COOK Access Control on an embedded Volta GPU

Benjamin Lesage, Frédéric Boniol, Claire Pagetti

COOK Access Control on an embedded Volta GPU

Benjamin Lesage
ONERA
Toulouse, France
benjamin.lesage@onera.fr

Frédéric Boniol
ONERA
Toulouse, France
frederic.boniol@onera.fr

Claire Pagetti
ONERA
Toulouse, France
claire.pagetti@onera.fr

***Abstract*—The last decade has seen the emergence of a new generation of multi-core in response to advances in machine learning, and in particular Deep Neural Network (DNN) training and inference tasks. These platforms, like the JETSON AGX XAVIER, embed several cores and accelerators in a SWaP-efficient (Size Weight and Power) package with a limited set of resources. However, concurrent applications tend to interfere on shared resources, resulting in high execution time variability for applications compared to their behaviour in isolation.**

***Access control* techniques aim to selectively restrict the flow of operations executed by a resource. To reduce the impact of interference on the JETSON Volta GPU, we specify and implement an access control technique to ensure each GPU operation executes in isolation to reduce its timing variability. We implement the controller using three different strategies and assess their complexity and impact on the application performance. Our evaluation shows the benefits of adding the access control: its transparency to applications, reduced timing variability, isolation between GPU operations, and small code complexity. However, the strategies may cause some potential slowdowns for applications even in isolation but which are reasonable.**

***Index Terms*—Interference, Shared resources, GPU, access control, JETSON AGX XAVIER, Locking, software hooks**

I. INTRODUCTION

The aeronautic domain faces two constraints: first, it is subject to certification and second, it relies massively on Commercial Off-The-Shelf (COTS) processors. Thus, to embed any multi-core processor – with or without accelerators –, it is mandatory to provide assurance of *time predictability*. Time predictability encompasses the capability to compute a safe and tight upper bound on the number of cycles, the *execution time*, required to execute an application in the worst case [1], [2]. Time predictability implies mastering the hardware platform mechanisms and their impact on execution time variability. Indeed, high variability may endanger the overall aircraft/system safety an thus should thus be avoided as much as possible. This is notably the case in the presence of interference, where applications compete for shared resources resulting in additional variability over their execution time in isolation.

Context: COTS platforms, multi-core- or accelerator-based, tend to lack a clear documentation which hinders the analysis efforts to understand and bound the effect of interference. They further tend to neglect predictable hardware and software mechanisms, in favour of delivering higher throughput. While they may exhibit little variability for applications in isolation, interference and uncertainty tend to be exacerbated when applications execute in parallel. *Mitigation* techniques aim to reduce or bound interference occurring on a platform, and as a result the related timing variability. *Access control* techniques in particular operate by regulating the flow of operations to specific resources, to reduce pressure on them and provide finer-grain control. Many papers tackle this problem for multi-core, but far less focus on accelerators.

Consider an application, as shown in Figure 1 (left), composed of *host code* that executes on a CPU core and offloads some *operations* onto the GPU. Application τ_1 offloads a single operation, and waits for the completion of the operation (star-shaped synchronisation point). Figure 1 (centre) shows another application also offloading some GPU operation. When two applications run in parallel, host code execution on the CPU is handled by the Operating System (OS) scheduler. At some point, GPU operations from both applications are offloaded to the GPU. Figure 1 (right) highlights this behaviour, as well as our questions: 1) What is the internal behaviour of the operations and how do they interfere? 2) What is the impact for an application on its execution time to run in parallel, over isolation? 3) How to account for, or mitigate said impact if not acceptable?

Contributions: Our objective is to mitigate the timing variability on GPU operations generated by concurrent applications. We propose a temporal access control technique to isolate the execution of said operations on the GPU, reducing the impact of interference on the operations' execution time variability. The method generates software hooks [3] from simple templates to modify the behaviour of existing GPU routines. The templates can thus easily be adapted to new or updated routines. Using hooks, the access controller is transparent to applications; it requires no modification of applications and can support a wide variety of runtime environments. Our target platform, the JETSON AGX XAVIER [4], is introduced in Section II to outline its behaviour w.r.t. the GPU, source of interference, and mitigation means. Section III describes related work in the context of the JETSON. Based on these observations, Section IV highlights constraints and desirable features for our access controller. Section V then introduces our access control technique, and compliant implementations. The evaluation in Sections VI and VII assess our claim that the proposed approach satisfies its objectives: transparency for the application, isolation of GPU operations, and mitigation of GPU operations timing variability.

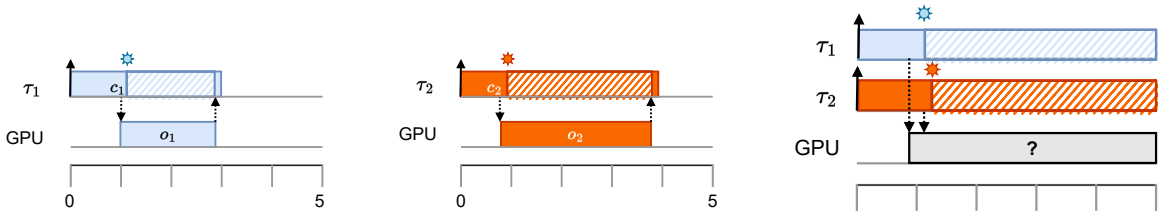


Fig. 1: Applications 1 and 2 running a kernel in *isolation* (resp. left and centre), or in *parallel* on the GPU (right).

II. Jetson AGX Xavier platform

This section presents the Volta GPU on the Jetson AGX Xavier with a focus on its execution model, to identify notable resources, sources of interference, and possible means of mitigation.

A. System model

We consider a simple system where one or more applications τ_i execute on the Jetson. It features the Xavier System-on-Chip (SoC) outlined in Figure 3. The Xavier SoC embeds a Volta GPU, an 8-core Carmel CPU complex, and dedicated accelerators for video and audio processing applications (omitted from the Figure). The CPU complex and GPU access the main memory through a shared memory controller fabric. In this work, we assume that only one of the Carmel cores per application is in use and that the other accelerators are not. Each application furthermore uses its own separate GPU context and stream to enqueue GPU operations (left of Figure 3).

An application τ_i, as illustrated in the Figure 2 chronogram, is composed of two parts:

- a *host code* that executes on the Carmel cores and offloads work on the GPU itself through *GPU routines*;
  - A *GPU routine* is an asynchronous call c_i on the host (dashed downward arrow) to execute one corresponding *GPU operation* o_i;
  - The host code may perform a number of host-side operations in-between calls to GPU routines;
  - A *synchronisation barrier* (star symbol) is a synchronous *GPU routine* awaiting for the completion of all prior GPU operations. The barrier may thus block the application (hashed blocks).
- a sequence of *GPU operations*, o_i, organised in *bursts*:
  - a *burst* is a sequence of *GPU operations* launched as an ordered group of GPU routines by the host code;
  - a *GPU operation* is either a *kernel*, or a *copy* operation that runs on the GPU;
    - a *copy* operation moves data between the host and GPU memory space;
    - a *kernel* operation is a function executed on the GPU following a *grid*, that is groups of threads, where a thread is the basic GPU execution unit running the kernel in parallel;
  - the *bursts* are ordered in a sequence: the host code sends a burst of operations, then waits for the results from the GPU using a *synchronisation barrier*. Only after the completion of the burst (dashed upward arrow), the application then sends the next burst of operations waiting for its results, and so on until all the bursts have been offloaded.

