

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	6
1. Wprowadzenie	7
1.1. Cel pracy.....	7
2. Wstęp do Api OpenCL	8
2.1. Linkowanie biblioteki OpenCL na platformie Android	8
2.2. Typowy przebieg aplikacji OpenCL.....	9
2.3. Możliwości i ograniczenia sprzętowe.....	11
2.4. Pomiary czasu wykonywania kerneli	14
2.5. Współdzielenie obiektów pomiędzy OpenCL i OpenGL	14
3. Specyfika testowanych Urządzeń	16
4. Testy Wydajnościowe OpenCL na Urządzeniach z systemem Android.....	17
4.1. Pomiar Mocy Obliczeniowej.....	17
4.2. Przepływ pamięci.....	19
4.3. Czas Oczekiwania na wykonanie kernela	19
4.4. Transfer pamięci wbudowanymi funkcjami	20
4.5. Mnożenie macierzy	20
4.6. OpenCL do filtrowania obrazu z cameraApi	21
4.6.1. Konwersja obrazu z kamery do RGB w aplikacji.....	22
4.6.2. Filtr Max Rgb	24
4.6.3. Podgląd w skali szarości	25
4.6.4. Filtr Uśredniający.....	26
5. Analiza Wyników, Wnioski.....	28
5.1. Wyniki Mocy Obliczeniowej	28
5.2. Wyniki Przepływu Pamięci	30
5.3. Czas Oczekiwania na wykonanie	32
5.4. Transfery Pamięci Aplikacja-Urządzenie	32
5.5. Mnożenie Macierzy	34
5.6. OpenCL z Kamera Api.....	35
6. Aplikacje wykorzystujące OpenCl Przypadki Użycia	36
Wykaz literatury	37
Wykaz rysunków.....	37
Wykaz tabel	38
Dodatek A. Przykładowy dodatek.....	39

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 posiadanej procesora głównego czy graficznego. Jest to duża zaleta w porównaniu na przykład do CUDA, która to jest interfejsem implementowanym jedynie przez Nvidia. 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 wstecznie. 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, lub cl_qcom_android_native_buffer_host_ptr dostępny na procesorach Qualcoma 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 kompletną 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 roz-

wiązania jest konieczność komplikacji takiej biblioteki, natomiast dzięki temu możemy zbudować aplikację działającą na wielu urządzeniach. <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
26 program = clCreateProgramWithSource(context, 1, &kernelStrings, 0,
27                                     &err);
28 err = clBuildProgram(program, 1, &device_id, nullptr, nullptr,
29                      nullptr);
30 kernel = clCreateKernel(program, "increment", &err);
31 cl_mem_flags flags = CL_MEM_READ_WRITE;
32 buffer = clCreateBuffer(context, flags, bufferSize, nullptr, &err);
33 void *ptr = clEnqueueMapBuffer(queue, buffer, CL_TRUE, CL_MAP_READ,
34                                0, bufferSize, 0, nullptr, nullptr, &err);
35 memset(ptr, 13, bufferSize);
36 err = clEnqueueUnmapMemObject(queue, buffer, ptr, 0, nullptr,
37                               nullptr);
38 err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer);
39 err = clEnqueueNDRangeKernel(queue, kernel, dimension, offset, gws,
40                             lws, 0, 0, nullptr);
41 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. Drugie wywołanie `clGetPlatformIDs` wpisze informacje o podanej liczbie platform i zapisze je w tablicy podanej w drugim argumencie. 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 buforami.

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

lejkę, która będzie używać obrazka. Jest to wykonywane w celu wywłaszczenia obiektu przez OpenCL. Po zakończeniu operacji na obiekcie należy go zwolnić polecienniem clEnqueueReleaseGLObjets.

3. SPECYFIKA TESTOWANYCH URZĄDZEŃ

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     "}" ;
```

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.

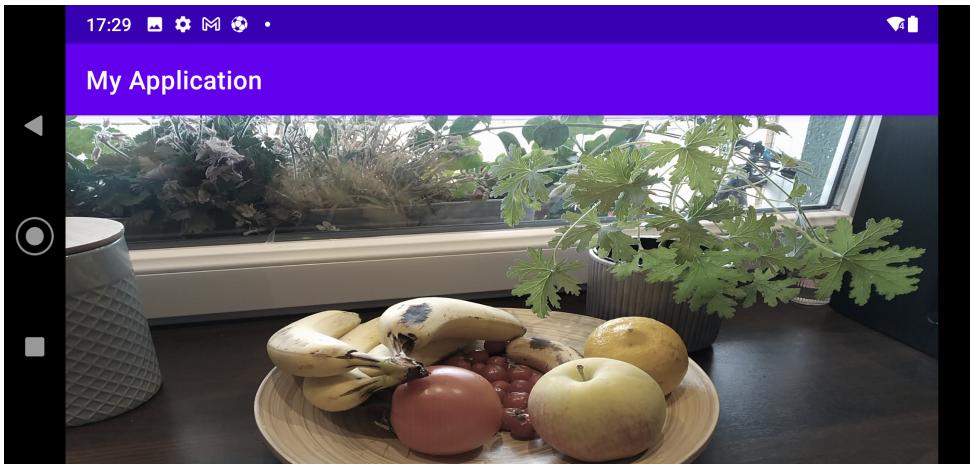
```

