

# Performance Analysis of the OpenCL Environment on Mobile Platforms

Przemysław Falkowski-Gilski<sup>a</sup> and Maciej Plewka<sup>b</sup>

*Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology,  
Narutowicza 11/12, Gdansk, Poland*

**Keywords:** Android OS, Mobile Devices, Multimedia Content, OpenCL, OpenGL, UX (User Experience).

**Abstract:** Today's smartphones have more and more features that so far were only assigned to personal computers. Every year these devices are composed of better and more efficient components. Everything indicates that modern smartphones are replacing ordinary computers in various activities. High computing power is required for tasks such as image processing, speech recognition and object detection. This paper analyses the performance of the OpenCL (Open Compute Language) environment on mobile devices, which is a library dedicated to high-speed parallel computing. This paper examines how smartphones can access a library that, as it turned out, is not officially supported on the Android platform, and briefly describes the evaluated library. As a part of the study, this API (Application Programming Interface) was tested in the context of the achieved computing power, memory flow rate, speed of matrix multiplication and the possibility of processing the image from the camera in real-time. The obtained results were presented in graphical format, described and commented. We also provide an insight on applications that use this API for teaching deep neural networks, image processing, etc.

## 1 INTRODUCTION

OpenCL (Open Compute Language) is a standard created by the Khronos group (Munshi, 2009), used to write programs that can be executed on various platforms such as CPU (Central Processing Unit), GPU (Graphics Processing Unit) or FPGA (Field Programmable Gate Array). The OpenCL specification defines an interface in C++ that allows an application to be programmed to execute specific code on a selected device (Seo et al., 2011; Jääskeläinen et al., 2015; Aydonat et al., 2017).


The OpenCL standard is mainly used for parallel computations such as vector math and image processing (Tay, 2013; Fang et al., 2020). The device manufacturer is responsible for the implementation of the driver that issues the API (Application Programming Interface), according to the version defined in the specification.

Due to the fact that the standard is open and most manufacturers have its implementation, it is possible to create a code that can be run regardless of the architecture or manufacturer of the central or graphics

processor. This is a major advantage compared to, e.g., CUDA (Compute Unified Device Architecture), which is an interface only implemented by NVidia (Karimi et al., 2010; Fang et al., 2011). The OpenCL standard is continuously developed and modified (Keryell et al., 2015). As a result, the API is defined in several versions.

While performing tests, the latest version of the specification was version 3.0. Of course all versions are backwards compatible. Additionally, there are extensions, such as `cl_khr_gl_sharing`, defining the API to share objects between OpenCL and OpenGL. Such additional API is also specified by the Khronos group within a specific OpenCL version, but this is not obligatory.

There are also manufacturer specific API extensions, such as `cl_intel_mem_force_host_memory`, which are available on Intel devices, or `cl_qcom_android_native_buffer_host_ptr`, available on Qualcomm's Android processors. Such additional API complements the core, allowing the specification to better match to particular hardware. OpenCL on the Android platform (Gilski and Stefański, 2015) is

<sup>a</sup>  <https://orcid.org/0000-0001-8920-696>

available only from the native C++ library, whereas on MacOS devices, this environment is not supported.

## 2 APPLICATIONS UNITIZING

### OpenCL

OpenCL is a library that can certainly be utilized in various Android applications. The lack of official API support on Android means that the amount of information about the possibilities of this library is limited. Such feedback would be particularly important for mobile developers. However, there are some information available, describing several possible applications (Wang et al., 2013; Ross et al., 2014; Wang et al., 2016; Acosta et al., 2018).

#### 2.1 Tensorflow

Tensorflow is a library used for ML (Machine Learning) and related AI (Artificial Intelligence) applications. This program is often used for DNN (Deep Neural Network) training. The process of training neural networks requires multiplication of many matrices. The very use of a learned model is also associated with the matrix multiplication process. Tensorflow is a software that can use various APIs used for calculations, such as: CUDA, SYCL or OpenCL. In case of Android, the main API used by Tensorflow is OpenCL. According to (Juhyun and Raman, 2020), the OpenCL API can improve performance by 100% compared to OpenGL. When optimizing the program using OpenCL, the tools provided by the GPU manufacturer Adreno proved to be helpful. Since the OpenCL API is not officially supported, the Tensorflow application polls the device for library availability at startup. When this library is not available, the engine running on the basis of the OpenGL API is loaded.

#### 2.2 OpenCV

OpenCV (Open Source Computer Vision Library) is a library that implements many functionalities such as machine learning and image processing. Many of the algorithms provided by this library are performed on GPUs. The API used by OpenCV to run algorithms on the GPU is OpenCL. From (McIntosh-Smith, 2020) we learn that using OpenCVT API, which utilizes OpenCL internally, improves the frame rate when modifying the camera image. The results achieved by the Sony Xperia Z3, when image modification was done with C/C++ code, was equal

to 3-5 FPS. However, when using OpenCVT, the frame rate increased to 1113.

#### 2.3 Adobe

Adobe is a software developer, focused on image and video processing. In case of Adobe products for Windows-operating devices, the OpenCL API is often used for GPU processing. In the case of applications developed for Android, the utilized API is Vulkan. However, according to (McIntosh-Smith, 2020), Adobe products on Android use OpenCL C kernels. These kernels are compiled into intermediate code called SPIR-V (Standard, Portable Intermediate Representation – V) and then executed using the Vulkan API.

