



## STRESZCZENIE

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maece-nas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetur.

**Słowa kluczowe:** słowa kluczowe

**Dziedziny nauki i techniki zgodne z wymogami OECD:** 1.2 Nauki o komputerze i informatyce

## **ABSTRACT**

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maece-nas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetur.

**Keywords:** keywords

**OECD field of science and technology:** 1.2 Computer and information sciences

## SPIS TREŚCI

Wykaz ważniejszych oznaczeń i skrótów .....	7
1. Wprowadzenie .....	8
1.1. Cel pracy.....	8
2. Wstęp do Api OpenCL .....	9
2.1. Linkowanie biblioteki OpenCL na platformie Android .....	9
2.2. Typowy przebieg aplikacji OpenCL.....	10
2.3. Możliwości i ograniczenia sprzętowe.....	13
2.4. Pomiary czasu wykonywania kerneli .....	16
2.5. Współdzielenie obiektów pomiędzy OpenCL i OpenGL .....	16
3. Specyfika testowanych Urządzeń .....	18
3.1. Porównanie Graficznych Procesorów Mobilnych.....	18
3.2. Urządzenia wykorzystane do testowania.....	19
3.2.1. Xiaomi Mi A2 Lite .....	19
3.2.2. Huawei P20 Lite .....	19
3.2.3. HTC Desire 820.....	19
3.2.4. Xiaomi Redmi Note 7 .....	19
3.2.5. Samsung Galaxy A70 .....	20
3.2.6. Porównanie .....	20
4. Testy Wydajnościowe OpenCL na Urządzeniach z systemem Android.....	21
4.1. Pomiar Mocy Obliczeniowej.....	21
4.2. Przepływ pamięci.....	23
4.3. Czas Oczekiwania na wykonanie kernela .....	23
4.4. Transfer pamięci wbudowanymi funkcjami .....	24
4.5. Mnożenie macierzy .....	24
4.6. OpenCL do filtrowania obrazu z cameraApi .....	25
4.6.1. Konwersja obrazu z kamery do RGB w aplikacji.....	26
4.6.2. Filtr Max Rgb .....	28
4.6.3. Podgląd w skali szarości .....	29
4.6.4. Filtr Uśredniający.....	30
5. Wyniki, Analiza i Wnioski .....	32
5.1. Wyniki Mocy Obliczeniowej .....	32
5.2. Wyniki Przepływu Pamięci .....	35
5.3. Czas Oczekiwania na wykonanie .....	37
5.4. Transfery Pamięci Aplikacja-Urządzenie .....	38
5.5. Mnożenie Macierzy .....	40
5.6. OpenCL z Kamera Api .....	41
6. Aplikacje wykorzystujące OpenCl Przypadki Użycia .....	44
Wykaz literatury .....	45

Wykaz rysunków.....	46
Wykaz tabel.....	47
Dodatek A. Przykładowy dodatek.....	48

## **WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW**

**PG** – Politechnika Gdańsk

**WETI** – Wydział Elektroniki, Telekomunikacji i Informatyki



## **1. WPROWADZENIE**

### ***1.1. Cel pracy***



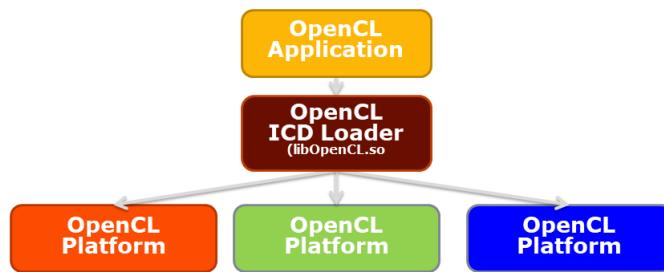
## 2. WSTĘP DO API OPENCL

OpenCL czyli Open Compute Language to standard tworzony obecnie przez grupę Khronos, służący do pisania programów, które mogą zostać wykonane na różnych platformach takich jak CPU, GPU czy FPGA. Specyfikacja OpenCL definiuje interfejs w języku c++, który umożliwia zaprogramowanie aplikacji by ta wykonała konkretny kod na wybranym urządzeniu. Standard OpenCL jest głównie wykorzystywany do równoległych obliczeń takie jak wektorowe operacje matematyczne czy przetwarzanie obrazów. Za implementacje sterownika który wystawia api zgodne z określona wersją specyfikacji odpowiedzialny jest producent urządzenia. Dzięki temu, że standard jest otwarty a jego implementacje posiada większość producentów możemy stworzyć kod który możemy uruchomić niezależnie od architektury czy producenta posiadanego procesora głównego czy graficznego. Jest to duża zaleta w porównaniu na przykład do CUDA, która to jest interfejsem implementowanym jedynie przez Nvidię. Standard OpenCL jest rozwijany i modyfikowany, przez co api zdefiniowane jest w kilku wersjach. Najnowsza wersja specyfikacji to wersja 3.0. Wszystkie wersje są kompatybilne wstępnie. Dodatkowo zdefiniowane są rozszerzenia api takie jak cl\_khr\_gl\_sharing czy definiujący api do sharingu obiektów między OpenCL a OpenGL, takie dodatkowe api jest też specyfikowane przez grupę Khonosa w ramach określonej wersji OpenCL, jednak nie jest obowiązkowe. Istnieją także rozszerzenia api wyspecyfikowane przez konkretnego producenta np cl\_intel\_mem\_force\_host\_memory, które jest dostępna na urządzeniach intel'a, lub cl\_qcom\_android\_native\_buffer\_host\_ptr dostępny na procesorach Qualcom'a z systemem Android. Takie dodatkowe api uzupełnia podstawę, umożliwiając lepsze dopasowanie specyfikacji do konkretnego sprzętu. OpenCL na platformie Android dostępny jest jedynie z poziomu natywnej biblioteki w języku c++.

### 2.1. Linkowanie biblioteki OpenCL na platformie Android

Aplikacje zwykle nie linkują się bezpośrednio ze sterownikiem posiadającym pełną implementację api. Do tego wykorzystywana jest dodatkowa biblioteka która wyszukuje implementacji sterownika dla wszystkich platform na urządzeniu. Dzięki wykorzystaniu takiej ładującej biblioteki Aplikacja może używać każdej dostępnej platformy wspierającej to Api, oraz nie jest na sztywno połączona z jednym sterownikiem w określonej wersji. Niestety nie istnieje binarna wersja takiej biblioteki dla systemu Android. Istnieje możliwość połączenia natywnej biblioteki ze sterownikiem znajdującym się w urządzeniu, jednak wadą takiego rozwiązania jest konieczność pobrania z urządzenia pliku binarnego z implementacją OpenCL oraz wszystkich zależnych od niej sterowników. Takie rozwiązanie powoduje, że skompilowana aplikacja będzie działać jedynie na urządzeniu, z którego zostały pobrane biblioteki. Innym rozwiązaniem jest pobranie źródeł z kodem biblioteki ładującej, zbudowanie biblioteki i połączenie jej z aplikacją. Sterownik który będzie łączył program z implementacją OpenCL ma zapisane domyślne ścieżki w których mogą znajdować się biblioteka OpenCL. Istnieje również możliwość zdefiniowania zmiennej środowiskowej w której wskazujemy lokalizację z której chcemy by sterownik został załadowany. Wadą

takiego rozwiązania jest konieczność komplikacji takiej biblioteki, natomiast dzięki temu możemy zbudować aplikację działającą na wielu urządzeniach. Poniższy obrazek ilustruje w jaki sposób aplikacje łączą się z implementacją sterownika dla konkretnego urządzenia. Aplikacja może odpytać każde urządzenie o jego właściwości, a następnie wybrać te na którym uruchomi się dalsza część aplikacji.



Rys. 2.1. Loader

<https://github.com/krrishnaraj/libopencl-stub>



## 2.2. Typowy przebieg aplikacji OpenCL

Poniżej widać jak wygląda przebieg prostego programu wykorzystującego API OpenCL do inkrementowania każdego elementu bufora pamięci.

```

1  cl_int err = 0;
2  std::unique_ptr<cl_platform_id> platforms;
3  cl_device_id device_id = 0;
4  cl_uint platformsCount = 0;
5  cl_context context = NULL;
6  cl_command_queue queue = NULL;
7  cl_program program = NULL;
8  cl_kernel kernel = NULL;
9  cl_mem buffer = NULL;
10 const size_t bufferSize = sizeof(int) * 1024;
11
12 cl_uint dimension = 1;
13 size_t offset[3] = {0, 0, 0};
14 size_t gws[3] = {bufferSize, 1, 1};
15 size_t lws[3] = {4, 1, 1};
16
17 err = clGetPlatformIDs(0, NULL, &platformsCount);
18 platforms = std::make_unique<cl_platform_id>(platformsCount);
19 err = clGetPlatformIDs(platformsCount, platforms.get(), NULL);
20 cl_device_type deviceType = CL_DEVICE_TYPE_GPU;
21 err = clGetDeviceIDs(platforms.get()[0], deviceType, 1, &device_id,
22                      NULL);
23 context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
24 queue = clCreateCommandQueue(context, device_id, 0, &err);
25
program = clCreateProgramWithSource(context, 1, &kernelStrings, 0,
                                    &err);
  
```

```

26     err = clBuildProgram(program, 1, &device_id, nullptr, nullptr,
27                           nullptr);
28     kernel = clCreateKernel(program, "increment", &err);
29     cl_mem_flags flags = CL_MEM_READ_WRITE;
30     buffer = clCreateBuffer(context, flags, bufferSize, nullptr, &err);
31     void *ptr = clEnqueueMapBuffer(queue, buffer, CL_TRUE, CL_MAP_READ,
32                                    0, bufferSize, 0, nullptr, nullptr, &err);
33     memset(ptr, 13, bufferSize);
34     err = clEnqueueUnmapMemObject(queue, buffer, ptr, 0, nullptr,
35                                   nullptr);
36     err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer);
37     err = clEnqueueNDRangeKernel(queue, kernel, dimension, offset, gws,
38                                  lws, 0, 0, nullptr);
39     err = clFinish(queue);

```

Pierwszym krokiem jest zwołanie **clGetPlatformIDs**. Podając drugi argument czyli `cl_platform_id` jako null, do trzeciego argumentu jakim jest `cl_uint` zostanie wpisana liczba dostępnych na urządzeniu platform wspierających OpenCL (Nvidia/Intel/Arm/Qualcom). Drugie wywołanie `clGetPlatformIDs` wpisze informacje o podanej liczbie platform i zapisze je w tablicy podanej w drugim argumencie. Następnie po wybraniu platformy, której chcemy używać pobieramy obiekt Device wołając `clGetDeviceIDs`, który zwraca listę urządzeń dla danego typu (CPU/GPU/FPGA). W tej pracy w testowanych aplikacjach na system android wykorzystywany będzie `CL_DEVICE_TYPE_GPU`, który jako jedyny jest dostępny na testowanych przeze mnie urządzeniach. Posiadając obiekt device, wołając `clCreateContext` możemy stworzyć context w ramach którego możliwe jest zarządzanie później stworzonymi obiektami na określonych przy tworzeniu contextu urządzeniach. Dalej w przykładowym kodzie stworzona jest kolejka `clCreateCommandQueue` kolejka powstaje w ramach contextu na konkretny device. Później na tym obiekcie kolejkowane będą zadania takie jak transfery pamięci czy wykonywane funkcje. W zależności czy kolejka zostanie stworzona z flagą `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` lub bez niej, zadania te będą mogły być wykonywane równolegle, lub jedno po drugim w kolejności dodania do kolejki. Kolejnym krokiem w powyższym kodzie jest stworzenie obiektu programu używając `clCreateProgramWithSource` lub `clCreateProgramWithBinary` pierwsza stworzy program zawierający nie skompilowany kod w języku OpenCL C, natomiast druga stworzy obiekt z binarnej wersji, wcześniej skompilowanej. Do stworzenia programu ze źródeł przekazywany jest ciąg znaków zawierający kernele, czyli funkcje które mogą zostać wykonane na urządzeniu. Do wykonania inkrementacji każdego elementu bufora w przykładzie użyty zostanie następujący `kernel`

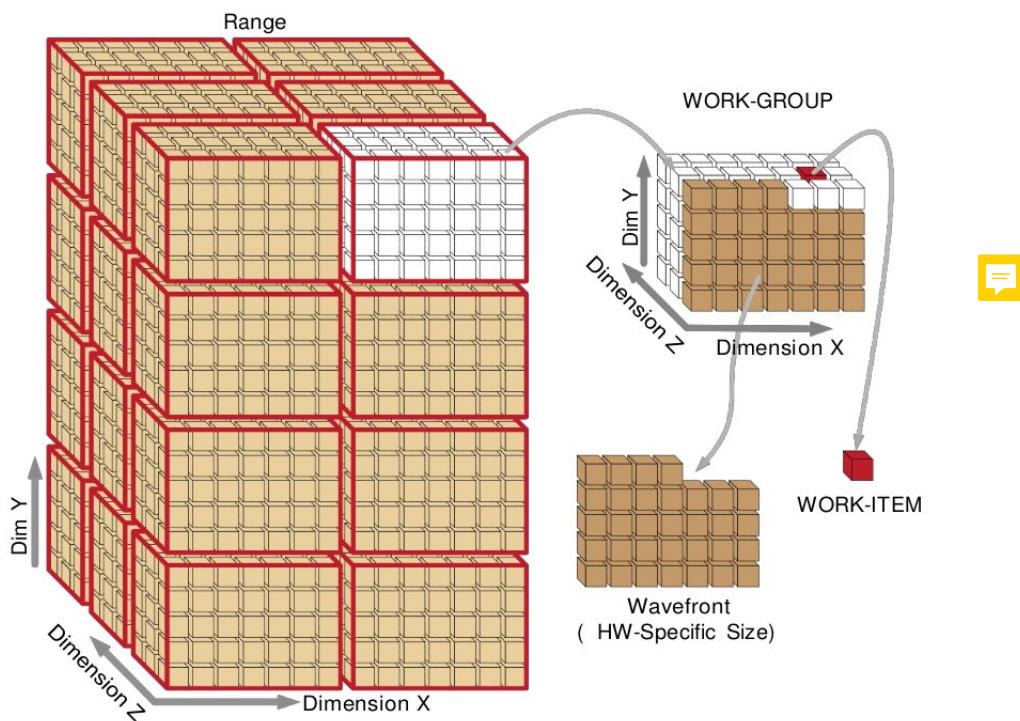
```

1 __kernel void increment(__global int* in){
2     int i = get_global_id(0);
3     in[i]++;
4 }