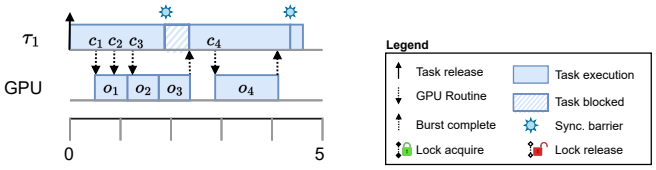


Fig. 2: Single application with two kernel bursts

The designer has many ways to architect their applications. Indeed, they need to choose all the application parameters: number, type, and order of GPU operations, number of bursts, size of bursts, and position of synchronisation barriers. Figure 2 shows such a trade-off for an application running in isolation composed of two bursts, of respectively three (o_1, o_2, o_3) and one (o_4) operation(s). In the first burst, the CPU has to wait for the completion of its GPU operations. The second burst of GPU operations completes before the synchronisation barrier, and the CPU can immediately cross said barrier.

B. Volta Execution Model

A user can define and execute kernels and other GPU operations through the CUDA Runtime or rely on high-level libraries, on top of the Runtime, such as cuDNN. The operations transit through the CUDA Runtime to the driver (left in Figure 3) which interacts with the GPU to setup the required structures. Most calls to the CUDA Runtime from the host code are asynchronous, allowing applications to submit a burst of GPU operations until synchronisation is required, e.g. to retrieve some results from the GPU. We focus in the following on the definition and execution of computation kernels.

A kernel call is shaped by the computation *grid* definition, i.e. the number of *thread blocks* and their shape. All thread blocks in a call are equally shaped [5]. The size of the kernel, the total number of threads invoked on a call, is thus the product of the number of threads per block and the number of blocks. Each thread is given a unique identifier to address different data segments. It is composed of its block index in the grid, and its thread index in the block. Both are accessible during kernel definition through the CUDA C extensions.

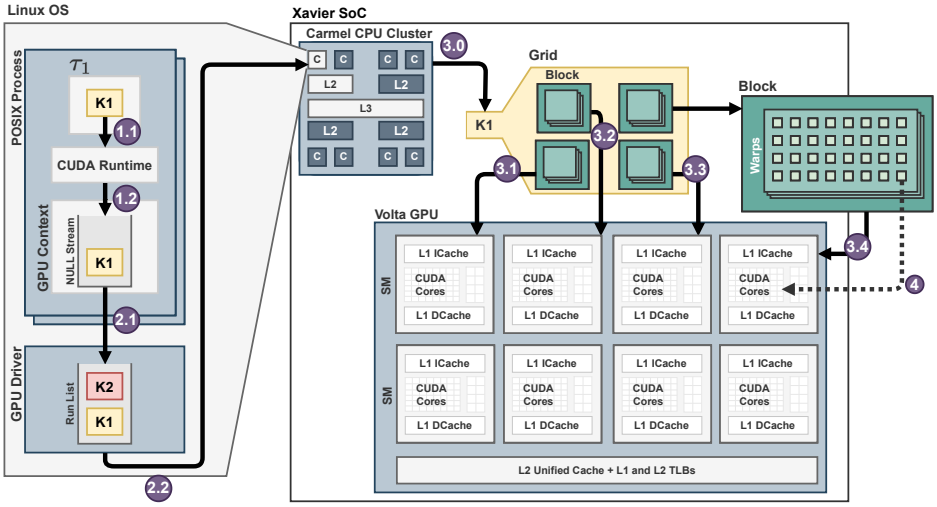


Fig. 3: Overview of the Volta GPU execution model

A kernel call is placed in a *stream*, a (possibly) user-defined queue of GPU operations (Steps 1.1 and 1.2 in Figure 3). This provides some guarantees on the ordering of related operations with regards to their execution on the GPU. The call travels through the software stack (Steps 2.1 and 2.2) through multiple queues. While the First-In First-Out (FIFO) ordering of operations in a stream is maintained, a kernel might be interleaved with kernels from other streams and run concurrently with them. The block scheduler (Steps 3.0–3.4) dispatches each block of the kernel to a Streaming Multiprocessor (SM), depending on the block's resource requirements and the SM occupancy. The block will remain on its allocated SM until its threads complete (unless a rescheduling occurs on preemption).

The JETSON AGX XAVIER Volta GPU is composed of 8 SMs, depicted in Figure 3, each with its own private L1 instruction and data caches (respectively ICache and DCache). All SMs share a unified L2 cache and 2 levels of TLB. Each SM is further divided into 4 processing blocks (SMP). The register file on an SMP, or L0 data cache, holds the context of multiple threads. Each SM is itself composed of 64 CUDA cores and 8 Tensor cores, split evenly across the SMP. Tensor cores are a class of Deep Learning Accelerators supporting multiply-add operations on matrices, as instructed by the user (or a library). Restrictions on the number of registers in each SM imply that not all threads in a kernel call can execute concurrently. Blocks can hold a maximum of 1024 threads, and at most 32 blocks can reside at once on a single SM.

SMs follow the Single Instruction Multiple Data (SIMD) execution model: one instruction is executed across multiple threads at the same time, possibly addressing different data[1]. Threads are scheduled and executed in groups of 32, called *warps*, such that each thread belongs to a single warp across its lifetime. Warp size and the allocation of threads in warps are not user-controlled parameters but platform specific. The warp scheduler on each SMP can schedule up to one warp every cycle. Instructions from two distinct warps may coexist on the same core provided they rely on different functional units, e.g. a long running load due to a cache miss and an integer addition. The combined register files on each SM can accommodate up to 64 warps to ensure the warp schedulers can maximise core occupancy.

[1]Note that the Volta GPU architecture introduces thread divergence, where threads may have different program counters. The scope and impact of thread divergence on the SIMD model is unclear.

III. RELATED WORK

There has been considerable effort to characterise the behaviour of GPU accelerators, in particular work on NVIDIA GPUs [6]–[9] and the assorted CUDA software stack [8]–[11]. [8], [11] do identify high-level scheduling rules for kernels in the CUDA software stack, down to the dispatching of their constituent blocks on the GPU resources. These rules notably highlight that software resources might induce interference between concurrent kernels. These contributions highlight the need for mitigation techniques to cope with interference on GPU.

A. Related Work on Access Control for GPU accelerators

Various resource managers have been proposed in the context of multi- and many-core architectures to build robust, reconfigurable platforms, e.g. in the context of the SCARLETT [12], ACTORS [13], DREAMS [14], or SECREDAS [15] projects. They allow for a unified view of the resources available across the whole system. All these methods share the same requirement for robust access control methods: to understand the available resources' capacity to serve queries without causing interference, to allocate a portion of said capacity, and to possibly reclaim said capacity at runtime.

The Volta GPU features a wide range of resources which might act as interference sources between between concurrent operations. Unfortunately, the Volta GPU architecture offers

no granularity of isolation between kernels. NVIDIA proposes Multi-Instance GPU (MIG) [16] to allocate hardware resources to a kernel. However, MIG is supported by neither the Volta GPU nor the JETSON platforms. We thus consider software access control techniques, in the absence of explicit hardware support. They operate on *who* does execute the kernel, and *when* to execute kernels. They may rely, or not, on the application *cooperation* to the access control by calling specific primitives or restricting its demand on the resource capacity.

The approach in [8] did consider the use of access control techniques on AMD platforms, comparing locking and dedicated GPU support. Although AMD GPUs do explicitly support mapping kernels to specific cores, such facilities are not officially supported on the JETSON product line. [17] did recently highlight through patent analysis and reverse-engineering that NVIDIA GPUs have had the capability for a decade. The authors do offer an API to exploit it on JETSON platforms and they achieve compute resource partitioning on JETSON platforms. The approach does rely on cooperative applications, and it is limited to CUDA versions which have been analysed.