#### 2.4 Benchmarking Applications

Most applications that clearly declare using OpenCL API are tests that measure performance of devices and their integrated hardware components. These programs are most often aimed at measuring computing power or obtaining basic information about a device, such as the number of execution units. Such data only inform about the specificity of the device under test.

## 3 LINKING OPENCL WITH THE ANDROID PLATFORM

Applications usually do not link directly to a driver having a complete API implementation. For this, an additional library is used that looks for driver implementations for all platforms on the device. Thanks to the use of such a loading library, the application can use any available platform supporting this API and is not rigidly connected to one driver in a specific version.

Unfortunately, an Android binary version of such a library does not exist. One option is to link the native library to the driver that comes with the device. The disadvantage of this solution is the need to download the binary file with the OpenCL implementation and all its dependent drivers with respect to a specific device. This solution means that the compiled application will run only on the device for which the libraries were downloaded. Another solution is to download the sources with the code for the loading library, build the library, and link it to the application. The driver that will link the program with the OpenCL implementation has default paths that

can contain the OpenCL library, as shown in Figure 1 (Ashbaugh, 2019).

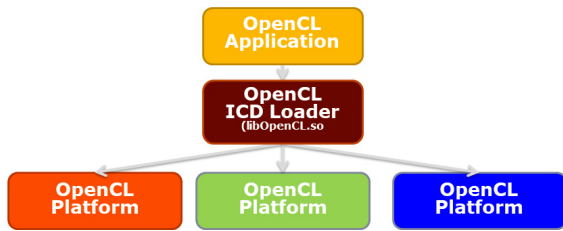


Figure 1: Linking the OpenCL library.

The application can query each device about its properties, and then select the one on which the rest of the application will run on. It is also possible to define an environment variable having the location from which we want the driver to be loaded. The disadvantage of such a solution is the need to compile such a library, but thanks to this, we can build an application that will work on multiple devices.

## 4 ABOUT THE STUDY

Due to the constraints related to energy consumption, mobile devices are rarely used for larger computing tasks. Typically, the work of graphics processors is mainly used to support graphics applications. GPUs operate in a single instruction processing model of several memory elements, so-called SIMDs (Single Instruction, Multiple Data). As a result, one can perform an operation with a single instruction at a time, e.g., on several pixels from an image.

### 4.1 Comparing Mobile GPUs

The two main manufacturers of GPUs for mobile platforms are Qualcomm, which makes the Adreno chipset, and ARM, which produces a chipset called Mali. A direct comparison of pros and cons of Adreno and MAPI GPUs is described in Table 1.

Table 1: Pros and cons of Adreno and Mali GPUs.

Factor	Adreno	Mali
efficiency	+	-
price	-	+
API support	+	-
clock frequency	-	+
rendering	-	+
heat dissipation	+	-

The main pros and cons between those two can be expressed as:

- Performance – Adreno devices achieve better performance results in most cases, e.g., when comparing the Adreno 660 and MaliG78 MP24 models, the former achieves 1486 GFLOPS, while the latter only achieving 1076 GFLOPS.
- Price – Qualcomm imposes higher licensing costs for its chipsets for device manufacturers. As a result, Mali systems are more likely to be found in cheaper devices.
- API support – older Adreno devices have support for more APIs in newer versions. The latest Mali GPU models support a similar API list.
- Clock frequency – in most cases, Mali CPUs run at higher clock rates than competitors, e.g., the Mali G51 processor runs at 1 GHz, whereas none of Adreno’s products achieves this value.
- Rendering – Mali has a bigger advantage over Adreno, e.g., the Adreno 660 is able to render 1524 million triangles per second, whereas the competing Mali G78 MP24 can render 2463 at the same time.
- Heat dissipation – as Mali chipsets run at a higher clock speed, it may cause system overheating. Thanks to the lower temperature, Adreno chipsets may be considered as more efficient.

As shown, users are quite limited, when it comes to selecting a device with, e.g., both high clock frequency and low heat production.

### 4.2 Tested Mobile Devices

The study involved a group of 5 mobile devices, coming from different manufacturers, as described in Table 2. The comparison of available build-in GPUs is described in Table 3, whereas, the comparison of supported APIs, in case of respective chipsets, is shown in Table 4.

The HTC Desire 820 is definitely the device with the weakest parameters. Xiaomi Mi A2 Lite and Huawei P20 Lite have competitive parameters, but they have build-in components from different manufacturers. The most powerful are the Redmi Note 7, which has the fastest memory and the fastest working processor, and the Samsung Galaxy A70, which has the latest graphics processor.

Table 2: Principle technical specification of tested mobile devices.

Device	Release date	CPU	RAM	GPU
Xiaomi Mi A2 Lite	Q3 2018	8-core, 2 GHz, Cortex-A53	3 GB	Adreno 506
HTC Desire 820	Q3 2014	4-core, 1.7 GHz Cortex-A53 + 4-core, 1 GHz Cortex-A53	2 GB	Adreno 405
Huawei P20 Lite	Q2 2018	4-core, 2.36 GHz, Cortex-A53 + 4-core, 1.7 GHz Cortex-A53	4 GB	Mali T830 MP2
Samsung Galaxy A70	Q1 2019	8-core, 2 GHz, Kryo 460	6 GB	Adreno 612
Xiaomi Redmi Note 7	Q1 2019	8-core, 2.2 GHz Kryo 260	6 GB	Adreno 512

Table 3: Tested integrated GPUs.

