,
5      local_size_z=1) in;
6
7  layout(std430,set=0,binding=0) buffer inA { float a[]; };
8  layout(std430,set=0,binding=1) buffer inB { float b[]; };
9  layout(std430,set=0,binding=2) buffer outR {
10     float result[]; };
11
12 layout(push_constant) uniform pushConstants {
13     uint m;
14     uint n;
15     uint k;
16 };
17
18 shared float tileA[BLOCK_SIZE][BLOCK_SIZE];
19 shared float tileB[BLOCK_SIZE][BLOCK_SIZE];
20
21 void main(){
22     uint    x = gl_WorkGroupID.x * gl_WorkGroupSize.x
23                               + gl_LocalInvocationID.x;
24     uint    y = gl_WorkGroupID.y * gl_WorkGroupSize.y
25                               + gl_LocalInvocationID.y;
26     uint   tx = gl_LocalInvocationID.x;
27     uint   ty = gl_LocalInvocationID.y;
28     float  r = 0.0;
29
30     if(x < k && y < m){
31
32         for (uint p = 0; p < n/BLOCK_SIZE; ++p) {
33             tileA[ty][tx]=a[y*n + p*BLOCK_SIZE+tx];
34             tileB[ty][tx]=b[(p*BLOCK_SIZE+ty)*k + x];
35             barrier();
36
37             for(uint i = 0; i < BLOCK_SIZE; i++) {
38                 r += tileA[ty][i] * tileB[i][tx];
39             }
40             barrier();
41         }
42         result[y*k + x] = r;
43     }
44 }
```

Fig. 5. Tiled matrix multiplication using local memory in GLSL.

TABLE VIII
MC/DC COVERAGE OF LINE 30 OF THE GPU CODE OF FIGURE 5 AND LINES 20 AND 41 OF THE CPU CODE OF FIGURE 6.

Test Case	$x < k$	$y < m$	result	$x < k$ independence	$y < m$ independence
1	T	T	T	3	
2	T	F	F	1	
3	F	-	F		1

TABLE IX
MC/DC COVERAGE OF LINE 32 OF THE GPU CODE OF FIGURE 5 AND LINES 22 AND 43 OF THE CPU CODE OF FIGURE 6.

Test Case	$p < n/BLOCK\_SIZE$	result	$i < n/BLOCK\_SIZE$ independence
4	T	T	5
5	F	F	4

TABLE X
MC/DC COVERAGE OF LINE 37 OF THE GPU CODE OF FIGURE 5 AND LINE 44 OF OF THE CPU CODE FIGURE 6.

Test Case	$i < BLOCK\_SIZE$	result	$i < BLOCK\_SIZE$ independence
6	T	T	7
7	F	F	6

```
1  for(unsigned int groupIdx=0;
2          groupIdx < number_of_workgroups_x; groupIdx++){
3      for(unsigned int groupIdy=0;
4          groupIdy < number_of_workgroups_y; groupIdy++){
5          float tileA[BLOCK_SIZE][BLOCK_SIZE];
6          float tileB[BLOCK_SIZE][BLOCK_SIZE];
7
8          for(unsigned int threadIdx=0;
9              threadIdx < group_size_x ; threadIdx++){
10             for(unsigned int threadIdy=0;
11                 threadIdy < group_size_y; threadIdy++){
12
13                 unsigned int x = groupIdx*group_size_x
14                                     + threadIdx;
15                 unsigned int y = groupIdy*group_size_y
16                                     + threadIdy;
17                 unsigned int tx = threadIdx;
18                 unsigned int ty = threadIdy;
19
20                 if (x < k && y < m)
21                 {
22                   for(int p=0; p < n/BLOCK_SIZE ; p++){
23                     tileA[ty][tx]=A[y*n+p*BLOCK_SIZE+tx];
24                     tileB[ty][tx]=B[(p*BLOCK_SIZE+ty)*k + x];
25                   }
26                 }
27             }
28         }
29         for(unsigned int threadIdx=0;
30                 threadIdx<group_size_x; threadIdx++){
31             for(unsigned int threadIdy=0;
32                 threadIdy<group_size_y; threadIdy++){
33                 unsigned int x = groupIdx*group_size_x
34                                     + threadIdx;
35                 unsigned int y = groupIdy*group_size_y
36                                     + threadIdy;
37                 unsigned int tx = threadIdx;
38                 unsigned int ty = threadIdy;
39                 float r = 0;
40
41                 if (x < k && y < m)
42                 {
43                     for(int p=0; p < n/BLOCK_SIZE ; p++){
44                         for(int i=0; i < BLOCK_SIZE ; i++){
45                             r += tileA[ty][i] * tileB[i][tx];
46                         }
47                     }
48                     C[y*k+x] = r;
49                 }
50             }
51         }
52     }
53 }
```

Fig. 6. Equivalent sequential C code implementation of the tiled matrix multiplication.