```

W funkcji tej został pobrany unikalny numer aktualnie wykonywanego kernela w ramach globalnej work grupy, następnie element bufora pod tym indeksem jest inkrementowany. Po stworzeniu programu zawierającego kernele, należy zwołać `clBuildProgram` by kod kerneli w języku OpenCL C został skompilowany dla wskazanego urządzenia. W przypadku stworzenia programu ze źródeł w formie binarnej, tego kroku się nie wykonuje. Następnym wykonanym krokiem jest stworzeniem

obiektu bufora poprzez **clCreateBuffer**. Stworzony został obiekt reprezentujący obszar pamięci o podanym rozmiarze, który może być wykorzystany przy wykonywaniu kernela. W podanym kodzie powstanie bufora o rozmiarze 4096 bajtów czyli 1024 elementów typu int. Następnie zostaje wykonane **clEnqueueMapBuffer** funkcja ta mapuje konkretny bufor na obszar pamięci dostępny z poziomu aplikacji. W tym przypadku w pamięci pod zwróconym wskaźnikiem ustawiamy w każdym bajcie wartość 13. Aby przesłać pamięć z powrotem do obiektu bufora dostępnego z poziomu urządzenia na którym będzie wykonywany kernel wołamy **clEnqueueUnmapMemObject**. Tak przygotowany bufor z danymi możemy ustawić jako argument kernela wołając **clSetKernelArg** podając w argumentach kernel, który w którym chcemy ustawić argument, index argumentu, jego typ oraz wskaźnik na obiekt który będzie argumentem funkcji. Funkcja która uruchomi wykonanie kernela na wskazanym wcześniej urządzeniu jest **clEnqueueNDRangeKernel**, która umieści w kolejce do wykonywania wskazany kernel. W tym momencie podane jest także w ilu wymiarach odbędzie się wykonywanie, podana jest wielkość lokalnej i globalnej work grupy. Rozmiar globalnej work grupy definiuje ilość wątków, które zostaną wykonane. Podział work grup i work itemów pokazuje poniższy obrazek.



Rys. 2.2. Wielowymiarowy ND Range

Na samym końcu zwołane zostaje **clFinish**, jest to funkcja blokująca po wykonaniu której mamy pewność, że wszystkie zakolejkowane na konkretnej kolejce operacje zostały wykonane. W wyniku działania takiego kodu w buforze znajdować się będzie 1024 elementy o wartości 13131314.

### 2.3. Możliwości i ograniczenia sprzętowe

Każde urządzenie posiada ograniczenia związane ze specyfiką implementacji sterownika oraz barkiem zasobów sprzętowych. By dowiedzieć się jakie są maksymalne dostępne wartości np dotyczące rozmiaru pamięci, ilości poszczególnych obiektów w kernelu czy maksymalnej liczby dostępnych work itemów w ramach lokalnej work grupy możemy odpytać sterownik wołając clGetDeviceInfo podając konkretny parametr. Dzięki temu wykonywana aplikacja może dostosować się do ograniczeń sprzętowych. Oto wynik działania aplikacji "clinfo", która wypisuje wszystkie dostępne informacje o urządzeniu. W tym wypadku jest to telefon Xiaomi Mi a2 lite z procesorem graficznym Adreno 506 

```
1 Number of platforms                                1
2 Platform Name                                     QUALCOMM Snapdragon(TM)
3 Platform Vendor                                    QUALCOMM
4 Platform Version                                   OpenCL 2.0 QUALCOMM
5           build: commit #2df12b3 changeid #I07da2d9908 Date: 10/04/18 Thu
6           Local Branch: Remote Branch:
7 Platform Profile                                  FULL_PROFILE
8 Platform Extensions
9
10 Platform Name                                    QUALCOMM Snapdragon(TM)
11 Device Name                                       1
12 Device Vendor                                     QUALCOMM
13 Device Vendor ID                                 0xbf4d3c4b
14 Device Version                                    OpenCL 2.0 Adreno(TM)
15           506
16 Driver Version                                   OpenCL 2.0 QUALCOMM
17           build: commit #2df12b3 changeid #I07da2d9908 Date: 10/04/18 Thu
18           Local Branch: Remote Branch: Compiler E031.36.02.00
19 Device OpenCL C Version                         OpenCL C 2.0 Adreno(TM)
20           506
21 Device Type                                       GPU
22 Device Profile                                     FULL_PROFILE
23 Device Available                                  Yes
24 Compiler Available                               Yes
25 Linker Available                                Yes
26 Max compute units                               1
27 Max clock frequency                            1MHz
28 Device Partition                                (core)
29     Max number of sub-devices                   1
30     Supported partition types                  None
31     Supported affinity domains                (n/a)
32 Max work item dimensions                      3
33 Max work item sizes                           1024x1024x1024
34 Max work group size                          1024
35 Preferred work group size multiple (kernel) 1024
36 Preferred / native vector sizes
37     char                                         1 / 1
38     short                                        1 / 1
39     int                                           1 / 1
40     long                                          1 / 0
41     half                                         1 / 1
42         cl_khr_fp16)
43     float                                         1 / 1
```

38	double	0 / 0	(
39	n/a)		
40	Half-precision Floating-point support	(cl_khr_fp16)	
41	Denormals	No	
42	Infinity and NaNs	Yes	
43	Round to nearest	Yes	
44	Round to zero	No	
45	Round to infinity	Yes	
46	IEEE754-2008 fused multiply-add	No	
47	Support is emulated in software	No	
48	Single-precision Floating-point support	(core)	
49	Denormals	No	
50	Infinity and NaNs	Yes	
51	Round to nearest	Yes	
52	Round to zero	No	
53	Round to infinity	Yes	
54	IEEE754-2008 fused multiply-add	No	
55	Support is emulated in software	No	
56	Correctly-rounded divide and sqrt operations	No	
57	Double-precision Floating-point support	(n/a)	
58	Address bits	64, Little-Endian	
59	Global memory size	1875912704 (1.747GiB)	
60	Error Correction support	No	
61	Max memory allocation	468978176 (447.3MiB)	
62	Unified memory for Host and Device	Yes	
63	Shared Virtual Memory (SVM) capabilities	(core)	
64	Coarse-grained buffer sharing	Yes	
65	Fine-grained buffer sharing	No	
66	Fine-grained system sharing	No	
67	Atomics	No	
68	Minimum alignment for any data type	128 bytes	
69	Alignment of base address	1024 bits (128 bytes)	
70	Page size (QCOM)	4096 bytes	
71	External memory padding (QCOM)	0 bytes	
72	Preferred alignment for atomics		
73	SVM	128 bytes	
74	Global	0 bytes	
75	Local	0 bytes	
76	Max size for global variable	65536 (64KiB)	
77	Preferred total size of global vars	1048576 (1024KiB)	
78	Global Memory cache type	Read/Write	
79	Global Memory cache size	16384 (16KiB)	
80	Global Memory cache line size	64 bytes	
81	Image support	Yes	
82	Max number of samplers per kernel	16	
83	Max size for 1D images from buffer	134217728 pixels	
84	Max 1D or 2D image array size	2048 images	
85	Base address alignment for 2D image buffers	64 bytes	
86	Pitch alignment for 2D image buffers	64 pixels	
87	Max 2D image size	16384x16384 pixels	
88	Max 3D image size	16384x16384x2048	
89	pixels		
90	Max number of read image args	128	
91	Max number of write image args	64	
92	Max number of read/write image args	64	
93	Max number of pipe args	16	
94	Max active pipe reservations	4096	
95	Max pipe packet size	1024	
96	Local memory type	Local	
97	Local memory size	32768 (32KiB)	
98	Max number of constant args	8	
99	Max constant buffer size	65536 (64KiB)	

```

98 Max size of kernel argument 1024
99 Queue properties (on host)
100   Out-of-order execution Yes
101   Profiling Yes
102 Queue properties (on device)
103   Out-of-order execution Yes
104   Profiling Yes
105   Preferred size 655376 (640KiB)
106   Max size 655376 (640KiB)
107 Max queues on device 1
108 Max events on device 1024
109 Prefer user sync for interop No
110 Profiling timer resolution 1000ns
111 Execution capabilities
112   Run OpenCL kernels Yes
113   Run native kernels No
114 printf() buffer size 1048576 (1024KiB)
115 Built-in kernels (n/a)
116 Device Extensions
117   cl_khr_3d_image_writes cl_img_egl_image
118   cl_khr_byte_addressable_store cl_khr_depth_images
119   cl_khr_egl_event cl_khr_egl_image cl_khr_fp16 cl_khr_gl_sharing
120   cl_khr_global_int32_base_atomics
121   cl_khr_global_int32_extended_atomics
122   cl_khr_local_int32_base_atomics
123   cl_khr_local_int32_extended_atomics cl_khr_image2d_from_buffer
124   cl_khr_mipmap_image cl_khr_srgb_image_writes cl_khr_subgroups
125   cl_qcom_create_buffer_from_image cl_qcom_ext_host_ptr
126   cl_qcom_ion_host_ptr cl_qcom_perf_hint cl_qcom_read_image_2x2
127   cl_qcom_android_native_buffer_host_ptr cl_qcom_protected_context
128   cl_qcom_priority_hint cl_qcom_compressed_yuv_image_read
129   cl_qcom_compressed_image
130
131 NULL platform behavior
132   clGetPlatformInfo(NULL, CL_PLATFORM_NAME, ...) No platform
133   clGetDeviceIDs(NULL, CL_DEVICE_TYPE_ALL, ...) No platform
134   clCreateContext(NULL, ...) [default] No platform
135   clCreateContext(NULL, ...) [other] Success [PO]
136   clCreateContextFromType(NULL, CL_DEVICE_TYPE_DEFAULT) Success (1)
137     Platform Name Qualcomm Snapdragon(TM)
138     Device Name Qualcomm Adreno(TM)
139   clCreateContextFromType(NULL, CL_DEVICE_TYPE_CPU) No devices found
140     in platform
141   clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU) Success (1)
142     Platform Name Qualcomm Snapdragon(TM)
143     Device Name Qualcomm Adreno(TM)
144   clCreateContextFromType(NULL, CL_DEVICE_TYPE_ACCELERATOR) No devices
145     found in platform
146   clCreateContextFromType(NULL, CL_DEVICE_TYPE_CUSTOM) Invalid device
147     type for platform
148   clCreateContextFromType(NULL, CL_DEVICE_TYPE_ALL) Success (1)
149     Platform Name Qualcomm Snapdragon(TM)
150     Device Name Qualcomm Adreno(TM)

```

## 2.4. Pomiary czasu wykonywania kerneli

Standard OpenCL zapewnia mechanizm do odczytywania stempli czasu z poszczególnych etapów wykonywania kernela. Służy do tego obiekt typu `cl_event` stworzony przez funkcję `clCreateUserEvent`. Obiekt taki może zostać przekazany jako argument funkcji, która zostanie wykonana na urządzeniu np `clEnqueueNDRangeKernel` czy `clEnqueueWriteBuffer`. Po wykonaniu kernela z obiektu eventa można odczytać stemple czasu z jego wykonania. By wydobyć wartości należy zwołać `clGetEventProfilingInfo` przekazując jako argument obiekt eventu oraz jeden z czterech parametrów

- `CL_PROFILING_COMMAND_QUEUED` wartość opisuje czas urządzenia w którym komenda została dodana do kolejki.
- `CL_PROFILING_COMMAND_SUBMIT` wartość opisuje czas urządzenia w którym komenda została wysłana do urządzenia na którym zostanie wykonana.
- `CL_PROFILING_COMMAND_START` wartość opisuje czas urządzenia w którym rozpoczęte zostało wykonywanie komendy na urządzeniu.
- `CL_PROFILING_COMMAND_END` wartość opisuje czas urządzenia w którym wykonywanie komendy zostaje zakończone.

## 2.5. Współdzielenie obiektów pomiędzy OpenCL i OpenGL

OpenGL czyli Open Graphics Library to standard, który tak jak OpenCL jest zdefiniowany i utrzymywany przez grupę Khronos. Jest on wykorzystywany głównie do renderowania obiektów graficznych. Istnieje możliwość stworzenia tekstury w OpenGL i użycie tego w kernelu OpenCL. Nie wszystkie urządzenia wspierają współdzielenie zasobów, urządzenie wspiera współdzielenie jeśli w liście rozszerzeń znajduje się `cl_khr_gl_sharing`. By móc skorzystać z tej możliwości, context w OpenCL musi zostać stworzony w oparciu o wcześniej stworzony context OpenGL. Przy tworzeniu contextu podajemy strukturę, w której umieszczone są wskaźniki na kontekst w OpenGL i uchwyt na EGLDisplay.

```
1     cl_context_properties props[] =  
2         { CL_GL_CONTEXT_KHR, (cl_context_properties) ctx,  
3           CL_EGL_DISPLAY_KHR, (cl_context_properties) dis,  
4           CL_CONTEXT_PLATFORM, 0,  
5           0  
6     };
```

. Posiadając tak stworzony kontekst, możemy stworzyć obiekt obrazka w OpenCL w oparciu o stworzoną teksturę w OpenGL 

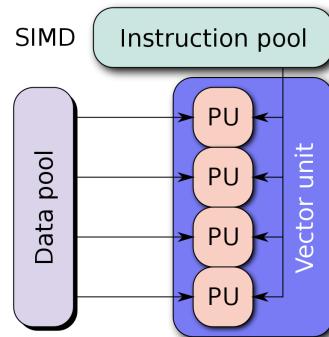
```
1     imageObj = clCreateFromGLTexture(context,  
2                                         CL_MEM_WRITE_ONLY,  
3                                         GL_TEXTURE_2D,  
4                                         0,  
5                                         texture_id,  
6                                         &status);
```

. gdzie `textureId` to identyfikator tekstury, którą chcemy współdzielić z OpenGL. Przed wykorzystaniem współdzielonego obrazka należy zwołać `clEnqueueAcquireGLObjets` wskazując kolejkę,

która będzie używać obrazka. Jest to wykonywane w celu wywłaszczenia obiektu przez OpenCL. Po zakończeniu operacji na obiekcie należy go zwolnić polecienniem `clEnqueueReleaseGLObj`ects.

