Original software publication

**vkpolybench**: A crossplatform Vulkan Compute port of the PolyBench/GPU benchmark suite

Nicola Capodieci *, Roberto Cavicchioli

*University of Modena and Reggio Emilia, Department of Physics, Informatics and Mathematics, Via Giuseppe Campi, 213/a, 41125 Modena, Italy

**A R T I C L E I N F O**

**Article history:**
Received 9 March 2020
Received in revised form 22 July 2021
Accepted 6 August 2021

**Keywords:**
GP-GPU computing
Heterogeneous architectures
High performance computing

**A B S T R A C T**

PolyBench is a well-known set of benchmarks characterized by embarrassingly parallel kernels able to run on Graphic Processing Units (GPUs). While Polybench GPU kernels leverage well-established GP-GPU APIs such as CUDA and OpenCL, in this paper we present **vkpolybench**, a crossplatform PolyBench/GPU port built on top of Vulkan. Vulkan is the recently released Khronos standard for heterogeneous CPU–GPU computing that is gaining significant traction lately. Compared to CUDA and OpenCL, the Vulkan API improves GPU utilization while reducing CPU overheads.

© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

**Code metadata**

Current code version 1.0
Permanent link to code/repository used for this code version https://github.com/ElsevierSoftwareX/ SOFTX_2020_86
Legal Code Licence BSD-3-Clause
Code versioning system used git
Software code languages, tools, and services used C++, Vulkan, GLSL
Compilation requirements, operating environments & dependencies Linux x86_64 and ARM v8, Windows 10 x86_64, Android 9 ARM v8.
If available Link to developer documentation/manual See README.md in the repo.
Support email for questions nicola.capodieci@unimore.it

1. **Motivation and significance**

The original version of PolyBench [1] was released 10 years ago and since then it is becoming one of the most commonly used reference set of benchmarks for compiler optimizations. Its benchmarks are characterized by highly memory intensive computations aimed at tackling common problems in linear algebra, statistics and numeric solvers, hence becoming appetizing to a wide variety of applications and research domains. In addition to compiler optimizations [2], other research domains are performance [3] and power consumption modelling [4]. PolyBench was later renamed PolyBench/C and due to the embarrassingly parallel nature of all of its constituent benchmarks, a GPU-accelerated version of PolyBench (PolyBench/GPU) [5,6] was released in 2012, providing GPU benchmark implementations that leverages well-established General Purpose computing for Graphic Processing Unit (GP-GPU) Application Programming Interfaces (APIs). GP-GPU APIs in PolyBench/GPU 1.0 are: CUDA (Compute Unified Device Architecture) [7] and OpenCL (Open Computing Language) [8]. CUDA is a widely adopted NVIDIA proprietary standard for heterogeneous GP-GPU programming, while OpenCL is an open standard for heterogeneous computing for massively parallel architectures, which is maintained by the Khronos Group.¹ Exploiting GP-GPU acceleration in mobile and embedded devices is sometimes hindered by the limited or absent support for CUDA and OpenCL. CUDA’s proprietary and closed nature implies that a CUDA application can only run on NVIDIA GPU devices. On the contrary, OpenCL’s open nature allows application developers to target generic accelerators other than NVIDIA GPUs. However, a recent market analysis [9] shows that the actual support of OpenCL in mobile systems is limited to a significantly low percentage of commercially available devices. In order to improve the portability of the PolyBench benchmark suite, we therefore present **vkpolybench**, a port of PolyBench/GPU

---

¹ https://www.khronos.org/.

**https://doi.org/10.1016/j.softx.2021.100793**

© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
built on top of the Vulkan Compute pipeline. Vulkan [10] is the last recently released open standard for GPU programming. Even if Vulkan has been proposed as both a graphics and a compute API, the Vulkan model is mostly agnostic to which of these two pipelines will be used in an application. Vulkan promises to be widely supported across different hardware and operating systems and, compared to OpenCL and CUDA, it enables a much finer control of CPU–GPU interactions. Vulkan achieves this by exposing a thinner layer of abstraction between the application and the device driver aiming at dramatically reducing CPU activity during commands submission. Other advantages of Vulkan with respect to OpenCL is the support for device-side advanced features such as tensor operations for applications in neural network inference.

Due to its recent release, only one Vulkan Compute benchmark is currently available as an open-source contribution: VComputeBench[11]. VComputeBench focuses on mobile and embedded systems and is designed to be characterized by compute intensive operations, whereas PolyBench kernels are known to feature a much higher memory-to-compute ratio [13,14]. We also highlight that the vkpolybench main module (vkcomp) significantly simplifies access to the otherwise extremely complex Vulkan API, so to minimize the effort needed for porting or extending additional compute benchmarks.