TABLE XI
MC/DC COVERAGE OF LINE 1 OF THE CPU CODE OF FIGURE 6.

Test Case	$groupIdx < groups\_x$	result	$groupIdx < groups\_x$ independence
8	T	T	9
9	F	F	8

TABLE XII
MC/DC COVERAGE OF LINE 3 OF THE CPU CODE OF FIGURE 6.

Test Case	$groupIdy < groups\_y$	result	$groupIdy < groups\_y$ independence
10	T	T	11
11	F	F	10

TABLE XIII
MC/DC COVERAGE OF LINES 8 AND 29 OF THE CPU CODE OF FIGURE 6.

Test Case	$threadIdx < group\_size\_x$	result	$threadIdx < group\_size\_x$ independence
12	T	T	13
13	F	F	12

TABLE XIV
MC/DC COVERAGE OF LINES 10 AND 31 OF THE CPU CODE OF FIGURE 6.

Test Case	$threadIdy < group\_size\_y$	result	$threadIdy < group\_size\_y$ independence
14	T	T	15
15	F	F	14

where the work-group execution is synchronised using the barrier. The barrier, divides the loop of the lines 32-41 of Figure 5 in two parts which need to be completed before executing the second part which is serialised. In order to do this, each of these parts of the loop needs to be executed for each local thread identifier within the work-group. This can be achieved by executing them within another set of nested loops, one for each work-group dimension, which take all values of the work-group's thread identifiers. Notice that again, the thread and work-group identifiers become the loop counters in the introduced loop constructs. The indices of the elements accessed by each thread are redundantly computed within the loops (lines 13-18 and lines 33-38), as well as the conditions depending on them (lines 20 and 41). Despite the code reorganisation of the sequential version, the conditional statements and the computed indices are identical to the GPU version shown in Figure 5.

Tables VIII, IX and X show the test cases required in order to achieve 100% MC/DC coverage for the GPU version, as well as for the conditions which are the same in the equivalent CPU code. In addition to these, Tables XI, XII, XIII and XIV provide the additional test cases which are unique to the equivalent sequential version.

Comparing the MC/DC conditions which are required to be covered, the CPU version has the additional loops which iterate over all working groups and their threads for both x and y dimensions, as well as the redundant conditions and loops which are repeated for each part of the GPU kernel which is divided by the barriers. However, the redundant conditions are identical, which means that they can be covered with the same test cases. For example, the `for` loops in lines 8 and 29 have the same condition, therefore can be covered with the same test cases which are provided in Table XIII. Similarly, the redundant `if` conditions in lines 20 and 41, are covered with the same test cases provided by Table VIII, which are also the same with the ones needed for the coverage of this condition in the GPU version (line 30).

Therefore, again we notice that the test cases required in order to achieve 100% MC/DC coverage in the tiled sequential version, is a superset of the test cases which can achieve 100% MC/DC coverage of the tiled kernel implementation.

In addition to the implications of MC/DC coverage which have examined in this example, when different threads are writing to buffers which are accessible by other threads either in the global or local memory, we need to make sure that they use different indices, otherwise a race condition can be introduced. In case the threads need to write in the same index because it is required by the algorithm as it is the case in histograms, atomic instructions need to be used for the write. Moreover, whenever a thread is reading from a memory position which another thread of the same work-group has written to, a barrier instruction is required, as we have seen in the case of the tiled matrix multiplication example.

Such programming mistakes can either be identified with static analysis methods [21] [22], or can manifest themselves with wrong output during the execution of the MC/DC tests. However, similar to the case of the existing qualified MC/DC coverage tools for CPUs, the existing static analysis GPU methods need to be extended to cover Vulkan SC and its GLSL variant, as well as to be qualified for use in avionics.