### 3. SPECYFIKA TESTOWANYCH URZĄDZEŃ

Ze względu na ograniczenia związane ze zużyciem energii, urządzenia mobilne są nie często wykorzystywane do większych zadań obliczeniowych. Zwykle praca procesorów graficznych wykorzystywana jest głównie do wspierania aplikacji graficznych. Procesory Graficzne działają w modelu przetwarzania pojedynczą instrukcją kilku elementów pamięci (SIMD). Tak by móc jedną instrukcją w tym samym czasie wykonać operacje na kilku pixelach z obrazka.



Rys. 3.1. SIMD

#### 3.1. Porównanie Graficznych Procesorów Mobilnych

Dwoma głównymi producentami procesorów graficznych na mobilne platformy są Qualcomm produkujący procesory Adreno i ARM tworzący GPU o nazwie Mali. Oto podstawowe wady i zalety wymienionych produktów.

- Adreno zalety:
  - Lepszy Performance
  - Wspiera nowsze API
  - Mniej się przegrzewa
- Adreno wady:
  - Droższy
  - Mniej zoptymalizowany
  - Słabszy przy renderowaniu.
- Mali zalety:
  - Bardziej rozpowszechnione, więc lepiej zoptymalizowane.
  - Niższa cena
  - Większe częstotliwości zegara
  - Szybsze renderowanie.
- Mali wady:
  - Gorszy performance
  - Gorszy support API
  - Bardziej się grzeje.



### **3.2. Urządzenia wykorzystane do testowania**

#### **3.2.1. Xiaomi Mi A2 Lite**



Jest to telefon z lipca 2018 roku. Posiadający ośmiordzeniowy procesor Cortex A53, z częstotliwością zegara 2.0GHz. Pamięć systemowa urządzenia to 3GB. Umieszczono w nim Procesor graficzny Adreno 506. Procesor ten posiada 96 jednostek mogących jednocześnie wykonywać wątki na urządzeniu. Częstotliwość zegara procesora wynosi 650MHz. Dodatkowo posiada on 128 KB pamięci wbudowanej w chip GPU oraz 8KB, jest to pamięć typu LPDDR3 933MHz i przepustowości 7.4GB/s pamięć cache. Wspiera następujące API: Vulkan 1.0, OpenGL 3.2, DX11, OpenCL 2.0. Telefon posiada system operacyjny Android w wersji 10. Sterownik OpenCL na urządzeniu jest w wersji #7331a27 z 11/13/19, posiada wsparcie dla typu zmiennoprzecinkowe Half, oraz dla współdzielenia zasobów z OpenGL. Nie posiada wsparcia dla typów zmiennoprzecinkowych podwójnej precyzji Double.

#### **3.2.2. Huawei P20 Lite**

Jest to Urządzenie z marca 2018 roku. Wyposażone jest w 8 rdzeniowy procesor 4x2.36 GHz Cortex-A53 + 4x1.7 GHz Cortex-A53. Posiada on 4GB pamięci systemowej. Wbudowany procesor graficzny to Mali-T830 MP2, procesor posiada dwie jednostki wykonawcze. Wg danych producenta pojedyncza jednostka w tym procesorze może maksymalnie wykonywać 256 wątków. Częstotliwość zegara wynosi 650MHz, posiada 16KB pamięci cache, jest to pamięć LPDDR3 o częstotliwości 933MHz i przepustowości do 14.9 GB/s. Wspiera następujące API: Vulkan 1.0, OpenCL 1.2, OpenGL 3.2. Wspiera typ Half oraz Double, nie wspiera współdzielenia zasobów z OpenGL. Telefon działa z systemem operacyjnym Android w wersji 8



#### **3.2.3. HTC Desire 820**

Telefon z 2014 roku, wyposażony w procesor ośmiordzeniowy 1,7 GHz quad-core Cortex-A53 + 1 GHz quad-core Cortex-A53. Posiada 2GB pamięci RAM. Posiada procesor graficzny Adreno 405. Gpu może wykonywać na raz 48 wątków, dodatkowo posiada 256KB pamięci dedykowanej. Wykorzystano pamięć typu LPDDR3 z częstotliwością 666.5MHz i szybkości 5.3GB/s. Wspierane przez urządzenia API to: OpenGL 3.2, OpenCL 1.2, DX11. Telefon Dla Systemu Android w wersji 5 posiada sterownik OpenCL w wersji z 29.04.15, a z androidem 6 w wersji z 23.03.2016. Wspiera typ Half, nie wspiera Double, możliwe do wykorzystania rozszerzenie współdzielenia zasobów między OpenCL i OpenGL.



#### **3.2.4. Xiaomi Redmi Note 7**

Telefon wydany na początku 2019 roku. Posiada procesor ośmiordzeniowy Qualcomm Kryo 260 z częstotliwością do 2,2 GHz. Wyposażony w 6GB pamięci systemowej. Dodatkowo posiada Procesor graficzny Adreno 512. GPU działa z częstotliwością 850MHz posiada 256KB pamięci dedykowanej i 16KB cache, pamięć typu LPDDR4 o częstotliwości 1866MHz i przepu-

stowości do 29.8GB/s. Może wykonywać na raz 128 wątków. Telefon działa z systemem android w wersji 10, posiada sterownik OpenCL w wersji:#9b15012 z 09/17/20.Wspiera Api Vulkan 1.0, OpenGL 3.2, DX11, OpenCL 2.0. OpenCL nie wspiera typu double, ale wspiera typ Half oraz współdzielenie zasobów z OpenGL.

### 3.2.5. *Samsung Galaxy A70*

To urządzenie, którego premiera odbyła się na początku 2019 roku. wyposażony w procesor ośmiordzeniowy Kryo 460 z częstotliwością zegara do 2 GHz. Posiada 6GB pamięci RAM. Dodatkowo posiada procesor graficzny Adreno 612 o częstotliwości 845Mhz, 128 wątkach, wbudowanej pamięci 256KB i 16KB pamięci cache, jest to pamięć typu LPDDR4X o częstotliwości 1866MHz i przepustowości do 14.9GB/s. Android w wersji 10 ze sterownikiem OpenCL w wersji #e1ac91e z 11/17/20. Wspiera typ Half, nie wspiera Double, możliwe do wykorzystania rozszerzenie współdzielenia zasobów między OpenCL i OpenGL.

### 3.2.6. *Porównanie*

Zdecydowanie urządzeniem z najsłabszymi parametrami jest HTC Desire 820. Xiaomi Mi A2 Lite oraz Huawei P20 lite posiadają konkurencyjne parametry, jednak są urządzeniami posiadającymi podzespoły od różnych producentów. Najmocniejszymi są Redmi Note 7, który ma najszybszą pamięć oraz najszybciej pracujący procesor, oraz SAmg Galaxy A70, który ma najnowszy procesor graficzny.



## 4. TESTY WYDAJNOŚCIOWE OPENCL NA URZĄDZENIACH Z SYSTEMEM ANDROID

### 4.1. Pomiar Mocy Obliczeniowej

Pomiary mocy obliczeniowej zostaną przeprowadzone za pomocą następującego testu. Wykonany zostanie jeden z poniższych kerneli. Wykorzystane zostaną wektorowe typy danych. Dla każdego z tych kerneli liczba wykonanych operacji zmiennoprzecinkowych powinna być taka sama i wynosić 4096 dla pojedynczego work itemu. W przykładowo kernelu flops\_float1 operacja mad zostanie wykonana 2048 razy funkcja ta składa się z pojedynczego mnożenia i dodawania.

```
1 #define MAD_4(x, y)      x = mad(y, x, y);      y = mad(x, y, x);      x = mad
2           (y, x, y);      y = mad(x, y, x);
3 #define MAD_16(x, y)     MAD_4(x, y);          MAD_4(x, y);          MAD_4(x
4           , y);          MAD_4(x, y);
5 #define MAD_64(x, y)     MAD_16(x, y);         MAD_16(x, y);         MAD_16(
6           x, y);          MAD_16(x, y);
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
```

```
_kernel void flops_float1(__global float *ptr, float _A)
{
    float x = _A;
    float y = (float)get_local_id(0);
    int gid = get_global_id(0);

    for(int i=0; i<128; i++)
    {
        MAD_16(x, y);
    }

    ptr[gid] = y;
}

_kernel void flops_float2(__global float *ptr, float _A)
{
    float2 x = (float2)(_A, (_A+1));
    float2 y = (float2)get_local_id(0);
    int gid = get_global_id(0);

    for(int i=0; i<64; i++)
    {
        MAD_16(x, y);
    }

    ptr[gid] = (y.S0) + (y.S1);
}

_kernel void flops_float4(__global float *ptr, float _A)
{
    float4 x = (float4)(_A, (_A+1), (_A+2), (_A+3));
    float4 y = (float4)get_local_id(0);

    for(int i=0; i<32; i++)
    {
        MAD_16(x, y);
    }
}
```

```

45     ptr[get_global_id(0)] = (y.S0) + (y.S1) + (y.S2) + (y.S3);
46 }
47
48
49 __kernel void flops_float8(__global float *ptr, float _A)
50 {
51     float8 x = (float8)(_A, (_A+1), (_A+2), (_A+3), (_A+4), (_A+5), (_A
52         +6), (_A+7));
53     float8 y = (float8)get_local_id(0);
54
55     for(int i=0; i<16; i++)
56     {
57         MAD_16(x, y);
58     }
59
60     ptr[get_global_id(0)] = (y.S0) + (y.S1) + (y.S2) + (y.S3) + (y.S4)
61         + (y.S5) + (y.S6) + (y.S7);
62 }
63
64 __kernel void flops_float16(__global float *ptr, float _A)
65 {
66     float16 x = (float16)(_A, (_A+1), (_A+2), (_A+3), (_A+4), (_A+5), (
67         _A+6), (_A+7),
68             (_A+8), (_A+9), (_A+10), (_A+11), (_A+12), (_A+13),
69                 (_A+14), (_A+15));
70     float16 y = (float16)get_local_id(0);
71
72     for(int i=0; i<8; i++)
73     {
74         MAD_16(x, y);
75     }
76
77     float2 t = (y.S01) + (y.S23) + (y.S45) + (y.S67) + (y.S89) + (y.SAB
78         ) + (y.SCD) + (y.SEF);
79     ptr[get_global_id(0)] = t.S0 + t.S1;
80 }

```



Argument kernela `_A` to przykładowa, zmiennoprzecinkowa wartość początkowa. W przeprowadzonych testach rozmiar lokalnej work grupy to maksymalny możliwy rozmiar dla kernela. Natomiast rozmiar globalnej work grupy to największy możliwy rozmiar lokalnej grupy przemnożony przez liczbę dostępnych jednostek wykonawczych razy 2048.

Uzyskany wynik przedstawiony zostaje w jednostce FLOPS jest to jednostka określająca liczbę wykonanych operacji zmiennoprzecinkowych na sekundę. W tym teście wartość w FLOPS otrzymamy przez pomnożenie liczby globalnych work itemów przez liczbę wykonywanych zmiennoprzecinkowych operacji w każdym z nich, a następnie podzielenie uzyskanej wartości przez czas w jakim te się wykonywały. Do zmierzenia czasu wykorzystano obiekt typu `cl_event`. Po wykonaniu kernela zostały odczytane wartości `CL_PROFILING_COMMAND_START` i `CL_PROFILING_COMMAND_END`. Różnica tych wartości to czas wykonywania funkcji na urządzeniu.



Analogiczne kernele zostaną wykorzystane do przetestowania innych typów danych takich jak `integer`, `half` i `double`, jeśli te są wspierane przez testowane urządzenie.

## 4.2. Przepływ pamięci

Zbadane zostało jak szybko dane zostają kopiowane pomiędzy różnymi obszarami pamięci. Do przetestowania został użyty prosty kernel.

```
1 #ifdef FLOAT1
2     typedef float Type;
3 #endif
4 #ifdef FLOAT2
5     typedef float2 Type;
6 #endif
7
8 #ifdef FLOAT4
9     typedef float4 Type;
10 #endif
11
12 #ifdef FLOAT8
13     typedef float8 Type;
14 #endif
15
16 #ifdef FLOAT16
17     typedef float16 Type;
18 #endif
19
20 __kernel void readFloatType(__global Type *dst, __global Type *src){
21     uint gid = get_global_id(0);
22     dst[gid] = src[gid];
23 }
```

W wykonywanym kernelu dla pojedynczego work itemu kopowana jest jedna komórka pamięci z bufora src do dst. Typ pojedynczego elementu bufora jest definiowany na etapie komilacji. W tym przykładzie może być to jedna z wektorowych wersji typu float.

W teście stworzone zostają dwa bufora pierwsi posiada inicjalne dane a drugi jest pusty. Po wykonaniu kernela w drugim buforze znajdują się dane z pierwszego. Zebrane informacje o czasie z obiektu typu cl\_event pozwalają nam obliczyć z jaką prędkością w bajtach na sekundę dochodzi o transferu pamięci. Analogicznie kernele używające buforów pamięci o typie danych integer half czy double zostaną także rzetestowane.

## 4.3. Czas Oczekiwania na wykonanie kernela

W celu sprawdzenia czasu oczekiwania na rozpoczęcie wykonywania kernela, wykonany został następujący test. Wykonany jest dowolny kernel w testowanym scenariuszu następujący.

```
1 __kernel void increment(__global int* in){
2     int i= get_global_id(0);
3     in[i]++;
4 }
```

Po wykonaniu kernela odczytane zostały wartości CL\_PROFILING\_COMMAND\_QUEUED i CL\_PROFILING\_COMMAND\_START. Różnica tych dwóch to czas potrzebny przesłanie kernela do urządzenia i rozpoczęcie jego wykonania. Kernel taki zostaje wykonany określoną ilość razy a wynik końcowy zostaje uśredniony.