GPU	Clock freq.	Memory	Memory type	Comput. units
Adreno 506	650 MHz	128 kB + 8 kB	LPDDR3-1866 933 MHz 7.4 GB/s	96
Adreno 405	550 MHz	256 kB	LPDDR3-1333 665.5 MHz 5.3 GB/s	48
Mali T830 MP2	900 MHz	128 kB	LPDDR3 933 MHz	2 x 32
Adreno 612	845 MHz	256 kB + 16 kB	LPDDR4X-3732 1866 MHz Dual channel 16 bit 14.9 GB/s	128
Adreno 512	850 MHz	256 kB + 16 kB	LPDDR4-3732 1866 MHz Quad channel 16 bit 29.8 GB/s	128

Table 4: Chipset compatibility with different APIs.

GPU	Vulcan	D3D	OpenCL	OpenGL
Adreno 506	1.0	DX11	2.0	3.2
Adreno 405	N/A	DX11	1.2	3.2
Mali T830 MP2	1.0	DX11	1.2	3.2
Adreno 612	1.1	DX12	2.0	3.2
Adreno 512	1.0	DX11	2.0	3.2

## 5 PERFORMANCE TESTS

The test procedure involved a number of parameters and factors, that would serve as a good benchmark of OpenCL capabilities on mobile devices.

### 5.1 Computational Power Measurement

It was performed with a test involving several kernels and vector data. For each of these kernels, the number of floating point operations performed should be the same and equal to 4096 for a single work item. For example, in the Float1 kernel, the mad operation will be performed 2048 times. This function consists of a single multiplication and addition. Similar kernels will be used to test other data types such as int, half and double, if these are supported by the device under test.

The obtained results will be presented in the FLOPS (Floating-Point Operations Per Second) unit. In this test, the value in FLOPS will be obtained by multiplying the number of global work items by the number of floating-point operations performed in each of them, and then dividing the obtained value by the time in which they were performed.

### 5.2 Memory Flow

It describes how fast data is copied between different memory areas. A simple kernel was used for testing. In the executing kernel for a single item work, one memory location was copied from the src buffer to the dst buffer. The type of a single buffer element is defined at the compilation stage. In this example, it could be one of the vector versions of the float type.

Two buffers were created in the test. The first had initial data and the second was empty. After executing the kernel, the second buffer will contain data from the first one. The collected time information from the

cl\_event type object will allow to calculate the speed in bytes per second of memory transfer. Similarly, kernels using int, half or double data type memory buffers will also be tested.

### 5.3 Kernel Execution Time

An exemplary kernel is started up/executed, after which the values of CL\_PROFILING\_COMMAND\_QUEUED and CL\_PROFILING\_COMMAND\_START were read and noted. Their difference defines the time it takes to transfer the kernel to the device and start executing it. The kernel will be executed several times and the final result will be averaged.

### 5.4 Memory Transfer with Build-in Functions

Designed to transfer data to and from the memory buffer in OpenCL, we can use the following functions:

- clEnqueueWriteBuffer – after executing this function, the data from the indicated memory will be written to the specified buffer, which can then be used in the executed kernel.
- clEnqueueReadBuffer – copies the data backwards from the buffer to the indicated memory area, so we can read the data after executing the kernels on the device.
- clEnqueueMapBuffer – the function returns a pointer to the memory where the memory was mapped from the buffer.
- clEnqueueUnmapMemObject – this function will map the memory from the pointer returned from clEnqueueMapBuffer or clEnqueueMapImage to the indicated buffer or image.

The above-mentioned functions will be performed a certain number of times, and data on the execution time will be collected from the event object. The averaged result of the function execution will show in what time the device is able to transfer data between the memory on the application side and the memory on the device side.

### 5.5 Matrix Multiplication

The product of matrices is a mathematical operation that can be easily divided into parts that can run in parallel. Each element of the result matrix can be computed independently from one another. A test has been implemented that enumerates each item in the resulting matrix as a separate work item.

In the constructed test, the product of two matrices with a size of 1024x1024 is performed, in which we measure the time of multiplication. The operation will be performed several times, and the final value will be averaged. The experiment will be repeated for several sizes of local work groups. Depending on the properties of the device, the first embodiment will have the maximum possible value of the local work group in the X dimension. In the next embodiment, the value in the X dimension will be reduced twice, while in the Y dimension it will be increased twice. In subsequent iterations, the procedure remains the same until the group's work size in the X dimension becomes 1. This test will illustrate how can the selection of work group's size can affect the kernel execution time.

### 5.6 Using OpenCL to Filter the Image from CameraAPI

It involved using a custom-build application to display an image from the camera on the screen of an Android OS device that passes textures from OpenGL to the camera object as previewTexture. The texture passed to the camera will be updated every frame. Refreshing the image object will cause a method to be called, which will render the obtained image and display it. This code will call the vertex shader and then the fragment shader, so that two triangles will be displayed filled with the image's content.

This will make the image from the camera displayed on the screen of the device. It will help verify how much the OpenCL kernels that are being executed, using the displayed image, will affect the number of displayed frames per second.

Images will be processed in four different ways, that is:

- convert camera image to RGB in the application;
- max Rgb filter;
- grayscale preview;
- average filter.