2. Software description

Compared to CUDA and OpenCL, implementing a Vulkan application is a significantly harder task [11,15–17]: the additional control over the thin driver layer exploited by the Vulkan API has a considerable cost in implementation complexity due to a much more involved coding style. Nevertheless, vkpolybench closely follows the typical blueprint of a heterogeneous CPU–GPU application which usually consists of the CPU (host) compiling a GPU kernel, orchestrating data movements from host to GPU (device) and dispatching compute kernels. In a Vulkan enabled application additional steps are needed: namely, pipelines’ creation and command buffer recording.

A pipeline is a pre-compiled description for compute kernels (also known as Vulkan terminology as compute shaders). Within a Vulkan pipeline the application developer is able to bind to each kernel its input and output data buffers as well as the launch configuration. In GP-GPU programming a launch configuration describes the degree of parallelism in which the work must be computed over a grid of parallel threads in the GPU.

Recording a command buffer means specifying in advance the sequence of commands (pipeline selection, kernel invocations and data buffer movements) to be submitted to the GPU. Preparing in advance pipelines and command buffers are the Vulkan features responsible for the minimization of the CPU–GPU driver interactions during the runtime execution of the application.

In this context, vkpolybench dramatically simplifies the usage of all of these Vulkan-specific artefacts so to be able to provide a clean, extensible and faithful port of all the 15 constituent benchmarks of the PolyBench/GPU 1.0 suite.

2.1. Software architecture

The software blueprint that characterizes vkpolybench is depicted in Fig. 1.

vkcomp is the core module within vkpolybench. It is divided into three sub-modules: (1) Vulkan Compute Interface, (2) Debug and Validation layers and (3) Shader Compiler interface.

The Vulkan Compute Interface is responsible for the creation of the Vulkan context, which itself manages the allocations for data, command buffers and pipeline state objects. It acts as a library to the compute pipeline of the Vulkan API that facilitates setting up all the necessary constructs able to transparently handling all the complex software artefacts we briefly summarized in the previous sections. A more in depth explanation can be found in [16] and [17].

The sub-module for the Debug and Validation layers exploits Vulkan’s layer-based mechanism for intercepting all or any subset of the API entry points, so to provide a custom level of debugging and validation for the benchmarks. The shader module compiler interface enables the compilation of binary Vulkan shader modules starting from compute shaders in SPIR-V [18] (Standard Portable Intermediate Representation) binary format to GLSL [19] compute shaders. The former is natively understood by the Vulkan API, but it is more of an intermediate format rather than a development language. On the other hand, being similar to both CUDA and OpenCL kernel language, GLSL is the language of choice for the developers of device code. In order to translate a GLSL kernel (.comp file) into a SPIR-V binary file (.spv file) the glslc executable is used.

This executable and the Debug & Validation layers definitions are installed as part of the only vkpolybench external dependency, the LunarG Vulkan SDK.3

The benchmarks macro-module contains the Vulkan implementation of the 15 constituent benchmarks of the PolyBench suite and the functions for timing measurements. For each benchmark, this sub-module provides for the single core CPU reference implementation, the Vulkan Host Code and the Vulkan device code.

The single core CPU reference is the baseline implementation of the compute operations of each benchmark. It is identical to the PolyBench/C original version and it is used to provide both a timing baseline comparison with respect to the GPU accelerated version and a sanity check.

The Vulkan Host Code sub-module exploits the vkcomp interface to implement the steps needed to compile a GLSL shader for the specific benchmark, orchestrate data movements between host and device and launch the GPU kernels.

The Vulkan Device Code sub-module contains, for each benchmark, the GLSL code for the device compute kernel(s). Kernels’ parameters, such as data types, and problem size are shared in a host and device common header file. The GLSL code implementation is a line-by-line translation from the CUDA kernels taken from PolyBench/GPU 1.0.

2.2. Software functionalities

Cross-compatibility: starting from the original GNU/Linux-only PolyBench/GPU 1.0 implementation, vkpolybench runs on every major combination of operating system and hardware platform, provided that a functioning Vulkan driver is made available by the device vendor. This includes most, if not all the recent x86 and ARM-based processors coupled with both discrete and integrated GPU devices for three most commonly installed Operating Systems (Microsoft Windows, Linux and Android based

---

2 Described in https://github.com/google/shaderc/tree/master/glslc.
3 Available at https://www.lunarg.com/vulkan-sdk/.
Benchmarking: vkpolybench implements all the 15 constituent benchmarks of the PolyBench/GPU 1.0. For all the benchmarks, tuning the test parameters for host and device code (e.g. problem size, constants and data types) can be done by modifying the macros defined in the header files for each benchmark. Benchmarks' parameter can be tuned by modifying the macros listed in Table 1.