#### 4.4. Transfer pamięci wbudowanymi funkcjami

Do przetransferowania danych do bufora i z bufora pamięci w OpenCL możemy następujące funkcje:

- **clEnqueueReadBuffer** Po wykonaniu tej funkcji dane z pod wskazanej pamięci zostaną zapisane w podanym buforze, który potem może być wykorzystany w wykonywanym kernelu.
- **clEnqueueReadBuffer** Kopiuje dane w drugą stronę. Z bufora do wskazanego obszaru pamięci, dzięki temu możemy odczytać dane po wykonaniu kerneli na urządzeniu.
- **clEnqueueMapBuffer** Funkcja zwraca wskaźnik na pamięć pod którą została zmapowana pamięć z bufora
- **clEnqueueUnmapMemObject** Funkcja ta zmapuje pamięć spod wskaźnika zwróconego z clEnqueueMapBuffer lub clEnqueueMapImage do wskazanego bufora lub obrazka.

Wyżej wskazane funkcje wykonane zostaną określoną ilość razy, a z obiektu event zostaną zebrane dane dotyczące czasu wykonania. Uśredniony wynik z wykonania funkcji pokaże w jakim czasie urządzenie jest w stanie transferować dane pomiędzy kodem pamięcią po stronie aplikacji a pamięcią po stronie urządzenia.

#### 4.5. Mnożenie Macierzy

Iloczyn macierzy to działanie matematyczne, które można w łatwy sposób podzielić na części, które mogą wykonywać się równolegle. Wyliczenie każdego elementu macierzy wynikowej może odbywać się niezależnie od innych. Zaimplementowano test, który wylicza każdy element w wynikowej macierzy jako osobny work item. Iloczyn macierzy przy użyciu OpenCL może być wykonany za pomocą następującego kernela

```
1 __kernel void matrixMul(__global int *A, __global int *B, __global int
2     *dst, int N, int M){
3     uint gidX = get_global_id(0);
4     uint gidY = get_global_id(1);
5     int tmp = 0;
6     for(int k = 0; k < N; k++) {
7         tmp += A[(gidY * N) + k] * B[(k * M) + gidX];
8     }
9     dst[gidY*M+gidX] = tmp;
}
```

W zbudowanym teście wykonywany jest iloczyn dwóch macierzy o rozmiarach 1024x1024. Zmierzony zostaje czas w jakim udało się je przez siebie przemożyć. Operacja wykonana jest określoną liczbę razy a wartość końcowa jest średnią z tych wykonań. Eksperyment powtórzony jest dla kilku rozmiarów lokalnych work grup. W zależności od właściwości urządzenia pierwsze wykonanie będzie miało maksymalną możliwą wartość lokalnej work grupy w wymiarze X. W kolejnym wykonaniu wartość w wymiarze X zmniejszona będzie dwukrotnie zmniejszona, natomiast w wymiarze Y będzie ona dwukrotnie zwiększona. W kolejnych iteracjach procedura wygląda tak samo, aż rozmiar work grupy w wymiarze X osiągnie wartość 1. Test ten ilustruje, jak dobór rozmiaru work grupy może mieć wpływ na czas wykonania kernela.

#### 4.6. OpenCL do filtrowania obrazu z cameraApi

Do wyświetlania obrazu z kamery na ekranie urządzenia z androidem, wykorzystała zostaną prosta aplikacja, która przekazuje tekstury z openGL do obiektu kamery jako previewTexture tekstura tworzona w następujący sposób.

```
1 static private int createTexture()
2 {
3     int[] texture = new int[1];
4
5     GLES20.glGenTextures(1, texture, 0);
6     GLES20 glBindTexture(GLES11Ext.GL_TEXTURE_EXTERNAL_OES, texture[0]);
7     GLES20.glTexParameterf(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
8         GL10.GL_TEXTURE_MIN_FILTER, GL10.GL_LINEAR);
9     GLES20.glTexParameterf(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
10        GL10.GL_TEXTURE_MAG_FILTER, GL10.GL_LINEAR);
11    GLES20.glTexParameteri(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
12        GL10.GL_TEXTURE_WRAP_S, GL10.GL_CLAMP_TO_EDGE);
13    GLES20.glTexParameteri(GLES11Ext.GL_TEXTURE_EXTERNAL_OES,
14        GL10.GL_TEXTURE_WRAP_T, GL10.GL_CLAMP_TO_EDGE);
15
16    return texture[0];
17 }
```

Tekstura przekazana do kamery jest aktualizowana co każdą klatkę. Odświeżenie obiektu obrazka powoduje wywołanie metody która wyrenderuje uzyskany obrazek i go wyświetli. Odpowiedzialny za to jest ten fragment kodu.

```
1 public void draw() {
2     GLES20 glUseProgram(mProgram);
3
4     GLES20.glActiveTexture(GLES20.GL_TEXTURE0);
5     GLES20 glBindTexture(GLES11Ext.GL_TEXTURE_EXTERNAL_OES, texture);
6
7     mPositionHandle = GLES20.glGetAttribLocation(mProgram, "position");
8     GLES20 glEnableVertexAttribArray(mPositionHandle);
9     GLES20 glVertexAttribPointer(mPositionHandle, COORDS_PER_VERTEX,
10         GLES20.GL_FLOAT, false, vertexStride, vertexBuffer);
11
12     mTextureCoordHandle = GLES20.glGetAttribLocation(mProgram, "inputTextureCoordinate");
13     GLES20 glEnableVertexAttribArray(mTextureCoordHandle);
14     GLES20 glVertexAttribPointer(mTextureCoordHandle,
15         COORDS_PER_VERTEX, GLES20.GL_FLOAT, false, vertexStride,
16         textureVerticesBuffer);
17
18     mColorHandle = GLES20.glGetAttribLocation(mProgram, "s_texture");
19
20     GLES20.glDrawElements(GLES20.GL_TRIANGLES, drawOrder.length,
21         GLES20.GL_UNSIGNED_SHORT, drawListBuffer);
22
23     // Disable vertex array
24     GLES20 glDisableVertexAttribArray(mPositionHandle);
25     GLES20 glDisableVertexAttribArray(mTextureCoordHandle);
26 }
```

Kod ten wywoła vertex shader a następnie fragment shader, dzięki temu wyświetlane zostaną dwa trójkąty wypełnione zawartością obrazka.

```
1 private final String vertexShaderCode =
2     "attribute vec4 position;" +
3         "attribute vec2 inputTextureCoordinate;" +
4             "varying vec2 textureCoordinate;" +
5                 "void main()" +
6                     "{" +
7                         "gl_Position = position;" +
8                             "textureCoordinate = inputTextureCoordinate;" +
9                             "}";
10
11 private final String fragmentShaderCode =
12     "#extension GL_OES_EGL_image_external : require\n" +
13         "precision mediump float;" +
14             "varying vec2 textureCoordinate;" +
15                 "\n" +
16                     "uniform samplerExternalOES s_texture;" +
17                         "\n" +
18                             "void main() {" +
19                                 "gl_FragColor = texture2D( s_texture,
20                                     textureCoordinate );\n" +
21                                 "}" ;
22
```

Powyższy kod sprawia, że obraz z kamery jest wyświetlany na ekranie urządzenia. Jest to bazowa aplikacja, dzięki której możemy zweryfikować jak bardzo wykonywane kernele OpenCL używające pokazywanego obrazka wpłyną na ilość wyświetlanych klatek na sekundę.

#### 4.6.1. Konwersja obrazu z kamery do RGB w aplikacji

Standard OpenCL umożliwia prace na obrazkach w kernelach, jednakże te muszą być w formacie RGBA lub podobnych zawierające kanał czerwony zielony i niebieski. Niestety obraz z kamery zazwyczaj dostępny jest tylko w mediowym formacie YUV. Dlatego by móc wykorzystać obraz z kamery w kernelu OpenCL musimy przechwycić dane w formacie YUV a następnie przepisać je na format RGBA. W tym celu modyfikujemy kod bazowej aplikacji by ta tworzyła teksturę typu GL\_TEXTURE\_2D zamiast GL\_TEXTURE\_EXTERNAL\_OES. Dane o obrazie zarejestrowanym z kamery przechwytyjemy funkcją onPreviewFrame, która zostaje ustawiona jako previewCallback w obiekcie kamery. W tablicy byte[] data, przekazanej jako pierwszy argument znajduje się obrazek w formacie YUV. Konwersja z formatu YUV do RGBA wygląda następująco.

```
1 public static void yuv2rgb(byte[] rgba, byte[] yuv, int width, int
2 height) {
3     int total = width * height;
4     int Y, Cb = 0, Cr = 0, index = 0;
5     int R = 0, G = 0, B = 0;
6
7     for (int y = 0; y < height; y++) {
8         for (int x = 0; x < width; x++) {
9             Y = yuv[y * width + x];
10            if (Y < 0) Y += 255;
11
12            if ((x & 1) == 0) {
13                Cr = yuv[(y >> 1) * (width) + x + total];
14                Cb = yuv[(y >> 1) * (width) + x + total + 1];
```

```

14         if (Cb < 0) Cb += 127;
15         else Cb -= 128;
16         if (Cr < 0) Cr += 127;
17         else Cr -= 128;
18     }
19
20
21     R = Y + Cr + (Cr >> 2) + (Cr >> 3) + (Cr >> 5);
22     G = Y - (Cb >> 2) + (Cb >> 4) + (Cb >> 5) - (Cr >> 1) +
23         (Cr >> 3) + (Cr >> 4) + (Cr >> 5);
24     B = Y + Cb + (Cb >> 1) + (Cb >> 2) + (Cb >> 6);
25
26     if (R < 0) R = 0;
27     else if (R > 255) R = 255;
28     if (G < 0) G = 0;
29     else if (G > 255) G = 255;
30     if (B < 0) B = 0;
31     else if (B > 255) B = 255;
32     rgba[4 * index + 0] = ((byte) (R));
33     rgba[4 * index + 1] = ((byte) (G));
34     rgba[4 * index + 2] = ((byte) (B));
35     rgba[4 * index + 3] = ((byte) (255));
36     index++;
37 }
38 }
```

. Po wykonaniu tej funkcji w tablicy `rgb[]` znajduje się obraz w formacie RGBA. W celu zaktualizowania danych w naszej wyświetlanej teksturze wołamy następującą metodę.

```

1 GLES20.glTexSubImage2D(GLES20.GL_TEXTURE_2D, 0,
2             0, 0,
3             size.x, size.y,
4             GLES20.GL_RGBA, GLES20.GL_UNSIGNED_BYTE,
5             ByteBuffer.wrap(texture_data));
```

Po jej wykonaniu dane z tablicy `texture_data`, w którym znajduje się obrazek w formacie RGBA, zostaną umieszczone w wyświetlanej teksturze. Wykonywanie konwersji do RGBA po stronie aplikacji jest dużym obciążeniem wydajnościowym. By poprawić wydajność, konwersja została wykonywana przez kernel w OpenCL. Po stronie OpenCL stworzone zostają dwa bufore, do pierwszego kopujemy dane z kamery, po wykonaniu dane z drugiego bufora kopowane są tablicy, która później zostanie wpisana do tekstury.

Następnym usprawnieniem jest pisanie do wyświetlanej tekstury bezpośrednio z wykonywanego kernela. Dzięki temu oszczędzamy czas potrzebny na kopiowanie danych po wykonaniu kernela a następnie kopiowania ich do tekstury. Wystarczy, że kontekst OpenCL zostanie stworzony w oparciu o uchwyty na kontekście OpenGLa, a do kernela zostanie przekazany współdzielony obraz. Różnicą wykonywanego kernela od wyżej prezentowanego kodu po stronie aplikacji są deklaracja kernela, która wygląda następująco.

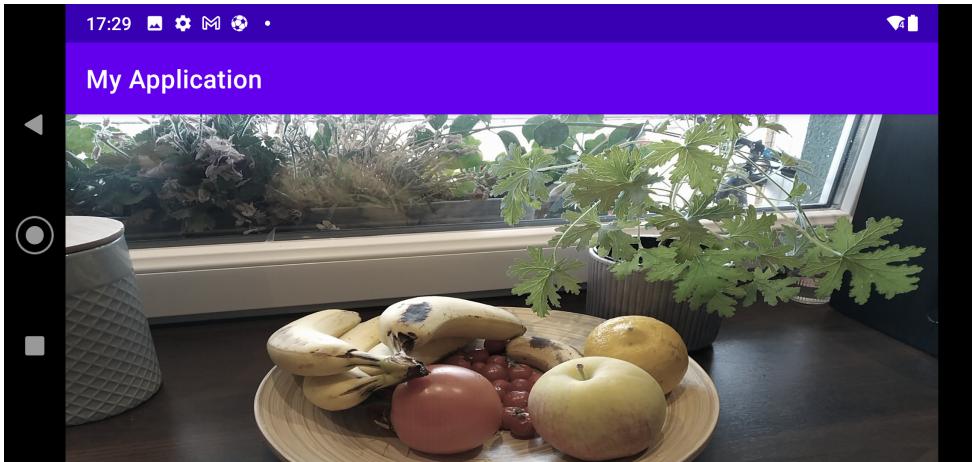
```

1 __kernel void convertNV21ToRGBImage(__write_only image2d_t img,
2                                     __global char *yuv, int width, int height);
```

By wyliczone wartości pixela zapisać w obiekcie obrazka należy sprowadzić je do postaci zde-noramlizowanej tj. do wartości z przedziału 0-1, a następnie wpisać je do obrazka funkcją write\_imagef.

```
1 write_imagef(img, (int2)(gidX, gidY), (float4)((float)(R)/(float)
2   (255)), ((float)(G)/(float)(255)), ((float)(B)/(float)(255)),
3   1.0));
```