Next, obtained results will be shown and discussed.

## 6 RESULTS

This chapter describes the results of several tests, carried out on 5 mobile devices.

### 6.1 Computing Power

Figures 2-4 present the performance of the tested devices in terms of the number of operations per second, including different data types. Figure 2 shows the number of possible calculations on half numbers per second, expressed in GHOPS (Giga Half Operations Per Second), for vector versions of this type.

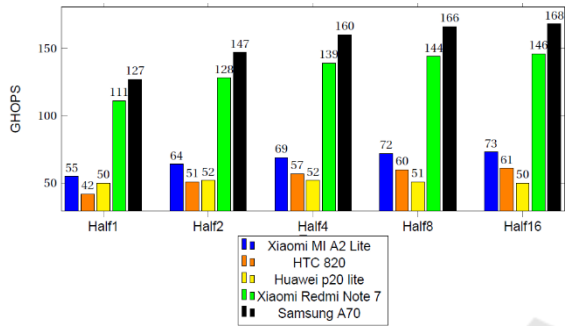


Figure 2: Processing power for type half.

According to obtained results, the Adreno processors use vector types to increase the number of operations per second. For these processors, the results are listed according to their numbers. The HTC 820 with the Adreno 405 is the worst, and the Samsung A70 with the Adreno 612 is the best. In the case of the Mali processor, regardless of the size of the variable type, the number of operations remains at the same level. For comparable models Xiaomi Mi A2 Lite and Huawei P20 Lite, the device fares better than Xiaomi.

Figure 3 presents the number of possible operations on int numbers per second, expressed in GIOPS (Giga Integer Operations Per Second), for vector versions of this type.

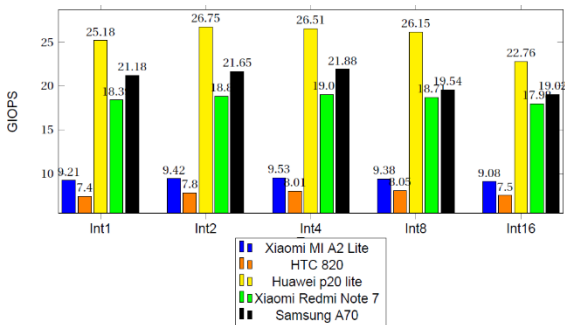


Figure 3: Computing power for the int type.

In case of operations performed on integers, the Mali processor is the best. Using vector types slightly improves performance, which is around

26 GIOPS. In case of Adreno chipsets, similarly as in Mali, the performance is significantly improved by using the int2 and int4 types. However, when using the int8 and int16 types, performance drops. In this case also the order from worst to best goes according to their numbers, preferably Adreno 612, worst Adreno 405.

Figure 4 presents the number of possible operations on float numbers per second, expressed in GFLOPS (Giga Floating Point Operations Per Second), for vector versions of this type.

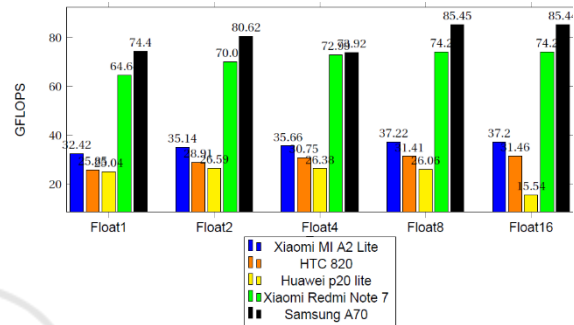


Figure 4: Calculation power for float type.

In performance tests, the Mali T830 2MP chipset proves to be by far the worst for single-precision float operations. The performance of this GPU for vector types is similar to that of a type with a single value. However, it is weaker for the float16 type by 40%. In the case of Adreno chipsets, the situation is similar to the half type, the use of vector types increases performance. For the float16 type, it can be seen that the value of operations per second is approx. half less than for the half16 type.

Figure 5 presents the number of possible calculations of the half type, per second, expressed in GDOPS (Giga Double Operations Per Second) for vector versions of this type.

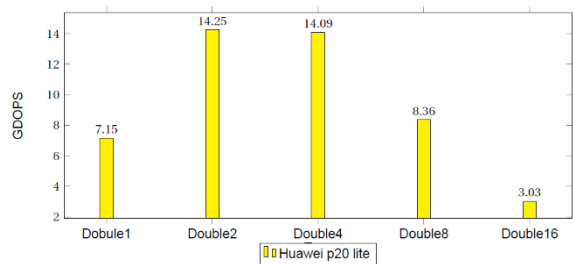


Figure 5: Processing power for the double type.

Huawei P20 Lite, as the only one of the tested devices, has support for double-precision floating-point types. Evidently, the use of vector types double2 and double4 improves performance by 100%

compared to the double type. Operations on double8 type improve performance by 17% compared to double, while operations on double16 type decrease performance by 57% compared to double.

### 6.2 Memory Flow

Figures 6-8 present the memory flow of different data types concerning tested devices.

