



WYDZIAŁ ELEKTRONIKI,  
TELEKOMUNIKACJI  
I INFORMATYKI

Imię i nazwisko studenta: Maciej Plewka

Nr albumu: 155170

Poziom kształcenia: Studia drugiego stopnia

Forma studiów: stacjonarne

Kierunek studiów: Informatyka

Specjalność: Technologie geoinformatyczne i mobilne

## PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Analiza wydajności środowiska OpenCL na platformach mobilnych

Tytuł pracy w języku angielskim: Performance Analysis of the OpenCL Environment on Mobile Platforms

Opiekun pracy: dr inż. Przemysław Falkowski-Gilski

Data ostatecznego zatwierdzenia raportu podobieństw w JSA:

## OŚWIADCZENIE dotyczące pracy dyplomowej zatytułowanej: Analiza wydajności środowiska OpenCL na platformach mobilnych

Imię i nazwisko studenta: Maciej Plewka

Data i miejsce urodzenia: 28.11.1995, Elbląg

Nr albumu: 155170

Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki

Kierunek: informatyka

Poziom kształcenia: drugi

Forma studiów: stacjonarne

Typ pracy: praca dyplomowa magisterska

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2019 r. poz. 1231, z późn. zm.) i konsekwencji dyscyplinarnych określonych w ustawie z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (t.j. Dz. U. z 2020 r. poz. 85, z późn. zm.),<sup>1</sup> a także odpowiedzialności cywilnoprawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

05.09.2021, Maciej Plewka

Data i podpis lub uwierzytelnienie w portalu uczelnianym Moja PG

*\*) Dokument został sporządzony w systemie teleinformatycznym, na podstawie §15 ust. 3b Rozporządzenia MNiSW z dnia 12 maja 2020 r. zmieniającego rozporządzenie w sprawie studiów (Dz.U. z 2020 r. poz. 853). Nie wymaga podpisu ani stempla.*

<sup>1</sup> Ustawa z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce:

Art. 312. ust. 3. W przypadku podejrzenia popełnienia przez studenta czynu, o którym mowa w art. 287 ust. 2 pkt 1–5, rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.

Art. 312. ust. 4. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 5, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o podejrzeniu popełnienia przestępstwa.

## **STRESZCZENIE**

Dzisiejsze inteligentne telefony posiadają coraz więcej cech, które dotychczas przypisywane były wyłącznie komputerom osobistym. Urządzenia te budowane są z coraz lepszych i bardziej wydajnych komponentów. Wszystko wskazuje na to, że nowoczesne aparaty komórkowe zastąpią zwykłe komputery. Do zadań takich jak obróbka obrazu, rozpoznawania mowy czy detekcji obiektów, potrzebna jest duża moc obliczeniowa. W tej pracy przeanalizowano jaką wydajność na urządzeniach mobilnych osiąga środowisko OpenCL, które jest biblioteką dedykowaną do szybkich równoległych obliczeń. W niniejszym dokumencie sprawdzono w jaki sposób inteligentne telefony mogą dostępuwać biblioteki, która jak się okazało nie jest oficjalnie wspierana na platformie Android oraz pokróćce opisano badaną biblioteki. W ramach pracy przetestowano to api, w kontekście osiąganej mocy obliczeniowej, prędkości przepływu pamięci, szybkości rozmnazania macierzy oraz możliwości obróbki obrazu z kamery w czasie rzeczywistym. Uzyskane wyniki zostały przedstawione w formacie wykresów, opisane i przeanalizowane. Omówione zostały aplikacje, które wykorzystują to api między innymi do uczenia głębokich sieci neuronowych czy obróbki obrazu.

## **ABSTRACT**

Today's smartphones have more and more features that so far were only assigned to personal computers. These devices are built of better and more efficient components. Everything indicates that modern phones will replace ordinary computers. High computing power is required for tasks such as image processing, speech recognition and object detection. This paper analyzes the performance of the OpenCL environment on mobile devices, which is a library dedicated to high-speed parallel computing. This document examines how smartphones can access library that, as it turned out, is not officially supported on the Android platform, and briefly describes the analyzed library. As a part of the study, this api was tested in the context of the achieved computing power, memory flow rate, speed of matrix multiplication and the possibility of processing the image from the camera in real time. The obtained results were presented in graph format, described and analyzed. Applications that use this api for teaching deep neural networks or image processing were also discussed.

## **SPIS TREŚCI**

Wykaz ważniejszych oznaczeń i skrótów .....	7
1. Wprowadzenie .....	8
1.1. Cel pracy.....	8
1.2. Zakres 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 .....	15
2.5. Współdzielenie obiektów pomiędzy OpenCL i OpenGL .....	16
3. Specyfika testowanych urządzeń.....	17
3.1. Porównanie graficznych procesorów mobilnych.....	17
3.2. Urządzenia wykorzystane do testowania.....	18
3.2.1. Xiaomi Mi A2 Lite .....	18
3.2.2. Huawei P20 Lite .....	18
3.2.3. HTC Desire 820.....	18
3.2.4. Xiaomi Redmi Note 7 .....	19
3.2.5. Samsung Galaxy A70 .....	19
3.2.6. Porównanie .....	19
4. Testy wydajnościowe OpenCL na urządzeniach z systemem Android .....	21
4.1. Pomiar mocy obliczeniowej .....	21
4.2. Przepływ pamięci .....	22
4.3. Czas oczekiwania na wykonanie kernela .....	23
4.4. Transfer pamięci wbudowanymi funkcjami .....	23
4.5. Mnożenie macierzy .....	24
4.6. OpenCL do filtrowania obrazu z cameraApi .....	24
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.....	31
5.1. Wyniki mocy obliczeniowej .....	31
5.2. Wyniki przepływu pamięci.....	34
5.3. Czas oczekiwania na wykonanie .....	36
5.4. Transfery pamięci aplikacja-urządzenie.....	37
5.5. Mnożenie macierzy .....	39
5.6. OpenCL z cameraApi .....	40
6. Aplikacje wykorzystujące OpenCl Przypadki Użycia .....	43

6.1. Tensorflow.....	43
6.2. OpenCV .....	43
6.3. Adobe .....	43
6.4. Testy mierzące wydajność i jakość urządzenia .....	44
7. Podsumowanie .....	45
Wykaz literatury .....	47
Wykaz rysunków.....	48
Wykaz tabel.....	49
Dodatek A. Przykładowy dodatek.....	50

## **WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW**

**PG** – Politechnika Gdańsk

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

**CPU** – z ang. Central Processing Unit - główny procesor urządzenia

## **1. WPROWADZENIE**

W dzisiejszym świecie nowoczesne telefony przejmują coraz więcej zadań komputerów osobistych. Urządzenia mobilne stają się coraz bardziej wszechstronne, a ich specyfikacja techniczna coraz mniej odstaje od układów stosowanych w zwykłych komputerach. Aktualne trendy takie jak telefony ze składanym ekranem, widoki pulpitów komputerowych czy stacje dokujące do telefonów, nastrajają na coraz większe zastępowanie komputerów osobistych na rzecz inteligentnych telefonów. Jedną z przewag jaką aktualnie mają komputery jest możliwość zrównoleglania obliczeń na dyskretnych kartach graficznych.

### **1.1. Cel pracy**

Jednym z api, które ma możliwość wykonywania obliczeń na procesorach graficznych jest OpenCL. Celem tej pracy jest przeanalizowanie działania tej biblioteki na urządzeniach mobilnych. OpenCL jest api niedostępnym na systemie macOS, natomiast na Androidzie nie posiada oficjalnego wsparcia. Pomimo tego producenci procesorów graficznych, używanych w telefonach z systemem Android, dostarczają sterownika z implementacją tej biblioteki. W niniejszej pracy urządzenia z systemem Android przetestowane zostaną pod względem mocy obliczeniowej, czasu wykonywania operacji matematycznych i przetwarzaniu obrazu.

### **1.2. Zakres pracy**

W ramach niniejszej pracy w drugim rozdziale opisano potrzebne kroki do połączenia aplikacji na Android z biblioteką OpenCL. Dalej pokazano przykładowy przebieg prostej aplikacji oraz wskazano, co potrzebne jest do współdzielenia zasobów z api OpenGL.

W rozdziale trzecim rozdziale porównane zostały główne wady i zalety procesorów graficznych dwóch najbardziej popularnych producentów, a następnie opisane zostały urządzenia, które w ramach analizy wykonanej na potrzeby tej pracy, zostały wykorzystane do testowania.

W rozdziale numer cztery opisane były testy, które wykorzystują OpenCL na Androidzie. Pokazane zostały kernele wykorzystane w testach oraz opisano szczegóły dotyczące specyfiki przeprowadzanych testów. Przedstawione w rozdziale testy dotyczą między innymi osiągów mocy obliczeniowej, szybkości transferów pamięci, czasu wykonywania operacji matematycznej takiej jak mnożenie macierzy oraz używanie OpenCL do nakładania filtrów na wyświetlanym obrazie z kamery.

W rozdziale piątym przedstawiono wyniki uzyskane przez urządzenia opisane w rozdziale trzecim podczas wykonywania testów zaprezentowanych w rozdziale czwartym. Uzyskane wyniki przedstawiono za pomocą wykresów a osiągnięte rezultaty zostały opisane i przeanalizowane.

Natępnie w rozdziale numer sześć zaprezentowane zostały programy, które na systemie Android korzystają z biblioteki OpenCL.

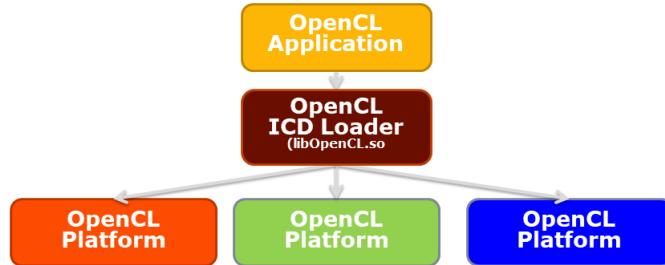
## **2. WSTĘP DO API OPENCL**

OpenCL (ang. Open Compute Language) to standard tworzony obecnie przez grupę Khronos, służący do pisania programów, które mogą zostać wykonane na różnych platformach takich jak CPU (ang. Central Processing Unit), GPU (ang. Graphics Processing Unit) czy FPGA (ang. Field-Programmable Gate Array). 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, takich 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, możliwe jest stworzenie kodu, który można 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 NVidie. 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`, czyli definiujące api do współdzielenia (tzw. 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` 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++. Na urządzeniach z systemem macOS środowisko te nie jest wspierane.

### **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 ładowanej biblioteki aplikacja może używać każdej dostępnej platformy wspierającej to api oraz nie jest na sztywno połączona z jednym sterownikiem w określonej wersji. Niestety binarna wersja takiej biblioteki dla systemu Android nie istnieje. Jedną z możliwości jest połączenie natywnej biblioteki ze sterownikiem znajdującym się w urządzeniu. 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 ładowanej, 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,

posiadającej 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.



Rys. 2.1. Linkowanie biblioteki OpenCL [2]

Powyższy 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.

## 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, testując aplikacje na system Android wykorzystywany będzie `CL_DEVICE_TYPE_GPU`, który jako jedyny jest dostępny na testowanych w tej pracy urządzeniach. Posiadając obiekt `device`, wołając **clCreateContext** możemy stworzyć `context` w ramach którego możliwe jest zarządzanie później stworzonymi obiektami na określonych przy tworzeniu `context` urządzeniach. Dalej w przykładowym kodzie stworzona jest kolejka **clCreateCommandQueue** kolejka powstaje w ramach `context` 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 nieskomplikowany 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.