Oto podgląd kamery po konwersji do RGB



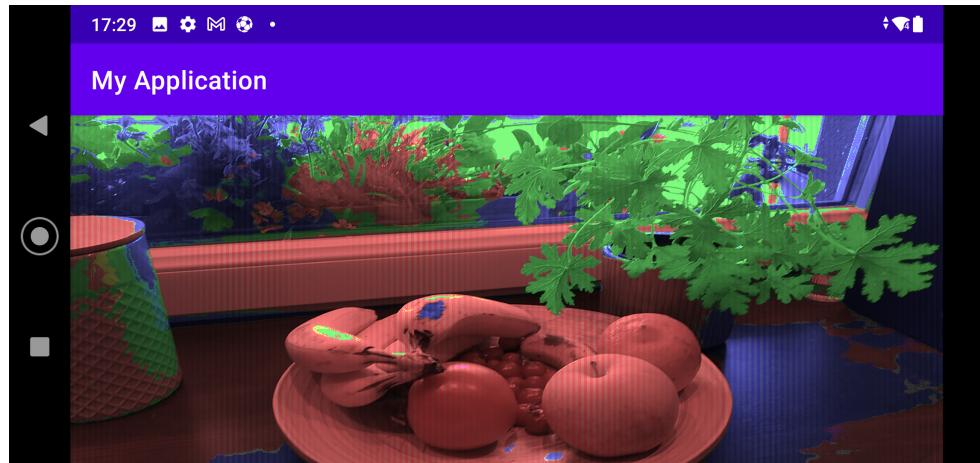
Rys. 4.1. Podgląd Kamery

#### 4.6.2. Filtr Max Rgb

Filtr Max RGB pozwala zwizualizować, który kanał posiada największą wartość dla danego piksela. By wykorzystać go przy wyświetlaniu obrazu z kamery należy uruchomić następujący kernel

```
1 __kernel void imageRgbMax(__write_only image2d_t dst ,__read_only
2   image2d_t src){
3     int gidX = get_global_id(0);
4     int gidY = get_global_id(1);
5     const int2 coord = (int2)(gidX, gidY);
6     const float4 pixel = read_imagef(src, coord);
7
8     float max = pixel.x;
9     if(max < pixel.y){
10       max = pixel.y;
11     }
12     if(max < pixel.z){
13       max = pixel.z;
14     }
15
16     if(pixel.x < max)
17       pixel.x = 0.0;
18     if(pixel.y < max)
19       pixel.y = 0.0;
20     if(pixel.z < max)
21       pixel.z = 0.0;
22     write_imagef(dst, coord, pixel);
}
```

Filtr ten wybiera kanał z największą wartością i zeruje wartości w pozostałych. OpenCL nie pozwala na używanie tego samego obrazka do czytania i pisania. Dlatego do kernela przekazane zostają dwa obrazki. Pierwszy z możliwością zapisu, jest to obrazek wynikowy, później wyświetlony. Drugi to obrazek z możliwością odczytu posiadający dane obrazu z kamery urządzenia. Podgląd kamery po zastosowaniu filtra wygląda następująco.



Rys. 4.2. Filtr Max Rgb

#### 4.6.3. Podgląd w skali szarości

Filtr pozwalający przedstawić obraz w skali szarości. By użyć filtra należy wykonać następujący kernel



```
1 __kernel void blackWhite(__write_only image2d_t dst ,__read_only
2     image2d_t src){
3     int gidX = get_global_id(0);
4     int gidY = get_global_id(1);
5     const int2 coord = (int2)(gidX, gidY);
6     const float4 pixel = read_imagef(src, coord);
7     float value = (pixel.x + pixel.y + pixel.z) / 3;
8     write_imagef(dst, coord, (float4)(value, value, value, 1.0));
}
```



Kernel ten wylicza średnią wartość wszystkich kanałów, następnie tę wartość przypisuje każdemu z nich. Podgląd z kamery z tym filtrem wygląda następująco



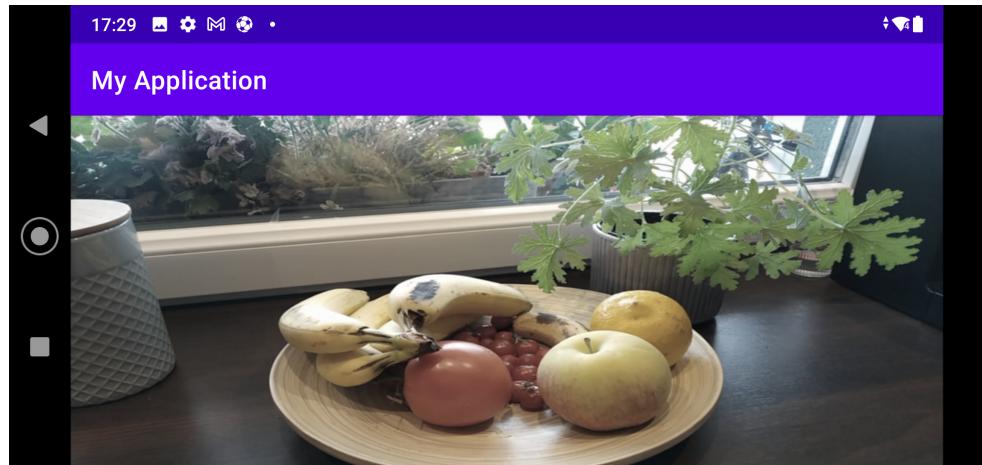
Rys. 4.3. Podgląd w skali szarości

#### 4.6.4. Filtr Uśredniający

Filtr Uśredniający polega na wyliczeniu wartości piksela na podstawie uśrednionej wartości pikseli sąsiadujących. Dzięki temu filtrowi dochodzi zmniejszenia różnic między sąsiednimi pikselami, przez co dochodzi do rozmycia obrazu i zmniejszenia kontrastu. Kernel nakładający filtr uśredniający wygląda następująco.

```
1  __kernel void avgFilter(__write_only image2d_t dst ,__read_only
2   image2d_t src){
3     int gidX = get_global_id(0);
4     int gidY = get_global_id(1);
5     int width = get_image_width(src);
6     int height = get_image_height(src);
7     float4 value = (float4)(0.0, 0.0, 0.0, 0.0);
8     for (int i = -2; i < 3;i++) {
9       for(int j = -2;j < 3; j++){
10         if(gidX + j < 0 || gidX + j > width)
11           continue;
12         if(gidY + i < 0 || gidY + i > height)
13           continue;
14         value += read_imagef(src, (int2)(gidX + j, gidY + i));
15     }
16     write_imagef(dst, (int2)(gidX, gidY), (float4)(value.x/25, value.y
17     /25, value.z/25, 1.0));
```

Kernel ten wylicza średnią wartość pixeli w kwadracie  $5 \times 5$  i wpisuje tą wartość do obrazu wynikowego. Dla każdego pixela zdjęcia źródłowego zostaje wyliczona jedna uśredniona wartość. Podgląd kamery z tym filtrem wygląda tak.



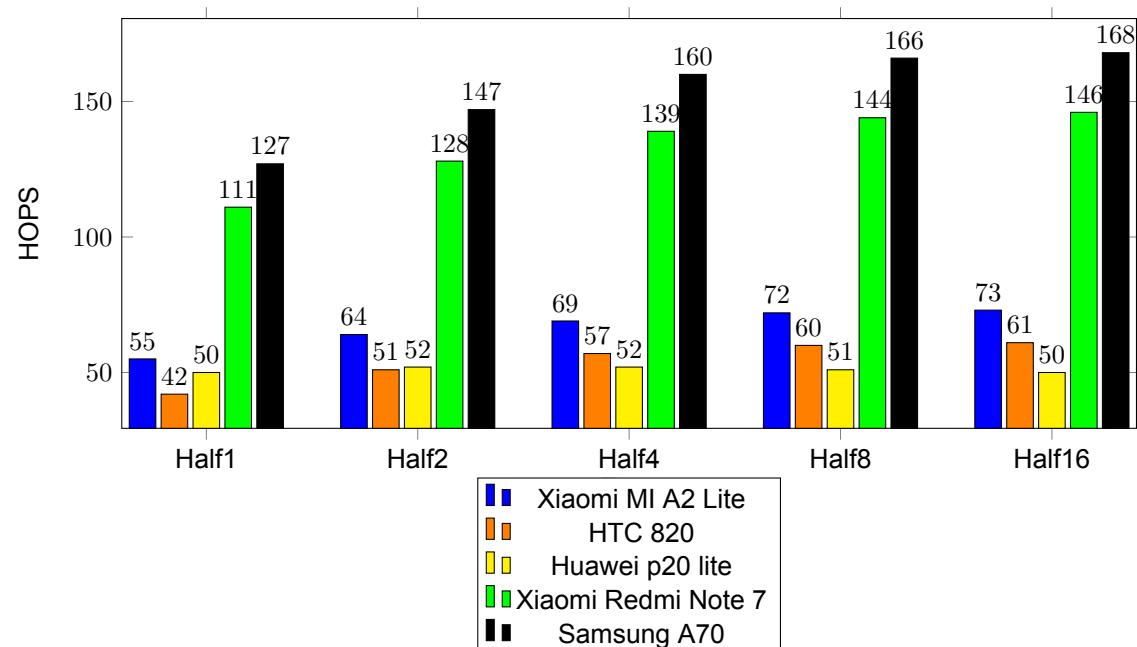
Rys. 4.4. Podgląd z filtrem uśredniającym

## 5. WYNIKI, ANALIZA I WNIOSKI

W tym rozdziale opisane zostały wyniki testów opisanych w rozdziale 4, na urządzeniach opisanych w rozdziale 3. Urządzenie HTC Desire 820 testowane było dla dwóch wersji systemu Android, które posiadały różne wersje sterownika OpenCL. Wyniki dla tych dwóch wersji nie różniły się, dlatego poniżej umieszczone zostały wyniki tylko dla nowszej wersji sterownika.

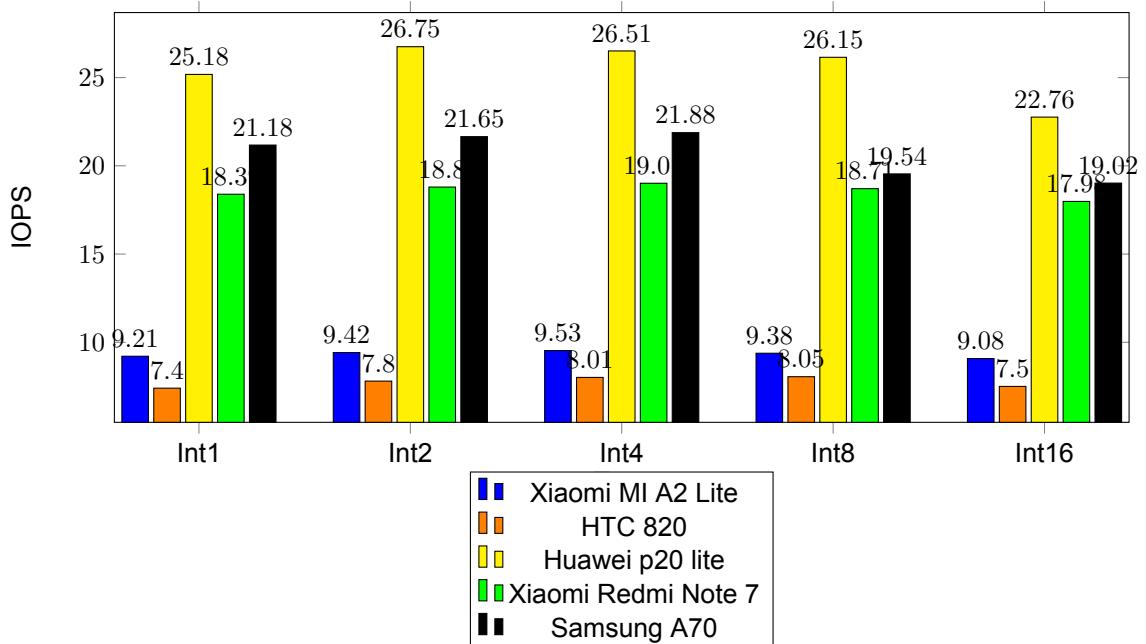
### 5.1. Wyniki Mocy Obliczeniowej

Poniższe wykresy prezentują osiągi testowanych urządzeń pod kątem ilości wykonywanych operacji na sekundę. Przetestowane zostały możliwości, dla różnych typów danych.



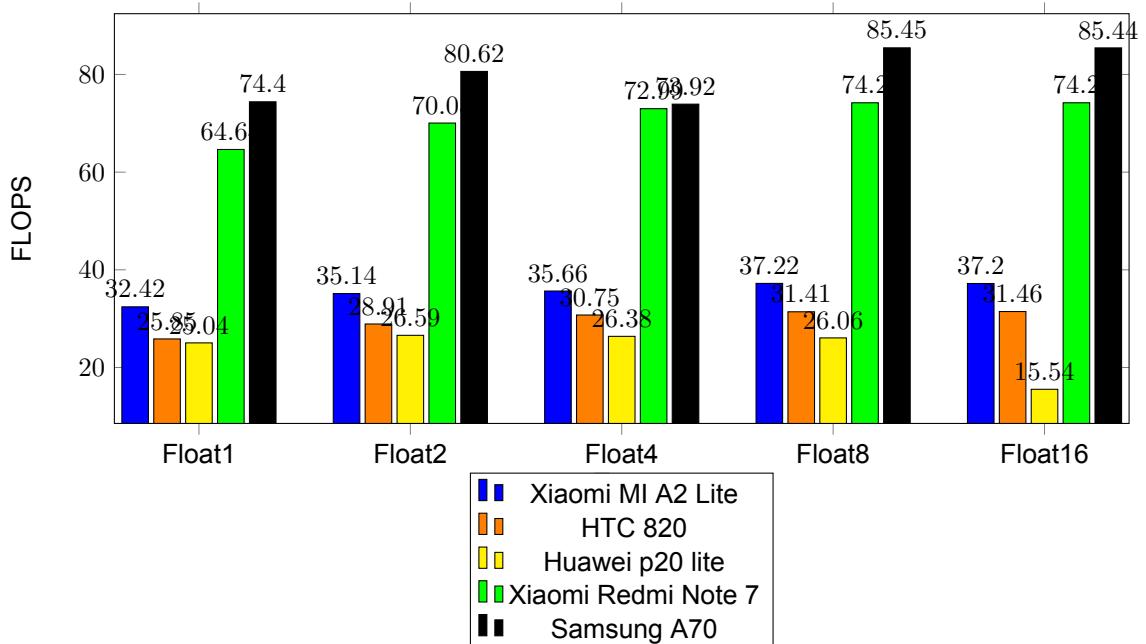
Rys. 5.1. Typ Half Moc Obliczeniowa

Na powyższym wykresie widać, że na procesorach Adreno dzięki użyciu wektorowych typów ilość operacji na sekundę rośnie . Dla tych procesorów wyniki układają się według ich numerów, najgorzej radzi sobie HTC 820 z Adreno 405 a najlepiej Samsung A70 z Adreno 612. W przypadku procesora Mali niezależnie od wielkości typu ilość operacji pozostaje na tym samym poziomie. Dla porównywalnych modeli Xiaomi Mi A2 Lite i Huawei P20 Lite lepiej wpada urządzenie od Xiaomi.



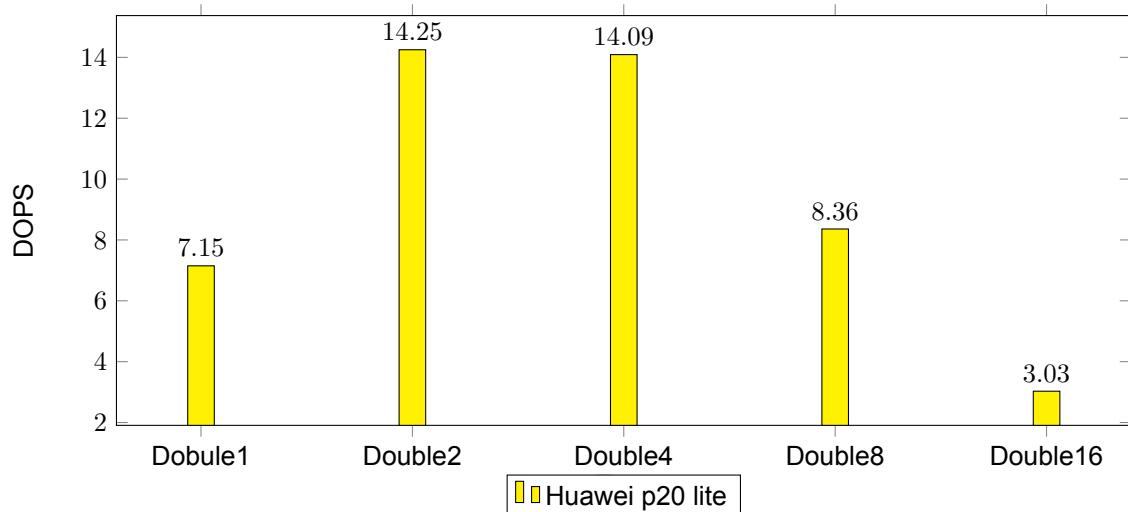
Rys. 5.2. Typ Int Moc Obliczeniowa

W przypadku operacji wykonywanych na liczbach całkowitych najlepiej radzi sobie procesor Mali, użycie typów wektorowych nieznacznie poprawia wydajność, a ta wynosi około 26 GIOPS. W przypadku procesorów Adreno podobnie jak procesor Mali wydajność znacznie poprawia się przy użyciu typów int2 i int4 natomiast przy użyciu typów int8 i int16 wydajność spada. W tym przypadku także kolejność od najgorszego do najlepszego układu się według ich numerów, najlepiej Adreno 612, najgorzej Adreno 405.



Rys. 5.3. Typ Float Moc Obliczeniowa

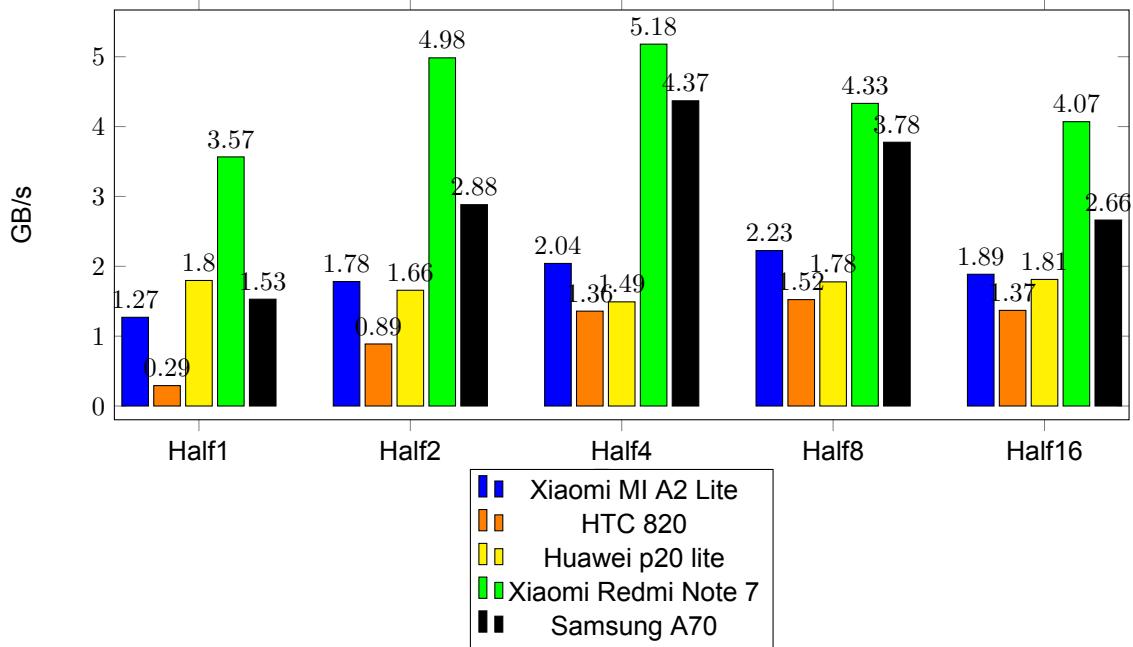
W testach wydajność, przy operacjach na typie zmiennoprzecinkowym pojedynczej precyzyji float, zdecydowanie najgorzej wypada na procesorze Mali T830 2MP. Wydajność dla tego procesora dla typów wektorowych wypada na podobnym poziomie, słabnie jednak dla typu float16 o 40%. W przypadku procesorów Adreno sytuacja wygląda podobnie jak dla typu Half, użycie typów wektorowych zwiększa wydajność, dla typu Float16 widać że wartość operacji na sekundę jest o mniej więcej połowę mniejsza niż dla typu Half16.



Rys. 5.4. Typ Double Moc Obliczeniowa

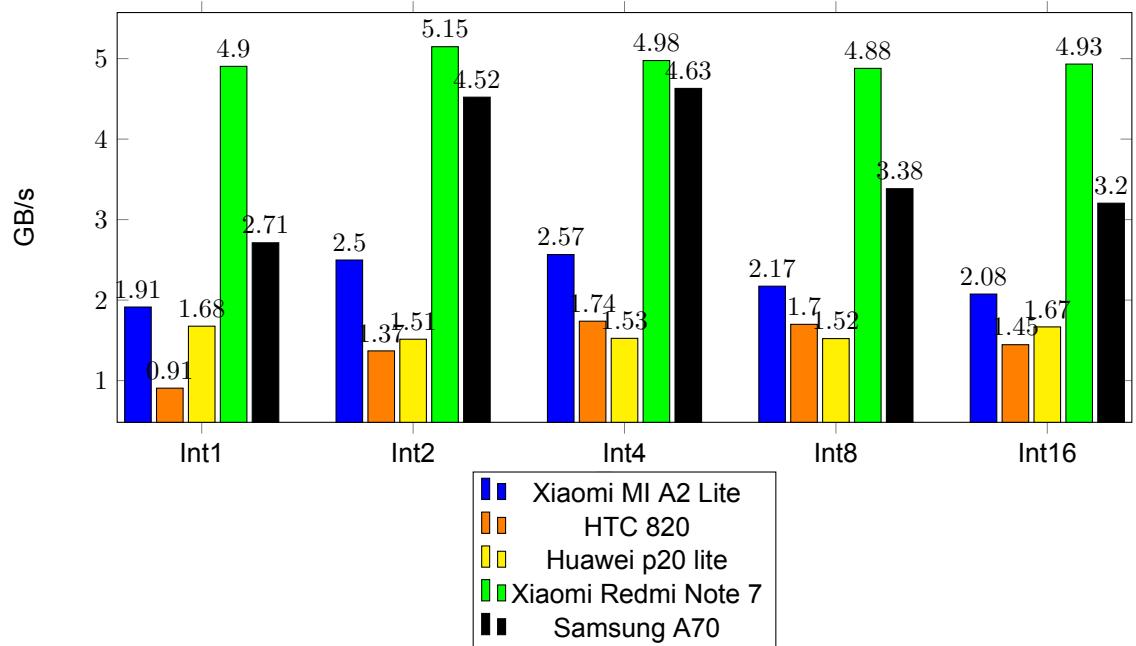
Huawei P20 Lite jako jedynie testowane urządzenie posiada obsługę typów zmiennoprzecinkowych podwójnej precyzyji. Ewidentnie użycie typów wektorowych double2 i double4 poprawia wydajność o 100% w stosunku do typu double. Operacje na typie double8 poprawiają wydajność względem double o 17%, natomiast operacje na typie double16 pogarszają wydajność o 57% względem typu double.

## 5.2. Wyniki Przepływu Pamięci

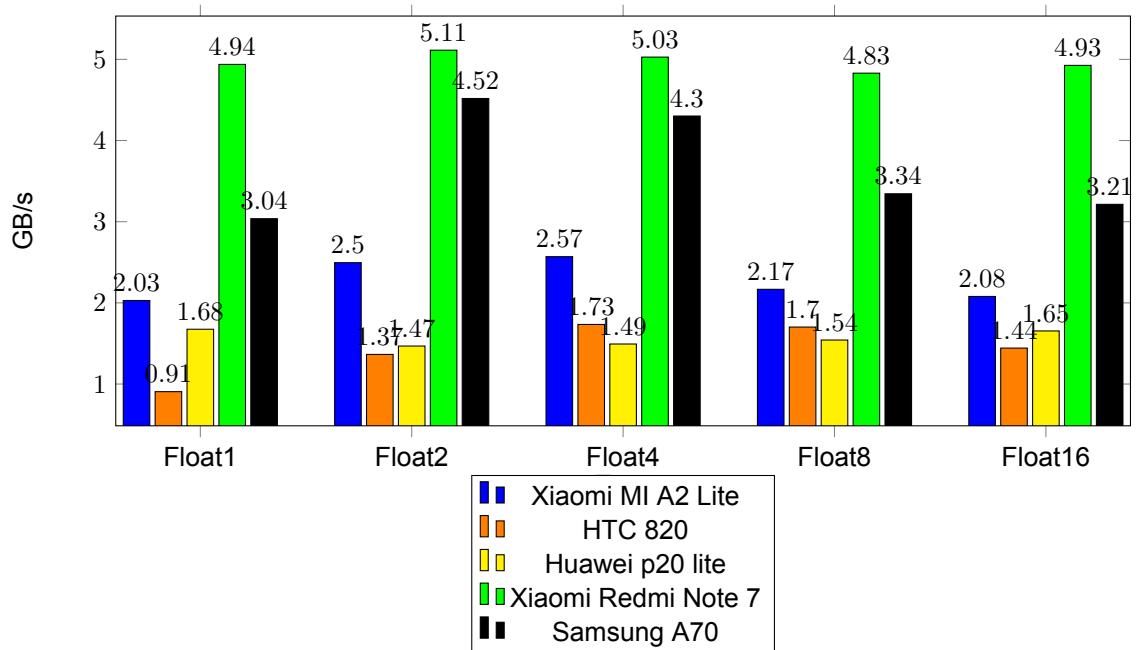


Rys. 5.5. Typ Half Przepływ Pamięci

Podobnie jak w przypadku testowania mocy obliczeniowej, testując szybkość przepływu danych między aplikacjami, widać, że procesor dla procesora Mali nie obserwujemy dużych korzyści korzystania z typów wektorowych. Dla typów Half1, Half2 i Half16 posiada lepsze wyniki niż urządzenie z procesorem Adreno 405. W pozostałych próbach wypada najgorzej. Spośród urządzeń z procesorami Adreno widać, że najgorsze wyniki uzyskane są dla typu Half 1 a najlepsze dla typu Half 4. Inaczej niż w przypadku mocy obliczeniowej, najlepiej wypada urządzenie z procesorem Adreno 512. Najprawdopodobniej osiąga lepszy wynik dzięki zastosowaniu lepszego typu pamięci.

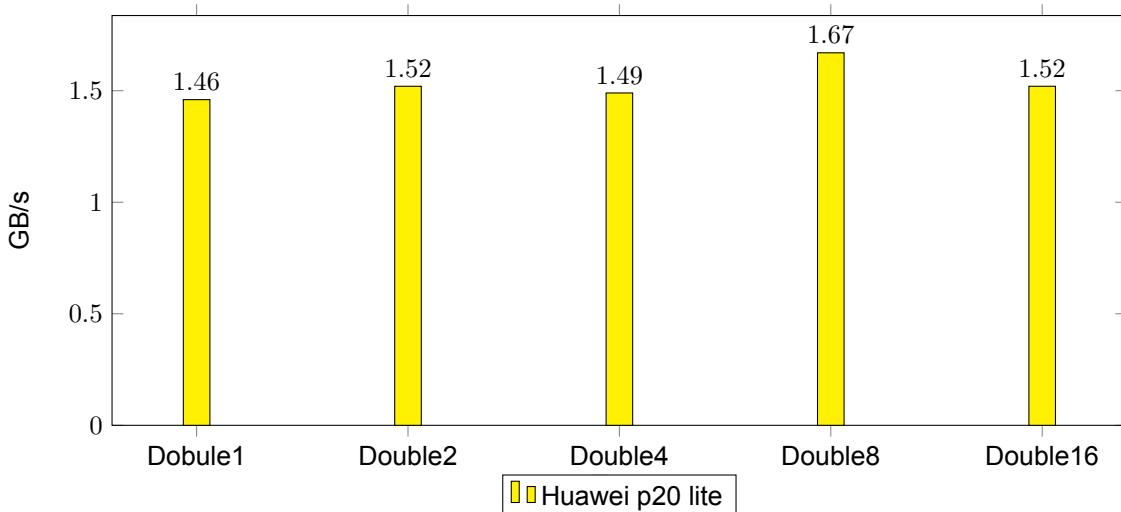


Rys. 5.6. Typ Int Przepływ Pamięci



Rys. 5.7. Typ Float Przepływ Pamięci

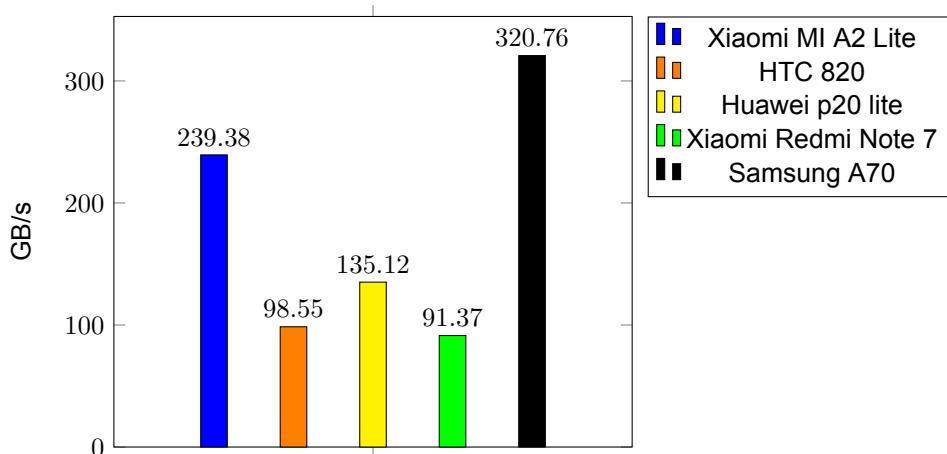
Wyniki przepływu danych dla typów float i int osiągają podobne wartości jak w przypadku wektorowych typów Half. Kopiowanie pojedynczych wartości typu Half wypada gorzej w porównaniu do typów 32 bitowych. Najlepsze wyniki osiąga urządzenie Xiaomi Redmi Note 7, które posiada procesor graficzny Adreno 512 i najlepszy typ pamięci spośród wszystkich testowanych urządzeń LPDDR4 o częstotliwości 1866MHz i przepustowości do 29.8GB/s.



Rys. 5.8. Typ Double Przepływ Pamięci

Szybkość transferu pamięci używając typu double nie różni się względem typów Half Int czy Float.

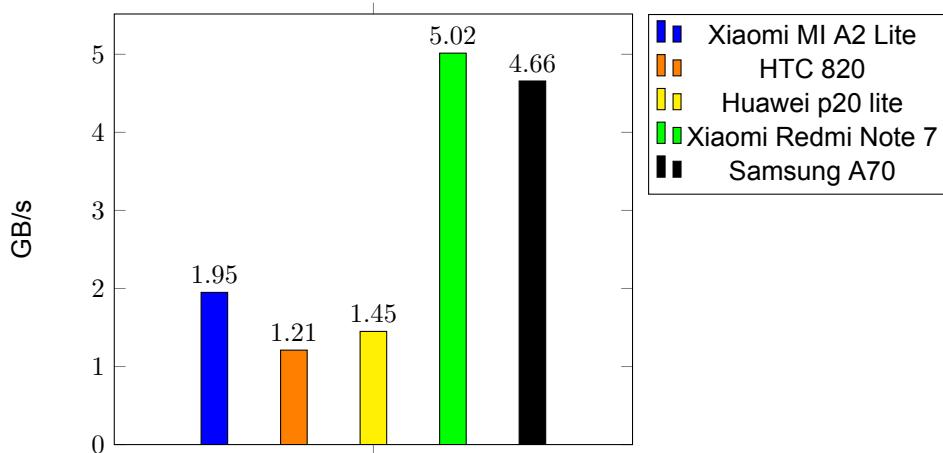
### 5.3. Czas Oczekiwania na wykonanie



Rys. 5.9. Czas oczekiwania na wykonanie

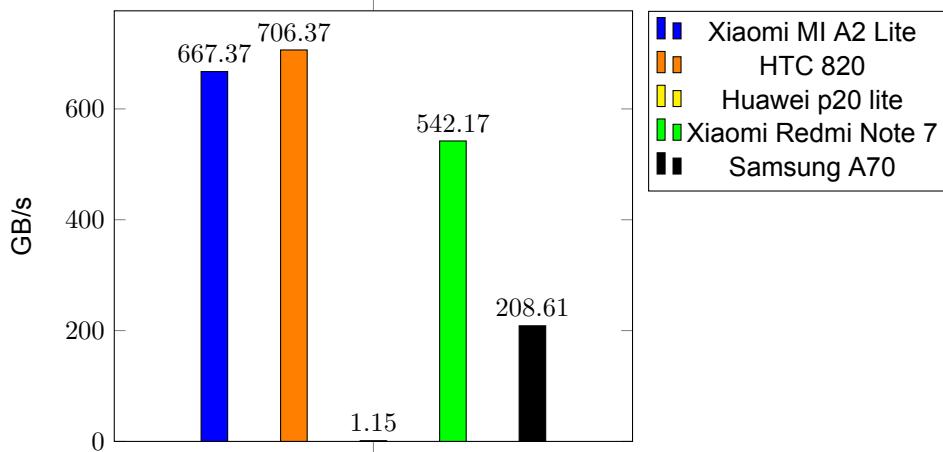
Powyższy wykres pokazuje wynik od zakolejkowany do rozpoczęcia wykonania na gpu. Im mniejszy czas tym lepiej. Ku zaskoczeniu najgorzej wypada Samsung Galaxy A70, który był testowany na najnowszym sterowniku OpenCL spośród testowanych. Za to drugi najlepszy wynik osiągnął HTC Desire 820, czyli najstarsze urządzenie, z najstarszym sterownikiem. Czas mierzony w tym teście to okres od umieszczenia zadania w kolejce openCL, przez sterownik kernela systemu, do urządzenia na którym będzie wykonywane. Zależy więc on bardziej od implementacji poszczególnych sterowników, niż od samego urządzenia.

#### 5.4. Transfery Pamięci Aplikacja-Urządzenie



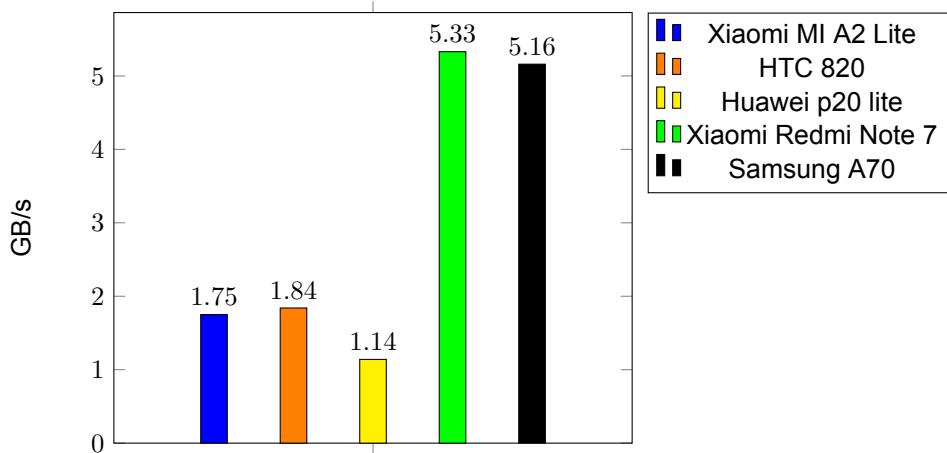
Rys. 5.10. czas wykonywania `clEnqueueReadBuffer`

Powyższy wykres pokazuje szybkość transferu danych z urządzenia do aplikacji. Transfery tego typu odbywają się przy pomocy funkcji zdefiniowanych przez bibliotekę OpenCL. `clEnqueueReadBuffer` kopiuje dokładną ilość bajtów, więc transfer najprawdopodobniej odbywa się przy pomocy 8bitowego typu danych. Wykres przedstawia zależności analogiczne do testu z przepływem danych wewnętrz urządzeniami.



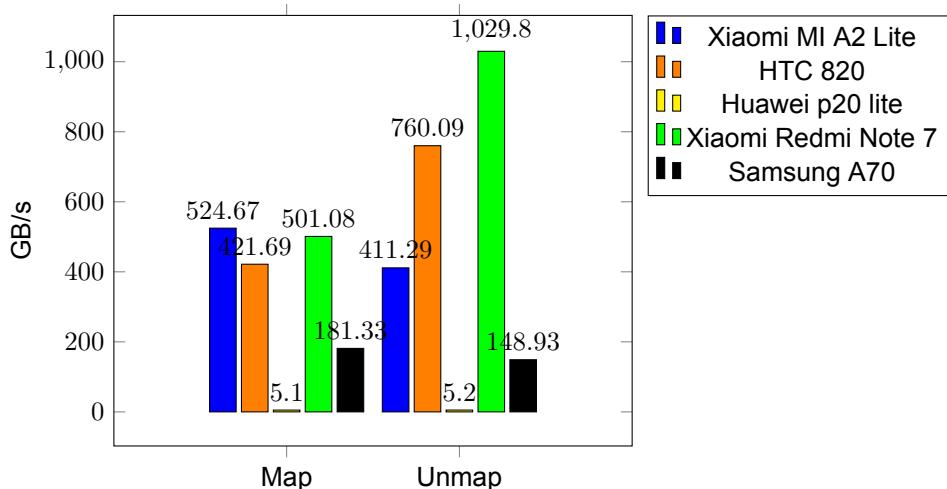
Rys. 5.11. czas wykonywania `clEnqueueWriteBuffer`

Wykres szybkości odczytu danych z alokacji w OpenCL, w przypadku procesorów Adreno pokazuje wartości nie możliwe do osiągnięcia w typach pamięci zastosowanych w urządzeniach. Zakładany maksymalny przepływ pamięci np dla Samsunga Galaxy A70 to 14.9GB/s. Wartości te zostały zebrane przy pomocy obiektów `clEvent`. Ewentualnie wartości zebrane podczas `clEnqueueWriteBuffer` są błędne. Dla pewności powtórzono test mierząc czas przy pomocy funkcji systemowych po stronie hosta. Uzyskane wyniki prezentują się na poniższym wykresie. Wartości są zbliżone do tych uzyskanych w poprzednich testach z przepływem pamięci w ramach urządzenia.



Rys. 5.12. czas wykonywania clEnqueueWriteBuffer czas z Aplikacji

Dla pewności powtórzone zostały pozostałe testy z wykorzystaniem mierzenia czasu po stronie aplikacji, wszystkie czasy pokrywały się z tymi zmierzonymi przy wykorzystaniu obiektów clEvent.

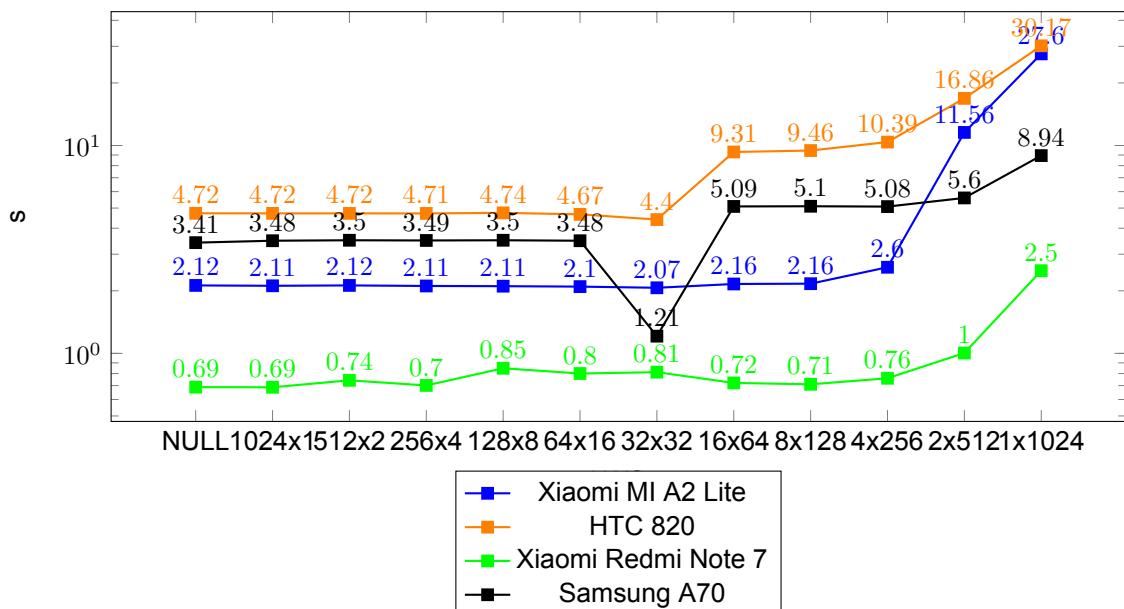


Rys. 5.13. czas wykonywania Map unMap

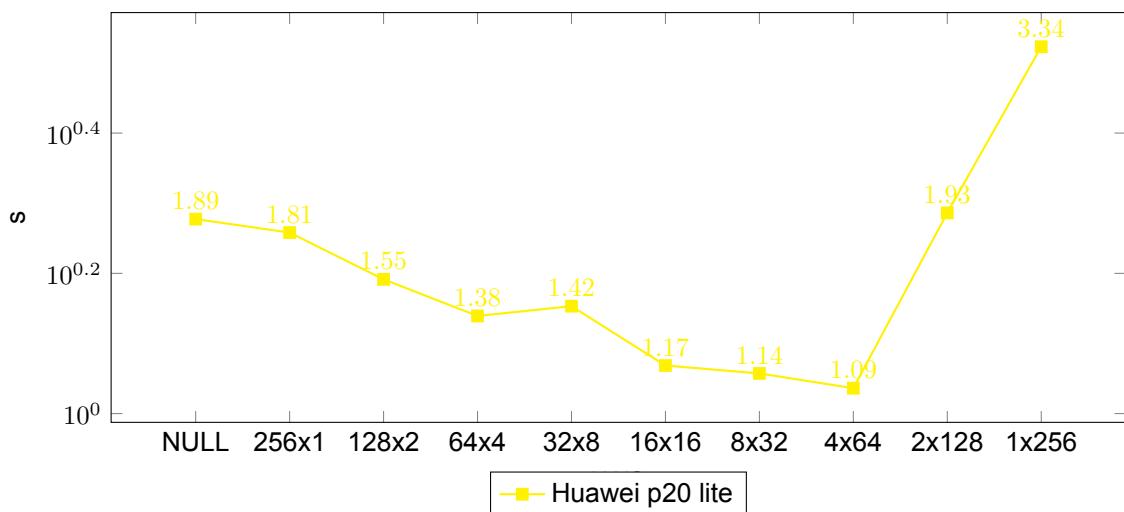
Powyższy wykres pokazuje czas potrzebny do mapowania pamięci po stronie urządzenia na pamięć po stronie aplikacji. Wartości uzyskane dla procesorów Adreno podobnie jak we wcześniejszym teście wydają się podejrzane, jednak powtórzone z mierzeniem czasu po stronie aplikacji zwraca podobne rezultaty. Procesory Adreno korzystają ze współdzielonej pamięci systemowej. Najprawdopodobniej zwracany przez funkcję clEnqueueMapBuffer jest wskaźnik na pamięć, która jest wykorzystywana przez urządzenie podczas wykonywania kerneli. Czas potrzebny na mapowanie i od mapowania pamięci to krótki czas w którym sterownik zwraca wskaźnik na pamięć używaną przez alokację OpenCL. W przypadku Urządzenia Huawei P20 Lite z procesorem Mali T830 MP pamięć nie jest współdzielona. Wyraźnie widać, że procesor Mali znacznie wolniej mapuje pamięć. Mimo braku współdzielenia proces mapowania odbywa się znacznie szybciej niż

odczytywanie z bufora za pomocą `c1EnqueueReadBuffer`. Prawdopodobnie wskaźnik, na który mapowany jest bufor, zlokalizowana jest w korzystniejszym miejscu pamięci fizycznej, niż pamięć przydzielona aplikacji przez system.