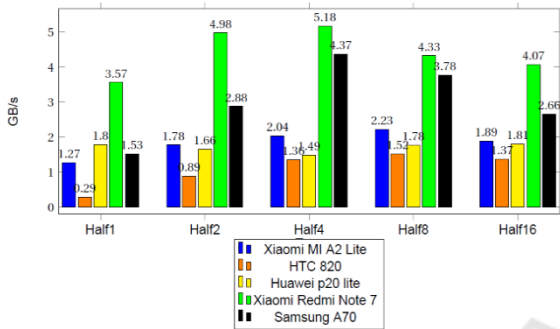


Figure 6: Memory flow for half type.

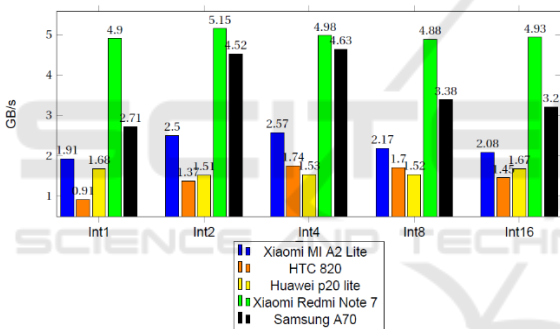


Figure 7: Memory flow for type int.

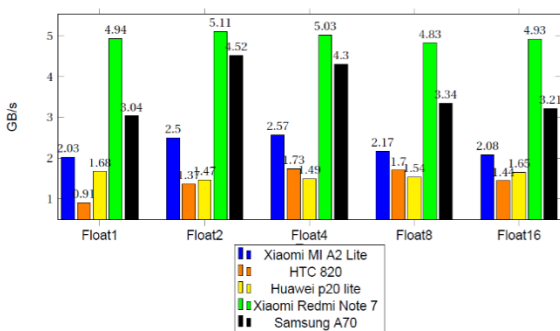


Figure 8: Memory flow for float.

As in the case of testing computing power, when examining the speed of data flow between allocations, it can be seen that for the Mali chipset we do not see much benefit from the use of vector types. For the types hlaf1, half2 and half16, it achieves better results than

the Adreno 405 chipset. In other cases, it proved to be the worst. Among devices with Adreno GPUs, it can be seen that the worst results are obtained for the half type, and the best for the half4 type.

Unlike computing power, the Adreno 512 chipset performs best. Most likely, it achieves a higher result by using a better type of memory. The data flow results for the types float and int reach similar values to those for the vector types half. Copying single values of the half type is worse compared to 32-bit types. The best results are achieved by the Xiaomi Redmi Note 7, which has an Adreno 512 GPU and the best memory type among all tested LPDDR4 devices, with a frequency of 1866 MHz and a bandwidth of up to 29.8 GB/s.

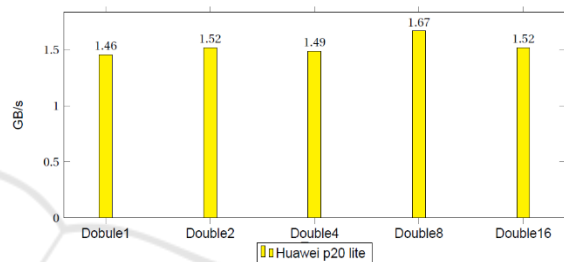


Figure 9: Memory flow for type double.

As shown in Figure 9, the memory transfer speed, when using the double type, does not differ from the types half int or float.

### 6.3 Waiting Time for Execution

Figure 10 present the waiting time for execution, the average time from queuing to starting execution on the GPU. The shorter the time obtained, the better the achieved result.

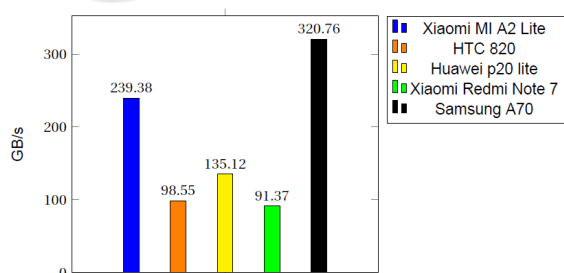


Figure 10: Waiting time for execution.

Surprisingly, the Samsung Galaxy A70 is the worst, which was tested on the latest OpenCL driver among available devices. The second best result was achieved by HTC Desire 820, the oldest device with the oldest driver. The time measured in this test is the period from placing the task in the OpenCL queue,

by the system’s kernel driver, to the device on which it will be executed. So it depends more on the implementation of individual drivers or system kernel than on the device itself.

### 6.4 Application-to-Device Memory Transfer

Figure 11 shows the data transfer rate from the device to the application.

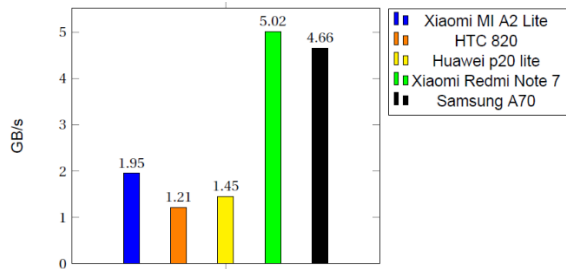


Figure 11: ClEnqueueReadBuffer execution time.

This type of transfers are performed using functions defined by the OpenCL library. clEnqueueReadBuffer copies the exact number of bytes, so the transfer is most likely done with the 8-bit data type. The graph shows the dependencies analogous to the test with data flow inside the device.

Figure 12 presents the data writing speed for allocation in OpenCL. In case of Adreno chipsets, it shows values that are impossible to achieve in the types of memory used in devices.