IV. CONCLUSION

In this paper we discussed in detail the concept of MC/DC coverage for Vulkan SC code using examples of GPU code. We have demonstrated that for the host (CPU) part of the Vulkan SC code, existing qualified code coverage tools can be used without modifications.

For the case of the GLSL GPU code, we have seen that GPU kernels can be easily converted to sequential code, by simply calling their SIMT code within appropriately translated loops. Based on this single thread equivalence, we argued that the requirements of MC/DC coverage need to be satisfied by at least one thread of the kernel, similar to the fact that in conventional sequential code they need to be satisfied by any loop iteration. This means that existing qualified code coverage tools for source code MC/DC can be extended to work in the same way on GPU code coverage. As we have seen in our example, the test cases used to achieve 100% MC/DC coverage in the equivalent CPU implementations are a superset of the test cases that can achieve 100% MC/DC coverage in the GPU implementations.

Finally, we discussed that in complex GPU kernels data races may be introduced when multiple threads are writing in

the same memory position and/or are reading from memory positions written by other threads if barrier synchronisation or atomic operations are not used. We argued that this situation can be revealed either with static analysis methods, or they can be manifested with MC/DC testing. However, in both cases existing qualified MC/DC code coverage tools for CPUs and static analysis methods for GPUs need to be extended with support for Vulkan SC and GLSL.

ACKNOWLEDGMENT

This work was performed within the Airbus TANIA-GPU Project ADS (E/200). 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-C21 and IJC-2020-045931-I (Spanish State Research Agency / Agencia Española de Investigación (AEI) / http://dx.doi.org/10.13039/501100011033) and the HiPEAC Network of Excellence.

REFERENCES

[1] 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.

[2] RTCA and EUROCAE, *DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification*, 2012.

[3] ——, *DO-254 / ED-80, Design Assurance Guidance for Airborne Electronic Hardware*, 2000.

[4] EASA, *AMC 20-193 Use of multi-core processors (MCPs)*, 2022.

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

[6] M. M. Trompouki and L. Kosmidis, "DO-178C Certification of General-Purpose GPU Software: Review of Existing Methods and Future Directions," in *40th Digital Avionics Systems Conference (DASC)*, 2021.

[7] Khronos Group, *Vulkan SC 1.0.10 - A Specification*, 2022.

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

[9] ——, *OpenGL SC 2.0.0 (Full Specification)*, 2016.

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

[11] M. P. Heimdahl, M. W. Whalen, A. Rajan, and M. Staats, "On MC/DC and Implementation Structure: An Empirical Study," in *27th Digital Avionics Systems Conference (DASC)*, 2008, pp. 5.B.3–1–5.B.3–13.

[12] S. Kandl and S. Chandrashekar, "Reasonability of MC/DC for Safety-relevant Software Implemented in Programming Languages with Short-circuit Evaluation," in *Computing*, vol. 97, 2015, p. 261279.

[13] 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 Politècnica 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.

[14] D. Steenari, L. Kosmidis, I. Rodriguez-Ferrandez, A. Jover-Alvarez, and K. Forster, "OBPMark (On-Board Processing Benchmarks) - Open Source Computational Performance Benchmarks for Space Applications," in *2nd European Workshop on On-Board Data Processing (OBDP)*, 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5638577

[15] D. Steenari et al., "On-Board Processing Benchmarks," 2021, http://obpmark.github.io/.

[16] ARINC, *ARINC Specification 664: Aircraft Data Network, Part 7-Avionics Full Duplex Switched Ethernet (AFDX) Network*, Aeronautical Radio, Inc, 2005.

[17] J. J. Chilenski and S. P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *IET Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, September 1994.

[18] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition/ Decision Coverage," NASA, Tech. Rep. NASA/TM-2001-210876, 2001.

[19] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Parallel Processing for Scientific Computing*, 1989.

[20] R. Schreiber, "Block Algorithms for Parallel Machines," in *Numerical Algorithms for Modern Parallel Computer Architectures*, 1988.

[21] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A Verifier for GPU Kernels," in *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012, p. 113132.

[22] E. Bardsley and A. Donaldson, "Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels," in *NASA Formal Methods*, 2014, pp. 230–245.