Nadjib Mammeri, Ben Juurlink

VComputeBench: A Vulkan Benchmark Suite for GPGPU on Mobile and Embedded GPUs

Nadjib Mammeri, Ben Juurlink (2018): VComputeBench: A Vulkan Benchmark Suite for GPGPU on Mobile and Embedded GPUs. In: 2018 IEEE International Symposium on Workload Characterization

VComputeBench: A Vulkan Benchmark Suite for GPGPU on Mobile and Embedded GPUs

Nadjib Mammeri
Technische Universität Berlin
mammeri@tu-berlin.de

Ben Juurlink
Technische Universität Berlin
b.juurlink@tu-berlin.de

***Abstract*—GPUs have become immensely important computational units on embedded and mobile devices. However, GPGPU developers are often not able to exploit the compute power offered by GPUs on these devices mainly due to the lack of support of traditional programming models such as CUDA and OpenCL. The recent introduction of the Vulkan API provides a new programming model that could be explored for GPGPU computing on these devices, as it supports compute and promises to be portable across different architectures.**

In this paper we propose VComputeBench, a set of benchmarks that help developers understand the differences in performance and portability of Vulkan. We also evaluate the suitability of Vulkan as an emerging cross-platform GPGPU framework by conducting a thorough analysis of its performance compared to CUDA and OpenCL on mobile as well as on desktop platforms.

Our experiments show that Vulkan provides better platform support on mobile devices and can be regarded as a good cross-platform GPGPU framework. It offers comparable performance and with some low-level optimizations it can offer average speedups of 1.53x and 1.66x compared to CUDA and OpenCL respectively on desktop platforms and 1.59x average speedup compared to OpenCL on mobile platforms. However, while Vulkan's low-level control can enhance performance, it requires a significantly higher programming effort.

***Index Terms*—VComputeBench, Vulkan, SPIR-V, GPGPU, CUDA, OpenCL, Rodinia, Mobile**

I. Introduction

Graphics Processing Units (GPUs) have become a dominant platform for parallel computing thanks to their massively parallel architecture, energy efficiency and availability to the masses. Several programming models have emerged enabling developers to harness the massive compute power offered by GPUs, while exploiting parallelism for different application domains. This is often referred to as GPGPU (General Purpose computing on the GPU) [1]. The most popular GPGPU programming models are CUDA [2] and OpenCL [3]. CUDA is a proprietary standard introduced by NVIDIA and targets only NVIDIA specific hardware, while OpenCL is an open standard maintained by the Khronos group and targets additional hardware devices including FPGAs, CPUs and DSPs. In this work we focus on the two most predominant programming models CUDA and OpenCL, but it is worth mentioning other frameworks such as OpenMP [4] and OpenACC [5]. OpenMP mainly targets shared memory multiprocessors and recently OpenMP 4.5 introduced the *target* directive enabling support for GPUs and other devices. OpenACC is mainly designed to program accelerators in heterogeneous systems with OpenMP-like directives.

To add to this mix of programming models, the Khronos group recently released the Vulkan API [6] along with SPIR-V [7]. Vulkan is a low level API with an abstraction closer to the behavior of the actual hardware. It promises cross-platform support, high-efficiency and better performance of GPU applications. Unlike CUDA, which is only supported on NVIDIA GPUs, and OpenCL, which has no official support on mobile GPUs, Vulkan is supported by all major GPU vendors[1] and considers non-desktop GPUs as first class citizens. Vulkan is officially supported on Android 7.0 [8] and on the new Tizen OS 3.0 [9] covering a full spectrum of mobile devices from phones and wearables to TVs and in-vehicle infotainment systems. This good platform support and the fact that it also supports compute, motivated us to examine it from the GPGPU perspective even though it was mainly designed to improve graphics performance. In this paper, we introduce Vulkan as a cross-platform GPGPU route that could open new perspectives for pertinent GPGPU computing on mobile devices and can be explored along with other more established frameworks on desktop architectures. However, there are some important questions yet to be answered:

- What kind of performance can we get out of Vulkan?
- Is there a viable study comparing Vulkan compute to established frameworks such as CUDA and OpenCL?
- If there are any performance gains, are these portable across different GPU architectures?
- Can Vulkan enable pertinent GPGPU computing on mobile and embedded GPUs?

Selecting which GPGPU framework to choose is a critical task for developers. Differences in performance, portability, programmability and platform support are all very important factors that need to be considered. Benchmarks play an important role in exposing these kind of differences between hardware architectures, compilers and more importantly across competing programming models. There are several benchmarks available to evaluate CUDA and OpenCL [10], [11] [12] [13] but currently none for Vulkan. To fill this gap and enable our study we propose VComputeBench, a set of Vulkan compute benchmarks that help developers understand

[1] Supported by major desktop GPU vendors: AMD, NVIDIA and Intel and mobile GPU vendors: Qualcomm, ARM, Imagination and VeriSilicon

the differences in performance and portability of Vulkan and provide guidance to GPU architects in the design and optimization of their drivers and runtime. VComputeBench was developed by extending the popular Rodinia benchmark suite [10], covering a diverse range of application domains with different computation patterns. The reason for selecting the Rodinia suite is that it provides OpenCL and CUDA implementations and with our VComputeBench implementations we can make fair comparisons and adequately evaluate Vulkan against other programming models.

In essence, the main contributions of this paper are:

- Illustrate the viability of Vulkan as a GPGPU framework notably on mobile devices.
- Propose a set of Vulkan compute benchmarks named VComputeBench and ported them onto mobile platforms.
- Perform a thorough analysis of performance, comparing Vulkan to CUDA and OpenCL on desktop and mobile GPUs and highlight a set of Vulkan specific optimization techniques.

II. Related Work

In recent years, GPGPU frameworks have received a great amount of attention from the research community. Although, several works studied and compared different programming models [14] [15] [16] [17] [18] [19] [20] [21], none of them studied Vulkan. To the best of our knowledge, our work is the first to investigate Vulkan from the compute not the graphics perspective and propose it as a viable cross-platform GPGPU programming model. One of the earliest and well cited works is those of Fang et al. [15] and Karimi et al. [14]. The authors compare CUDA to OpenCL in terms of performance on old desktop GPU architectures. Our work, on the other hand, was carried out on recent architectures and analyses performance on desktop as well as mobile GPUs. Du et al. [17] studies OpenCL performance portability and Wang et al. [21] examines OpenCL on FPGAs. The authors of these papers demonstrate that performance is not necessarily portable across architectures. Their findings instigated us to study and port our benchmarks onto mobile GPUs in order to evaluate Vulkan's portability and examine its performance implications.