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



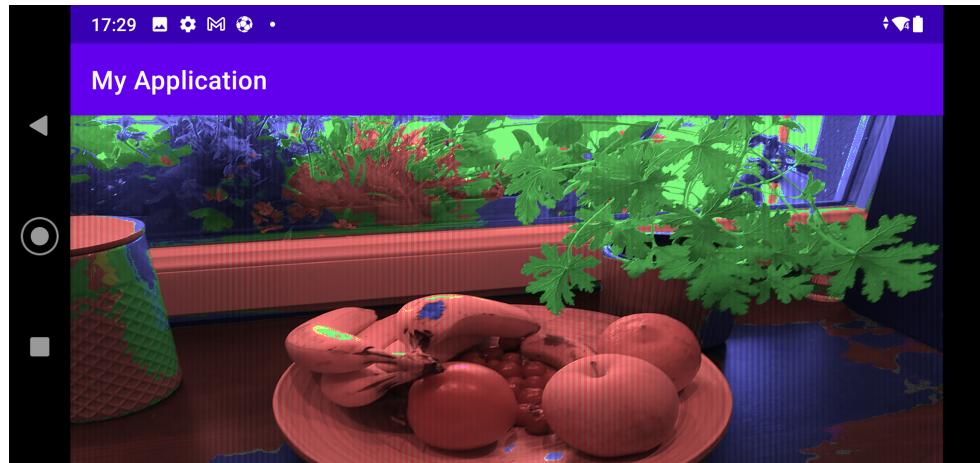
Rys. 4.1. Podgląd Kamery

4.6.2. *Filtr Max Rgb*

Filtr Max RGB pozwala zwizualizować, który kanał posiada największą wartość dla danego piksela. By wykorzystać go przy wyświetaniu 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*

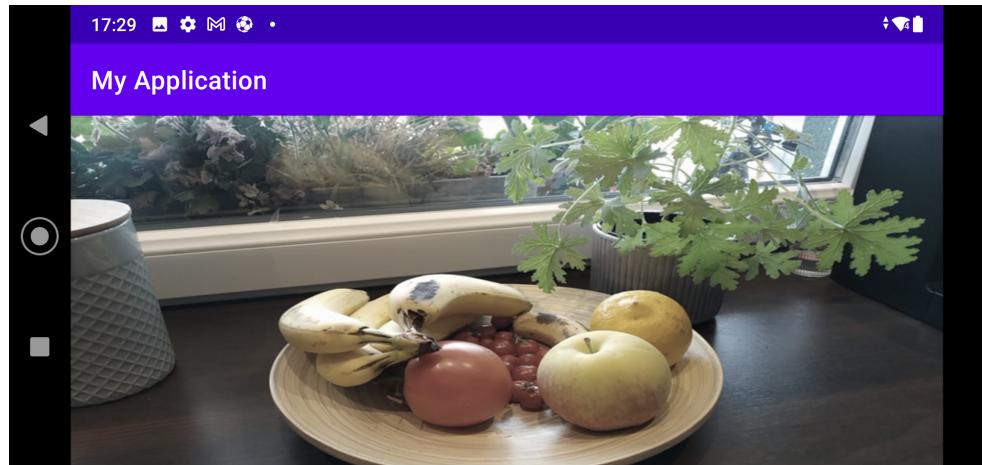
Filtr Uśredniający polega na wyliczeniu wartości piksela na podstawie uśrednionej wartości pikseli sąsiadujących. Dzięki temu filtrowi dochodzi zmniejszenia różnic między sąsiednimi pikselami, przez co dochodzi do rozmycia obrazu i zmniejszenia kontrastu. Kernel nakładający filtr uśredniający wygląda 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 pikseli w kwadracie 5x5 i wpisuje tą wartość do obrazu wynikowego. Dla każdego pixela zdjęcia źródłowego zostaje wyliczona jedna uśredniona wartość. Podgląd kamery z tym filtrem wygląda tak.

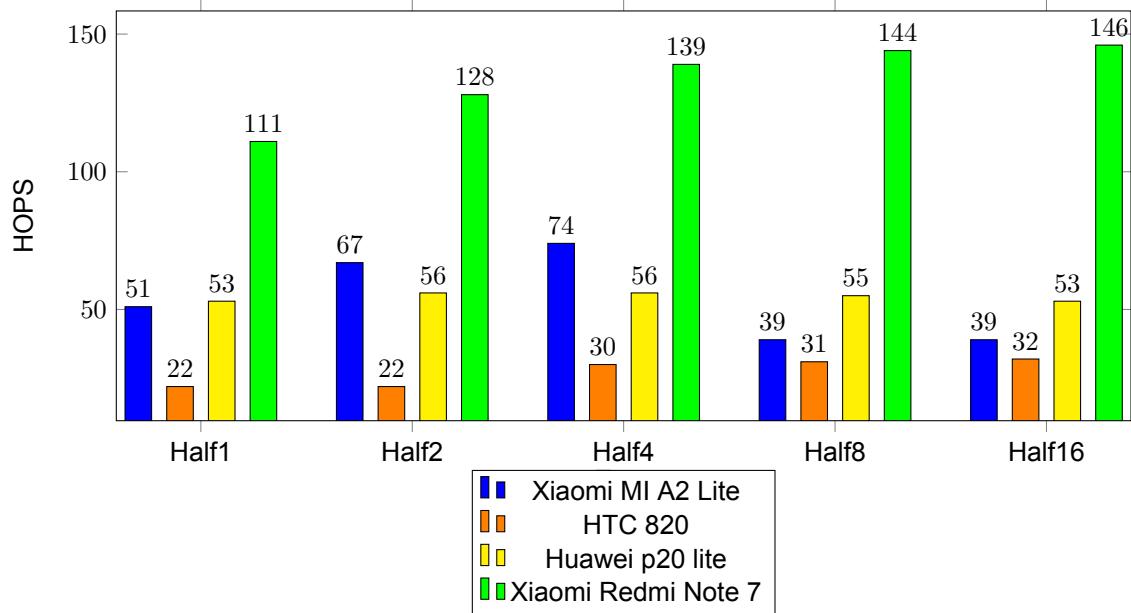


Rys. 4.4. Podgląd z filtrem uśredniającym

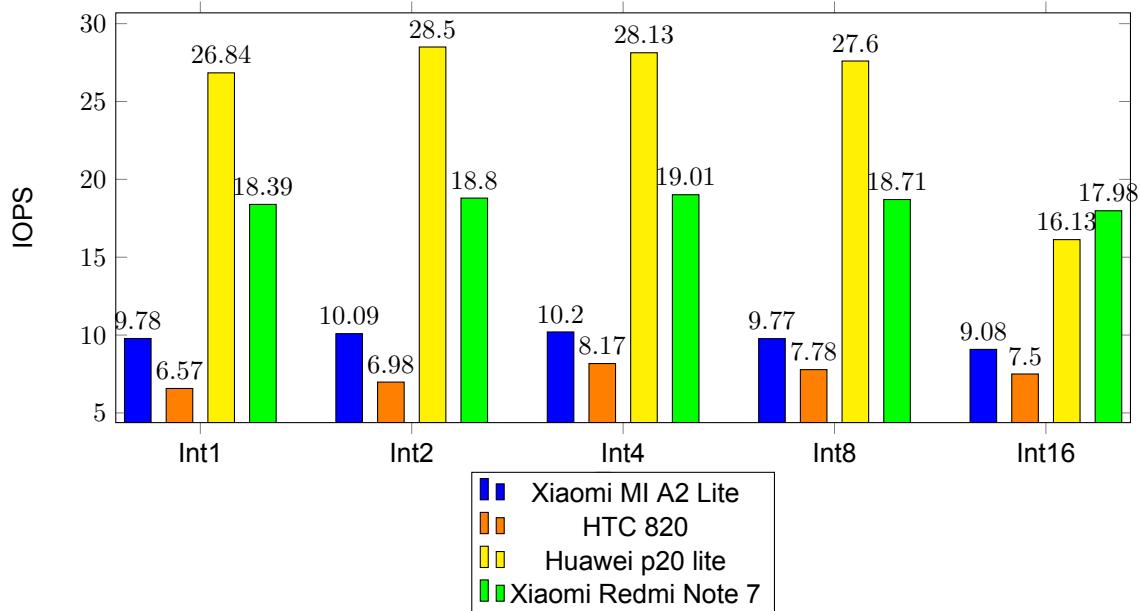
5. ANALIZA WYNIKÓW, WNIOSKI

W tym rozdziale opisane zostały wyniki testów opisanych w rozdziale 4, na urządzeniach opisanych w rozdziale 3.

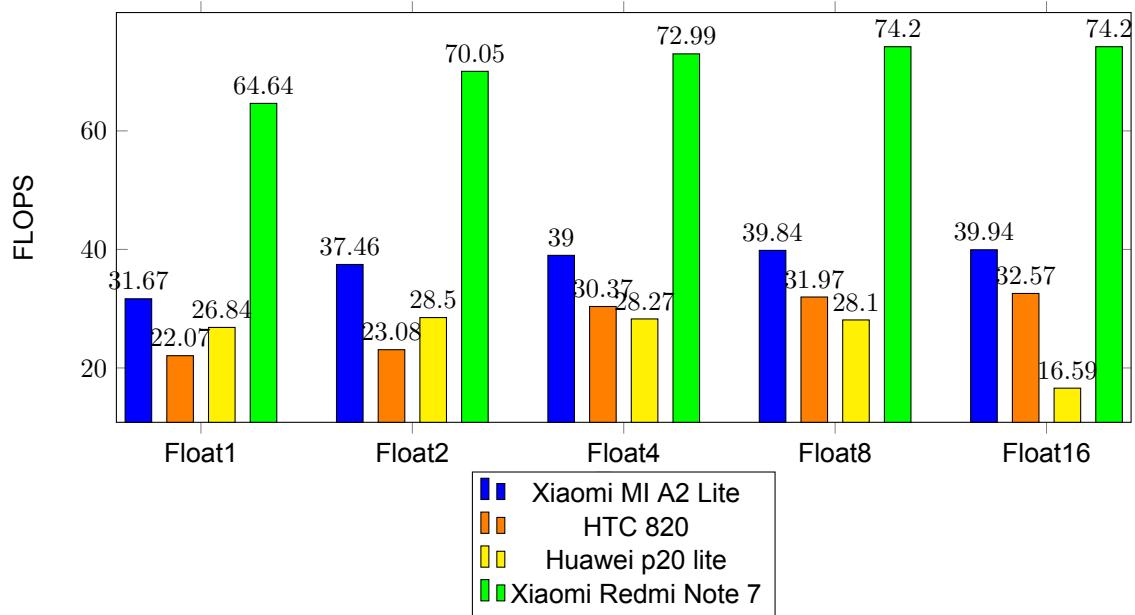
5.1. Wyniki Mocy Obliczeniowej



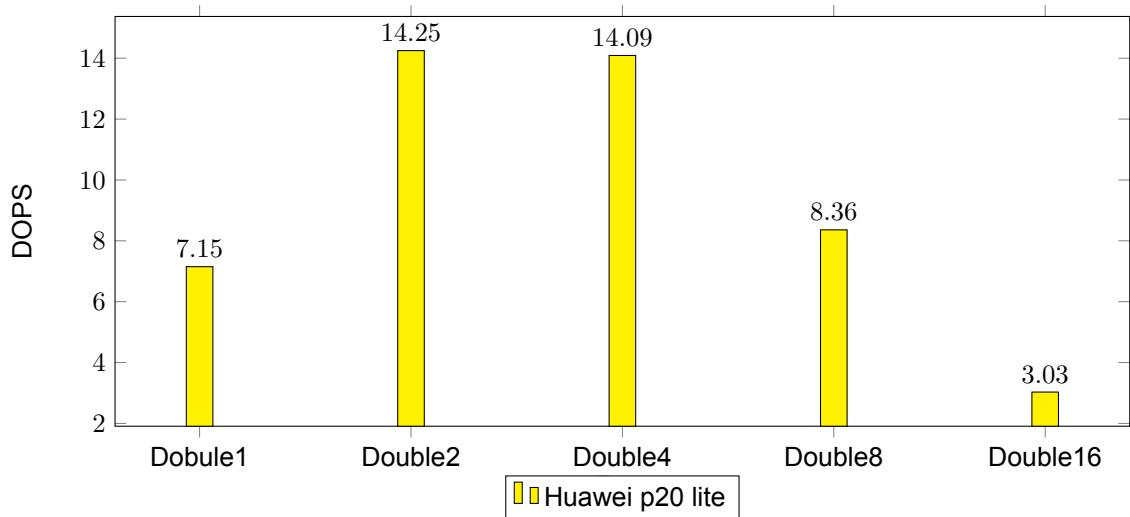
Rys. 5.1. Typ Half Moc Obliczeniowa



Rys. 5.2. Typ Int Moc Obliczeniowa

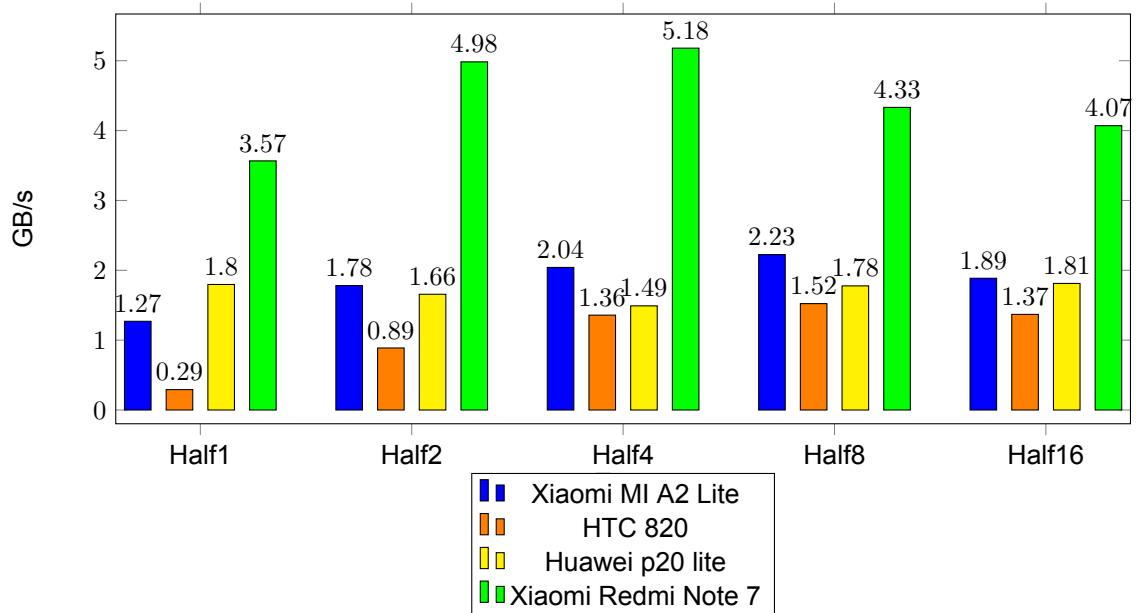


Rys. 5.3. Typ Float Moc Obliczeniowa

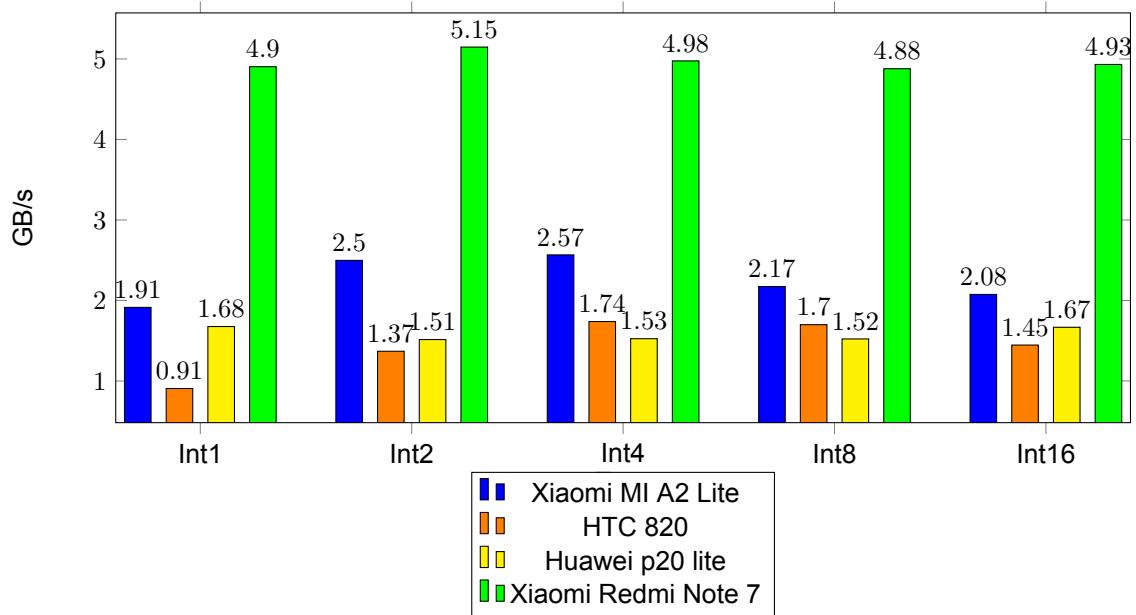


Rys. 5.4. Typ Double Moc Obliczeniowa

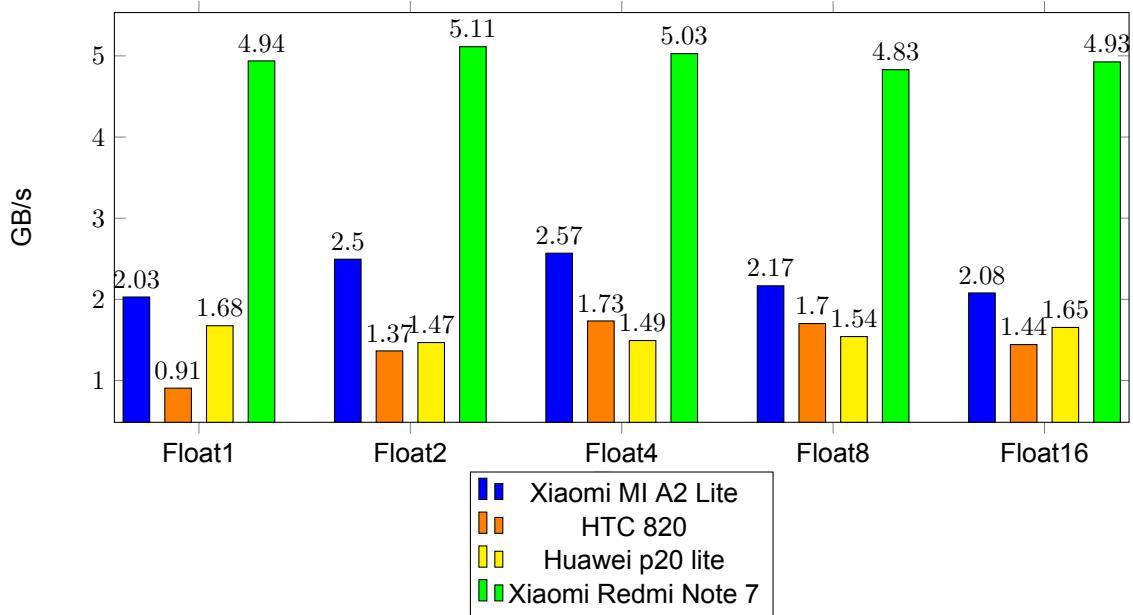
5.2. Wyniki Przepływu Pamięci



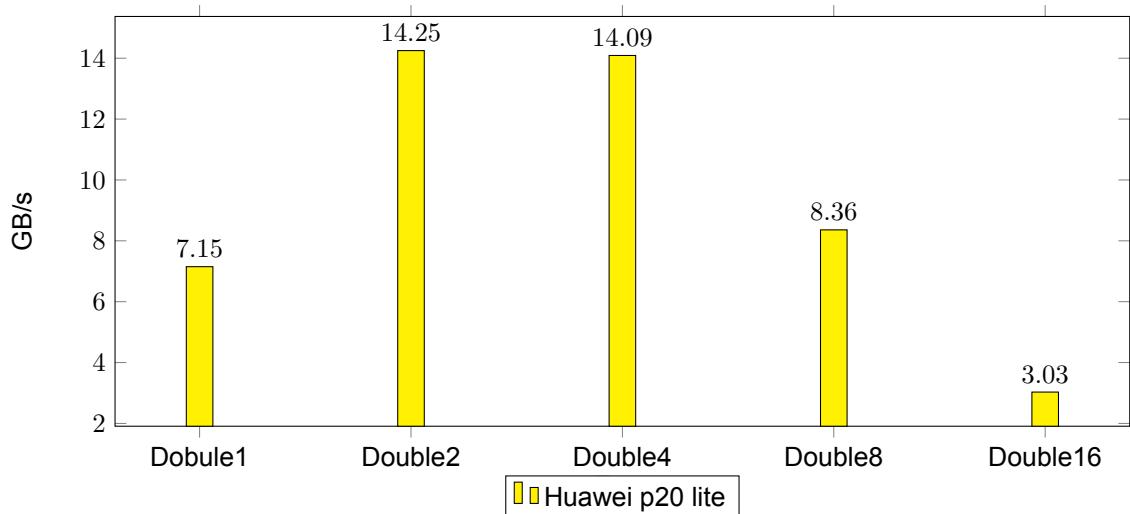
Rys. 5.5. Typ Half Przepływy Pamięci



Rys. 5.6. Typ Int Przepływy Pamięci

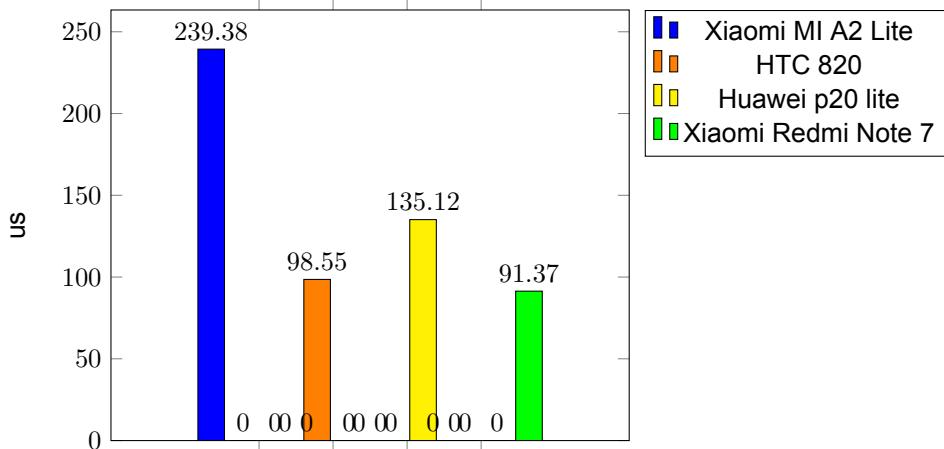


Rys. 5.7. Typ Float Przepływ Pamięci



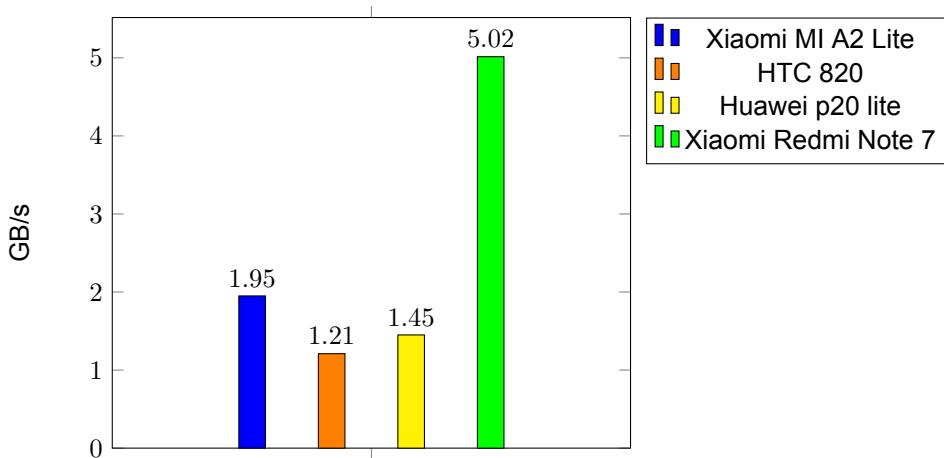
Rys. 5.8. Typ Double Przepływ Pamięci

5.3. Czas Oczekiwania na wykonanie

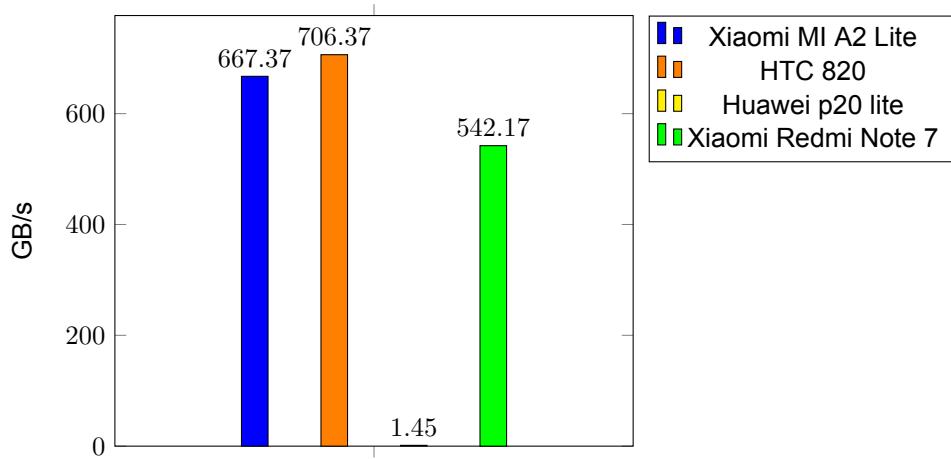


Rys. 5.9. Czas oczekiwania na wykonanie

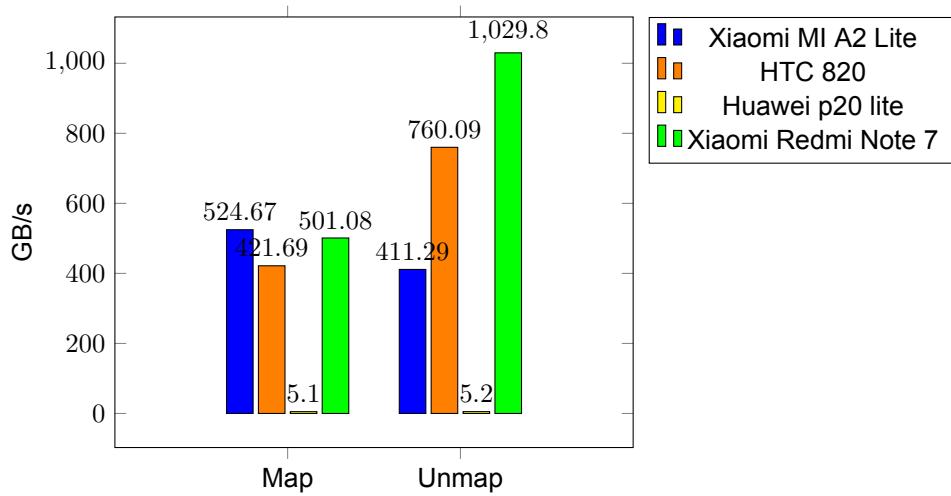
5.4. Transfery Pamięci Aplikacja-Urządzenie