Temporal access control techniques restrict the kernel(s) which can run on the GPU at any given time (*when*). *Locking* [18]–[20] relies on applications acquiring a shared lock before accessing the resource to guarantee mutual exclusion, either at the GPU-level [18] or between the GPU and CPU [19]. Similarly, a server [21]–[23] acts as a concentrator or proxy for applications' requests, scheduling kernels and redirecting them to the resource. RGEM [23] in particular replaces the GPU libraries with a custom implementation to schedule GPU operations in isolation. Locks and servers intervene on the CPU-side, before operations are dispatched to the GPU.

Persistent Thread Blocks (PTB) [24]–[26] provide a spatial access control method to allocate compute resources (SM) on the GPU. PTB act as runners for a kernel (*who*). A PTB executes the kernel's blocks, fetched from a work queue, as its own. It will persist, on a specific SM, repeating the process until completion or otherwise instructed. Reducing the number of PTB for a kernel thus reduces the kernel's demand on the GPU capacity. PTB however require a cooperative application, modifying its kernels to wrap their definition and execution into the PTB execution loop. Such modifications can be performed during kernel definition or automatically, as source-to-source transformations.

We focus in the following on locking-based approaches. They have no impact on the GPU software stack. Furthermore, each application remains in charge of its own accesses to the GPU. In comparison, a server acts as a unique concentrator for GPU operations. Consider as an example the NVIDIA Multi-Process Service (MPS) server [27]. Its error containment is such that a fatal exception in a kernel may propagate to others sharing the same GPU. We also leverage the principle of library hooking to offer a platform to deploy different mitigation strategies, even in presence on non-cooperative applications.

B. Related Work on CUDA Hooking

A hook [3] is a function which (often dynamically) replaces an existing one in an application or library. Hooking may be used in a wide range of contexts to extend applications when recompilation is not an option, notably for providing debugging, instrumentation, or analysis capabilities. NVIDIA relies on hooking for a number of its tool. `nsys` does inject replacement CUDA libraries into an instrumented application to trace their execution and collect relevant statistics. However, `nsys` does not allow for users to define their own hooks. This was supported by NVBit [28]: user-defined tools are loaded as dynamic libraries to intercept calls to the CUDA stack. However the tool is no longer maintained.

BWLOCK [19] replaces CUDA API calls to automatically acquire/release the bandwidth lock upon kernel execution. RGEM [23] entirely replaces the CUDA API implementation with its own. Similarly, numerous works focused on GPU virtualisation [29]–[31] provide replacements for the CUDA API to defer the execution of GPU kernels to remote hosts. An application can thus transparently execute kernels from a host without a GPU. However, CUDA hooks tend to be ad-hoc replacements, for only a limited subset of the CUDA API. Applications are thus limited to the supported methods. Furthermore, some CUDA libraries circumvent the hook injection methods used by these approaches.

GPUSync [20] does address similar scheduling concerns as our approach, considering GPU and copy engines allocation, worker preemptions, and closed-source driver threads. The approach further benefits from its integration into the platform kernel, currently unsupported by the JETSON product line. Information on the hooks implemented by GPUSync is sparse, and it is unclear if it could support a runtimes such as the ONNX one which circumvent some library loading mechanisms. Our contribution supports (without aiming for) the definition of a scheduler, e.g. generating hooks for approaches like GPUSync.

IV. ACCESS CONTROL ON THE JETSON

The behaviour and control offered by the JETSON AGX XAVIER platform (Section II) restricts the scope of applicable techniques, to mitigate the effect of interference on the execution time of the GPU operations of an application. We also need to consider lessons learned from existing work (Section III).

A. Constraints

Considering the kernel execution model in Figure 3 highlights a number of shared resources on the Volta GPU, both at the software (left) level, and hardware (right) one. We need to consider potential interference sources, support for any granularity of access control, and understand the level of control available on the platform.

Cores (their registers and functional units) are split evenly between SMs, each with its own private L1 caches. Any method allocating resources to applications at the SM level would still suffer interference on the shared L2 and TLB

caches. Due to a lack of documentation, we did not represent any communication medium inside the GPU. However, prior work [9] did highlight that concurrent accesses to the same resource may suffer interference due to a shared communication medium. The software stack suffers from a similar lack of clarity in spite of recent reverse-engineering efforts [11]. All layers of the platform, from the libraries to the GPU, may contribute to some extent to scheduling the GPU operations between applications. A *stream* does provide guarantees on the ordering of its own GPU operations, but few on the ordering or isolation of operations between streams in the same or separate GPU contexts. Separate OS processes do default to separate GPU contexts, thus providing some isolation. But the software stack still funnels GPU operations into a limited number of shared queues, within and between GPU contexts, and then at the driver level.

The GPU hardware offers only limited visibility and control on the resources used by or allocated to a running GPU operation, and no granularity of isolation between operations. Similarly at the application level, once a GPU operation has been inserted into a stream the application loses control over its execution or schedule. In the absence of a formal platform model or analysis tool, such as OTAWA [32], PasTiS [7] or AIT [33], each GPU operation is considered as a non-preemptable black box, and we rely on end-to-end execution timing measurements. We further focus on software-level techniques, in the absence of hardware support. We discuss target features in the following section.

B. Specification

We consider a number of desirable aspects for the proposed access control technique. As discussed earlier, those consider related work, their benefits and drawbacks.

Aspect 1 (Transparency): The method should be transparent to the application. It should require no modification to the application source code or model, nor cooperation with a specific API, nor modification to its runtime if any. The aspect notably ensures the method can be applied to a wide range of applications, or models without restriction on their runtime or additional maintenance.

Aspect 2 (Resiliency): The method should require no modification to the operating system kernel or vendor-provided code. The kernel and the driver on the JETSON AGX XAVIER are open-source. However such modifications require costly maintenance to be kept up-to-date with the upstream code.

Aspect 3 (Maintenance): The method should not be tied to a specific revision of the software platform, or offer a clear path for revisions. Updates to the software platform tend to be backwards compatible, allowing older software or methods to work. Methods which rely on deprecated or undocumented behaviours risk becoming outdated.

Aspect 4 (Scope): The access control shall focus on interference between operations running on the GPU issued by different applications and their latency. Interference between the CPU and the GPU, or stemming from other accelerators on the platform are outside the scope of this work.

Aspect 5 (Temporal): The access control shall focus on temporal access control techniques, for the GPU as a whole. The JETSON GPU offers no hardware support or guarantees on the isolation of its resources.

Aspect 6 (Burst preservation): The access control shall maintain existing synchronisation barriers. The resource manager may introduce new synchronisation barriers, splitting an existing burst into one or more bursts. This is required to maintain the correctness of executed bursts. Additional synchronisation points may provide some flexibility without jeopardising the functional correctness of applications.

Aspect 7 (Order preservation): The access control shall maintain the relative ordering between GPU operations. The access control may introduce stronger guarantees on the ordering between GPU operations. As per Aspect 6.

Aspect 8 (Throttling): The access control defers the insertion of GPU operations into the CUDA software stack to improve isolation between applications. Once operations enter the CUDA Software stack, and in particular streams, their execution is deferred to the platform, and only limited control and guarantees are available.

Aspect 9 (Software): The access control shall rely on user-level, software-only methods. As per Aspect 5, hardware and software support on the platform is limited.

V. COOK ACCESS CONTROLLER

We introduce in the following an access control technique based on deferring *when* GPU operations enter GPU streams to control their execution, effectively throttling the flow of GPU operations to the GPU and its driver. Our approach relies on the configurable generation of C hooks (COOK) to alter the behaviour of GPU routines (Section V-A). We focus on hooks for the CUDA Runtime library, which all applications will call, directly or indirectly, to schedule operations on the GPU. While the driver and driver API would be appropriate candidates for hooking, their interfaces are not as well documented and are more volatile. We propose policies to enforce access control in line with our requirements in Section V-B.