Such research works heavily rely on benchmarks for their evaluations. Several GPGPU benchmarks were proposed by researchers such as Rodinia [10], Parboil [11] SHOC [12] and the recent Hetero-Mark [13]. Most of these benchmark suites include CUDA, OpenCL or OpenMP implementations but none include Vulkan implementations. This can be a limitation especially for researchers and developers wanting to target this new emerging programming model. In this work, we aim to enrich the GPGPU community with such Vulkan benchmarks by extending the popular Rodinia suite, enabling researchers and developers to evaluate Vulkan along with other GPGPU programming models. Likewise, most of these benchmark suites mainly target desktop GPUs or multicore systems with their CUDA and OpenCL implementations. Our benchmarks, on the other hand, target both mobile and desktop GPUs. We chose Vulkan because of its cross-platform capabilities and good support on mobile devices.

III. Vulkan a Compute Perspective

In this section we present an overview of the Vulkan programming model illustrating why it is a promising GPGPU framework especially for mobile and embedded GPUs.

A. Vulkan Overview

Vulkan is often referred to as the next generation graphics and compute API for modern GPUs. It is an open standard that aims to address the inefficiencies of traditional APIs such as OpenGL, which were designed for single-core processors and lag to map well to modern hardware [22]. Vulkan on the other hand, was designed from the ground-up with *multi-threading* support in mind. Better parallelization can be achieved by asynchronously generating work across multiple threads feeding the GPU in an efficient manner. This is attained in Vulkan by having no global state, no synchronizations in the driver and separating work generation from work submission. All state is localized in *command buffers*, which can be generated on multiple threads and only start executing on the GPU after submission.

The other key characteristic of Vulkan is that it provides a much lower-level fine-grained control over the GPU enabling developers to maximize performance across many platforms. It achieves this by being *explicit* in nature rather than relying on hidden heuristics in the driver. Operations such as resource tracking, synchronization, memory allocation, and work submission are all pushed into application space resulting in higher predictability and better control of when and where work happens. Likewise, unnecessary background tasks such as error checking, hazard tracking, state validation and shader compilation are delegated to the tooling layers, which are present during development and removed at runtime, resulting in low driver overhead and less CPU usage [23].

B. The Programming Model

Vulkan can be viewed as a pipeline with some programmable stages that are invoked by a set of operations. To the programmer, it is simply an API with a set of routines allowing for the specification of shaders or kernels, state controlling aspects as well as data used by those kernels. From the compute perspective though, the pipeline has only one programmable stage represented in the kernel program to be executed [6].

a) Execution Model: A Vulkan-capable system exposes one or more *devices*, each of theses physical devices exposes one or more *queues*. These queues are partitioned into *queue families* and can process work asynchronously to one another. Each queue family supports a number of functionalities and may contain multiple queues with similar characteristics. There are four types of queue functionalities defined in Vulkan: graphics, compute, transfer, and sparse memory management. The reason for having queue families is that queues within a single family are considered compatible

with one another, and work produced for one queue family can be executed on any queue within that family.

A *queue* is considered as the interface between the application and the execution engines of a device. Commands for these execution engines are recorded into *command buffers* ahead of execution time. Once recorded, a command buffer can be cached and submitted to a queue for execution as many times as required. Command buffer construction is expensive and the application may employ multiple threads to construct multiple command buffers in parallel. These command buffers are then submitted to queues for execution in a number of batches. Once submitted to a queue, the commands within a command buffer begin and complete execution without further application intervention. The order in which these commands are executed is dependent on a number of implicit and explicit ordering constraints.

In addition, command buffers submitted to different queues may execute in parallel or even out of order with respect to one another. Command buffers submitted to a single queue though respect submission order. Host execution is also asynchronous to command buffer execution on the device. Control may return to the application as soon as the command buffer is submitted and the application should take responsibility for any synchronizations between different queues as well as between the device and host.

b) Compute Model: In Vulkan, compute workloads are initiated by recording dispatching commands $\mathrm{vkCmdDispatch}*$ in a command buffer. Once a command buffer is submitted to a queue, execution starts according to the currently bound *compute pipeline*. Compute pipelines consist of a single compute shader stage, describing the kernel to be executed and a pipeline layout, describing the input and output resources to that kernel. Dispatching commands take three input parameters: $\mathrm{groupCountX}$, $\mathrm{groupCountY}$ and $\mathrm{groupCountZ}$ defining the total number of *workgroups* or the so called global workgroup size in the X, Y and Z directions respectively. A workgroup is the smallest amount of compute operations that an application can execute. Within a single workgroup, there may be many *workitems* or compute shader invocations. This is called the local workgroup size and is defined by the compute shader itself using SPIR-V built-in decorations [7].

c) SPIR-V: All shaders and compute kernels in Vulkan are defined using the Standard Portable Intermediate Representation (SPIR-V), which is a platform-independent intermediate language for describing graphical shaders and compute kernels [24]. SPIR-V is a self-contained binary format. Logically, it is a header and a linear stream of instructions and physically it is just a stream of 32-bit words, encoding a collection of annotations and decorations as well as functions, which in turn encode control-flow graphs (CFG) of blocks. Variables are accessed using load store instructions and any intermediate results bypassing the load store are represented in a single static-assignment form (SSA). Hierarchical type information of data objects is preserved to not lose information needed for further optimizations on the target device.

C. Why Vulkan for Mobile and Embedded GPUs?

Considering that Vulkan was mainly designed to achieve higher graphics performance, we can make several interesting observations: *(i)* its enhancements and low-level nature can also be utilized to achieve higher performance for GPGPU applications. *(ii)* Vulkan's main focus on graphics allowed it to have better support among GPU vendors than other open frameworks such as OpenCL, which for instance is not fully supported by NVIDIA because it considered as a competitor to its propriety CUDA framework [2]. *(iii)* Vulkan is considered as the first framework to have official support on mobile platforms [8] [9] and the API was designed with mobile GPU features in mind such as tiled rendering. Hence, it has the potential of being the framework of choice for GPGPU on mobile devices, which is the quest of many recent research works [25] [26] [27]. This leads us to our final observation: *(iv)* that Vulkan can be the appropriate framework for achieving true cross-platform GPGPU without sacrificing on performance.

IV. Benchmarks

Benchmarks play an important role in exposing differences in performance, portability and programmability across competing programming models. Since Vulkan was recently released and its main focus is on graphics not GPGPU, there are currently few graphics but no compute benchmarks that can be of use to our study. In order to enable our work as well as to enrich the research community with such benchmarks, we extended the popular Rodinia benchmark suite [10] by developing Vulkan equivalents of most of its workloads, referred to as VComputeBench, and made them publicly available to the wider GPGPU community.

Before describing our VComputeBench benchmarks, we first present one of the microbenchmarks that we used in our study to better illustrate this new programming model and give an overview of what is required to write a Vulkan compute application.