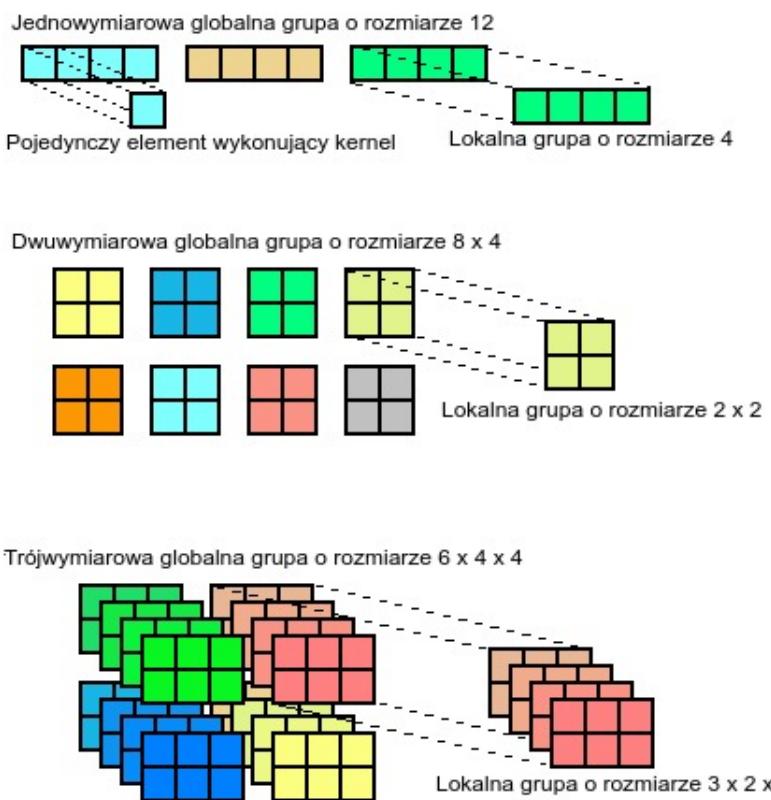
```

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**, umieci ona 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. N wymiarowy podział kernela

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 znajdująć się będą 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(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 Platform Name      QUALCOMM Snapdragon(TM)
10 Number of devices 1
11 Device Name        QUALCOMM Adreno(TM)
12 Device Vendor      QUALCOMM
13 Device Vendor ID   0xbff4d3c4b
14 Device Version     OpenCL 2.0 Adreno(TM)
15 Device Version     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 Device OpenCL C Version 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 funkcje

`clCreateUserEvent`. Obiekt taki może zostać przekazany jako argument funkcji, która zostanie wykonana na urządzeniu, np. `clEnqueueNDRangeKernel` czy `clWnqueueWriteBuffer`. 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 z momentu wysłania komendy 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 (z ang. 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 jej w kernelu OpenCL. Nie wszystkie urządzenia wspierają współdzielenie zasobów. Jeśli w liście rozszerzeń znajduje się `cl_khr_gl_sharing`, możliwe wtedy jest użycie z tego rozwiązania. Chcąc skorzystać tej możliwości, kontekst w OpenCL musi zostać stworzony w oparciu o wcześniej stworzony kontekst OpenGL. Przy tworzeniu kontekstu 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      };
```

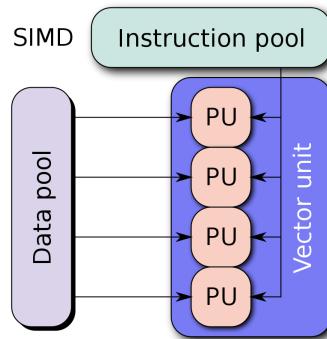
Posiadając tak stworzony kontekst, możemy stworzyć obiekt obrazka w OpenCL w oparciu o stworzoną teksturę w OpenGL.

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

W podanym wyżej kodzie, `textureId` to identyfikator tekstury, którą chcemy współdzielić z OpenGL. Przed wykorzystaniem współdzielonego obrazka należy zwołać `clEnqueueAcquireGLObjects`, wskazując kolejkę, która będzie używać obrazka. Jest to wykonywane w celu wywłaszczenia obiektu przez OpenCL. Po zakończeniu operacji należy zwolnić obiekt polecienniem `clEnqueueReleaseGLObjects`.

### 3. SPECYFIKA TESTOWANYCH URZĄDZEŃ

Ze względu na ograniczenia związane ze zużyciem energii, urządzenia mobilne są nieczęsto 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 operację, na przykład na kilku pikselach z obrazka.



Rys. 3.1. SIMD [19]

#### 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, które odróżniają wymienione produkty

- **Wydajność:** Urządzenia Adreno w większości przypadków osiągają lepsze wyniki wydajnościowe. Dla porównywalnych modeli Adreno 660 i Mali-G78 MP24 ten pierwszy osiąga 1486 GFLOPS, a drugi tylko 1076 GFLOPS.
- **Wsparcie api:** Starsze urządzenia Adreno posiadają wsparcie dla większej ilości api w nowszych wersjach. Najnowsze modele procesorów Mali wspierają podobną listę api.
- **Częstotliwość zegara:** W większości przypadków procesory Mali działają z wyższymi częstotliwościami zegara od konkurentów. Procesor Mali G51 pracuje z częstotliwością 1 GHz. Pośród produktów Adreno żaden nie osiąga takiej wartości.
- **Cena:** Qualcom jest firmą z USA, która narzuca większe koszty licencyjne swoich układów dla producentów telefonów. Sprawia to, że układy Mali częściej znajdują się w tańszych urządzeniach.
- **Renderowanie:** w tej kategorii układy Mali mają dużą przewagę nad Adreno. Przykładowo procesor Adreno 660 jest w stanie renderować 1524 milionów trójkątów na sekundę, a konkurencyjny Mali G78 MP24 może ich wyprowadzić 2463 w tym samym czasie.
- **Problem z odprowadzaniem ciepła:** Jako że procesory Mali działają z wyższym taktowaniem zegara, często dochodzi do przegrzania układów. Dzięki niższej temperaturze procesory Adreno mogą być bardziej wydajne.

**Tabela 3.1.** Bezpośrednie porównanie wad i zalet procesorów Mali i Adreno

Cecha	Adreno	Mali
Wydajność	+	-
Cena	-	+
Wsparcie Api	+	-
Częstotliwość zegara	-	+
Renderowanie	-	+
Przegrzewanie	+	-

### **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 2,0 GHz. Pamięć systemowa urządzenia to 3 GB. Umieszczono w nim procesor graficzny Adreno 506. Posiada on 96 jednostek mogących jednocześnie wykonywać wątki na urządzeniu. Częstotliwość zegara procesora wynosi 650 MHz. Dodatkowo posiada on 128 kB pamięci wbudowanej w chip GPU oraz 8 kB pamięci cache. Wykorzystuje pamięć typu LPDDR3 933 MHz i przepustowości 7,4 GB/s. 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 13.11.2019 r. Posiada wsparcie dla typu zmiennoprzecinkowego half oraz dla współdzielenia zasobów z OpenGL. Natomiast nie posiada wsparcia dla typów zmiennoprzecinkowych podwójnej precyzji double.

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

Jest to urządzenie z marca 2018 roku wyposażone w ośmiordzeniowy procesor 4x2,36 GHz Cortex-A53 + 4x1,7 GHz Cortex-A53. Posiada on 4 GB pamięci systemowej. Wbudowany procesor graficzny to Mali-T830 MP2. Posiada on dwie jednostki wykonawcze, a każda może wykonywać 32 wątki. Posiada 16 kB pamięci cache, a częstotliwość zegara wynosi 900 MHz. Urządzenie korzysta z pamięci typu LPDDR3 o częstotliwości 933 MHz i przepustowości do 14,9 GB/s. Wspiera następujące api: Vulkan 1.0, OpenCL 1.2, OpenGL 3.2. Ma możliwość użycia typów half oraz double, natomiast nie wspiera współdzielenia zasobów z OpenGL. Telefon działa z systemem operacyjnym Android w wersji 9.

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

Telefon z 2014 roku wyposażony jest w procesor ośmiordzeniowy 1,7 GHz quad-core Cortex-A53 + 1 GHz quad-core Cortex-A53. Posiada 2 GB pamięci RAM oraz procesor graficzny Adreno 405. GPU może wykonywać na raz 48 wątków. Dodatkowo posiada 256 kB pamięci dedykowanej. Wykorzystano pamięć typu LPDDR3 z częstotliwością 666,5 MHz i szybkości 5,3 GB/s. Api wspierane przez urządzenia to: OpenGL 3.2, OpenCL 1.2, DX11. Telefon dla Systemu Android w wersji 5 posiada sterownik OpenCL w wersji z 29.04.2015 r. A z androidem 6 w wersji z 23.03.2016 r. Wspiera typ half, nie wspiera typu double. Możliwe do wykorzystania jest rozszerzenie współdzielenia zasobów między OpenCL i OpenGL.

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

Telefon wydany został na początku 2019 roku. Posiada procesor ośmiodzienny Qualcomm Kryo 260 z częstotliwością do 2,2 GHz. Wyposażony jest w 6 GB pamięci systemowej. Dodatkowo posiada procesor graficzny Adreno 512. GPU działa z częstotliwością 850 MHz i posiada 256 kB pamięci dedykowanej oraz 16 kB cache. Urządzenie korzysta z pamięci typu LPDDR4 o częstotliwości 1866 MHz i przepustowości do 29,8 GB/s. Może wykonywać na raz 128 wątków. Telefon działa z systemem android w wersji 10, posiada sterownik OpenCL w wersji: #9b15012 z 17.09.2020 r. 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 ośmiodzienny Kryo 460 z częstotliwością zegara do 2 GHz. Posiada 6 GB pamięci RAM. Dodatkowo posiada procesor graficzny Adreno 612 o częstotliwości 845 MHz, 128 wątkach, wbudowanej pamięci 256 kB i 16 kB pamięci cache. Jest to pamięć typu LPDDR4X o częstotliwości 1866MHz i przepustowości do 14,9 GB/s. Android w wersji 10 ze sterownikiem OpenCL w wersji #e1ac91e z 17.11.2020 r. Wspiera typ half, nie wspiera typu double. Możliwe do wykorzystania jest 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 i najszybciej pracujący procesor oraz Samsung Galaxy A70, który ma najnowszy procesor graficzny.

**Tabela 3.2.** Porównanie testowanych telefonów

<b>Telefon</b>	<b>Data wydania</b>	<b>CPU</b>	<b>RAM</b>	<b>GPU</b>
Xiaomi Mi A2 Lite	Q3 2018	8x Cortex-A53 do 2 GHz	3 GB	Adreno 506
HTCDesire 820	Q3 2014	1,7 GHz quad-core Cortex-A53 + 1 GHz quad-core Cortex-A53	2 GB	Adreno 405
Huawei P20 Lite	Q2 2018	4x2,36 GHz Cortex-A53 + 4x1,7 GHz Cortex-A53	4 GB	Mali T830 MP2
Samsung Galaxy A70	Q1 2019	8x Kryo 460 do 2 GHz	6 GB	Adreno 612
Xiaomi Redmi Note 7	Q1 2019	8x Qualcomm Kryo 260 do 2,2 GHz	6 GB	Adreno 512

**Tabela 3.3.** Porównanie dostępności api na testowanych urządzeniach

GPU	Vulkan	D3D	OpenCL	OGL	Data komplikacji sterownika OpenCL	Wspierane rozszerzenia
Adreno 506	1.0	DX11	2.0	3.2	13.11.2019 r.	cl_khr_gl_sharing cl_khr_fp16
Adreno 405	N/A	DX11	1.2	3.2	23.03.2016 r.	cl_khr_gl_sharing cl_khr_fp16
Mali T830 2MP	1.0	DX11	1.2	3.2	N/A	cl_khr_fp64 cl_khr_fp16
Adreno 612	1.1	DX12	2.0	3.2	17.11.2020 r.	cl_khr_gl_sharing cl_khr_fp16
Adreno 512	1.0	DX11	2.0	3.2	17.09.2020 r.	cl_khr_gl_sharing cl_khr_fp16