The definition of a configurable process is justified by Aspect 3. New versions of the software platform may deprecate, add to, remove from, or alter the behaviour of the hooked library. Neither deprecation nor removal should alter the COOK configuration or the generated hooks. At best, the same configuration may apply to multiple versions of the hooked library. Additions and alterations need to be considered separately to assess whether the existing hook templates remain applicable. The configuration may be revised, with new conditions, but still rely on existing hook templates. At worst, a whole new configuration is required.

A. Hook generation

The COOK toolchain generates a hook library to intercept all calls to a selected hooked library. Specifically, we hook onto the CUDA Runtime library, *libcudart.so*[2], as it is on the

[2]This contribution focuses on dynamic libraries without loss of generality. Static library hooks are generated in a similar fashion.

critical path to the GPU for the CUDA software stack. The generation of the hook library is a process configured by a set of hook templates and conditions. Each hook template is a code template instantiated with a function declaration to create a corresponding hook. Hook conditions capture the list of functions to hook onto for each template. The overall workflow of the COOK toolchain for a given library is outlined in Figure 4. It is as follows:

- *Extract symbols*: list the symbols exported by the hooked library to capture its interface;
- *Find symbol declaration*: for each symbol s, find its signature in the library headers to capture its arguments and their types;
- *Generate a hook*: for each symbol s, if a hook condition matches then apply its template to generate a hook, i.e. a function s^h that intercept calls to s;
- *Generate a trampoline*: for each symbol s undefined or condition-less, generate a default, minimal hook s^h configured to simply raises an error, or simply redirect calls to the hooked function s [34];
- *Compilation*: gather all generated hooks and trampolines and compile the hook library.

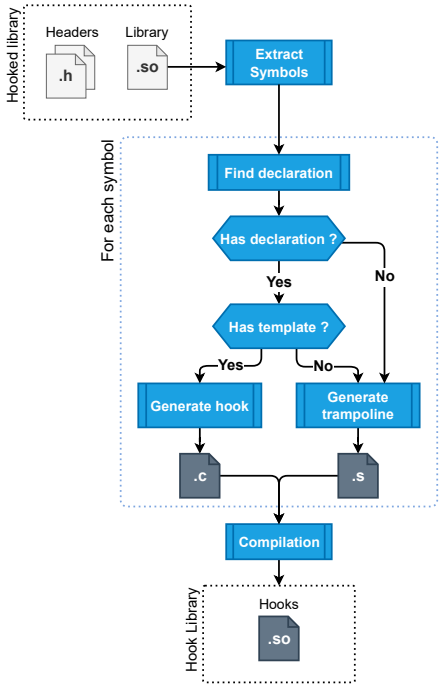


Fig. 4: Generation of a hook library

During compilation (Figure 5), an application may rely on symbol declarations provided by library headers to generate calls, listing undefined symbols. Undefined symbols are then resolved at runtime by the loader which match the libraries required by the application with available ones. A match must have the correct name, e.g. *libcudart.so*, and expose all symbols required for the execution of the application executable. The hook library could expose only the hooked symbols, without trampolines, with the loader resolving potential gaps. In practice, however, some CUDA libraries and applications may circumvent the loader to load the CUDA Runtime, looking instead into specific paths. As such, the generated hook library must be able to replace the CUDA Runtime in-place with all its symbols (Aspect 1).

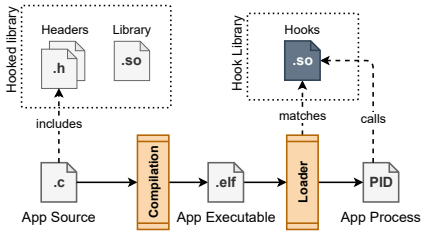


Fig. 5: Use of a hook library by an application

B. Access control strategies

We propose three access control strategies to mitigate the impact of interference on GPU operations scheduled by concurrent applications. They effectively aim to schedule GPU operations in a non-preemptive fashion. We depict for each strategy how it hooks related GPU routines. All strategies rely on the same principles:

- Any operation running on the GPU should have exclusive use of the GPU resources (Aspect 5). This is achieved by ensuring an operation only executes if its application has first acquired the global GPU lock `GPU_LOCK`. Only one application can `acquire` the lock at any given time. Calls to `acquire` from other applications will be blocked until a `release` from the current owner of the GPU lock[3].
- Strategies hook into the GPU routines generating *copy* and *kernel* GPU operations, respectively the `cudaLaunchKernel` and the `cudaMemcpy` methods in the CUDA runtime API. Each routine inserts the corresponding operation and its parameters into the application stream as depicted in Algorithms 1 and 2.

Algorithm 1 Kernel Launch method in the CUDA Runtime

1: **procedure** CUDALAUNCHKERNEL($func, grid, args, stream$)
2: **insert op** Execute(func, grid, args) **in** stream
3: **end procedure**

Algorithm 2 Memory copy method in the CUDA Runtime

1: **procedure** CUDAMEMCPY($dst, src, size, mode, stream$)
2: **insert op** Copy(dst, src, size, mode) **in** stream
3: **end procedure**

[3]Our implementation uses a semaphore from the POSIX threads library, and the underlying scheduling policy, included as part of the JETSON AGX XAVIER platform. The election of an operation amongst all pending candidates for execution, that is the definition of a GPU-specific scheduler, is an orthogonal problem outside the scope of this work.

1) Host Callback (callback) strategy: The callback strategy relies on the execution order guaranteed by a GPU stream, i.e. operations inserted in a GPU stream are executed in a First-In First-Out order. An operation cannot start until all the previous ones completed. As illustrated in Figure 6 (hook-related changes use a lighter shade), the callback strategy adds an `acquire` operation (first block on the GPU) before any GPU operation (second block on the GPU) with a matching `release` afterwards (last block on the GPU).

The operation acquiring the GPU mutex will thus block the execution of operations in its stream until the GPU is available. Conversely, the release operation will only free the GPU once all prior operations complete. Only one stream across all applications can proceed through the GPU lock at any time. Others are either ineligible for scheduling by the GPU, or waiting on a blocking `acquire` operation. The approach uses the CUDA Runtime `cudaLaunchHostFunc` to revert control back to the CPU to execute a specific host method (`acquire` and `release`). Algorithm 3 outlines the generated hook for the kernel launch method. (The memory copy uses the same code template resulting in a similar hook.)

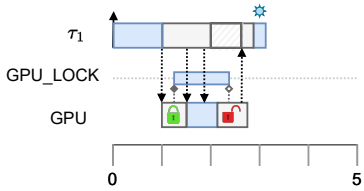


Fig. 6: Illustration of the callback strategy

Algorithm 3 Kernel Launch hook (callback strategy)

1: **procedure** CookLaunchKernel($func, grid, args, stream$)
2: **insert op** Callback(`acquire` GPU_LOCK) **in** stream
3: **insert op** Execute(func, grid, args) **in** stream
4: **insert op** Callback(`release` GPU_LOCK) **in** stream
5: **end procedure**

2) Synchronised Operation (synced) strategy: The synced strategy transforms GPU routines into synchronisation points. A call to a routine only completes after the related GPU operation does. This is comparable to the RGEM [23] approach. Figure 7 shows the behaviour of the hook. The hook first acquires the GPU lock (downward arrow to the lock), before actually calling the GPU routine (downward arrow to the GPU). The hook then waits for the operation's completion (star symbol). Once the operation is complete (upward arrow from the GPU), the hook releases the lock (downward arrow to the lock). An application thus schedules and executes at most one GPU operation at a time (thanks to the barrier). And only one application can schedule a GPU operation at any time (thanks to the lock). Algorithm 4 outlines the generated hook for the kernel launch method. (Like the prior strategy, the memory copy uses the same code template resulting in a similar hook.)