A. Vector Addition Microbenchmark

This microbenchmark is a simple application adding two vectors X and Y of size n saving the output in vector Z. The *kernel code*, or the compute shader in Vulkan terminology, is a SPIR-V binary that was compiled offline from a 10-line GLSL source implementing:

$$Z[i] = X[i] + Y[i] \quad \forall i \in [0, 1, \ldots, n]$$

The index space is one dimensional and i is defined using the SPIR-V decoration $\mathrm{GlobalInvocationId}$, which returns the global ID of the workitem executing the kernel. The vectors X, Y and Z are bounded in to the kernel as storage buffers.

The *host code*, on the other hand, is more complicated. Listing 1 shows a pseudo-code listing of the host program highlighting only the important API calls.

[2]Current OpenCL version is 2.2 but NVIDIA only supports version 1.2

```
int main ()
 std::size_t N = 1000000; // Number of elements in a vector
 int numWorkGroups = N / 256; // Workgroup size is 256
 // Enumerate devices then create instance, queues and device
 VkInstance instance; VkInstanceCreateInfo instanceInfo = -" ...
 vkCreateInstance(&instanceInfo, nullptr, &instance);
 vkEnumeratePhysicalDevices(instance, ..., &gpuList);
 vkGetPhysicalDeviceQueueFamilyProperties(gpuList[0], ...);
 ...
 VkDeviceQueueCreateInfo queueCreateInfo-" ...
 VkDevice device; VkDeviceCreateInfo deviceInfo = -" ...
 vkCreateDevice(gpuList[0], &deviceInfo, ..., &device);
 VkQueue computeQueue;
 vkGetDeviceQueue(device, queueFamilyIndex, 0, &computeQueue);
 ...
 // Create buffer then bind the buffer to the allocated memory
 VkBuffer bufferX; VkBufferCreateInfo bufferCreateInfo-" ...
 bufCreateInfo.size  = N*sizeof(float);
 bufCreateInfo.usage =
  ↪   VK_BUFFER_USAGE_STORAGE_BUFFER_BIT —
                  VK_BUFFER_USAGE_TRANSFER_DST_BIT;
 vkCreateBuffer(device, &bufferCreateInfo, nullptr, &bufferX);
 VkMemoryRequirements xBuffMemReqs;
 vkGetBufferMemoryRequirements(device, bufferX, &xBuffMemReqs);
 int xMemIndex = findMemType(xBuffMemReqs.memoryTypeBits,
                  VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT);
 VkDeviceMemory memory; VkMemoryAllocateInfo memAllocInfo-" ...
 memAllocInfo.allocationSize  = xBuffMemReqs.size;
 memAllocInfo.memoryTypeIndex = xMemIndex;
 vkAllocateMemory(device, &memAllocInfo, nullptr, &memory);
 vkBindBufferMemory(device, bufferX, memory, 0);
 ...
 // Create the compute shader and the compute pipeline
 VkShaderModule module; VkShaderModuleCreateInfo
  ↪   shadCreatInfo-" ...
 shadCreatInfo.pCode = readSpirvBinary("vectorAdd.spv");
 vkCreateShaderModule(device, &shadCreatInfo, NULL, &module);
 VkPipelineShaderStageCreateInfo shaderStageCreateInfo-" ...
 shaderStageCreateInfo.module = module;
 shaderStageCreateInfo.stage  =
  ↪   VK_SHADER_STAGE_COMPUTE_BIT;
 VkPipelineLayout pipelineLayout;
 ...
 vkCreatePipelineLayout(device, ..., &pipelineLayout);
 VkPipeline ppline; VkComputePipelineCreateInfo ppCreateInfo-" ...
 ppCreateInfo.stage  = shaderStageCreateInfo;
 ppCreateInfo.layout = pipelineLayout;
 vkCreateComputePipelines(device, &ppCreateInfo, &ppline ...);
 ...
 // Bind buffers to compute pipeline
 VkWriteDescriptorSet writeDescripSet-" ...
 writeDescripSet.descriptorType =
  ↪   VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
 writeDescripSet.dstBinding = 0; // Same as SPIRV Binding
  ↪   decoration
 writeDescripSet.pBufferInfo = xBufferDescriptor;
 vkUpdateDescriptorSets(device, 1, &writeDescripSet, 0, NULL);
 ...
 // Create command pool and allocate a command buffer
 VkCommandPool cmdPool; VkCommandPoolCreateInfo
  ↪   cmdPoolInfo-" ...
 vkCreateCommandPool(device, &cmdPoolInfo, nullptr, &cmdPool);
 VkCommandBuffer cmdBuffer; VkCommandBufferAllocateInfo
  ↪   allcInfo-"..
 allcInfo.commandPool = cmdPool;
 vkAllocateCommandBuffers(device, &allcInfo, &cmdBuffer);
 ...
 // Bind the pipeline and record commands to the command buffer
 vkCmdBindPipeline(cmdBuffer,
 VK_PIPELINE_BIND_POINT_COMPUTE,ppline);
 vkCmdDispatch(commandBuffer, numWorkGroups, 1, 1);
 vkEndCommandBuffer(commandBuffer);
 ...
 // Submit to queue
 VkSubmitInfo submitInfo -VK_STRUCTURE_TYPE_SUBMIT_INFO";
 submitInfo.commandBufferCount = 1;
 submitInfo.pCommandBuffers = &cmdBuffer;
 vkQueueSubmit(computeQueue, 1, &submitInfo ...);
 ... // Clean up and free all resources
```

Listing 1: VectorAdd host code using low-level Vulkan API

TABLE I: VComputeBench benchmarks

Name	Application	Dwarf	Domain
backprop	Back Propagation	Unstructured Grid	Deep Learning
bfs	Breadth-First Search	Graph Traversal	Graph Theory
cfd	CFD Solver	Unstructured Grid	Fluid Dynamics
gaussian	Gaussian Elimination	Dense Linear Algebra	Linear Algebra
hotspot	Hotspot Simulation	Structured Grid	Physics
lud	LU Decomposition	Dense Linear Algebra	Linear Algebra
nn	K-Nearest Neighbors	Dense Linear Algebra	Data Mining
nw	Needleman-Wunsch	Dynamic Programming	Bioinformatics
pathfinder	Path Finder	Dynamic Programming	Grid Traversal

Vulkan applications are linked against a common library referred to as the *loader*, which gets initialized at the time of VkInstance creation. The loader loads any enabled tooling layers and initializes the low-level driver provided by the GPU vendor. Accordingly, the example program depicted in Listing 1, starts initializing Vulkan by creating a VkInstance and querying the system for any available devices with all their properties including all available queue families.