### 5.5. Mnożenie Macierzy



Rys. 5.14. Mnożenie macierzy (Max Lws 1024)



Rys. 5.15. Mnożenie macierzy (Max Lws 256)

Powyższe dwa wykresy pokazują zależność wielkości lokalnej work grupy od czasu w jakim przemnożone zostaną dwie macierze, w tym wypadku obie o rozmiarze 1024x1024. Wyraźnie widać, że rozmiar lokalnej work grupy wpływa na czas wykonania zadania jakim jest mnożenie macierzy. Wszystkie urządzenia osiągają najgorszy czas w przypadku gdy rozmiar lokalnej grupy w wymiarze X wynosi 1, a w wymiarze Y mam maksymalną możliwą dla urządzenia wartość. Naj-

prawdopodobniej jest to spowodowane czytaniem odległych od siebie komórek pamięci w ramach work grupy. Pojedynczy wiersz macierzy zajmuje 4KB, zatem elementy z pod indexu (0,0) i (0,1) są odległe od siebie o 4KB. W dodatku 4KB to odległość adresu wirtualnego, fizycznie w zależności od rozmiaru stron pamięci, te elementy mogą znajdować się w innych miejscach pamięci fizycznej. Cała macierz zajmuje 4MB pamięci, podczas wykonywania mnożenia używane są trzy takie macierze. Mobilne procesory graficzne nie posiadają tak dużej pamięci dedykowanej, a tym bardziej pamięci cache, dlatego przeładowywane każdorazowe pamięci dla każdego elementu lokalnej grupy jest bardzo kosztowne.

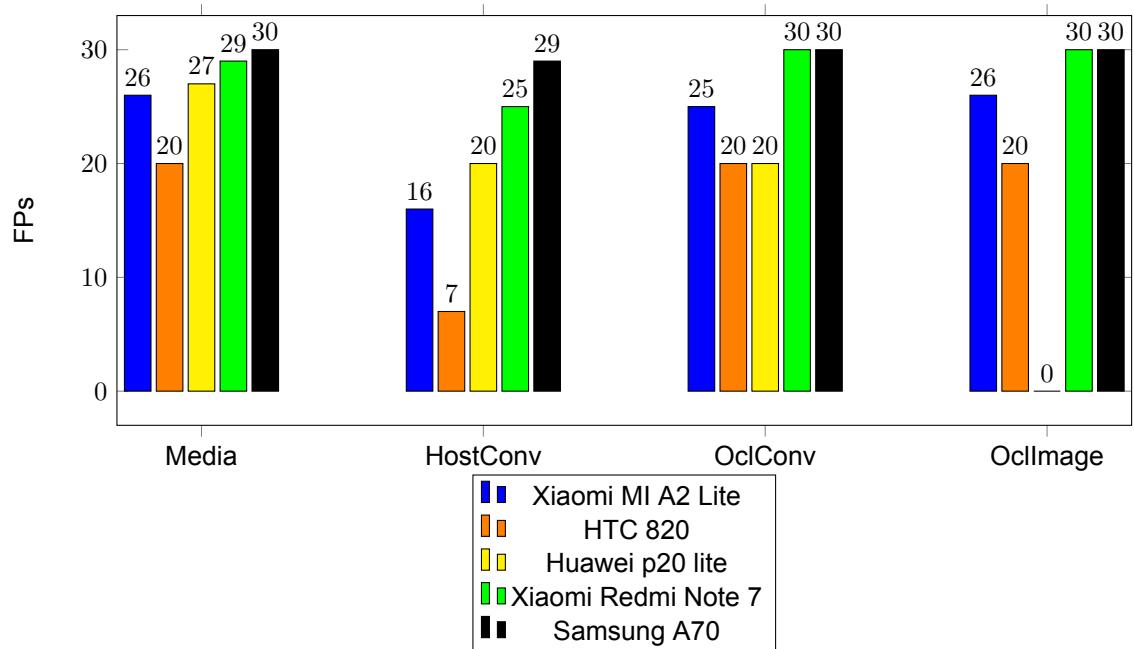
Dla urządzeń z procesorami Adreno 405, 506 i 612, najbardziej optymalnym rozmiarem lokalnej work grupy wydaje się rozmiar 32x32. W przypadku gdy mnożymy macierz A razy Macierz B, rozmnażamy rząd z pierwszej przez kolumnę z drugiej. Przy postępowaniu pamięci macierzy B w ramach pojedynczego elementu work grupy, także musimy dostępować odległe od siebie elementy pamięci. Najprawdopodobniej gdy rozmiar grupy wymiarze y wynosi 32, pamięć używana przez wątek w ramach SIMD jest dostępna w dedykowanej pamięci procesora graficznego.

Najszybciej mnożenie macierzy wykonało urządzenie Xiaomi Redmi Note 7 najprawdopodobniej to dzięki najlepszemu typowi pamięci spośród testowanych telefonów, koszty dostępu i odczytu pamięci były najmniejsze. Drugim najszybszym urządzeniem jest Huawei P20 Lite z procesorem graficznym od Mali. Test korzystał z macierzy danych o typie całkowitym, jak wcześniej sprawdzono to właśnie te urządzenie może wykonać najwięcej operacji tego typu w określonym czasie.

### **5.6. OpenCL z Kamera Api**

Wykresy w tym podrozdziale pokazują jak użycie OpenCL do obróbki danych z kamery w czasie rzeczywistym wpływa na ilość wyświetlanych klatek na sekundę.



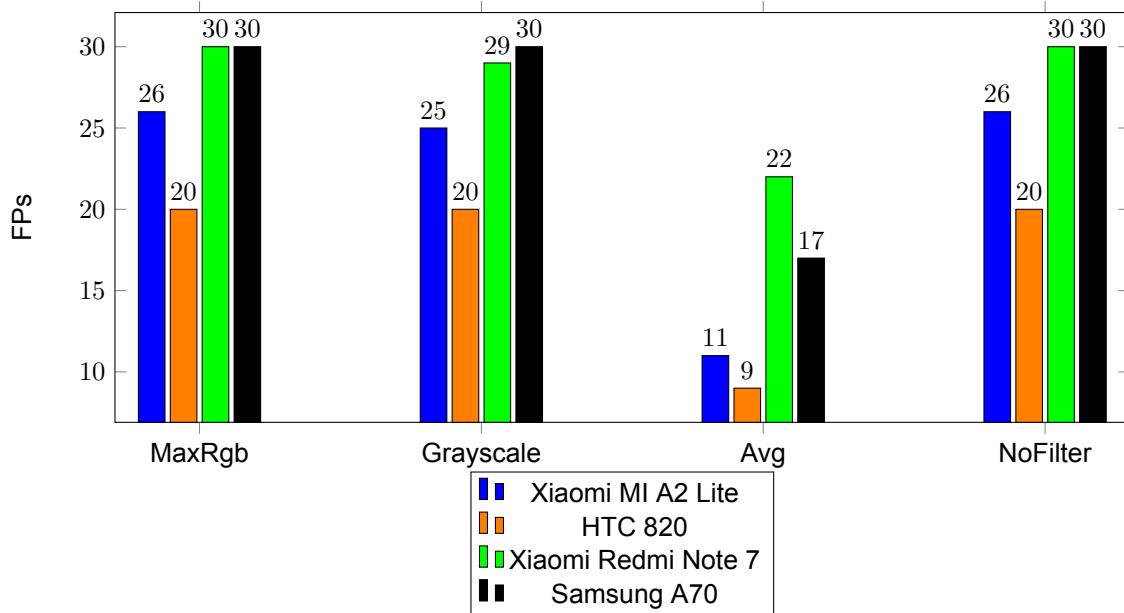


Rys. 5.16. Konwersja do RGB

Pierwsza kolumna(Media) pokazuje wartości w klatkach na sekundę, jakie osiągają urządzenia przy wyświetlanie używając środowiska OpenGL, tekstury otrzymaną z CameraApi. Textura posiada obrazek zakodowany w formacie mediowym NV21.

Niestety Api OpenCL umożliwia współdzielenia zasobów z OpenGL tylko w formacie RGBA, więc niemożliwe jest dzielenie tekstury zwróconej przez CameraApi. Potrzebna jest transformacja z formatu NV21 do RGBA. Druga kolumna wykresu pokazuje jak ilość klatek na sekundę wygląda, gdy konwersja do RGBA jest wykonywana po stronie aplikacji. Taka operacja ma duży wpływ na wszystkie urządzenia, widać wyraźny spadek wyświetlanych klatek na sekundę w każdym z nich. Trzecia kolumna opisuje sytuacje gdy konwersja wykonywana jest w środowisku openCL. Wyraźnie poprawia to wydajność na urządzeniach z procesorami graficznymi Adreno. Telefony te osiągają wartości z pierwszej kolumny, uwzględniając błąd pomiaru. Telefon Huawei z procesorem Mali nie poprawia rezultatu. Najprawdopodobniej dla tego urządzenia czas potrzebny na kopowanie danych do alokacji w OpenCL i przetworzenie kernela jest równie kosztowne co brak zrównoleglenia konwersji po stronie CPU.

W ostatniej kolumnie widnieją rezultaty dla przypadku w którym w kernelu OpenCL zapisujemy bezpośrednio do współdzielonego obrazka, który potem jest wyświetlany, zaoszczędzimy w ten sposób czas na dwa kopowania. Pierwszym z alokacji w OpenCL do pamięci po stronie aplikacji, i drugie z aplikacji do tekstury OpenGL. Na wykresie nie widać, żeby to jakkolwiek wpłynęło na liczbę wyświetlanych klatek. Najprawdopodobniej inne optymalizacje, nie przyniosłoby żadnych poprawek wydajności. Wąskie gardło, które sprawia, że nie można przekroczyć pewnej liczby klatek jest po stronie CameraApi, które nie jest w stanie dostarczyć większej ilości klatek na sekundę, lub środowiska OpenGL, które nie jest w stanie wyświetlić więcej klatek.



Rys. 5.17. Filtry

Powyższy wykres pokazuje wpływ filtrów wykonanych przez kernele w OpenCL na ilość wyświetlanego klatek na sekundę. Można zauważyć, że użycie prostych filtrów takich jak max rgb czy transformacja na obrazek w skali szarości, nie wpływają na ilość wyświetlanego klatek. Prawdopodobnie przez wąskie gardło które występuje gdzieś w czasie od zebrania podglądu z kamery do jego wyświetlania. Natomiast przy filtrze uśredniającym, który w kernelu dostępuje wielu pixeli znacząco wpływa na wydajność. W tym wypadku najlepiej wypada telefon z procesorem graficznym Adreno 512, w którym zastosowano najlepszy spośród testowanych typ pamięci.

## **6. APLIKACJE WYKORZYSTUJĄCE OPENCL PRZYPADKI UŻYCIA**

## WYKAZ LITERATURY

1. Munshi A. "The OpenCL Specification". W: *Khronos OpenCL Working Group* (2012).
2. Sawerwain M. "OpenCL. Akceleracja GPU w praktyce". W: *PWN* (2014).
3. Tay R. "OpenCL Parallel Programming Development Cookbook". W: *PACKT* (2013).
4. Fang J Huang C Tang T Wang Z. "Parallel Programming Models for Heterogeneous Many-Cores". W: *A Survey, preprint*, (2020).



## WYKAZ RYSUNKÓW

2.1.	Loader.....	10
2.2.	Wielowymiarowy ND Range .....	12
3.1.	SIMD .....	18
4.1.	Podgląd Kamery .....	28
4.2.	Filtr Max Rgb .....	29
4.3.	Podgląd w skali szarości.....	30
4.4.	Podgląd z filtrem uśredniającym.....	31
5.1.	Typ Half Moc Obliczeniowa .....	32
5.2.	Typ Int Moc Obliczeniowa.....	33
5.3.	Typ Float Moc Obliczeniowa.....	33
5.4.	Typ Double Moc Obliczeniowa .....	34
5.5.	Typ Half Przepływ Pamięci .....	35
5.6.	Typ Int Przepływ Pamięci.....	36
5.7.	Typ Float Przepływ Pamięci.....	36
5.8.	Typ Double Przepływ Pamięci .....	37
5.9.	Czas oczekiwania na wykonanie .....	37
5.10.	czas wykonywania clEnqueueReadBuffer .....	38
5.11.	czas wykonywania clEnqueueWriteBuffer .....	38
5.12.	czas wykonywania clEnqueueWriteBuffer czas z Aplikacji.....	39
5.13.	czas wykonywania Map unMap .....	39
5.14.	Mnożenie macierzy(Max Lws 1024) .....	40
5.15.	Mnożenie macierzy(Max Lws 256) .....	40
5.16.	Konwersja do RGB .....	42
5.17.	Filtры .....	43

## **WYKAZ TABEL**

## **Dodatek A: PRZYKŁADOWY DODATEK**

### **A.1 *Sekcja***