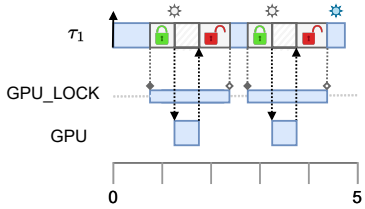


Fig. 7: Illustration of the synced strategy

Algorithm 4 Kernel Launch hook under the synced strategy

1: **procedure** CookLaunchKernel($func, grid, args, stream$)
2: `acquire` GPU_LOCK
3: **insert op** Execute(func, grid, args) **in** stream
4: **sync on** device
5: `release` GPU_LOCK
6: **end procedure**

3) Deferred Worker (worker) strategy: The worker strategy defers the execution of GPU routines and operations to a separate *worker* thread running on a separate CPU core for each application. As illustrated in Figure 8, the hooked GPU routine call from the task (downward arrow) notifies the worker instead of the GPU. Then for each application, and similarly to the synced strategy, its *worker* acquires the GPU lock (downward arrow to the lock), queues the GPU operation for execution (first downward arrow to the GPU), waits for the operation to complete (upward arrow from the GPU), and releases the lock (last downward arrow to the lock). The GPU operation effectively transits through the worker (using a new $worker\_queue$ stream per worker) to the GPU. Each worker acts as a synchronisation source, ensuring a synchronisation barrier only releases an application once the worker completed all queued work (upward arrow from the worker). Workers from different applications synchronise to ensure only one of them schedules an operation at any given time.

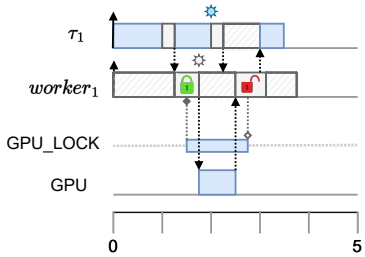


Fig. 8: Illustration of the worker strategy

Algorithm 5 outlines the generated hook for the kernel launch method. (Like the prior policies, the memory copy uses the same code template resulting in a similar hook.) The kernel launch operations and its parameters are copied into the $worker\_queue$ instead of the designated stream. The worker behaviour is described in Algorithm 6. It loops over the $worker\_queue$ and operates by dequeuing an operation, then executing it, and waiting for its completion. The execution first requires the worker to acquire the GPU lock, released once the

operation completes.

Algorithm 5 Kernel Launch hook under the worker strategy

```
1: procedure CookLaunchKernel(func, grid, args, stream)
2:     insert op Execute(func, grid, args) in worker_queue
3: end procedure
```

Algorithm 6 Worker thread implementation

```
1: procedure CookWorkerThread
2:     while true do
3:         op ← head(worker_queue)
4:         pop(worker_queue)
5:         if op is Execute(...) then
6:             acquire GPU_LOCK
7:             insert op Execute(...) in stream
8:             sync on stream
9:             release GPU_LOCK
10:        else if op is Copy(...) then
11:            acquire GPU_LOCK
12:            insert op Copy(...) in stream
13:            sync on stream
14:            release GPU_LOCK
15:        end if
16:    end while
17: end procedure
```

A burst of operations may contain operations other than kernel launches and memory copies, e.g. host callbacks. Those still need to be executed in a FIFO order to maintain the semantic and behaviour of the application (Aspect 7). The worker could support all stream-ordered operations, however this would increase its complexity. Instead, we focus on the *kernel* and *copy* GPU operation types. Other operations must synchronise with the worker, ensuring it has completed all prior operations in the burst before proceeding themselves. This is achieved through the shared code template in Algorithm 7.

Algorithm 7 Operations hook under the worker strategy

```
1: procedure CookOrderedOp(args...)
2:     sync on worker_stream
3:     insert op Op(args...) in stream
4: end procedure
```

Note that upon a kernel launch, we need to copy the list of arguments passed to the kernel for its deferred execution by the worker. This argument list may be allocated on the stack, as is the case for code generated by the GPU compiler. The list may thus have been deallocated when the actual execution occurs. To create the argument copy, our implementation references a list of known kernels in the application. For each kernel, the list holds the number of parameters it requires, their size, and the memory layout of the argument list. The worker strategy currently intercepts calls to the CUDA Runtime kernel registration primitives, `__cudaRegisterFunction`, to create said list. The kernel registration primitives are used by the CUDA runtime to register information about kernels on the host side, notably the kernel name, memory address, and binary code. The kernel registration primitives are undocumented functions of the CUDA Runtime, thus our implementation may not be resilient to CUDA Runtime updates, failing to adhere to Aspect 3. However, the principles of the strategy still hold. The kernel list could be built through an off-line analysis of the application.

VI. Methodology

We aim to evaluate the impact of interference resulting from different applications sharing the GPU, the benefits of temporal access control to mitigate said interference, and the complexity of the hooks. This section presents the methodology used in our evaluation: the considered platform, instrumentation for kernel measurements, collected metrics, and application configurations.

A. Platform Configuration

All experiments were collected on a Jetson AGX Xavier setup with Jetpack version 5.0.2. CUDA version 11.4 was used for all benchmarks. The platform is running Ubuntu 20.04.6 LTS (Kernel 5.10.104-tegra). It is configured under the *MAXN* power profile, all cores are active, and the CPU and GPU frequency are allowed to vary, respectively in the 114MHz-1.3GHz and 1.19GHz-2.27GHz ranges, in response to workload changes or throttling constraints.

B. Instrumentation

We instrument the executed kernels to monitor their behaviour. We consider two methods for instrumentation, application- and kernel-level. Application-level instrumentation collects a trace of CUDA calls, using the Nvidia *nsys* tracing solution, a high-level view of the application execution. *nsys* probes into unmodified applications. Kernel-level instrumentation traces the end-to-end execution of each thread block, using our own instrumentation primitives, for an overview of how each kernel is dispatched and executed. Applications, where possible, are modified to call the instrumentation routines and collect traces. Instrumentation, in spite of our best efforts, will introduce noise in the measurements. However, all runs should suffer comparable noise, under a given instrumentation technique.

C. Benchmarks

We consider two types of benchmarks:

- `cuda_mmult` is an ad-hoc CUDA benchmark, with explicit kernel definitions and calls to the CUDA Runtime. It is provided as a sample of CUDA code by Nvidia. The benchmark is composed of a single burst which repeatedly calls the same matrix multiplication kernel ($300\times$) over the same input data. Measurements are collected for a single run of the benchmark.
- `onnx_dna` is a model-based benchmark, using the ONNX runtime [35] to schedule a DNN model and offload computation to the GPU. It is an industrial case study which performs drone detection and avoidance. Each inference is composed of long bursts with few

synchronisation points. Input data is randomly generated for each inference. Measurements are collected over a 60s sampling period after a 30s warm-up time.

D. Configurations

We consider each application under different configurations to assess the impact of interference, and that of each hook strategy. A configuration is denoted as such:

`bench-isol-strategy`

Where `bench` is one of the benchmarks identified in Section VI-C. `isol` is one of the following, identifying if the benchmark is running in parallel:

- isolation: the benchmark is running in isolation;
- parallel: 2 instances of the benchmark application are running in parallel (mirrored).

The `strategy` modifier indicates the hook strategy used when running the application (or applications): `none` (No hook), `callback`, `synced`, and `worker`. The `cuda_mmult-parallel-synced` configuration is thus the `cuda_mmult` benchmark running in parallel with itself under the synchronised operation strategy.

E. Metrics

Our evaluation aims to assess a number of aspects, notably the impact of the proposed strategies on interference between GPU operations, their impact on the performance of an application in isolation, and their complexity in terms of deployment and maintenance. To that end, we collect and present the following metrics:

- The Normalised Kernel runtime (NET) captures the variation in the execution time of kernels. Large NET ranges indicate respectively high execution time variability, inherent to the kernel or due to interference on shared resources. $NET^i_{k,c}$ for the ith instance of a kerne (k) under a given configuration (c) is computed as the ratio between the observed execution time of the kernel ($ET^i_{k,c}$) and the lowest observed execution time of the kernel under the same configuration:

$$NET^i_{k,c} = \frac{ET^i_{k,c}}{min_j(ET^j_{k,c})} \quad (1)$$

- The Inferences per Second (IPS) is a performance metric for applications. Low IPS indicate a slow-running application, and comparing IPS in isolation and in parallel is indicative of the slow down suffered by the application due to interference. $IPS^t_{a,c}$ is computed as the ratio between the number of completed executions (N) of the application (a) under a given configuration (c) in the measurement interval (t). It is measured by looping over the application under randomised or fixed inputs, counting completed executions at regular intervals (1s).

$$IPS^t_{a,c} = \frac{N^t_{a,c}}{duration(t)} \quad (2)$$

- The number of Lines of Code (LoC) Lines of Code is a metric of code complexity. High LoC is indicative of a large code base and correlates to high development and maintenance costs for an application or a tool. LoC is measured for configuration, hook template, and generated code files using *cloc* [36].

VII. Evaluation

A. Assessing the impact of interference

We first assess the impact of interference on kernels from different applications running on the GPU. To that end, we compare the Normalised Kernel Runtime (NET) for each benchmark `bench` in isolation (`bench-isolation-none`) and in parallel (`bench-parallel-none`), measured with the *nsys* instrumentation tool. The results are presented in Figures 9 to 10 respectively for `cuda_mmult` and `onnx_dna`. Each Figure captures the distribution of NET for each instance of the benchmark across all executed kernels as a boxplot. The box captures the 50% of the samples around the median, the whiskers capture 99% of the data, and outliers in the lowest and highest 0.5% have been omitted for readability.

All benchmarks follow a similar trend where their performance in parallel is impeded by sharing of the GPU with another application. Interference does result in higher kernel execution time variability, as identified by a larger range of values. Some benchmarks, e.g. `onnx_dna`, exhibit an inherent variability of kernel execution times but minor effects of interference on the majority of kernel calls. Interference also results in occasionally large slowdowns, $5.5\times$ for `cuda_mmult` and, very rarely $1200\times$ for `onnx_dna` (less than 0.5% of kernels exceed a $10\times$ slowdown). Outliers for `cuda_mmult` never exceed a $5.5\times$ slowdown, and all policies reduce their occurrence such that less than 0.5% of kernels suffer so. The policies also affect outliers for `onnx_dna`, reducing the maximum observed slowdown from $1200\times$ (`onnx_dna-parallel-none` and`onnx_dna-parallel-callback`) to the same as in isolation around $200\times$ (respectively $800\times$) under `onnx_dna-parallel-synced` (respectively `onnx_dna-parallel-worker`).

Configuration	none	callback	synced	worker
isolation	113	37	67	84
parallel	49	32	25	26

TABLE I: Inference per Second (IPS) achieved by the onnx_dna benchmark for the different configurations

In terms of overall application performance, we consider the IPS achieved by the application, and in particular of `onnx_dna` in Table I (results include all configurations, including the mitigation strategies). In isolation (`onnx_dna-isolation-none`), the application achieves an IPS of 113. This falls down to 49 in the matching parallel configuration (`onnx_dna-parallel-none`). While a decrease of the IPS could be expected as two instances share

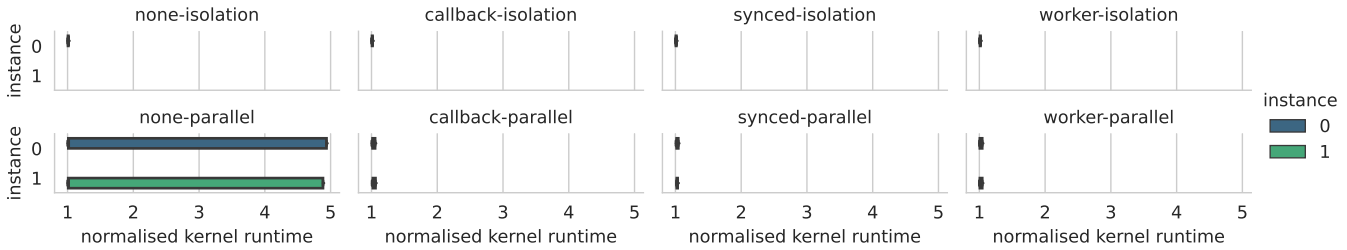


Fig. 9: Distribution of normalised kernel runtimes for the `cuda_mmult` benchmark under all configurations

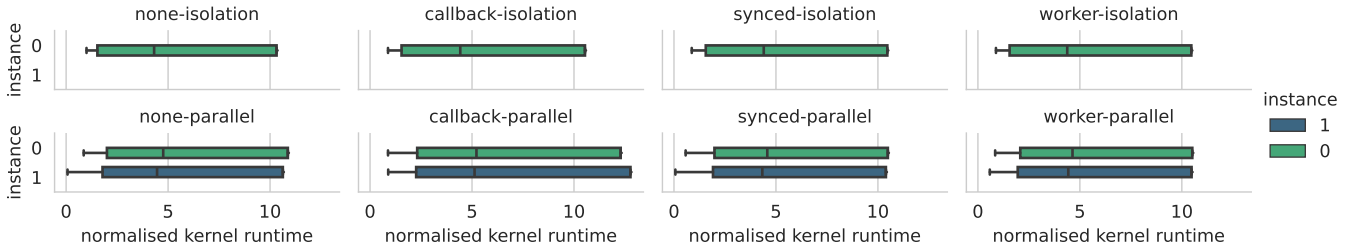


Fig. 10: Distribution of normalised kernel runtimes for the `onnx_dna` benchmark under all configurations

the GPU, it is still slightly more than $2\times$ for only 2 parallel instances. It however does not reflect the highest slowdown suffered by individual kernel calls.

B. Assessing the impact of mitigation

We now assess the impact of the proposed mitigation strategies on the execution of parallel applications. We focus in particular on ensuring the proposed strategies do manage to isolate the execution of GPU operations from different applications. To that end, we instrument the kernel of the `cuda_mmult` benchmark to trace the execution of each executed kernel on the GPU. The results are presented as chronograms in Figure 11. Each chronogram captures the trace of kernel executions, from the beginning of their first executed block (top) to the completion of their last (bottom). Different columns indicate blocks belonging to different instances of the same benchmark.

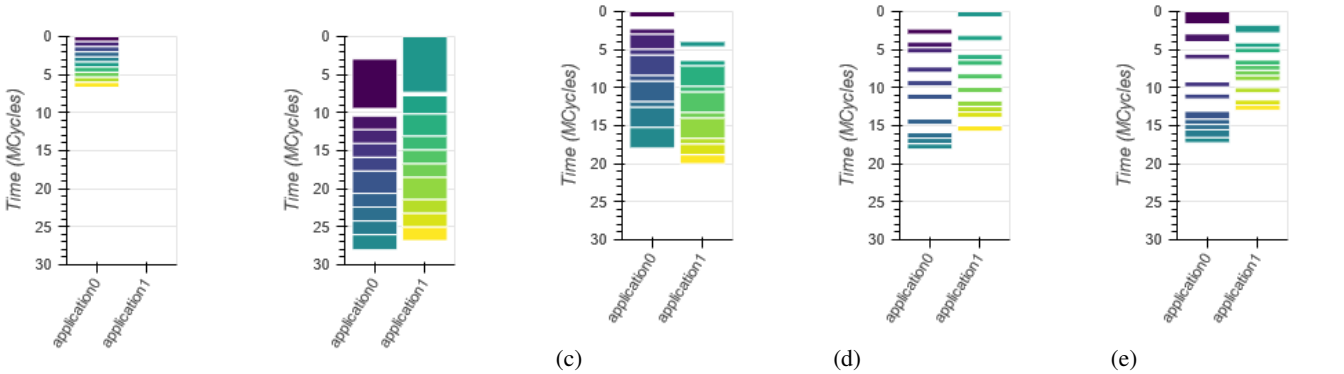
Figure 11 first compares the execution of `cuda_mmult` in isolation and in parallel without any hooked library. We first note that the application suffers a $4\times$ slowdown due to sharing the GPU, taking about 28 Million Cycles to complete over 8 in isolation. This is similar to the slowdown observed in the previous section on normalised kernel runtimes. Some kernels appear to take much longer when their execution overlaps. In practice, the JETSON AGX XAVIER does not allow two applications (and two GPU contexts) to run concurrently; it constantly switches contexts to allow both instances to progress without starvation. As was illustrated in Figure 1, the unknown behaviour of the platform in parallel does not allow us to highlight when and how often such switches occur. Context switches are costly on the GPU as all registers of all SM may need to be saved to memory. Kernels from an application further suffer from cache-related preemption delays as they have to reload useful blocks in the caches evicted by the other.

We also consider the impact of the hook strategies in terms of mitigation in Figure 11. Both the synced and worker strategies do manage to isolate the execution of kernels from different instances of `cuda_mmult`, with no visible overlap between their blocks. The callback strategy however fails to isolate GPU operations (host callbacks possibly cause a GPU context switch but none of the cache polluting effects). All strategies outperform the configuration without mitigation, and they achieve a similar performance with a slight benefit for the worker one. All strategies also outperform a PTB solution (using [26], omitted for the sake of brevity), where both instances were allocated 4 GPU SMs. The PTB implementation did rely on modifications of the application (Aspect 1). The benchmark still suffers a slowdown greater than the number of running instances. The impact of the hook strategies on performance is considered in the next section.

C. Impact of the hook strategies

We now consider the impact of the various hook strategies to mitigate the interference observed in Section VII-A. We repeat the same experiment running each `bench` in isolation and in parallel configurations, but with all instances running under one of the hook `strategy`. The results are presented in Figures 9 to 10 respectively for `cuda_mmult` and `onnx_dna`. The top row captures configurations in isolation, while the bottom one is for benchmark in parallel ones. Each column holds the result for a specific strategy, from left to right: none, callback, synced, and worker.

All benchmarks again follow a similar trend. They still suffer from long running kernels, with worst-case slowdowns



(a) `-isolation-none` (b) `-parallel-none` (c) `-parallel-callback` (d) `-parallel-synced` (e) `-parallel-worker`

Fig. 11: Chronogram of the `cuda_mmult` benchmark execution under the various configurations

similar to the ones observed without mitigation: 5.5× for `cuda_mmult` and 1200× for `onnx_dna`. However these slowdowns are less frequent under all hook strategies, as is most notable for the `cuda_mmult` benchmarks where 99% of executed kernels suffer negligible slowdowns. There is little difference overall in the impact of the various strategies for `cuda_mmult`. For `onnx_dna`, the callback strategy does increase the variability of kernel execution times. Conversely, both the synced and worker strategies manage to reduce the variability introduced by the second instance of the benchmark and the maximum observed slowdown respectively to 200× (close to the maximum in isolation) and 800×. The synced and worker strategies do isolate the execution of individual GPU operations (see Section VII-B). The high slowdowns hint that sharing the GPU during the lifetime of applications even at a high granularity still results in slowdowns. Interference occurs outside the execution of GPU operations, e.g. in shared software resources, or due to cache pollution or costly context switches.

We assess the overall performance of the `onnx_dna` application using the IPS metric in Table I. Like Figure 10, the top row holds the information for the isolation configurations, and each column is a strategy. The various strategies do result in significant slowdown for the benchmark, in isolation and in parallel. The hook strategies result in worse performance than the unmitigated scenario. This may be due to the additional synchronisation between applications and a reduced use of the GPU resources within each instance of the application.

All strategies do introduce additional synchronisation points in the application, potentially after each GPU operation (callback and synced). The callbacks further add a considerable overhead to the application execution. This breaks up the long bursts of `onnx_dna` into smaller ones thus slowing the preparation and scheduling of GPU operations on the host-side. The worker strategy is the less invasive one, as it allows the application to proceed while it schedules synchronise GPU operations itself. However, unmanaged GPU operations (outside copy and execution) still need to synchronise with the worker to guarantee the overall order of operations scheduled on the GPU.

D. Complexity of the hook generation

We assess in the following the complexity of the hook strategies, as a proxy for the resiliency of the method. Simple methods or strategies shall be easier to maintain over time. We first present in Table II the Lines of Code for different artefacts of the toolchain, namely the configuration file and templates used to generate the hooked library. In the absence of hook generation, that code would have to be maintained in whole.

Strategy	Configuration	Templates	Generated code
callback	153	151	6804
synced	153	149	6813
worker	171	1056	8383

TABLE II: Lines of Code (LoC) required and generated for the different strategies

All strategies rely on small configuration files, mostly defining symbols and the templates to generate the related hook definition. The templates LoC accounts for both the common code for the strategy, as well as symbol-specific ones. The worker strategy is the more complex one in both aspects. Additional templates apply to GPU operations managed by or synchronised with the worker queue, thus requiring additional entries in the configuration. The measure also includes all the code related to the deferred worker execution and worker queue management. Overall, this represents at most a thousand hundred lines of code to maintain. But the hook generation results in thousands of lines of code.

We configured the tool to raise an error on calls to all CUDA Runtime methods by default, unless explicitly hooked or ignored. This ensures that an application cannot call methods which may generate unmanaged GPU operations. Such methods need further consideration and an appropriate hook definition. We further distinguish between *implicit* and *unknown* symbols in this category. *Implicit* symbols have no explicit hook or exclusion rule. *Unknown* symbols are ones for which the tool could find no matching declaration. In practice, unknown symbols are not used in our benchmarks. They correspond to variants of CUDA methods where the default CUDA stream is not shared within a GPU context. Their

declaration is generated in the CUDA Runtime headers based on some compilation-time configuration. We have considered workarounds to retrieve the generated definition, e.g. preprocess said headers, to circumvent it, e.g. the trampoline could call C pre- and post hooks, or to map the variants to the original declaration. But we have yet to explore them.

Considering symbols hooked by trampolines, the hook strategies intercept less than 70 methods of the CUDA Runtime library, out of 385. Those are mostly variants of copy (`cudaMemcpy`) and execution (`cudaKernelLaunch`) GPU routines. The worker strategy in addition distinguishes methods which create or depend on synchronisation points to ensure these also synchronise with the worker queue. The same template is used for all of them. While this requires a separate assessment, as to which methods should be included or not in the hooked methods, it only needs to be performed once and then simply updated upon new versions of the library.

VIII. Conclusion

The NVIDIA JETSON AGX XAVIER can deliver high performance and throughput in a small SWaP package, by combining general purpose and specialised cores. However, interference, especially on its GPU, does result in significant slowdowns and execution time variability. The problem is compounded by the closed nature of the platform. We considered the specification and implementation of an access controller to mitigate the interference suffered by applications sharing the JETSON GPU. From the state of the art, we outlined a number of aspects to minimise the impact on applications w.r.t. deployment and development. A second concern was to ease the deployment and maintenance of the method across variants of the platform.