Then a logical VkDevice is created and a queue is acquired. The next step is to create storage buffers for the vectors. VkBuffer objects are created, the system is queried for suitable heaps according to the buffer memory requirements, then memory is allocated on that heap and buffers are bounded to their allocated memory. Next, a compute VkPipeline is created by specifying the kernel's SPIR-V binary as its shader stage and creating a VkPipelineLayout describing all the resources used by that kernel. Then, the buffers are bound to the pipeline by specifying the kernel's binding value of each buffer as the destination binding of the write descriptor set. This is similar to specifying the kernel arguments in OpenCL using clSetKernelArg. Now that the compute pipeline is set up, the kernel can be launched by creating a VkCommandBuffer, binding the pipeline to that command buffer and recording the dispatch command with the number of workgroups to be launched. The command buffer is then submitted to the acquired queue for execution. Finally, the application waits for execution to finish then cleans up and frees all used resources and objects.

B. VComputeBench Benchmarks

The Rodinia suite includes both CUDA and OpenCL versions for each of its benchmarks. While developing their Vulkan equivalents, we made sure not to introduce any algorithmic changes to the kernel codes. In this way, we will be able to make fair comparisons in the sense that any differences in performance can be related to the programming model and not to the algorithm. By using the latest Rodinia version 3.1, we assume that we are already starting from a decent baseline since these benchmarks were optimized many times in several research works [28] [29].

The kernels were developed in GLSL and their corresponding SPIR-V binaries were automatically generated using the glslangvalidator compiler [30] provided by Khronos. We have chosen GLSL as our kernel language because it has the best support. We provide both the SPIR-V binaries and the GLSL

sources as part of our VComputeBench benchmarks. The host code translation on the other hand, was challenging because the Rodinia source code was collected from different sources resulting in a hard-to-read code with different styles, very little comments and hardly any documentation. We made sure this is not the case with our benchmarks, which we implemented using C++11 features with unified style and appropriate comments. As far as functional testing is concerned, we validated our developed VCompute benchmarks against both CUDA and OpenCL outputs for different input sets.

Our VComputeBench benchmarks cover a diverse range of application domains with different computation patterns. The benchmarks were selected so that they also cover different sets of dwarves [31]. Table I shows a list of the developed benchmarks including their dwarf and application domains. Here, we just include brief descriptions of these benchmarks, but full descriptions and characterizations of these workloads can be found at [10]:

***Back Propagation (bp)**:* is an algorithm that is commonly used in training deep neural networks to adjust the network's weights. It is composed of two phases a forward pass, where the activations are propagated from the input to the output layer, and a backward pass, where the error is propagated backwards from the output to the input layer to adjust the weights and bias values.

***Breadth-First Search (bfs)**:* is a graph algorithm that traverses or searches a graph of connected nodes, which could include millions of nodes. It starts at a root node and explores neighboring nodes first, before moving to the next level neighbors.

***Computational Fluid Dynamics (cfd)**:* is a fluid dynamics solver of three-dimensional Euler equations representing an unstructured grid, finite volume of compressible flow.

***Gaussian Elimination (gaussian)**:* is a linear algebra algorithm for solving a set of linear equations. It works by performing a sequence of row reduction operations on a matrix until the lower left-hand corner of the matrix is filled with zeros, as much as possible.

***Hotspot Simulation (hotspot)**:* is a thermal simulation tool that tries to estimate processor temperature based on an architectural floor plan and simulated power measurements.

***LU Decomposition (lud)**:* is an a linear algebra algorithm that tries to calculate the solution of a set of linear equations. It works by decomposing a matrix into a product of a lower triangular matrix and upper triangular matrix.

***K-Nearest Neighbors (nn)**:* is a dense linear algebra algorithm used to find the closest K neighbors in a set of reference data points in an n-dimensional space to query point q. The data in our case is latitude and longitude data and the calculated distances are euclidean distances.

***Needleman-Wunsch (nw)**:* is a dynamic programming algorithm that is used for DNA sequence alignment. The algorithm tries to fill a matrix of potential pairs of DNA sequences with scores, representing the value of the maximum weighted path ending at that cell. Then a trace-back process is used to search for an optimal alignment.

***Pathfinder (pfinder)**:* is another dynamic programming algorithm that computes the path on a 2-dimensional grid with the smallest total cost. The grid is represented as a matrix, and the path is computed in blocks of rows.

C. Vulkan-specific optimizations

As shown in in the example code in Listing 1, Vulkan uses completely different abstractions from CUDA and OpenCL. Effectively, in Vulkan, the programmer is not dealing with kernels, kernel arguments and kernel launches but they are dealing with low level command buffers, recording commands in these buffers such as binding compute pipelines, setting descriptor sets and binding buffers to descriptor sets. One of key synchronization mechanisms of Vulkan that we used when writing our benchmarks and produced performance improvements, as shown in section V-A2, is memory barriers. Memory barrier commands can be recorded in a command buffer, ensuring that commands recorded prior to it are executed before the commands recorded after it. This allowed us to reduce the kernel launch overhead compared to CUDA and OpenCL implementations, resulting in better performance as shown in sections V-A2 and V-B2.

Most of our benchmarks use iterative algorithms. The CUDA and OpenCL implementations invoke the kernel multiple times for every iteration, whereas in our Vulkan implementations we record the work of all iterations in one command buffer and synchronize using memory barriers between iterations, instead of naively creating a command buffer for every iteration. Effectively, we incur only a single communication overhead when the command buffer is submitted compared to the CUDA and OpenCL implementations which incur kernel launch overheads on every iteration.

One can argue that the CUDA and OpenCL implementations can be changed to enqueue iterations ahead of time without blocking. The problem with this solution is that it does not honor the data dependencies between iterations. Subsequent iterations depend on the data generated in previous iterations. Both CUDA and OpenCL do not offer any inter-workgroup synchronization mechanism that can be used to honor these dependency requirements. This is a well known limitation of these programming models and the safest portable solution to achieve such synchronization is to use what's called *multi-kernel* method. In this method the application is split into multiple kernels. Whenever a inter-workgroup synchronization is required, a transition from one kernel to another is made or in the case of having only one kernel this kernel is launched again. The transfer of control from the GPU to the CPU implicitly provides the required barrier semantics. The Rodinia CUDA and OpenCL implementations use this method to achieve such inter-workgroup synchronization and satisfy the data dependencies between iterations.