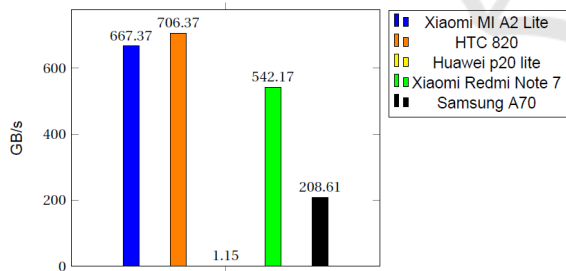


Figure 12: ClEnqueueWriteBuffer execution time.

The assumed maximum memory flow in case of the Samsung Galaxy A70 is 14.9 GB/s. These values were collected using clEvent objects. Evidently the values collected during clEnqueueWriteBuffer, shown in Figure 12, are wrong. To be sure, the test was repeated, measuring the time with system functions on the host side. The obtained results are presented in Figure 13. The values are similar to those obtained in previous tests with the memory flow within the device.

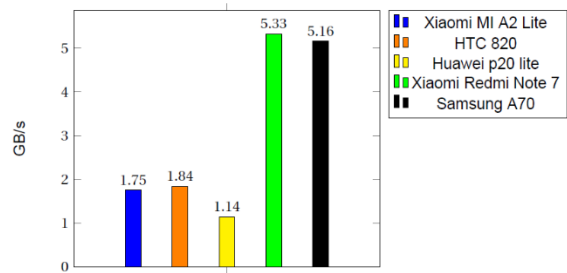


Figure 13: ClEnqueueWriteBuffer execution time Application time.

To be sure, the other tests using time measurement on the application side were repeated. All times coincided with those measured using clEvent objects. Figure 14 shows the time it takes to map device-side memory to application-side memory.

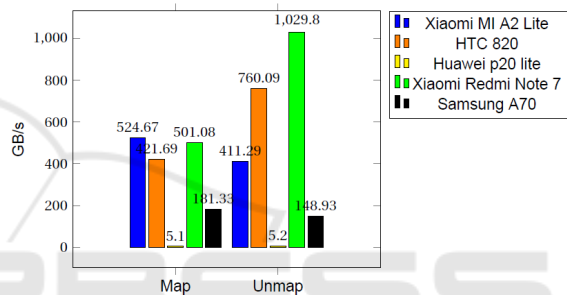


Figure 14: Map unMap execution time.

The values obtained for Adreno processors, similarly to the previous test, seem to be incorrect, but when repeated with the time measurement on the application side, they returned similar results.

Adreno processors use shared system memory. Most likely, the clEnqueueMapBuffer function returns a direct pointer to the memory that is used by the device when executing kernels. The time needed for memory mapping and unmapping is the short time for the driver to return a pointer to the memory used by the OpenCL allocation. In the case of Huawei P20 Lite device with the Mali T830 MP chipset, the memory is not shared. It is clear that the Mali GPU maps memory much slower. Despite the lack of sharing, the mapping process is much faster than reading from the buffer with clEnqueueReadBuffer. Probably the pointer to which the cache is mapped is located in a more advantageous place of the physical memory than the memory allocated to the application by the system.



### 6.5 Matrix Multiplication

Figures 15 and 16 show the dependence of the size of the local work group on the time in which two matrices will be multiplied, in this case both of the size of 1024x1024. It can be seen that the size of the local work group affects the task execution time of matrix multiplication.

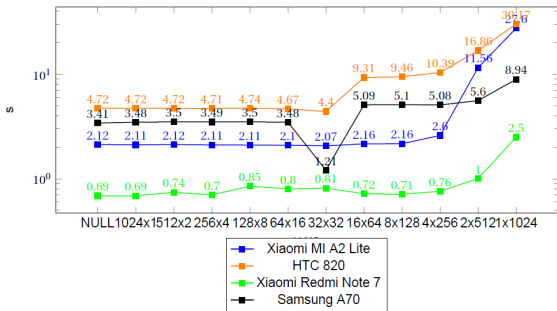


Figure 15: Matrix multiplication (Max Lws 1024).

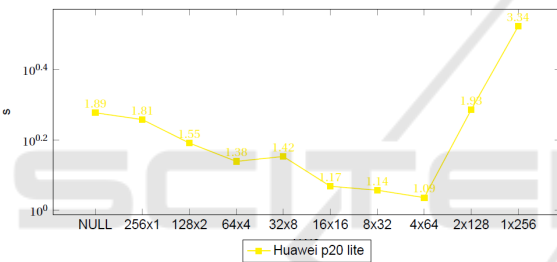


Figure 16: Matrix Multiplication (Max Lws 256).

All devices have their worst time when the local group size on the X dimension is 1, and in the Y dimension it has the maximum possible value for the device. The poor result for this setup is most likely due to reading distant memory cells within the work group. A single row of the matrix takes 4 kB, so the elements from the index (0,0) and (0,1) are separated by 4 kB. In addition, 4 kB is the distance of the virtual address, and physically depending on the size of the memory pages. These items can be located in different parts of the physical memory. The entire matrix takes up 4 MB of memory, and 3 such matrices are used when performing the multiplication. Mobile GPUs do not have such a large dedicated memory, and certainly not that much cache memory, so reloading the memory each time for each element of a local group is very expensive and extends the execution of the kernel.