Extensibility: vkpolybench's wrapper module (vkcomp) has been engineered to drastically simplify access to an otherwise extremely complex API (Vulkan). As a consequence of that, creating a new benchmark to be added to the suite is as practical as writing a CUDA or OpenCl benchmark implementation from the original PolyBench. In the next section an example for this feature is provided.

3. Illustrative examples: adding a benchmark

Let us suppose to add a $xAXPY$ kernel as an added benchmark to vkpolybench. As taken from the list of BLAS [20] (Basic Linear Algebra Subprograms), $xAXPY$ stands for $x$-precision $A \cdot X + Y$, with $A$ being a scalar constant, and $X$ and $Y$ being $N$ dimensional vectors. In a header file, we define the constants $A$ and $N$ and the data type (i.e. float, so to benchmark a $xAXPY$ as in single-precision $AXPY$). Then, we write the GLSL kernel as shown in Listing 1. According to Listing 1, $xAXPY$ kernel's entry point is a void main() procedure. Line 20 shows how GPU threads obtain their ID within the $X$ dimension on the compute grid. Its CUDA equivalent is threadIdx.x+blockDim.x*blockIdx.x; in OpenCl one should write get_global_id(0). Actual computations (lines from 22 to 25) follow the same rules and semantics of a CUDA or OpenCl kernel, with the way in which GLSL defines its input buffers (lines from 11 to 15) being the only difference: input buffers must be embedded within a layout struct and each buffer must be bound to an integer. In Listing 2 the host code for $xAXPY$ is presented; do note that error checking, timing measurements and necessary safe type casting are omitted for brevity. Host-side, the previously written GLSL kernel can be compiled with the loadAndCompileShader method of a VulkanCompute instance which is the core part of the vkcomp module (lines 2, 3 and 4). Input buffers $X$ and $Y$ are then allocated for both host and device (lines 5 to 7). Then a pipeline bound to the $xAXPY$ shader must be created within the two methods named startCreate/ finalizePipeline: within these methods, we indicate the launch configuration and the kernel argument binding points (lines from 13 to 17). The launch configuration is described through a data structure characterized by two tuples of three integer values each (lines 10 and 11). The kernel's arguments are bound with the setArg method, which takes as input the buffer layout binding integers (values 4 and 5 in lines 14 and 15). Then, delimited by the startCreate/finalizeCommandList we record in advance the sequences of commands to be later submitted to the GPU: specifically, we orchestrate the movements between host and device of the $X$ and $Y$ buffers (lines 21, 22 and 24), we select the previously constructed pipeline (line 20) and we launch the $xAXPY$ kernel (line 23). Actual GPU command submission is triggered by the submitWork method.
if (i<N)
    Y.v[i] = DATA_TYPE(A) * X.v[i] + Y.v[i];
}

Listing 1: Device code (xaxpyKernel.comp)

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  

Listing 2: Host code (xaxpy.cpp)

4. Impact and conclusions

We believe that vkpolybench has the potential to become a welcomed addition to the commonly used set of benchmarks in heterogeneous computing. The reasons for this can be found in the improved support that the Vulkan API offers to the end users in terms of supported platforms, the easiness in which vkpolybench can be extended despite Vulkan intrinsic complexity and the overall ever-increasing adoption of Vulkan as a more open GP-GPU solution compared to CUDA. Examples of Vulkan early adopters can be found in HPC literature [15], predictable Real-Time systems [16] and power consumption modelling in heterogeneous platforms [21]. In all these works, improvements compared to the current state-of-the-art have been observed.

Commercial exploitation is also a possibility for vkpolybench: as the interest in Vulkan grows, the well-known Geekbench\(^4\) suite has recently seen its fifth major release with Vulkan support. vkpolybench maintains the same level of cross-compatibility, but as opposed to Geekbench, our contribution is free, open-source and features a higher number of compute benchmarks (15 vs 11). We compared vkpolybench with the PolyBench/GPU OpenCL and CUDA implementations in a variety of different platforms. For space constraints, we refer the reader to the vkpolybench git repository in which several test results are shown.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The research leading to these results has received funding from the European Union’s Horizon 2020 Programme under the CLASS Project (https://class-project.eu/), grant agreement 780622.

References