D. Porting to mobile devices

One of the major strengths of Vulkan is its portability. However, performance improvements are not necessarily portable

TABLE II: Desktop GPUs Experimental Setup

	NVIDIA GTX105Ti	AMD RX560
Operating System	Ubuntu 16.04 64-bit	
CPU	Intel(R) Core(TM) i5-2500K CPU 3.30GHz x4	
Memory	CPU Memory=16 GB, GPU Memory=4GB	
Driver	Linux Display Driver 381.22	AMDGPU-Pro Driver 17.10
OpenCL	OpenCL 1.2	OpenCL 2.0
CUDA	CUDA 8.0	-
Vulkan	API Version 1.0.42	API Version 1.0.37

and often developers have to adapt and re-write their applications with respect to the targeted architecture. In fact, it has been shown that performance is not portable when running OpenCL applications targeting GPUs on CPU or FPGA like architectures [17] [21]. To address this concern and assess whether Vulkan is a good candidate for GPGPU computing on mobile devices, we ported our benchmarks plus their corresponding Rodinia OpenCL implementations onto mobile GPUs. We chose Android 7.0 as our OS because it supports Vulkan out of the box, allowing us to target many mobile GPUs. We cross-compiled all of our benchmarks for x86, x86-64, armeabi-v7a, arm64-v8a binary targets and developed an Android application that bundles these benchmarks with their required data sets. We set a requirement when developing the VComputeBench Android application of not requiring root access so that it can be released on the Android application store allowing millions of users to check and compare the performance of the GPUs and Vulkan implementations inside their devices. This was challenging and we had to resort to bundling the benchmarks as libraries in order to satisfy Android security restrictions on binary executables.

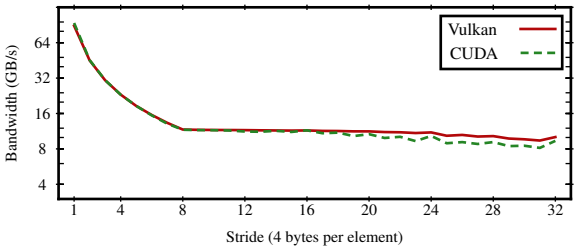
V. Experimental Results

In this section we report the results of our empirical evaluation of Vulkan performed on several GPU architectures. We use two types of benchmarks self-written micro benchmarks to highlight and assess specific attributes and our VComputeBench plus Rodinia benchmarks to assess performance using representative real world applications. We compare Vulkan results to those of CUDA and OpenCL on two desktop GPUs and two mobile GPU platforms. For consistency, we measure the execution times on the CPU using C++11 std::chrono. To minimize measurement errors, we execute several times and report the average of the obtained execution times.

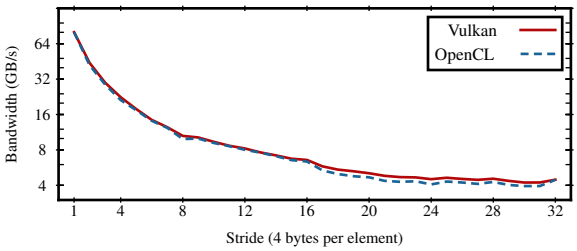
A. Evaluations on Desktop Platforms

We chose two recent desktop GPUs employing latest and advanced GPU architectures: NVIDIA GTX1050Ti employing NVIDIA's Pascal architecture and AMD RX560 employing AMD's Polaris architecture. Table II shows the configuration details of these platforms.

1) Memory Bandwidth Evaluation: To evaluate how the programming model affects memory bandwidth and asses whether we can achieve high memory bandwidth when using Vulkan, we developed a strided memory access micro-benchmark in



(a) NVIDIA GTX1050Ti



(b) AMD RX560

Fig. 1: Vulkan memory bandwidth vs CUDA and OpenCL

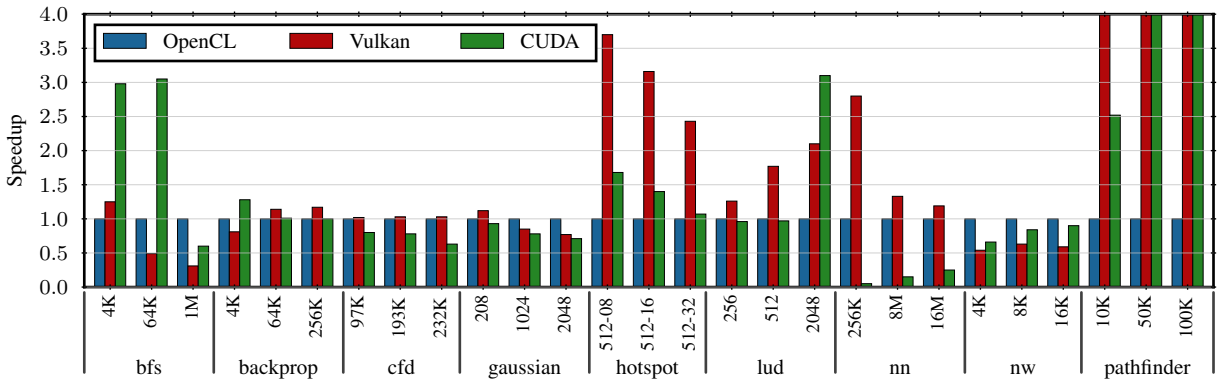
Vulkan, CUDA and OpenCL. We vary the stride when reading array elements and measure the achieved bandwidth. For reference, both of our platforms use GDDR5 memory with an effective memory clock of 7GHz and 128 bit memory interface width, resulting in theoretical bandwidth of 112 GB/s, which can be calculated using:

$$BW_{peak} = Freq \cdot (BusWidth/8) \cdot 10^{-9}$$

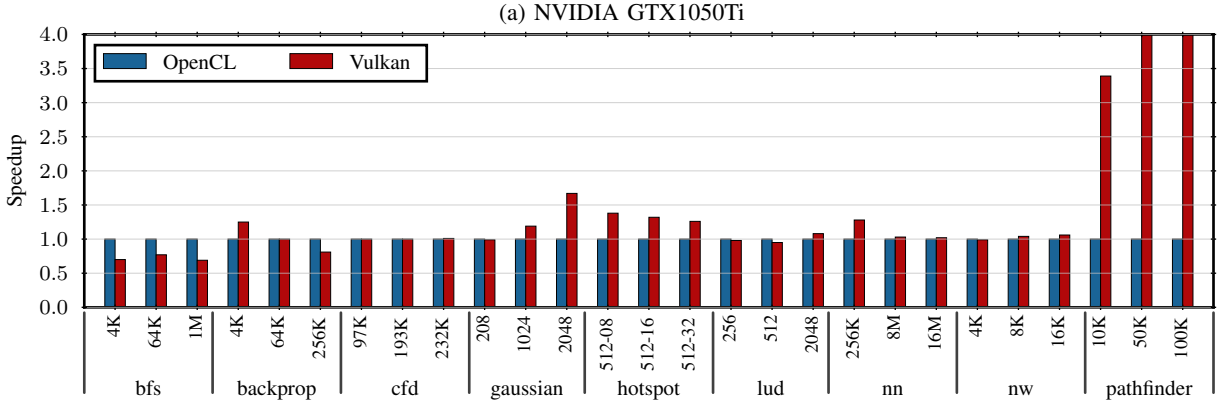
The obtained results are shown in Figure 1. On both platforms, Vulkan provides comparable performance to CUDA and OpenCL for strides less than 64 bytes and slightly better performance for strides larger than 64 bytes. As expected, unit stride provides maximum achieved bandwidth of 84% and 79.6% of the peak bandwidth for CUDA and Vulkan respectively on the GTX1050. Likewise, on the RX560, Vulkan achieves 71.6% of the peak bandwidth compared to 71.5% for OpenCL. Overall, this test shows that high memory bandwidth can be attained using Vulkan and data layout in memory is more important than the used programming model.