**Tabela 3.4.** Porównanie testowanych procesorów graficznych

GPU	Częstotliwość zegara	Pamięć	Typ pamięci	Ilość jednostek wykonawczych
Adreno 506	650 MHz	128 kB + 8 kB	LPDDR3-1866 933 MHz 7,4 GB/s	96
Adreno 405	550 MHz	256 kB	LPDDR3-1333 665,5 MHz 5,3 GB/s	48
Mali T830 2MP	900 MHz	128 kB	LPDDR3 933 MHz	2 x 32
Adreno 612	845 MHz	256 kB + 16 kB	LPDDR4X-3732 1866 MHz Dual chanel 16 bit 14.9 GB/s	128
Adreno 512	850 MHz	256kB + 16kB	LPDDR4-3732 1866 MHz Quad chanel 16 bit 29.8 GB/s	128

## 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. Przykładowo w 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 __kernel void Float1(__global float *ptr, float _fp)
9 {
10    float fp = (float)_fp;
11    float x = fp;
12    float y = (float)get_local_id(0);
13
14    for(int i=0; i<128; i++)
15    {
16        MAD_16(x, y);
17    }
18
19    ptr[get_global_id(0)] = y;
20}
21
22 __kernel void Float2(__global float *ptr, float _fp)
23 {
24    float fp = (float)_fp;
25    float2 x = (float2)(fp, fp);
26    float2 y = (float2)get_local_id(0);
27
28    for(int i=0; i<64; i++)
29    {
30        MAD_16(x, y);
31    }
32
33    ptr[get_global_id(0)] = (y.s0) + (y.s1);
34}
35
36 __kernel void Float4(__global float *ptr, float _fp)
37 {
38    float fp = (float)_fp;
39    float4 x = (float4)(fp, fp, fp, fp);
40    float4 y = (float4)get_local_id(0);
41
42    for(int i=0; i<32; i++)
43    {
44        MAD_16(x, y);
45    }
46
47    ptr[get_global_id(0)] = (y.s0) + (y.s1) + (y.s2) + (y.s3);
48}
```

```

46
47 --kernel void Float8(_global float *ptr, float _fp)
48 {
49     float fp = (float)_fp;
50     float8 x = (float8)(fp, fp, fp, fp, fp, fp, fp, fp);
51     float8 y = (float8)get_local_id(0);
52
53     for(int i=0; i<16; i++)
54     {
55         MAD_16(x, y);
56     }
57
58     ptr[get_global_id(0)] = (y.S0) + (y.S1) + (y.S2) + (y.S3) + (y.S4)
59     + (y.S5) + (y.S6) + (y.S7);
60 }
61
62 --kernel void Float16(_global float *ptr, float _fp)
63 {
64     float fp = (float)_fp;
65     float16 x = (float16)(fp, fp, fp, fp, fp, fp, fp, fp, fp,
66                           fp, fp, fp, fp, fp);
67     float16 y = (float16)get_local_id(0);
68
69     for(int i=0; i<8; i++)
70     {
71         MAD_16(x, y);
72     }
73
74     ptr[get_global_id(0)] = (y.S0) + (y.S1) + (y.S2) + (y.S3) + (y.S4)
75     + (y.S5) + (y.S6) + (y.S7) + (y.S8) + (y.S9) + (y.SA) + (y.SB)
76     + (y.SC) + (y.SD) + (y.SE) + (y.SF);
77 }
```

Argument kernela \_A to przykładowa, zmiennoprzecinkowa wartość początkowa. W przeprowadzonych testach wielkość 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 int 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 kopiowana 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 bufore 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, także zostaną przetestowane.

#### **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 na 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:

- **clEnqueueWriteBuffer** 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, 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, mogące 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 przemnożyć. Operacja wykonana jest określoną liczbę razy a wartość końcowa jest średnią z tych wykonień. 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, 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, wykorzystana została 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
7      [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
8      mPositionHandle = GLES20.glGetAttribLocation(mProgram , "position");
9      GLES20 glEnableVertexAttribArray(mPositionHandle);
10     GLES20 glVertexAttribPointer(mPositionHandle , COORDS_PER_VERTEX
11         , GLES20.GL_FLOAT , false , vertexStride , vertexBuffer);
12
13     mTextureCoordHandle = GLES20.glGetAttribLocation(mProgram , "inputTextureCoordinate");
14     GLES20 glEnableVertexAttribArray(mTextureCoordHandle);
15     GLES20 glVertexAttribPointer(mTextureCoordHandle ,
16         COORDS_PER_VERTEX , GLES20.GL_FLOAT , false , vertexStride ,
17         textureVerticesBuffer);
18
19     mColorHandle = GLES20.glGetAttribLocation(mProgram , "s_texture"
20         );
21
22     GLES20.glDrawElements(GLES20.GL_TRIANGLES , drawOrder.length ,
23         GLES20.GL_UNSIGNED_SHORT , drawListBuffer);
24
25     // Disable vertex array
26     GLES20.glDisableVertexAttribArray(mPositionHandle);
27     GLES20.glDisableVertexAttribArray(mTextureCoordHandle);
28 }

```

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() {" +
        "gl_FragColor = texture2D( s_texture,
            textureCoordinate );\n" +
    "}";

```

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 przechwytyujemy 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];
15
16                if (Cb < 0) Cb += 127;
17                else Cb -= 128;
18                if (Cr < 0) Cr += 127;
19                else Cr -= 128;
20            }
21
22            R = Y + Cr + (Cr >> 2) + (Cr >> 3) + (Cr >> 5);
23            G = Y - (Cb >> 2) + (Cb >> 4) + (Cb >> 5) - (Cr >> 1) +
24                (Cr >> 3) + (Cr >> 4) + (Cr >> 5);
25            B = Y + Cb + (Cb >> 1) + (Cb >> 2) + (Cb >> 6);
26
27            if (R < 0) R = 0;
28            else if (R > 255) R = 255;
29            if (G < 0) G = 0;
30            else if (G > 255) G = 255;
31
32            if (B < 0) B = 0;
33            else if (B > 255) B = 255;
34
35            index = y * width + x;
36            rgba[index] = R;
37            rgba[index + 1] = G;
38            rgba[index + 2] = B;
39            rgba[index + 3] = 255;
40        }
41    }
42}
```

```

29         if (B < 0) B = 0;
30     else if (B > 255) B = 255;
31     rgba[4 * index + 0] = ((byte) (R));
32     rgba[4 * index + 1] = ((byte) (G));
33     rgba[4 * index + 2] = ((byte) (B));
34     rgba[4 * index + 3] = ((byte) (255));
35     index++;
36   }
37 }
38 }
```

. Po wykonaniu tej funkcji w tablicy `rgb[]` znajduje się obraz w formacie RGBA. W celu zaktualizowania danych w naszej wyświetlonej 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 wyświetlonej 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 bufory, do pierwszego kopujemy dane z kamery, po wykonaniu dane z drugiego bufora kopiowane są tablicy, która później zostanie wpisana do tekstury.

Następnym usprawnieniem jest pisanie do wyświetlonej tekstury bezpośrednio z wykonanego 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.

```

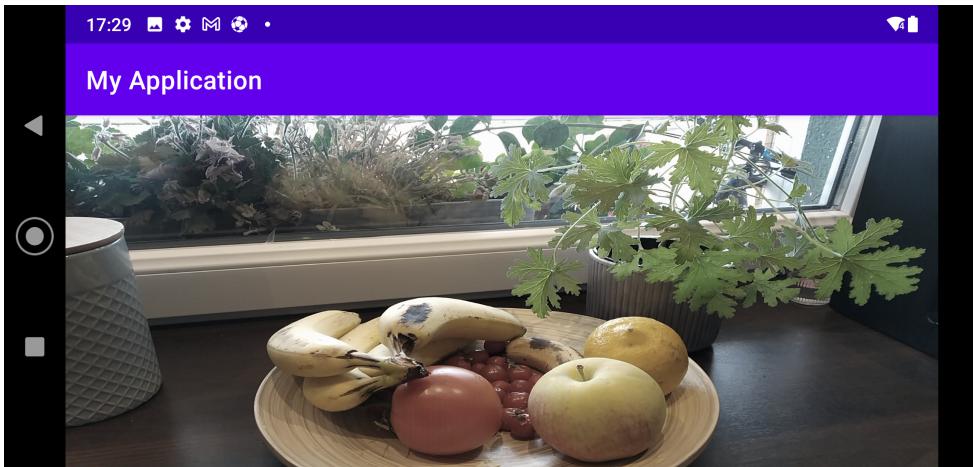
1 __kernel void convertNV21ToRGBImage(__write_only image2d_t img,
2                                     __global char *yuv, int width, int height);
```

By wyliczone wartości pikela zapisać w obiekcie obrazka należy sprowadzić je do postaci zdenormalizowanej 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