Rys. 5.10. czas wykonywania `clEnqueueReadBuffer`

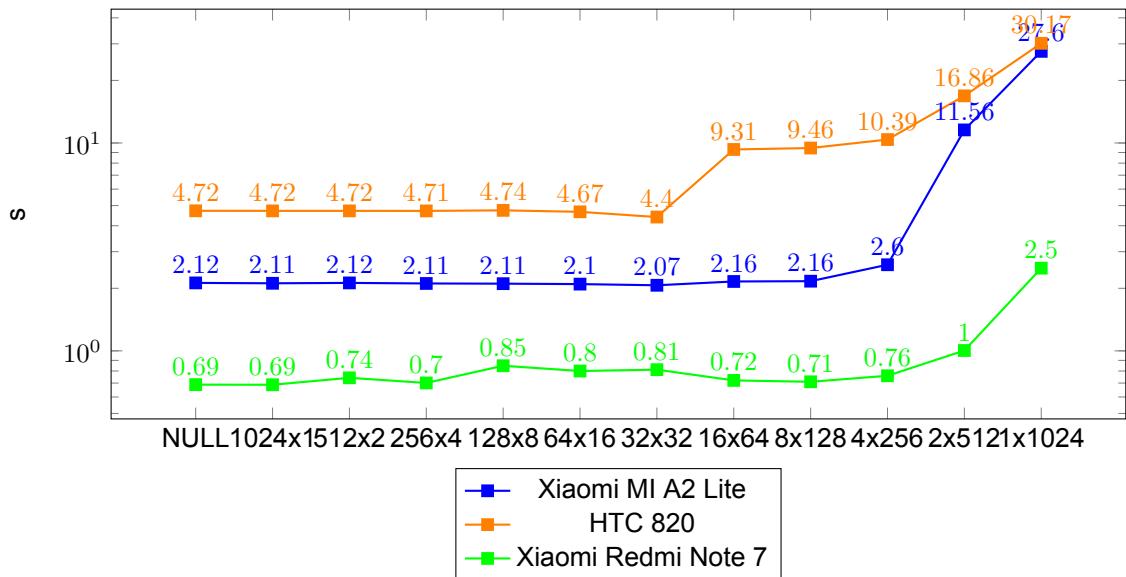


Rys. 5.11. czas wykonywania `clEnqueueWriteBuffer`

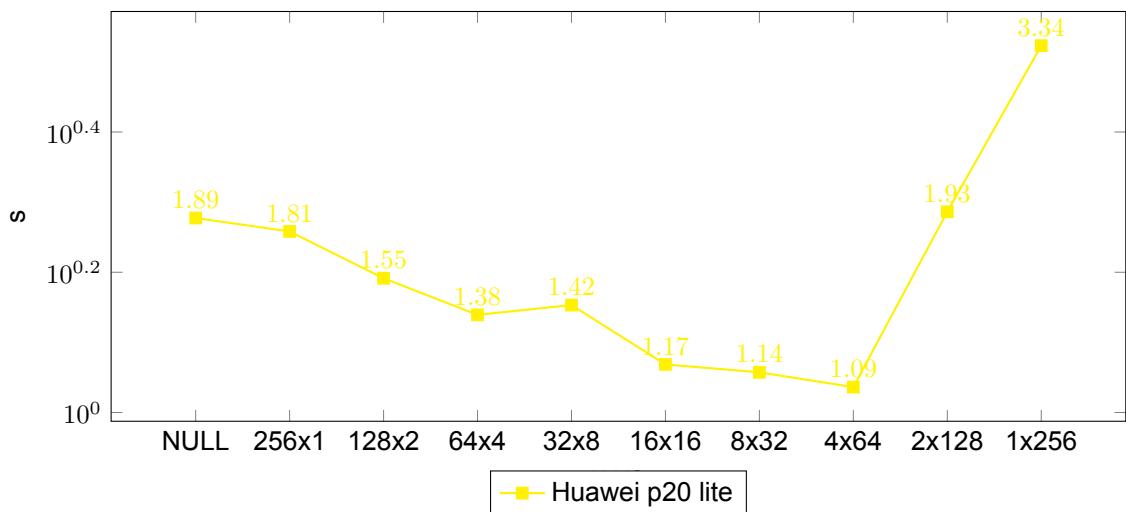


Rys. 5.12. czas wykonywania `Map` `unMap`

5.5. Mnożenie Macierzy

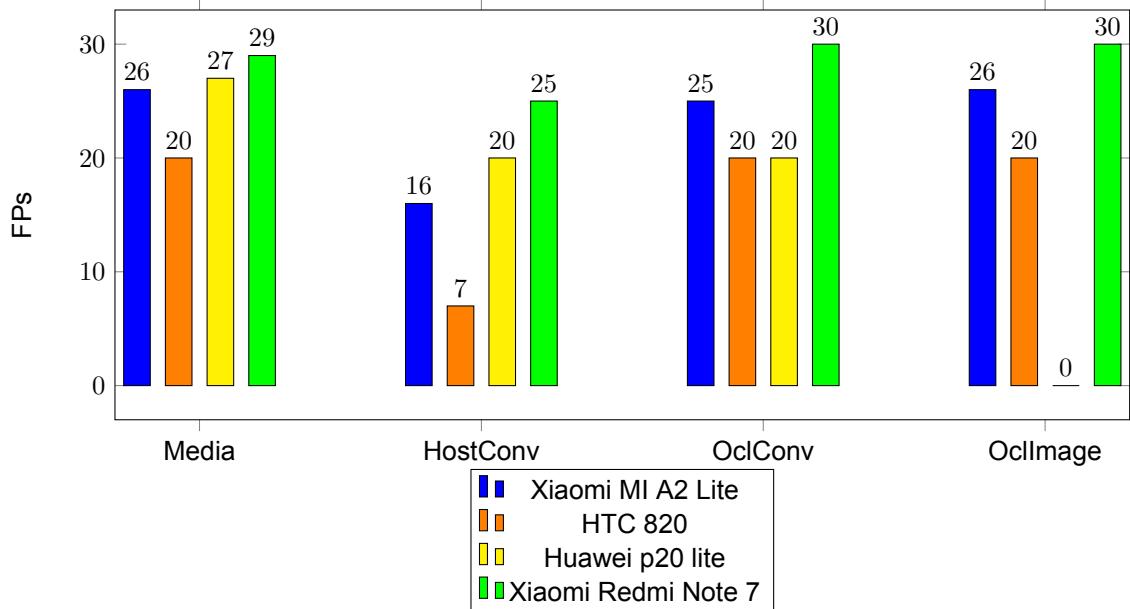


Rys. 5.13. Mnożenie macierzy(Max Lws 1024)

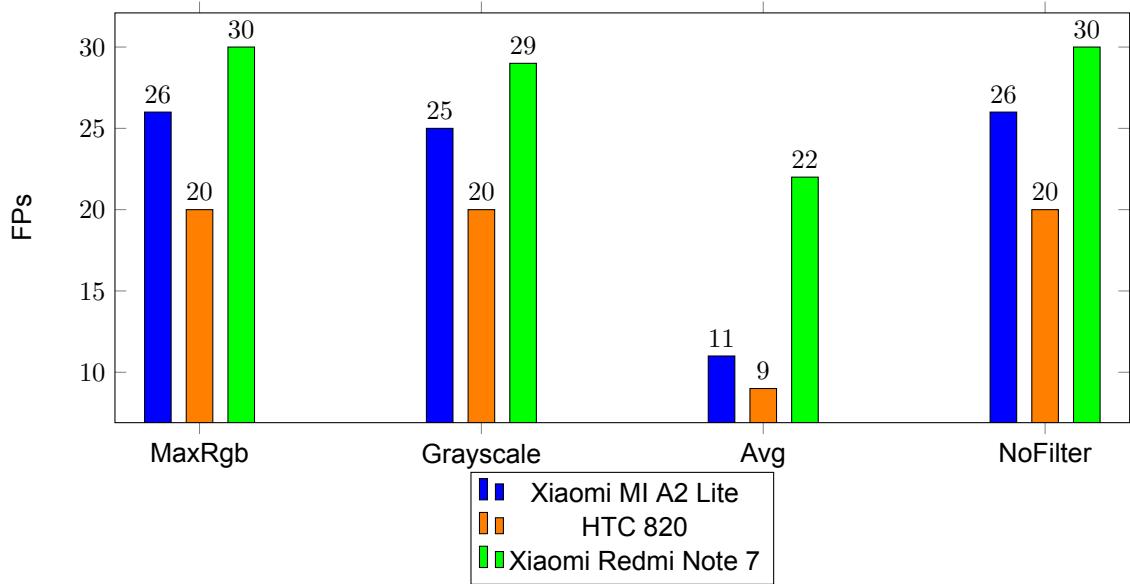


Rys. 5.14. Mnożenie macierzy(Max Lws 256)

5.6. OpenCL z Kamera Api



Rys. 5.15. Konwersja do RGB



Rys. 5.16. Konwersja do RGB

6. APLIKACJE WYKORZYSTUJĄCE OPENCL PRZYPADKI UŻYCIA

WYKAZ RYSUNKÓW

4.1.	Podgląd Kamery	24
4.2.	Filtr Max Rgb	25
4.3.	Podgląd w skali szarości.....	26
4.4.	Podgląd z filtrem uśredniającym.....	27
5.1.	Typ Half Moc Obliczeniowa	28
5.2.	Typ Int Moc Obliczeniowa.....	28
5.3.	Typ Float Moc Obliczeniowa	29
5.4.	Typ Double Moc Obliczeniowa	29
5.5.	Typ Half Przepływ Pamięci	30
5.6.	Typ Int Przepływ Pamięci.....	30
5.7.	Typ Float Przepływ Pamięci.....	31
5.8.	Typ Double Przepływ Pamięci	31
5.9.	Czas oczekiwania na wykonanie	32
5.10.	czas wykonywania clEnqueueReadBuffer.....	32
5.11.	czas wykonywania clEnqueueWriteBuffer	33
5.12.	czas wykonywania Map unMap	33
5.13.	Mnożenie macierzy(Max Lws 1024)	34
5.14.	Mnożenie macierzy(Max Lws 256)	34
5.15.	Konwersja do RGB	35
5.16.	Konwersja do RGB	35

WYKAZ TABEL

Dodatek A: PRZYKŁADOWY DODATEK

A.1 *Sekcja*