2) Benchmarks Evaluations: Figure 2 shows the speedup results of the selected benchmarks comparing Vulkan, CUDA and OpenCL for different workloads. We chose OpenCL as our baseline for speedup calculations because it is supported on both platforms. To make a fair comparison, we only report kernel execution times not total benchmark times because a high overhead is generally exhibited by OpenCL JIT compilation and explicit context management resulting in longer total times [32] [17].

Overall, for most benchmarks Vulkan provides better performance than CUDA and OpenCL resulting in geometric mean speedups of $1.53x$ with respect to CUDA on the GTX1050 and $1.26x$ with respect to OpenCL on the RX560. However,



(a) NVIDIA GTX1050Ti



(b) AMD RX560

Fig. 2: Vulkan speedup vs CUDA and OpenCL for the Rodinia benchmarks

since the benchmarks exhibit different computation patterns, there are variations on their individual results.

The best speedups are attained with *pathfinder*, *hotspot*, *lud* and *gaussian* benchmarks. The reason for this is that these benchmarks use iterative algorithms, invoking the kernel multiple times. Subsequent invocations utilize data generated in previous iterations, requiring control to return back to the CPU and incurring kernel launch overhead on every iteration. Vulkan enable us to eliminate these kernel launches and communication overheads altogether by recording the work of all iterations in one command buffer and adding memory barriers between iterations to satisfy the dependency requirements. Effectively, we incur a single communication overhead when the command buffer is submitted. Our results commensurate with the kernel launch overhead findings of [15]. Figure 2 also shows that, for most of these workloads, the speedup increases as we increase the input size. Larger input means more iterations and less overhead compared to CUDA and OpenCL, thus better Vulkan performance.

An interesting result is that of *cfd*. Although it uses an iterative algorithm, we do not get similar speedups. This benchmark has 3 compute intensive kernels and for every iteration we have to bind 3 different compute pipelines, representing these kernels, to our single command buffer. This overhead of binding compute pipelines plus the longer kernel computation times make the launch overhead savings not that significant. It also does not scale well with input size because the number of iterations is fixed and not dependent on input size. Vulkan *cfd* achieves $1.38x$ speedup vs CUDA and $1.04x$ speedup vs OpenCL averaged on both platforms.

On the contrary, we get a slowdown for *bfs* on both platforms. To investigate this, we disassembled the Vulkan and OpenCL kernels using the AMD CodeXL tool [33]. We discovered that the OpenCL generated ISA code is optimized to use workgroup local memory compared to the Vulkan generated ISA, which uses plain buffer loads from global memory. This optimization of memory accesses significantly affects performance because *bfs* is memory-bound [34]; it predominately performs loads and stores with very few ALU operations. Although we use the same driver, the generated ISA is different for Vulkan. We can therefore deduce that the Vulkan SPIR-V compiler inside the driver is not as mature as the OpenCL one. This is expected as Vulkan was recently released and support will improve in the future.

The remaining benchmarks *backprop*, *nn* and *nw* do not involve any dependencies between kernel invocations. The Vulkan implementations record these kernels onto different command buffers and submits them simultaneously to the GPU resulting in pretty much similar performance to CUDA and OpenCL with slight variations between the platforms.

TABLE III: Mobile GPUs Experimental Setup

	Qualcomm Snapdragon 625	Google Nexus Player
Operating System	Andorid 7.0	Andorid 7.1
CPU	ARM Cortex A53 x8	Intel Atom(TM) x4
GPU	Adreno 506	Rogue G6430
OpenCL	OpenCL 2.0	OpenCL 1.2
Vulkan	API Version 1.0.20	API Version 1.0.30

B. Evaluations on Mobile Platforms

We used two platforms: Google's Nexus Player and Qualcomm's Snapdragon 625 employing the Imagination G6430 and the Adreno 506 GPUs respectively. The platforms were chosen because both GPU vendors provide unofficial OpenCL support[3]. Table III summarizes the configuration details of these two platforms.

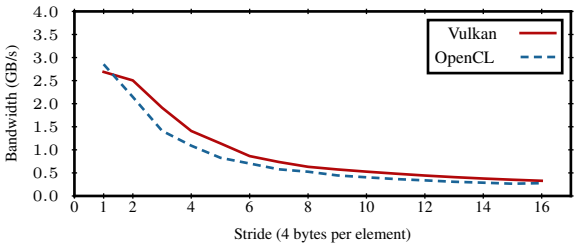
1) Memory Bandwidth Evaluation: We run the same strided memory access micro benchmark, described in section V-A1, on our selected mobile platforms. The obtained results are shown in Figure 3. On the Nexus platform OpenCL achieves a bandwidth of 2.85 GB/s at unit stride, whereas Vulkan only achieves 2.69 GB/s, resulting in about 89% and 84% of peak bandwidth respectively. Then for strides larger than 4 bytes, Vulkan surprisingly performs slightly better than OpenCL. However, on the Snapdragon platform, Vulkan performs worst than OpenCL at strides less than 16 bytes but we get pretty much the same bandwidth for strides above 16 bytes. We suspect that the Snapdragon driver doesn't properly support Vulkan's push constants, that we use to set the stride constant inside the command buffer when varying the stride number, and treating them as normal storage buffers instead. This can result in worst performance because binding these buffers is required for every iteration. For larger strides this effect becomes negligible due to the fact that the exhibited execution times are longer. Overall, the main observation we can make here is that on these mobile platforms, Vulkan can provide comparable performance to OpenCL but with slight degradation and again data layout in memory is more important than the used programming model.

2) Benchmarks Evaluations: Due to memory size restrictions on these platforms, we had to choose smaller workload input sizes. *cfd* could not fit on both platforms as it uses larger data sets describing flux flow data. Also the *backprop* OpenCL and Vulkan implementations failed to run on Nexus and on Snapdragon only the *lud* OpenCL failed because of driver issues. The results are shown in Figure 4.