For devices with Adreno 405, 506 and 612 chipsets, the most optimal local work group size seems to be 32x32. When we multiply matrix A by matrix B, we are multiplying the row of the first

matrix by the column of the second matrix. When accessing the memory of matrix B within a single work group element, we also need to access distant memory elements. Most likely, when the group size of the Y dimension is equal to 32, the memory used by a thread within the local group is available in dedicated GPU memory.

Xiaomi Redmi Note 7 performed the matrix multiplication the fastest, most likely due to the best memory type among the tested phones. The costs of accessing and reading the memory were the lowest. The second fastest device was the Huawei P20 Lite with a GPU from Mali. The test used an integer type data matrix. As previously verified, this device can perform the most operations of this type in a given time.

### 6.6 OpenCL with CameraAPI

Figure 17 shows how the use of OpenCL to process camera data in real-time affects the number of displayed frames per second.

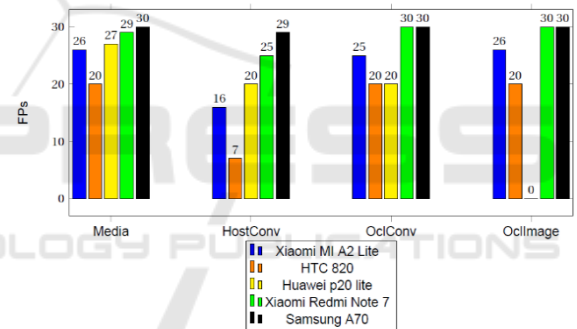


Figure 17: Conversion to RGB.

The first column (Media) in the graph shows the FPS values that devices achieve when displaying texture obtained from CameraAPI using the OpenGL environment. This texture has an image encoded in the NV21 media format.

Unfortunately, the OpenCL API allows sharing resources from OpenGL only in the RGBA (Red Green Blue Alpha) format, so it is impossible to split the texture returned by CameraAPI. A transformation from NV21 to RGBA is needed.

The second column of the graph shows how does the FPS look like when the conversion to RGBA is done on the application side. All devices are negatively affected by such an operation. A clear drop in the displayed frames per second in each case is visible.

The third column describes a situation when the conversion is performed in the OpenCL environment.

This clearly improves performance on devices with Adreno GPUs. These phones reach the values from the first column, including the error of measurement. A Huawei phone with a Mali chipset does not improve the result. Most likely for this device the time needed to copy data for allocation in OpenCL and kernel processing is as expensive as the lack of parallelization of conversion on the CPU side.

The last column shows the results for the case where in the OpenCL kernel we save directly a shared image, which is then displayed. This way we will save time for two copies. First from OpenCL allocation to application side memory and second from application to OpenGL texture. The graph does not show that it had any impact on the number of displayed frames. Most likely, the other optimizations would not bring any performance improvements. The bottleneck that prevents a certain number of frames from being exceeded is either on the CameraAPI side, which is not able to deliver more frames per second, or the OpenGL environment, which cannot display more frames.

Figure 18 shows the effect of filters executed in OpenCL kernels on the number of displayed frames per second. It can be seen that when using simple filters such as max rgb or transforming to grayscale, it does not affect the number of frames displayed, possibly due to a bottleneck that occurs somewhere in the time from collecting the camera preview to displaying it.

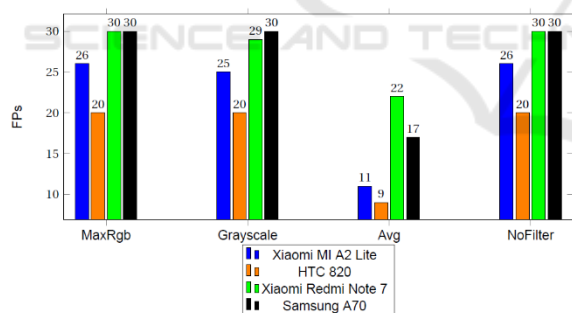


Figure 18: Image filtering results.

On the other hand, the use of an averaging filter, which accesses many pixels in the kernel, has a significant impact on performance. In this case, the phone with the Adreno 512 GPU, which uses the best type of memory among the tested ones, fares the best.

## 7 SUMMARY

The purpose of this work was to analyze the OpenCL environment on mobile platforms. It has been verified

that among the most popular mobile systems, only Android has the ability to support this API. Several tests were designed and performed to check the capabilities of Android devices together with the OpenCL library in terms of performance and cooperation with OpenGL, especially when processing the image from the device’s camera. The tests were performed with a set of 5 smartphones, and included a custom-build software.

The lack of official support for OpenCL by the Android system turned out to be a challenge in the implementation part. Despite this fact, GPU manufacturers for mobile devices continue to develop software that implements the functionality of the OpenCL API. In order to use a library, one must either directly connect the application with a specific dynamic library, or load it while the code is running. In the case of the first solution, the application can be run only on the device from which the library is permanently connected with the program. In the latter case, the library is dynamically loaded. The OpenCL interface on the Android platform is available only from the native code in C++. In order to use it in the application code, it is necessary to execute the OpenCL part of the native library or to utilize an additional driver that will be an overlay for C++ functions.

