Novel Methodologies for Predictable CPU-To-GPU Command Offloading

Roberto Cavicchioli
Università di Modena e Reggio Emilia, Italy
roberto.cavicchioli@unimore.it

Nicola Capodieci
Università di Modena e Reggio Emilia, Italy
nicola.capodieci@unimore.it

Marco Solieri
Università di Modena e Reggio Emilia, Italy
ms@xt3.it

Marko Bertogna
Università di Modena e Reggio Emilia, Italy
marko.bertogna@unimore.it

Abstract

There is an increasing industrial and academic interest towards a more predictable characterization of real-time tasks on high-performance heterogeneous embedded platforms, where a host system offloads parallel workloads to an integrated accelerator, such as General Purpose-Graphic Processing Units (GP-GPUs). In this paper, we analyze an important aspect that has not yet been considered in the real-time literature, and that may significantly affect real-time performance if not properly treated, i.e., the time spent by the CPU for submitting GP-GPU operations. We will show that the impact of CPU-to-GPU kernel submissions may be indeed relevant for typical real-time workloads, and that it should be properly factored in when deriving an integrated schedulability analysis for the considered platforms.

This is the case when an application is composed of many small and consecutive GPU compute/copy operations. While existing techniques mitigate this issue by batching kernel calls into a reduced number of *persistent kernel* invocations, in this work we present and evaluate three other approaches that are made possible by recently released versions of the NVIDIA CUDA GP-GPU API, and by Vulkan, a novel open standard GPU API that allows an improved control of GPU command submissions. We will show that this added control may significantly improve the application performance and predictability due to a substantial reduction in CPU-to-GPU driver interactions, making Vulkan an interesting candidate for becoming the state-of-the-art API for heterogeneous Real-Time systems.

Our findings are evaluated on a latest generation NVIDIA Jetson AGX Xavier embedded board, executing typical workloads involving Deep Neural Networks of parameterized complexity.

2012 ACM Subject Classification Computer systems organization → System on a chip; Computer systems organization → Real-time system architecture

Keywords and phrases Heterogeneous systems, GPU, CUDA, Vulkan

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.22

Supplement Material ECRTS 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.1.4`

Funding Work supported by the CLASS Project, European Union's H2020 G.A. No. 780622.

31st Euromicro Conference on Real-Time Systems (ECRTS 2019).
Editor: Sophie Quinton; Article No. 22; pp. 22:1–22:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Modern high-performance embedded platforms feature heterogeneous systems, where a multicore CPU host is integrated with one or more parallel accelerators. These platforms are used to run cyber-physical real-time systems, requiring workload-intensive tasks to be executed within given deadlines. While there are many works addressing the schedulability analysis of real-time tasks on multi-core systems [4], there is an increasing interest in understanding and refining the adopted task models to better capture the timing behavior of real-time workloads in practical scheduling settings on heterogeneous embedded platforms. Thanks to the dramatic improvement of performance-per-Watt figures over multicore CPUs, GP-GPU (General Purpose GPU) computing is a widely adopted programming model to perform embarrassingly parallel computations in both embedded and discrete devices. Typical usage scenarios of heterogeneous embedded platforms are found in the domain of autonomous driving, avionics and industrial robotics, presenting important design challenges due to the safety-critical nature of these domains [12, 25].

To address this challenge, we first briefly summarize how computations are orchestrated in an embedded platform that features a multicore CPU (host) and an integrated GPU accelerator (device).

1.1 CPU-to-GPU interaction

The interactions between CPU-threads and the GPU are described through different APIs (Application Programming Interfaces) that allow the host to control the offloading of data and workload description, to be later executed by the parallel processing units within the GPU. A way to abstract a higher-level description of the processing units of a GPU is to think of it as being composed of two engines: the Execution Engine (EE) for computation, and the Copy Engine (CE) for high throughput DMA data transfers.

The Execution Engine enables a high level of SIMD parallelism by exploiting hundreds of ALU pipelines, that are usually grouped into processing clusters. Within each of these processing clusters, a hardware scheduler dispatches small groups of threads in lockstep to the ALU pipelines. A well-known API for modelling such interactions is CUDA [24], a proprietary API developed by NVIDIA. In such a model, ALU pipelines are called *CUDA cores*, and processing clusters are known as *Streaming Multiprocessors (SMs)*. SMs schedule *warps*, i.e. groups of 32 lockstep threads. Considering different GP-GPU APIs (such as OpenCL), the terminology may slightly vary, but these higher-level concepts remain unchanged. On the host side, the application developer has the responsibility to trigger data movements and the workload to be executed to the Execution Engine of the GPU by using API function calls. This operation is known as *kernel invocation*. When many workloads have to be dispatched within short time intervals, the time spent on the CPU for these offloading operations becomes non-negligible, and it should be properly accounted for when pursuing a sound schedulability analysis.

A task may be modeled using a Directed Acyclic Graph (DAG), where nodes represent workloads executing on a CPU core or on a GPU engine, while edges are precedence constraints between nodes. In a typical application, CPU nodes are constantly interleaved with GPU nodes. Despite its intrinsic parallelism, the GPU is considered as a single resource. This is due to its lockstep nature that limits task-level parallelism, especially for embedded GPUs with a smaller number of SMs. In other words, a GPU node represents a kernel occupying all the processing units of the considered GPU engine.

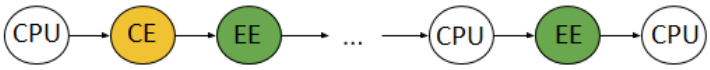


Figure 1 Sequences of sub-tasks of a CPU core submitting work to both compute and copy GPU engines. White nodes are CPU offloading operations, yellow and green nodes represent GPU copy and compute operations.

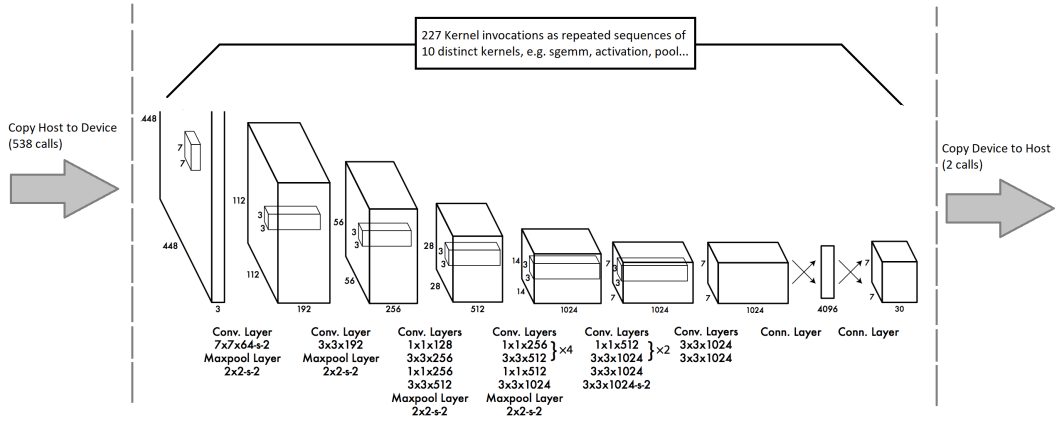


Figure 2 Data transfers and Layer-to-kernel invocations over the YOLO network [26].

Moreover, CPU nodes have to account for a variable cost of GPU commands' submission time, related to the translation of API function calls and to trigger the GPU operations, independently of this operation being synchronous (i.e., the CPU shall wait for the offloaded GPU kernel to complete before continuing) or asynchronous (i.e., the CPU may proceed to the next node without needing to wait for the offloaded kernel to complete).

The example depicted in Figure 1 shows a submission-intensive workload, in which a variable number of CPU-nodes offload-operations interleave among different kernels and copy invocations. Even if the represented chain may appear simplistic w.r.t. a fully-fledged DAG, it is not far from real world applications. In Figure 2, we profile CPU and GPU traces of a single iteration of the inference operation on a widely used YOLOv2 neural network [27]. The inference over YOLO's 28 network layers requires heavy data movements and the invocation of a very large number of kernels, each preceded by a CPU offloading phase. Using *nvprof*, the NVIDIA CUDA profiler, we discovered that kernels and copies are invoked as a sequence of implicitly synchronized operations.

There are two negative effects of interleaving CPU offloads and GPU workloads:

1. the completion of a submission node is a mandatory requirement for the activation of the subsequent GPU node. This implies that jitters or delays in the completion of the CPU node will postpone the actual release of the GPU sub-task, increasing the complexity of the schedulability analysis.
2. In case of shorter GPU kernels, CPU offloading becomes a performance bottleneck.

As a consequence of these considerations, we aim at analyzing and minimizing the time spent by the CPU for submitting commands related to GPU operations.

In the literature, these issues have been addressed using *persistent kernels* [15], i.e. a reduced number of kernel invocations that delegates the responsibilities of future workload invocations to GPU threads. We hereafter discuss the limitations of CUDA persistent

threads, showing how novel features of the latest CUDA SDK release allows the programmer to mitigate the problem of submission-intensive operations. We characterize novel submission methods such as CUDA Dynamic Parallelism (CDP) and pre-constructed CUDA Graphs, and investigate a novel API for GPU programming called *Vulkan*. Vulkan is a next generation *bare-metal* API allowing a dramatic improvement in the control that the application programmer has over the interaction between the application layer and the GPU driver. Aspects that were commonly well hidden by traditional APIs are completely exposed to the application programmer: fine grained synchronization, low level mechanisms of memory allocation and coherency, state tracking and commands validation are all essential parts in direct control of the application developer. According to the Khronos Group, the industrial consortium that released and maintains the Vulkan technical specification [21], this increased level of control enhances applications performance and predictability due to substantially reduced CPU-to-GPU driver interaction. We will prove that this is indeed the case, especially for submission intensive workloads, making Vulkan a very promising open standard for real-time systems executing on heterogeneous platforms.

We experimentally validate our considerations executing representative kernels on an NVIDIA Jetson AGX Xavier[1], a recently released embedded development platform, featuring a Tegra System on Chip (Xavier SoC) composed of 8 NVIDIA Carmel CPU Cores (superscalar architecture compatible with ARMv8.2 ISA) and an integrated GPU based on the NVIDIA Volta architecture with 512 CUDA cores. On this platform, we executed a parametrized version of a Deep Neural Network, varying the number and position of convolutional and fully-connected layers.

The remainder of the paper is structured as follows. Section 2 presents a recent overview on the challenges of achieving timing predictability on GPU-accelerated SoCs, with specific emphasis on the impact of submission procedures. In Section 3, we describe the relevant CUDA features that have been introduced in the latest SDK versions, and how we leveraged them in our experiments. Section 4 provides a brief explanation of the Vulkan API peculiarities and related programming model, while Section 5 introduces our open source Vulkan Compute wrapper, that we implemented to simplify the generation (and easy reproduction for artifact evaluation) of the experiments. Experimental settings and related discussion on the results are provided in Sections 6 and 7. Section 8 concludes the paper with further remarks and proposals for future work.

2 Related Work

The recent literature on GPU-based real-time systems identified multiple sources of unpredictability, related to GPU scheduling [17], CPU interactions with the GPU API/driver [13, 14] and memory contention [28, 2, 9]. In this paper, we focus on minimizing driver interactions by exploiting recently proposed instruments to pre-record and pre-validate the GPU commands that characterize CPU-to-GPU offloading operations. Previously, a way to minimize CPU offloading times was to batch compute kernel calls into a single or reduced number of command submissions. This approach based on *persistent kernel* proved to have beneficial effects in terms of average performance [5, 15, 6, 16] and for obtaining a higher degree of control when scheduling blocks of GPU threads [31, 10]. The problem with persistent kernels is that they often require a substantial code rewriting effort and they are not able to properly manage Copy Engine operations mixed with computing kernels. Moreover, a persistent

[1] `https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit`

kernel approach would require managing GPU kernels with different safety criticality levels within a single application context, posing a significant threat to the overall system safety integrity assessment.

Our work tries to overcome these aspects by exploiting more recent methodologies, such as CUDA Dynamic Parallelism (CDP) and CUDA graphs. CDP allows the programmer to minimize kernel launch latencies in a similar way as a persistent kernel, without the need of substantial code rewriting and still allowing the different GPU tasks to reside in different process address spaces. CDP has been introduced in the CUDA SDK since version 5.0 (released in 2014).

More recently, the latest release of the CUDA SDK (version 10, released in late 2018) introduced the concept of CUDA graphs: a CUDA graph allows the developer to construct a graph of copy, compute and even host sub-tasks together with their related dependencies in advance. Then, at runtime, the offloading operation only involves a single call to the CUDA runtime driver. As pointed out in the introductory section, we model CPU and GPU interactions as sequences of operations that can be generalized to a DAG: this is similar to the OpenVX programming interface [32].

To the best of our knowledge, we are exploring for the first time not only these novel CUDA functionalities, but also the real-time capabilities of the Vulkan API, a recently released open standard for GPU programming. The work on novel low level and low overhead GPU APIs started in late 2013, with Apple's *Metal* and AMD's *Mantle* APIs representing the first concrete efforts toward the implementation of such a novel bare-metal approach. While the first one is a proprietary effort limited to Apple devices, Mantle development stopped around mid 2015, but served to the Khronos group as a base to start developing the Vulkan specifications (Version 1.0 was released in February 2016). In parallel, Microsoft's *Direct3D 12* saw its release at the same time as Windows 10. All these APIs share a common programming model that imposes minimal driver overhead as a result of a more explicit and direct control over GPU related commands.

The problem of minimizing CPU-GPU driver overhead is an issue that the Khronos group considered also for newer *OpenGL* releases. The so-called *AZDO* methodology (Approaching Zero Driver Overhead) aimed at minimizing the driver interaction by batching draw calls, state changes, reduced texture bindings and data factorization into command lists to be submitted within a reduced number of OpenGL API calls. As we already highlighted in the previous section, we are more interested on reducing submission latencies to improve the predictability of real-time applications. Hence, our work will focus on measuring how effective is the compute pipeline of Vulkan compared to traditional and to recently released CUDA-based submission models. We believe this document may be an instructive reference for real-time practitioners looking for more predictable settings and programming models on modern heterogeneous embedded platforms.

3 Alternative submission models for CUDA

In this section we provide an in-depth explanation of the approaches we tested for minimizing CPU-GPU driver interactions with relation to NVIDIA CUDA. These mechanisms are made possible by recent releases of the CUDA SDK, and they describe the motivation and typical usage scenarios of CDP and CUDA graphs.

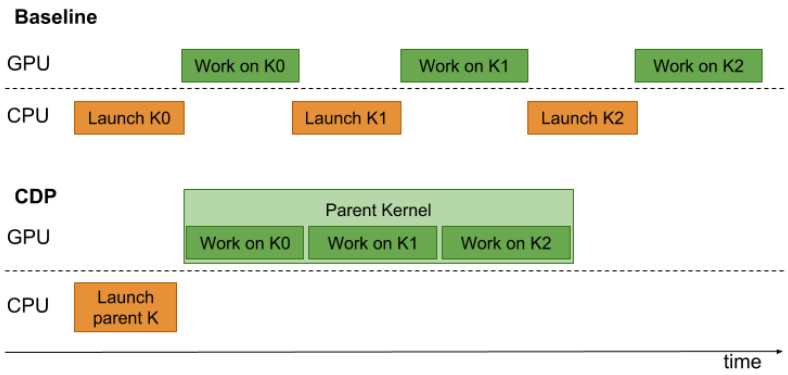


Figure 3 Timelines of the baseline CUDA submission model compared to CDP. Note that K0, K1 and K2 implementations are identical in both cases and the launch configuration of Ki depends on the result of $K(i-1)$.

3.1 CDP

GPU kernel invocations are characterized by a *launch configuration*. A launch configuration describes the degree of parallelism of the GPU computation. Implementation-wise, this implies that the programmer has to decide the size and number of CUDA blocks for the specific launch configuration. A CUDA block is a collection of concurrent threads that share the same L1 cache and/or scratchpad memory inside a single SM. The size of a block is the actual number of threads within each block. Trivially, if a GPU task is an algorithm in which each step presents a different parallelization degree, then the programmer has to break down this task into multiple kernel launches, each with a potentially different launch configuration. As discussed in the previous section, this can easily translate in additional overhead on the CPU side for performing offloading operations, especially when launch configurations between different launches are not known a priori.

Nested parallelism is a known construct in the parallel programming literature that allows the programmer to express variable parallelism as a series of nested kernel invocations with adaptive launch configurations. Nested parallelism has therefore the potential to fully exploit the GPU parallelism for workloads involving variable depth recursion (e.g: sorting [23], graph exploration [34], clustering algorithms [1] ...). In CUDA, this is implemented by having a parent kernel invoking child kernels with a varying block/thread count, without involving the CPU for launching the child kernels.

An example scenario is depicted in Figure 3, where a baseline offloading (top) is compared against a CDP sequence of invocations (bottom). In the depicted corner case, the CPU submission time is comparable to the GPU execution time for the considered kernels. In the baseline scenario, there is an interleaved work between host and GPU, where the CPU thread has to adjust the launch configuration for each subsequent kernel. A similar situation happens even for asynchronous submissions. Conversely, with the CPD scenario the CPU is only active during the launch of the parent kernel, while subsequent kernel invocations are managed by the device, allowing the concurrent execution of GPU workloads [3, 24]. This drastically reduces the response time of the entire task. Another interesting advantage is related to the simplification of the related worst-case response-time analysis, as the task graph to consider has significantly less nodes (CPU sub-tasks) and edges (CPU-GPU interactions). However, we will see in the evaluation section that CDP presents some limitations when data is requested by the CPU through the Copy Engine and when the sequence of kernels is not characterized by variable parallelism.

3.2 CUDA Graphs

CUDA graphs are the most recent innovation to the CUDA runtime set of functions. Graphs are a step forward compared to the more traditional CUDA streams: a stream in CUDA is a queue of copy and compute commands. Within a stream, enqueued operations are implicitly synchronized by the GPU so to execute them in the same order as they are placed into the stream by the programmer. Streams allow for asynchronous compute and copy, meaning that the CPU cores dispatch commands without waiting for their GPU-side completion: even in asynchronous submissions, little to no control is left to the programmer with respect to when commands are inserted/fetched to/from the stream and then dispatched to the GPU engines, with these operations potentially overlapping in time.

Graphs improve on this approach by allowing the programmer to construct a graph of compute, host and copy operations with arbitrary intra- and inter-stream synchronization, to then dispatch the previously described operations within a single CPU runtime function. Dispatching a CUDA graph can be an iterative or periodic operation, so to implement GPU-CPU tasksets as periodic DAGs. This aspect represents an appealing mechanism for the real-time system engineer. Legacy CUDA applications built on streams can be converted to CUDA graphs by capturing pre-existing stream operations, as it is shown in Listing 1.

In Listing 1 a baseline implementation of asynchronous offloading of copy, compute and CPU-side operations is shown (lines from 1 to 7): while the CPU is still inserting operations within the stream, the GPU fetches previously inserted commands on behalf of the GPU engines. This mechanism improves the average performance for long GPU-side workloads, but it is difficult to precisely model this CPU-GPU interaction for scheduling purposes. Moreover, the CPU can act as a bottleneck in case of a long sequences of small GPU operations. Also, in case of dispatching periodic work, CPU-GPU interaction will be repeated and the timing cost for each command submission and validation is bound to increase.

Lines from 9 to 15 show how to capture the same stream operations of the baseline approach to construct a CUDA graph. Lines from 18 to 32 show a graph construction that is equivalent to the previous methodology, but nodes and their dependencies are explicitly instantiated instead of being inferred from pre-existing stream operations. When building a graph, no operations are submitted to the GPU. This allows the developer to describe in advance complex work pipelines. Work is then dispatched with a single or periodic call to *cudaGraphLaunch* (lines 50 and 58). In the experimental section of this paper, we will show how submission latencies and CPU-GPU interaction timeline vary by exploiting these novel CUDA constructs.

4 The Vulkan API

Although Vulkan is defined as a graphics and compute API, the Vulkan model is agnostic to which of these two pipelines will be mostly used within an application. Initial benchmarks on the Vulkan API are related to graphics applications, hence measuring the performance of the same application with a Vulkan renderer and over an OpenGL/OpenGLES or Direct 3D 11 renderer [29]. One of the recent fields of applications that has been shown to provide sensible benefits from this new API paradigm is Virtual Reality (VR), due to the very stringent latency requirements required to mitigate motion/cyber sickness for users wearing VR-enabled devices [30]. We instead investigate the possibility of exploiting Vulkan to minimize driver interactions for allowing a more predictable response time of computing kernels. To the best of our knowledge, we are the first to propose the adoption of this newer generation API for real-time systems. The following overview of the Vulkan API from the programmer point of view can be followed on Figure 4.

Listing 1 CUDA baseline streams – node graphs and stream capture.

```
void baseline(cudaStream_t &s){
 cudaMemcpyAsync(....,s);
 ...
 cudaStreamAddCallback(s,..,cpuFunction);
 kernelCall<<<...,s>>>(...);
...
}

cudaGraph_t cudaGraphFromStreamCapture(&s){
 cudaGraph_t graph;
 cudaStreamBeginCapture(s);
  baseline(&s); //no modifications to baseline
 cudaStreamEndCapture(s,&graph);
 //stream operations are not yet submitted.
}

cudaGraph_t cudaGraphCreation(){
 cudaGraph_t graph;
 //node pointers
 cudaGraphNode_t *memcpyNodes, *kernelNodes, *hostFuncNodes;
 //node params
 cudaKernelNodeParams *memcpyParams, *kerNodeParams,
    *hostNodeParams;
 //define params for all the previously declared nodes
 //(addesses, function names etc...)
 cudaGraphCreate(&graph,0);
 //for each host/device node and respective params ...
 cudaGraphAdd<Memcpy/Kernel/Host>Node
 (&nodePtrs, graph, ..., ..., &nodeParams);
 //first param: node ptrs, second and third: depedencies
 //info, forth: node params.
 //No stream operations are submitted to the GPU
 }

 void mainPrepareAndOffload(){
  cudaStream_t s, sGraph;
  cudaStreamCreate(&s); //regular stream
  cudaStreamCreate(&sGraph); //stream for graph launch
  cudaGraphExec_t graphExec; //graph execution data structure

  //enqueue in stream s and launch with baseline behaviour:
  baseline(&s);
  //wait for finish
  //ALTERNATIVE METHOD 1:
  //create and define graph from pre-existing stream
  cudaGraph_t graph0 = cudaGraphFromStreamCapture(&s);
  //graph instantiation
  cudaGraphInstantiate(&graphExec, graph0, ....);
  //and launch
  cudaGraphLaunch(graphExec, sGraph);
  //wait for finish
  //ALTERNATIVE METHOD 2:
  //create and define graph from node structures
  cudaGraph_t graph1 = cudaGraphCreation();
  //graph instantiation
  cudaGraphInstantiate(&graphExec, graph1, ....);
  //and launch
  cudaGraphLaunch(graphExec, sGraph);
  // ...
 }
```

The improved control over the thin driver layer exploited by a Vulkan application comes at the cost of a significantly increased implementation complexity. While in the CUDA runtime API establishing a context occurs transparently when the first CUDA API call is performed, a Vulkan context involves the explicit creation of a *vkInstance* object[2]. Such object stores global states information at application level. A *vkInstance* is created by default with no added debug/validation layers: in order to minimize the impact of driver level runtime checks and hidden optimizations (which are always performed in traditional APIs, such as CUDA), the application programmer has to explicitly add/activate third-party API layers at *vkInstance* creation time.

The layer-based mechanism is also one of the novel features enabled by these next generation APIs. Once the instance is created, the programmer retrieves a handle to the physical device(s) to use: from a single physical device (*vkPhysicalDevice* object), one or more logical devices (as software constructs able to interface with the chosen physical device) are created. From the logical device, memory for a *vkCommandPool* is allocated: a command pool in Vulkan is used to amortize the cost of resource creation across multiple command buffers. Finally, the programmer has to create one or more command buffers (*vkCommandBuffer*), which are objects that allow storing commands to be later sent to the GPU via a *vkQueue*. A *vkQueue* is the Vulkan abstraction of a queue of commands, with each set of created queues belonging to specialized family of capabilities (compute, graphics, data transfer etc...). A *vkQueue* is roughly the equivalent of a CUDA stream, but with a much finer granularity in terms of CPU-GPU and intra/inter queue synchronizations.

Compute kernels (more specifically, *Vulkan compute shaders*) are stored in Khronos's SPIR-V (Standard Portable Intermediate Representation [19]) binary cross-compatible format and compiled at runtime into device specific code (*vkShaderModule*). Then, the programmer has to explicitly describe how resources are bound to the compute shader, i.e. how input, output and constant buffers are mapped to the shader arguments. Vulkan *descriptor sets* are allocated within *descriptor pools* and further specified thorough *descriptor layouts*. This allows the developer to efficiently re-use the same bindings in different compute shaders.

For what refers to kernel arguments, constants are specified and updated through *Vulkan push and specialization constants*. Associating constants and descriptor layouts to one or more compute shaders occurs at pipeline creation time. During this phase, different pipeline stages are defined to allow associating different *vkShaderModule*s to different pre-allocated descriptor sets. The pipelines are then "baked" into static Pipeline State Objects (PSOs) for later use within a command buffer recording phase. Recording a command buffer allows the programmer to switch different pre-constructed PSOs, so to set in advance a plurality of compute kernels, each with its own descriptor set(s)/layout(s). Recorded commands are then executed when calling the *vkQueueSubmit* function. Once such function is called, little to no other driver interaction is necessary.

This long sequence of operations happens behind a context creation and subsequent kernel invocations in a Vulkan application, and it gives a very plausible overview of the lower-level details that are instead transparently managed in CUDA applications. The hidden handling of these aspects in the closed CUDA implementation makes it less suitable for predictable settings. Instead, we argue that *the explicit handling allowed by Vulkan may make this API much more suitable for real-time systems*, especially considering that aspects like bindings, sequence of kernel invocations and launch parameters are known in advance for predictable real-time applications. The Vulkan model is thus a perfect fit in this setting, since all these parameters are taken into account in pre-compiled structures such as PSOs.

[2] We use the term *object* for brevity, but this has nothing to do with object oriented programming: being the Vulkan client driver implemented in C, such objects are actually non dispatchable handles to C *structs*.

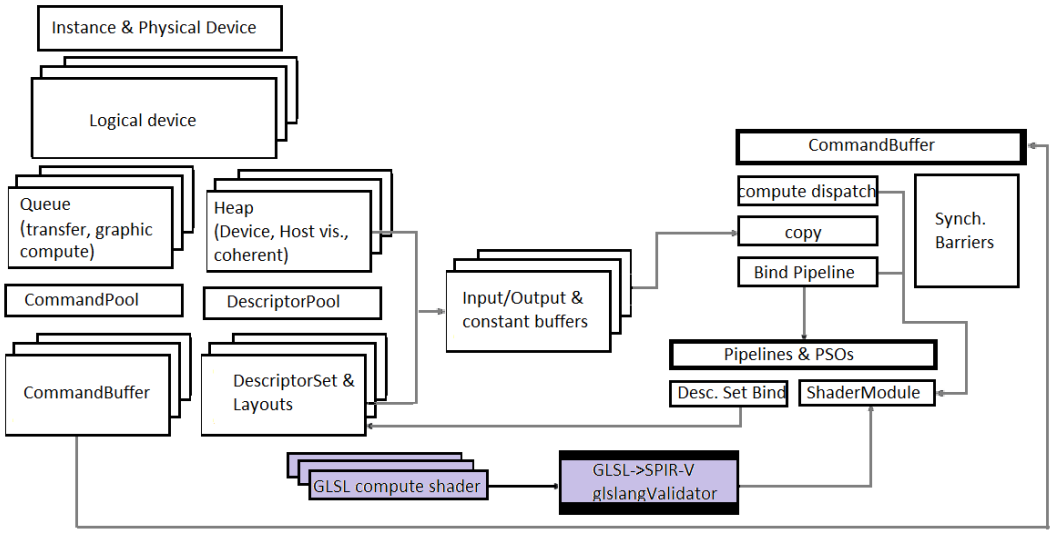


Figure 4 Schematic representation of the operations and modeling artifacts involved in a simplified Vulkan Compute pipeline. Shaded elements are external to the Khronos Vulkan specifications.

5 VkComp: an open-source wrapper for the Vulkan predictable compute pipeline

One of the strongest point of the CUDA model is its ease of use, which, as mentioned, is substantially better than Vulkan. We therefore decided to create a C++ wrapper to the Vulkan API that easily allows setting up operations such as context creation, buffer allocations, memory transfers and kernel invocations, to make these operations as easy as they were using the CUDA runtime API. Our wrapper allows the developer to instantiate pipeline creation blocks and command buffer recording blocks. These functionalities (roughly similar to CUDA graphs), are responsible for creating in advance reusable resources such as PSOs and pre-recorded command buffers. We plan to release this wrapper as an open-source contribution.

5.1 Wrapper structure

Our C++ wrapper to the Vulkan API exposes the functionalities described by the interface class represented in Figure 5. The figure shows the member functions are going to be called within a pipeline creation block, as opposed to command buffer recording blocks and application initialization.

In our experiments, we only use a single command buffer enqueued in a single queue. A generic application composed of many kernel invocations and multiple buffers can be summarized as shown in Pseudo-Algorithm 1.

In the pipeline creation block, a compute kernel is specified together with its launch configuration (set as specialization constants) and descriptor layouts. Argument buffers are pointers to SSBOs (Shader Storage Buffer Objects), whereas constants are specified as Vulkan *push constants*. Recording a command buffer allows us to order the sequence of kernel invocations (by selecting previously baked PSOs), data movements between CPU-GPU (any direction) and constant values updates.

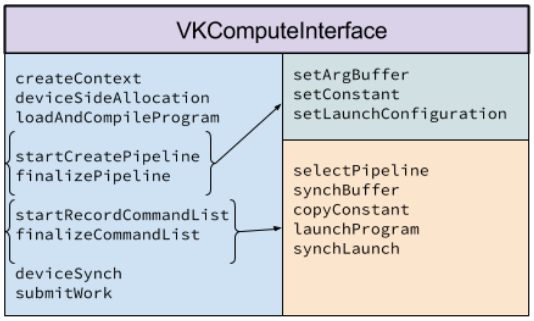


Figure 5 Vulkan based compute interface describing the exposed functionalities of our model.

Algorithm 1 Pseudocode of our VkComp Vulkan wrapper example application.

{application initialization:}
createContext
for each buffer **do**
  buffer ← *deviceSideAllocation*
end for
{pipeline creation block}
for each kernel **do**
  program ← *loadAndCompileProgram*
  startCreatePipeline(program)
    setArgBuffer(*buffer...*) for each related buffer
    setConstant for each related constant
    setLaunchConfiguration
  PSO ← **finalizePipeline**
end for
{Record command list block:}
startRecordCommandList
  for each PSO **do**
    selectPipeline(*PSO*) for each related PSO
    synchBuffer(*buffer*) for each related buffer
    copyConstant for each related constant
    launchProgram(*kernel*)
  end for
cmdbuf ← **finalizeCommandList**
submitWork(*cmdbuf*)
deviceSynch

5.2 Buffers and programs

At context creation, a persistently mapped staging buffer is allocated. Its allocation size is large enough to contain all the data used by the application. This persistently mapped buffer is a driver-owned CPU-side buffer in which we can store buffer data to be later sent to the GPU. This allows the programmer to avoid mapping and unmapping buffers from CPU-only visible addresses to GPU-visible memory areas. An allocation table takes care of sub-allocation segmentation. In our interface, this is transparent to the application designer.

A compute kernel (called *program* in Figure 5 and Algorithm 1) is written in GLSL[3] [20] (OpenGL Shading Language) and translated into SPIR-V using the *glslangValidator* program that is provided by Valve's LunarG Vulkan SDK[4]. Then, a *vkShaderModule* (see section 5) is created and bound to a pipeline.

6 Experimental setting

In this section, we validate the proposed considerations on a representative hardware platform. We adopted an NVIDIA Jetson AGX Xavier development board, featuring a host processor with 8 NVIDIA Carmel cores, an ARM-based superscalar architecture with aggressive out-of-order execution and dynamic code optimizations. Carmel Instruction Set Architecture (ISA) is compatible with ARMv8.2. The Carmel CPU complex presents 4 islands of 2 cores each: each core has a dedicated L1, while each island presents a shared L2 Cache (2 MiB). Moreover, a L3 cache (4 MiB) is contended by all the CPU cores. In this SoC, the main accelerator is represented by an integrated GPU designed with the NVIDIA Volta architecture, featuring 512 CUDA cores distributed in 8 SMs. Each SM has a 128 KiB L1 cache, and a shared L2 Cache (512 KiB) is shared among all the SMs. Other application specific accelerators are present in this SoC, such as the DLA (Deep Learning Accelerator) and the PVA (Programmable Vision Accelerator): further discussions about these additional accelerators are beyond the scope of this paper.

The software stack used in our experiments is mostly contained within the latest version of the NVIDIA Jetpack (4.1.1). It contains the L4T (Linux For Tegra) Operating System (version 31.1), which is an NVIDIA custom version of Ubuntu, featuring a linux kernel version 4.9. The CUDA SDK version is 10, whereas the Vulkan driver is updated to version 1.1.76. Lunarg SDK for glslangValidator is updated to version 1.1.97.0.

6.1 Experimental use case

The application we selected for our tests relates to one of the most peculiar workloads to be accelerated on a GPU: Deep Neural Networks (DNNs). Inference of DNNs is an extremely common use case for heterogeneous SoCs. For example, latest Advanced Driving Assitance Systems (ADAS) and Autonomous Driving (AD) prototypes make a heavy use of neural networks for the detection/classification of objects on the road, making inferencing latencies a critical aspect of the system.

Analyzing the layers of a DNN, breaking them down into algorithmic steps, each layer presents algebraic computations that are typical of other generic signal processing tasks: basic algebraic operations such as matrix/matrix and matrix/vector multiplications, convolution filters and element-wise linear and/or non-linear functions are perfect example of compute building blocks for a wide variety of embarrassingly parallel computations.

For these reasons, we believe that parameterizing neural network topologies provides a synthetic benchmark that is realistic and general enough for our proposed approaches. More specifically, we characterize our neural network as a sequence of kernel invocations, with each invocation corresponding to a different layer. We therefore parameterize both the length of this sequence of kernels and the size of the input data, as shown in Figure 6.

[3] Our Vulkan compute wrapper allows the user to write compute shaders in GLSL. This facilitates porting from CUDA or OpenCL pre-existing kernels.

[4] `https://www.lunarg.com/vulkan-sdk/`

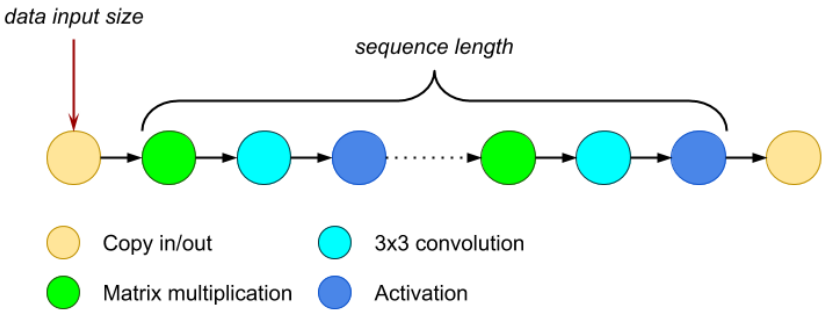


Figure 6 Schematic description of the application used for benchmarking the different submission models. Matrix multiplications are used to compute fully connected layers, as we assume dense matrices. The 3x3 convolution represents a convolutional layer. Simple activation functions (RELU and sigmoid) conclude the sequence.

In the DAG in Figure 6, copy-in operations are implemented as Copy Engine operations in which input layers, DNN weights and convolutional filters are made visible to the GPU (CUDA memcpy host to device). The copy-out operations only relates to the output buffer (CUDA memcpy device to host). Input and output buffers for each layer are switched between subsequent kernel calls. We also alternate RELU and sigmoid activation functions between subsequent iterations. For statistical analysis, the entire sequence of operations is periodically repeated. Input size matrices for both input and output buffers are $(k * block) \times (k * block)$, with $k \in [1, 2, 4, 8]$ and $block = 16$ and the total sequence length of kernel launches is given by the number of kernel launches for a single iteration (three) multiplied by the length of the sequence: $3 \times l, l \in [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000]$.

Trivially, by increasing the length of the sequence of kernel launches, the CPU is forced to increase its interactions with the GPU driver. By adjusting the data size, it is possible to change the ratio between CPU and GPU compute latencies. In our experiments, the range in which we vary the kernel iterations is compliant with the layer count of known neural networks, as deployed in real world applications: YOLO [26] is composed of 26 convolutional layers and 2 fully connected layers, whereas tiny-YOLO has 9 layers in total. Google's MobileNet [18] also features 28 layers, whereas larger networks can feature up to hundreds of layers (e.g. ImageNet [22], R-FCN [11] and latest YOLO versions).

For Vulkan, compute shader implementations are derived from CUDA device kernel code, that is then converted to GLSL compute shader with a line-by-line exact translation. Operating this translation is easy, as only the name of the built-in variables used for thread/block indexing differs between the two languages.

The Volta GPU architecture features application specific circuitry (i.e. *tensor cores*) for further acceleration of matrix-multiply-accumulate operations. However, we are not aware of any Vulkan/GLSL support for tensor core operations. To allow a fair comparison, also our CUDA kernel implementation uses only regular CUDA cores.

All the experiments have been performed with the highest supported frequency and voltage for the whole SoC with no dynamic frequency and voltage scaling (i.e. `nvpmodel -m 0 && jetson_clocks.sh`). Processes are pinned to one CPU core and set to FIFO99 priority. Display servers/compositors or any other GPU application other than our benchmarks are not permitted to execute. Code for our experiments can be found at `https://git.hipert.unimore.it/rcavicchioli/cpu_gpu_submission`.

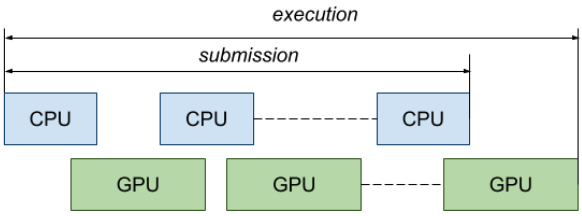


Figure 7 Simplified visualization of the submission and execution metrics evaluated in our experiments. Depending on the submission model, both the CPU and GPU tasks might be represented as a variable number of small blocks.

7 Results

We assume that context set up, memory allocations and kernel compiling procedures are operations performed only once, during the whole system initialization phase. Therefore, we did not characterize these phases, as they do not impact the runtime latencies of recurring real-time workloads. For CUDA graphs and Vulkan, we considered graph creation, command buffer recording and pipeline creation as part of the initialization phase. CUDA graph is created by capturing the same memory and kernel operations enqueued in the baseline stream.

For this experimental setting, we measured two response times: submission completion time (*submission*) and total execution time (*execution*). The submission time is the time spent on the CPU submitting thread between the asynchronous submission of a task to the time in which the control is given back to the CPU submitting thread. In the baseline scenario, this translates in measuring the time taken for enqueuing all the necessary operations in the CUDA stream. For CDP, this latency is only observed for submitting the parent kernel launch, whereas for CUDA graphs we only measure the latency of the *cudaGraphLaunch* call. Similarly, for Vulkan we only measure the submission of the pre-recorded command buffer (*VkQueueSubmit* function call, equivalent to *submitWork* in our VkComp wrapper). The total execution time is the time taken for completing the submitted job. As shown in Figure 7, this latter measure includes both the CPU submission work and the GPU sequence of copy and compute kernel operations.

The results are shown in Figure 8 and 9 (pp. 15, 16) and are quite surprising (please note the logarithmic scale of the plots). In the leftmost column, where the submission time is shown, a drastic reduction can be noticed for Vulkan compared to all the CUDA alternatives. With Vulkan, the worst-case time needed to regain control of the CPU submitting thread can be *orders of magnitude smaller* than in the CUDA-based approaches. More specifically, comparing Vulkan with CDP (the best performing among the CUDA methodologies), the improvement ranges from 4× to 11×.

Even more interestingly from a predictability perspective, while increasing the sequence length causes significant variations in submission times for CUDA baseline and graphs, such variations in Vulkan are almost negligible. Figure 10a shows the measured jitter for different sequence lengths, i.e. the difference between recorded maximum and minimum submission times.

When comparing the three CUDA methodologies, we see a consistent behaviour throughout all the tested configurations and data block sizes. As expected, the CUDA baseline performance are lower then the other two methodologies when considering both average and worst cases.

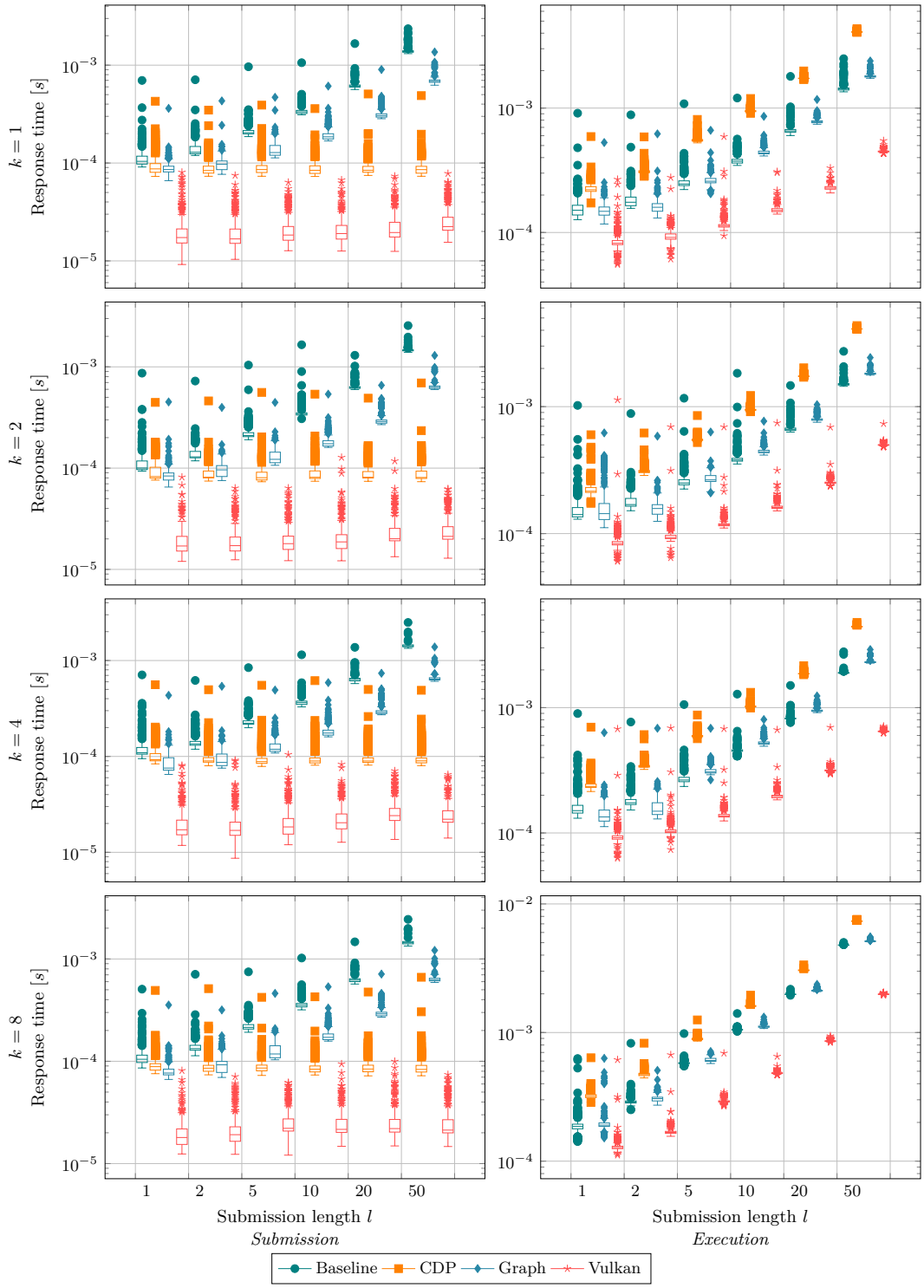


Figure 8 Results of response time for submission (left) and execution (right) for $l \in [1, 2, 5, 10, 20, 50]$, $block = 16$ and $k \in [1, 2, 4, 8]$. Y-scale is logarithmic and different between left and right column.

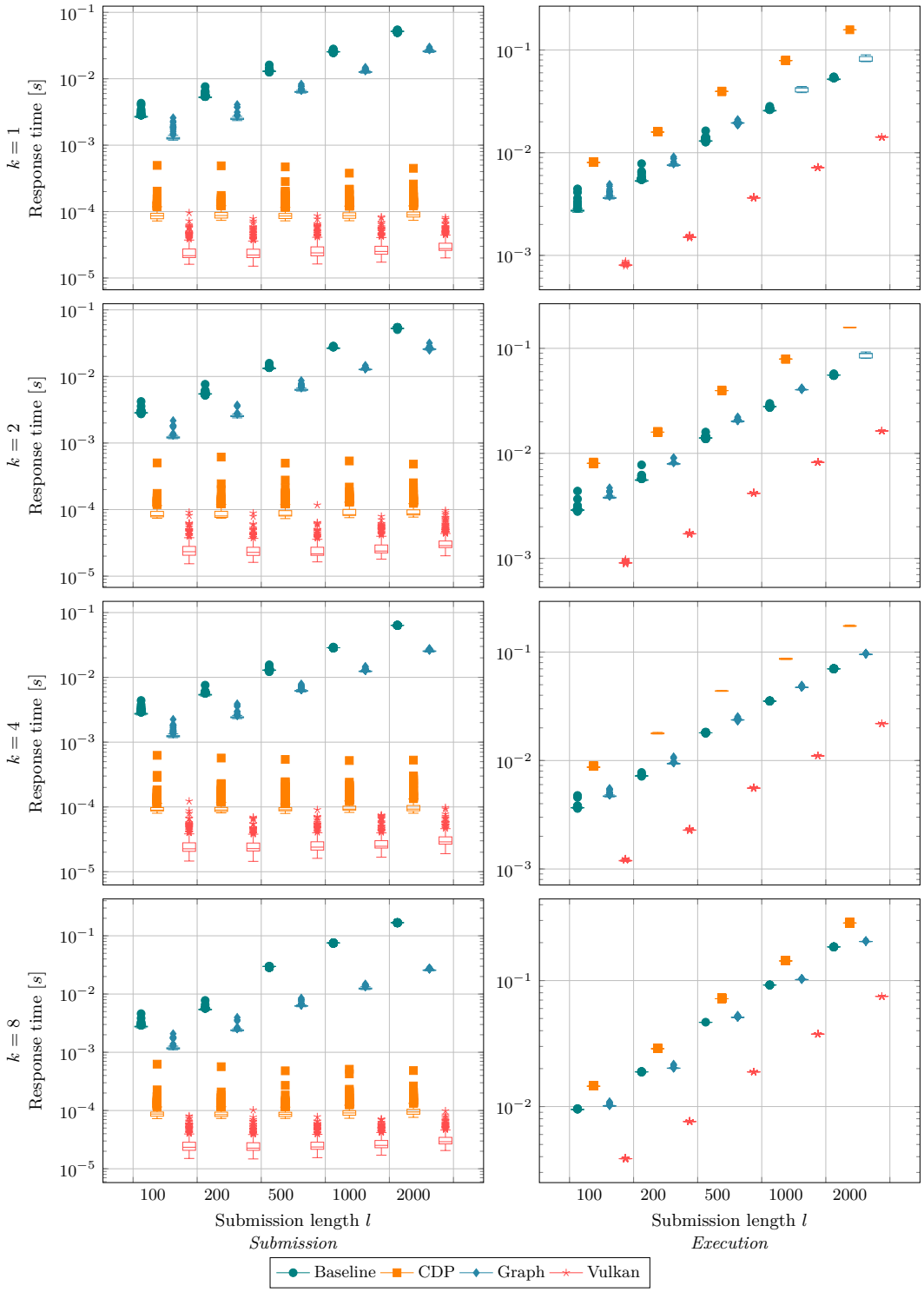
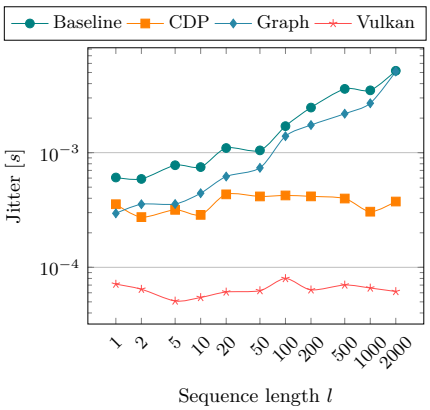
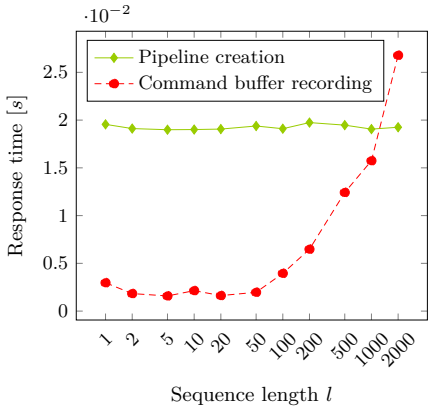


Figure 9 Results of response time for submission (left) and execution (right) for $l \in [100, 200, 500, 1000, 2000]$, $block = 16$ and $k \in [1, 2, 4, 8]$. Y-scale is logarithmic and different between left and right column.



(a) Difference between maximum and minimum submission times for each model, $k = 1$.



(b) Job-creation overheads for Vulkan.

Figure 10 Submission jitter and Vulkan creation latencies. X-Axis scale is logarithmic.

We now discuss the right side of Figures 8 and 9 detailing the total response times. The Vulkan approach still outperforms the other three methodologies. Increasing l, the measured response times increase in all the tested methodologies, as expected. However, Vulkan performance improvement over CUDA becomes more evident and stabilizes with larger l values (same slope observable in Figure 9), starting from $2\times$ for $l = 1$ up to $\sim 6\times$ for $l \geq 10$. CDP always performs worse then the other two CUDA based approaches and the deterioration of CDP performance gets more noticeable when increasing k and l. CUDA graphs execution times are comparable to the baseline implementation.

7.1 Discussion

The Vulkan design philosophy is responsible for such surprisingly short submission times. More specifically, by forcing the application developer to explicitly instantiate a pre-recorded version of the command pipeline and by avoiding any runtime validation checks, it manages to drastically reduce the interactions between the CPU and the GPU. On the contrary, by the time a CUDA runtime function is called to when it is actually enqueued in the command buffer, an overhead is added in terms of validation and error checking: such operations act as added overhead on the CPU side as they operate in an undisclosed and uncontrolled fashion.

An experiment with just four lines of code can serve as an example to reveal validation mechanisms at API level: let us create a CUDA runtime application that creates a host and a device pointer (respectively a pointer to CPU-visible memory region, and a GPU-only visible pointer). We allocate memory for both and we use pinned (*cudaMallocHost*) memory for the host pointer. We then set up a DMA transfer indicating the host pointer as the source, the device pointer as destination, and we wrongfully flag this to be a device to host memory copy. This program compiles, runs and – according to *nvprof*, the NVIDIA CUDA profiler – executes a host to device to memory copy, hence indicating that pointers were validated with respect to their address spaces, setting the actual DMA transfer differently than originally specified. The experiment suggests that calls were checked, validated and (if necessary) corrected. It is reasonable to assume that heavier driver mechanisms are also in place for kernel invocations.

Recall that kernels are identical in both the CUDA and Vulkan version. The performance improvement shown by Vulkan is higher for submission intensive workloads, where CPU times are not negligible with respect to GPU compute times. In these cases, minimizing submission

Table 1 Tracing result summary by tracing NVIDIA GPU kernel module: number of submissions, average duration, number of additional calls required to increment l by 1, memory management calls.

Submission model	*Number of submit calls*	*Average time [s]*	Δ	*Number of MM operations*
Baseline	611	$2.66776 \cdot 10^{-6}$	3	1,484
CDP	610	$1.92295 \cdot 10^{-6}$	0	1,484
Graph	635	$2.15276 \cdot 10^{-6}$	27	1,484
Vulkan	24	$2.05 \cdot 10^{-5}$	0	1,110

times allows drastically decreasing the total execution time. If the examined workload is characterized by few kernel calls and reasonably sized data buffers, total execution times between Vulkan and CUDA would not present such significant differences. This is partially visible in our results for smaller l and larger k.

A way to avoid paying the price of driver hidden mechanisms is to relieve the CPU from heavy submission work. The CDP methodology moves the responsibility of future kernel calls to threads that run on the GPU. From our results, we discovered that this is actually a very effective approach for minimizing CPU interactions for submission, at least for reasonable sequence lengths. However, the device-side overhead for nested kernel invocations causes significant performance deterioration with respect to GPU response time. This effect was already observed in older GPU architectures [6]. Likely, the CDP model cannot provide benefits to the kind of GPU workloads used in our tests: a sequence of kernel invocations, with little variations to their launch configuration, does not resemble a tree-like structure of nested kernel invocations. We however argue that the tree-like kernel invocations suitable to CDP are not as common as simpler sequential kernel invocations, at least for typical embedded workloads.

CUDA graphs performed below our expectations for both submission and execution metrics. We assumed that this recently-introduced CUDA feature would make it behave similarly to the Vulkan API – the idea of capturing stream operations strongly resembles Vulkan's command buffer recording. Despite launching a CUDA graph involves only calling a single CUDA runtime function, the hidden driver overhead is comparable to the one observed in the baseline methodology. Since our measurements only considered user space runtime functions, the performance of CUDA graph motivated us to further examine kernel driver overheads.

7.2 Tracing the kernel driver

Although the NVIDIA GPU runtime is proprietary and closed source, its kernel driver module is open source and freely available at the NVIDIA official website. In order to understand the submission mechanisms below the user space application that we implemented for our previous tests, we can thus analyze kernel driver operations with all the tested methodologies. Basic GPU kernel driver tracing allows us to obtain initial and final timestamps of the GPU command submission into the actual command push buffer, which is a memory region written by the CPU and read by the GPU [7, 8]. By enabling the debug interface of the *gk20a* module, we discover that command push buffer writes are delimited by *gk20a_channel_submit_gpfifo* and *gk20a_channel_submitted_gpfifo* function traces. We also traced driver functions related to memory management *gk20a_mm_** to understand their impact in the kernel submission process. We recorded those traces using the *trace-cmd-record* utility, reporting our findings in Table 1.

The second column of Table 1 shows the number of submit operations when $l = 1$ and $k = 1$, accounting for both context initialization and actual work submission. The average time for these submission calls has been calculated by subtracting the timestamps of *gk20a_channel_submitted_gpfifo* and its respective *gk20a_channel_submit_gpfifo* function trace. The Δ column is simply the difference of submit calls when $l = 2$, and represents the number of additional submit calls required by any submission after the first one. The last column is the counter of memory management operations.

In terms of sheer number, submit calls show no significant difference among the CUDA methods. Regarding CUDA graphs, despite the fact that launching a graph only involves a single user space runtime function, the number of submit calls is even larger than the baseline. This explains the reason why CUDA graphs response times are comparable to the baseline. Vulkan is characterized by a $25\times$ reduction in submit calls compared to CUDA. However, Vulkan submit duration is a magnitude larger than all the other approaches. The Δ column shows the results we expected for baseline, CDP and Vulkan: by increasing l by a single unit, the baseline presents 3 additional submit calls (1 per additional kernel invocation), whereas CPD does not present any additional calls, as nested invocations are operated inside the GPU. Vulkan driver activity continues to follow the user space application model: increasing l leads to a larger recorded command buffer, but this does not cause an increased number of submit calls as every additional invocation is prerecorded in the same Vulkan command buffer/queue. We are unable to explain the $9 \times \Delta$ for the CUDA graph value compared to the baseline, as if we assume that driver interactions for CUDA graphs would resemble the baseline, having the same Δ as the baseline.

The captured traces for memory operations are identical in all the CUDA methodologies and do not increase with l, with Vulkan presenting 25% less memory management operations compared to CUDA. For what we were able to infer from a mostly closed source system, it is safe to assume that Vulkan is able to cache and pre-record GPU commands to then minimize CPU and GPU interactions (both in user and kernel space) when offloading periodic workloads. This implies spending a significant amount of time for initialization procedures (i.e. pipeline creation and command buffer recording) to then utilize the CPU for the bare minimum for submitting an already well-formatted sequence of operations. For the sake of completeness, we profiled the time taken by the CPU for creating pipelines and recording a command buffer of variable size. These metrics are reported in Figure 10b, where we characterized the time spent on the CPU during the initialization phase of a Vulkan application.

Recall that pipeline creation manages the binding of each distinct compute program with its input/output buffer layouts: these do not change when increasing l. On the contrary, the command buffer should record all the kernel invocations in advance. As a consequence, larger invocation sequences lead to larger command buffers, with a proportional increase in recording time. Pipeline creation is the most expensive operation (tens of ms). However, if the number of invocations reaches unrealistically large values (i.e. thousands of invocation in a single batch), recording times tend to dominate over pipeline creation.

Our findings show that no matter which submission model is selected for CUDA, Vulkan bare metal approach manages to minimize and better distribute the CPU overhead, especially for pathological workloads in which the CPU offloading times tend to dominate over the actual GPU compute and DMA transfer times.

8 Conclusion and future work

In this work, we aimed at characterizing and modeling CPU-to-GPU submission latencies for real-time systems executing on heterogeneous embedded platforms. We compared recently released CUDA submission models and the novel open standard Vulkan API for GPU accelerated workloads. In an extensive set of experiments, we considered typical workloads, consisting of inferencing on a neural network of parameterized topology, to profile GPU command submission times and total execution times. The results show that CPU offloading latencies can act as a bottleneck for performance, negatively impacting predictability and jitter, and making the schedulability analysis significantly more complex due to the large number of CPU-GPU interactions.

Considering CUDA approaches, recently introduced CUDA submission models can slightly improve performance (both on submission and execution times) compared to the commonly utilized baseline approach. However, considering a deeper neural network and buffer data size, the performance penalties during the actual kernel computations reduce or invalidate the benefits of a reduced CPU activity gained for submission operations, especially for the CDP approach.

On the other hand, the Vulkan API was able to minimize and better distribute the CPU overhead in all the tested configurations. This led to significant improvements, i.e. up to 11× faster submissions and from 2× to 6× faster GPU operations, with almost negligible jitter. Moreover, the significant reduction of CPU-GPU interactions significantly simplifies the topology of the DAG to consider for deriving an end-to-end response-time analysis, allowing a tighter characterization of the system schedulability. For these reasons, we argue that the Vulkan API should be seriously considered by real-time systems designers for dispatching periodic workloads upon predictable heterogeneous SoCs.

Despite the closed nature of the NVIDIA runtime, we traced the relevant GPU driver kernel module calls to explain the performance gap between CUDA and Vulkan. In this latter, driver interactions between application and GPU driver are kept to the bare minimum, hence providing the surprising results we discussed. Moreover, it is worth noticing that Vulkan is a cross platform API, with no hardware vendor or operating system restrictions and specified as an open standard, whose importance has been recently stressed [33]. Limitations for the Vulkan approach, when compared to CUDA, are to be found in the substantially higher implementation complexity for the developer, in the notable time needed to pre-record commands/pipelines, and in the lack of an ecosystem of utility libraries (e.g. *cuDNN* for Deep Neural Network operations, *cuBLAS* for linear algebra, etc.)

As a future work, we aim to include OpenCL in this comparison to experiment on different hardware platforms. We are also in the process of investigating different use cases, in which multiple concurrent and parallel kernels overlap in time, described by DAGs with parametrized breadth. This analysis is important when considering recurrent neural network such as R-FCN [11], which includes loops among different layers. This aspect differs from the feed-forward networks that inspired the use case in our experiments. Finally, we plan to benchmark the energy consumption for all the tested approaches. We speculate that the reduced CPU-GPU interaction might bring substantial benefits to the power consumption, a crucial aspect for the considered platforms.

References

1 Mohammed Alandoli, Mahmoud Al-Ayyoub, Mohammad Al-Smadi, Yaser Jararweh, and Elhadj Benkhelifa. Using Dynamic Parallelism to Speed Up Clustering-Based Community Detection in Social Networks. In *Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on*, pages 240–245. IEEE, 2016.

2 Waqar Ali and Heechul Yun. Work-in-progress: Protecting real-time GPU applications on integrated CPU-GPU SoC platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 141–144. IEEE, 2017.

3 Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115. IEEE, 2017.

4 Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

5 Jens Breitbart. Static GPU threads and an improved scan algorithm. In *European Conference on Parallel Processing*, pages 373–380. Springer, 2010.

6 Nicola Capodieci and Paolo Burgio. Efficient implementation of Genetic Algorithms on GP-GPU with scheduled persistent CUDA threads. In *Parallel Architectures, Algorithms and Programming (PAAP), 2015 Seventh International Symposium on*, pages 6–12. IEEE, 2015.

7 Nicola Capodieci, Roberto Cavicchioli, and Marko Bertogna. Work-in-Progress: NVIDIA GPU Scheduling Details in Virtualized Environments. In *2018 International Conference on Embedded Software (EMSOFT)*, pages 1–3. IEEE, 2018.

8 Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based Scheduling for GPU with Preemption Support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018.

9 Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory Interference Characterization between CPU cores and integrated GPUs in Mixed-Criticality Platforms. In *22nd IEEE International Conference on Emerging Technologies And Factory Automation (ETFA)*, 2017.

10 Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. EffiSha: A software framework for enabling effficient preemptive scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16. ACM, 2017.

11 Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016.

12 Glenn A Elliott and James H Anderson. Real-world constraints of GPUs in real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 2, pages 48–54. IEEE, 2011.

13 Glenn A Elliott and James H Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 267–276. IEEE, 2012.

14 Glenn A Elliott, Bryan C Ward, and James H Anderson. GPUSync: A framework for real-time GPU management. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 33–44. IEEE, 2013.

15 Kshitij Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

16 Islam Harb and Wu-Chun Feng. Characterizing Performance and Power towards Efficient Synchronization of GPU Kernels. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pages 451–456. IEEE, 2016.

17 Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. GPU virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3):35, 2017.

18 Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint*, 2017. `arXiv:1704.04861`.

19 Khronos Group Khronos. Khronos SPIR-V Registry. *Khronos Group*, 2016. URL: `https://www.khronos.org/registry/spir-v/#spec`.

20 Khronos Group Khronos. The OpenGL Shading Language Language Version: 4.50. *Khronos Group*, 2016. URL: `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf`.

21 Khronos Group Khronos. Vulkan 1.0.98 - A Specification. *Khronos Group*, 2019. URL: `https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html`.

22 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

23 B Neelima, Bharath Shamsundar, Anjjan Narayan, Rithesh Prabhu, and Crystal Gomes. Kepler GPU accelerated recursive sorting using dynamic parallelism. *Concurrency and Computation: Practice and Experience*, 29(4):e3865, 2017.

24 CUDA Nvidia. Programming Guide Version 10.0. *Nvidia Corporation*, 2018. URL: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

25 Ignacio Sañudo Olmedo, Nicola Capodieci, and Roberto Cavicchioli. A Perspective on Safety and Real-Time Issues for GPU Accelerated ADAS. In *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*, pages 4071–4077. IEEE, 2018.

26 Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

27 Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *arXiv preprint*, 2016. `arXiv:1612.08242`.

28 Davesh Shingari, Akhil Arunkumar, and Carole-Jean Wu. Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones. In *2015 IEEE International Symposium on Workload Characterization (IISWC)*, pages 22–33. IEEE, 2015.

29 Joseph A Shiraef. An exploratory study of high performance graphics application programming interfaces. Master's thesis, University of Tennessee at Chattanooga, 2016.

30 Jan-Philipp Stauffert, Florian Niebling, and Marc Erich Latoschik. Towards comparable evaluation methods and measures for timing behavior of virtual reality systems. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*, pages 47–50. ACM, 2016.

31 Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

32 Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H Anderson, F Donelson Smith, and Shige Wang. Making OpenVX Really "Real Time". In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 80–93. IEEE, 2018.

33 Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H Anderson, and F Donelson Smith. Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

34 Peter Zhang, Eric Holk, John Matty, Samantha Misurda, Marcin Zalewski, Jonathan Chu, Scott McMillan, and Andrew Lumsdaine. Dynamic parallelism for simple and efficient GPU graph algorithms. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 11. ACM, 2015.