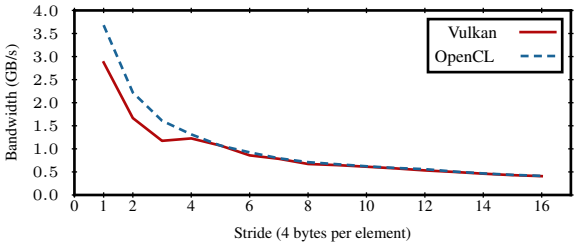
Figure 4 shows that Vulkan does well on Nexus compared to Snapdragon, achieving geometric mean speedups of $1.59x$ on Nexus and $0.83x$ on Snapdragon. On the Nexus platform, Vulkan shows speedups across most benchmarks except *hotspot*, which pretty much commensurate with the results obtained on desktop GPUs. The best speedups are again attained with *pathfinder*, *gaussian* and *lud* benchmarks because of minimizing the kernel launch overhead. On the snapdragon platform, further investigations are required to explain the exhibited slowdown. However, since all benchmarks exhibited slowdowns except *pathfinder*, we think this can be related to the immaturity of the Vulkan drivers on this platform compared to the OpenCL ones. We expect this will improve in the future as better Vulkan support is rolled out.

Overall these results are very interesting in the sense that they demonstrate that performance portability is not necessarily guaranteed, even though the programming model is portable. We can conclude that Vulkan performance improvements can be portable to mobile GPUs as long as there is good driver support from vendors.

[3]The OpenCL library on the Nexus player is not even called libOpenCL.so. It is provided as libpvrcpt.so.



(a) Nexus Player



(b) Snapdragon 625

Fig. 3: Vulkan memory bandwidth vs CUDA and OpenCL

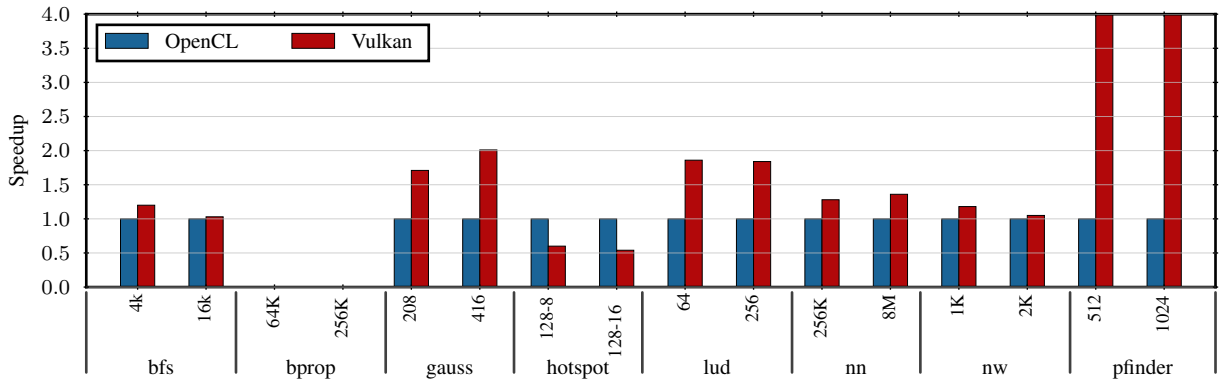
VI. Discussion

A. Vulkan Limitations

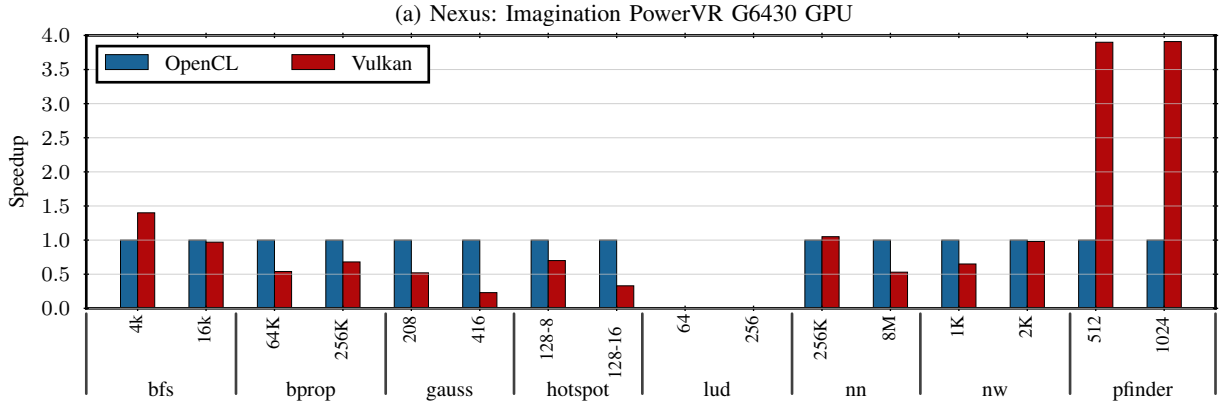
As you may have observed from the example application described in Listing 1, the key limitation of Vulkan is its *verbosity*. Vulkan's low-level nature makes it very verbose with a high programming effort. For example, to create a simple buffer one has to:

- Create a buffer object
- Get the memory requirements for that object
- Decide which memory heap to use
- Allocate memory on the chosen heap
- Bind the buffer object to the memory allocation

This simple buffer creation requires about 40 lines of code in Vulkan compared to just one line in CUDA or OpenCL, where cudaMalloc and clCreateBuffer are used respectively. In addition, Vulkan's principle of explicit control pushes a lot of responsibility onto the programmer. The application layer is proportionally more complex. Programmers have to deal



(a) Nexus: Imagination PowerVR G6430 GPU



(b) Snapdragon: Qualcomm Adreno 506 GPU

Fig. 4: Vulkan speedup vs OpenCL on mobile devices

with issues such as memory allocation, resources tracking, object creation and destruction and so on. Experience shows that programming in such style can be error-prone and less productive. Vulkan's verbosity and the additional responsibility it imposes on the programmer introduce issues with productivity and hence can be a burden to adopting it as a GPGPU programming model.

B. Recommended Vulkan Optimizations

Vulkan introduce some low-level controls that can be utilized for extra performance. As a takeaway from our experience writing the VComputeBench benchmarks, we recommend the following for better Vulkan performance :

- For iterative algorithms, use one single command buffer and synchronize using memory barriers. This proved to be effective in our evaluations.
- For parameter changes of small data types, it is better to use PushConstants rather than binding a whole parameters buffer. Push constants are specific to a pipeline. For instance on GTX1050 and RX560 you get maximum sizes of 256B and 128B respectively. On both Nexus and Snapdragon platforms you get a maximum of 128 bytes.
- Try to minimize going back to the CPU for control and leverage Vulkan's synchronization primitives to stay as much as possible on the GPU.
- For large memory transfers use transfer queues. These specific transfer queues should be used for large copy commands as they are usually tied to DMAs inside the hardware.
- For better workload balancing, make use of multiple compute queues whenever possible. This will give the GPU's scheduler more room for manoeuvre resulting in better utilization.