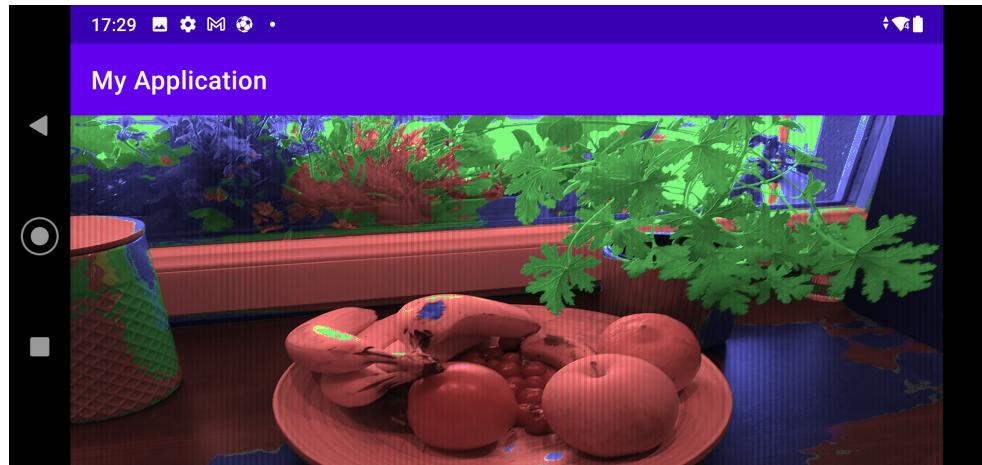
Rys. 4.1. Podgląd kamery

#### 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         if(pixel.x < max)
16             pixel.x = 0.0;
17         if(pixel.y < max)
18             pixel.y = 0.0;
19         if(pixel.z < max)
20             pixel.z = 0.0;
21         write_imagef(dst, coord, pixel);
22 }
```

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 filtru wygląda następująco.



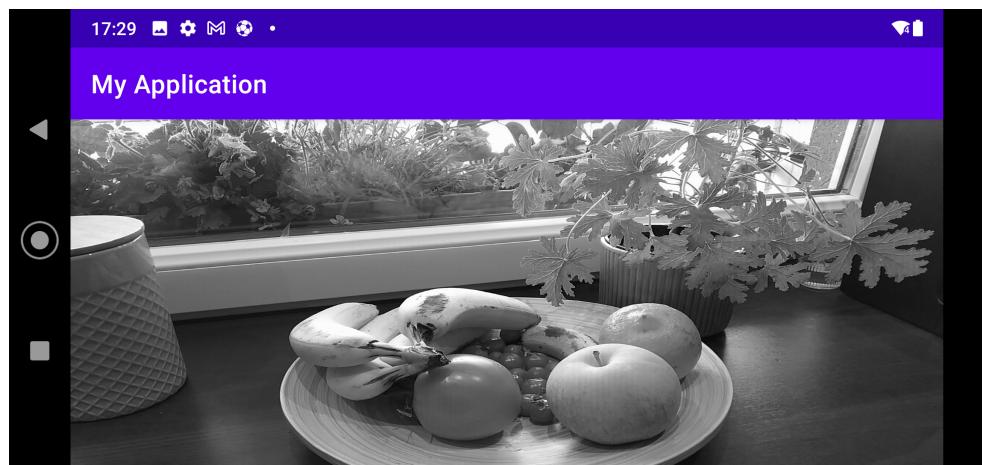
Rys. 4.2. Filtr max rgb

#### 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.



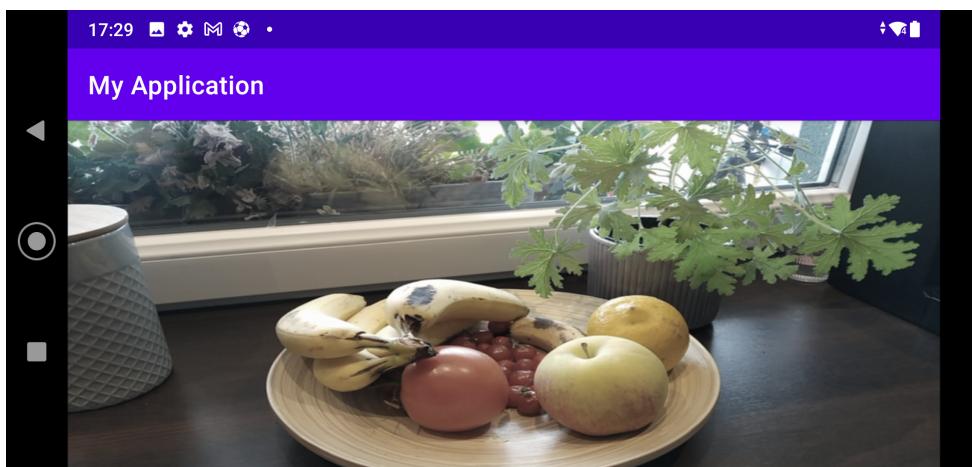
Rys. 4.3. Podgląd w skali szarości

#### 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 5x5 i wpisuje tą wartość do obrazu wynikowego. Dla każdego pixela zdjęcia źródłowego zostaje wyliczona jedna uśredniona wartość. Podgląd kamery z tym filtrem zaprezentowano na rysunku 4.4.



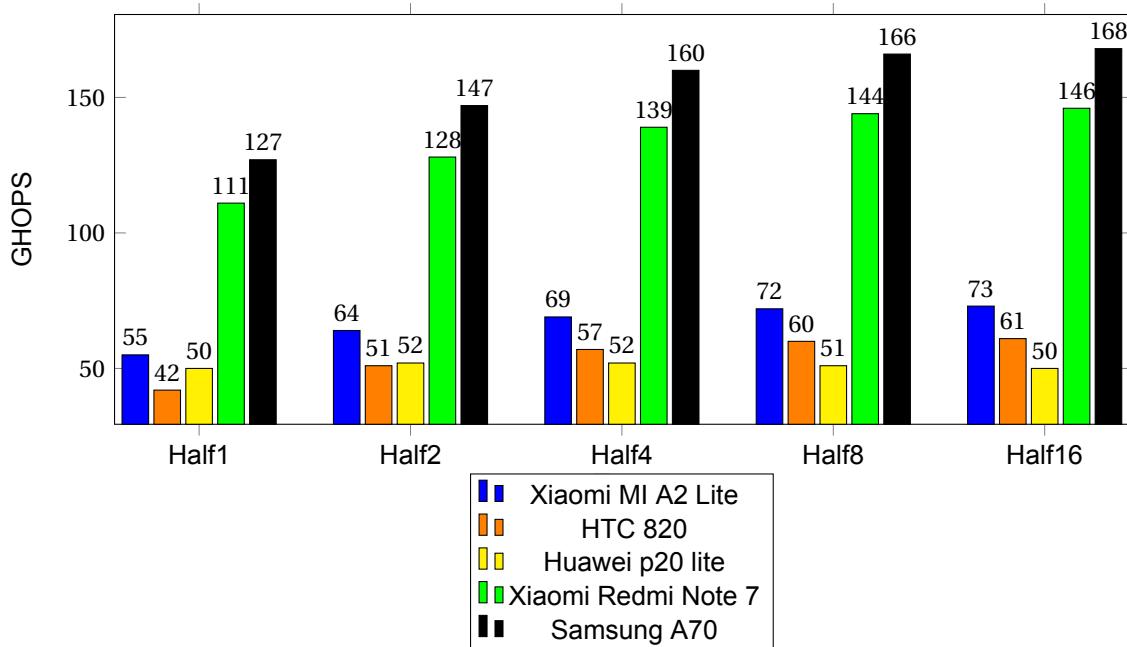
Rys. 4.4. Podgląd z filtrem uśredniającym

## 5. WYNIKI, ANALIZA I WNIOSKI

W tym rozdziale opisane zostały wyniki testów opisanych w rozdziale czwartym, na urządzeniach przedstawionych w rozdziale trzecim. 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.

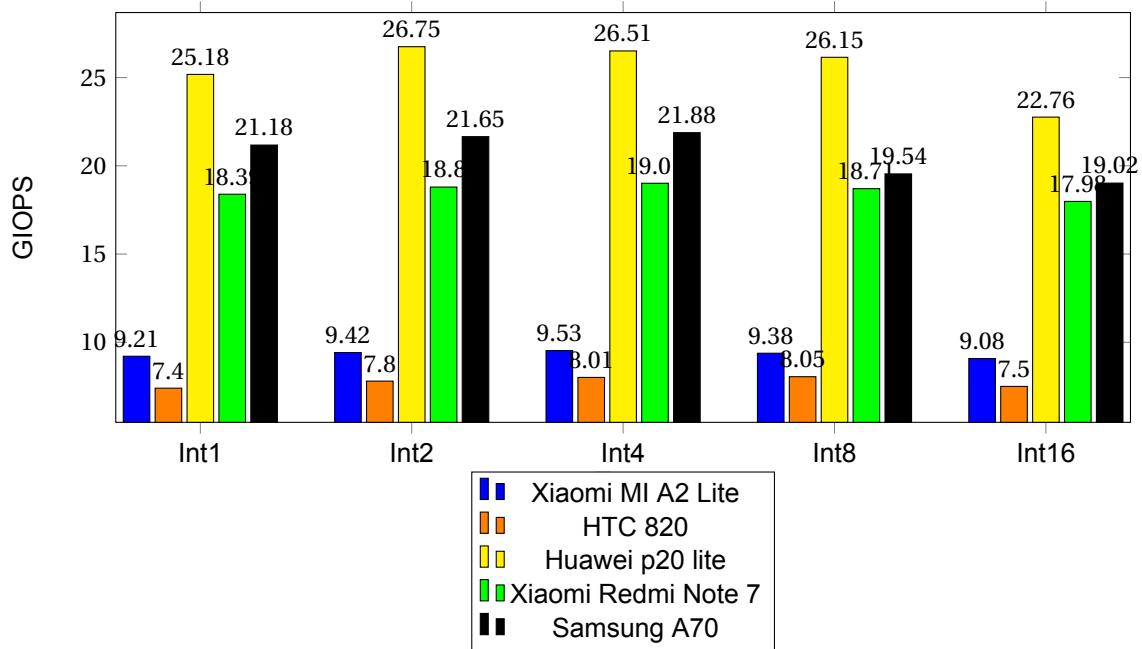
### 5.1. Wyniki mocy obliczeniowej

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



Rys. 5.1. Moc obliczeniowa dla typu half

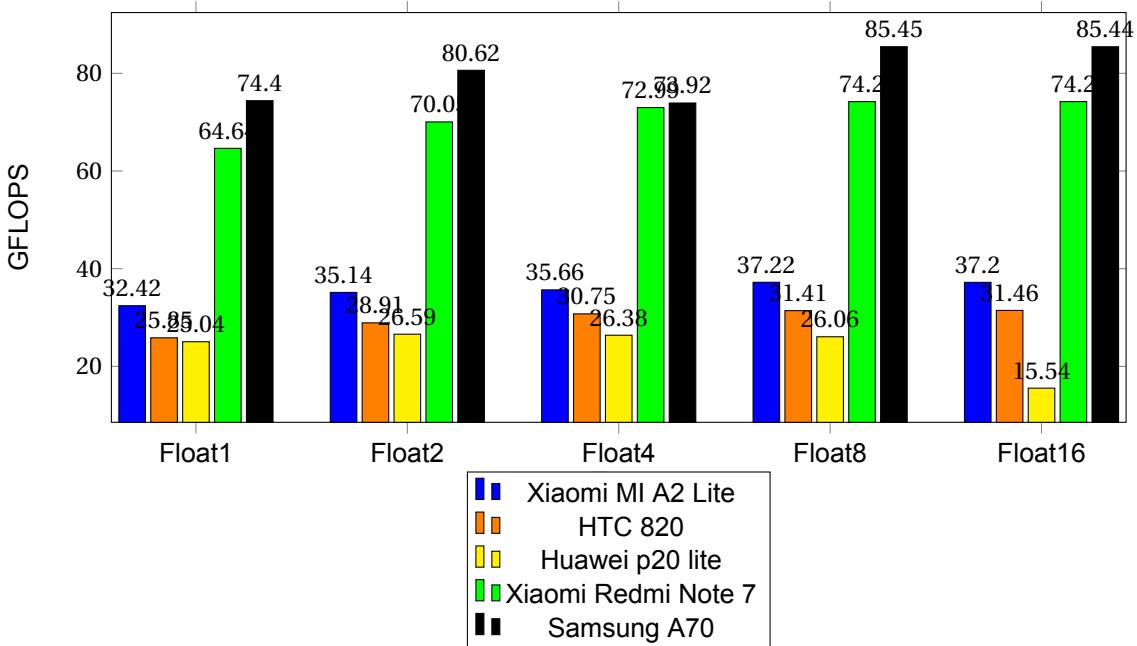
Powyższy wykres prezentuje ilość możliwych do wykonania obliczeń na liczbach typu half na sekundę, wyrażoną w GHOPS (z ang. Giga Half Operations Per Second), dla wektorowych wersji tego typu. Na wykresie widać, że na procesorach Adreno dzięki użyciu wektorowych typów ilość operacji na sekundę rośnie. 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.



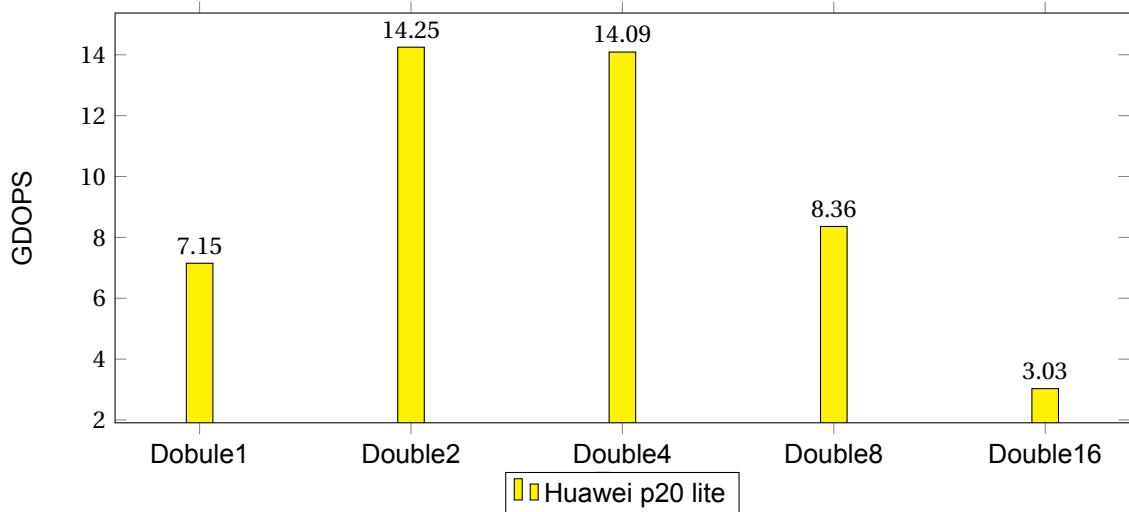
Rys. 5.2. Moc obliczeniowa dla typu int

Na powyższym wykresie zaprezentowano ilość możliwych do wykonania operacji na liczbach typu int na sekundę, wyrażoną w GIOPS(z ang. Giga Integer Operations Per Second), dla wersji wektorowych tego typu. 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.

Na umieszczonej poniżej wykresie zaprezentowano ilość możliwych do wykonania operacji na liczbach typu float na sekundę, wyrażoną w GFLOPS(z ang. Giga Floating-Point Operations Per Second), dla wersji wektorowych tego typu. W testach wydajności, przy operacjach na typie zmienoprzecinkowym pojedynczej precyzji float, zdecydowanie najgorzej wypada procesor Mali T830 2MP. Wydajność tego procesora dla typów wektorowych wypada na podobnym poziomie jak w przypadku typu z pojedynczą wartością. 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.



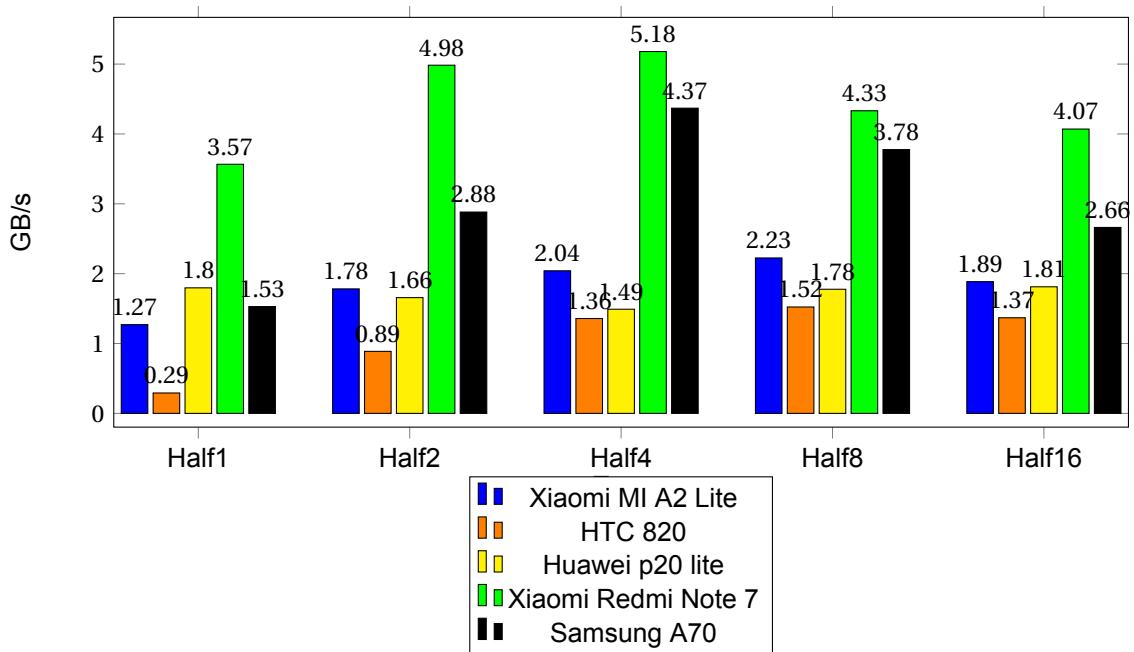
Rys. 5.3. Moc obliczeniowa dla typu float



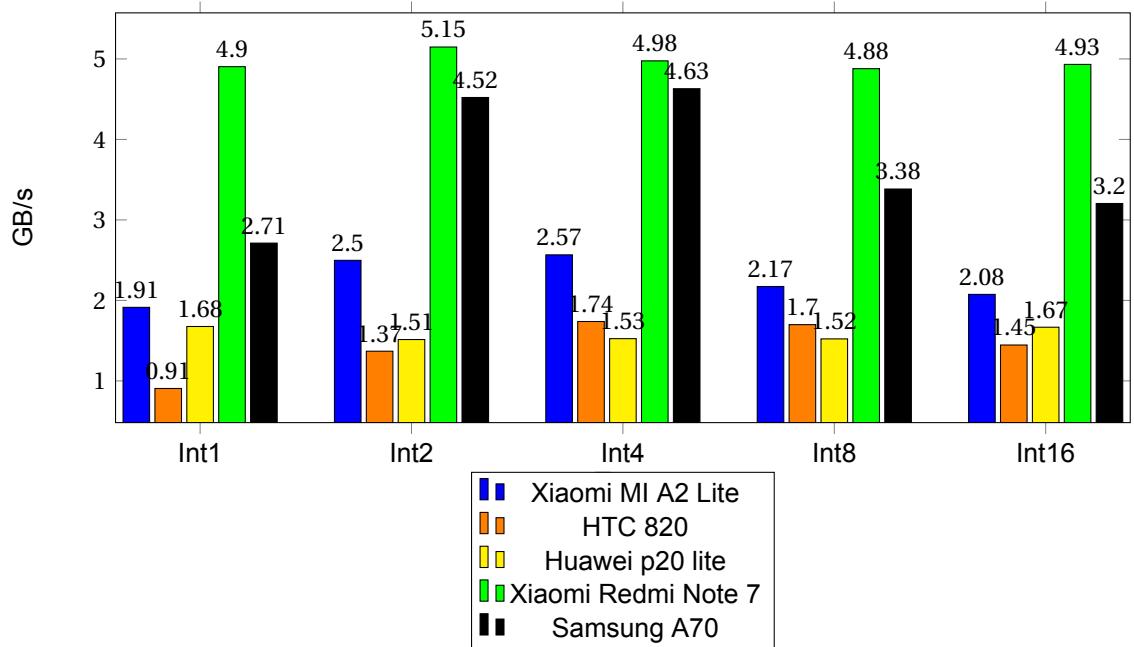
Rys. 5.4. Moc obliczeniowa dla typu double

Powyższy wykres prezentuje ilość możliwych do wykonania obliczeń typu half, na sekundę wyrażoną w GDOPS (z ang. Giga Double Operations Per Second) dla wektorowych wersji tego typu. Huawei P20 Lite jako jedyne testowane urządzenie posiada obsługę typów zmienoprzecinkowych podwójnej precyzji. Evidentnie 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.

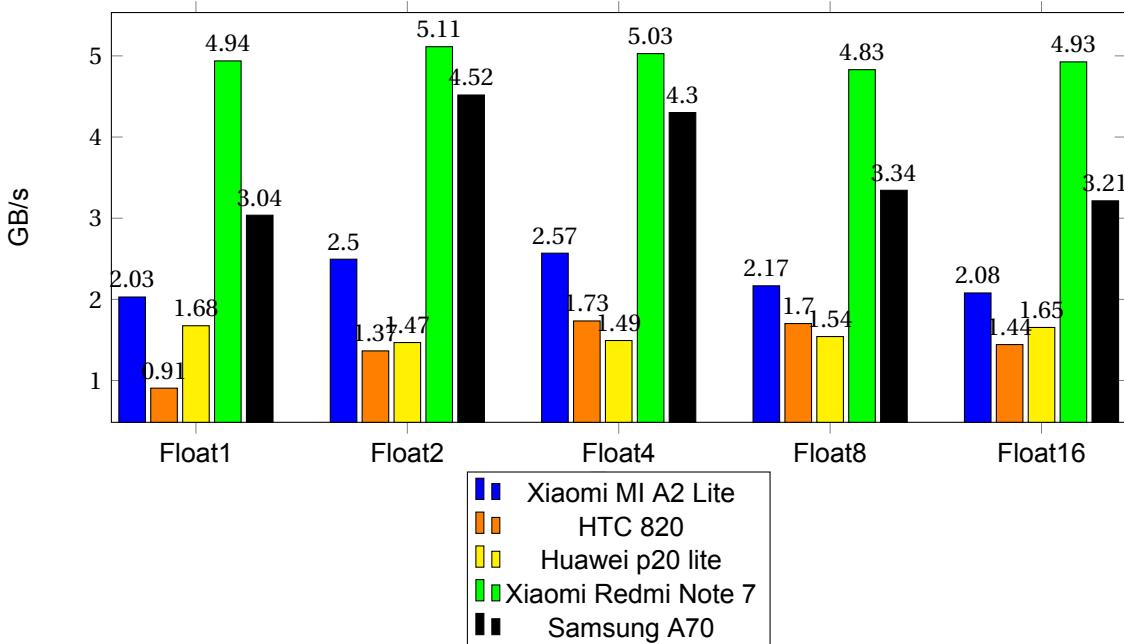
## 5.2. Wyniki przepływu pamięci



Rys. 5.5. Przepływ pamięci dla typu half



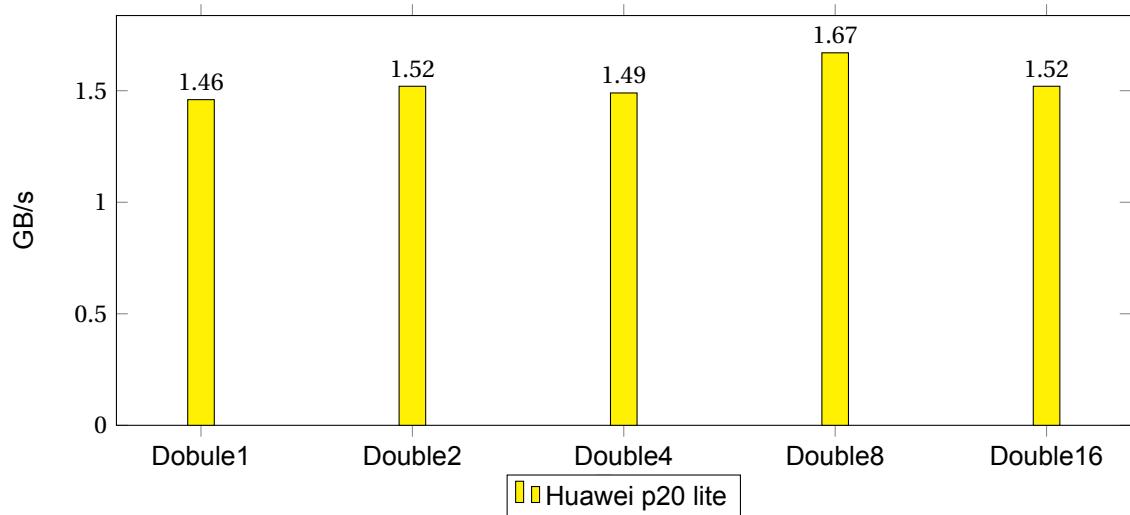
Rys. 5.6. Przepływ pamięci dla typu int



Rys. 5.7. Przepływ pamięci dla typu float

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 osiąga lepsze rezultaty niż urządzenie z procesorem Adreno 405. W pozostałych próbach wypada najgorzej. Spośród urządzeń z procesorami Adreno widać, że najlepsze rezultaty uzyskane są dla typu half a najlepsze dla typu half4. Inaczej niż w przypadku mocy obliczeniowej, najlepiej wypada urządzenie z procesorem Adreno 512. Najprawdopodobniej osiąga lepszy rezultat dzięki zastosowaniu lepszego typu pamięci.

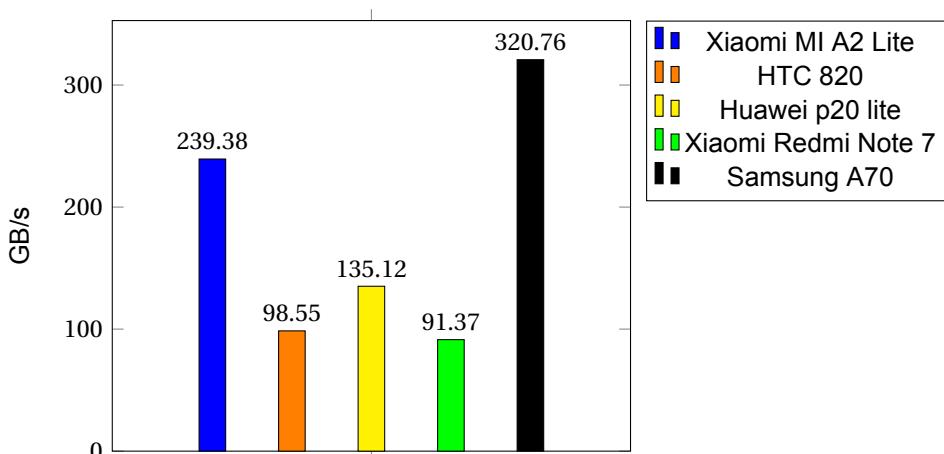
Wyniki przepływu danych dla typów float i int osiągają podobne wartości jak w przypadku wektorowych typów half. Kopiowanie pojedynczych wartości typu half wypada gorzej w porównaniu do typów 32 bitowych. Najlepsze wyniki osiąga urządzenie Xiaomi Redmi Note 7, które posiada procesor graficzny Adreno 512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



Rys. 5.8. przepływ pamięci dla typu double

Szybkość transferu pamięci używając typu double nie różni się względem typów half int czy float.

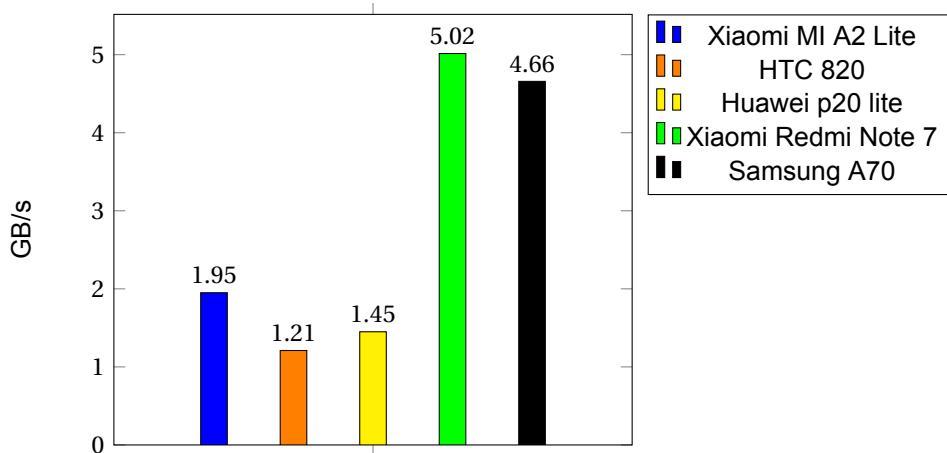
### 5.3. Czas oczekiwania na wykonanie



Rys. 5.9. Czas oczekiwania na wykonanie

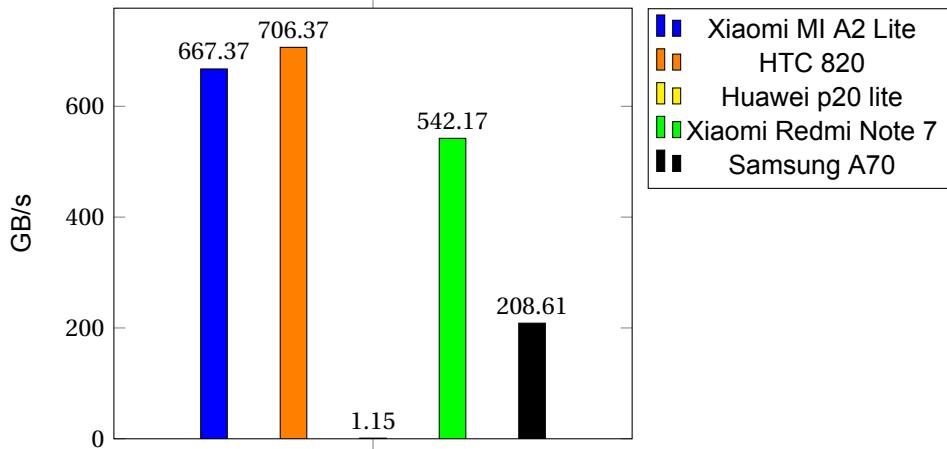
Powyższy wykres pokazuje średni czas od zakolejkowania do rozpoczęcia wykonania na gpu. Im mniejszy czas tym lepiej. Ku zaskoczeniu najgorzej wypada Samsung Galaxy A70, który był testowany na najnowszym sterowniku OpenCL spośród testowanych urządzeń. Za to drugi najlepszy wynik osiągnął HTC Desire 820, 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 czy kernela systemowego, niż od samego urządzenia.

#### 5.4. Transfery pamięci aplikacja-Urządzenie



Rys. 5.10. czas wykonywania clEnqueueReadBuffer

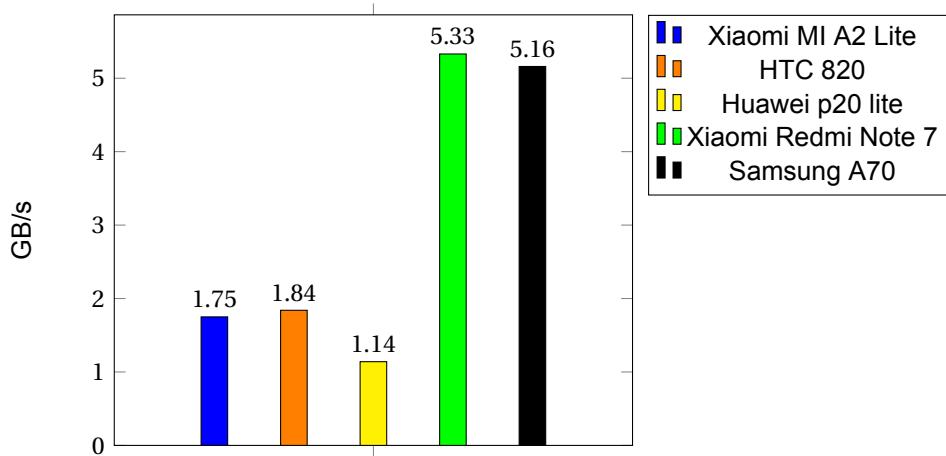
Powyższy wykres pokazuje szybkość transferu danych z urządzenia do aplikacji. Transfery tego typu odbywają się przy pomocy funkcji zdefiniowanych przez bibliotekę OpenCL. clEnqueueReadBuffer kopiuje dokładną ilość bajtów, więc transfer najprawdopodobniej odbywa się przy pomocy 8bitowego typu danych.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



Rys. 5.11. czas wykonywania clEnqueueWriteBuffer

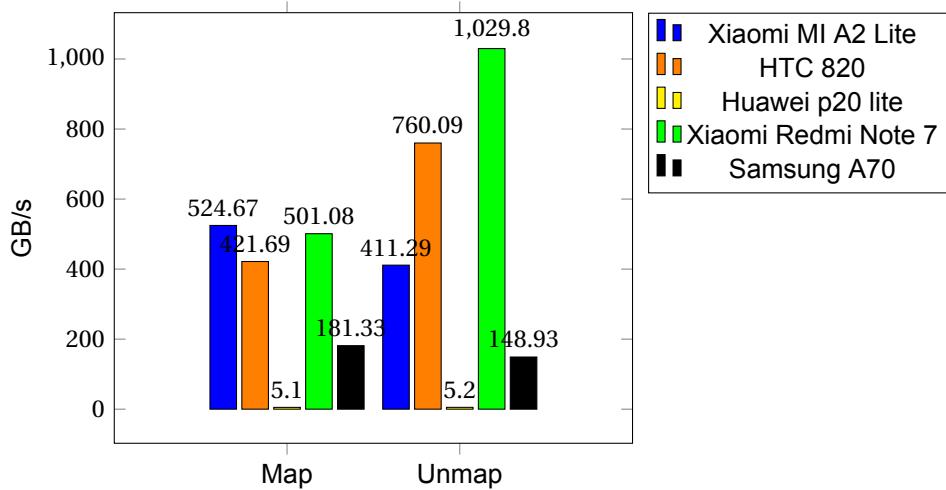
Wykres szybkości zapisu danych do alokacji w OpenCL. W przypadku procesorów Adreno pokazuje wartości nie możliwe do osiągnięcia w rodzajach pamięci zastosowanych w urządzeniach. Zakładany maksymalny przepływ pamięci np. Dla Samsunga Galaxy A70 to 14.9GB/s. Wartości te zostały zebrane przy pomocy obiektów 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.

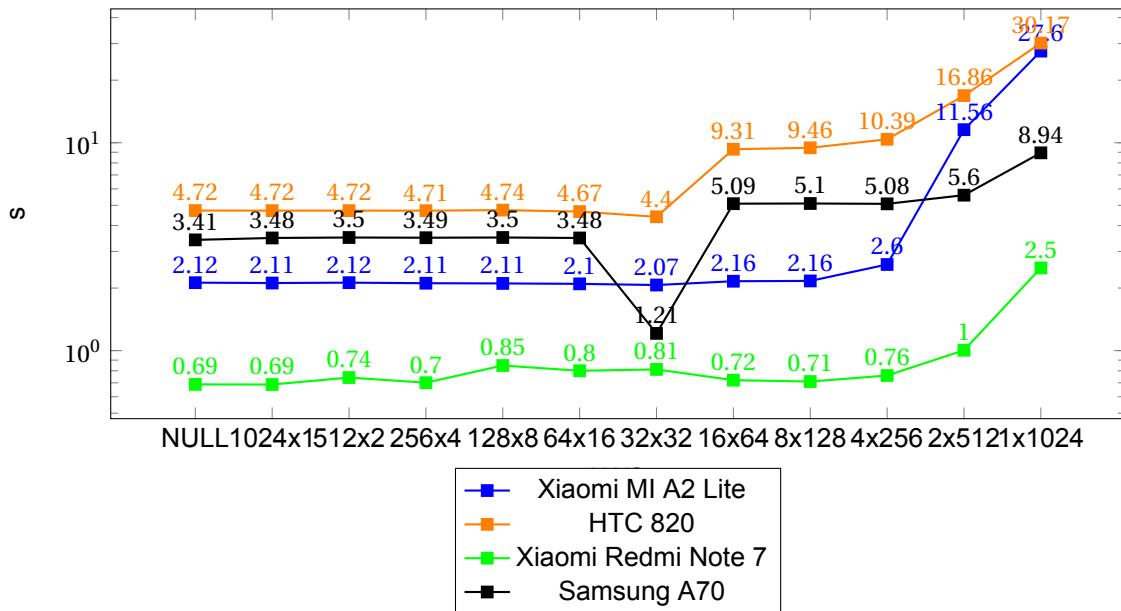


Rys. 5.13. czas wykonywania Map unMap

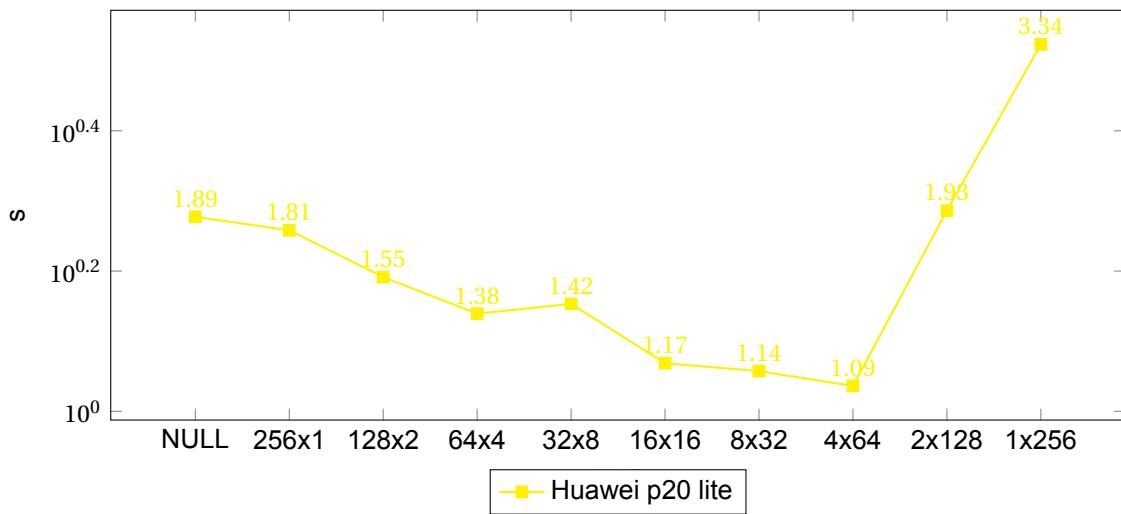
Powyższy wykres pokazuje czas potrzebny do mapowania pamięci po stronie urządzenia na pamięć po stronie aplikacji. Wartości uzyskane dla procesorów Adreno podobnie jak we wcześniejszym teście wydają się podejrzane, jednak powtórzone z mierzeniem czasu po stronie aplikacji zwróciły podobne rezultaty. Procesory Adreno korzystają ze współdzielonej pamięci systemowej. Najprawdopodobniej funkcja clEnqueueMapBuffer zwraca bezpośredni 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 clEnqueueReadBuffer. 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 ma maksymalną możliwą dla urządzenia wartość. Słaby rezultat dla takiego ustawienia, najprawdopodobniej spowodowany jest czytaniem odległych od siebie komórek pamięci w ramach work grupy. Pojedynczy wiersz macierzy zajmuje 4 KB, 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 różnych częściach 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 i wydłuża wykonywanie kernela.

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, przemnażamy rząd z pierwszej przez kolumnę z drugiej. Przy dostępowaniu 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.

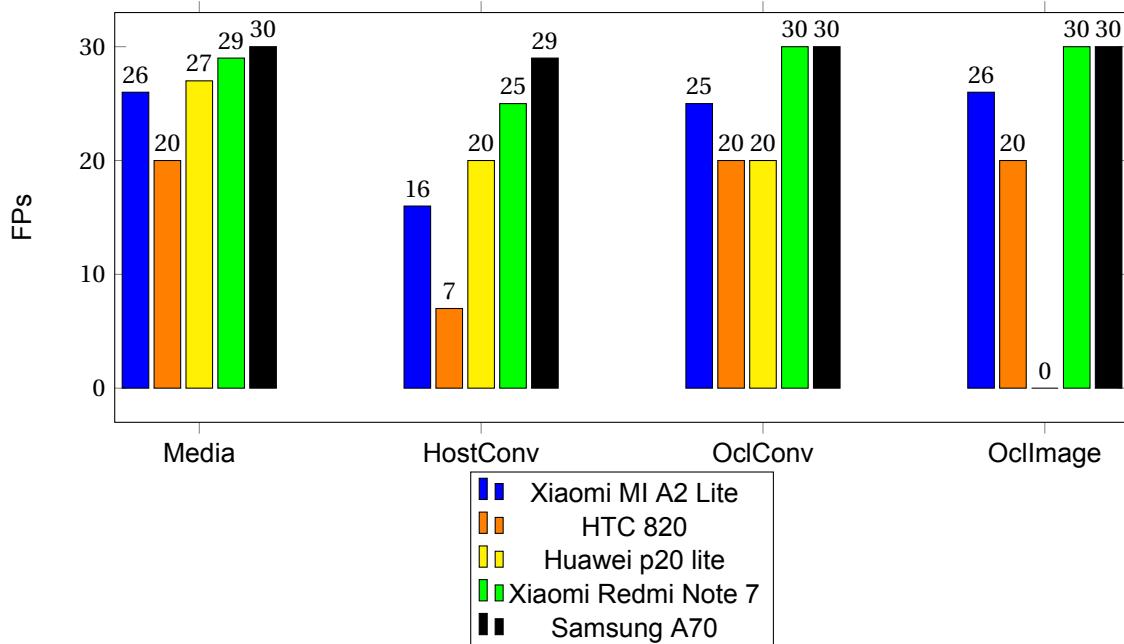
Najszybciej mnożenie macierzy wykonało urządzenie Xiaomi Redmi Note 7 najprawdopodobniej to dzięki najlepszemu typowi pamięci spośród testowanych telefonów, koszty dostępu i odczytu pamięci były najmniejsze. Drugim najszybszym urządzeniem jest Huawei P20 Lite z procesorem graficznym od Mali. Test korzystał 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 cameraApi**

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

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

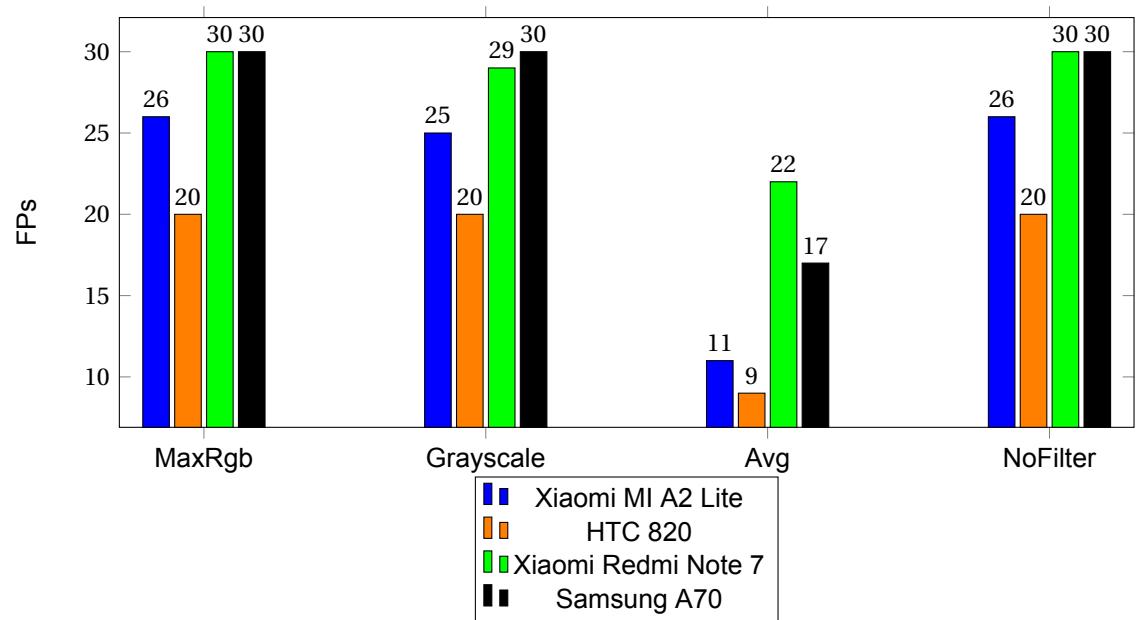
Niestety api OpenCL umożliwia współdzielenie zasobów z OpenGL tylko w formacie RGBA, więc niemożliwe jest dzielenie textury zwrotnej przez CameraApi. Potrzebna jest transformacja z formatu NV21 do RGBA. Druga kolumna wykresu pokazuje jak ilość klatek na sekundę wygląda, gdy konwersja do RGBA jest wykonywana po stronie aplikacji. Taka operacja ma duży wpływ na wszystkie urządzenia, widać wyraźny spadek wyświetlanych klatek na sekundę na 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 również kosztowne co brak zrównoleglenia konwersji po stronie CPU.



Rys. 5.16. Konwersja do RGB

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 kopiowania. Pierwszym, z alokacji w OpenCL do pamięci po stronie aplikacji, i drugim, z aplikacji do textury OpenGL. Na wykresie nie widać, żeby to jakkolwiek wpłynęło na liczbę wyświetlanych klatek. Najprawdopodobniej inne optymalizacje, nie przyniosły żadnych poprawek wydajności. Wąskie gardło, które sprawia, że nie można przekroczyć pewnej liczby klatek jest albo po stronie CameraApi, które nie jest w stanie dostarczyć większej ilości klatek na sekundę, albo środowiska OpenGL, które nie jest w stanie wyświetlić więcej klatek.

Wykres na rysunku 5.17, pokazuje wpływ filtrów wykonanych przez kernele w OpenCL na ilość wyświetlanych klatek na sekundę. Można zauważyć, że użycie prostych filtrów takich jak max rgb czy transformacja na obrazek w skali szarości, nie wpływają na ilość wyświetlanych klatek. Prawdopodobnie przez wąskie gardło które występuje gdzieś w czasie od zebrania podglądu z kamery do jego wyświetlania. Natomiast zastosowanie filtra uśredniającego, 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.



Rys. 5.17. Filtry

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

OpenCL jest biblioteką, która z pewnością może znaleźć zastosowanie w wielu aplikacjach z systemem Android. Brak oficjalnego wsparcia api na Androidzie sprawia, że ilość informacji o używaniu tej biblioteki przez programy na platformie Android jest znikoma. O używaniu tego api przez aplikacje można znaleźć tylko kilka wzorców w internecie. W poniższych podrozdziałach opisano wykorzystanie tej biblioteki w kilku aplikacjach.

### **6.1. Tensorflow**

Tensorflow to biblioteka, wykorzystywana do uczenia maszynowego. Tensorflow jest często wykorzystywany do treningów głębszych sieci neuronowych. Proces uczenia sieci neuronowych, wymaga wielokrotnego przemnażania wielu macierzy. Samo wykorzystanie wyuczonego modelu również wiąże się z procesem mnożenia macierzy. Tensorflow to oprogramowanie potrafiące wykorzystać różne api służące do obliczeń, takie jak Cuda, Sycl czy OpenCL. W przypadku wersji programu na system Android głównym api używanym przez tensorflow jest OpenCL. Jak wynika z artykułu *Even Faster Mobile GPU Inference with OpenCL* [11], użycie api OpenCL umożliwia poprawienie wydajności o 100% w stosunku do użycia OpenGL. Przy optymalizacji programu używając OpenCL pomocne były narzędzia dostarczone przez producenta GPU Adreno. Ponieważ api OpenCL nie jest oficjalnie wspierane, aplikacja tensorflow na starcie odpytuje urządzenie o dostępność biblioteki. W przypadku gdy biblioteka ta jest nie dostępna ładowany jest silnik działający w oparciu o api OpenGL.

### **6.2. OpenCV**

OpenCV (z ang. Open Source Computer Vision Library), to biblioteka, która implementuje wiele funkcjonalności służących do uczenia maszynowego i przetwarzania obrazów. Wiele z algorytmów dostarczanych przez bibliotekę wykonywanych jest na procesorach graficznych. Api wykorzywane przez OpenCV do uruchamiania algorytmów na GPU to OpenCL. W artykule *Use OpenCL in Android camera preview based CV application* [16] opisano, że użycie OpenCV-T Api, które wewnętrznie używa OpenCL, przy modyfikowaniu obrazu z kamery, poprawia ilość wyświetlanego klatek na sekundę. Wynik osiągany przez urządzenie Sony Xperia Z3, gdy modyfikacja obrazu odbywała się przy pomocy kodu w C/C++ to 3-5 klatek na sekundę. Podczas użycia OpenCV-T ilość klatek wzrosła do 11-13.

### **6.3. Adobe**

Adobe to producent oprogramowania, służącego między innymi do obróbki zdjęć i materiałów wideo. W przypadku produktów Adobe tworzonych na komputery z systemem Windows, api OpenCL jest często używane do uruchomienia przetwarzania na procesorach graficznych. W przypadku aplikacji tworzonych na system Android, wykorzystywanym api jest Vulkan. Jednak według wypowiedzi Erica Berdahla [13] produkty Adobe na Androidzie, wykorzystują kernele w języku OpenCL C, napisane dla komputerowych wersji programu. Kernele te są komplikowane do kodu pośredniego (SPIRV) a następnie wykonywane przy użyciu api Vulkana.

#### **6.4. Testy mierzące wydajność i jakość urządzenia**

Najwięcej aplikacji, które w najwidoczniejszy sposób deklarują używanie api OpenCL, to testy które mierzą wydajność urządzenia. Programy te najczęściej mają na celu zmierzenie mocy obliczeniowej czy uzyskanie podstawowych informacji o urządzeniu takich jak ilość jednostek wykonawczych. Pozyskiwane, przez te programy dane, mają wartość informacyjną o specyfice testowanego urządzenia.

## 7. PODSUMOWANIE

Celem niniejszej pracy była analiza środowiska jakim jest OpenCL na platformach mobilnych. Sprawdzono, że z pośród najpopularniejszych systemów mobilnych, tylko Android posiada możliwość obsługi tego api. Przy realizacji zadania sprawdzono, w jaki sposób programy, mogą łączyć się z biblioteką implementującą interfejs i jak to wpływa na ich przenośność. W ramach pracy skonstruowano i wykonano kilka testów mających za zadanie przetestować możliwości urządzeń z systemem Android w pracy z biblioteką OpenCL, pod kątem wydajnościowym, jak i współpracy api OpenCL z OpenGL przy wyświetlaniu obrazu z kamery urządzenia. Testy zostały przeprowadzone na urządzeniach opisanych i porównanych w niniejszym dokumencie. Wykonane aplikacje zostały dokładnie opisane, a ich wyniki w formie wykresów zostały przedstawione w pracy i przeanalizowane. W ramach pracy został wykonane poszukiwanie aplikacji wykorzystujących api OpenCL.

Przy realizacji pracy, problemem okazał się brak oficjalnego wsparcia dla OpenCL przez system Android. Pomimo to producenci procesorów graficznych na urządzenia mobilne, tworzą oprogramowanie, które implementuje funkcjonalność api OpenCL. By móc skorzystać z biblioteki należy albo bezpośrednio połączyć aplikację z konkretną dynamiczną biblioteką, albo ładować ją w trakcie wykonywania kodu. Przy zastosowaniu pierwszego rozwiązania aplikacja może być wykonywana tylko na urządzeniu z którego pochodzi biblioteka, na stałe połączona z programem. W drugim przypadku biblioteka jest ładowana dynamicznie. Interfejs OpenCL na platformie Android dostępny jest jedynie z poziomu natywnego kodu w C++, by używać jej w kodzie aplikacji, niezbędne jest wykonywanie części OpenCL po stronie natywnej biblioteki, lub wykonanie dodatkowej biblioteki, która będzie nakładką na funkcje w C++.

W ramach przeprowadzonych testów sprawdzono jaką moc obliczeniową mają poszczególne urządzenia przy użyciu api OpenCL. Wyraźnie zaobserwowano, że urządzenia wyprodukowane przez Adreno, reagują podobnie na zmianę typu danych i w raz z używaniem większych typów wektorowych ich moc obliczeniowa rosła, natomiast w przypadku procesorów Mali przy zastosowaniu typów wektorowych moc obliczeniowa pozostawała na tym samym poziomie. Procesory Mali są w stanie wykonywać więcej obliczeń na typach całkowitych, natomiast układy Adreno lepiej radzą sobie z operacjami na typach zmiennoprzecinkowych. Zweryfikowano, że rozmiary lokalnych work grup znaczco wpływają na czas wykonywania mnożenia macierzy, które wymaga czytania dalekich od siebie komórek pamięci. Urządzenia najlepsze czasy osiągały gdy rozmiary grup w wymiarze X i Y są równomierne. W przypadku takich grup podczas wykonywania wątki w ramach simd, mogły dostępować elementy z pamięci podręcznej. W takim przypadku kosztowne transfery alokacji do pamięci cache występują rzadziej. W tym teście kluczową rolą jest szybki dostęp do pamięci, dlatego im urządzenia miały lepszy typ pamięci, tym lepiej radziły sobie z tym zadaniem.

W ramach pracy, sprawdzono możliwości urządzeń do modyfikacji podglądu z kamery przy pomocy OpenCL. Okazało się, że podgląd z kamery dostarczany jest w formacie NV21, ob-

sługi którego api OpenCL nie definiuje. Dlatego by móc użyć obrazu z kamery w wykonywanym kernelu, należało przetransformować go do formatu RGBA. Konwersja znaczco wpływała jedynie na urządzenie z procesorem graficznym Mali. Na urządzeniach z GPU od Adreno proces zamiany obrazka na format RGBA i wpisanie go do wyświetlanej tekstury nie wpływał na ilość wyświetlanych klatek na sekundę. Po nałożeniu prostych filtrów jak zamianę na obraz w skali szarości lub filtr max rgb, liczba prezentowanych obrazów na sekundę nie zmieniła się. Natomiast użycie filtra uśredniającego, w którym do wyliczenia wartości jednego piksela potrzebne było odczytanie i uśrednienie wartości 25 pikseli, znaczco obniża ilość wyświetlanych klatek, dla niektórych urządzeń nawet o ponad połowę.

Istnieje mało informacji na temat korzystania z api OpenCL w aplikacjach użytkowych. Każdy z producentów procesorów graficznych dla urządzeń z systemem Android, posiada implementację takiego sterownika, co pozwala przypuszczać, że jest on częściej używany niż tylko do testów sprawdzających możliwości urządzenia. Problem z dotarciem do informacji na temat używania tego api przez aplikacje najprawdopodobniej wynika z tego, że twórcy programów zazwyczaj nie umieszczają nigdzie listy wszystkich użytych api, frameworków, czy wersji języka programowania.

W przypadku kontynuowania pracy nad tematem, można by było znaleźć sposób na sprawdzenie zużycia energii przy wykonywaniu kerneli. Innym kierunkiem kontynuowania prac mogłoby być sprawdzenie jak OpenCL na platformie Android działa na urządzeniach z kartami graficznymi Intel Amd czy Nvidi.

## WYKAZ LITERATURY

1. *Adreno*. 2021. URL: <https://en.wikipedia.org/wiki/Adreno>. (dostęp: 07.09.2021).
2. Ben Ashbaugh. *OpenCL on Linux*. 2019. URL: <https://bashbaug.github.io/opencl/2019/07/06/OpenCL-On-Linux.html>. (dostęp: 07.09.2021).
3. Krishnaraj Bhat. *libopencl-stub*. 2021. URL: <https://github.com/krrishnaraj/libopencl-stub>. (dostęp: 07.09.2021).
4. *Camera API*. 2021. URL: <https://developer.android.com/guide/topics/media/camera>. (dostęp: 07.09.2021).
5. *cl\_khr\_gl\_sharing(3) Manual Page*. 2020. URL: [https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/cl\\_khr\\_gl\\_sharing.html](https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/cl_khr_gl_sharing.html). (dostęp: 07.09.2021).
6. J Fang i in. "Parallel Programming Models for Heterogeneous Many-Cores". W: *A Survey, preprint*, (2020).
7. *FLOPS*. 2021. URL: <https://pl.wikipedia.org/wiki/FLOPS>. (dostęp: 07.09.2021).
8. *GLES20*. 2021. URL: <https://developer.android.com/reference/android/opengl/GLES20>. (dostęp: 07.09.2021).
9. *Grayscale to RGB Conversion*. URL: [https://www.tutorialspoint.com/dip/grayscale\\_to\\_rgb\\_conversion.htm](https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm). (dostęp: 07.09.2021).
10. <https://rocmdocs.amd.com/en/latest/ProgrammingGuides/Opencl-programming-guide.html>. 2021. (dostęp: 07.09.2021).
11. Juhyun Lee i Raman Sarokin. *Even Faster Mobile GPU Inference with OpenCL*. 2020. URL: <https://blog.tensorflow.org/2020/08/faster-mobile-gpu-inference-with-opencl.html>. (dostęp: 07.09.2021).
12. *Mali (GPU)*. 2021. URL: [https://en.wikipedia.org/wiki/Mali\\_\(GPU\)](https://en.wikipedia.org/wiki/Mali_(GPU)). (dostęp: 07.09.2021).
13. Simon McIntosh-Smith. *Catching Up with Khronos: Experts' QA on OpenCL 3.0 and SYCL 2020*. 2020. URL: <https://www.khronos.org/blog/catching-up-with-khronos-experts-qa-on-opencl-3.0-and-sycl-2020?fbclid=IwAR2Z6hhFXwM8LVv8MeTmTlqW8mqTkRzbCNKrntpAQ1> (dostęp: 08.09.2021).
14. A Munshi. "The OpenCL Specification". W: *Khronos OpenCL Working Group* (2012).
15. Basim Parapathil. "Adreno VS Mali | Which is Better for Me?" W: *tecvalue.com* (2021).
16. Andrey Pavlenko. *Use OpenCL in Android camera preview based CV application*. URL: [https://docs.opencv.org/master/d7/dbd/tutorial\\_android\\_ocl\\_intro.html#gsc.tab=0](https://docs.opencv.org/master/d7/dbd/tutorial_android_ocl_intro.html#gsc.tab=0). (dostęp: 08.09.2021).
17. Adrian Rosebrock. *Implementing the Max RGB filter in OpenCV*. 2015. URL: <https://www.pyimagesearch.com/2015/09/28/implementing-the-max-rgb-filter-in-opencv/>. (dostęp: 07.09.2021).
18. M Sawerwain. "OpenCL. Akceleracja GPU w praktyce". W: *PWN* (2014).
19. *SIMD*. 2021. URL: <https://en.wikipedia.org/wiki/SIMD>. (accessed: 07.09.2021).
20. R Tay. "OpenCL Parallel Programming Development Cookbook". W: *PACKT* (2013).
21. *Tutorial 1: Image Filtering*. URL: <https://ai.stanford.edu/~syueung/cvweb/tutorial1.html>. (accessed: 07.09.2021).

## WYKAZ RYSUNKÓW

2.1.	Linkowanie biblioteki OpenCL [2] .....	10
2.2.	N wymiarowy podział kernela .....	12
3.1.	SIMD [19].....	17
4.1.	Podgląd kamery.....	28
4.2.	Filtr max rgb.....	29
4.3.	Podgląd w skali szarości.....	29
4.4.	Podgląd z filtrem uśredniającym.....	30
5.1.	Moc obliczeniowa dla typu half .....	31
5.2.	Moc obliczeniowa dla typu int .....	32
5.3.	Moc obliczeniowa dla typu float .....	33
5.4.	Moc obliczeniowa dla typu double .....	33
5.5.	Przepływ pamięci dla typu half .....	34
5.6.	Przepływ pamięci dla typu int .....	34
5.7.	Przepływ pamięci dla typu float .....	35
5.8.	przepływ pamięci dla typu double .....	36
5.9.	Czas oczekiwania na wykonanie .....	36
5.10.	czas wykonywania clEnqueueReadBuffer .....	37
5.11.	czas wykonywania clEnqueueWriteBuffer .....	37
5.12.	czas wykonywania clEnqueueWriteBuffer czas z Aplikacji.....	38
5.13.	czas wykonywania Map unMap .....	38
5.14.	Mnożenie macierzy (Max Lws 1024) .....	39
5.15.	Mnożenie macierzy (Max Lws 256) .....	39
5.16.	Konwersja do RGB .....	41
5.17.	Filtры .....	42

## **WYKAZ TABEL**

3.1.	Bezpośrednie porównanie wad i zalet procesorów Mali i Adreno.....	18
3.2.	Porównanie testowanych telefonów .....	19
3.3.	Porównanie dostępności api na testowanych urządzeniach .....	20
3.4.	Porównanie testowanych procesorów graficznych .....	20

## **Dodatek A: PRZYKŁADOWY DODATEK**

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