Our approach focuses on ensuring GPU operations from separate applications have exclusive access to the GPU during execution. We proposed various strategies reliant on intercepting GPU operations calls through hooks on the GPU routines. All strategies manage to reduce the impact of GPU interference on operations themselves, with two strategies achieving isolation. Benchmarks such as the ONNX runtime, which benefit from the CPU and the GPU working in tandem, do suffer a performance loss at the application from the additional synchronisation barriers introduced by our approach. The use of a generative approach to generate hooks allows for a small, manageable implementation for all strategies. Changes to the software stack can be accounted for by adding new hook templates or reusing existing ones.

There is still work to be done in the context of interference within the GPU. Even when individual operations' execution is isolated, they suffer from additional delays, related to unknown resources, or effects similar to preemption-related delays. We did not consider interference between the CPU and the GPU, or other accelerators. Similar aspects, at least in terms of transparency and resilience, and methods should benefit to applications. The proposed requirements may be also lifted to reduce the impact of the method on the performance of applications. Operations from the same or different applications may notably be allowed to run concurrently provided they are known to not interfere with each other. Applications or implementations tailored towards reduced execution time variability, e.g. relying on cooperative methods, should also be considered once a better understanding of the platform an applications has been established. Finally, we need to consider the application of this work to support a platform-level resource manager such as [12]–[15].

Acknowledgement

The work presented in this paper is part of the PHYLOG 2 project supported by the Directorate General of Civil Aviation (DGAC). It is funded by the French government through the France Relance program, based on the funding from the European Union through the NextGenerationEU program.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools," *ACM Transactions Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[2] R. Wilhelm and J. Reineke, "Embedded systems: Many cores - many problems," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, 2012, pp. 176–180.

[3] J. Berdajs and Z. Bosnić, "Extending applications using an advanced approach to dll injection and api hooking," *Software: Practice and Experience*, vol. 40, no. 7, pp. 567–584, 2010. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.973

[4] NVIDIA, *NVIDIA Jetson AGX Xavier Series System-on-Module: DATA SHEET*, NVIDIA Corporation, Santa Clara, California, Aug. 2022.

[5] ——, "Nvidia heterogeneous computing on cuda platforms," https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#heterogeneous-computing, 2022, accessed: 2022-11.

[6] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826

[7] M. Adalbert, T. Carle, and C. Rochange, "PasTiS: building an NVIDIA Pascal GPU simulator for embedded AI applications," in *11th European Congress on Embedded Real-Time Systems (ERTS 2022)*. 3AF Midi-Pyrénées: the French Society of Aeronautic and Aerospace and SEE : the French Society for Electricity, Electronics, and Information & Communication Technologies, Jun. 2022. [Online]. Available: https://ut3-toulouseinp.hal.science/hal-03684680

[8] N. M. Otterness, "Developing Real-Time GPU-Sharing Platforms for Artificial-Intelligence Applications," Ph.D. dissertation, 2022.

[9] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 213–225.

[10] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems," in *ECRTS*, 2018.

[11] T. Amert, "Enabling Real-Time Certification of Autonomous Driving Applications," Ph.D. dissertation, 2021, aAI28650154.

[12] P. Bieber, F. Boniol, M. Boyer, E. Noulard, and C. Pagetti, "New Challenges for Future Avionic Architectures." *Aerospace Lab*, no. 4, pp. p. 1–10, 2012. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01184101

[13] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. Romero, and C. Scordino, "Resource Management on Multicore Systems: The ACTORS Approach," *IEEE Micro*, vol. 31, no. 3, pp. 72–81, 2011.

[14] G. Fohler, G. Gala, G. Pérez, Daniel, and P. Claire, "Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator," in *9th European Congress on Embedded Real Time Software and Systems (ERTS'18)*, 2018.

[15] G. J. Gala, "Resource Management for Real-time and Mixed-Critical Systems," doctoral thesis, Technische Universität Kaiserslautern, 2022. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:hbz:386-kluedo-67099

[16] NVIDIA Corporation, "Multi-Instange GPU (MIG) — NVIDIA," https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2022, accessed: 2023-02-16.

[17] J. Bakita and J. H. Anderson, "Hardware Compute Partitioning on NVIDIA GPUs*," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023, pp. 54–66.

[18] T. Amert, Z. Tong, S. Voronov, J. Bakita, F. D. Smith, and J. H. Anderson, "TimeWall: Enabling Time Partitioning for Real-Time Multicore+Accelerator Platforms," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 455–468.

[19] W. Ali and H. Yun, "Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 19:1–19:22. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8983

[20] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A Framework for Real-Time GPU Management," in *2013 IEEE 34th Real-Time Systems Symposium*, 2013, pp. 33–44.

[21] H. Kim, P. Patel, S. Wang, and R. (Raj) Rajkumar, "A server-based approach for predictable GPU access with improved analysis," *Journal of Systems Architecture*, vol. 88, pp. 97–109, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762117303880

[22] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-Based Scheduling for GPU with Preemption Support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 119–130.

[23] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "Rgem: A responsive gpgpu execution model for runtime engines," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 57–66.

[24] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and Exploiting Flexible Task Assignment on GPU through SM-Centric Program Transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. Association for Computing Machinery, 2015, p. 119–130. [Online]. Available: https://doi.org/10.1145/2751205.2751213

[25] C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan, and D. Qian, "SM-Guard: A Flexible and Fine-Grained Resource Management Framework for GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2849–2862, 2018.

[26] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional GPUs: Software-Based Compute and Memory Bandwidth Reservation for GPUs," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 29–41.

[27] N. Corporation, "GPU Management and Deployment: Multi-Process Service," https://docs.nvidia.com/deploy/mps/index.html, 2022, accessed: 2023-02-16.

[28] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 372–383. [Online]. Available: https://doi.org/10.1145/3352460.3358307

[29] A. Mentone, D. Di Luccio, L. Landolfi, S. Kosta, and R. Montella, "Cuda virtualization and remoting for gpgpu based acceleration offloading at the edge," in *Internet and Distributed Computing Systems*, R. Montella, A. Ciaramella, G. Fortino, A. Guerrieri, and A. Liotta, Eds. Cham: Springer International Publishing, 2019, pp. 414–423.

[30] J. Duato, A. J. Pena, F. Silla, J. C. Fernandez, R. Mayo, and E. S. Quintana-Orti, "Enabling cuda acceleration within virtual machines using rcuda," in *Proceedings of the 2011 18th International Conference on High Performance Computing*, ser. HIPC '11. USA: IEEE Computer Society, 2011, p. 1–10. [Online]. Available: https://doi.org/10.1109/HiPC.2011.6152718

[31] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, "Ds-cuda: A middleware to use many gpus in the cloud environment," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 1207–1214.

[32] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An Open Toolbox for Adaptive WCET Analysis," in *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP*, ser. Lecture Notes in Computer Science, S. L. Min, R. G. P. IV, P. P. Puschner, and T. Ungerer, Eds., vol. 6399. Springer, 2010, pp. 35–46.

[33] C. Ferdinand and R. Heckmann, "aiT: Worst-case execution time prediction by static program analysis," in *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*. Springer, 2004, pp. 377–383.

[34] Yury Gribov, "Implib.so," 2023. [Online]. Available: https://github.com/yugr/Implib.so

[35] The Linux Foundation, "The onnx runtime," 2023. [Online]. Available: https://onnx.ai/

[36] Al Danial, "cloc," 2023. [Online]. Available: https://github.com/AlDanial/cloc