VII. Conclusion

This paper presented Vulkan as new programming model for cross-platform GPGPU computing notably on mobile and embedded GPUs. We developed a set of compute benchmarks by extending the Rodinia suite with Vulkan benchmarks and used them to evaluate this emerging programming model.
Indeed, Vulkan's low-level control over the underlying hardware offers opportunities for better performance. Our results show that, by exploiting Vulkan's synchronization mechanisms, average speedups of $1.53x$ and $1.66x$ versus CUDA and OpenCL were attained across the selected benchmarks. We also, show that similar performance improvements can be seen on some mobile GPU architectures but performance portability is not necessarily guaranteed. Issues such as driver support and implementation quality come into play.
Finally, we illustrate that these performance improvements come at a cost manifested in a high programming effort. These

programmability issues can be a burden to adopting Vulkan as a GPGPU programming model. Directions for future work could include improving the programmability of this emerging programming model.

Acknowledgment

This material is based upon work supported by the European Union Horizon 2020 research and innovation programme under Grant No.688759, Project LPGPU2.

References

[1] J. D. Owens, D. Luebke, N. Govindraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A Survey of General Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, pp. 80–113, 2006.

[2] Nvidia Corporation, "CUDA Toolkit Documentation," 2017. [Online]. Available: http://docs.nvidia.com/cuda/

[3] The Khronos OpenCL Working Group, "The OpenCL Specification," 2017. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.html

[4] OpenMP Architecture Review Board, "OpenMP Application Programming Interface," 2015. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[5] The OpenACC Standard.org, "The OpenACC Application Programming Interface," 2015. [Online]. Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final-changes.pdf

[6] The Khronos Vulkan Working Group, "The Vulkan Specification," 2017. [Online]. Available: https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html

[7] J. Kessenich, B. Ouriel, and R. Krisch, "SPIR-V Specification," 2017. [Online]. Available: https://www.khronos.org/registry/spir-v/specs/1.2/SPIRV.html

[8] N. M. Dongre, "A Research On Android Technology With New Version Naugat(7.0,7.1)," *IOSR Journal of Computer Engineering*, vol. 19, no. 02, pp. 65–77, 2017.

[9] T. Linux Foundation Project, "Tizen 3.0 Public M2 Release Notes," 2017. [Online]. Available: https://developer.tizen.org/tizen/tizen/release-notes/tizen-3.0-public-m2

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, S. Lee, J. W. Sheaffer, and K. Skadron, "A Benchmark Suite for Heterogeneous Computing," *IEEE International Symposium on Workload Characterization*, pp. 44–54, 2009.

[11] J. a. Stratton, C. Rodrigues, I.-j. Sung, N. Obeid, L.-w. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *IMPACT Technical Report*, 2012.

[12] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite Categories and Subject Descriptors," *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 63–74, 2010.

[13] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for CPU-GPU collaborative computing," in *Proceedings of the 2016 IEEE International Symposium on Workload Characterization, IISWC 2016*, 2016, pp. 13–22.

[14] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *ArXiv e-prints*, vol. arXiv, no. 1, p. 1005.2581, 2010.

[15] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," *Proceedings of the International Conference on Parallel Processing*, pp. 216–225, 2011.

[16] R. Sachetto Oliveira, B. M. Rocha, R. M. Amorim, F. O. Campos, W. Meira, E. M. Toledo, and R. W. dos Santos, "Comparing CUDA, OpenCL and OpenGL Implementations of the Cardiac Monodomain Equations." Springer, Berlin, Heidelberg, 2012, pp. 111–120.

[17] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[18] C.-L. Su, P.-Y. Chen, C.-C. Lan, L.-S. Huang, and K.-H. Wu, "Overview and comparison of OpenCL and CUDA technology for GPGPU," in *2012 IEEE Asia Pacific Conference on Circuits and Systems*. IEEE, 12 2012, pp. 448–451.

[19] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee, "Bridging OpenCL and CUDA," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*, no. November, pp. 1–12, 2015.

[20] H. C. D. Silva, F. Pisani, and E. Borin, "A Comparative Study of SYCL, OpenCL, and OpenMP," *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pp. 61–66, 2016.

[21] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing OpenCL applications on FPGAs," in *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2016-April, 2016, pp. 114–125.

[22] A. Sampson, "Let's Fix OpenGL," *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, vol. 71, pp. –, 2017.

[23] A. Blackert, *Evaluation of Multi-Threading in Vulkan*. Linköping University, 2016.

[24] J. Kessenich, "SPIR-V A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels," 2015. [Online]. Available: https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf

[25] G. Wang and Y. Xiong, "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU-a case study," *IEEE International Conference on*

Acoustics, Speech and Signal Processing, 2013.

[26] M. M. Trompouki, L. Kosmidis, and U. Polit, "Optimisation Opportunities and Evaluation for GPGPU applications on Low-End Mobile GPUs," *Date*, pp. 950–953, 2017.

[27] L. Tobias, A. Ducournau, F. Rousseau, G. Mercier, and R. Fablet, "Convolutional Neural Networks for object recognition on mobile devices: A case study," *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 3530–3535, 2016.

[28] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *IEEE International Symposium on Workload Characterization, IISWC'10*, 2010.

[29] G. Misra, N. Kurkure, A. Das, M. Valmiki, S. Das, and A. Gupta, "Evaluation of rodinia codes on Intel Xeon Phi," in *Proceedings - International Conference on Intelligent Systems, Modelling and Simulation, ISMS*, 2013, pp. 415–419.

[30] The Khronos Group, "Glslang Reference Compiler," 2017. [Online]. Available: https://github.com/KhronosGroup/glslang

[31] K. Asanovic, B. C. Catanzaro, D. Patterson, and K. Yelick, "The Landscape of Parallel Computing Research : A View from Berkeley," Tech. Rep., 2006.

[32] J. H. Lee, N. Nigania, H. Kim, K. Patel, and H. Kim, "OpenCL Performance Evaluation on Modern Multicore CPUs," *Scientific Programming*, vol. 2015, pp. 1–20, 10 2015.

[33] GPUOpen AMD, "CodeXL Tool Suite," 2017. [Online]. Available: https://github.com/GPUOpen-Tools/CodeXL

[34] S. Lal, J. Lucas, and B. Juurlink, "Eˆ2MC: Entropy Encoding Based Memory Compression for GPUs," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 5 2017, pp. 1119–1128.