As part of the tests, it was evaluated how much computing power do individual devices have using the OpenCL API. It was clearly observed that the devices with integrated chipsets manufactured by Adreno responded similarly to the change of data type, and with the use of larger vector types, their computing power increased. However, in case of Mali chipsets, when using vector types, the computing power remained at the same level. Mali GPUs were able to perform more calculations on integer types, while Adreno were better at operations on floating point types. It was verified that the sizes of local work groups significantly affect the time of performing matrix multiplication, which requires reading memory cells that are distant from each other. Devices achieve their best times when the group sizes in the X and Y dimensions are equal. In case of such groups, when performing matrix multiplication, threads within a local group could access elements from the cache memory. In this case, costly cache allocation transfers occur less frequently. In this test, the key role is to access memory quickly, so the better type of memory the devices had, the better they handled this task.

As part of the work, the device’s capabilities for modifying the view from the camera with OpenCL were also checked. It turned out that the view from

the camera is delivered in the NV21 format, which the OpenCL API does not define. Therefore, in order to use the camera image in the executing kernel, it had to be transformed to RGBA format. The conversion significantly affected only the device with the Mali GPU. In case of devices with the Adreno GPU, the process of transforming the image to RGBA format and entering it into the displayed texture did not affect the number of displayed FPS. After applying a set of filters, like grayscale conversion or max rgb filter, the number of images presented per second did not change. On the other hand, the use of an averaging filter, in which it was necessary to read and average the value of 25 pixels to calculate the value of one pixel, significantly reduced the number of displayed frames, for some devices even by more than a half.

At the end, there is limited information on using the OpenCL API in utility applications. Each of the GPU manufacturers for Android devices has an implementation of such a driver, which suggests that it is more often used than just for testing the device's capabilities. The problem with finding information about the use of this API by various applications is most likely due to the fact that mobile developers usually do not publish a list of all used APIs, frameworks or versions of the programming language. Future studies could involve checking how to determine power consumption in case of kernel execution. Another direction would be to evaluate how would OpenCL perform on a variety of devices with integrated Intel, AMD or Nvidia GPUs.

## REFERENCES

- Acosta, A., Merino, C., Totz, J. (2018). Analysis of OpenCL support for mobile GPUs on Android. In *IWOCL'18, Proceedings of the International Workshop on OpenCL*. ACM.
- Ashbaugh, B. (2019). OpenCL on Linux. <https://bashbaug.github.io/openc1/2019/07/06/OpenCL-On-Linux.html> (access: 26.06.2022).
- Aydonat, U., O'Connell, S., Capalija, D., Ling, A. C., Chiu, G. R. (2017). An OpenCL deep learning accelerator on Arria 10. In *FPGA'17, Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM.
- Fang, J., Huang, C., Tang, T., Wang, Z. (2020). Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing*, 2(4), 382-400.
- Fang, J., Varbanescu, A. L., Sips, H. (2011). A comprehensive performance comparison of CUDA and OpenCL. In *ICPP'11, 2011 International Conference on Parallel Processing*. IEEE.
- Gilski, P., Stefański, J. (2015). Android OS: a review. *Tem Journal*, 4(1), 116-120.
- Jääskeläinen, P., de La Lama, C. S., Schnetter, E., Raiskila, K., Takala, J., Berg, H. (2015). pocl: a performance-portable OpenCL implementation. *International Journal of Parallel Programming*, 43(5), 752-785.
- Juhyun, L., Raman S. (2020). Even faster mobile GPU inference with OpenCL. <https://blog.tensorflow.org/2020/08/faster-mobile-gpu-inference-with-openc1.html> (access: 26.06.2022).
- Karimi, K., Dickson, N. G., Hamze, F. (2010). A performance comparison of CUDA and OpenCL. *arXiv preprint*, arXiv:1005.2581.
- Keryell, R., Reyes, R., Howes, L. (2015). Khronos SYCL for OpenCL: A tutorial. In *IWOCL'15, Proceedings of the 3rd International Workshop on OpenCL*. ACM.
- McIntosh-Smith, S. (2020). Catching up with Khronos: Experts' QA on OpenCL 3.0 and SYCL. <https://www.khronos.org/blog/catching-up-with-khronos-experts-qa-on-openc1-3.0-and-sycl-2020> (access: 26.06.2022).
- Munshi, A. (2009). The OpenCL specification. In *HCS'09, 2009 IEEE Hot Chips 21 Symposium*. IEEE.
- Ross, J. A., Richie, D. A., Park, S. J., Shires, D. R., Pollock, L. L. (2014). A case study of OpenCL on an Android mobile GPU. In *HPEC'14, 2014 IEEE High Performance Extreme Computing Conference*. IEEE.
- Seo, S., Jo, G., Lee, J. (2011). Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *IISWC'11, 2011 IEEE International Symposium on Workload Characterization*. IEEE.
- Tay, R. (2013). OpenCL parallel programming development cookbook. Birmingham: Packt Publishing.
- Wang, K., Nurmi, J., Ahonen, T. (2016). Accelerating computation on an Android phone with OpenCL parallelism and optimizing workload distribution between a phone and a cloud service. In *UIC/ATC/ScalCom/CBDCOM/IoP/SmartWorld'16, 2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress*. IEEE.
- Wang, G., Xiong, Y., Yun, J., Cavallaro, J. R. (2013). Accelerating computer vision algorithms using OpenCL framework on the mobile GPU – A case study. In *ICASSP'13, 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE.