Marek Sawerwain

OpenCL

Akceleracja GPU w praktyce

PWN

Marek Sawerwain

OpenCL

Akceleracja GPU w praktyce

PWN

Ten ebook jest chroniony znakiem wodnym

helion kopia dla:

Tomasz Gawlik morris@kozacki.pl

helion.pl

Projekt okładki i stron tytułowych
Grzegorz Laskowski

Ilustracja na okładce
Ociacia/Shutterstock

Wydawca
Łukasz Łopuszański

Redaktor
Henryka Walas

Koordynator produkcji
Anna Bączkowska

Skład wersji elektronicznej na zlecenie Wydawnictwa Naukowego PWN
Magdalena Wojtas / 88em.eu

Książka, którą nabyłeś, jest dziełem twórcy i wydawcy. Prosimy, abyś przestrzegał praw, jakie im przysługują. Jej zawartość możesz udostępnić nieodpłatnie osobom bliskim lub osobiście znanym. Ale nie publikuj jej w internecie. Jeśli cytujesz jej fragmenty, nie zmieniaj ich treści i koniecznie zaznacz, czyje to dzieło. A kopiując jej część, rób to jedynie na użytek osobisty.

Szanujmy cudzą własność i prawo

Więcej na www.legalnakultura.pl

Polska Izba Książki

Copyright © by Wydawnictwo Naukowe PWN SA

Warszawa 2014

eBook został przygotowany na podstawie wydania papierowego z 2014 r.

(wyd. I)

Warszawa 2016

ISBN 978-83-01-18045-4 (ePub)

ISBN 978-83-01-18046-1 (Mobi)

Wydawnictwo Naukowe PWN SA

02-460 Warszawa, ul. Gottlieba Daimlera 2

tel. 22 69 54 321, faks 22 69 54 288

infolinia 801 33 33 88

e-mail: pwn@pwn.com.pl

www.pwn.pl

Spis treści

1. Wprowadzenie

2. Standard OpenCL

2.1. Model platformy

2.2. Model pamięci

2.3. Model wykonawczy

2.3.1. Siatka obliczeniowa – NDRange

2.3.2. Kontekst obliczeń oraz kolejka poleceń

2.4. Model programowania

3. Język programowania dostępny w OpenCL

3.1. Typy danych

3.1.2. Typy wektorowe

3.1.3. Typy do obsługi obrazów

3.1.4. Inne typy oraz nazwy zarezerwowane

3.2. Dostępne operatory

3.3. Funkcje wbudowane

3.4. Dodatkowe kwalifikatory oraz atrybuty

3.5. Operacje na typach wektorowych OpenCL

3.5.1. Dane typu wektorowego – dostęp do składowych

3.5.2. Operacje na typach wektorowych

3.5.3. Funkcje obsługujące dane typu wektorowego

3.6. Obsługa siatki obliczeniowej

3.7. Funkcje synchronizacji i funkcje atomowe

3.8. Funkcje do obsługi obrazów

3.9. Preprocesor OpenCL C

3.10. Ograniczenia OpenCL C

4. OpenCL – łatwe przykłady na początek

4.1. Informacje o urządzeniu obliczeniowym

4.1.1. Wersja dla API w języku C

4.1.2. Informacje o urządzeniu obliczeniowym obiektowo i w C++

4.2. Witaj Świecie!

4.2.1. Przygotowania

4.2.2. Uruchomienie jądra obliczeniowego

4.3. Dodawanie wektorów

4.3.1. Jądro obliczeniowe

4.3.2. Wersja dla C++

4.3.3. Inne operacje na wektorach do ćwiczeń

4.4. Siatka lokalna oraz globalna

4.5. Zlecanie i wykonywanie zadań

4.6. Uwagi o kompilacji jądra obliczeniowego

5. Przykłady obliczeń w OpenCL

5.1. Obsługa printf w OpenCL

5.2. Operacja redukcji

5.2.1. Operacja redukcji – wersja szeregowa

5.2.2. Operacja redukcji – jądro obliczeniowe podejście pierwsze

5.2.3. Operacja redukcji – jądro obliczeniowe podejście drugie

5.2.4. Operacja redukcji – jądro obliczeniowe poddane optymalizacji

5.3. Wyznaczanie histogramu

5.3.1. Wersja szeregowa

5.3.2. Wersja dla OpenCL

5.4. Równoległe szukanie prawie binarne

5.4.1. Wersja równoległa dla OpenCL

5.5. Przybliżanie wartości liczby π

5.5.1. Wersja szeregowa

5.5.2. Wersja OpenCL – podejście bezpośrednie

5.5.3. Wersja OpenCL – podejście o wyższej wydajności

5.6. Gra w życie

5.6.1. Gra w życie – procedura obliczeniowa

5.6.2. Gra w życie – obsługa symulacji

5.7. Zbiór Mandelbrota

5.7.1. Zbiór Julii

5.7.2. Tworzenie obrazu zbioru Mandelbrota – wersja szeregowa

5.7.3. Tworzenie obrazu zbioru Mandelbrota – wersja OpenCL

5.7.4. Tworzenie obrazu zbioru Julii

5.7.5. Kolorowanie zbioru fraktalnego

5.8. Algorytm sortowania bitonicznego

5.8.1. Sieci sortujące

5.8.2. Bitoniczna sieć sortująca

5.8.3. Szeregowa wersja algorytmu sortowania bitonicznego

5.8.4. Równoległa wersja algorytmu sortowania bitonicznego dla OpenCL

5.9. Przetwarzanie obrazów

5.9.1. Podstawy – wypełnienie obrazu

5.9.2. Skalowanie obrazu

5.9.3. Rozmywanie obrazu – filtr Gaussa

5.9.4. Wykrywanie krawędzi w obrazie za pomocą filtru Sobela

5.10. Współpraca OpenCL z OpenGL

5.10.1. Procedury obliczeniowe OpenCL

5.10.2. Czynności wykonywane w programie gospodarza

6. OpenCL w innych językach programowania

6.1. Język Python

6.1.1. Przegląd API pakietu PyOpenCL

6.1.2. Znów dodawanie wektorów, ale w Pythonie

6.1.3. Skrypt do testowania siatki obliczeniowej

6.1.4. Stosowanie typów użytkownika

6.1.5. Klasa w Pythonie dla łatwiejszej współpracy z OpenCL

6.2. Pakiet APARAPI dla języka Java

6.2.1. Dodawanie wektorów

6.2.2. Przegląd API pakietu APARAPI

6.2.3. Algorytm Blacka-Sholesa

6.2.4. Mechanizm rozszerzeń

6.3. Inne rozwiązania, VexCL i nie tylko

7. Zamiast zakończenia

1.1.1. Analiza wydajności symulacji przykładu „Gra w życie”

Dodatek A. Kompilacja przykładów

Dodatek B. Instalacja pakietu OpenCL dla języka Python

Dodatek C. Opis funkcji pomocniczych

C.1. Zapis do formatu PPM

C.2. Konwersja kodów formatu koloru do wartości znakowych

C.3. Zapis i odczyt danych w formacie PNG

C.4. Konwersja z modelu kolorów HSV do RGB

C.5. Utworzenie macierzy dla filtru rozmywającego

Dodatek D. Zestawianie używanych funkcji API OpenCL

D.1. Funkcja clGetPlatformIDs

D.2. Funkcja clGetPlatformInfo

D.3. Funkcja clGetDeviceInfo

D.4. Funkcja clGetSupportedImageFormats

D.5. Funkcja clGetDeviceIDs

D.6. Funkcja clCreateContext

D.7. Funkcja clBuildProgram

D.8. Funkcja clGetProgramBuildInfo

D.9. Funkcja clCreateCommandQueue

D.10. Funkcja clCreateBuffer

D.11. Funkcja clEnqueueReadBuffer

D.12. Funkcja clEnqueueWriteBuffer

D.13. Funkcja clCreateProgramWithSource

D.14. Funkcja clCreateKernel

D.15. Funkcja clSetKernelArg

D.16. Funkcja clEnqueueNDRangeKernel

D.17. Funkcja clEnqueueTask

D.18. Funkcja clGetKernelWorkGroupInfo

D.19. Funkcja clCreateImage2D

D.20. Funkcja clEnqueueReadImage

D.21. Funkcja clEnqueueWriteImage

D.22. Funkcja clFinish

D.23. Funkcja clReleaseKernel

D.24. Funkcja clReleaseProgram

D.25. Funkcja clReleaseCommandQueue

D.26. Funkcja clReleaseMemObject

D.27. Funkcja clReleaseContext

D.29. Funkcja clCreateFromGLTexture2D

D.30. Funkcja clEnqueueAcquireGLObjects

D.31. Funkcja clEnqueueReleaseGLObjects

Dodatek E. Spis kodów błędów API OpenCL

Dodatek F. Najczęściej spotykane nazwy rozszerzeń

Literatura

1 Wprowadzenie

Współczesne systemy komputerowe cechują się równoległością. Komputery klasy PC, telefony, tablety czy ogólnie urządzenia mobilne mają przynajmniej dwa procesory przeznaczone do wykonywania programów użytkownika. Karty graficzne również są wyposażone w wiele jednostek przetwarzających – karty średniej klasy mają nawet 1000 rdzeni obliczeniowych, tańsze około 300 rdzeni, a największe i najdroższe nawet 5000[1].

Techniki programowania odpowiednie dla tradycyjnego procesora z jednym rdzeniem nie dają możliwości wykorzystania mocy obliczeniowej urządzenia mającego choćby tylko 100 rdzeni obliczeniowych. Jednak powstało wiele różnych rozwiązań obsługujących programowanie równoległe.

Bardzo dużą popularność w dziedzinie uniwersalnych obliczeń wykonywanych z pomocą kart graficznych zdobyła technologia CUDA (ang. *Compute Unified Device Architecture*). Jednak standardowym rozwiązaniem w tej dziedzinie jest OpenCL (w skrócie standard OCL). Istotną zaletą standardu OCL jest to, że działa z kartami graficznymi różnych producentów, a także może być stosowany dla innych rozwiązań technologicznych niż karty graficzne. Warto też pamiętać o technologii C++AMP opracowanej przez Microsoft, która również jest niezależna od producentów kart graficznych, ale działa tylko w systemach operacyjnych z rodziny Windows.

Jak się wydaje, standard OpenCL będzie coraz bardziej popularny w programowaniu nie tylko kart graficznych, lecz także tradycyjnych procesorów czy innych rozwiązań o wysokiej wydajności. Nie jest on ograniczony do konkretnego systemu operacyjnego czy komputerów klasy PC, ale można go zastosować też do najnowszych układów stosowanych w telefonach i tabletach.

Duża popularność techonologii CUDA wynika również z jej daleko posuniętej integracji z językiem C/C++. W programie napisanym w technologii CUDA łatwo przeoczyć, który fragment jest wykonywany przez kartę graficzną, a który przez tradycyjny procesor. W przypadku OpenCL przygotowuje się oddzielny program, stanowiący tzw. **procedurę obliczeniową** (jądro obliczeniowe albo funkcję obliczeniową), a wewnętrzne mechanizmy OpenCL dopasowują podany program do aktualnie używanej platformy wykorzystywanej do obliczeń w ramach OpenCL. Oznacza to zasadniczą przewagę standardu OpenCL nad technologią CUDA, gdyż może być stosowany nie tylko do kart graficznych, ale również do tradycyjnych procesorów, które obecnie są wyposażone w wektorowe jednostki oferujące instrukcje typu SSE, AltiVec, Neon czy AVX. Nowością jest też wykorzystywanie standardu OpenCL do układów FPGA[2], które po odpowiednim zaprogramowaniu również mogą stanowić wydajną jednostkę do realizacji obliczeń.

Warto zapoznać się z technologią OpenCL, bowiem wykorzystanie kart graficznych wiodących producentów do realizacji obliczeń uniwersalnych różnego typu jest już od kilku lat bardzo popularne. Wiele aplikacji (np. związanych z przetwarzaniem grafiki, multimediów), a nawet programy do kompresji danych pozwalają na wykorzystanie mocy obliczeniowej kart graficznych do przyspieszenia obliczeń. Znakomitym przykładem jest GIMP – jeden z ogólnie dostępnych programów o otwartym źródle, a dokładnie podsystem GEGL, w którego najnowszej wersji standard OCL będzie szeroko stosowany. Innym przykładem jest program WinZIP, który wykorzystuje OpenCL do przyspieszenia kompresji oraz dekompresji.

1.1. Obliczenia równoległe

Choć obecnie istnieją dwa główne standardy tworzenia programów dla kart graficznych: wymieniona technologia CUDA firmy NVIDIA oraz standard OpenCL (który można traktować jako odpowiednik OpenGL do realizacji obliczeń). Oba rozwiązania mają swoje zalety i wady. CUDA jest bardzo popularna, jednakże działa tylko z kartami firmy NVIDIA. OpenCL to standard ustanowiony przez organizację Khronos – aplikacje opracowane za jego pomocą działają na kartach graficznych głównych producentów kart graficznych, obecnie są to firmy NVIDIA oraz AMD.

Wymienione technologie mają podobny model programowania. Zaletą technologii CUDA, jak wspomniano we wstępie, jest pełna integracja z językiem C/C++. Właściwie to należy mówić o językach programowania CUDA C i CUDA C++. OpenCL oferuje odrębny język programowania – podobny do C, ale nie jest to C++. Stanowi to też pewną wadę, ponieważ nie są dostępne np. szablony. W CUDA można tworzyć klasy języka C++, których metody są realizowane przez kartę graficzną. W przypadku OpenCL tworzone są tzw. **kernele** (w dalszej części książki będą one nazywane **jądrami obliczeniowymi** albo **procedurami/funkcjami obliczeniowymi**) stanowiące oddzielne programy do wykonania przez odpowiednie urządzenia obliczeniowe dostępne w ramach tzw. platformy OpenCL. Warto tu wspomnieć o pakiecie APARAPI dla języka Java (zostanie on szerzej przedstawiony w rozdz. 6.2), który funkcjonuje podobnie jak technologia CUDA. Program przeznaczony do realizacji na karcie graficznej opracowuje się w języku Java, następnie jest on tłumaczony na OpenCL, aby mógł być wykonany przez kartę graficzną (nawiasem mówiąc jest on również tłumaczony z OpenCL na odpowiedni assembler).

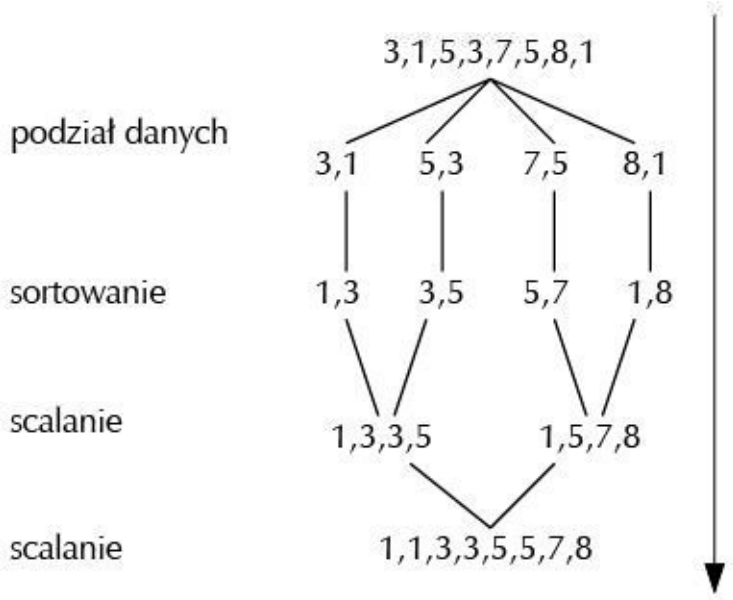
W obydwu wiodących technologiach można spotkać te same rozwiązania, które opierają się na podstawowych pojęciach, jakie występują w obliczeniach równoległych. Sposób programowania w pracy z jednym procesorem polega na odczytaniu zestawu danych i element po elemencie wykonaniu obliczeń. Ten sposób niestety nie sprawdzi się w przypadku, gdy mamy do dyspozycji wiele jednostek obliczeniowych, bowiem nie wykorzystamy optymalnie dostępnej mocy obliczeniowej.

Równoległe rozwiązanie problemu wymaga podzielenia zbioru na mniejsze części. Ogólnie można mówić o dwóch sposobach podziału danego zadania: pierwszy to „dziel i rządź” (ang. *divide-and-conquer*), stosowany także w programowaniu szeregowym. Drugi to „rozprosz i zbierz” (ang. *scatter-and-gather*).

W sposobie „dziel i rządź” dany problem jest dzielony na mniejsze problemy, które są łatwiejsze do rozwiązania. Odpowiednie połączenie rozwiązań mniejszych problemów pozwala rozwiązać problem główny. Drugie podejście jest podobne, ale zamiast dzielenia całego problemu na mniejsze podproblemy, rozdziela się tylko dane i przekazuje te fragmenty do jednostek obliczeniowych. Późniejszy odbiór danych oraz ich dodatkowe uporządkowanie pozwala na poprawne zakończenie obliczeń. Jak widać, w obydwu podejściach stosuje się rozdział problemu lub danych na mniejsze części.

Na rysunku 1.1 przedstawiono schemat podziału danych, a także ogólnie całego problemu na mniejsze zadania. Przedstawia on proces sortowania przez scalanie ośmiu elementów. Zestaw danych został podzielony na cztery grupy po dwa elementy. Każda z grup może być przetwarzana niezależnie od pozostałych. Następnie grupy należy połączyć ponownie. To zadanie może być realizowane

niezależnie przez dwa procesory, choć ostatecznie proces sortowania może zostać zakończony przez łączenie danych wykonane przez jeden procesor.



Rys. 1.1. Sortowanie przez scalanie jako algorytm równoległy (poszczególne etapy mogą być realizowane przez kilka różnych jednostek obliczeniowych)

Pokazuje to też jeden bardzo ważny aspekt obliczeń realizowanych równolegle, a mianowicie, niezależność danych. Wysoką wydajność można otrzymać w dość prosty sposób, jeśli dane można łatwo podzielić na niezależne od siebie grupy. W przypadku sortowania przez scalanie, istotnie w pierwszym kroku można łatwo wydzielić poszczególne grupy. Jednak dalsze działania, polegające na scalaniu danych, niestety łączą je w większe grupy, co utrudnia zastosowanie przetwarzania równoległego. Mimo to, sortowanie przez scalenie jest algorytmem, który dość łatwo zaimplementować w środowisku równoległym, choć istnieje inne podejście, tzw. **sortowanie bitoniczne**, które w środowisku, gdzie dostępnych jest wiele jednostek przetwarzających, daje lepszą wydajność (algorytm sortowania bitonicznego został przedstawiony w rozdz. 5.8).

Innym przykładem operacji, która łatwo daje się zaimplementować w środowisku równoległym, jest mnożenie dwóch wektorów element po elemencie:

```
for (i=0; i<N; i++)

    C[i] = A[i] * B[i];
```

Rezultat operacji jest umieszczany w wektorze wynikowym reprezentowanym przez zmienną `C`. Istotnie, poszczególne mnożenia można wykonywać w sposób zupełnie niezależny od siebie. Jednak zlecanie wykonania tylko jednej operacji mnożenia i zapisania wyniku tylko dla jednej jednostki obliczeniowej również nie jest najlepszym rozwiązaniem pod względem wydajności. Wykonanie tych operacji jest zbyt proste i dostęp do pamięci, aby odczytać kolejny element, może być bardziej czasochłonny niż wykonanie mnożenia, toteż należy tak przetwarzać dane, aby grupować je w odpowiednio duże grupy.

Ten problem w obszarze programowania równoległego nazywa się **ziarnistością**, (ang. *grain*). Podany przykład mnożenia wektorów to tzw. **problem drobnoziarnisty** (ang. *fine-grained*), bowiem

mamy wiele drobnych obliczeń, które mogą być realizowane równolegle. Dodatkowo, nie ma znaczenia kolejność wykonywania poszczególnych mnożeń. Łatwo także grupować poszczególne operacje. W efekcie ułatwia to rozłożenie obciążenia obliczeniami na poszczególne jednostki obliczeniowe, co również jest bardzo istotne, aby wykorzystać moc wszystkich dostępnych jednostek.

Przeciwieństwem obliczeń drobnoziarnistych są **obliczenia gruboziarniste** (ang. *coarse-grained*), w których istnieje możliwość wskazania etapów obliczeń równoległych prowadzonych w sposób niezależny. Jednakże nie jest łatwo je rozdzielić na małe podproblemy. Trudno także zrównoważyć obciążenie poszczególnych jednostek obliczeniowych. Zazwyczaj jedną grupą danych zajmuje się jeden **wątek obliczeniowy**. W opisywanej technologii OpenCL jeden wątek nazywany jest też **jednostką roboczą** (więcej informacji można znaleźć w rozdz. 2).

Przykłady omawiane w tej książce należą do problemów drobnoziarnistych, do których rozwiązania stosunkowo łatwo można wykorzystać wiele jednostek roboczych. Większość problemów cechuje niezależność danych, ale nie wszystkie. Zdarza się nawet w przypadku obliczeń drobnoziarnistych, że wymagane jest przeprowadzenie dodatkowych operacji, aby przetwarzane dane pozostały spójne.

Dlatego w przypadku programowania równoległego potrzebne są mechanizmy synchronizacji danych zarówno na poziomie pojedynczych wątków, czy też jednostek roboczych, jak też między poszczególnymi etapami realizacji całego algorytmu. Technologia OpenCL dostarcza takie rozwiązanie na poziomie pojedynczych wątków poprzez polecenie `barrier` oraz na poziomie wyższym, gdzie zleca się wykonanie pewnych obliczeń przez tzw. jądra obliczeniowe. W takim przypadku należy np. zaczekać na zakończenie realizacji obliczeń, aby rozpocząć odczytywanie danych lub zlecić wykonanie kolejnej procedury obliczeniowej.

W tym krótkim wprowadzeniu nie będą dokładnie omawiane problemy programowania równoległego, w tym także współbieżnego, gdzie istotną rolę odgrywa dostęp do różnorakich zasobów. Warto jeszcze podkreślić rolę wątków w programowaniu równoległym. To zazwyczaj wątki wykonują operacje na wydzielonym zbiorze danych. Jeśli uda się tak podzielić dane, że nie ma potrzeby, aby wątki się ze sobą komunikowały, to równoległa procedura obliczeniowa ma dość prostą postać. Jednak w wielu przypadkach konieczna będzie komunikacja między wątkami. Wykonuje się ona za pomocą tzw. **mechanizmu pamięci dzielonej** (tego typu pamięć również jest dostępna w ramach OpenCL, nazwana jest **pamięcią lokalną**). W programowaniu równoległym, a dokładnie we współbieżnym, istotnym elementem jest synchronizacja danych np. za pomocą **semaforów** czy tzw. **mutexów**. W przypadku OpenCL możliwości synchronizacji wątków są dość ograniczone, bowiem wspierany jest tylko mechanizm bariery, dostępne są także tzw. instrukcje atomowe, zapewniające spójność operacji na pamięci.

Istnieją także inne sposoby realizacji aplikacji równoległych, np. przez przekazywanie komunikatów, bądź poprzez tzw. aktorów. Pojęcia te wymagałyby dokładniejszego omówienia, jednakże nie jest to celem tej książki, dlatego zachęcamy w tym miejscu do zapoznania się ze spisem literatury zamieszczonym na końcu. Pozycje [Czech2013], [HerlihyShavit2010] oraz [Cormen2012] dokładniej omawiają ogólne zagadnienia związane z programowaniem równoległym.

1.2. Zawartość książki

Książek z dziedziny informatyki zazwyczaj nie czyta się od przysłowiowej deski do deski. Pozycja na temat standardu OpenCL również jest tego typu książką.

W pierwszej kolejności warto przeczytać rozdział 2, w którym znajduje się materiał wstępny opisujący architekturę OpenCL, a także rozdział 3, w którym opisano język OpenCL C. Następnie można przejść do lektury rozdziału 4, w którym pokazano mniej skomplikowane przykłady aplikacji w OpenCL. Pozwalają one zapoznać się z podstawowym API OpenCL, jakie trzeba stosować, aby wykorzystać możliwości OpenCL. Głównym językiem programowania, jaki będzie stosowany, jest język C, ale pokazano też przykłady dla języka C++.

Rozdział 5 zawiera dziesięć przykładów rozwiązań różnych zagadnień, jak np. operację redukcji, wyznaczanie histogramu, a także dobrze znany przykład rysowania zbioru fraktalnego. OpenCL oferuje też znaczne udogodnienia w przetwarzaniu obrazów – w rozdziale 5.9 podano przykłady względnie łatwego zaimplementowania kilku filtrów do szybkiego przetwarzania obrazu. Sposób współpracy OpenCL z OpenGL omówiono w rozdziale 5.10. Pokazany przykład, choć dość prosty, jednakże obrazuje zastosowanie karty graficznej do wykonywania obliczeń, a także wizualizacji za pomocą funkcji OpenGL. W niemal wszystkich rozdziałach język C/C++ stanowi główne narzędzie do sterowania obliczeniami w OpenCL, ale wyjątkiem jest rozdział 5.5, gdzie zamiast języka C/C++ stosowany jest język Python.

W rozdziale 6 omówiono wykorzystanie OpenCL w innych językach programowania. Funkcje OpenCL, wzorem OpenGL, jest stosunkowo łatwo przenieść do innego języka programowania. W rozdziale tym przedstawimy dokładniej tylko dwa rozwiązania wykorzystujące język Python oraz pakiet APARAPI współpracujący z językiem Java.

Uzupełnieniem treści książki są dodatki. W Dodatku A przedstawiono sposób kompilacji przykładów, jakie pojawiają się w całej książce. Podano też skrypt dla programu makefile, który ułatwia przeprowadzanie kompilacji np. w systemach Linux, czy MacOS, ale także funkcjonuje w rodzinie systemów Windows. Kilka informacji o tym, jak dokonać instalacji pakietu OpenCL dla języka Python podano w kolejnym dodatku.

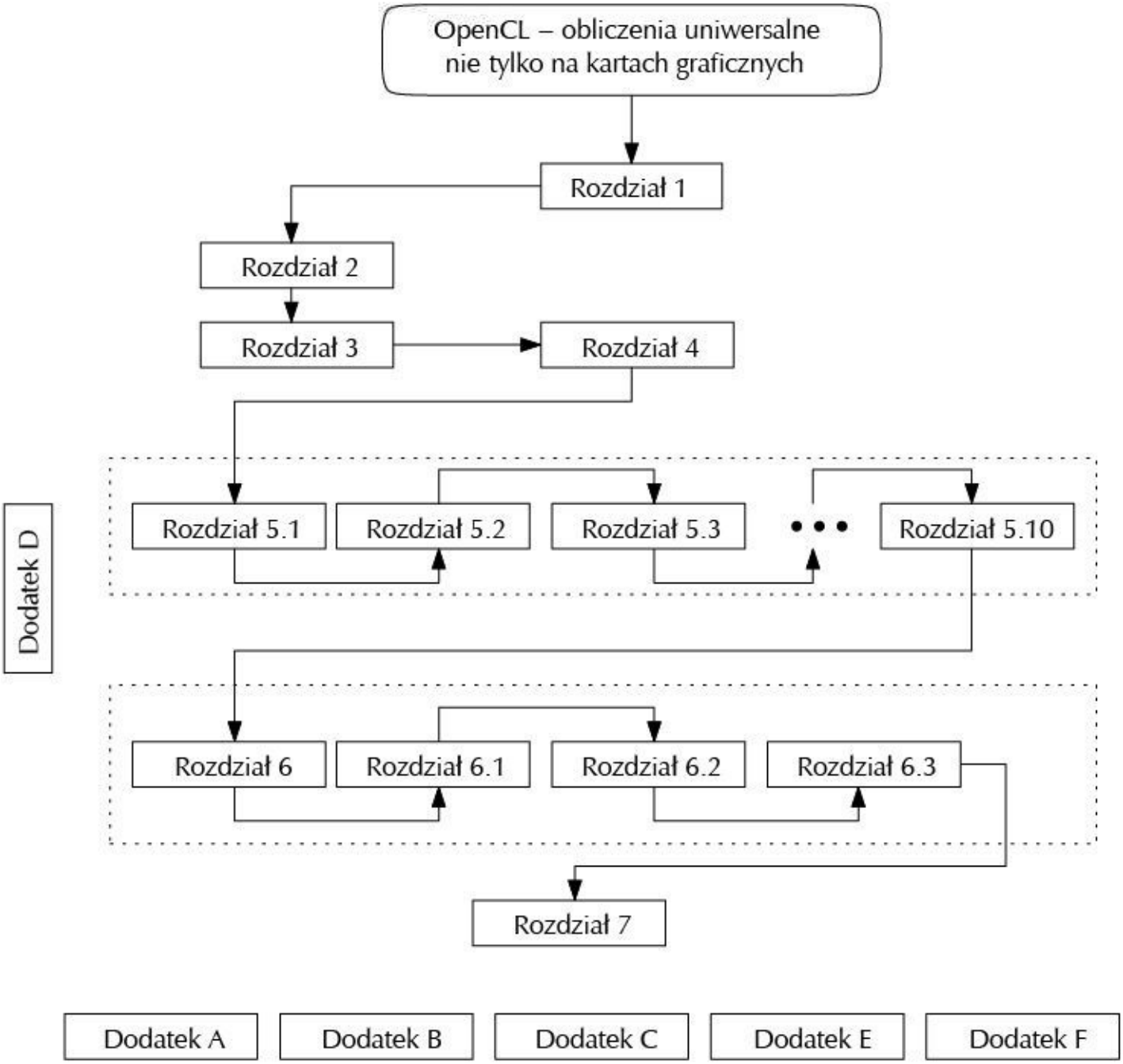
Przykłady zaprezentowane w rozdziale 5 korzystają z kilku dodatkowych funkcji, które zostały omówione w Dodatku C. W miarę możliwości zaprezentowano też kody źródłowe tych funkcji. Istotną rolę pełni bardzo obszerny Dodatek D, gdzie w sposób syntetyczny przedstawiono wybrany zbiór funkcji API OpenCL, jaki jest używany w przykładach przedstawionych w rozdz. 4 oraz 5. Dla każdej funkcji podano także przykłady jej użycia, co odróżnia ten zestaw od podobnego, jaki można znaleźć w specyfikacji dostępnej na stronach organizacji standaryzującej technologię OpenCL.

Pomocne są dodatki E oraz F, w których zgromadzono kody błędów, jakie zwracane są przez funkcje API OpenCL, oraz spis rozszerzeń, jakie można napotkać w obecnie dostępnych implementacjach OpenCL dla najbardziej popularnych producentów kart graficznych, a także procesorów.

Ostatnim elementem jest spis literatury, która uzupełnia informacje zawarte w tej książce, oraz indeks ułatwiający odszukanie informacji o różnych funkcjach, czy pojęciach, jakie pojawiają się na stronach tej książki.

Schemat zawartości książki i relacje między poszczególnymi rozdziałami zostały pokazane na rys. 1.2. Rozdziały 1, 2, 3 oraz 4 dostarczają podstawowych informacji, dlatego warto przeczytać ich zawartość wcześniej. Natomiast kolejność lektury rozdziałów 5 i 6 może być dowolna. Warto także sięgać do dodatku D, gdzie przedstawione zostały funkcje API OpenCL, jakie pojawiają się w przykładach w rozdziałach 5 i 6, stanowiących główne rozdziały książki. Rozdział 7 nie stanowi

typowego zakończenia, przedstawia bowiem sposób wykorzystania narzędzia CodeXL do testowania wydajności jednego z przykładów, jaki został przytoczony w rozdziale 5.



Rys. 1.2. Struktura rozdziałów oraz relacje między poszczególnymi rozdziałami

Przypisy

[1] Podane liczby dotyczą rozwiązań w roku 2014, z pewnością w następnych latach liczba rdzeni obliczeniowych zarówno w przypadku tradycyjnych procesorów, jak i kart graficznych zostanie podwojona.

[2] FPGA – *Field Programmable Gate Array*, w bezpośrednim tłumaczeniu: programowalna macierz bramek. Układ taki można wielokrotnie programować, aby zmieniać jego przeznaczenie i funkcjonalność, nawet po zamontowaniu w urządzeniu docelowym.

2 Standard OpenCL

Technologia OpenCL to środowisko dostarczające narzędzia do tworzenia programów równoległych dla wielu dostępnych jednostek przetwarzających. OpenCL nie określa konkretnego typu urządzenia, na którym powinna być uruchamiana tworzona aplikacja. Oczywiście, obecnie najwydajniejsze urządzenia obliczeniowe korzystające z dobrodziejstw OpenCL to m.in. karty graficzne. Mimo to istnieją implementacje OpenCL dla tradycyjnych procesorów, a także możliwe jest stosowanie OpenCL w systemach przetwarzania danych opartych na technologii układów FPGA.

Istotne było opracowanie takiej technologii, która będzie elastyczna w stosunku do różnych urządzeń obliczeniowych. Aby to osiągnąć, standard OpenCL oparto na czterech modelach opisujących podstawowe obszary:

• modelu platformy,

• modelu pamięci,

• modelu wykonawczym,

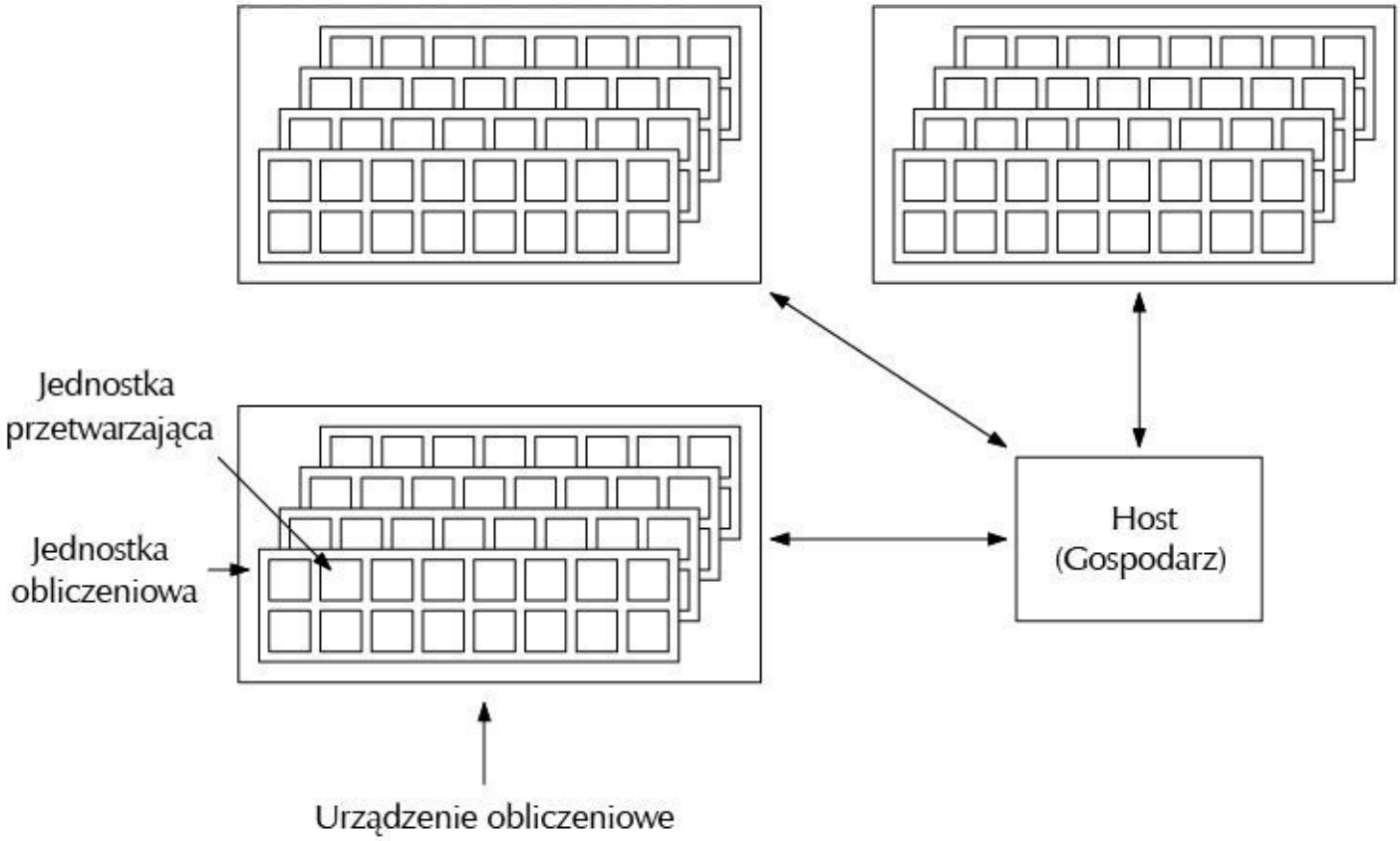
• modelu programowania.

W tym rozdziale przedstawiono krótko te cztery obszary oraz podstawowe pojęcia pojawiające się w ich kontekście, mają one bowiem bezpośredni wpływ na sposób tworzenia oprogramowania korzystającego z możliwości technologii OpenCL.

2.1. Model platformy

Zakłada się, że OpenCL będzie tzw. **technologią heterogeniczną**, tj. taką, która obsługuje różne platformy sprzętowo-programowe różnych producentów. Co więcej, w ramach jednego systemu OpenCL mogą współistnieć rozwiązania różnych producentów. Sterownik OpenCL umożliwi współpracę między tymi urządzeniami.

Model platformy OpenCL w schematyczny sposób przedstawiono na rys. 2.1, gdzie podstawowym elementem jest tzw. **system hosta** (gospodarza). Do systemu gospodarza podłączone są oddzielne urządzenia obliczeniowe zgodne z technologią OpenCL. Nie muszą to być takie same urządzenia w sensie sprzętowym, np. mogą stosować różne układy bitów. Obecnie standard OpenCL jest implementowany na czterech typach urządzeń obliczeniowych. Pierwsze to tradycyjne procesory, gdzie oczywiście wykorzystuje się też dostępne rdzenie obliczeniowe oraz instrukcje typu SIMD[1]. Drugim typem urządzeń są karty graficzne wyposażone w wiele jednostek przetwarzających. Implementacje OpenCL można też spotkać dla układów DSP[2], a czwartym rodzajem urządzeń są układy typu FPGA. Naturalnie, tradycyjne procesory oraz karty graficzne to najbardziej popularne urządzenia, szczególnie w domowych komputerach klasy PC.



Rys. 2.1. Model platformy OpenCL – gospodarz / host to zazwyczaj komputer PC oraz przynajmniej jedno urządzenie obliczeniowe (jeden host może wykorzystywać kilka różnych urządzeń obliczeniowych, którymi zazwyczaj są karty graficzne, choć standard OpenCL może być implementowany także na zwykłych procesorach oraz układach programowalnych FPGA)

Dla urządzeń obliczeniowych OpenCL zakłada się, że w każdym z nich dostępnych jest wiele jednostek obliczeniowych, które dodatkowo posiadają więcej niż jedną jednostkę przetwarzającą informacje.

Głównym zadaniem systemu gospodarza (hosta) jest sterowanie procesem przetwarzania informacji przez podział i zlecanie zadań do wykonania poszczególnym urządzeniom obliczeniowym, co obejmuje:

• przesłanie danych do urządzenia obliczeniowego,

• przesłanie procedur obliczeniowych do urządzenia,

• zalecenie wykonania procedur obliczeniowych,

• odczytanie danych.

System gospodarza ogrywa rolę nadzorcy w procesie przeprowadzania obliczeń. Taki podział pozwala na optymalizację całego procesu, moc obliczeniowa systemu gospodarza nie musi bowiem być tak duża, jak poszczególnych urządzeń obliczeniowych. Choć trzeba pamiętać, że od sprawności hosta zależy, jak szybko będą przekazywane dane do poszczególnych jednostek obliczeniowych.

Platforma obliczeniowa korzysta z dwóch rodzajów programów – jeden dla systemu gospodarza oraz zestaw procedur obliczeniowych nazywanych jądrami obliczeniowym dla urządzeń

obliczeniowych. W dalszej części książki tego typu procedury będą nazywane jądrami bądź procedurami obliczeniowymi, a czasem zostanie użyte słowo **kernel**, aby nie powtarzać słów jądro bądź procedura obliczeniowa.

Procedury obliczeniowe nie mogą być, niestety, zupełnie dowolnymi programami, gdyż standard OpenCL określa postać języka programowania, który został nazwany OpenCL C. Zgodnie ze swoją nazwą język ten jest podobny do języka C i jest częściowo zgodny ze standardem o nazwie C99, choć ma istotne ograniczenia.

Przełożenie kodu procedury obliczeniowej na, czasem bardzo różne, rozwiązania sprzętowe to zadanie dla sterownika OpenCL. System gospodarza może wspierać ten proces, jednak ostateczne wykonanie procedury obliczeniowej to zadanie dla urządzenia obliczeniowego.

2.2. Model pamięci

Procedury obliczeniowe tworzone w ramach OpenCL wykorzystują dostępność wielu rdzeni obliczeniowych, które są zdolne do wykonywania wielu wątków równocześnie. Najlepszym rozwiązaniem jest sytuacja, gdy poszczególne wątki nie muszą się ze sobą komunikować. Jednak OpenCL pozwala też na to, aby jeden zespół wątków komunikował się za pomocą mechanizmu pamięci dzielonej. Stąd też istnieje kilka rodzajów pamięci, a dokładnie cztery, z których korzystają procedury obliczeniowe:

• pamięć globalna (ang. *global memory*),

• pamięć stała (ang. *constant memory*),

• pamięć lokalna (ang. *local memory*),

• pamięć prywatna (ang. *private memory*).

Pamięć globalna to główny rodzaj pamięci, gdzie są odczytywane i zapisywane dane przez procedury obliczeniowe. Pamięć globalna jest też największa – w przypadku powszechnie stosowanych kart graficznych obecne spotykane wielkości to 1GB, 2GB, a czasem i 3GB bądź 4GB pamięci dostępnej dla danych. W zależności od konkretnego rozwiązania sprzętowego pamięć globalna może być wyposażona w system buforowania (ang. *cache*).

Istotnym elementem jest dostęp do pamięci globalnej, który powinien odbywać się w ciągłych blokach, zarówno w przypadku zapisu, jak też odczytu. Współczesne implementacje pamięci oferują bowiem wysoką wydajność, jeśli dostęp do danych odbywa się w postaci ciągłych bloków. Taki dostęp danych w przypadku OpenCL określa się angielskim słowem *coalesced*.

Użycie pamięci globalnej w procedurze obliczeniowej oznacza się za pomocą atrybutu `kernel` lub `__kernel`. Dotyczy to parametrów jądra obliczeniowego (ang. *computation kernel*). Jądro obliczeniowe nie może alokować tego typu pamięci. Przydział pamięci dla jądra obliczeniowego jest wykonywany przez system gospodarza.

Drugim typem pamięci dostępnym dla programisty procedur obliczeniowych jest tzw. **pamięć stała** (ang. *constant memory*). Jest to pamięć przeznaczona tylko do odczytu przez jądro obliczeniowe.

Podobnie jak pamięć globalna, może być buforowana. Przygotowanie danych naturalnie musi zostać zrealizowane przez gospodarza. Dość często wielkość pamięci stałej jest bardzo ograniczona, więc w przypadku większej ilości danych można zastosować pamięć globalną oraz bufor tworzony po stronie gospodarza opatrzyć odpowiednią flagą informującą, iż pamięć będzie tylko odczytywana.

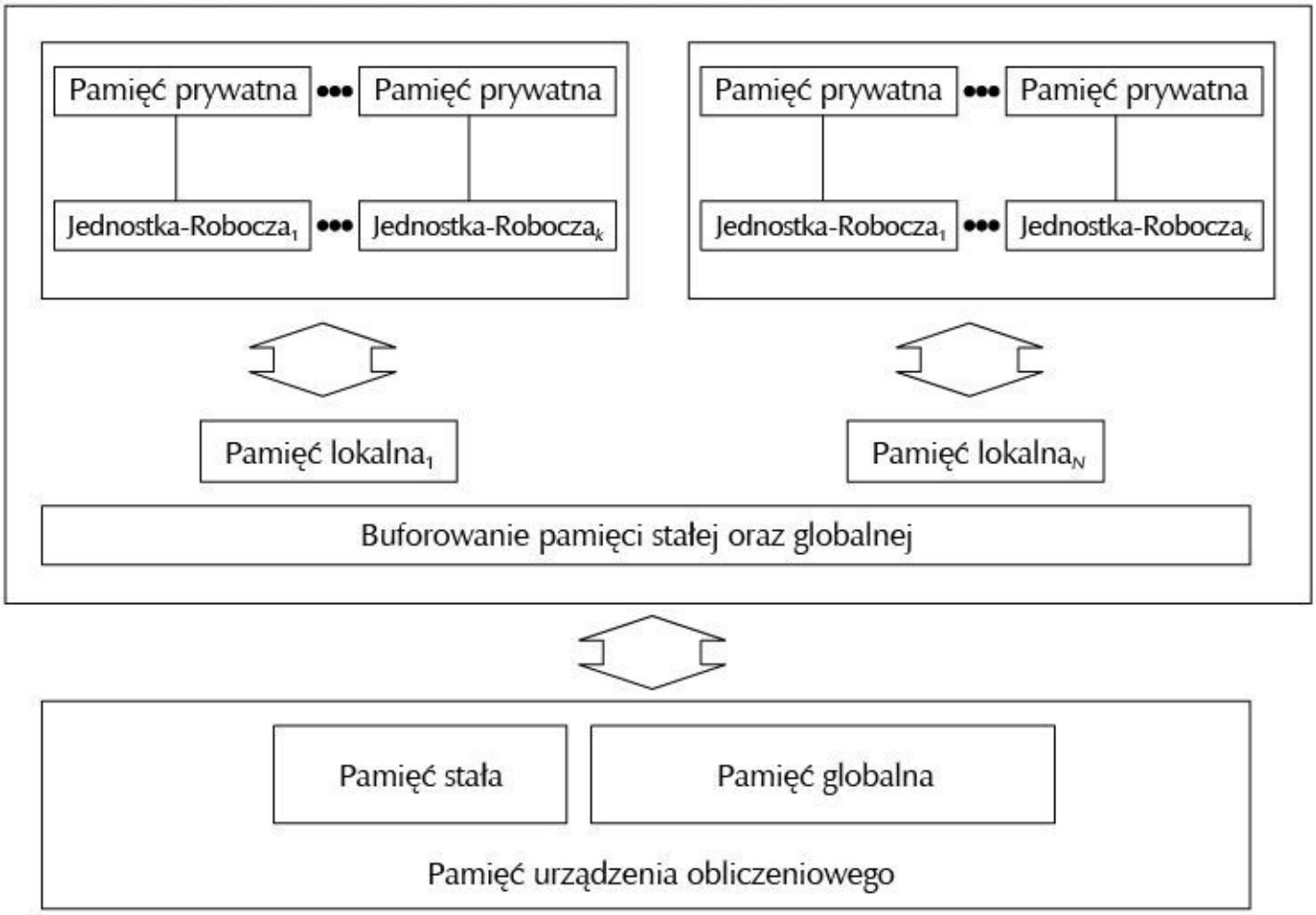
Trzeci typ pamięci – w nomenklaturze OpenCL nazywany **pamięcią lokalną** – można by równie dobrze nazwać **pamięcią dzieloną**. Jest ona wykorzystywana przez grupę roboczą do tego, aby wątki w ramach jednej grupy roboczej mogły się komunikować. Dostęp do pamięci lokalnej jest zazwyczaj znacznie szybszy niż do pamięci globalnej, stąd też prawidłowe wykorzystanie pamięci lokalnej decyduje o końcowej wydajności całej aplikacji. Niestety, w obecnie stosowanych urządzeniach obsługujących standard OpenCL wielkość pamięci dzielonej jest ograniczona do 16, 48, 64 kilobajtów na jedną grupę. Choć nie są to duże wielkości, to jednak podzielenie danych na odpowiednio małe porcje umożliwia osiągniecie znacznej wydajności.

Słowo kluczowe, które pozwala oznaczyć pamięć lokalną to `local` lub `__local`. Choć dostęp do tej pamięci możliwy jest tylko z poziomu jądra obliczeniowego, to po stronie gospodarza można przygotować odpowiednią zmienną i przekazać ją jako argument do jądra obliczeniowego.

Pamięć prywatna jest dostępna tylko dla pojedynczego wątku (jednostki roboczej) i nie jest dzielona z innymi wątkami. Dostęp do tej pamięci jest oczywiście najszybszy, ale zarazem jest to pamięć o najmniejszym rozmiarze. Deklarując w jądrze zmienne obliczeniowe i funkcje pomocnicze, zużywamy pamięć prywatną. Niestety, ze względu na ograniczoną wielkość pamięci prywatnej, przy projektowaniu procedur obliczeniowych należy zwrócić uwagę na to, aby nie przekroczyć dostępnych zasobów pamięci prywatnej.

Tabela 2.1. Typy pamięci i możliwości dostępu ze strony gospodarza oraz procedury obliczeniowej

Poziom	Pamięć globalna	Pamięć stała	Pamięć lokalna (dzielona)	Pamięć prywatna
Gospodarza	Alokacja dynamiczna oraz możliwy zapis oraz odczyt.	Alokacja dynamiczna oraz możliwy zapis oraz odczyt.	Alokacja dynamiczna, brak możliwości zapisu oraz odczytu.	Brak możliwości zapisu oraz odczytu oraz możliwości alokacji.
Jądra obliczeniowego	Możliwy zapis oraz odczyt, brak możliwości dynamicznej alokacji.	Alokacja statyczna, możliwość wyłącznie odczytu.	Alokacja statyczna, możliwy zapis oraz odczyt.	Alokacja statyczna, możliwy zapis oraz odczyt.



Rys. 2.2. Schemat układu pamięci między grupą roboczą (ang. *work-group*) a pojedynczą jednostką obliczeniową (ang. *work-item*) – idea schematu zaczerpnięta ze specyfikacji OpenCL, rozdz. 3.3

Podsumowując, najważniejsze typy pamięci to naturalnie pamięć globalna oraz lokalna. Ogólny podział na cztery rodzaje pamięci pozwala na lepsze wykorzystanie możliwości sprzętowych urządzeń, na których uruchamiane są procedury obliczeniowe. Tabela 2.1 pokazuje dodatkowo możliwości w dostępie do pamięci, gdy używany jest system gospodarza oraz do jakiej pamięci ma dostęp jądro obliczeniowe. Na rysunku 2.2 pokazano raz jeszcze podział pamięci, choć tym razem z perspektywy grupy roboczej oraz pojedynczej jednostki roboczej (ang. *item-work*).

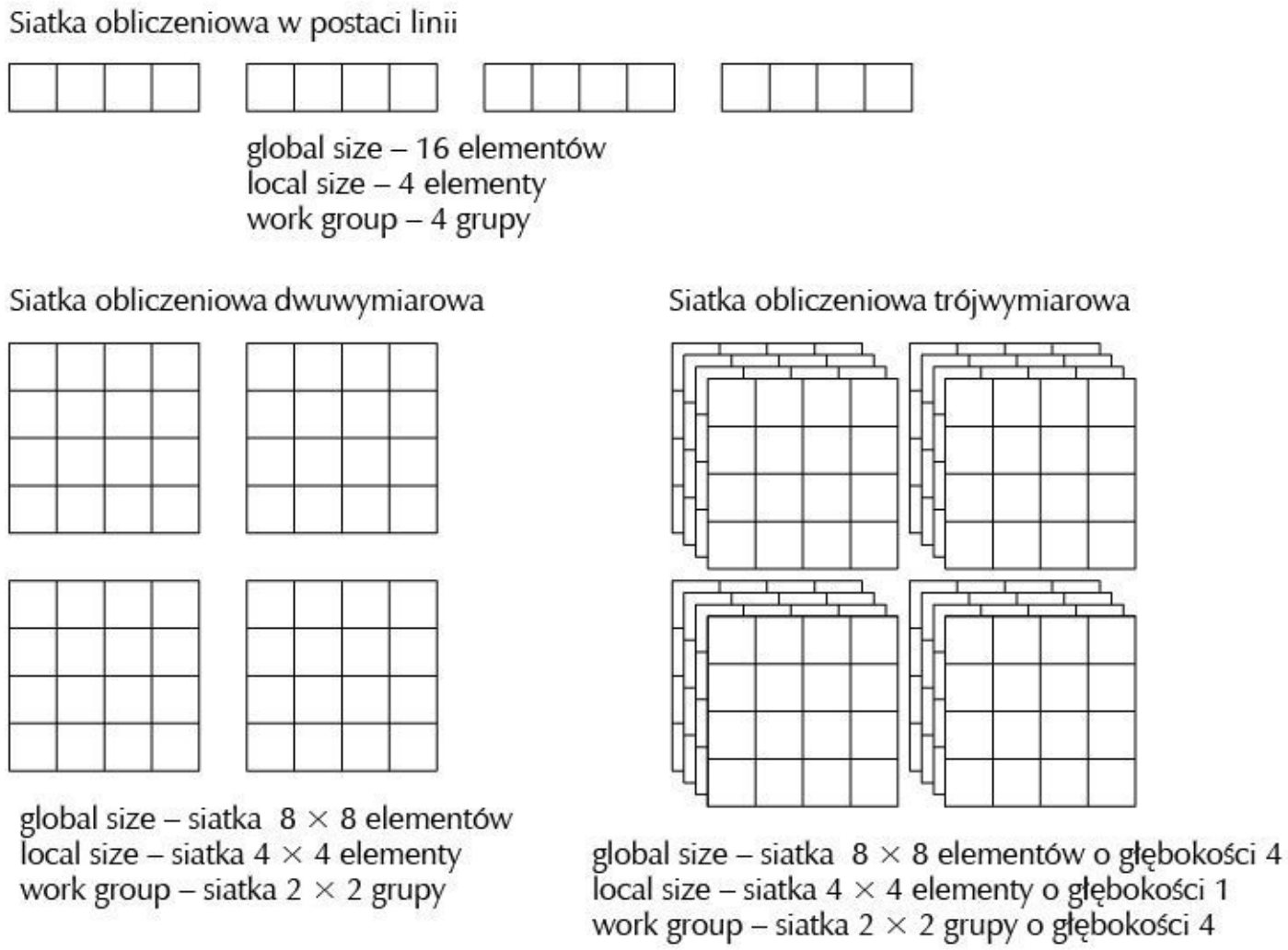
2.3. Model wykonawczy

W tradycyjnym programowaniu, np. w systemie Linux, do dyspozycji programisty jest podstawowy kompilator o nazwie GCC (ang. *GNU Compiler Collection*). Tworząc program, w wielkim uproszczeniu, tworzymy zestaw procedur, klas, a kod źródłowy poddawany jest kompilacji za pomocą kompilatora GCC. Inaczej mówiąc, tworzymy tylko jeden program do wykonania przez procesor główny komputera, na którym jest uruchomiony system.

W przypadku OpenCL jest nieco inaczej, mamy bowiem dwa poziomy wykonawcze. Pierwszy poziom to API biblioteki OpenCL, który pracuje np. na poziomie systemu operacyjnego Linux. Taki system nazywa się **hostem** lub **gospodarzem**, lub **systemem goszczącym**. Za pomocą API OpenCL przygotowujemy dodatkowy nowy program, który jest przekazywany do wykonania na innym urządzeniu obliczeniowym. Inaczej mówiąc, na poziomie hosta następuje przygotowanie tzw. jądra

obliczeniowego bądź też procedury obliczeniowej. System goszczący zleca wykonanie przygotowanej procedury obliczeniowej do urządzenia obliczeniowego OpenCL. Urządzenia obliczeniowe dysponują własnym językiem programowania o nazwie OpenCL C, który jest podobny do języka C. Inaczej przedstawia się także sposób wykonywania programów napisanych dla urządzenia obliczeniowego. Dostępnych jest bowiem wiele jednostek przetwarzających, na których wykonuje się wiele wątków, realizujących obliczenia zlecone przez procedury obliczeniowe.

Głównym elementem, jaki się wykorzystuje w procedurach obliczeniowych, jest tzw. **siatka obliczeniowa** (ang. *computational grid*). Za pomocą siatki określa się sposób podziału danych na poszczególne rdzenie obliczeniowe. W przypadku OpenCL siatka obliczeniowa może mieć strukturę jedno-, dwu- lub trójwymiarową. Struktura jednowymiarowa jest reprezentowana przez wektor, wtedy przybiera kształt linii. Drugi przypadek to struktura dwuwymiarowa siatki obliczeniowej, czyli kształt prostokąta lub kwadratu. Trzecia możliwość to struktura trójwymiarowa, czyli prostopadłościan. Na rysunku 2.3 przedstawiono trzy podstawowe kształty, jakie może przyjąć siatka obliczeniowa.



Rys. 2.3. Trzy podstawowe struktury przestrzeni indeksującej dane (siatki obliczeniowej) dla jąder obliczeniowych OpenCL

2.3.1. Siatka obliczeniowa – NDRange

Siatka obliczeniowa (zamiennie można używać pojęcia przestrzeni indeksującej dane) w OpenCL określana jest skrótem **NDRange**. Jest to kluczowe pojęcie w przypadku procedur obliczeniowych

OpenCL. Jak już to zostało powiedziane wcześniej, mamy trzy możliwe kształty przestrzeni indeksowej czy też indeksującej: wektor, płaszczyzna, sześcian oraz prostopadłościan. Zatem wymiar oznaczony jako N przybiera jedną z następujących wartości: 1, 2, 3.

Przestrzeń indeksowa ma postać hierarchiczną. Najmniejszym elementem jest tzw. **jednostka robocza**, która posiada unikalny identyfikator nazywany **identyfikatorem globalnym**. Zestaw jednostek roboczych tworzy tzw. **grupę roboczą**, której wymiarowość również może przybierać wartości 1, 2, 3. Każda jednostka robocza, oprócz identyfikatora globalnego, posiada także **identyfikator lokalny** w ramach danej grupy roboczej.

Kilka grup roboczych tworzy **siatkę roboczą**. Poszczególne grupy posiadają także własne identyfikatory. Oznacza to, iż jednostka robocza może być identyfikowana przez identyfikator globalny, identyfikator lokalny, a także przez identyfikator grupy roboczej, do której należy.

Kształt siatki obliczeniowej determinuje sposób, w jaki obliczamy poszczególne identyfikatory, co zostało pokazane na rys. 2.4 na przykładzie siatki o dwóch wymiarach. Siatka globalna zbudowana ze 192 jednostek roboczych została podzielona na 12 grup roboczych, a każda grupa robocza obejmuje 16 jednostek roboczych również ułożonych w strukturę dwuwymiarową o wymiarach 4 na 4 elementy.

Poszczególne indeksy, co bardzo ważne, są numerowane od zera. Wartości indeksów globalnych to liczby od zera do jedenastu dla pierwszej współrzędnej (X) oraz od zera do piętnastu dla drugiej współrzędnej. Współrzędne lokalne to liczby od zera do trzech dla obu współrzędnych, każda grupa robocza ma bowiem takie same wymiary. Indeksy dla grup roboczych mają wartości od zera do dwóch dla pierwszej współrzędnej oraz od zera do trzech dla drugiej współrzędnej.

Identyfikatory lokalne (LID) pozwalają na identyfikację jednostki roboczej w ramach grupy. Jednakże identyfikator globalny można obliczyć, znając LID, identyfikator całej grupy roboczej (WGID) oraz wymiar grupy roboczej. Korzystając z tych informacji, identyfikator globalny (GID) dla danej jednostki roboczej można obliczyć w następujący sposób:

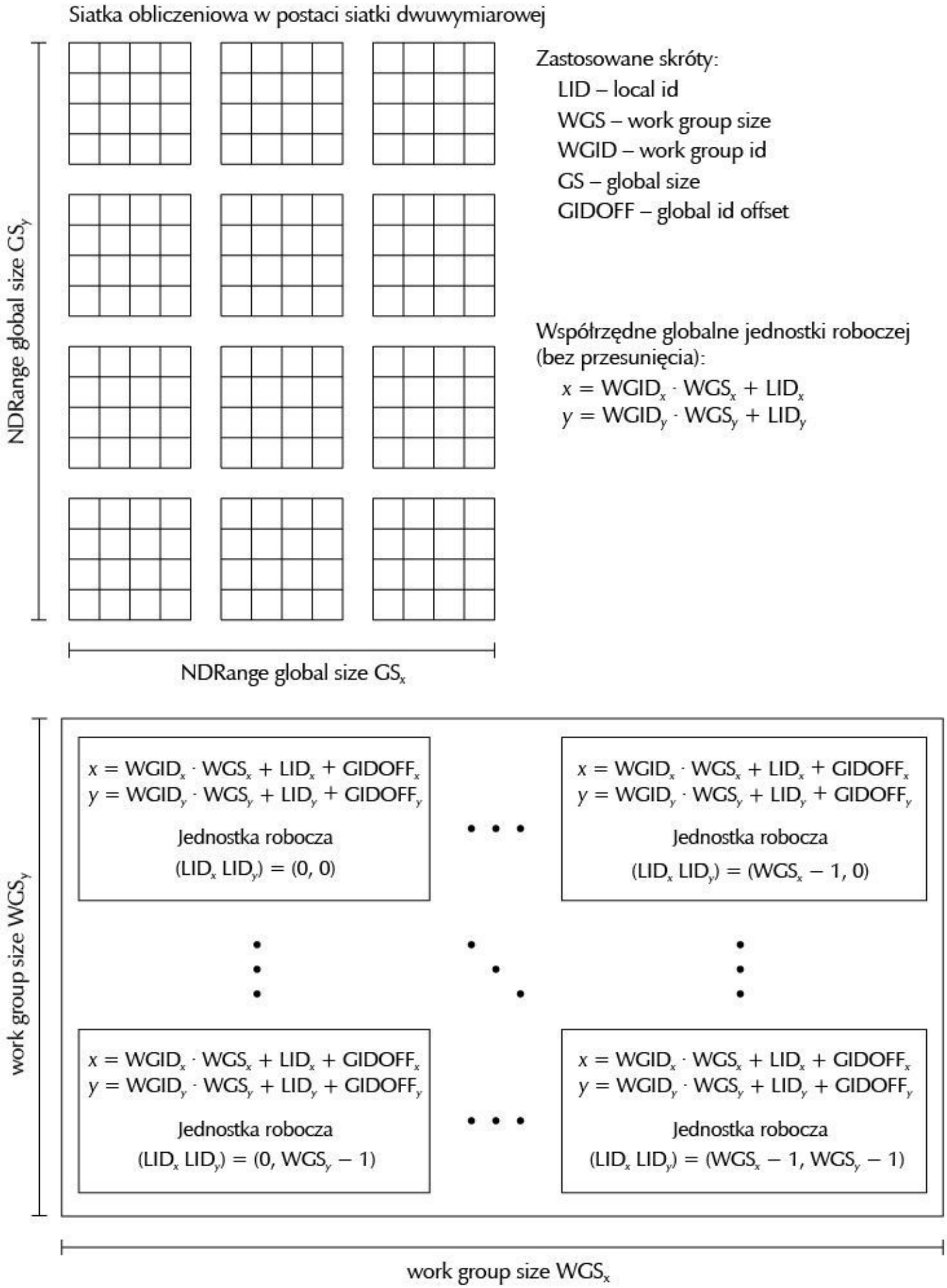
$$GID_x = WGID_x * WGS_x + LID_x$$

$$GID_y = WGID_y * WGS_y + LID_y$$

gdzie: WGID – identyfikator grupy, WGS – wymiar grupy (w przykładzie mamy dwa wymiary, gdyż siatka jest dwuwymiarowa).

W standardzie OpenCL wersja 1.2 można wprowadzać tzw. **offset**, czyli dodatkowe przesunięcie w indeksowaniu poszczególnych elementów.

Ponieważ poszczególne grupy mają takie same wymiary, łatwo obliczyć ich liczbę, dzieląc globalną wielkość przestrzeni indeksującej przez liczbę grup dla danego wymiaru:



Rys. 2.4. Opis przestrzeni indeksującej dla dwuwymiarowej siatki obliczeniowej

$WGN_x = GS_x / WGS_x$

$WGN_y = GS_y / WGS_y$

gdzie: `WGN` – liczba grup dla danego wymiaru.

Obecność lokalnego oraz globalnego identyfikatora umożliwia również obliczenie identyfikatora grupy:

$WGID_x = (GID_x - LID_x) / WGS_x$

$WGID_y = (GID_y - LID_y) / WGS_y$

W OpenCL oraz w podanych w dalszej części książki procedurach obliczeniowych nie ma potrzeby obliczania poszczególnych identyfikatorów osobno, istnieje bowiem kilka funkcji, za pomocą których można odczytać potrzebne informacje:

- `get_global_id(int val)`,
- `get_local_id(int val)`,
- `get_num_groups(int val)`,
- `get_group_size(int val)`,
- `get_group_id(int val)`.

Dokładniejsze informacje o tych funkcjach zostały przedstawione w rozdz. 3.6.

Podstawowe pojęcia omówione powyżej są następujące (będziemy podawać angielską nazwę oraz polski odpowiednik stosowany w dalszej części książki):

- *work-item* (jednostka robocza) – pojedynczy wątek realizujący jedną instancję procedury obliczeniowej,
- *work-group* (grupa robocza) – zespół kilkunastu jednostek roboczych, w ramach jednej grupy poszczególne wątki/jednostki mogą się komunikować za pomocą tzw. pamięci dzielonej,
- *global-id* (identyfikator globalny) – unikatowy numer przydzielony każdej jednostce roboczej,
- *local-id* (identyfikator lokalny) – unikatowy numer przydzielony każdej jednostce roboczej, ale tylko w obrębie grupy roboczej. Inaczej mówiąc, numer ten może zostać powtórzony w innej grupie roboczej.

2.3.2. Kontekst obliczeń oraz kolejka poleceń

Kolejnym pojęciem, jakie pojawia się w standardzie OpenCL, jest tzw. **kontekst obliczeń**. Określa on środowisko pracy OpenCL, obejmując urządzenia obliczeniowe, obiekty reprezentujące jądra obliczeniowe, obiekty odnoszące się do danych umieszczonych w pamięci operacyjnej urządzenia

obliczeniowego, a także skompilowane jądra obliczeniowe, które są programami dla urządzenia obliczeniowego.

Utworzenie kontekstu to zadanie dla API OpenCL wykonywane z poziomu gospodarza. Kontekst zawsze tworzy się w odniesieniu do platformy OpenCL, jaka jest dostępna w ramach systemu OpenCL.

Jednym z ważniejszych zadań kontekstu jest utworzenie **kolejki poleceń**. Sam kontekst obliczeniowy jest wykorzystywany tylko do tworzenia obiektów, natomiast główną funkcją kolejki poleceń jest zlecanie zadań do wykonania przez urządzenia obliczeniowe. Najważniejszym zadaniem jest oczywiście wykonanie jądra obliczeniowego, ale kolejka jest odpowiedzialna również za przesyłanie danych z pamięci gospodarza do urządzenia obliczeniowego i na odwrót. Zadaniem kolejki jest więc koordynacja operacji obliczeniowych między systemem gospodarza oraz systemem urządzeń obliczeniowych OpenCL.

OpenCL pozwala, aby system gospodarza utworzył wiele kolejek do urządzenia lub urządzeń obliczeniowych, np. do realizacji różnych zadań. Wykorzystując system zdarzeń dostępny w OpenCL można np. po zakończeniu zadania kopiowania danych od razu wywołać jądro obliczeniowe, w obsłudze zdarzenia, poza głównym nurtem programu, jakie wykonuje aktualnie host. Należy jednak podkreślić, że jądro obliczeniowe musi odwoływać się do tego samego kontekstu, co dana kolejka poleceń, której zlecono wykonanie jądra obliczeniowego.

2.4. Model programowania

Model programowania w OpenCL został podzielony na dwa modele: model dla danych oraz model dla zadań. W tej książce oraz wszędzie tam, gdzie omawia się procedury obliczeniowe OpenCL, stosowany jest model dla danych, w którym siatka obliczeniowa adresuje dane.

Siatka obliczeniowa wskazuje strukturę przetwarzania, bowiem (jak już to zostało pokazane w rozdz. 2.3) hierarchia siatki wyróżnia jednostki robocze jako najmniejsze elementy przetwarzania informacji. Zazwyczaj jednostka robocza zajmuje się jednym konkretnym obiektem umieszczonym w pamięci urządzenia, np. jedną liczbą. Jednakże nie zawsze musi tak być, jednostka robocza może także przetwarzać kilka elementów pobranych z pamięci, choć na poziomie siatki obliczeniowej stanowi najmniejszy element przetwarzania informacji.

Zespół jednostek roboczych tworzy grupę roboczą. W grupie wszystkie jednostki robocze działają równolegle. Inaczej mówiąc, grupa równolegle przetwarza zestaw obiektów pobranych z pamięci urządzenia obliczeniowego. Do zadań programisty należy ustalenie rozmiaru pojedynczej grupy roboczej oraz liczby grup roboczych wymaganych do rozwiązywanego zadania.

Istotnym elementem w modelu danych jest synchronizacja. Idealny przypadek to taki, w którym dane w ramach pojedynczych jednostek są w pełni niezależne i nie ma potrzeby synchronizacji. W rozwiązywaniu rzeczywistych problemów sytuacja przedstawia się inaczej, dlatego OpenCL dostarcza dwa poziomy synchronizacji. Pierwszy poziom odnosi się do grupy roboczej – można zastosować tzw. barierę, która wstrzyma pracę poszczególnych jednostek roboczych do czasu, aż wszystkie jednostki w danej grupie do niej dotrą. Jak na razie (w wersji o numerach 1.0, 1.1 oraz 1.2) standard OpenCL nie dostarcza mechanizmu do synchronizacji między różnymi grupami roboczymi.

Drugi poziom synchronizacji odnosi się do kolejki poleceń. Postawiona bariera na tym poziomie oznacza, iż wszystkie polecenia umieszczone w kolejce muszą zakończyć swoje działania, a także wszystkie operacje wykonywane na pamięci również muszą zostać zakończone.

Pewną odmianą synchronizacji na drugim poziomie są zdarzenia OpenCL. Dostępne API pozwala na tworzenie zdarzeń, które będą wywoływane po realizacji pewnych zadań, np. po zakończeniu operacji kopiowania danych z pamięci urządzenia do pamięci hosta. Gwarantuje to spójność danych, ale pozwala też na pracę asynchroniczną, bowiem system gospodarza nie jest blokowany przez pracę urządzenia obliczeniowego OpenCL.

Drugim modelem programowania jest **model zadaniowy**. Jest to procedura obliczeniowa wykonywana przez urządzenie obliczeniowe OpenCL, która nie korzysta z siatki obliczeniowej. Inaczej mówiąc, jądro obliczeniowe wykorzystuje w ramach grupy roboczej tylko jedną jednostkę roboczą. Przy tym nadal można wykonywać obliczenia równoległe, można bowiem zlecić uruchomienie wielu zadań, mimo iż nie można skorzystać z siatki obliczeniowej.

Przypisy

[1] SIMD – skrót od ang. *Single Instruction Multiply Data*, tj. pojedyncza instrukcja wiele danych. Oznacza to, że za pomocą jednej instrukcji wykonujemy tę samą operację na zestawie danych, np. dodajemy wartość skalarną (liczbę) do poszczególnych elementów w wektorze.

[2] DSP – skrót od *Digital Signal Processor*, inaczej procesor sygnałowy. Zazwyczaj jest to oddzielny układ, który realizuje przetwarzanie sygnałów cyfrowych. Sygnał taki może być opisany za pomocą liczby całkowitej lub też zmienno- bądź stałoprzecinkowej wartości.

3 Język programowania dostępny w OpenCL

Tworzenie procedury obliczeniowej albo jądra obliczeniowego (ang. *computation kernel*) w ramach standardu OpenCL jest realizowane w języku programowania, który został oparty na standardzie ISO języka C, a dokładnie jest to standard C99 oznaczony ISO/IEC 9899:1999.

Co prawda nie mamy do dyspozycji całego języka C wraz z biblioteką funkcji standardowych, jednakże, tworząc procedury obliczeniowe, możemy dość swobodnie posługiwać się wieloma elementami języka C. Obecny jest też preprocesor. Choć są także istotne ograniczenia, np. nie ma możliwości budowy funkcji rekurencyjnych. Dodatkowo, jak na razie, nie ma dostępu choćby do wybranych elementów języka C++, jak przeciążanie operatorów czy wzorców. W OpenCL istnieją też dodatkowe konstrukcje, jakie nie występują w standardowym języku C.

W tym rozdziale przedstawiono odmianę języka C stosowaną w ramach standardu OpenCL, a w szczególności dodatkowe elementy dostępne do tworzenia procedur obliczeniowych. Wiele informacji o podstawowych typach można także odszukać w oficjalnym opisie standardu OpenCL, dostępnym na stronie organizacji Khronos.

Istnieje kilka standardów OpenCL, jakie można spotkać w dostępnych urządzeniach współpracujących z OpenCL. Są to wersje o numerach 1.0, 1.1, 1.2. Standardy 1.1 oraz 1.2 są najbardziej popularne. Opracowany został też standard 2.0, choć jak na razie (rok 2014) nie ma jeszcze urządzeń, które go obsługują.

Informacje podane w dalszej części książki zostały tak przygotowane, aby były jak najbardziej uniwersalne, mimo że za bazę został wybrany standard 1.2. Jeśli omawiane zagadnienie dotyczy jakieś szczególnej wersji standardu OpenCL, zostało to wyraźnie podkreślone.

3.1. Typy danych

OpenCL oferuje kilka rodzajów typów danych – podstawowe to typy skalarne oraz wektorowe. Jednakże są także dostępne dodatkowe typy do obsługi obrazów, które pozwalają w dość prosty sposób wdrażać operacje związane z przetwarzaniem grafiki.

3.1.1. Typy podstawowe (typy skalarne)

W każdym języku programowania podstawowym elementem są typy danych. Tak samo jest w odmianie języka C stosowanej do tworzenia procedur obliczeniowych OpenCL (w dalszej części książki stosowane będzie wyrażenie OpenCL C do oznaczenia tej odmiany języka C).

Tabela 3.1. Typy skalarne dostępne w OpenCL

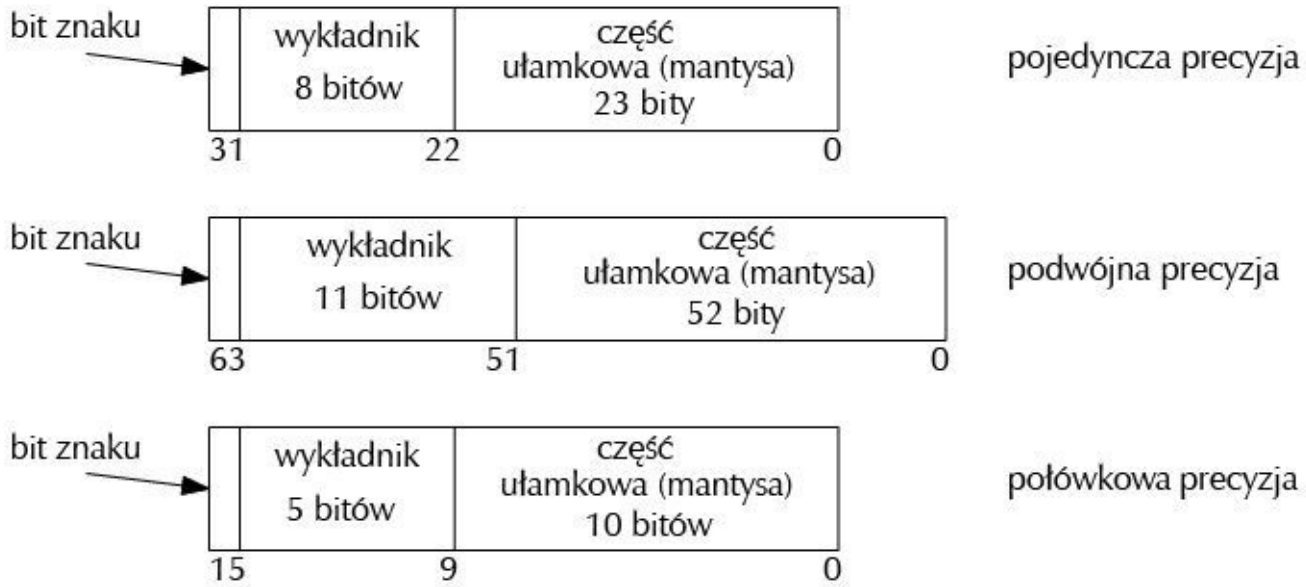
Typ	Krótki opis

`bool`	Wartość logiczna, prawda (true) lub fałsz (false), zakłada się, że zero oznacza fałsz, a jedność prawdę.
`char`	Liczba 8-bitowa ze znakiem.
`unsigned char, uchar`	Liczba 8-bitowa bez znaku.
`short`	Liczba 16-bitowa ze znakiem.
`unsigned short, ushort`	Liczba 16-bitowa bez znaku.
`int`	Liczba 32-bitowa ze znakiem.
`unsigned int, uint`	Liczba 32-bitowa bez znaku.
`long`	Liczba 64-bitowa ze znakiem.
`unsigned long, ulong`	Liczba 64-bitowa bez znaku.
`float`	Zmiennoprzecinkowa 32-bitowa liczba pojedynczej precyzji w standardzie IEEE 754.
`double`	Zmiennoprzecinkowa 64-bitowa liczba podwójnej precyzji w standardzie IEEE 754.
`half`	Zmiennoprzecinkowa 16-bitowa liczba w standardzie IEEE 754-2008.
	Liczba całkowita bez znaku, jest to typ zwracany przez operator `sizeof`. W przypadku, gdy urządzenie obliczeniowe OpenCL stosuje 32-bitowe adresowanie liczba ta jest liczbą 32-bitową, a 64-bitową, jeśli stosowane jest adresowanie 64-

`size_t`	bitowe.
`ptrdiff_t`	Liczba całkowita ze znakiem, typ reprezentuje rezultat różnicy między dwoma wskaźnikami. Wartość jest 32-bitowa, jeśli urządzenie OpenCL stosuje 32-bitowe adresowanie, lub, analogicznie jak w przypadku `size_t`, 64-bitowa dla adresowania 64-bitowego.
`intptr_t`	Liczba całkowita ze znakiem, która reprezentuje wskaźnik typu `void`, inaczej mówiąc dowolny wskaźnik typu `void` może zostać skonwertowany do typu `intptr_t`. Ponowna konwersja do typu `void` zachowa oryginalną wartość wskaźnika.
`uintptr_t`	Identycznie jak dla typu `intptr_t`, przy czym wskaźnik jest zamieniany na liczbę całkowitą ze znakiem.
`void`	Typ pusty, oznaczający pusty zbiór wartości. Jest też interpretowany jako typ niekompletny, który nie może zostać uzupełniony.

W tabeli 3.1 wymieniono wszystkie tzw. typy skalarne danych (czyli pojedyncze wartości logiczne, znakowe, liczbowe) dostępne w OpenCL C. Większość oznaczeń jest zgodnych ze standardem języka C. Występują też rzadko spotykane typy danych, takie jak `half`, reprezentujący 16-bitową liczbę zmiennoprzecinkową (w dalszej części typ ten będzie nazywany typem o ograniczonej precyzji lub typem o precyzji połówkowej). Natomiast typy zwierające sekwencję `ptr` odnoszą się do wskaźników i reprezentują liczby całkowite, które są interpretowane jako adresy. Mogą to być wartości 32- lub 64-bitowe w zależności od wartości stałej `CL_DEVICE_ADDRESS_BITS`[1].

Większość wymienionych w tab. 3.1 typów to typy spotykane w tradycyjnym języku C (na rys. 3.1 schematycznie przedstawiono budowę liczb zmiennoprzecinkowych dostępnych w OpenCL C).



Rys. 3.1. Graficzna ilustracja typów liczb zmiennoprzecinkowych dostępnych w OpenCL

Wyjątkiem jest typ `half`. Jest to 16-bitowa zmiennoprzecinkowa liczba w formacie IEEE 754-2008. Liczby `half` mają bit znaku, wykładnik oraz mantysę identycznie jak typy `float` czy `double`. Są to liczby opisane za pomocą tylko szesnastu bitów, więc na wykładnik przydzielono pięć bitów („exponent bias” równy 15), a mantysa jest 10-bitowa. Analogicznie jak większe odpowiedniki, liczby typu `half` muszą reprezentować liczby w postaci normalnej, denormalnej, oferować symbol nieskończoności oraz symbol NaN, oznaczający, że wartość nie jest liczbą.

Standard OpenCL nie wymaga obecności typu `half` w danym urządzeniu obliczeniowym. Dodatkowo używanie typu `half` jest obłożone pewnymi restrykcjami, choć można zadeklarować parametr danej funkcji za pomocą typu `half`:

```
void fnc1(__global half *ptrToData) { . . . }
```

Deklarowany wskaźnik musi być opatrzony słowem `__global`:

```
__global half *offsetToData;

offsetToData = ptrToData+ 1024;
```

żeby można było na nim wykonywać typowe operacje na wskaźnikach.

Nie można utworzyć zmiennej czy też tablicy w pamięci lokalnej w oparciu o typy `half`. Odczyt danych wymaga stosowania funkcji `vload_half`. Zapis również wymaga stosowania oddzielnej funkcji `vstore_half`. Ograniczenia dla tego typu wynikają z jego głównego przeznaczenia do danych graficznych, np. dla tekstur używanych do generacji grafiki. Krótko mówiąc, nie jest to typ szeroko wykorzystywany w obliczeniach numerycznych. Łatwo to zauważyć w następującym zestawieniu wartości minimalnych i maksymalnych dla podstawowych typów zmiennoprzecinkowych dostępnych w OpenCL C:

• typ `half` – liczba o połówkowej precyzji, wartość najmniejsza $2^{-14} \approx 6.10 \times 10^{-5}$, wartość największa $2^{15} \times (2 - 2^{-10}) = 65504$.

• typ `float` – liczba pojedynczej precyzji, wartość najmniejsza $2^{-126} \approx 1.18 \times 10^{-38}$, wartość największa $2^{127} \times (2 - 2^{-23}) \approx 3.40 \times 10^{38}$.

• typ `double` – liczba podwójnej precyzji, wartość najmniejsza $2^{-1022} \approx 1.00 \times 10^{323.3}$, wartość największa $2^{1023} \times (2 - 2^{-52}) \approx 1.00 \times 10^{308.3}$.

Niektóre z wymienionych typów skalarnych OpenCL mają swoje bezpośrednie odpowiedniki w nazewnictwie typów dla API OpenCL stosowane po stronie gospodarza. W tabeli 3.2 przedstawiono zestawienie typów OpenCL oraz ich odpowiedników dostępnych w API OpenCL.

Należy pamiętać też o tym, że OpenCL jest standardem uniwersalnym, więc można napotkać na inny porządek bitów dla typów skalarnych systemu gospodarza oraz inny dla urządzeń obliczeniowych OpenCL. W dwóch podstawowych typach porządku bitów bit najmniej znaczący jest umieszczony po prawej (typ `little-endian`) lub po lewej stronie (typ `big-endian`). Kopiując dane do urządzenia obliczeniowego, należy odpowiednio zmienić porządek bitów i podobnie zmienić dane po odczytaniu danych z urządzenia obliczeniowego.

Z poziomu API OpenCL możemy odczytać sposób układu bitów za pomocą funkcji `clGetDeviceInfo` i stałej `CL_DEVICE_ENDIAN_LITTLE` (w rozdz. 4.1 podano przykład odczytujący tę informację). Odczytanie wartości `CL_TRUE` oznacza, że urządzenie ma porządek typu `little-endian`.

Również w przypadku kodu dla urządzenia obliczeniowego można sprawdzić, jaki obowiązuje porządek bitów dla danych. Zdefiniowane zostało makro `__ENDIAN_LITTLE__`, którego obecność oznacza, że mamy porządek `little-endian`, natomiast brak definicji tego makra oznacza porządek `big-endian`.

Używanie typu `double` również wymaga dodania odpowiedniej **definicji preprocesora**, tzw. **pragmy**, w kodzie jądra obliczeniowego:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
```

Tabela 3.2. Tabela odpowiedników typów OpenCL C oraz API OpenCL

Typ OpenCL C	Typ API OpenCL
`bool`	brak odpowiednika
`char`	`cl_char`
`unsigned char`	`cl_uchar`
`short`	`cl_short`
`unsigned short`	`cl_ushort`
`int`	`cl_int`
`unsigned int`	`cl_uint`
`long`	`cl_long`

`unsigned long`	`cl_ulong`
`float`	`cl_float`
`double`	`cl_double`
`half`	`cl_half`
`size_t`	brak odpowiednika
`ptrdiff_t`	brak odpowiednika
`intptr_t`	brak odpowiednika
`uintptr_t`	brak odpowiednika
`void`	`void`

Definicja ta włączy w momencie kompilacji jądra obliczeniowego obsługę liczb o podwójnej precyzji. Można także wprowadzić odpowiednie makro podczas kompilacji za pomocą `clBuildProgram`, np. `__DOUBLE_TYPE_SUPPORTED__` i w jądrze obliczeniowym za pomocą `#ifdef` możliwa jest warunkowa kompilacja tego typu danych. Ogólnie liczby typu `double` są traktowane jako rozszerzenie, dlatego, że pierwsze karty graficzne obsługiwały tylko liczby typu `float`, a obsługa liczby podwójnej precyzji pojawiła się później. Więcej informacji na temat definicji `#pragma` podano w rozdz. 3.7.

3.1.2. Typy wektorowe

Dodatkiem do typów skalarnych obsługiwanych przez OpenCL są typy wektorowe oparte na podstawowych typach skalarnych. Analogicznie po stronie API OpenCL przygotowano odpowiedni typ dla łatwiejszej komunikacji między urządzeniem obliczeniowym a systemem gospodarza.

Typy wektorowe, jak pokazano to w tab. 3.3, są określone dla wszystkich podstawowych typów skalarnych, różnią się wielkością, choć długość wektora jest ściśle określona i może przybierać wartości 2, 3, 4, 8, 16. Jednak nie każde urządzenie musi realizować działania na podanych

długościach wektora bezpośrednio na poziomie sprzętowym. W trakcie kompilacji jądra obliczeniowego zostanie wygenerowany odpowiedni kod, który zapewni poprawność działań na wektorze o większej długości niż długość obsługiwana przez urządzenie obliczeniowe. Taka operacja może trwać dłużej, dlatego warto sprawdzić, jakie są preferowane długości wektora, odczytując odpowiednie informacje za pomocą funkcji `clGetDeviceInfo` oraz stałej, np. `CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT`, aby otrzymać preferowaną wielkość wektora dla typu `float`. W rozdziale 4.1 omówiono przykład odczytywania informacji o możliwościach obliczeniowych, w tym także o wielkości wektorów. Obecnie (rok 2014) wektor typu `float` ma zazwyczaj cztery elementy, co oznacza, że cały wektor jest reprezentowany przez 128 bitów. Zastosowanie typu `double` ogranicza długość wektora do dwóch elementów. Łatwo wywnioskować, iż po stronie sprzętowej typy wektorowe mają długość do 128 bitów.

Tabela 3.3. Typy wektorowe dostępne w OpenCL oraz ich odpowiedniki w API OpenCL (dopuszczalne wartości dla N to 2, 3, 4, 8, 16)

Typ OpenCL	Odpowiednik API	Krótki opis
`charN`	`cl_charN`	Wektor zawierający N elementów typu liczba 8-bitowa ze znakiem.
`ucharN`	`cl_ucharN`	Wektor zawierający N elementów typu liczba 8-bitowa bez znaku.
`shortN`	`cl_shortN`	Wektor zawierający N elementów typu liczba 16-bitowa ze znakiem.
`ushortN`	`cl_ushortN`	Wektor zawierający N elementów typu liczba 16-bitowa bez znaku.
`intN`	`cl_intN`	Wektor zawierający N elementów typu liczba 32-bitowa ze znakiem.
`uintN`	`cl_uintN`	Wektor zawierający N elementów typu liczba 32-bitowa bez znaku.
`longN`	`cl_longN`	Wektor zawierający N elementów typu liczba 64-bitowa ze znakiem.
`ulongN`	`cl_ulongN`	Wektor zawierający N elementów typu liczba 64-bitowa bez znaku.

`floatN`	`cl_floatN`	Wektor zawierający N elementów typu zmiennoprzecinkowa liczba 32-bitowa pojedynczej precyzji.
`double`	`cl_doubleN`	Wektor zawierający N elementów typu zmiennoprzecinkowa liczba 64-bitowa podwójnej precyzji.

Dla łatwiejszej obsługi typów wektorowych dodano zestaw operatorów oraz funkcji, które zostały omówione w rozdz. 3.4.

3.1.3. Typy do obsługi obrazów

W tabeli 3.4 przedstawiono podstawowe typy danych do przetwarzania obrazów jednowymiarowych, dwuwymiarowych oraz trójwymiarowych. Należy podkreślić, że urządzenia obliczeniowe OpenCL nie muszą obsługiwać tego typu danych, dlatego należy się upewnić (stała `CL_DEVICE_IMAGE_SUPPORT`), że urządzenie pozwala na bezpośrednią obsługę obrazów.

OpenCL dostarcza też do obsługi obrazów szereg funkcji pomocniczych do odczytu czy zapisu danych. Więcej informacji na ten temat znajduje się w przy omawianiu przykładów związanych z przetwarzaniem obrazów (rozdz. 5.10).

W nowym planowanym standardzie OpenCL 2.0 dostępne będą także nowe typy danych dla obrazów w trybie multi-samplingu oraz obrazów zawierających tylko dane o głębokości (mapy głębokości). Ten ostatni typ to obraz w skali szarości, gdzie np. kolor biały oznacza obiekt znajdujący się najdalej od kamery, a kolor czarny obiekt znajdujący się najbliżej kamery.

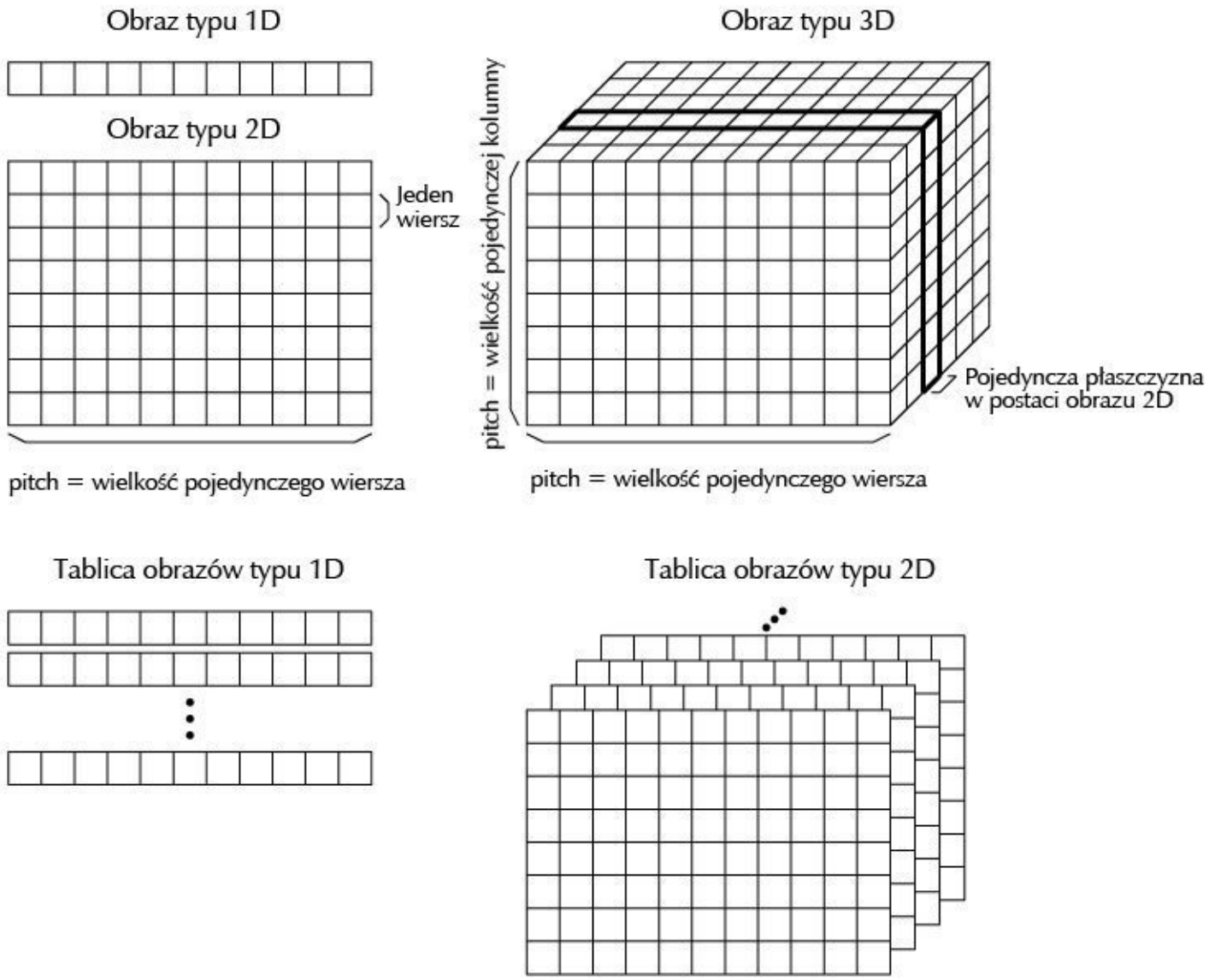
Tabela 3.4. Typy związane z obsługą obrazów 1D, 2D oraz 3D

Typ OpenCL	Opis
`image2d_t`	Reprezentuje obraz dwuwymiarowy.
`image3d_t`	Reprezentuje obraz trójwymiarowy.
`image2d_array_t`	Tablica do reprezentacji obrazu 2D.
`image1d_t`	Reprezentuje obraz jednowymiarowy.

`image1d_buffer_t` Obraz 1D utworzony z obiektu typu bufor.

`image1d_array_t` Tablica do reprezentacji obrazu 1D.

`sampler_t` Obiekt przeznaczony do określania sposobu odczytu danych z obrazu.



Rys. 3.2. Graficzna ilustracja formatów graficznych obsługiwanych przez OpenCL w standardzie 1.2

Tabela 3.5. Nowe typy danych dostępne w standardzie OpenCL

Typ OpenCL	Opis

`image2d_depth_t`	Reprezentuje dane o głębokości obrazu 2D.
`image2d_array_depth_t`	Tablica zawierająca głębokości kilku obrazów 2D.
`image2d_msaa_t`	Dla obrazu 2D z kolorem w trybie multi-samplingu. Typ będzie dostępny, jeśli będzie dostępne rozszerzenie `cl_khr_gl_msaa_sharing`.
`image2d_array_msaa_t`	Tablica złożona z obrazów 2D z kolorem w trybie multi-samplingu. Typ będzie dostępny, jeśli będzie dostępne rozszerzenie `cl_khr_gl_msaa_sharing`.
`image2d_msaa_depth_t`	Dane o głębokości dla obrazu 2D w trybie multi-samplingu. Typ będzie dostępny, jeśli będzie dostępne rozszerzenie `cl_khr_gl_msaa_sharing`.
`image2d_array_msaa_depth_t`	Tablica zawierająca dane o głębokości kilku obrazów 2D w trybie multi-samplingu. Typ będzie dostępny, jeśli będzie dostępne rozszerzenie `cl_khr_gl_msaa_sharing`.

3.1.4. Inne typy oraz nazwy zarezerwowane

W tabeli 3.6 podano nazwy dodatkowych typów, jakie można spotkać w OpenCL. Pierwszy typ `event_t` jest już obecny w API OpenCL, a pozostałe nazwy typów są tylko zarezerwowane. W przyszłości będzie można wykorzystać nowe typy liczbowe do tworzenia oprogramowania.

Tabela 3.6. Typ zdarzeniowy oraz nazwy zarezerwowane innych typów

Typ OpenCL	**Opis**
`event_t`	Reprezentuje zdarzenie, np. pojawiające się po zakończeniu procesu kopiowania danych z pamięci urządzenia do pamięci gospodarza.
	Nazwy zarezerwowane
`boolN`	Wektorowy o elementach logicznych.

`halfN`	Wektorowy o zmiennoprzecinkowych liczbach 16-bitowych
`quad`, `quadN`	Skalarny oraz wektorowy oparty na typie zmiennoprzecinkowym o długości 128 bitów.
`complex half`, `complex halfN`	Skalarny oraz wektorowy dla liczb zespolonych opartych na typie liczby zmiennoprzecinkowej o długości 16 bitów.
`imaginary half`, `imaginary halfN`	Wartości urojone skalarne lub wektorowe oparte na typie liczby zmiennoprzecinkowej o długości 16 bitów.
`complex float`, `complex floatN`	Skalarny oraz wektorowy dla liczb zespolonych opartych na typie liczby zmiennoprzecinkowej o długości 32 bitów.
`imaginary float`, `imaginary floatN`	Wartości urojone skalarne lub wektorowe oparte na typie liczby zmiennoprzecinkowej o długości 32 bitów.
`complex double`, `complex doubleN`	Skalarny oraz wektorowy dla liczb zespolonych opartych na typie liczby zmiennoprzecinkowej o długości 64 bitów.
`imaginary double`, `imaginary doubleN`	Wartości urojone skalarne lub wektorowe oparte na typie liczby zmiennoprzecinkowej o długości 64 bitów.
`complex quad`, `complex quadN`	Skalarny oraz wektorowy dla liczb zespolonych opartych na typie liczby zmiennoprzecinkowej o długości 128 bitów.
`imaginary quad`, `imaginary quadN`	Wartości urojone skalarne lub wektorowe oparte na typie liczby zmiennoprzecinkowej o długości 128 bitów.

`float NxM`	Macierz o elementach typu float zapamiętana w postaci kolumnowej.
`double NxM`	Macierz o elementach typu double zapamiętana w postaci kolumnowej.
`long double`, `long doubleN`	Skalarny oraz wektorowy dla liczb zespolonych opartych na typie liczby zmiennoprzecinkowej o dokładności większej niż `double`, ale mniejszej niż `quad`.
`long long`, `long longN`	Skalarny oraz wektorowy dla liczb całkowitych ze znakiem o długości 128 bitów.
`unsigned long long`, `ulong long`, `ulong longN`	Typ skalarny oraz wektorowy dla liczb całkowitych bez znaku o długości 128 bitów.

Nowa wersja OpenCL 2.0 przewiduje jeszcze kilkanaście dodatkowych typów, które zostały przedstawione w tab. 3.7. Dostępny jest np. typ `queue_t`, odnoszący się do kolejki poleceń OpenCL, oraz `range` dostarczający opis siatki obliczeniowej. Nowa odsłona standardu OpenCL będzie umożliwiać wywoływanie jąder obliczeniowych z własnymi parametrami już na poziomie urządzenia obliczeniowego.

Tabela 3.7. Wybrane nowe typy danych, jakie będą dostępne w standardzie OpenCL 2.0

Typ OpenCL	Opis
`queue_t`	Reprezentuje kolejkę poleceń; zadania mogą być zlecane tylko przez jądro, które używa tej kolejki.
`ndrange_t`	Siatka obliczeniowa używana przez jądro obliczeniowe.
`clk_event_t`	Zdarzenie urządzenia, które potwierdza, iż polecenie zostało przekazane do kolejki poleceń.
`reserve_id_t`	Zarezerwowany do deklaracji potoku, gdzie można będzie zapisywać oraz odczytywać dane.

`cl_mem_fence_flags`	Opisuje rodzaj bariery; jego wartość może być kombinacją następujących flag: `CLK_GLOBAL_MEM_FENCE`, `CLK_LOCAL_MEM_FENCE`, `CLK_IMAGE_MEM_FENCE`.

3.2. Dostępne operatory

W tabeli 3.8 przedstawiono zestaw operatorów obsługiwanych przez język OpenCL C. Podane operatory obsługują nie tylko dane typu skalarnego, lecz także typu wektorowego (więcej na ten temat w rozdz. 3.5). Możliwe jest też łączenie typów, np. w celu sprawdzenia, czy poszczególne składniki wektora są większe niż zadany skalar. Więcej informacji na temat operatorów w kontekście typów wektorowych podano w rozdz. 3.5.2.

Tabela 3.8. Operatory dostępne w standardzie OpenCL

Operator	Opis	Operator	Opis
`+,-,*,/`	Podstawowe operatory arytmetyczne.	`||`	Operator logiczny „lub”.
`%`	Reszta z dzielenia (modulo).	`!`	Operator logiczny „negacja”.
`++`	Operator inkrementacji.	`&`	Operacja bitowa „i”.
`--`	Operator dekrementacji.	`|`	Operacja bitowa „lub”.
`!=, ==`	Operatory: nierówny, równy.	`^`	Operacja bitowa alternatywa wykluczająca (xor).
`<, >`	Operatory: mniejszy, większy.	`~`	Operacja bitowa „negacja”.
`<=, >=`	Operatory: mniejszy lub równy, większy lub równy.	`>>, <<`	Przesunięcie bitowe w prawo, przesunięcie bitowe w lewo.
`&&`	Operator logiczny „i”.	`? :`	Instrukcja wyboru warunkowego.

3.3. Funkcje wbudowane

Odmiana języka C dostępna w OpenCL oferuje dla wygody programisty obszerny zestaw funkcji wbudowanych realizujących różne funkcje matematyczne zgodnie ze standardem języka C99. Przy czym, w odróżnieniu od C99, nie trzeba tych funkcji dołączać do programu za pomocą pliku nagłówkowego `math.h`, bowiem w OpenCL C ten zestaw funkcji został wbudowany w język stosowany na poziomie procedur obliczeniowych.

Funkcje wymienione w tab. 3.9 obsługują typy danych dostępne w ramach procedur obliczeniowych. Dopuszczalny jest typ skalarny `float` oraz `double`, jeśli jest obsługiwany, a także typy wektorowe, jak np. `float2`, `float4`, `double2`. W takim przypadku funkcje te działają na poszczególnych komponentach wektora.

Tabela 3.9. Funkcje matematyczne dostępne w OpenCL C

Funkcja	Opis
`T acos(T x);`	Oblicza wartości funkcji `arc cos` dla argumentu x.
`T acosh(T x);`	Oblicza wartość odwrotności hiperbolicznej funkcji `cos` dla argumentu x.
`T acospi(T x);`	Oblicza wartości funkcji `arc cos` argumentu x, a wynik dzieli przez liczbę π.
`T asin(T x);`	Oblicza wartości funkcji `arc sin` argumentu x.
`T asinh(T x);`	Oblicza wartość odwrotności hiperbolicznej funkcji `sin` dla argumentu x.
`T asinpi(T x);`	Oblicza wartości funkcji `arc sin` argumentu x, a wynik dzieli przez liczbę π.
`T atan(T x);`	Oblicza wartości funkcji `arc tan` argumentu x.
`T atan2(T x, T y);`	Oblicza wartości funkcji `arc tan` argumentu x / y.
`T atanh(T x);`	Oblicza wartości funkcji hiperbolicznej `arc tan` argumentu x.

`T atanpi(T x);`	Oblicza wartości funkcji `arc tan` podzielonej przez liczbę π.
`T atan2pi(T x, T y);`	Oblicza wartości funkcji `arc tan` argumentu x / y, dodatkowo wartość jest dzielona przez liczbę π.
`T cbrt(T x);`	Oblicza wartość pierwiastka trzeciego stopnia z wartości x.
`T ceil(T x);`	Zaokrąglenie do najbliższej wartości całkowitej w kierunku „plus-nieskończoność”.
`T copysign(T x,T y);`	Zwraca wartość x ze znakiem zgodnym z wartością y.
`T cos(T x);`	Oblicza wartości funkcji `cos` dla argumentu x.
`T cosh(T x);`	Oblicza wartości hiperbolicznej funkcji `cos` dla argumentu x.
`T cospi(T x);`	Oblicza wartości funkcji `cos` dla argumentu (x pomnożonego najpierw przez liczbę π).
`T erfc(T x);`	Oblicza dopełnienie funkcji błędu: `1.0 – erf(x)`.
`T erf(T x);`	Wyznacza wartość funkcji błędu, dla argumentu x oblicza wartość następującego wyrażenia: $$erf(x) := \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$
`T exp(T x);`	Oblicza wartość funkcji wykładniczej o podstawie *e* dla argumentu x.
`T exp2(T x);`	Oblicza wartość funkcji wykładniczej o podstawie 2 dla argumentu x.

`T exp10(T x);`	Oblicza wartość funkcji wykładniczej o podstawie 10 dla argumentu x.
`T expm1(T x);`	Oblicza wartości funkcji e^x – `1.0`.
`T fabs(T x);`	Oblicza wartość bezwzględną liczby zmiennoprzecinkowej.
`T fdim(T x,T y);`	Zwraca wartość `x – y`, jeśli `x > y`, bądź 0, jeśli `x <= y`.
`T floor(T x);`	Zaokrągla do najbliższej wartości całkowitej w kierunku „minus-nieskończoność”.
`T fma(T a, T b, T c);`	Zwraca poprawnie zaokrągloną i obliczoną wartość zmiennoprzecinkową sumy wartości `c` oraz iloczynu `a` i `b`. Wartość pośrednia iloczynu zazwyczaj nie jest zaokrąglana, ale liczona dokładnie. Zachowanie przypadków brzegowych jest zgodne ze standardem IEEE 754-2008.

`T fmax(T x,T y);`
`Tf fmax(Tf x, float y);`
`Td fmax(Td x, double y);`
	Zwracają wartość większą z dwóch podanych argumentów, czyli y, jeśli `x<y`, w przeciwnym przypadku x. Jeśli jeden z argumentów przyjmuje wartość `NaN`, funkcja zwraca wartość drugiego argumentu. Jeśli oba argumenty to `NaN`, zwraca wartości `NaN`.

`T fmin(T x, T y);`
`Tf fmin(Tf x, float y);`
`Td fmin(Td x, double y);`
	Zwracają wartość mniejszą z dwóch podanych argumentów, czyli y, jeśli `y<x`, w przeciwnym przypadku x. Jeśli jeden z argumentów przyjmuje wartość `NaN`, funkcja zwraca drugi z argumentów. Jeśli oba argumenty to `NaN`, zwraca wartości `NaN`

`T fmod(T x,T y);`	Funkcja modulo zwraca wartość `x – y * trunc(x / y)`.
`T fract(T x, global T *iptr);` `T fract (T x, local T *iptr);`	Obliczają wartość `fmin(x– floor(x), 0x1.fffffep-1f)`, dodatkowo wartość `floor(x)` jest umieszczana we wskaźniku `iptr`.
`T fract (T x,`	

`private T *iptr);`	
`T frexp(T x, global intn*exp);`	
`T frexp (T x, local intn *exp);`	Zwracają wartość mantysy oraz wykładnika w podanej wartości. Mantysa jest zwracana jako wartość znormalizowana z przedziału lewostronnie domkniętego [0.5 ,1) lub zwracane jest zero.
`T frexp (T x, private intn*exp);`	
`T hypot(T x, T y);`	Oblicza wartość pierwiastka kwadratowego wyrażenia $x^2 + y^2$ bez nadmiaru lub niedomiaru.
`intn ilogb(T x);`	Zwraca wartość wykładnika dla argumentu x jako wartość całkowitą (liczba typu integer).
`T ldexp( T x, intn e);`	Obliczają wartość wyrażenia: $x * 2^e$.
`T ldexp( T x, int e);`	
`T lgamma(T x);`	
`T lgamma_r(T x, global intn*signp);`	Obliczają wartość logarytmu z funkcji gamma, co można zapisać w następujący sposób: $\log_e \lvert\Gamma(x)\rvert \;\; gdzie \;\; \Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt.$
`T lgamma_r(T x, local intn*signp);`	
`T lgamma_r(T x, private intn*signp);`	Wartość znaku funkcji gamma jest zwracana w parametrze `signp` , jeśli użyjemy funkcji `lgamma_r`.
`T log(T x);`	Oblicza wartość logarytmu naturalnego dla argumentu x.
`T log2(T x);`	Oblicza wartość logarytmu o podstawie 2 dla argumentu x.
`T log10(T x);`	Oblicza wartość logarytmu dziesiętnego dla argumentu x.

`T log1p(T x);`	Oblicza wartość następującego wyrażenia: $\log_e(1.0 + x)$.
`T logb(T x);`	Wyznacza wykładnik argumentu x , będący częścią wyrażenia $\log_r\lvert x\rvert$.
`T mad(T a, T b, T c);`	Oblicza przybliżoną wartość wyrażenia `a*b+c`. Nie jest określone, w jaki sposób zaokrągla się wynik iloczynu a i b, dlatego funkcja `mad` powinna być stosowana w przypadkach, gdy dokładność nie jest najważniejsza, a preferowana jest szybkość działania.
`T maxmag(T x, T y);`	Zwraca wartość x, jeśli `|x|>|y|`, bądź y, jeśli `|y|>|x|`, w innym przypadku wynik to wartość wyrażenia `fmax(x,y)`.
`T minmag(T x, T y);`	Zwraca wartość x, jeśli `|x|<|y|`, bądź y, jeśli `|y|<|x|`, w innym przypadku wynik to wartość wyrażenia `fmin(x,y)`.

`T modf(T x, global T *iptr);`
`T modf(T x, local T *iptr);`
`T modf(T x, private T *iptr);`
	Rozkłada liczbę zmiennoprzecinkową na część całkowitą oraz ułamkową. W obydwu przypadkach zachowuje znak argumentu. Część całkowita jest umieszczana w parametrze `iptr`, a część ułamkowa zwracana jako rezultat.

`float nan(uint nancode);`
`floatn nan(uintn nancode);`
`double nan(uint nancode);`
`doublen nan(uintn nancode);`
	Zwraca kod `NaN` . Kod może zostać umieszczony w najbardziej znaczących bitach zwróconej wartości.

`T nextafter(T x, T y);`	Oblicza następną wartość w pojedynczej lub podwójnej precyzji następującej po podanej wartości x w kierunku wartości y. Jeżeli podana w argumencie y wartość jest mniejsza, funkcja zwróci najbliższą wartość, jaką można reprezentować, choć mniejszą niż wartość `x`.
`T pow(T x, T`	

`y);`	Oblicza wartość wyrażenia x^y.
`T pown(T x, intn y);`	Oblicza wartość wyrażenia x^y, gdzie y jest liczbą całkowitą.
`T powr(T x, T y);`	Oblicza wartość wyrażenia x^y, gdzie wartość `x` spełnia warunek `x>=0`.
`T remainder(T x, T y);`	Reszta z dzielenia `x` przez y, tj. oblicza wartość `r` wg wzoru `r = x - n*y`, gdzie n jest liczbą całkowitą najbliższą dokładnej wartości `x / y`. Jeśli są dwie liczby równo odległe od `x / y`, to zostanie wybrana wartość parzysta. Dodatkowo, jeśli `r` jest równe zero, rezultat otrzyma znak taki sam jak wartość `x`.
`T remquo(T x, gentype y, global Ti *quo);`	
`T remquo(T x, gentype y, local gTi *quo);`	Oblicza wartość reszty z dzielenia identycznie jak `remainder`. Dodatkowo, młodsze siedem bitów wartości `x / y` jest umieszczane w parametrze `quo`. Zachowany zostaje również znak wartości `x / y`.
`T remquo(T x, gentype y, private Ti *quo);`	
`T rint(T x);`	Zaokrągla do wartości całkowitej w formacie zmiennoprzecinkowym.
`T rootn(T x, intn y);`	Oblicza wartość `x` podniesioną do potęgi `1/y`, czyli pierwiastek stopnia y z wartości `x`.
`T round(T x);`	Zaokrągla wartość `x` do najbliższej liczby całkowitej.
`T rsqrt(T x);`	Oblicza wartości odwrotności pierwiastka kwadratowego argumentu `x`.
`T sin(T x);`	Oblicza wartość funkcji `sin` argumentu `x`.
`T sincos(T x, global T`	

`*cosval);`	
`T sincos(T x, local T *cosval);`	Obliczają wartości funkcji `sin` oraz `cos` dla argumentu `x`. Zwraca wartość funkcji `sin` , a wartość funkcji `cos` umieszcza w parametrze `cosval`.
`T sincos(T x, private T *cosval);`	
`T sinh(T x);`	Oblicza wartość hiperbolicznej funkcji `sin` dla argumentu `x`.
`T sinpi(T x);`	Oblicza wartość `sin(π*x)`.
`T sqrt(T x);`	Pierwiastek kwadratowy argumentu `x`.
`T tan(T x);`	Oblicza wartość funkcji tangens dla argumentu `x`.
`T tanh(T x);`	Oblicza wartość funkcji hiperbolicznego tangensa dla argumentu `x`.
`T tanpi(T x);`	Oblicza wartość `tan(π*x)`.
`T tgamma(T x);`	Oblicza wartość funkcji `gamma`.
`T trunc(T x);`	Zaokrągla do wartości całkowitej w kierunku zera.

Oznaczenia: `T` – dowolny typ skalarny lub wektorowy, `Tf` – typ skalarny i wektorowy oparty na liczbach pojedynczej precyzji, tj. `float`, `Td` – liczby `double`, `Ti` – dowolny typ skalarny czy wektorowy oparty na liczbach całkowitych.

Open CL dysponuje także dość obszernym podzbiorem funkcji obsługujących typ `half`, lista tych funkcji została przedstawiona w tab. 3.10.

Tabela 3.10. Funkcje matematyczne dostępne w OpenCL C obsługujące typ `half` oraz tzw. tryb natywny

Funkcja	Opis

`Thf half_cos( Thf x );`	Oblicza wartość funkcji `cos` z podanego argumentu x, wartość x musi być w zakresie od -2^{16} do 2^{16}.
`Tf native_cos( Tf x );`	Oblicza wartość funkcji `cos` w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_divide( Thf x, Thf y);`	Oblicza wartość ilorazu x przez y.
`Tf native_divide( Tf x, Tf y );`	Oblicza wartość ilorazu x przez y w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_exp( Thf x );`	Oblicza funkcję wykładniczą o podstawie e.
`Tf native_exp( Tf x );`	Oblicza funkcję wykładniczą o podstawie e w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_exp2( Thf x );`	Oblicza funkcję wykładniczą o podstawie 2.
`Tf native_exp2( Tf x );`	Oblicza funkcję wykładniczą o podstawie 2 w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_exp10( Thf x );`	Oblicza funkcję wykładniczą o podstawie 10.
`Tf native_exp10( Tf x );`	Oblicza funkcję wykładniczą o podstawie 10 w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_log( Thf x );`	Oblicza wartość logarytmu o podstawie e.
`Tf native_log( Tf x );`	Oblicza wartość logarytmu o podstawie e w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_log2( Thf x);`	Oblicza wartość logarytmu o podstawie 2.
`Tf native_log2( Tf x );`	Oblicza wartość logarytmu o podstawie 2 w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_log10( Thf x);`	Oblicza wartość logarytmu o podstawie 10.
`Tf native_log10( Tf x );`	Oblicza wartość logarytmu o podstawie 10 w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.

`Thf half_powr( Thf x, Thf y);`	Oblicza wartość potęgi x do y, gdzie wartość x jest większa bądź równa zero.
`(brak odpowiednika native)`	
`Thf half_recip( Thf x);`	Oblicza odwrotność argumentu x.
`Tf native_recip( Tf x );`	Oblicza odwrotność argumentu x w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_rsqrt( Thf x);`	Oblicza odwrotność pierwiastka kwadratowego z argumentu x.
`Tf native_rsqrt( Tf x );`	Oblicza odwrotność pierwiastka kwadratowego z argumentu x w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_sin( Thf x);`	Oblicza wartość funkcji `sin` z podanego argumentu x, wartość x musi być w zakresie od -2^{16} do 2^{16}.
`Tf native_sin( Tf x );`	Oblicza wartość funkcji `sin` w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_sqrt( Thf x);`	Oblicza wartość pierwiastka kwadratowego z argumentu x.
`Tf native_sqrt( Tf x );`	Oblicza wartość pierwiastka kwadratowego w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.
`Thf half_tan( Thf x);`	Oblicza wartość funkcji `tan` z podanego argumentu x, wartość x musi być w zakresie od -2^{16} do 2^{16}.
`Tf native_tan( Tf x );`	Oblicza wartość funkcji `tan` w dziedzinie zależnej od implementacji. Wartość błędu także zależy od użytej implementacji.

Oznaczenia: `Thf` – typ skalarny czy wektorowy o ograniczonej precyzji, tj. oparty na typie `half`, funkcje z przyrostkiem `native` – funkcje natywne dla danej platformy obliczeniowej, np. ich dokładność zależy od sposobu implementacji.

Wsparcie dla operacji matematycznych to także definicje stałych. Definicje kończące się przyrostkiem „`_F`” odnoszą się do liczby typu `float`, a definicje stałych typu `double` nie mają tego przyrostka.

- `M_E_F`, `M_E` – wartość liczby e,
- `M_LOG2E_F`, `M_LOG2E` – wartość wyrażenia $\log_2 e$,

- `M_LOG10E_F`, `M_LOG10E` – wartość wyrażenia $\log_{10} e$,
- `M_LN2_F`, `M_LN2` – wartość wyrażenia $\log_e 2$,
- `M_LN10_F`, `M_LN10` – wartość wyrażenia $\log_e 10$,
- `M_PI_F`, `M_PI` – wartość liczby π,
- `M_PI_2_F`, `M_PI_2` – wartość liczby π/2,
- `M_PI_4_F`, `M_PI_4` – wartość liczby π/4,
- `M_1_PI_F`, `M_1_PI` – wartość liczby 1/π,
- `M_2_PI_F`, `M_2_PI` – wartość liczby 2/π,
- `M_2_SQRTPI_F`, `M_2_SQRTPI` – wartość wyrażenia 2/√π,
- `M_SQRT2_F`, `M_SQRT2` – wartość wyrażenia √π,
- `M_SQRT1_2_F`, `M_SQRT1_2` – wartość wyrażenia 1/√π.

Analogicznie jak dla funkcji matematycznych, które są przystosowane do liczb zmiennoprzecinkowych, w ramach standardu OpenCL określony został też zestaw funkcji dla typów liczb całkowitych. Funkcje wymienione w tab. 3.11 mogą być wykorzystywane przez zmienne skalarne oraz wektorowe, jak np. `char2`, `long4` etc.

Tabela 3.11. Zestaw funkcji dla typów skalarnych i wektorowych opartych na typach liczb całkowitych

Funkcja	Opis
`Tui abs(Ti x);`	Zwraca wartość bezwzględną argumentu x.
`Tui abs_diff( Ti x, Ti y);`	Oblicza wartość |x – y| bez obsługi przepełnienia.
`Ti add_sat(Ti x, Ti y);`	Oblicza x + y, a następnie, jeśli nastąpiło przepełnienie, ogranicza wynik do wartości maksymalnej danego typu (saturacja).
`Ti hadd( Ti x, Ti y);`	Oblicza wartość (x+ y) >> 1, sumę pośrednią oblicza bez obsługi przepełnienia.

`Ti rhadd( Ti x, Ti y);`	Oblicza wartość (x+ y + 1) >> 1, sumę pośrednią oblicza bez obsługi przepełnienia.

`Ti clamp( Ti x, Ti minval, Ti maxval);`
`Ti clamp( Ti x, Tsi minval, Tsi maxval);`
`T clamp(T x, T minval, T maxval);`
`Tf clamp(Tf x, float minval, float maxval);`
`Td clamp(Td x, double minval, double maxval);`
	Obliczają wartość wyrażenia `min(max(x,minval), maxval)`, jeśli `minval` > `maxval`, to wynik jest nieokreślony.

`Ti clz( Ti x);`	Zwraca liczbę wiodących zer w parametrze x, począwszy od bitu najbardziej znaczącego.
`Ti mad_hi( Ti a, Ti b, Ti c);`	Oblicza wartość wyrażenia `mul_hi(a,b) + c`.
`Ti mad_sat(Ti a, Ti b, Ti c);`	Oblicza `a * b + c`, a następnie, jeśli nastąpiło przepełnienie, ogranicza wynik do wartości maksymalnej danego typu (saturacja).

`Ti max( Ti x, Ti y);`
`Ti max( Ti x, Tsi y);`
`T max(T x, T y);`
`Tf max(Tf x, float y);`
`Td max(Td x, double y);`
	Zwraca większy argument z podanych dwóch wartości.

`Ti min(Ti x, Ti y);` `Ti min(Ti x, Tsi y);`	

`T min(T x, T y);`
`Tf min(Tf x, float y);`
`Td min(Td x, double y);`
	Zwraca mniejszy argument z podanych dwóch wartości.

`Ti mul_hi(Ti x, Ti y);`	Oblicza x * y i zwraca wyższą połowę reprezentacji bitowej wartości iloczynu x oraz y.
`Ti rotate(Ti v, Ti i);`	Przesuwa bity w lewo o liczbę określoną w parametrze `i`. Bity przesunięte maksymalnie w lewo powracają po prawej stronie.
`Ti sub_sat(Ti x, Ti y);`	Oblicza x – y, a następnie, jeśli nastąpiło przepełnienie, ogranicza wynik do wartości maksymalnej danego typu (saturacja).

`short upsample(charhi, uchar lo);`
`ushort upsample(ucharhi, uchar lo);`
`shortn upsample(charn hi, ucharn lo);`
`ushortn upsample(ucharn hi, ucharn lo);`
`int upsample(shorthi, ushort lo);`
`uint upsample(ushorthi, ushort lo);`
`intn upsample(shortn hi, ushortn lo);`
`uintn upsample(ushortn hi, ushortn lo);`
`long upsample(inthi, uint lo);`
`ulong upsample(uinthi, uint lo);`
	Przeprowadzają operację budowy wartości o szerszej postaci za pomocą dwóch wielkości `hi` oraz `lo`, będących typem o połowę mniejszym. Jeśli `hi` oraz `lo` są skalarami, to poszczególne operacje można zapisać w następujący sposób (zmienna `r` oznacza rezultat):
	• `r = ((short)hi<< 8) | lo`
	• `r = ((ushort)hi<< 8) | lo`
	• `r = ((int)hi<< 16) | lo`
	• `r = ((uint)hi<< 16) | lo`
	• `r = ((long)hi<< 32) | lo`
	• `r = ((ulong)hi<< 32) | lo`
	Jeżeli argumenty to typy wektorowe, przeprowadzają podobną operację, ale na poszczególnych komponentach (zmienna `r` oznacza rezultat):
	• `r[i] = ((short)hi[i] << 8) | lo[i]`
	• `r[i] = ((ushort)hi[i] << 8) | lo[i]`
	• `r[i] = ((int)hi[i] << 16) | lo[i]`
	• `r[i] = ((uint)hi[i] << 16) | lo[i]`
	• `r[i] = ((long)hi[i] << 32) | lo[i]`

`longn upsample(intn hi, uintn lo);`	• `r[i] = ((ulong)hi[i] << 32) | lo[i]`
`ulongn upsample(uintn hi, uintn lo);`	
`Ti mad24(Ti x, Ti y, Ti z);`	Wykonuje 24-bitowe mnożenie na argumentach x oraz y, a następnie dodaje wartość z w arytmetyce 32-bitowej.
`Ti mul24(Ti x, Ti y);`	Wykonuje 24-bitowe mnożenie na argumentach x oraz y, dlatego należy ją stosować, gdy argumenty mają wartości w zakresie $[-2^{23}, 2^{23} - 1]$ lub $[0, 2^{24}-1]$. Mnożenie większych wartości może powodować nieprzewidziane rezultaty, bowiem w tym przypadku zachowanie się funkcji `mul24` zależy od danej implementacji sprzętowej.

Oznaczenia: `Tui` – dowolny typ skalarny lub wektorowy oparty na typie liczby całkowitej bez znaku, `Tsi` – typ liczby całkowitej ze znakiem, `Ti` – dowolny typ oparty na liczbie całkowitej.

Analogicznie jak w przypadku stałych zmiennoprzecinkowych, dostępny jest zestaw stałych dla liczb całkowitych, w całości przeniesiony z języka C. Oto lista wybranych stałych:

- `CHAR_BIT 8`
- `CHAR_MAX SCHAR_MAX`
- `CHAR_MIN SCHAR_MIN`
- `INT_MAX 2147483647`
- `INT_MIN (-2147483647 – 1)`
- `LONG_MAX 0x7fffffffffffffffL`
- `LONG_MIN (-0x7fffffffffffffffL – 1)`
- `SCHAR_MAX 127`
- `SCHAR_MIN (-127 – 1)`
- `SHRT_MAX 32767`
- `SHRT_MIN (-32767 – 1)`
- `UCHAR_MAX 255`
- `USHRT_MAX 65535`
- `UINT_MAX 0xffffffff`

- `ULONG_MAX 0xffffffffffffffffUL`

W tabeli 3.12 przedstawiono kolejny zestaw funkcji pomocniczych, np. funkcje do konwersji wartości kątów w radianach oraz w stopniach, a także zestaw funkcji do porównywania wartości odmiennych typów. Podane funkcje obsługują nie tylko dane typu skalarnego, lecz także typu wektorowego.

Tabela 3.12. Zestaw funkcji pomocniczych oraz relacyjnych

Funkcja	Opis
	Funkcje pomocnicze
`T degrees(T radians);`	Zamienia wartości kąta w radianach na stopnie.
`T radians(T degrees);`	Zamienia wartości kąta w stopniach na radiany.

`T step(T edge, T x);`
`Tf step(float edge, Tf x);`
`Td step(double edge, Td x);`
	Zwraca wartość zero, jeśli `x < edge`, w przeciwnym przypadku otrzymuje się wartość jeden.

`T smoothstep(T edge0, T edge1, T x);`
`Tf smoothstep(float edge0, float edge1, Tf x);`
`Td smoothstep(double edge0, double edge1, Td x);`
	Zwraca wartość zero, jeśli `x < edge0`, wartość jeden, jeśli `x >= edge1`, w pozostałych przypadkach, tzn. `edge0 < x < edge1` przeprowadza interpolację i zwraca wartość z przedziału od 0 do 1.

`T sign(T x);`	Zwraca wartość 1.0, jeśli x > 0, –1, jeśli x < 0. Jeśli x jest zerem, zwraca wartość x uzupełnioną o znak, wartość zero zwraca wówczas, gdy x jest `NaN`.
	Funkcje relacyjne

```
int isequal(float x,
float y);

int isequal(double x,
double y);

intn isequal(floatn x,
floatn y);

 longn isequal(doublen
x, doublen y);
```

Wykonuje operacje relacji: `x == y`, w przypadku wektorów porównywane są odpowiednie składowe.

```
int isnotequal(float
x, float y);

int isnotequal(double
x, double y);

intn isnotequal(floatn
x, floatn y);

 longn
isnotequal(doublen x,
doublen y);
```

Wykonuje operacje relacji: `x != y`, w przypadku wektorów porównywane są odpowiednie składowe.

```
int isgreater(floatx,
float y);

int isgreater(double x,
double y);

intn isgreater(floatn
x, floatn y);

 longn
isgreater(doublen x,
doublen y);
```

Wykonuje operacje relacji: `x > y`, w przypadku wektorów porównywane są odpowiednie składowe.

```
int
isgreaterequal(float x,
float y);

int
isgreaterequal(double
x, double y);

intn
isgreaterequal(floatn
x, floatn y);

 longn
isgreaterequal(doublen
x,
doublen y);
```

Wykonuje operacje relacji: `x >= y`, w przypadku wektorów porównywane są odpowiednie składowe.

```
int isless(floatx,
float y);
```

```
int isless(double x,
double y);
```

```
intn isless(floatn x,
floatn y);
```

Wykonuje operacje relacji: `x < y`, w przypadku wektorów porównywane są odpowiednie składowe.

```
longn isless(doublen
x, doublen y);
```

```
int islessequal(float
x, float y);

int islessequal(double
x, double y);
```

```
intn islessequal(floatn
x, floatn y);
```

Wykonuje operacje relacji: `x <= y`, w przypadku wektorów porównywane są odpowiednie składowe.

```
longn
islessequal(doublen x,
doublen y);
```

```
int
islessgreater(floatx,
float y);

int
islessgreater(doublex,
double y);
```

```
intn
islessgreater(floatn x,
floatn y);
```

Wykonuje operacje relacji: `(x < y) || (x > y)`, w przypadku wektorów porównywane są odpowiednie składowe.

```
longn
islessgreater(doublenx,
doublen y);
```

```
int isfinite(float x);

int isfinite(double x);
```

```
intn isfinite(floatn
x);
```

Sprawdza, czy argument x jest wartością skończoną.

```
longn isfinite(doublen
x);
```

```
int isinf(float x);

int isinf(double x);
```

```
intn isinf(floatn x);
```

Sprawdza, czy argument x jest wartością nieskończoną.

```
longn isinf(doublen
x);
```

`int isnan(float x);`
`int isnan(double x);`
`intn isnan(floatn x);`
`longn isnan(doublen x);`
	Sprawdza, czy podana wartość to `NaN`.

`int isnormal(float x);`
`int isnormal(double x);`
`intn isnormal(floatn x);`
`longn isnormal(doublen x);`
	Sprawdza, czy argument x jest liczbą w postaci znormalizowanej (tzn. nie może być zerem, liczbą w postaci nieznormalizowanej, nieskończonością lub NaN).

`int isordered(float x, float y);`
`int isordered(double x, double y);`
`intnisordered(floatn x, floatn y);`
`longnisordered(doublen x, doublen y);`
	Sprawdza uporządkowanie argumentów, co odpowiada wykonaniu następującego testu: `isequal(x,x) && isequal(y,y)`.

`int isunordered(floatx, float y) ;`
`int isunordered(double x, double y);`
`intn isunordered(floatn x, floatn y);`
`longn isunordered(doublen x, doublen y);`
	Sprawdza nieuporządkowanie argumentów. Zwraca wartość niezerową, jeśli `x` lub `y` są wartościami `NaN`, lub zero w przeciwnym przypadku.

`int signbit(float x);`
`int signbit(double x);`
`intn signbit(floatn x);`
	Odczytują bit znaku, zwracają wartość 1, jeśli bit znaku jest ustawiony na 1 lub wartość 0, jeśli jest ustawiony na 0. W przypadku wektora poszczególne jego składowe otrzymują wartość –1, jeśli bit znaku jest ustawiony na 1, bądź zero w przeciwnym przypadku.

```
longn signbit(doublen
x);
```

3.4. Dodatkowe kwalifikatory oraz atrybuty

Język OpenCL C dostarcza kilkunastu dodatkowych kwalifikatorów (atrybutów) odnoszących się do funkcji, adresów, trybów dostępu oraz typów danych, jakie są używane w procedurach obliczeniowych. Lista dodatkowych kwalifikatorów jest następująca:

• `kernel` lub `__kernel`,

• `global` lub `__global`, `local` lub `__local`, `constant` lub `__constant`, `private` lub `__private`,

• `read_only` lub `__read_only`, `write-only` lub `__write_only`.

Atrybut `__kernel` (bądź `kernel`) oznacza, że dana funkcja będzie traktowana jako jądro obliczeniowe i będzie można ją wywołać z poziomu gospodarza, aby przeprowadzić obliczenia. Funkcje bez tego atrybutu są kompilowane i uruchamiane w ramach urządzenia obliczeniowego OpenCL, ale tylko przez inne procedury umieszczone w tym urządzeniu. Z poziomu systemu gospodarza można uruchamiać tylko procedury opatrzone atrybutem `__kernel`. Deklaracja funkcji jako jądra obliczeniowego przypomina normalną funkcję języka C:

```
__kernel void kernel_func(__global float *f, const int N) {

    // treść funkcji obliczeniowej

}
```

OpenCL C wymaga, aby funkcja pełniąca rolę jądra obliczeniowego spełniała trzy podstawowe warunki:

• typ zwracanej wartości to `void`, co oznacza iż jądro obliczeniowe nie zwraca żadnej wartości,

• uruchomienie jądra obliczeniowego wraz z konfiguracją siatki obliczeniowej następuje tylko z poziomu gospodarza[2],

• możliwa jest sytuacja, że jądro obliczeniowe zostanie uruchomione przez inną funkcję obliczeniową, ale wówczas będzie się ono zachowywać jak zwykła funkcja (pozbawiona atrybutu `__kernel`).

Należy unikać uruchamiania innego jądra obliczeniowego na poziomie urządzenia obliczeniowego. Szczególnie, jeśli wykorzystuje się zmienne lokalne umieszczone w pamięci współdzielonej przez wszystkie jednostki robocze w danej grupie roboczej. Ponowny dostęp do zmiennych lokalnych w takim przypadku zależy od implementacji – tego typu rozwiązanie w obecnych wersjach standardu OpenCL nie powinno być stosowane.

Atrybut `__global` (lub `global`) oznacza, iż dany obiekt (bufor lub obraz) odnosić się będzie do pamięci globalnej urządzenia obliczeniowego. W przypadku bufora można z niego korzystać jak z tablicy wartości skalarnej. Obiekt globalny może też być strukturą określoną przez użytkownika.

Dostęp do obiektu opatrzonego słowem `__global` w ramach jądra obliczeniowego obejmuje operacje zapisu oraz odczytu w dowolnym miejscu danego obiektu, a jego wielkość jest określona w momencie tworzenia (funkcja API `clCreateBuffer` – dla bufora w pamięci globalnej urządzenia OpenCL).

Definicja wskaźnika do zmiennej opisującej kolor RGB wraz z kanałem alfa może przyjąć następującą postać:

```
__global float4 *colorRGBA;
```

Wykorzystano tu definicję wektora o czterech elementach typu zmiennoprzecinkowego.

Słowem `__global` można opatrywać też struktury definiowane przez użytkownika, np.:

```
typedef struct {

    float fieldA[8];

    int fieldB[4];

    char fieldC[2]

} simpleStructure_t;
```

Definicja struktury w pamięci globalnej jest następująca:

```
__global simpleStructure_t *my_structure;
```

Można deklarować tylko wskaźniki do pamięci globalnej, jak np:

```
__kernel void kernel_func(__global float *inData) {

    __global float *ptr = inData + 10;

    float f = *ptr;

}
```

Nie zawsze trzeba stosować słowo `__global`, jak w przypadku obrazów typu `image_2d`. Dostęp do zmiennych tego typu nie jest możliwy w sposób bezpośredni, trzeba stosować dodatkowe funkcje, jak np. `read_imageui`. Oprócz tego, każdy obraz zawsze jest składowany w pamięci globalnej. Jeśli chcemy, aby zmienna globalna była przeznaczona tylko do odczytu, można dodać słowo kluczowe `const`.

Powyższa uwaga nie przenosi się bezpośrednio na typy dla obrazów, bowiem jeśli chcemy, aby dany obraz, np. typu `image2d_t`, był tylko do odczytu, użyjemy kwalifikatora `read_only`, jeśli tylko do zapisu, użyjemy – `write_only`. Jeżeli nie podamy żadnego atrybutu, dane można odczytywać i zapisywać bez przeszkód.

Nie należy słowa `const` mylić ze słowem `constant` lub `__constant`, bo choć zmienne opatrzone tym słowem pozostają zmiennymi tylko do odczytu, to są umieszczone w dodatkowym specjalnym buforze. Nie jest to zbyt duży obszar pamięci, zazwyczaj w przypadku kart graficznych jego rozmiar

to około 64 kb. Ograniczona jest też liczba tego typu zmiennych, wartość tę można odczytać za pomocą stałej `CL_DEVICE_MAX_CONSTANT_ARGS`, a także za pomocą funkcji `clGetDeviceInfo` przedstawionej w dodatku D. Sposób odczytu pokazano w przykładzie w rozdz. 4.1. Dane oznaczone jako `__constant` są dostępne dla wszystkich jednostek roboczych, dodatkowo należy pamiętać, aby nadać tego typu zmiennym wartości początkowe przed uruchomieniem jądra obliczeniowego.

Oprócz pamięci globalnej dostępne są także pamięć lokalna oraz prywatna. Deklarując zmienne, należy stosować odpowiednio atrybuty `local` bądź `private`.

Zmienne lokalne są dostępne dla wszystkich jednostek roboczych w ramach jednej grupy roboczej, dlatego ten typ pamięci można wykorzystać do szybkiej komunikacji między wątkami (jednostkami roboczymi), a także do przyspieszenia obliczeń. Potrzebny fragment danych można odczytać z pamięci globalnej, wykonać niezbędne obliczenia w pamięci lokalnej, a następnie umieścić ten przetworzony fragment ponownie w pamięci globalnej.

Na zmienne lokalne nałożone zostały pewne restrykcje, które wynikają z ich zastosowania. Mimo że można przekazać wskaźnik do obszaru pamięci lokalnej w celu wykonania procedury obliczeniowej, to deklaracja zmiennych lokalnych wewnątrz funkcji obliczeniowej jest możliwa tylko w przestrzeni głównej jądra obliczeniowego, co pokazuje następujący przykład:

```
__kernel void krnFunc(global float4 *inData, local float4 *lNums){

    local float4 *ptr; // OK

    local float avar; // OK

    ….

    avar = 1;

    …

    if (…){

        local float cvar; // BŁĄD

        …

    }

}
```

Deklaracje zmiennych `ptr` oraz `avar` są poprawne, bowiem są umieszczone w przestrzeni głównej, natomiast deklaracja zmiennej `cvar` jest błędna, gdyż zmienna jest deklarowana w przestrzeni lokalnej instrukcji `if`.

Drugim istotnym obostrzeniem jest brak możliwości zainicjowania zmiennej lokalnej, gdyż dane jądro obliczeniowe jest uruchamiane wielokrotnie, więc zainicjowanie musiałaby być wykonane przez wszystkie jednostki robocze w grupie. W efekcie otrzymano by wiele konfliktów zapisu w pamięci lokalnej, szczególnie wówczas, gdy zainicjowanie wykonywane przez poszczególne jednostki robocze zapisywałaby do danej zmiennej różne wartości, np. w zależności od identyfikatora

jednostki roboczej.

Atrybut `private` oznacza, że zmienne nim opatrzone są w pełni prywatne i dostępne tylko dla aktualnej jednostki roboczej. W zasadzie nie potrzeba używać tego atrybutu, gdyż zmienne deklarowane w jądrze obliczeniowym bez dodatkowych atrybutów są domyślnie zmiennymi prywatnymi. Jednakże ilość pamięci, jaka może zostać zadeklarowana jako prywatna jest niewielka, dlatego należy zwrócić uwagę na to, aby nie przekroczyć możliwości danego urządzenia obliczeniowego w tym zakresie.

W kontekście różnych zmiennych należy zwrócić też uwagę na konwersję między wskaźnikami z różnych przestrzeni adresowych. Jak łatwo odgadnąć, nie jest ona możliwa – nie możemy wykonać konwersji wskaźnika globalnego do lokalnego czy prywatnego obszaru pamięci, nawet jeśli wskazywany obszar pamięci zmieści się w innym obszarze pamięci.

Do samego jądra obliczeniowego odnoszą się jeszcze trzy dość często stosowane atrybuty, które dotyczą przestrzeni indeksującej:

• `__attribute__((work_group_size_hint(X, Y, Z)))` – pozwala zasugerować wielkość grupy roboczej podczas kompilacji jądra obliczeniowego,

• `__attribute__((reqd_work_group_size(X, Y, Z)))` – pozwala podać dokładną wielkość grupy roboczej, jaka jest stosowana w danym jądrze obliczeniowym, w efekcie czego można uzyskać optymalizację jądra obliczeniowego,

• `__attribute__((vec_type_hint(<type>)))` – pozwala określić rozmiary typów, jakie są przetwarzane przez jądro obliczeniowe. Domyślny jest rozmiar typu `int`, co oznacza, że jądro obliczeniowe będzie przetwarzać dane skalarnie. Zwiększenie rozmiaru typu danych może skutkować autowektoryzacją, np. w zastosowanych pętlach `for`, co może przyczynić się do zwiększenia wydajności działania całego jądra obliczeniowego.

OpenCL C oferuje też trzy kwalifikatory dla typów `const` (wspomniany w kontekście atrybutu `__global`), `restrict` oraz `volatile`, które zostały przeniesione ze standardu C99. Nie można ich stosować w odniesieniu do typów reprezentujących obrazy dwu- oraz trójwymiarowe: `image2d_t`, `image3d_t`. Dodatkowo, w przypadku typów innych niż typy wskaźnikowe nie można stosować kwalifikatora `restrict`.

3.5. Operacje na typach wektorowych OpenCL

Jak to zostało pokazane w rozdz. 3.3, OpenCL i OpenCL C oferują dość szeroki zestaw funkcji pracujących na tych samych typach. Dotyczy to także typów wektorowych. Oprócz wymienionych już funkcji matematycznych OCL oferuje dodatkowo uproszczenie wielu podstawowych operacji, jakie trzeba wykonywać na zmiennych wektorowych.

3.5.1. Dane typu wektorowego – dostęp do składowych

Posługiwanie się danymi typu wektorowego nie nastręcza dodatkowych kłopotów, tym bardziej, iż zdefiniowano szereg operatorów i dodatkowych konstrukcji syntaktycznych ułatwiających obsługę tych danych w OpenCL C.

Najbardziej pierwotną operacją jest zainicjowanie oraz dostęp do elementów wektora. Inicjowanie zmiennej wektorowej można zrealizować wprost, podając poszczególne składowe w postaci ciągu wartości objętych nawiasami okrągłymi i rozdzielonych przecinkami:

```
float4 v1= (float4)(1.0, 2.0, 3.0, 4.0);
```

Należy także wpisać wyrażenie `(float4)`, aby podany ciąg wartości został zamieniony na typ wektorowy o czterech elementach, będących liczbami zmiennoprzecinkowymi. W przypadku typów wektorowych `int4`, `char4` w zakresie zainicjowania postępujemy analogicznie.

Składowe wektora, które mają być zainicjowane tą samą wielkością, zapisujemy następująco:

```
float4 v2= (float4)(5.0);
```

Do zainicjowania wektora można użyć innej zmiennej skalarnej:

```
float v= 0.5f;

float4 v3= (float4)(v);
```

Możliwości w wykorzystaniu do zainicjowania innych zmiennych, w tym także zmiennych typu wektorowego, są znacznie większe. Nie ma przeszkód, aby wykorzystać dwie zmienne typu `float2` do utworzenia wektora o czterech elementach:

```
float2 a1 = (1, 2);

float2 a2 = (3, 4);

float4 a = (float4)(a2, a1);
```

Można także łączyć wektory z wartościami skalarnymi:

```
float s=0.75;

float3 t=(float3)(3,2,1);

float4 f4 = (float4)(t, s);
```

Możliwych kombinacji jest znacznie więcej. Oto przykłady niektórych z nich:

```
float4 f4 = (float4)( float, float, float, float )

float4 f4 = (float4)( float2, float, float )

float4 f4 = (float4)( float, float2, float )

float4 f4 = (float4)( float, float, float2 )

float4 f4 = (float4)( float2, float2 )

float4 f4 = (float4)( float3, float )

float4 f4 = (float4)( float, float3 )
```

```
float4 f4 = (float4)( float )
```

Przykłady te dotyczą także innych typów (np. `int4`), gdzie mamy do dyspozycji cztery elementy.

Należy przypomnieć, iż OpenCL C nie pozwala na łączenie wektorów w zupełnie dowolny sposób. Wymiary zawsze muszą się zgadzać z wartościami 2, 3, 4, 8, 16, a wielkość wektora zależy także od zastosowanego typu (patrz rozdz. 3.1.2).

W OpenCL C określa się sposób dostępu do poszczególnych elementów typu wektorowego. Można to zrobić za pomocą indeksowania poszczególnych elementów wektora, ale można też wskazywać pary czy też np. czwórki elementów naraz.

Dla zmiennej typu `float4` dostęp do poszczególnych elementów wektora możemy określić w następujący sposób:

```
float4 c;

c.x = 10.0;

c.y = 11.0;

c.z = 12.0;

c.w = 13.0;
```

Zastosowanie krótszego wektora, np. typu `int2` albo `int3`, ogranicza oznaczenia do x, y lub też do x, y, z. W OpenCL dopuszcza się powielanie oznaczeń komponentów i zmianę ich kolejności, co może się szczególnie przydać np. podczas operacji przypisania wartości wybranych elementów wektora:

```
float4 a = (float4)(1,2,3,4);

float4 b = a.xxyy;

float4 c = a.zxyw;

float4 d = a.wzyx;
```

Wektor `b` zostanie zainicjowany tylko wartościami `x` oraz y, a wektor `c` zawiera elementy `b`, ale w innych miejscach, natomiast wektor `d` ma elementy umieszczone w odwrotnej kolejności względem wektora `a`:

```
wektor b: (1,1,2,2)

wektor c: (3,1,2,4)

wektor d: (4,3,2,1)
```

Dla wektorów zwierających więcej niż cztery elementy należy stosować sposób oznaczenia indeksów za pomocą litery „s” (lub małej litery „s”) oraz wartości w kodzie szesnastkowym (litery w kodzie szesnastkowym ponownie mogą być wielkie lub małe). Należy jednak podkreślić, że nie można mieszać oznaczeń „x”, „y”, „z”, „w” z oznaczeniami indeksów rozpoczynających się od litery „s”.

Tabela 3.13. Oznaczenia indeksów stosowane dla typów wektorowych (cyfry szesnastkowe mogą być zapisane wielkimi lub małymi

literami)

Typ wektora	Oznaczenia literowe	Oznaczenia liczbowe poprzedzone literą „s” lub „S”
2-elementowy	x,y	0, 1
3-elementowy	x,y,z	0, 1, 2
4-elementowy	x,y,z,w	0, 1, 2, 3
8-elementowy	brak	0, 1, 2, 3, 4, 5, 6, 7
16-elementowy	brak	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

Trzeci sposób dostępu do składowych wektora to oznaczenia `lo`, `hi`, `even` oraz `odd`. Nazwy `lo` oraz `hi` oznaczają pierwszą oraz drugą połowę wektora. Jeśli stosujemy nazwę `lo`, to odwołujemy się do indeksów o numerach 0, 1, 2, …, N/2 – 1. W przypadku `hi` odwołanie dotyczy indeksów o numerach N/2, N/2 + 1, …, N – 1. Dla porządku warto dodać, iż indeksy są numerowane od 0 do N – 1. W przypadku wektorów o parzystej liczbie elementów nie ma problemu z podziałem na pierwszą i drugą połowę. W przypadku wektora o nieparzystej liczbie elementów, element środkowy jest klasyfikowany do części wektora oznaczonej jako `lo`.

Zapis `even` oraz `odd` oznacza odpowiednio parzyste oraz nieparzyste elementy wektora. Uwzględniając długości wektorów, łatwo wyciągać zestawy danych, np.:

```
uint8 a = (uint8)(1,2,3,4,5,6,7,8);

uint4 e = a.even;

uint4 o = a.odd;
```

Do wektorów `e` i `o` przeniesione zostaną cztery elementy o indeksach parzystych oraz nieparzystych:

```
a.lo = (uint4)(4,3,2,1);

a.hi=(uint4)(1,2,3,4);
```

Zawartość wektora o czterech elementach zostanie wstawiona w pierwszą oraz drugą połowę wektora ośmioelementowego. Jak widać, OpenCL jest bardzo elastyczny w obsłudze wektorów, dzięki czemu

można efektywnie realizować różne operacje zmieniające kolejność elementów w wektorze lub tworzyć nowe wektory.

Możliwe jest także tworzenie tablic opartych na typach wektorowych, np. definicja

```
float4 tab[16];
```

utworzy tablicę o szesnastu elementach, w której każdy element to wektor o czterech elementach.

3.5.2. Operacje na typach wektorowych

W języku OpenCL C mamy obszerny zestaw podstawowych operatorów obsługujących również dane typu wektorowego (patrz rozdz. 3.2). Dzięki temu np. dodanie dwóch wektorów 4-elementowych realizuje się w następujący sposób:

```
float4 a, b, c;

c = a + b;
```

Ta sama operacja w typowym kodzie dla języka C będzie zapisana następująco:

```
float a[4], b[4], c[4];

int i;

const int N = 4;

for(i=0;i<N;i++) {

    c[i] = a[i] + b[i];

}
```

Mnożenie przez skalar również nie wymaga stosowania oddzielnych funkcji, OpenCL C oferuje odpowiednio predefiniowane operatory dla operacji wektor – skalar, np:

```
float4 t = (float4)(1.0f, –2.0f, 3.0f, –4.0f);

float4 r = t * 2.5f;
```

Rezultatem będzie wektor o elementach:

```
(2.5f, –5.0f, 7.5f, –10.0f)
```

Powyższą operację można też zapisać jako `r*=2.5f`, gdyż operatory przypisania typu `– =`, `*=`, `/*`, `%=`, `<<=`, `>>=`, `&=`, `|=` oraz operator `^=` również obsługują dane typu wektorowego. Operatory bitowe, np. `&=`, nie obsługują liczb zmiennoprzecinkowych.

Operatory relacji również obsługują dane typu wektorowego, ale dodano pewne reguły dotyczące

składowych wektora. Na przykład operatory relacji, tj. `<, >` etc, zwracają dwie wartości: `0` – jeśli test zawiódł, a `1` – jeśli test był prawdziwy, jednak tylko w przypadku danych typu skalarnego. Podczas porównywania danych wektorowych prawda jest reprezentowana przez `0xFF` (inaczej mówiąc, wszystkie bity są ustawione na jeden), jeśli mamy jedną składową, lub `0xFFFF`, jeśli mamy dwie składowe. W pozostałych przypadkach wartość jest odpowiednio większa. Zapis `0xFF` dla liczb ze znakiem oznacza wartość `-1`, co w przypadku wektorów także oznacza logiczną prawdę, bowiem tylko wartość `0x00` oznacza fałsz.

Na przykład chcemy sprawdzić, czy piąta oraz szósta składowa wektora `b` są mniejsze niż dziesięć:

```
a.s01 = b.s56 < 10;
```

Wyniki porównania trafią do zerowej oraz pierwszej składowej wektora `a`. Mogą one być różne, np. prawda reprezentowana przez `0xFFFFFFFF` trafi do zerowego indeksu oraz zero do składowej o indeksie jeden.

Niestety, instrukcje `while` oraz `if` nie obsługują wektorów. Wolno porównywać składowe, jednak, jeśli stosowane są wyrażenia typu x > 0, gdzie x jest wektorem, należy użyć słowa `all`, gdy chcemy sprawdzić, czy istotnie wszystkie elementy wektora x są większe od zera. Zapis `all` oznacza test, czy wszystkie najbardziej znaczące bity składowej wektora x są ustawione na jeden. Sam test x > 0 sprawdzi tylko, czy poszczególne składowe są większe od zera, ale dopiero zastosowanie `all` upewni nas, że wszystkie składowe spełniły ten warunek:

```
while ( all(x > 0) ) {

    // kod pętli

}
```

Dostępny jest też operator `any` (choć `all` oraz `any` to funkcje) zwracający prawdę, gdy przynajmniej jedna ze składowych ma ustawiony bit najbardziej znaczący.

Działania takie, jak mnożenie i dzielenie na wektorach, oznaczają iż wykonane zostaną operacje na składowych wektorów, np. poszczególne elementy pierwszego wektora zostaną podzielone przez odpowiednie elementy drugiego wektora.

Przekazywanie danych pomiędzy wektorami jest realizowane przez operator przypisania „=”. Jeśli jednak chcemy odczytać dane np. tablicy liczb typu `float` i kolejne cztery elementy tejże tablicy zapisać do wektora, to możemy użyć dostępu o nazwach pól `a.x`, `a.y` etc. Jednak lepiej zastosować przygotowane do tego celu funkcje: `vload` – wczytuje dane do wektora oraz `vstore` – wykonuje operację odwrotną, dane z wektora są przekazywane do innej zmiennej, która nie jest typu wektorowego.

Składnia funkcji `vload` jest następująca:

```
TVector vloadN(size_t offset, const __(MemType) TScalar *memptr);
```

Gdzie `TVector` to zmienna typu wynikowego określana na podstawie kontekstu, inaczej mówiąc, kompilator dopasuje jej typ do zmiennej, do której przypisujemy odczytywane wartości. Wielkość `N` oznacza długość wektora, przy czym dopuszczalne wielkości to 2, 3, 4, 8, 16. Parametr `offset`

oznacza przesunięcie w odczytywanej pamięci względem początku. Drugi parametr to wskaźnik do obszaru, z którego odczytujemy dane do wektora. Może to być pamięć globalna, lokalna, stała czy prywatna.

Adres, z którego odczytamy potrzebne dane, jest określany w następujący sposób:

```
memory address = (memptr + ( offset * N))
```

Jeśli stosujemy wektor o czterech elementach, to używana jest funkcja `vload4`, która odczytuje po cztery elementy i umieszcza je w wektorze wynikowym:

```
vector = vload4(1, array);
```

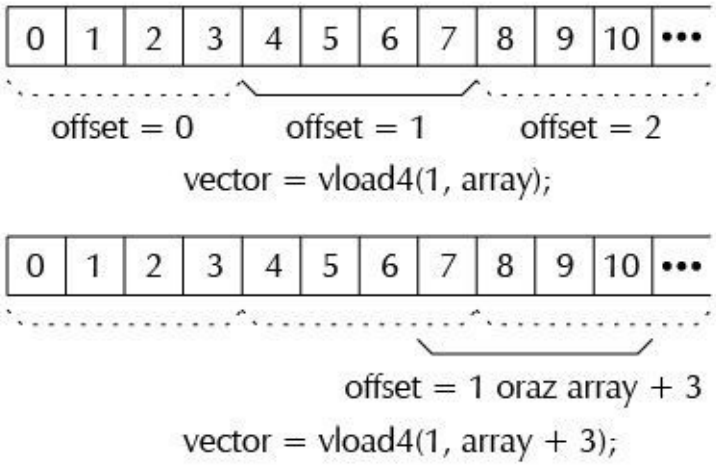
Jeśli do wskaźnika `memptr` zostanie dodana jakaś wartość skalarna, to odczyt nadal zostanie zrealizowany, ale z dodatkowym przesunięciem, gdyż wzór do adresowania przyjmie postać:

```
memory address = ((memptr + UserOffset) + ( offset * N))
```

co oznacza, że np. funkcja `vload4` została uruchomiona w następujący sposób:

```
vector = vload4(1, array + UserOffset)
```

gdzie `UserOffset` jest liczbą całkowitą. Przykład obrazujący tę sytuację został pokazany na rys. 3.3.



Rys. 3.3. Odczyt danych za pomocą `vload4` wraz z dodatkowym przesunięciem

Funkcja `vstore`, która wykonuje zadanie odwrotne, tzn. przesyła dane z wektora do pamięci wskazywanej przez `memptr`, ma nieco inny nagłówek:

```
void vstoreN(TVector vec, size_t offset, __(MEMType) TScalar *memptr);
```

Pierwszy parametr reprezentuje dane źródłowe, czyli wektor o `N` elementach, następny parametr to `offset`, a trzeci to wskaźnik do pamięci docelowej, gdzie dane z argumentu `vec` zostaną umieszczone z odpowiednim przesunięciem. Sposób obliczenia adresu jest identyczny jak w funkcji `vload`. Analogicznie jak w `vload` wielkość `N` oznacza długość wektora, a dopuszczalne wartości to 2, 3, 4, 8, 16. Jednak nie obsługuje ona wszystkich typów pamięci, np. nie ma obsługi pamięci stałej, co jest

zrozumiałe, bowiem z pamięci tego typu możemy tylko odczytywać dane.

Istnieją też funkcje `vload_halfN`, `vstore_halfN` obsługujące typ `half`, jednak nie ma ich w standardzie OpenCL. Aby uzyskać do nich dostęp, należy zastosować rozszerzenie o nazwie `cl_khr_fp16`.

3.5.3. Funkcje obsługujące dane typu wektorowego

Standard OpenCL dostarcza nie tylko duży zestaw funkcji dla danych skalarnych (podane w tab. 3.9–3.13), większość z nich obsługuje także dane typu wektorowego. W szczególności są to funkcje wymienione w tab. 3.14.

Tabela 3.14. Funkcje matematyczne obsługujące dane typu wektorowego

Funkcja	Opis
`floatN fabs(floatN x);`	Wyznacza wartość bezwzględną podanego argumentu, oddzielnie dla każdej składowej wektora (analogicznie funkcjonują wszystkie pozostałe funkcje podane w tej tabeli).
`floatN fma(floatN a, floatN b, floatN c);`	Zwraca wartość wyrażenia `a * b + c`, przy czym mnożenie jest przeprowadzane w pełnej dokładności.
`floatn fmod(floatn x, floatn y);`	Wyznacza resztę z dzielenia `modulo` dla argumentów `x` oraz `y`, inaczej mówiąc oblicza wartość wyrażenia `x – ( y * trunc(y/x))`.
`floatn mad(floatn a, floatn b, floatn c);`	Oblicza wartość wyrażenia `a * b + c`.
`floatn remainder(floatn x, floatn y);`	Zwraca resztę z dzielenia `x` przez `y`, tj. `x – n*y`, gdzie `n` ma wartość `x/y` zaokrągloną do liczby całkowitej.
`floatn remquo(floatn x, floatn y, Ti *quo);`	Zwraca resztę z dzielenia `x` przez `y`, tj. `x – n*y`, gdzie `n` ma wartość `x/y` zaokrągloną do liczby całkowitej, a dodatkowo siedem mniej znaczących bitów ilorazu `x/y` umieszcza w parametrze `quo`.

`floatn rint(floatn x);`	Zwraca liczbę całkowitą najbliższą wartości `x`, jeśli są dwie takie liczby równo odległe od `x`, to zwraca liczbę parzystą.
`floatn round(floatn x);`	Zwraca liczbę całkowitą najbliższą wartości `x`, jeśli są dwie takie liczby równo odległe od `x`, to zwraca tę, która leży dalej od zera.
`floatn ceil(floatn x);`	Zwraca liczbę całkowitą najbliższą wartości `x`, ale większą od niej.
`floatn floor(floatn x);`	Zwraca liczbę całkowitą najbliższą wartości `x`, ale mniejszą od niej.
`floatn trunc(floatn x);`	Zwraca część całkowitą liczby `x`.

Istnieje też grupa funkcji predysponowanych do obsługi danych typu wektorowego, ponieważ implementują one operacje na wektorach, np. obliczają długości wektorów lub dystans pomiędzy nimi. W tabeli 3.15 przedstawiono te funkcje, a także kilka funkcji przeznaczonych do operacji na bitach, które również obsługują dane typu wektorowego.

Tabela 3.15. Funkcje odczytu wartości bitowych obsługujące dane typu wektorowego oraz funkcje geometryczne

Funkcja	Opis
`int any(Tsi x);`	Zwraca wartość `1`, jeśli jest ustawiony dowolny najbardziej znaczący bit w składowych wektora (lub skalarze). W innym przypadku zwraca wartość `0`.
`int all(Tsi x);`	Zwraca wartość `1`, jeśli są ustawione wszystkie najbardziej znaczące bity w składowych wektora (lub skalarze). W innym przypadku zwraca wartość `0`.
`T bitselect(T a, T b, T c);`	Zwraca bity parametru `a`, jeśli odpowiednie bity parametru `c` są ustawione na zero, jeśli nie, to zwraca bity parametru `b`.
`T select(T a, T b,`	Zwraca poszczególne bity parametru `b` lub `a`, jeśli są ustawione najbardziej znaczące bity `c`. Można to zapisać w następujący sposób (`r` oznacza rezultat, MSB – najbardziej znaczący bit):

`Tsi c);`	`r`[i] = jeśli MSB argumentu `c`[i] == 1 ? `b`[i] : `a`[i]
`T select(T a, T b, Tui c);`	W przypadku danych typu skalarnego operacje są następujące: `r= c ? b : a`

Funkcje geometryczne

`float4 cross(float4 p0, float4 p1);`
 	
`float3 cross(float3 p0, float3 p1);`
 	
`double4 cross(double4 p0, double4 p1);`
 	
`double3 cross(double3 p0, double3 p1);`
	Oblicza iloczyn wektorowy elementów `p0`.xyz oraz `p1`.xyz. Czwarty komponent wektora w otrzymuje wartość `0`. Oblicza iloczyn wektorowy tylko dla wektorów o trzech lub czterech komponentach.

`float dot(Tf p0, Tf p1);`
 	
`double dot(Td p0, Td p1);`
	Oblicza wartość iloczynu skalarnego `p0` i `p1`.

`float distance(Tf p0, Tf p1);`
 	
`double distance(Td p0, Td p1);`
	Oblicza odległość między wektorami `p0` oraz `p1`.

`float length(Tf p);`
 	
`double length(gTd p);`
	Oblicza długość wektora bez kontroli nadmiaru oraz niedomiaru.

`Tf normalize(Tf p);`
 	
`Td normalize(Td p);`
	Przeprowadza normalizację podanego wektora, tj. zwrot pozostaje taki sam, ale długość jest równa jedności. Jeśli podany wektor zawiera wyłącznie zera, to zwraca wektor zerowy. Jeśli w jakiejkolwiek składowej znalazła się wartość NaN, to zwraca wartość NaN

`float fast_distance(Tf p0, Tf p1);`	Oblicza odległość między wektorami, co sprowadza się to do wywołania funkcji: `fast_length(p0– p1)`.
`float fast_length(Tf p);`	Oblicza długość wektora za pomocą `half_sqrt`.
`T ffast_normalize(Tf p);`	Wyznacza wartość wektora normalnego, ponownie za pomocą funkcji opartych na danych typu `half`.

3.6. Obsługa siatki obliczeniowej

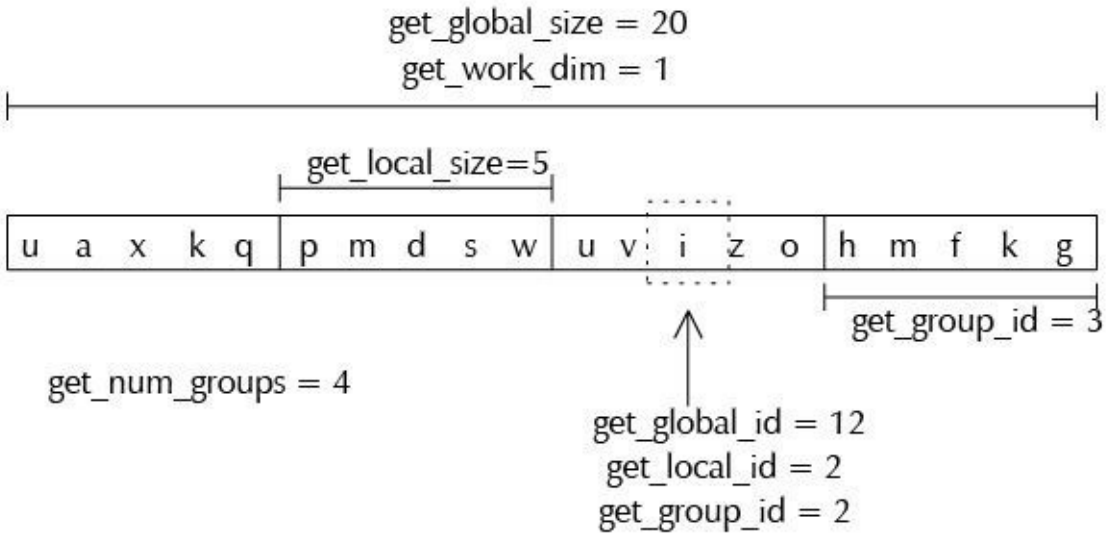
Istnieją dwie siatki obliczeniowe: globalna oraz lokalna. Siatka obliczeniowa może być 1-, 2-, a nawet 3-wymiarowa. Konfiguracja siatki obliczeniowej, a także korzystanie z informacji o siatce obliczeniowej (przestrzeni indeksowej) w jądrze obliczeniowym, mają zasadnicze znaczenie w procedurach obliczeniowych OpenCL. Konfiguracja siatki jest przekazywana w momencie wywołania funkcji `clEnqueueNDRangeKernel`.

Do obsługi siatki globalnej służą funkcje o nazwach: `get_work_dim`, `get_global_size`, `get_global_id` oraz `get_global_offset`. Przy czym ta ostatnia pojawiła się w wersji standardu OpenCL 1.1. W nomenklaturze OpenCL **siatka globalna** jest nazywana też **globalną przestrzenią indeksującą** zawierającą jednostki robocze (ang. *work-items*), natomiast siatka lokalna to grupa robocza (ang. *work-group*).

Liczba wymiarów siatki globalnej jest odczytywana przez funkcję `get_work_dim`. Siatka obliczeniowa może być linią – struktura jednowymiarowa (rys. 3.4), może mieć kształt kwadratu czy prostokąta – struktura dwuwymiarowa, czy też kształt prostopadłościanu – struktura trójwymiarowa.

Liczbę elementów globalnej siatki danego wymiaru można odczytać za pomocą `get_global_size`, gdzie argument to numer wymiaru. Dla siatki liniowej argument musi przyjąć wartość zero.

Odczytanie globalnego unikatowego identyfikatora jednostki roboczej w przypadku, gdy siatka ma strukturę dwuwymiarową, wykonuje się w następujący sposób:



Rys. 3.4. Wartości funkcji stosowanych w kontekście siatki obliczeniowej (przestrzeni indeksującej) na przykładzie siatki liniowej (siatka dla wektora o dwudziestu elementach zawiera cztery grupy robocze po pięć elementów – jednostek roboczych)

```
xid = get_global_id(0);

yid = get_global_id(1);
```

Odczyt pozostałych informacji dotyczących siatki globalnej, jak jej wielkość, czy też informacje o przesunięciu (offset), sprowadza się do wywołania właściwych funkcji:

```
size_t global_size0 = get_global_size(0);

size_t global_size1 = get_global_size(1);

size_t offset_0 = get_global_offset(0);

size_t offset_1 = get_global_offset(1);
```

Należy podkreślić, że kształt siatki globalnej nie wpływa na kształt siatki lokalnej, która może być liniowa, a nawet może to być pojedynczy punkt, podczas gdy siatka globalna pozostaje dwuwymiarowa. Niemniej, w analogiczny sposób odczytujemy współrzędne w ramach siatki lokalnej obowiązującej w grupie roboczej:

```
size_t local_id_0 = get_local_id(0);

size_t local_id_1 = get_local_id(1);
```

Wymiary grupy roboczej odczytujemy za pomocą funkcji `get_local_size`. Podając, analogicznie jak przy wymiarze globalnym, numer wymiaru, dla którego chcemy poznać aktualny rozmiar.

Każda grupa robocza także ma swój własny identyfikator, oddzielnie dla każdego wymiaru. Identyfikator ten, np. dla wymiaru indeksowanego przez zero, odczytamy podobnie jak we wcześniejszych przykładach:

```
size_t grp_id_0 = get_group_id(0);
```

Jeśli zachodzi taka potrzeba, można odczytać liczbę grup lokalnych, przypadających na dany wymiar:

```
size_t grp_size_for_dim0 = get_num_groups(0);
```

W tabeli 3.16 zamieszczono wszystkie funkcje OpenCL C związane z obsługą siatki obliczeniowej, jakie mogą być używane w procedurach obliczeniowych.

Tabela 3.16. Funkcje związane z obsługą siatki obliczeniowej

Funkcja	Opis
	Siatka globalna
`uint get_work_dim();`	Zwraca liczbę wymiarów używanych przez siatkę obliczeniową dla danego jądra obliczeniowego.
`size_t get_global_size(uint dim);`	Zwraca liczbę jednostek roboczych przypadających na podany wymiar.
`size_t get_global_id(uint dim);`	Zwraca identyfikator globalny jednostki roboczej dla podanego wymiaru.
`size_t get_global_offset (uint dim);`	Zwraca wartość początkową przesunięcia (`offset`), jakiego użyto do wyznaczenia wartości globalnego identyfikatora.
	Siatka lokalna
`size_t get_num_groups(uint dim);`	Zwraca liczbę grup roboczych przypadających na podany wymiar.
`size_t get_group_id(uint dim);`	Zwraca identyfikator grupy roboczej, w której znajduje się dana jednostka robocza dla określonego wymiaru.
`size_t get_local_id(uint dim);`	Zwraca lokalny identyfikator jednostki roboczej w ramach grupy roboczej dla określonego wymiaru.

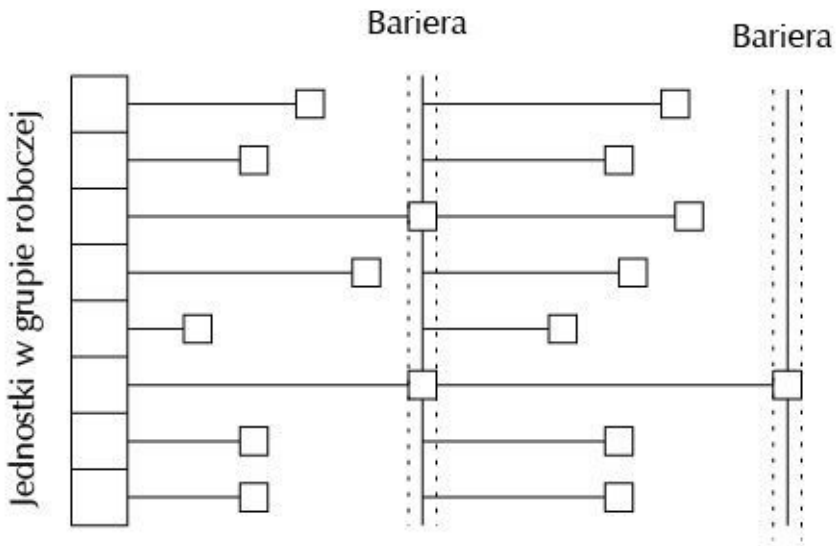
`size_t get_local_size(uint dim);`	Zwraca liczbę jednostek roboczych w grupie roboczej dla podanego wymiaru.

3.7. Funkcje synchronizacji i funkcje atomowe

OpenCL C oferuje jedną funkcję przeznaczoną do synchronizacji wątków w ramach grupy roboczej. Jej nagłówek jest następujący:

```
void barrier( cl_mem_fence_flags flags );
```

Zadaniem funkcji `barrier` jest postawienie bariery, którą dany wątek może przekroczyć tylko wówczas, gdy wszystkie pozostałe wątki w grupie również dotrą do tej samej bariery. Pokazano to na rys. 3.5.



Rys. 3.5. Ilustracja działania bariery w ramach grupy roboczej (dwa pierwsze wątki, które dotarły do pierwszej bariery, muszą poczekać, aż wszystkie pozostałe wątki również dotrą do tego miejsca)

Głównym zadaniem bariery jest regulowanie dostępu do pamięci, zarówno lokalnej, jak też globalnej. Dlatego argument `flags` dopuszcza stosowanie dwóch stałych:

- `CLK_LOCAL_MEM_FENCE`,
- `CLK_GLOBAL_MEM_FENCE`.

Pierwsza stała oznacza postawienie bariery w dostępie do pamięci lokalnej współdzielonej w ramach grupy roboczej. Druga stała ma za zadanie regulowanie dostępu do pamięci globalnej.

Należy pamiętać, że stawianie jakiejkolwiek bariery powinno być zawsze bezwarunkowe. Połączenie bariery z instrukcją warunkową, jak w poniższym przykładzie, może doprowadzić do błędów związanych z obsługą pamięci. Tylko wątki o indeksie globalnym mniejszym niż 10 dotrą do bariery. Część jednostek roboczych (wątków) otrzyma warunek niespełniony i w rezultacie nie dotrze do bariery, co może spowodować zawieszenie się urządzenia obliczeniowego.

```
__kernel void krnBarrier(__global int *inData, __local int *localMem) {

    if (get_global_id(0) < 10) {

        …

        barrier(CLK_GLOBAL_MEM_FENCE);

        …

    }

    else {

        …

        k = inData[0];

        …

    }

}
```

OpenCL oferuje też dodatkowy zestaw funkcji do rozwiązywania konfliktów w dostępie do pamięci, jednak bez konieczności stosowania mechanizmów bariery. Są to tzw. **funkcje atomowe**, realizujące podstawowe operacje, jak zwiększenie lub zmniejszenie wartości zmiennej, operacje arytmetyczne etc. Ideę funkcji atomowych najlepiej zrozumieć na następującym przykładzie. Jeśli dana jednostka robocza próbuje zapisać informacje w określonym miejscu pamięci lokalnej albo globalnej (ale również i odczytać dane z tego miejsca), a następnie inna jednostka robocza również chce wykonać zapis w tym samym miejscu, to jej zadanie zostanie wstrzymane do czasu zakończenia operacji przez pierwszą jednostkę.

Inaczej mówiąc, operacje zapisu we wskazanym obszarze pamięci zostaną wykonane szeregowo w kolejności zgłoszeń. Zapewni to spójność danych, choć niestety oznacza spowolnienie operacji, bowiem praca innych wątków korzystających z obszaru pamięci wskazanego w funkcji atomowej będzie wstrzymana do czasu zakończenia wcześniejszych operacji.

W tabeli 3.17 przedstawiono podstawowy zestaw funkcji oferowanych w standardzie OpenCL. Producenci konkretnych implementacji zazwyczaj dostarczają szerszy zestaw funkcji, dlatego warto zawsze sprawdzić dokumentację producenta, aby odszukać informacje o funkcjach atomowych.

Tabela 3.17. Podstawowy zestaw funkcji atomowych

Funkcja	Opis
`Tu atomic_add (volatile __(g|l) Tu *p, Tu val);`	Wartość `val` jest dodawana do wartości wskazanej przez `p`, co można zapisać jako: `*p += val`.

`Tu atomic_sub (volatile __(g|l) Tu *p, Tu val);`	Wartość `val` jest odejmowania od wartości wskazanej przez `p`, co można zapisać jako: `*p -= val`.
`Tu atomic_inc (volatile __(g|l) Tu *p);`	Zwiększenie wartości wskazywanej przez `p`, tj. wykonanie operacji `*p += 1`.
`Tu atomic_dec (volatile __(g|l) Tu *p);`	Zmniejszenie wartości wskazywanej przez `p`, tj. wykonanie operacji `*p -= 1`.
`Tu atomic_and (volatile __(g|l) Tu *p, Tu val);`	Obliczenie koniunkcji logicznej pomiędzy `val` a wartością wskazaną przez `p`: `*p &= val`.
`Tu atomic_or (volatile __(g|l) Tu *p, Tu val);`	Obliczenie sumy logicznej pomiędzy `val` a wartością wskazaną przez `p`: `*p |= val`.
`Tu atomic_xor (volatile __(g|l) Tu *p, Tu val);`	Obliczenie alternatywy wykluczającej pomiędzy `val` a wartością wskazaną przez `p`: `*p ^= val`.
`Tu atomic_max (volatile __(g|l) Tu *p, Tu val);`	Wybranie wartości większej, z wartości `val` oraz wskazanej przez `p`, i zapisanie maksimum pod adresem `p`.
`Tu atomic_min (volatile __(g|l) Tu *p, Tu val);`	Wybranie wartości mniejszej, z wartości `val` oraz wskazanej przez `p`, i zapisanie minimum pod adresem `p`.
`Tu atomic_xchg (volatile __(g|l) Tu *p, Tu val);`	Funkcja zapisuje pod wskazany adres wartość `val`, ale stara wartość `p` jest zwracana jako wynik.
`float atomic_xchg (volatile __(g|l) float *p, float val)`	Funkcja zapisuje pod wskazany adres wartość `val`, ale stara wartość `p` jest zwracana jako wynik.

`Tu atomic_cmpxchg (volatile __(g|l) Tu *p, Tu cmp, Tu val);`	Wartość wskazywana przez `p` jest porównywana z argumentem `cmp`, i jeśli są równe, to wartość `val` jest kopiowana do `p`, a stara wartość `p` jest zwracana jako rezultat.

Oznaczenia: `Tu` – typ `int` lub `unsigned int`, `__(g|l)` – atrybut rodzaju pamięci globalnej lub lokalnej

3.8. Funkcje do obsługi obrazów

Zaletą OpenCL C jest ułatwiona obsługa obrazów, dodano bowiem kilka przydatnych funkcji do ich łatwiejszego przetwarzania. Istnieją trzy podstawowe typy obrazów `image_1d`, `image_2d` oraz `image_3d`, reprezentujące odpowiednio obrazy 1-, 2- oraz 3-wymiarowe. Język OpenCL C oferuje dwie podstawowe funkcje do operacji na obrazie, to jest:

• `read_image(f|i|ui)` – do odczytu informacji o pikselu ze wskazanego obrazu,

• `write_image(f|i|ui)` – do zapisu informacji o pikselu we wskazanym obrazie.

Funkcje te występują w kilku odmianach (tab. 3.18), stąd też pojawiły się dodatkowe oznaczenia „`f`” „`i`” oraz „`ui`” dotyczące typu danych, jakie są zwracane. Jednakże odczytanie informacji o pikselu, który ma trzy główne składowe RGB oraz kanał alfa, można zrealizować w następujący bezpośredni sposób:

```
int2 input_coord;

uint4 pixel;

pixel = read_imageui(src_image, sampler, input_coord );
```

Zastosowana funkcja `read_imageui` odczytuje informacje za pomocą tzw. **samplera** (o którym będzie mowa dalej), natomiast składowe piksela to liczby typu `unsigned int`. Jeśli trzeba odczytać informacje o pikselu w postaci liczb typu `float`, to stosuje się funkcję `read_imagef`.

Zapis piksela do obrazu jest równie prosty – trzeba podać współrzędne wyjściowe, nie potrzeba stosować samplera:

```
int2 output_coord;

write_imageui(dst_image, output_coord, pixel);
```

W zależności od typu danych opisujących obraz (np. kolory pikseli mogą być podane jako liczby rzeczywiste albo całkowite ze znakiem lub bez znaku) należy stosować odpowiednie odmiany funkcji odczytujących i zapisujących. Należy też zwrócić uwagę na parametry `coord` oraz `output_coord`, stanowiące współrzędne, które mogą być liczbami całkowitymi bezpośrednio odpowiadającymi rozdzielczości obrazu. Jednakże mogą to być też liczby znormalizowane w zakresie od 0 do 1, jeśli zastosowano wartości normalne bez znaku, a jeśli ze znakiem, to od –1 do 1. Dobór sposobu adresowania zależy od realizowanego zadania, zobaczymy to na przykładzie skalowania obrazu (rozdz. 5.9.1).

Bardzo ważną rolę w procesie odczytu pełni tzw. **sampler**. Jest to obiekt odpowiedzialny za sposób odczytu informacji o pikselu z obrazu. Przykładowa deklaracja obiektu samplera bezpośrednio

w jądrze obliczeniowym jest następująca:

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;
```

Tabela 3.18. Zestawienie funkcji do odczytu oraz zapisu danych do obrazów

Funkcja	Opis
Funkcje do odczytu	
`float4 read_imagef(image2d_t img, sampler_t sampler, int2|float2 coord);`	Odczytuje informacje o pikselu z obrazu 2D i zapisuje dane jako wektor `float4`.
`int4 read_imagei(image2d_t img, sampler_t sampler, int2|float2 coord);`	Odczytuje informacje o pikselu z obrazu 2D i zapisuje dane jako wektor `int4`.
`uint4 read_imageui(image2d_t img, sampler_t sampler, int2|float2 coord);`	Odczytuje informacje o pikselu z obrazu 2D i zapisuje dane jako wektor `uint4`.
`float4 read_imagef(image3d_t img, sampler_t sampler, int4|float4 coord);`	Odczytuje informacje o pikselu z obrazu 3D i zapisuje dane jako wektor `float4`.
`int4 read_imagei(image3d_t img, sampler_t sampler, int4|float4 coord);`	Odczytuje informacje o pikselu z obrazu 3D i zapisuje dane jako wektor `int4`.
`uint4 read_imageui(image3d_t img, sampler_t sampler, int4|float4 coord);`	Odczytuje informacje o pikselu z obrazu 3D i zapisuje dane jako wektor `uint4`.
Funkcje do zapisu	
`void write_imagef(image2d_t img, int2 coord, float4 color);`	Zapisuje piksel opisany przez typ `float4` do obrazu 2D.

`void write_imagei(image2d_t img, int2 coord, int4 color);`	Zapisuje piksel opisany przez typ `int4` do obrazu 2D.
`void write_imageui(image2d_t img, int2 coord, uint4 color);`	Zapisuje piksel opisany przez typ `uint4` do obrazu 2D.
`void write_imagef(image3d_t img, int4 coord, float4 color);`	Zapisuje piksel opisany przez typ `float4` do obrazu 3D.
`void write_imagei(image3d_t img, int4 coord, int4 color);`	Zapisuje piksel opisany przez typ `int4` do obrazu 3D.
`void write_imageui(image3d_t img, int4 coord, uint4 color);`	Zapisuje piksel opisany przez typ `uint4` do obrazu 3D.

Zadeklarowany został obiekt, który wykorzystuje trzy podstawowe wartości, jakie powinny znaleźć się w deklaracji każdego samplera:

• typ stosowanych współrzędnych, dostępne wartości: `CLK_NORMALIZED_COORDS_FALSE`, `CLK_NORMALIZED_COORDS_TRUE`,

• rodzaj adresowania, dostępne wartości: `CLK_ADDRESS_NONE`, `CLK_ADDRESS_CLAMP`, `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_REPEAT`, `CLK_ADDRESS_MIRRORED_REPEAT`,

• rodzaj stosowanego filtru, dostępne wartości: `CLK_FILTER_NEAREST`, `CLK_FILTER_LINEAR`.

Wartości współrzędnych mogą być liczbami znormalizowanymi od 0 do 1 lub od –1 do +1. W przypadku `CLK_NORMALIZED_COORDS_FALSE` współrzędne przyjmują wartości odnoszące się do rozdzielczości obrazu, np. od 0 do 1023, jeśli obraz ma szerokość 1024 pikseli.

Znaczenie poszczególnych trybów adresowania można krótko przedstawić następująco:

• `CL_ADDRESS_NONE` – nie jest stosowane dodatkowe postępowanie w przypadku przekroczenia współrzędnych piksela.

• `CL_ADDRESS_CLAMP` – jeśli następuje próba odczytu koloru spoza obszaru obrazu, to zwracany kolor ma wartość koloru domyślnego, zazwyczaj jest to kolor czarny.

• `CL_ADDRESS_CLAMP_TO_EDGE` – jeśli następuje próba odczytu koloru spoza obszaru obrazu, to zwracany kolor ma wartość koloru znajdującego się na krawędzi obrazu.

• `CL_ADDRESS_REPEAT` – w przypadku przekroczenia rozmiaru obrazu dla danego wymiaru, nowe współrzędne są obliczane wg wzoru `x mod X`, gdzie `x` to współrzędna, a `X` to rozmiar obrazu, np. szerokość.

• `CL_ADDRESS_MIRRORED_REPEAT` – w przypadku przekroczenia rozmiaru obrazu, nowe współrzędne

są generowane na podstawie lustrzanego odbicia obrazu. Ten typ adresowania może być stosowany tylko w przypadku użycia adresowania normalnego.

Dopuszczalne są tylko dwie wartości filtrów, jakie można stosować. Wartość `CLK_FILTER_NEAREST` oznacza, że przy próbie odczytu wartości piksela może ona być powielana. Taka sytuacja może się pojawić np. przy odczycie z obrazu dla współrzędnych w trybie adresowania normalnego. Krótko mówiąc, obliczone współrzędne będą wskazywać na adres między dwoma istniejącymi pikselami. W efekcie tak odczytany obraz będzie niskiej jakości, bowiem brakujący piksel przyjmie wartość jednego z sąsiadów. Lepszym rozwiązaniem jest zastosowanie filtru `CLK_FILTER_LINEAR`, który przy próbie odczytania piksela leżącego pomiędzy istniejącymi pikselami będzie generował nowy piksel na podstawie istniejącego otoczenia. W efekcie otrzymany obraz będzie znacznie lepszej jakości. Wykorzystamy tę możliwość w przykładzie skalowania obrazów w rozdz. 5.9.1.

Uzupełnieniem funkcji do obsługi obrazów jest zestaw funkcji pomocniczych do odczytu dodatkowych informacji, m.in. o szerokości, wysokości czy głębokości przetwarzanego obrazu. Zostały one krótko przedstawione w tab. 3.19.

Tabela 3.19. Dodatkowe funkcje wspomagające przetwarzanie obrazów, typ `imageXd_t` oznacza typ reprezentujący obraz dla danej funkcji

Funkcja	Opis
`int get_image_width(imageXd_t image);`	Szerokość obrazu 2D lub 3D.
`int get_image_height(imageXd_t image);`	Wysokość obrazu 2D lub 3D.
`int get_image_depth(image3d_t image);`	Głębokość obrazu 3D.
`int2 get_image_dim(image2d_t image);`	Wymiary obrazu 2D zostaną zwrócone jako wektor typu `int2`.
`int4 get_image_dim(image3d_t image);`	Wymiary obrazu 3D zostaną zwrócone jako wektor typu `int4` (ostatnia składowa zostanie ustalona na zero).
`int get_image_channel_data_type (imageXd_t);`	Zwraca typ danych opisujących jeden kanał kolorów.

`int get_image_channel_order (imageXd_t image);`	Zwraca sposób ułożenia kolorów opisujących jeden piksel.

Oprócz funkcji do przetwarzania obrazu na poziomie jądra obliczeniowego dostępne są też funkcje API OpenCL. Jednak zostaną one przedstawione w kontekście przykładów, jakie są omawiane w rozdz. 5.9.

3.9. Preprocesor OpenCL C

Język OpenCL C jest oparty na standardzie C99 języka C, więc dostępny jest w nim także preprocesor. Obsługiwane są następujące dyrektywy preprocesora:

- `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`,
- `#include`, `#define`, `#undef`,
- `#line`, `#error`, `#pragma`,

a także operator `defined`, który używany jest w połączeniu z dyrektywą `#if`.

Kompilacja warunkowa danego fragmentu kodu jest przeprowadzana tak samo jak w tradycyjnym języku C. Oto przykład możliwości wyboru odpowiedniego fragmentu kodu do kompilacji dla dwóch stałych `CODE_V1` oraz `CODE_V2` w zależności od wartości `CODE_TYPE`:

```
#define CODE_V1 1

#define CODE_V2 2

__kernel void name_fnc(global const float *a, __global float *b)

{

    size_t gid = get_global_id(0);

    …

#if CODE_TYPE == CODE_V1

b[gid] = 2 * a[gid];

#elif CODE_TYPE == CODE_V2

b[gid] = 2 / a[gid];

#endif

    …

}
```

Pozostaje do rozwiązania problem przekazania wartości `CODE_TYPE`, co można zrobić za pomocą funkcji `clBuildProgram`, która dokonuje kompilacji programu (wcześniej należy utworzyć obiekt

cl_kernel_program za pomocą funkcji clCreateProgramWithSource):

```
cli_err_num = clBuildProgram(cl_kernel_program, clui_num_devices, platform_devices, "-DCODE_TYPE = 1”, NULL, NULL);
```

W czwartym parametrze możemy podać wyrażenie –D, które utworzy odpowiednią definicję dla preprocesora. Można tam też dodać inne opcje obsługiwane przez kompilator OpenCL C.

W kontekście preprocesora oraz OpenCL C szczególnie ważną rolę pełni dyrektywa #pragma. Zwłaszcza w kontekście rozszerzeń, jakie mogą być oferowane przez różnych producentów urządzeń obliczeniowych współpracujących ze standardem OpenCL. W najbardziej ogólnym przypadku składnia dla polecenia #pragma jest następująca:

```
#pragma TOKEN OPCJE
```

Trudno jest określić postać parametrów dla dyrektywy #pragma, bowiem nie jest ona ściśle ustalona. W przypadku OpenCL C #pragma pozwala włączać oraz wyłączać dostępne rozszerzenia. W takim przypadku można precyzyjniej określić postać polecenia:

```
#pragma OPENCL EXTENSION nazwa_rozszerzenia : opcja
```

gdzie: „nazwa_rozszerzenia” – nazwa funkcjonalności, jaką chcemy włączyć bądź wyłączyć za pomocą wyrażenia „opcja”.

Nazwy rozszerzeń są predefiniowane, np. obsługa liczb o podwójnej precyzji jest reprezentowana przez zapis cl_khr_fp64.

Włączenie obsługi liczb o podwójnej precyzji realizuje się za pomocą makra następującej postaci:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
```

W kodzie źródłowym jądra obliczeniowego można wykryć, czy została włączona obsługa typu double, wykorzystując makro cl_khr_fp64, które powinno być zadeklarowane po wyłączeniu rozszerzenia:

```
#if defined(cl_khr_fp64)

    // kod dla double

#else

    // kod gdy brakuje typu double

#endif
```

Wyłączenie rozszerzenia sprowadza się do podania w zapisie makra po dwukropku wyrażenia disable. Możliwe jest także wyłączenie wszystkich rozszerzeń w następujący sposób:

```
#pragma OPENCL EXTENSION all : disable
```

W OpenCL C, pomijając rozszerzenia, istnieje tylko jedna wbudowana pragma o nazwie FP_CONTRACT:

```
#pragma OPENCL FP_CONTRACT przełącznik
```

gdzie: `przełącznik` – jedno z trzech możliwych wyrażeń `ON`, `OFF` lub `DEFAULT`.

Zadaniem tej pragmy jest uruchomienie tzw. kontrakcji wyrażeń, tj. zamiany podanego wyrażenia na inny ciąg wyrażeń, które obliczą tę samą wartość. Dotyczy to np. wyrażeń typu a*b + c, które mogą zostać zamienione na instrukcje typu FMA, tj. jedną operację realizującą iloczyn oraz sumę. Pragma `FP_CONTRACT` działa też obszarowo, czyli od miejsca jej wystąpienia aż do kolejnego wystąpienia `FP_CONTRACT`, które np. może wyłączyć stosowanie kontrakcji wyrażeń.

Dostępne są także dodatkowe makra przeniesione ze standardu C99, np. numer linii, który można wykorzystywać w jądrze obliczeniowym:

• makro `__FILE__` zawiera nazwę aktualnego pliku źródłowego, analogicznie jak dla preprocesora w standardzie C99,

• makro `__LINE__` reprezentuje numer linii w aktualnym pliku źródłowym w postaci liczby całkowitej,

• definicje `CL_VERSION_1_0`, `CL_VERSION_1_1`, `CL_VERSION_1_2` etc., reprezentują numer standardu języka OpenCL C, jaki jest obsługiwany przez kompilator, numer jest zapisany w postaci liczby całkowitej jako wartości 100, 110, 120 etc.

Dostępne są także makra charakterystyczne dla OpenCL:

• makro `__OPENCL_VERSION__` – zawiera liczbę całkowitą, która reprezentuje numer wersji OpenCL, jaka jest obsługiwana przez dane urządzenie obliczeniowe, dotyczy to języka OpenCL C oraz samego urządzenia. Jeśli obsługiwany jest standard OpenCL C 1.1, to wartością makra będzie 110, jeśli 1.2 to wartość makra wynosi 120 etc.,

• `__ENDIAN_LITTLE__` – obecność tego makra świadczy o tym, że dane urządzenie OpenCL posiada organizację `little-endian`, brak tego makra oznacza, iż mamy do czynienia z urządzeniem o organizacji `big-endian`,

• `__kernel_exec(X, typen)` (a także `kernel_exec(X, typen)`) – makro, które skraca definicję atrybutu opisującego wymagania co do typu siatki obliczeniowej. Po rozwinięciu przyjmuje następującą postać:

```
__kernel __attribute__((work_group_size_hint(X, 1, 1))) \
__attribute__((vec_type_hint(typen)))
```

• `__IMAGE_SUPPORT__` – obecność tego makra oznacza, że urządzenie OpenCL, dla którego kompilowana jest procedura obliczeniowa, obsługuje przetwarzanie obrazów,

• `__FAST_RELAXED_MATH__` – obecność definicji tego makra oznacza, że podczas kompilacji i budowy procedury obliczeniowej zastosowano opcję `-cl-fastrelaxed-math` (obliczenia numeryczne są szybsze kosztem dokładności).

3.10. Ograniczenia OpenCL C

Podsumowując rozdział opisujący język OpenCL C, warto przypomnieć, jakie główne ograniczenia znajdują się w odmianie języka C dla jąder obliczeniowych. Wynikają one z przeznaczenia procedur tworzonych tylko do realizacji różnego typu obliczeń (nie ma możliwości obsługi plików), ale też z ograniczeń urządzeń, jakie obecnie są stosowane do implementacji standardu OpenCL. Lista najważniejszych ograniczeń jest następująca:

• brak dostępu do plików nagłówkowych standardowych dla języka C, np. `stdlib.h`, `complex.h`, `time.h`,

• nie można stosować funkcji rekurencyjnych,

• nie ma dostępu do pól bitowych znanych z języka C,

• tablice o zmiennej długości nie są obsługiwane,

• nie są obsługiwane makra do obsługi wielu parametrów przekazywanych do funkcji,

• nie jest możliwe stosowanie wskaźników do funkcji,

• słowa kluczowe reprezentujące tryb dostępu, tj. `extern`, `static`, `auto`, `register`, nie są obsługiwane przez OpenCL C,

• argumenty dla jądra obliczeniowego, które są wskaźnikami, wymagają stosowania atrybutów `global`, `constant`, `local`, nie jest też możliwe stosowanie podwójnych wskaźników,

• argumenty jądra obliczeniowego nie mogą być danymi następujących typów: `bool`, `half`, `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t`, `event_t`. Rezultat musi zawsze być typu `void`,

• argumenty oparte na strukturze nie mogą mieć pól opartych na obiektach OpenCL C,

• typy danych dla obrazu `image2d_t`, `image3d_t` mogą być przekazywane do jądra tylko przez argumenty jądra obliczeniowego, nie można ich deklarować jako zmiennych lokalnych, co oznacza, że nie mogą też być deklarowane jako zmienne o dostępie `private`, `local` czy też `constant`, i nie mogą być użyte jako pola struktury. Nie można także tworzyć tablic, których elementami będą obrazy typu `image2d_t` bądź `image3d_t`,

• podobne ograniczenia dotyczą również typu `sampler_t`, może on być stosowany jako argument funkcji nie tylko samego jądra obliczeniowego, lecz także innych funkcji wywoływanych przez jądro obliczeniowe, może też być zadeklarowany jako zmienna, ale nie może być polem struktury czy typem tablicy, a także typem powrotnym. Nie można też tworzyć zmiennych opartych na danych typu `sampler_t` z atrybutami `local` oraz `global`,

• ograniczenia dla typu `event_t` również są podobne do ograniczeń typu `sampler_t`, bowiem można tworzyć argumenty oparte na tym typie, a także zmienne lokalne. Możliwe jest również tworzenie tablic elementów typu `event_t`. Dopuszczalne są wskaźniki do zdarzeń, choć nie można tworzyć pól struktur opartych na typie zdarzeniowym. Nie można tworzyć zmiennych opartych na typie `event_t` z atrybutami `constant`, `local` oraz `global`.

Przypisy

[1] Wartość stałej `CL_DEVICE_ADDRESS_BITS` można odczytać z profilu urządzenia, informacje o sposobie odczytywania parametrów urządzenia obliczeniowego zostały przedstawione w rozdz. 4.1.

[2] Nowo przygotowywany standard OpenCL 2.0 będzie już pozwalał, aby na poziomie jądra obliczeniowego można było uruchomić inne jądro obliczeniowe wraz z na nowo skonfigurowaną siatką obliczeniową.

4 OpenCL – łatwe przykłady na początek

W poprzednich rozdziałach omówione zostały pojęcia dotyczące technologii OpenCL oraz języka OpenCL C, jaki jest stosowany do tworzenia tzw. jąder, czy też kerneli obliczeniowych. Poznane informacje wystarczą do napisania pierwszych przykładów praktycznego wykorzystania możliwości API technologii OpenCL. Nasze pierwsze przykłady będą następujące:

• wykorzystanie podstawowych informacji o dostępnych urządzeniach obliczeniowych (rozdz. 4.1),

• przykład „Witaj Świecie” (rozdz. 4.2), czyli pierwsze jądro obliczeniowe,

• dodawanie dwóch wektorów tej samej wielkości (rozdz. 4.3),

• program pomocniczy obrazujący postać przestrzeni indeksującej/siatki obliczeniowej (rozdz. 4.4),

• wykorzystanie jądra obliczeniowego do wykonania tzw. zadań OpenCL (rozdz. 4.5),

• sposób przekazywania dodatkowych definicji lub opcji dla jądra obliczeniowego oraz odczyt informacji o ewentualnych błędach, jakie zostaną wykryte w trakcie kompilacji jądra obliczeniowego (rozdz. 4.6).

Przykłady pozwolą zapoznać się z API OpenCL dla języka C, a także C++, gdyż oficjalnie Khronos Group dostarcza odpowiedni plik nagłówkowy dla języka C++. Użycie API dla C++ upraszcza kilka zadań, jakie trzeba wykonać przed przystąpieniem do obliczeń OpenCL, szczególnie jeśli porównamy API C++ do podstawowego interfejsu OpenCL dla języka C.

4.1. Informacje o urządzeniu obliczeniowym

W pierwszym przykładzie przedstawimy podstawowe informacje o dostępnych urządzeniach obliczeniowych zgodnych z technologią OpenCL. Wszystkie przykłady w tej książce są przeznaczone dla komputerów PC. Urządzenia obliczeniowe w komputerach PC to zazwyczaj karty graficzne (w skrócie GPU), bądź też procesor główny komputera (w skrócie CPU). Program może zatem wykazać, że mamy np. dwa dostępne urządzenia obliczeniowe zgodne z OpenCL, tj. CPU oraz GPU.

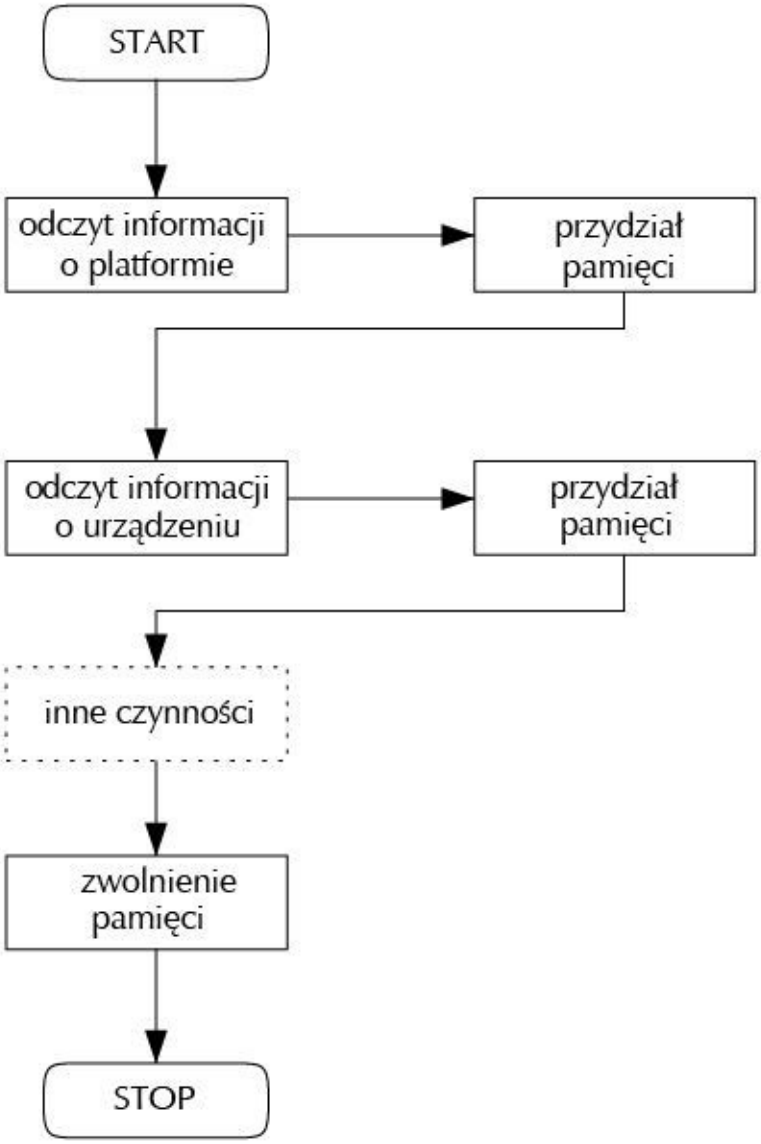
W tym miejscu nie będziemy zajmować się aspektami związanymi z kompilacją. Podstawowe informacje o kompilacji przykładów dla systemów Linux, Windows oraz MacOS przedstawione zostały w Dodatku A.

Odczyt informacji wymaga od nas tylko dwóch czynności. Pierwsza to odczyt informacji o platformie, co można także przyrównać do zainicjowania OpenCL, a druga to odczytanie dostępnych urządzeń obliczeniowych. Jeśli uda się uzyskać dostęp do platformy, to możemy założyć, że urządzenie, na którym uruchomiono nasz program, oferuje możliwość korzystania z technologii OpenCL. Po uzyskaniu dostępu do platformy możemy już odczytać m.in. nazwę platformy, informacje o jej dostawcy czy listę dostępnych rozszerzeń obsługiwanych przez platformę OpenCL.

Parametrów do odczytania jest wiele, m.in. rozmiary siatki obliczeniowej, typ urządzenia obliczeniowego, obsługa liczb podwójnej precyzji i inne opisujące możliwości danego urządzenia

obliczeniowego.

Zazwyczaj obiekty platformy oraz dostępnych urządzeń są wystarczające do odczytania informacji o możliwościach urządzenia obliczeniowego. Jednak np. informacje o obsługiwanych formatach obrazu wymagają uzyskania kontekstu obliczeniowego. Niemniej, jeśli udało się nam uzyskać informacje o platformie i urządzeniach, to mamy za sobą pierwszy etap inicjowania OpenCL. Można to podsumować za pomocą prostego schematu blokowego (rys. 4.1).



Rys. 4.1. Zadania wykonywane w pierwszym etapie inicjowania OpenCL

4.1.1. Wersja dla API w języku C

Zamieniając opisane czynności oraz schemat z rys. 4.1 na program[1] w języku C należy dołączyć pliki nagłówkowe związane z technologią OpenCL. Choć główny plik nagłówkowy *opencl.h* ma zawsze taką samą nazwę, to w zależności od systemu operacyjnego znajduje się on w katalogu *OpenCL* w systemie MacOS X lub *CL* w systemie Windows bądź Linux.

```
#if defined (__APPLE__) || defined(MACOSX)

    #include <OpenCL/opencl.h>
```

```
#else

    #include <CL/opencl.h>

#endif
```

Potrzebne także będą zmienne pomocnicze np. do indeksowania informacji o poszczególnych platformach bądź urządzeniach:

```
cl_uint ocl_plat_idx = 0;

cl_uint ocl_dev_idx = 0;

cl_uint clui_num_platforms = 0;

cl_uint clui_num_devices = 0;
```

Przyda się zmienna przechowująca ostatni kod błędu:

```
cl_int cli_err_num;
```

Powyższe zmienne to liczby całkowite ze znakiem oraz bez znaku. Jednak zamiast oznaczeń typów C lub C++ należy stosować oznaczenia typów OpenCL, żeby uniknąć dodatkowych niepotrzebnych konwersji.

Następne zmienne odnoszą się do typu urządzenia oraz platformy i urządzeń obliczeniowych:

```
cl_device_type device_type;

cl_platform_id *opencl_platforms = NULL;

cl_device_id *platform_devices = NULL;
```

Po tych przygotowaniach możemy wykonać pierwszy etap inicjowania, tj. odczyt informacji o liczbie dostępnych platform obliczeniowych:

```
cli_err_num = clGetPlatformIDs(0, NULL, &clui_num_platforms);
```

Koniecznie trzeba sprawdzić, czy udało się odczytać informacje o platformach:

```
if (cli_err_num != CL_SUCCESS) {

    // brak dostępu do plaformy (lub platform)

    // zgodnych z OpenCL

}
```

Jeśli się nie udało, to otrzymaliśmy sygnał, że nie mamy dostępu do technologii OpenCL w środowisku, w którym uruchomiono naszą aplikację.

Gdy wywołanie `clGetPlatformIDs` zakończyło się sukcesem, to w zmiennej `clui_num_platforms` pojawi się wartość oznaczająca liczbę dostępnych platform OpenCL. Kolejna czynność to odczytanie

obiektów reprezentujących poszczególne platformy. Jednak należy wcześniej przydzielić pamięć, w naszym przypadku informacje o platformach zostały zapisane w zmiennej wskaźnikowej `opencl_platforms`:

```
cl_platform_id *opencl_platforms = NULL;
```

Przydział pamięci wykonujemy tradycyjnie za pomocą funkcji `malloc`:

```
opencl_platforms = (cl_platform_id*)malloc(clui_num_platforms *sizeof(cl_platform_id));
```

Odczyt informacji do przydzielonej pamięci realizujemy ponownie za pomocą `clGetPlatformIDs`:

```
cli_err_num = clGetPlatformIDs (clui_num_platforms, opencl_platforms, NULL);
```

Żeby odczytać informacje o platformie, a nie liczbę dostępnych platform, stosujemy odpowiedni układ parametrów. W pierwszym argumencie wstawiamy liczbę dostępnych platform, w drugim zmienną z przydzieloną pamięcią, a w trzecim wartość `NULL`. W ten sposób odczytamy informacje o platformach. Podobnie jak poprzednio, sprawdzamy wartość `cli_err_num`, aby stwierdzić, czy operacja zakończyła się sukcesem.

Dane zapisane w zmiennej `opencl_platforms` można traktować jak tablicę. Informacje o pierwszej platformie są zapamiętane pod indeksem `0`, o drugiej pod indeksem `1` i tak dalej. Odczytanie nazwy platformy realizuje się za pomocą funkcji `clGetPlatformInfo`, która umożliwia też odczyt wielu innych informacji o platformie OpenCL, np. informacje o dostawcy, numer wersji, a także liczbę urządzeń obliczeniowych, bowiem obliczenia nie są zlecanie bezpośrednio do platformy, lecz do urządzenia obliczeniowego.

Odczytanie nazwy platformy wymaga określenia bufora (zmienna `buffer1`), gdzie zostanie umieszczony ciąg znaków reprezentujący nazwę platformy:

```
char buffer1[4096];

cli_err_num = clGetPlatformInfo( opencl_platforms[0], CL_PLATFORM_NAME, 4096, buffer1,
NULL);
```

Jeśli chcemy odczytać nazwę drugiej platformy, to wystarczy odwołać się do pierwszego indeksu tablicy `opencl_platforms` (zerowy indeks tradycyjnie w języku C wskazuje na pierwszy element tablicy). Wyświetlenie nazwy sprowadzi się do użycia polecenia `printf`:

```
printf(”Nazwa platformy: %s\n”, &buffer1[0]);
```

Podobnie odbywa się pobranie wersji platformy:

```
cli_err_num = clGetPlatformInfo ( opencl_platforms[ocl_plat_idx], CL_PLATFORM_VERSION,
4096, buffer1, NULL);
```

Odczytane informacje to także ciąg znaków, który można wyświetlić za pomocą `printf`, lub porównać za pomocą `strcmp`. Zakładamy, że numer interesującej nas platformy obliczeniowej zawarty jest w zmiennej `ocl_plat_idx`. W naszym programie odczytującym informacje dotyczące urządzeń OpenCL mamy pętlę `for`, której zadaniem jest przeglądnięcie wszystkich dostępnych platform:

```
for( ocl_plat_idx=0; ocl_plat_idx < clui_num_platforms; ocl_plat_idx++)

{

            cli_err_num   =   clGetPlatformInfo(   opencl_platforms   [ocl_plat_idx],
CL_PLATFORM_NAME, 4096, buffer1, NULL);

    if (cli_err_num == CL_SUCCESS)

    {

        // informacje o platformie oraz urządzeniach

    }

}
```

W pętli za każdym razem próbujemy odczytać nazwę platformy, a jeśli uda się wykonać odczyt, to mamy informację, że platforma jest aktywna i możemy odczytać dalsze informacje, np. dostępne rozszerzenia w postaci ciągu nazw rozdzielonych spacjami. Możemy wyświetlić odczytany ciąg znaków bezpośrednio za pomocą `printf` lub też wykorzystać fakt, iż spacja pełni rolę separatora i za pomocą np. funkcji z biblioteki standardowej języka C `strtok` wydzielić poszczególne nazwy, a następnie wyświetlić nazwy rozszerzeń oddzielnie w każdej linii:

```
printf(”\tDostępne rozszerzenia:\n”);

char *tmpstr;

tmpstr = strtok(&buffer1[0], ” ”);

while(tmpstr!=NULL) {

    printf(”\t\t %s\n”, tmpstr);

    tmpstr = strtok(NULL, ” ”);

}
```

Istotnym etapem po uzyskaniu dostępu do platformy jest uzyskanie dostępu do urządzeń obliczeniowych. Każda platforma ma własne urządzenia obliczeniowe, dlatego podczas odczytu liczby dostępnych urządzeń obliczeniowych musimy wskazać, z której platformy obliczeniowej będziemy korzystać oraz typ urządzeń obliczeniowych. Dla projektowanego programu zakładamy, że interesują nas wszystkie urządzenia obliczeniowe[2], jakie mamy dostępne, dlatego w drugim argumencie podajemy stałą `CL_DEVICE_TYPE_ALL`. Identycznie jak wcześniej, korzystamy ze zmiennej `ocl_plat_idx`, aby wskazać interesującą nas platformę:

```
cli_err_num = clGetDeviceIDs ( opencl_platforms[ocl_plat_idx], CL_DEVICE_TYPE_ALL, 0,
NULL, &clui_num_devices);
```

Liczba dostępnych urządzeń obliczeniowych zostanie zapisana w zmiennej `clui_num_devices`. Mając tę wiedzę, możemy przydzielić właściwą ilość pamięci:

```
platform_devices = (cl_device_id*)malloc(sizeof(cl_device_id) *clui_num_devices);
```

Podobnie jak w przypadku odczytu obiektów reprezentujących platformy obliczeniowe, odczytujemy urządzenia, wywołując ponownie `clGetDeviceIDs`, gdzie w trzecim argumencie podajemy liczbę urządzeń obliczeniowych, a w czwartym wskaźnik, pod którym zostaną umieszczone informacje reprezentujące poszczególne urządzenia:

```
cli_err_num = clGetDeviceIDs ( opencl_platforms[ocl_plat_idx], CL_DEVICE_TYPE_ALL, clui_num_devices, platform_devices, &clui_num_devices);
```

Należy sprawdzać, czy odczyt informacji udało się zrealizować. Jeśli tak, to można przystąpić do odczytu informacji o urządzeniu. Na przykład nazwę urządzenia odczytujemy podobnie jak nazwę platformy:

```
cli_err_num = clGetDeviceInfo(platform_devices[ocl_dev_idx], CL_DEVICE_NAME, 4096, buffer1, NULL);

printf(”\tNazwa urządzenia: %s\n”, buffer1);
```

Analogicznie odczytamy wersję języka OpenCL C, jaka jest preferowana przez wybrane urządzenie obliczeniowe:

```
cli_err_num = clGetDeviceInfo(platform_devices[ocl_dev_idx], CL_DEVICE_OPENCL_C_VERSION, 4096, buffer1, NULL);

printf(”\tWersja OpenCL C: %s\n”, buffer1);
```

Odczytanie potrzebnych informacji w obydwu przypadkach sprowadziło się tylko do podania odpowiednich stałych, jak `CL_DEVICE_NAME` oraz `CL_DEVICE_OPENCL_C_VERSION`. Część odczytywanych informacji jest reprezentowana za pomocą innych typów danych, np. typ urządzenia obliczeniowego wymaga stosowania typu danych o nazwie `cl_device_type`:

```
cl_device_type device_type;
```

Stosując stałą `CL_DEVICE_TYPE`, informacje o typie urządzenia odczytamy w następujący sposób:

```
cli_err_num = clGetDeviceInfo(platform_devices[ocl_dev_idx], CL_DEVICE_TYPE, sizeof(device_type), &device_type, NULL);
```

Wykrycie typu urządzenia, z jakim mamy do czynienia, realizujemy, wykonując test bitowy z odpowiednią flagą:

```
if ( device_type & CL_DEVICE_TYPE_CPU ) printf(”\tTyp urządzenia: CPU.\n”);

if ( device_type & CL_DEVICE_TYPE_GPU ) printf(”\tTyp urządzenia: GPU.\n”);

if ( device_type & CL_DEVICE_TYPE_ACCELERATOR ) printf(”\tTyp urządzenia: ACCELERATOR.\n”);
```

Obecnie[3] mamy tylko trzy typy urządzeń OpenCL, tj. tradycyjny procesor, kartę graficzną oraz układy FPGA, które są nazywane **akceleratorami obliczeń**. W przypadku komputerów domowych czy też urządzeń mobilnych, spotkamy się raczej tylko z procesorami i kartami graficznymi. Układy FPGA to na razie domena wyspecjalizowanych systemów obliczeniowych.

Innym przykładem, gdzie otrzymana informacja to np. liczba całkowita typu `cl_uint`, jest odczyt liczby dostępnych jednostek przetwarzających w urządzeniu obliczeniowym:

```
cli_err_num = clGetDeviceInfo(platform_devices[ocl_dev_idx], CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &tmp, NULL);

printf(”\tLiczba jednostek przetwarzających: %d\n”, tmp);
```

Jeżeli odczytujemy maksymalne wymiary siatki obliczeniowej, funkcja `clGetDeviceInfo` oczekuje, że zostanie podany adres do tablicy o przynajmniej trzech elementach:

```
cl_uint tmpTab[10] = {0,0,0,0,0,0,0,0,0,0};
```

Sam odczyt wymiarów jest podobny jak w poprzednich przypadkach:

```
cli_err_num = clGetDeviceInfo(platform_devices[ocl_dev_idx], CL_DEVICE_MAX_WORK_ITEM_SIZES, sizeof(size_t)*tmp, &tmpTab, NULL);
```

Interesujące nas informacje zostały umieszczone w trzech pierwszych elementach tablicy:

```
printf(”\tMaksymalne wielkości siatki obliczeniowej: %d / %d / %d \n”, tmpTab[0], tmpTab[1], tmpTab[2]);
```

Wiele pozostałych informacji odczytujemy w bardzo podobny sposób. Listing 4.1 przedstawia pełny kod źródłowy programu, który udostępnia podstawowe informacje o urządzeniach obliczeniowych OpenCL.

Nieco inaczej odczytuje się informacje o dostępnych formatach obrazu, które można przetwarzać za pomocą OpenCL. Informacje o obsługiwanych formatach obrazów 2D i 3D wymagają bowiem utworzenia kontekstu obliczeniowego. Kontekst ten jest tworzony dla urządzeń obliczeniowych:

```
ctx = clCreateContext(0, clui_num_devices, platform_devices, NULL, NULL, &cli_err_num);
```

Po utworzeniu kontekstu, analogicznie jak przy platformach bądź urządzeniach, należy odczytać liczbę dostępnych formatów, aby przydzielić odpowiednią ilość pamięci:

```
clGetSupportedImageFormats(ctx, CL_MEM_READ_ONLY, CL_MEM_OBJECT_IMAGE2D, NULL, NULL, &tmp);
```

Pamięć przydzielamy za pomocą funkcji `malloc`:

```
img_formats = (cl_image_format*)malloc(sizeof(cl_image_format) *tmp);
```

Pozostaje tylko odczytać informacje o dostępnych formatach dla obrazu 2D:

```
clGetSupportedImageFormats(ctx, CL_MEM_READ_ONLY, CL_MEM_OBJECT_IMAGE2D, tmp, img_formats, NULL);
```

Typ `cl_image_format` ma dwa pola, które nas interesują bezpośrednio – jedno to `image_channel_order`, które reprezentuje układ kanałów opisujących kolory piksela, oraz drugie, które określa typ kanału – `image_channel_data_type`. Oznacza to, że np. wartość koloru jest reprezentowana przez liczbę całkowitą, liczbę zmiennoprzecinkową etc. Aby przedstawić te

informacje, wystarczy prosta pętla `for`:

```
for(unsigned int i = 0; i < tmp; i++) {

    printf(”\t\t%i \t %-17s %-23s\n”, i+1,

imgFormatToChars(img_formats[i].image_channel_order),
imgFormatToChars(img_formats[i].image_channel_data_type));

}
```

Pola struktury `img_formats` to liczby, więc za pomocą funkcji `imgFormatToChars` kody poszczególnych formatów są zamieniane na odpowiednie reprezentacje znakowe. OpenCL oferuje także wsparcie dla obrazów 3D, odczytanie dostępnych formatów sprowadza się do identycznego kodu jak dla obrazów 2D, choć oczywiście należy zastosować inną stałą: `CL_MEM_OBJECT_IMAGE3D`.

Wspomniany już listing 4.1 zawiera obszerne fragmenty kodu źródłowego naszego pierwszego programu w OpenCL, w istocie nie zawiera tylko jednej funkcji o nazwie `imgFormatToChars`, której zadanie polega na zmianie kodu formatu obrazu na odpowiedni ciąg znaków. Funkcja ta została skomentowana w Dodatku C.

Listing 4.1. Obszerne fragmenty kodu źródłowego programu odczytującego informacje o dostępnych urządzeniach obliczeniowych zgodnych z technologią OpenCL

```
#include ”stdafx.h”

#include <stdio.h>

#include <stdlib.h>

#if defined (__APPLE__) || defined(MACOSX)

    #include <OpenCL/opencl.h>

#else

    #include <CL/opencl.h>

#endif

cl_uint ocl_plat_idx = 0;

cl_uint ocl_dev_idx = 0;

cl_uint clui_num_platforms = 0;

cl_uint clui_num_devices = 0;

cl_int cli_err_num;

cl_device_type device_type;

cl_platform_id *opencl_platforms = NULL;

cl_device_id *platform_devices = NULL;
```

```
cl_context ctx;

cl_bool bool_tmp;

cl_uint tmp = 0;

cl_ulong ulong_tmp = 0;

cl_uint tmpTab[10] = {0,0,0,0,0,0,0,0,0,0};

cl_device_local_mem_type dlmt_tmp;

cl_command_queue_properties cqpt_tmp;

cl_device_fp_config dfpc_tmp;

cl_image_format* img_formats;

size_t st_tmp;

const char* imgFormatToChars(cl_uint imgFormatCode)

{ } // opis funkcji w Dodatku C

int main(int argc, char* argv[])

{

    char buffer1[4096];

    cli_err_num = clGetPlatformIDs(0, NULL, &clui_num_platforms);

    if (cli_err_num != CL_SUCCESS)

    {

        printf(”Brak OpenCL!!!\n”);

        return 0;

    }

    printf(”Dostępna liczba platform OpenCL: (%d).\n”, clui_num_platforms);

    opencl_platforms = (cl_platform_id*)malloc(clui_num_platforms

                                                                    * sizeof(cl_platform_id));

    if (opencl_platforms == NULL)

    {

        printf(”Brak alokacji pamięci dla platform OpenCL!!!\n”);

        return 0;

    }
```

```
    cli_err_num = clGetPlatformIDs (clui_num_platforms, opencl_platforms, NULL);

    if (cli_err_num != CL_SUCCESS)

    {

        free((void*)opencl_platforms);

        opencl_platforms = NULL;

        return 0;

    }

    for(ocl_plat_idx=0; ocl_plat_idx < clui_num_platforms; ocl_plat_idx++)

    {

                    cli_err_num  =  clGetPlatformInfo(opencl_platforms  [ocl_plat_idx],
CL_PLATFORM_NAME, 4096, buffer1, NULL);

        if (cli_err_num == CL_SUCCESS)

        {

            printf(”Nazwa platformy: %s\n”, &buffer1[0]);

                    cli_err_num  =  clGetPlatformInfo  (opencl_platforms  [ocl_plat_idx],
CL_PLATFORM_VERSION, 4096, buffer1, NULL);

            printf(”\tWersja platformy: %s\n”, &buffer1[0]);

                    cli_err_num  =  clGetPlatformInfo  (opencl_platforms  [ocl_plat_idx],
CL_PLATFORM_VENDOR, 4096, buffer1, NULL);

            printf(”\tSprzedawca: %s\n”, &buffer1[0]);

                    cli_err_num  =  clGetPlatformInfo  (opencl_platforms  [ocl_plat_idx],
CL_PLATFORM_EXTENSIONS, 4096, buffer1, NULL);

            printf(”\tDostępne rozszerzenia:\n”);

            char *tmpstr;

            tmpstr = strtok(&buffer1[0], ” ”);

            while(tmpstr!=NULL)

            {

                printf(”\t\t %s\n”, tmpstr);

                tmpstr = strtok(NULL, ” ”);

            }

                    cli_err_num  =  clGetDeviceIDs  (  opencl_platforms[ocl_  plat_idx],
```

```
CL_DEVICE_TYPE_ALL, 0, NULL, &clui_num_devices);

            printf(”Ilość urządzeń obliczeniowych: (%d)\n”, clui_num_devices);

                    platform_devices  =  (cl_device_id*)malloc(sizeof  (cl_device_id)  *
clui_num_devices);

                     cli_err_num  =  clGetDeviceIDs  (opencl_platforms  [ocl_plat_idx],
CL_DEVICE_TYPE_ALL, clui_num_devices,

                                                        platform_devices, &clui_num_devices);

            for(ocl_dev_idx=0; ocl_dev_idx<clui_num_devices; ocl_dev_idx++)

            {

                        cli_err_num  =  clGetDeviceInfo(platform_devices  [ocl_dev_idx],
CL_DEVICE_NAME, 4096, buffer1, NULL);

                printf(”\tNazwa urządzenia: %s\n”, buffer1);

                        cli_err_num  =  clGetDeviceInfo(platform_devices  [ocl_dev_idx],
CL_DEVICE_VENDOR, 4096, buffer1, NULL);

                printf(”\tSprzedawca urządzenia: %s\n”, buffer1);

                        cli_err_num  =  clGetDeviceInfo(platform_devices  [ocl_dev_idx],
CL_DRIVER_VERSION, 4096, buffer1, NULL);

                printf(”\tWersja sterownika: %s\n”, buffer1);

                        cli_err_num  =  clGetDeviceInfo(platform_devices  [ocl_dev_idx],
CL_DEVICE_VERSION, 4096, buffer1, NULL);

                printf(”\tWersja urządzenia: %s\n”, buffer1);

                        cli_err_num  =  clGetDeviceInfo(platform_devices  [ocl_dev_idx],
CL_DEVICE_OPENCL_C_VERSION, 4096, buffer1, NULL);

                printf(”\tWersja OpenCL C: %s\n”, buffer1);

                        cli_err_num  =  clGetDeviceInfo(platform_devices  [ocl_dev_idx],
CL_DEVICE_TYPE, sizeof(device_type),

                                                                  &device_type, NULL);

                if ( device_type & CL_DEVICE_TYPE_CPU )

                    printf(”\tTyp urządzenia: CPU.\n”);

                if ( device_type & CL_DEVICE_TYPE_GPU )

                    printf(”\tTyp urządzenia: GPU.\n”);

                if ( device_type & CL_DEVICE_TYPE_ACCELERATOR )

                    printf(”\tTyp urządzenia: ACCELERATOR.\n”);
```

```
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_COMPUTE_UNITS,

                                                        sizeof(cl_uint), &tmp, NULL);

                printf("\tLiczba jednostek przetwarzających: %d\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,

                                                        sizeof(cl_uint), &tmp, NULL);

                printf("\tMaksymalny wymiar siatki obliczeniowej: %d\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_WORK_ITEM_SIZES, sizeof(size_t)*tmp, &tmpTab, NULL);

                printf("\tMaksymalne wielkości siatki obliczeniowej: %d / %d / %d \n",
tmpTab[0], tmpTab[1], tmpTab[2]);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_WORK_GROUP_SIZE,

                                                            sizeof(tmp), &tmp, NULL);

                printf("\tMaksymalna wielkość grupy roboczej: %d \n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_CLOCK_FREQUENCY,

                                                        sizeof(cl_uint), &tmp, NULL);

                printf("\tMaksymalna częstotliwość pracy urządzenia: %d Mhz\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_ADDRESS_BITS, sizeof(cl_uint),

                                                                    &tmp, NULL);

                printf("\tPrzestrzeń adresowa: %d\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_MEM_ALLOC_SIZE ,

                                                    sizeof(cl_ulong), &ulong_tmp, NULL);

                   printf("\tMaksymalny blok alokacji pamięci: %d MB \n", ulong_tmp /
1024 / 1024);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_GLOBAL_MEM_SIZE ,

                                                    sizeof(cl_ulong), &ulong_tmp, NULL);

                printf("\tDostępna pamięć: %d MB \n", ulong_tmp / 1024 / 1024);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
```

```
CL_DEVICE_ERROR_CORRECTION_SUPPORT ,
                                                    sizeof(cl_bool), &bool_tmp, NULL);
                printf(”\tBit korekcji pamięci: %s \n”, bool_tmp ? ”tak” : ”nie”);
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_LOCAL_MEM_TYPE,
                                                     sizeof(dlmt_tmp), &dlmt_tmp, NULL);
                      printf(”\tTyp pamięci lokalnej: %s \n”, dlmt_tmp ? ”lokalna” :
”globalna”);
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_LOCAL_MEM_SIZE ,
                                                   sizeof(cl_ulong), &ulong_tmp, NULL);
                printf(”\tWielkość pamięci lokalnej: %d KB \n”, ulong_tmp / 1024);
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE ,
                                                   sizeof(cl_ulong), &ulong_tmp, NULL);
                printf(”\tWielkość pamięci stałej: %d KB \n”, ulong_tmp / 1024);
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_QUEUE_PROPERTIES,
                                                     sizeof(cqpt_tmp), &cqpt_tmp, NULL);
                if( cqpt_tmp & CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE )
                                                printf(”\tWłasności kolejki urządzenia:
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE jest dostępne\n”);
                if( cqpt_tmp & CL_QUEUE_PROFILING_ENABLE )
                      printf(”\tWłasności kolejki urządzenia: CL_QUEUE_PROFILING_ENABLE
jest dostępne\n”);
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_IMAGE_SUPPORT, sizeof(bool_tmp),
                                                                  &bool_tmp, NULL);
                      printf(”\tWsparcie dla obrazu: CL_DEVICE_IMAGE_ SUPPORT: %s\n”,
bool_tmp ? ”tak” : ”nie”);
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_READ_IMAGE_ARGS,
                                                             sizeof(tmp), &tmp, NULL);
                printf(”\tMaksymalna liczba obrazów do odczytu: %d\n”, tmp);
```

```
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_MAX_WRITE_IMAGE_ARGS,
                                                                sizeof(tmp), &tmp, NULL);
                printf("\tMaksymalna liczba obrazów do zapisu: %d\n", tmp);
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_SINGLE_FP_CONFIG,
                                                        sizeof(dfpc_tmp), &dfpc_tmp, NULL);
                printf("\tWłasności typu single fp: %s%s%s%s%s%s\n",
                    dfpc_tmp & CL_FP_DENORM ? "denorms " : "",
                    dfpc_tmp & CL_FP_INF_NAN ? "INF-quietNaNs " : "",
                    dfpc_tmp & CL_FP_ROUND_TO_NEAREST ? "round-to-nearest " : "",
                    dfpc_tmp & CL_FP_ROUND_TO_ZERO ? "round-to-zero " : "",
                    dfpc_tmp & CL_FP_ROUND_TO_INF ? "round-to-inf " : "",
                    dfpc_tmp & CL_FP_FMA ? "fma" : "");
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_DOUBLE_FP_CONFIG,
                                                        sizeof(dfpc_tmp), &dfpc_tmp, NULL);
                printf("\tWłasności typu double fp: %s%s%s%s%s%s\n",
                    dfpc_tmp & CL_FP_DENORM ? "denorms " : "",
                    dfpc_tmp & CL_FP_INF_NAN ? "INF-quietNaNs " : "",
                    dfpc_tmp & CL_FP_ROUND_TO_NEAREST ? "round-to-nearest " : "",
                    dfpc_tmp & CL_FP_ROUND_TO_ZERO ? "round-to-zero " : "",
                    dfpc_tmp & CL_FP_ROUND_TO_INF ? "round-to-inf " : "",
                    dfpc_tmp & CL_FP_FMA ? "FMA" : "");
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_IMAGE2D_MAX_WIDTH,
                                                            sizeof(st_tmp), &st_tmp, NULL);
                printf("\tMaksymalna szerokość obrazu 2D: %d\n", st_tmp);
                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_IMAGE2D_MAX_HEIGHT,
                                                            sizeof(st_tmp), &st_tmp, NULL);
```

```
                printf("\tMaksymalna wysokość obrazu 2D: %d\n", st_tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_IMAGE3D_MAX_WIDTH,

                                                        sizeof(st_tmp), &st_tmp, NULL);

                printf("\tMaksymalna szerokość obrazu 3D: %d\n", st_tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_IMAGE3D_MAX_HEIGHT,

                                                        sizeof(st_tmp), &st_tmp, NULL);

                printf("\tMaksymalna wysokość obrazu 3D: %d\n", st_tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_IMAGE3D_MAX_DEPTH,

                                                        sizeof(st_tmp), &st_tmp, NULL);

                printf("\tMaksymalna głębokość obrazu 3D: %d\n", st_tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR, sizeof(tmp), &tmp, NULL);

                printf("\tPreferowana wielkość wektora dla typu CHAR: %d\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT, sizeof(tmp), &tmp, NULL);

                printf("\tPreferowana wielkość wektora dla typu SHORT: %d\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT, sizeof(tmp), &tmp, NULL);

                printf("\tPreferowana wielkość wektora dla typu INT: %d\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG, sizeof(tmp), &tmp, NULL);

                printf("\tPreferowana wielkość wektora dla typu LONG: %d\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT, sizeof(tmp), &tmp, NULL);

                printf("\tPreferowana wielkość wektora dla typu FLOAT: %d\n", tmp);

                        cli_err_num = clGetDeviceInfo(platform_devices [ocl_dev_idx],
CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE, sizeof(tmp), &tmp, NULL);

                printf("\tPreferowana wielkość wektora dla typu DOUBLE: %d\n", tmp);

                    ctx = clCreateContext(0, clui_num_devices, platform_devices, NULL,
NULL, &cli_err_num);

                                        clGetSupportedImageFormats(ctx,  CL_MEM_READ_ONLY,
```

```
CL_MEM_OBJECT_IMAGE2D, NULL, NULL, &tmp);

                img_formats = (cl_image_format*)malloc(sizeof(cl_image_format)*tmp);

                                        clGetSupportedImageFormats(ctx,   CL_MEM_READ_ONLY,
CL_MEM_OBJECT_IMAGE2D, tmp, img_formats, NULL);

                printf(”\tWsparcie dla obrazu 2D\n”);

                printf(”\t\t#      Rodzaj składowej Typ kanału\n”);

                for(unsigned int i = 0; i < tmp; i++)

                {

                                                printf(”\t\t%i   \t   %-17s   %-23s\n”,   i+1,
imgFormatToChars(img_formats[i].image_channel_order),
imgFormatToChars(img_formats[i].image_channel_data_type));

                }

                free( (void*)img_formats );

                                        clGetSupportedImageFormats(ctx,   CL_MEM_READ_ONLY,
CL_MEM_OBJECT_IMAGE3D, NULL, NULL, &tmp);

                img_formats = (cl_image_format*)malloc (sizeof(cl_image_format)*tmp);

                                        clGetSupportedImageFormats(ctx,   CL_MEM_READ_ONLY,
CL_MEM_OBJECT_IMAGE3D, tmp, img_formats, NULL);

                printf(”\tWsparcie dla obrazu 3D\n”);

                printf(”\t\t# Rodzaj składowej Typ kanału\n”);

                for(unsigned int i = 0; i < tmp; i++)

                {

                                                printf(”\t\t%i   \t   %-17s   %-23s\n”,   i+1,
imgFormatToChars(img_formats[i].image_channel_order),
imgFormatToChars(img_formats[i].image_channel_data_type));

                }

                free( (void*)img_formats );

            }

            free((void*)platform_devices);

        }

    }

    free((void*)opencl_platforms);
```

```
    return 0;

}
```

4.1.2. Informacje o urządzeniu obliczeniowym obiektowo i w C++

Wykorzystanie języka C++ w technologii OpenCL sprowadza się nie tylko i wyłącznie do korzystania z funkcji dostępnych dla języka C. Autorzy standardu przygotowali cały zestaw klas oraz dodatkowych funkcji, które w niektórych aspektach upraszczają współpracę z OpenCL.

Raz jeszcze zostanie przedstawiony program do odczytu informacji o dostępnych urządzeniach zgodnych z technologią OpenCL, aby pokazać ułatwienia, które są oferowane przez API w ramach języka C++.

Rozpoczynamy od włączenia pliku nagłówkowego *cl.hpp* (w zależności od stosowanego systemu operacyjnego – Mac OS, Windows czy Linux – plik nagłówkowy znajduje się w innym katalogu):

```
#define __NO_STD_VECTOR

#define __MAX_DEFAULT_VECTOR_SIZE 128

#if defined (__APPLE__) || defined(MACOSX)

    #include <OpenCL/cl.hpp>

#else

    #include <CL/cl.hpp>

#endif
```

Bardzo istotna jest definicja `__NO_STD_VECTOR`, oznaczająca że nie będzie stosowany typ `vector` dostępny w ramach standardowej biblioteki wzorców STL (ang. *Standard Template Library*). Użycie typu `vector` z pakietu STL może niestety powodować problemy dotyczące zgodności typów podczas kompilacji. Definicje typów zawartych w *cl.hpp* stosują własny typ `cl::vector`. Dostępny jest też typ `cl::string`, ale do podstawowych zadań na ciągach znaków lepiej stosować `std::string`. Implementacja typu `cl::string` oferowana przez OpenCL jest bardzo ograniczona, np. nie jest możliwa bezpośrednia współpraca z obiektami `std::cout` czy `std::cin`.

Istotna jest też definicja stałej dotyczącej maksymalnej długości wektora:

```
#define __MAX_DEFAULT_VECTOR_SIZE 128
```

Szablon `vector` ma dość prostą implementację i domyślnie zawiera tylko 10 elementów. Wystarczy to do odczytu informacji o platformie czy np. listy urządzeń, ale rozmiar wektora jest stanowczo zbyt mały, aby przechować listę wspieranych formatów dla obrazów. Dlatego należy zwiększyć wielkość wektora. Wartość 128 zapewnia, że odbierzemy w komplecie listę formatów, która zazwyczaj liczy około 80 pozycji. Można też zwiększyć wielkość wektora np. do 512 elementów.

Odczytanie informacji o możliwościach danego urządzenia obliczeniowego wymaga zdefiniowania listy, na której zostaną umieszczone informacje o dostępnych platformach oraz drugiej listy

dostępnych urządzeń obliczeniowych. Odpowiednia definicja, dzięki zastosowaniu typu `vector`, jest bardzo prosta:

```
cl::vector< cl::Platform > platforms;

cl::vector< cl::Device > devices;
```

Jak widać zwalnia nas to z konieczności zarządzania pamięcią, a dodatkowo sam proces odczytania dostępnych platform to tylko jedna linia kodu:

```
cl::Platform::get( &platforms );
```

Sprawdzenie liczby dostępnych platform jest również uproszczone, bowiem sprowadza się do odczytania liczby elementów umieszczonych na liście `platforms`:

```
std::cout << ”Dostępna liczba platform OpenCL: ” << platforms.size();
```

Ponieważ zmienna `platforms` jest wektorem, to mamy także dostęp do operatora indeksowania, co pozwala na bardzo łatwe odczytywanie informacji o wybranej platformie. Aby odczytać nazwę pierwszej platformy, wystarczy zdefiniować pomocniczą zmienną typu `string`:

```
std::string tmpstr;
```

Następnie możemy odczytać nazwę platformy przy wykorzystaniu metody `getInfo` oraz podanej w poprzednim przykładzie stałej `CL_PLATFORM_NAME`, po czym za pomocą `cout` wyświetlamy odczytaną nazwę:

```
platforms[0].getInfo((cl_platform_info)CL_PLATFORM_NAME, &tmpstr);

std::cout << ”Nazwa platformy: ” << tmpstr << ”\n”;
```

Postać metody `getInfo` jest znacznie wygodniejsza w użyciu niż funkcja `clGetPlatformInfo` dla języka C. Na przykład odczytanie ciągu znaków reprezentujących listę z dostępnymi rozszerzeniami sprowadza się do dwóch linii kodu:

```
p.getInfo((cl_platform_info)CL_PLATFORM_EXTENSIONS, &tmpstr);

std::cout << ”\tRozszerzenia: ” << tmpstr << ”\n”;
```

Jak widać, są to wyrażenia identyczne jak w przypadku odczytania nazwy platformy (choć nie są rozdzielone na poszczególne nazwy).

Odczytanie urządzeń obliczeniowych w ramach używanej platformy, jak można odgadnąć, także zostało sprowadzone do jednej metody o nazwie `getDevices`. Identycznie jak w przypadku `platform` deklarujemy listę, na której zostaną umieszczone odniesienia do poszczególnych urządzeń:

```
cl::vector< cl::Device > devices;
```

Odczytanie listy urządzeń obliczeniowych zgodnych z technologią OpenCL dowolnego typu, tj. opartych np. na tradycyjnym procesorze, a nie tylko na karcie graficznej, uzyskamy za pomocą jednej linii kodu:

```
p.getDevices(CL_DEVICE_TYPE_ALL, &devices);
```

Zmienna `p` to wybrana wcześniej platforma OpenCL. Odczytanie informacji o urządzeniu jest analogiczne, jak dla platformy, np. uzyskanie nazwy urządzenia sprowadza się do następującej linii kodu:

```
d.getInfo( CL_DEVICE_NAME, &tmpstr);
```

gdzie `d` reprezentuje wybrane przez nas urządzenie. Wykorzystując możliwości nowego standardu C++11, deklarację zmiennej `d` można zrealizować w następujący sposób:

```
auto d = devices[0];
```

Metoda `getInfo` obsługuje także inne typy danych niż tylko ciąg znaków. Uzyskanie informacji na temat typu urządzenia wymaga wcześniejszej deklaracji zmiennej pomocniczej:

```
cl_device_type device_type;
```

Odczytanie informacji sprowadza się do ponownego użycia metody `getInfo`:

```
d.getInfo( CL_DEVICE_TYPE, &device_type);
```

Sprawdzenie, czy dane urządzenie jest oparte na GPU, podobnie jak w odmianie dla języka C, wymaga tylko operacji bitowej „`and`”:

```
if ( device_type & CL_DEVICE_TYPE_GPU )

    std::cout << ”\tTyp urządzenia: GPU.” << ”\n”;
```

Uzyskanie pozostałych informacji praktycznie zawsze sprowadza się do użycia metody `getInfo`, czego przykładem jest odczytanie liczby jednostek przetwarzających:

```
d.getInfo(CL_DEVICE_MAX_COMPUTE_UNITS, &tmp);

std::cout << ”\tLiczba jednostek przetwarzających:” << tmp << ”\n”;
```

Należy zwrócić uwagę na typ informacji, jaki próbuje się uzyskać. W powyższym przykładzie jest to liczba całkowita bez znaku `cl_uint`, wcześniej był to ciąg znaków. Może to także być wartość logiczna, jeśli konieczne jest sprawdzenie, czy dostępna jest obsługa przetwarzania obrazów:

```
d.getInfo(CL_DEVICE_IMAGE_SUPPORT, &bool_tmp);
```

Tym razem otrzymana informacja trafia do zmiennej `bool_tmp` typu `cl_bool`. Inny typ jest stosowany w przypadku odczytywania informacji o dostępnych możliwościach dla liczb zmiennoprzecinkowych. Należy zadeklarować odpowiednią zmienną:

```
cl_device_fp_config dfpc_tmp;
```

Następnie należy uzyskać informacje, np. dla liczb zmiennoprzecinkowych o pojedynczej precyzji:

```
d.getInfo(CL_DEVICE_SINGLE_FP_CONFIG, &dfpc_tmp);
```

Teraz możemy wyświetlać informacje o dostępnych własnościach, stosując odpowiednie flagi oraz operację bitowego iloczynu:

```
std::cout << ”\tWłasności typu single fp: ” <<

    ((dfpc_tmp & CL_FP_DENORM) ? ”denorms ” : ””) <<

    // pozostałe fragmenty na Listingu 4.1.2

    ((dfpc_tmp & CL_FP_FMA) ? ”FMA” : ””) << std::endl;
```

Inaczej odczytuje się informacje o dostępnych obrazach. Podobnie jak w wersji dla języka C, należy utworzyć kontekst obliczeniowy. Podczas tworzenia obiektu własności:

```
cl_context_properties properties[] = { CL_CONTEXT_PLATFORM, (cl_context_properties)
(platforms[0])(), 0};
```

można utworzyć kontekst oraz podać typ urządzenia, jaki nas interesuje. Nie ma potrzeby podawania konkretnego obiektu, dla którego chcemy utworzyć kontekst:

```
cl::Context context(CL_DEVICE_TYPE_GPU, properties);
```

Odczytanie listy dostępnych formatów nie jest trudne. Dla porządku kasujemy zawartość wektora `image_formats`:

```
image_formats.clear();
```

Następnie wywołujemy metodę `getSupportedImageFormats`:

```
context.getSupportedImageFormats( CL_MEM_READ_ONLY, CL_MEM_OBJECT_IMAGE2D,
&image_formats);
```

Stała `CL_MEM_OBJECT_IMAGE2D` oznacza, że będziemy odczytywać formaty dla obrazów dwuwymiarowych. Wyświetlenie dostępnych informacji można zrealizować za pomocą pętli `for`, ponownie wykorzystując możliwość C++11:

```
int i = 1;

for( auto iformat : image_formats )

{

            std::cout << ”\t\t” << i << std::setw(17) << imgFormatToChars(
iformat.image_channel_order)
<< std::setw(23) << imgFormatToChars(iformat.image_channel_data_type) << std::endl;

    i++;

    }

}
```

Listing 4.2 przedstawia kod źródłowy programu odczytującego informacje o urządzeniach obliczeniowych. Chociaż jest to wersja skrócona, która wyświetla zaledwie kilka informacji, jednak

jest kompletna i po uzupełnieniu funkcji `imgFormatToChars` stanowi pełny program, który można skompilować i uruchomić.

Listing 4.2. Odczytanie wybranych informacji o urządzeniu za pomocą API OpenCL w odmianie dla języka C++

```
#include <stdio.h>

#include <stdlib.h>

#include <string>

#include <iostream>

#include <iomanip>

#define __NO_STD_VECTOR

#define __MAX_DEFAULT_VECTOR_SIZE 128

#if defined (__APPLE__) || defined(MACOSX)

    #include <OpenCL/cl.hpp>

#else

    #include <CL/cl.hpp>

#endif

std::string tmpstr;

cl_uint tmp;

cl_bool bool_tmp;

cl_device_fp_config dfpc_tmp;

cl_uint ocl_plat_idx = 0;

cl_uint ocl_dev_idx = 0;

cl_uint clui_num_platforms = 0;

cl_uint clui_num_devices = 0;

cl_int cli_err_num;

cl_device_type device_type;

cl::vector< cl::Platform > platforms;

cl::vector< cl::Device > devices;

cl::vector< cl::ImageFormat > image_formats;

const char* imgFormatToChars(cl_uint imgFormatCode)
```

```
{ } // opis funkcji w Dodatku C

int main(int argc, char* argv[])

{

    cl::Platform::get( &platforms );

     std::cout << ”Dostępna liczba platform OpenCL: ” << platforms.size() << std::endl
<< std::endl;

    for (auto p : platforms)

    {

        p.getInfo((cl_platform_info)CL_PLATFORM_NAME, &tmpstr);

        std::cout << ”Nazwa platformy: ” << tmpstr << ”\n”;

        p.getInfo((cl_platform_info)CL_PLATFORM_VENDOR, &tmpstr);

        std::cout << ”\tSprzedawca: ” << tmpstr << ”\n”;

        p.getInfo((cl_platform_info)CL_PLATFORM_EXTENSIONS, &tmpstr);

        std::cout << ”\tRozszerzenia: ” << tmpstr << ”\n”;

        p.getDevices(CL_DEVICE_TYPE_ALL, &devices);

        std::cout << ”Ilość urządzeń obliczeniowych: ” << devices.size() << ”\n”;

        for( auto d : devices )

        {

            d.getInfo( CL_DEVICE_NAME, &tmpstr);

            std::cout << ”\tNazwa urządzenia: ” << tmpstr << ”\n”;

            d.getInfo( CL_DEVICE_VENDOR, &tmpstr);

            std::cout << ”\tSprzedawca urządzenia: ” << tmpstr << ”\n”;

            d.getInfo( CL_DEVICE_TYPE, &device_type);

            if ( device_type & CL_DEVICE_TYPE_CPU )

                std::cout << ”\tTyp urządzenia: CPU.” << ”\n”;

            if ( device_type & CL_DEVICE_TYPE_GPU )

                std::cout << ”\tTyp urządzenia: GPU.” << ”\n”;

            if ( device_type & CL_DEVICE_TYPE_ACCELERATOR )

                std::cout << ”\tTyp urządzenia: ACCELERATOR.” << ”\n”;
```

```
            d.getInfo(CL_DEVICE_MAX_COMPUTE_UNITS, &tmp);

            std::cout << ”\tLiczba jednostek przetwarzających: ” << tmp << ”\n”;

            d.getInfo(CL_DEVICE_IMAGE_SUPPORT, &bool_tmp);

               std::cout << ”\tWsparcie dla obrazu: CL_DEVICE_ IMAGE_SUPPORT: ” << (
(bool_tmp == CL_TRUE) ?

                                std::string(”tak”) : std::string(”nie”) ) << ”\n”;

            d.getInfo(CL_DEVICE_SINGLE_FP_CONFIG, &dfpc_tmp);

            std::cout << ”\tWłasności typu single fp: ” <<

                ((dfpc_tmp & CL_FP_DENORM) ? ”denorms ” : ””) <<

                ((dfpc_tmp & CL_FP_INF_NAN) ? ”INF-quietNaNs ” : ””) <<

                ((dfpc_tmp & CL_FP_ROUND_TO_NEAREST) ? ”round-to-nearest ” : ””) <<

                ((dfpc_tmp & CL_FP_ROUND_TO_ZERO) ? ”round-to-zero ” : ””) <<

                ((dfpc_tmp & CL_FP_ROUND_TO_INF) ? ”round-to-inf ” : ””) <<

                ((dfpc_tmp & CL_FP_FMA) ? ”fma” : ””) << std::endl;

                      cl_context_properties  properties[]  =  {  CL_CONTEXT_  PLATFORM,
(cl_context_properties)(platforms[0])(), 0};

            cl::Context context(CL_DEVICE_TYPE_GPU, properties);

            image_formats.clear();

             context.getSupportedImageFormats( CL_MEM_READ_ONLY, CL_MEM_OBJECT_IMAGE2D,
&image_formats);

            int i = 1;

            for( auto iformat : image_formats )

            {

                      std::cout  <<  ”\t\t”  <<  i  <<  std::setw(17)  <<  imgFormatToChars(
iformat.image_channel_order) <<

                                        std::setw(23) << imgFormatToChars(iformat.image_

                                                          channel_data_type) << std::endl;

                i++;

            }

        }

        std::cout << std::endl;
```

```
    }

    getchar();

    return 0;

}
```

4.2. Witaj Świecie!

W tym przykładzie pokażemy użycie jądra obliczeniowego do wykonania obliczeń polegających na wykonaniu operacji dodawania na każdym elemencie wektora, jaki zostanie podany na wejście procedury obliczeniowej. Elementami wektora będą kody ASCII. Zostały one tak dobrane, że po dodaniu np. wartości 5 do każdego elementu otrzymamy napis `Hello World`! Takie kodowanie jest prostym odpowiednikiem szyfru Cezara, w którym poszczególne litery są przesuwane o pewną pozycję w prawo bądź w lewo.

4.2.1. Przygotowania

Ciąg znaków do obliczeń w naszym przykładzie może zostać zadeklarowany w następujący sposób:

```
char host_in_txt[]={'H'+5, 'e'+5, 'l'+5, 'l'+5, 'o'+5,' '+5, 'W'+5, 'o'+5, 'r'+5, 'l'+5, 'd'+5, '!'+5, 0};
```

Do każdej litery dodaliśmy wartość pięć, oczywiście możemy samodzielnie policzyć, jakie litery otrzyma się po przesunięciu o pięć pozycji w kodzie ASCII, i otrzymamy następującą deklarację:

```
char host_in_txt[] = {'M', 'j', 'q', 'q', 't', '%', '\\', 't', 'w', 'q', 'i', '&', 0};
```

Potrzebny jest też ciąg wyjściowy, po zakończeniu obliczeń przepiszemy rozkodowany ciąg znaków do podanej zmiennej:

```
char host_out_txt[]={0,0,0,0,0,0,0,0,0,0,0,0,0};
```

Kolejnym elementem, jaki możemy od razu pokazać, jest też jądro obliczeniowe, które zostało podane w listingu 4.3. Parametry, które przekazujemy, odzwierciedlają nasze zadanie. Parametr `inTxt` określa wskaźnik na dane wejściowe. Dodatkowe słowo `__global` oznacza, że dane będziemy czerpać z pamięci globalnej urządzenia obliczeniowego. Parametr `outTxt` przekazuje wskaźnik na dane wyjściowe. Ostatni parametr `N` oznacza długość napisu, jaki będziemy odkodowywać.

W treści funkcji wykonujemy dwie główne czynności: odczytanie identyfikatora wątku oraz przepisanie znaku z tablicy wejściowej pomniejszonej o wartość pięć. Na początku odczytujemy identyfikator wątku:

```
int idx = get_global_id(0);
```

Nie posługujemy się pętlą, np. typu `for`, aby odczytać elementy ze zmiennej `inTxt`, ale wykorzystujemy fakt, iż mamy wiele niezależnych wątków, a każdy z nich będzie odpowiedzialny za operację na jednym znaku. Inaczej mówiąc, dany wątek będzie się zajmować tylko jedną literą naszego napisu niezależnie od pozostałych wątków.

Przepisanie znaku z tablicy wejściowej pomniejszonej o wartość pięć jest wykonywane pod warunkiem, że numer wątku `idx` jest mniejszy niż całkowita liczba znaków reprezentowana przez parametr `N`. W ten sposób zabezpieczamy się przed przekroczeniem obszaru pamięci, jaki zostanie przydzielony w pamięci globalnej urządzenia obliczeniowego.

Listing 4.3. Jądro obliczeniowe odejmujące wartość 5 od każdego znaku w ciągu znaków podanym przez parametr `inTxt`

```
__kernel void helloDecode(__global char *inTxt, __global char *outTxt, int N)

{

    int idx = get_global_id(0);

    if(idx < N)

    {

        outTxt[idx] = inTxt[idx] – 5;

    }

}
```

4.2.2. Uruchomienie jądra obliczeniowego

Na początek wykonujemy czynności związane z odczytaniem informacji o platformie oraz o urządzeniu obliczeniowym. Kod związany z tymi czynnościami jest bardzo podobny do kodu podanego w rozdz. 4.1.1. Poszczególne czynności można przedstawić za pomocą prostego diagramu (rys. 4.2). Pominiemy tu opis tworzenia elementów dotyczących platformy oraz urządzeń i od razu przejdziemy do utworzenia kontekstu obliczeń:

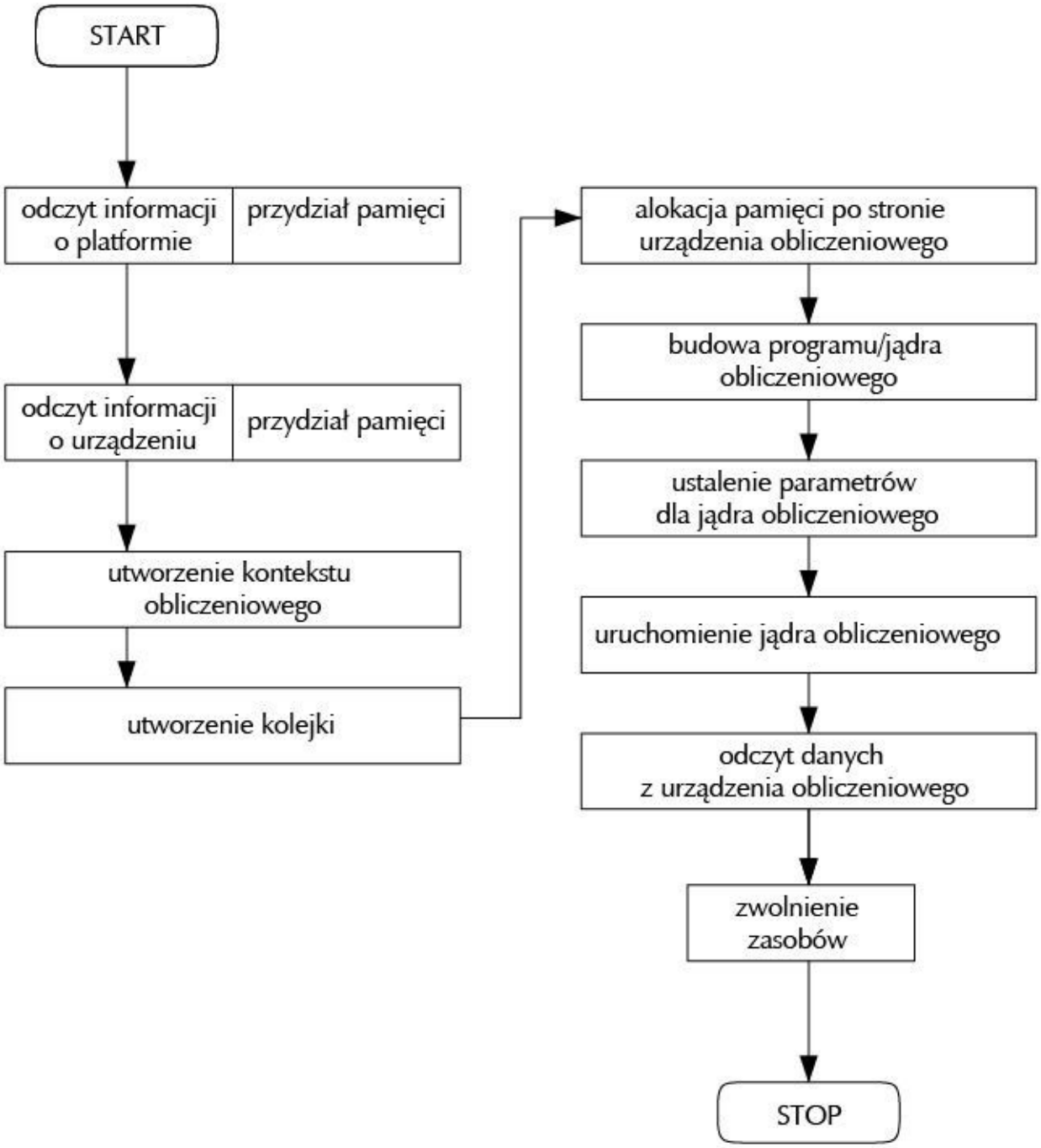
```
cl_compute_context = clCreateContext( NULL, clui_num_devices, &platform_devices[0], NULL, NULL, &cli_err_num);
```

Wskazano platformę, dla której utworzono kontekst, warto też sprawdzić, czy udało się utworzyć poprawnie kontekst:

```
if (!cl_compute_context)

{

    printf(”Błąd: Failed to create a compute context!\n”);

    return EXIT_FAILURE;

}
```

Kod błędu zawarty jest w zmiennej `cli_err_num`. Kolejne zadanie to utworzenie kolejki, które wymaga wskazania kontekstu oraz ponownie platformy obliczeniowej:

```
cl_compute_command_queue = clCreateCommandQueue( cl_compute_context, platform_devices[0], 0, &cli_err_num);
```



Rys. 4.2. Ogólny schemat czynności wykonywanych w OpenCL podczas uruchamiania jądra obliczeniowego

Analogicznie jak poprzednio, również warto sprawdzić, czy utworzenie kolejki się powiodło. Możemy teraz przystąpić do utworzenia buforów umieszczonych w pamięci globalnej urządzenia, do których przeniesiemy zawartość zmiennych znajdujących się w pamięci hosta. Dwie zmienne deklarujemy w następujący sposób:

```
cl_mem device_in_txt;

cl_mem device_out_txt;
```

Utworzenie odpowiednich buforów to już tylko pojedyncze linie kodu. W pierwszej kolejności utworzymy bufor, z którego będziemy odczytywać dane:

```
device_in_txt = clCreateBuffer( cl_compute_context, CL_MEM_READ_ONLY, dataBytes, NULL,
&cli_err_num);
```

Znaczenie poszczególnych parametrów jest następujące: na początek podajemy obiekt reprezentujący kontekst obliczeń, w drugim parametrze określamy, czy dane będą przeznaczone tylko do odczytu. Kolejny parametr to ilość danych w bajtach. Parametr z wartością `NULL` może być wskaźnikiem do danych po stronie hosta. W ostatnim parametrze podajemy adres zmiennej, gdzie zostanie umieszczony numer ewentualnego błędu lub po prostu wartość 0, jeśli operacja zostanie wykonana poprawnie.

Utworzenie bufora, do którego będziemy zapisywać rozkodowany napis, sprowadza się do wywołania funkcji `clCreateBuffer`:

```
device_out_txt = clCreateBuffer( cl_compute_context, CL_MEM_WRITE_ONLY, dataBytes, NULL, &cli_err_num);
```

Różnicę stanowi wartość drugiego parametru `CL_MEM_WRITE_ONLY` oznaczająca, że tym razem pamięć będzie przeznaczona tylko do zapisu danych.

Skopiowanie danych do zmiennej wejściowej zrealizujemy za pomocą funkcji `clEnqueueWriteBuffer`. Jej wywołanie w naszym przypadku przedstawia się następująco:

```
cli_err_num = clEnqueueWriteBuffer( cl_compute_command_queue, device_in_txt, CL_TRUE, 0, dataBytes, &host_in_txt[0], 0, NULL, NULL);
```

Pierwszy parametr to utworzona wcześniej kolejka poleceń. Następnie podajemy wskaźniki do danych, które będziemy przenosić. W pierwszej kolejności podajemy wskaźnik reprezentujący obszar pamięci w urządzeniu obliczeniowym (wskaźnik `device_int_txt`). Wartość `CL_TRUE` oznacza, że proces kopiowania będzie blokowany, inaczej mówiąc, nie przejdziemy do wykonywania dalszych czynności dopóki kopiowanie danych nie zostanie zakończone. Ponieważ kopiujemy bardzo małą ilość danych, więc możemy poczekać na zakończenie tego procesu.

Parametr czwarty oznacza przesunięcie danych (ang. *offset*). Ponieważ chcemy skopiować cały obszar danych, więc podajemy zero. Ilość danych to naturalnie wielkość `dataBytes`, analogicznie jak poprzednio jest to ilość bajtów. Kolejny parametr `host_in_txt` to wskaźnik po stronie hosta/gospodarza, z którego czerpiemy dane przesyłane do pamięci urządzenia obliczeniowego. Ostatnie trzy parametry są wykorzystywane do obsługi zdarzeń, mają znaczenie, jeśli proces kopiowania danych jest procesem nieblokującym. My jednak stosujemy prostsze rozwiązanie z zapisem blokowanym, więc na razie pominiemy znaczenie tych parametrów.

Po przeniesieniu danych możemy przystąpić do kompilacji jądra obliczeniowego. Zakładamy, że tekst źródłowy jądra obliczeniowego został umieszczony w zmiennej `kernelSource`. Pierwsze zadanie to utworzenie tzw. obiektu programu (nie jest to jeszcze proces kompilacji):

```
cl_kernel_program = clCreateProgramWithSource( cl_compute_context, 1, (const char**)&kernelSource, NULL, &cli_err_num);
```

Pierwszy parametr to kontekst obliczeniowy. Wartość 1 w drugim parametrze oznacza, że dostarczamy tylko jeden kod źródłowy. W następnym, trzecim parametrze, mamy podwójny wskaźnik, co oznacza tablicę ciągów znaków do tekstu lub tekstów źródłowych (inaczej mówiąc, mamy wskaźnik na tablicę wskaźników). Pozwala on dostarczyć kod źródłowy podzielony na kilka części (najczęściej obiekt tworzy się tylko z jednego fragmentu kodu źródłowego). Związanym z tym

jest też kolejny parametr – tablica, w której powinnyśmy podać długości w znakach kolejnych elementów tablicy `kernelSource`. Podanie wartości `NULL` oznacza, że to wartość `NULL` będzie sygnalizować koniec kodu źródłowego, a nie ilość w znakach. Piąty parametr to (jak w wielu poprzednich funkcjach API systemu OpenCL) zmienna z kodem o powodzeniu lub niepowodzeniu operacji tworzenia obiektu programu.

Następne zadanie to właściwa kompilacja programu, czy też budowanie programu binarnego za pomocą funkcji `clBuildProgram`. Parametry, które przekazujemy, mają następujące znaczenie: pierwszy to obiekt programu, następnie podajemy liczbę urządzeń i wcześniej odczytaną tablicę urządzeń. Nie zakładamy, że mamy tylko jedno urządzenie, bowiem program zostanie zbudowany dla wszystkich dostępnych urządzeń, jakie wcześniej zostały wybrane. Dla uproszczenia jednak zakładamy, że będziemy wybierać tylko jedno urządzenie. W ostatnich trzech parametrach można podać dodatkowe opcje oraz podłączyć dodatkową funkcję, która zostanie wywołana po zakończeniu procesu budowy programu obliczeniowego przez funkcję `clBuildProgram`. Nie korzystamy z tych możliwości, więc ostatecznie wywołanie naszej procedury budującej jest następujące:

```
cli_err_num = clBuildProgram( cl_kernel_program, clui_num_devices, platform_devices,
NULL, NULL, NULL);
```

Ewentualne błędy możemy odczytać ze zmiennej `cli_err_num` (więcej informacji o odczycie błędów związanych z budową programu podano w rozdz. 4.6). Zakładając, że kompilacja zakończyła się sukcesem, można utworzyć obiekt reprezentujący jądro obliczeniowe:

```
kernel = clCreateKernel(cl_kernel_program, ”helloDecode”, &cli_err_num );
```

Tym razem niezwykle istotny jest drugi parametr, bowiem podajemy nazwę procedury, którą będziemy wywoływać za pomocą `clEnqueueNDRangeKernel`. Nazwa ta musi być taka sama jak główna funkcja w kodzie źródłowym, opatrzona atrybutem `__kernel`. Jako wynik otrzymamy obiekt jądra obliczeniowego, które jest prawie gotowe do uruchomienia, ale wymaga jeszcze określenia parametrów.

Ustalenie parametrów jądra wykonujemy za pomocą funkcji `clSetKernelArg`, niestety każdy parametr musimy ustalić oddzielnie. Dla funkcji `helloDecode`, dwa pierwsze parametry to wskaźniki do pamięci globalnej, a trzeci parametr to wielkość skalarna.

Pierwszy parametr to dane, które będziemy odczytywać, dlatego podamy adres wskaźnika `device_in_txt`, który reprezentuje dane do odczytania:

```
cli_err_num = clSetKernelArg( kernel, 0, sizeof(cl_mem), &device_in_txt );
```

Analogicznie przy drugim parametrze musimy podać adres wskaźnika reprezentującego obszar pamięci, gdzie będą umieszczane dane po przetworzeniu przez jądro obliczeniowe:

```
cli_err_num |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &device_out_txt );
```

Trzeci parametr to liczba elementów, na których będziemy wykonywać operacje, choć jest to liczba typu `int`, to sposób określenia trzeciego parametru jest identyczny jak poprzednio:

```
cli_err_num |= clSetKernelArg( kernel, 2, sizeof(int), &N );
```

Wspólnym elementem w każdym przypadku jest oczywiście pierwszy argument, tj. obiekt reprezentujący jądro obliczeniowe, drugi argument reprezentuje numer parametru. W trzecim argumencie podajemy wielkość danych, przy czym nie jest to rzeczywista wielkość danej struktury danych, ale tylko wskaźnika lub skalara, jak np. w przypadku wartości `N`. Na koniec w czwartym argumencie podajemy adres wskaźnika lub wartości skalarnej.

Po tych przygotowaniach możemy przystąpić do uruchomienia jądra obliczeniowego, jednak musimy określić postać siatki obliczeniowej. Ponieważ nasza procedura działa na ciągu znaków, czyli wektorze, to zastosujemy tylko globalną siatkę obliczeniową o rozmiarze `N`.

```
globalWorkSize = N;
```

Uruchomienie jądra zrealizujemy za pomocą funkcji `clEnqueueNDRangeKernel`. W pierwszych dwóch argumentach podajemy obiekty kolejki oraz jądra. Ważną rolę pełni trzeci parametr, bowiem w nim określamy wymiar siatki obliczeniowej. W naszym przypadku przyjmuje ona kształt wektora, więc jest strukturą jednowymiarową. Kolejne trzy parametry to odpowiednio offset, wielkość siatki globalnej oraz wielkość siatki lokalnej. W naszym przypadku określamy tylko wielkość globalną. Ostatnie trzy parametry pozwalają wskazać funkcję do obsługi zdarzeń, np. do sygnalizacji zakończenia obliczeń przez jądro obliczeniowe. Podajemy wartości 0 oraz `NULL`, co oznacza, że poczekamy aż obliczenia się zakończą.

Powyższy opis zamieniamy na następujący kod:

```
cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 1, NULL, &globalWorkSize, NULL, 0, NULL, NULL );
```

Jądro obliczeniowe wykonało swoje zadanie, ale nie mamy jeszcze bezpośredniego dostępu do danych znajdujących się w pamięci urządzenia obliczeniowego. Dane te muszą zostać skopiowane do pamięci gospodarza. Wykorzystamy w tym celu funkcję `clEnqueueReadBuffer`, której zestaw parametrów jest bardzo podobny do wcześniej omówionej funkcji przenoszącej dane z pamięci gospodarza do pamięci urządzenia obliczeniowego – `clEnqueueWriteBuffer`.

Sposób wywołania tej funkcji jest następujący:

```
cli_err_num = clEnqueueReadBuffer( cl_compute_command_queue, device_out_txt, CL_TRUE, 0, dataBytes, &host_out_txt[0], 0, NULL, NULL);
```

Warto porównać powyższe wywołanie z wcześniej przedstawionym wywołaniem zapisu danych do pamięci urządzenia obliczeniowego, aby przekonać się, iż są to bliźniacze funkcje, różnią się tylko wskaźnikami do źródeł danych. Tym razem dane będą kopiowane ze wskaźnika `device_out_txt` do pamięci hosta wskazywanej przez adres tablicy `host_out_txt`.

Na tym kończy się obsługa procedury obliczeniowej, jednak utworzyliśmy kilka nowych obiektów, które mogą powodować wycieki pamięci, więc obiekty te powinny zostać usunięte. Usunięcie obiektu jądra czy obiektu kontekstu obliczeniowego realizujemy w następujący sposób:

```
clReleaseKernel(kernel);

clReleaseContext(cl_compute_context);
```

Należy także usunąć wskaźniki wskazujące pamięć dla danych umieszczonych w pamięci globalnej urządzenia obliczeniowego:

```
clReleaseMemObject(device_out_txt);

clReleaseMemObject(device_in_txt);
```

Jak widać, trzeba wykonać sporo dodatkowych czynności, aby uruchomić jądro obliczeniowe. Cały kod źródłowy dotyczący przykładu opisanego w tym rozdziale, bez żadnych skrótów, znajduje się w listingu 4.4.

Listing 4.4. Kompletny kod źródłowy przykładu *Hello World*!

```
#include <stdio.h>

#include <stdlib.h>

#if defined (__APPLE__) || defined(MACOSX)

    #include <OpenCL/opencl.h>

#else

    #include <CL/opencl.h>

#endif

const char* kernelSource =

    "__kernel void helloDecode(__global char *inTxt, __global char *outTxt, int N)\n"

    "{ \n"

    "        int idx = get_global_id(0); \n"

    " \n"

    "        if(idx < N) \n"

    "        { \n"

    "            outTxt[idx] = inTxt[idx] – 5;\n"

    "        }\n"

    "}\n"

    "\n";

cl_uint ocl_plat_idx = 0;

cl_uint ocl_dev_idx = 0;

cl_uint clui_num_platforms = 0;

cl_uint clui_num_devices = 0;
```

```
cl_int cli_err_num;

cl_device_type device_type;

cl_platform_id *opencl_platforms = NULL;

cl_device_id *platform_devices = NULL;

cl_context cl_compute_context = NULL;

cl_command_queue cl_compute_command_queue = NULL;

cl_mem device_in_txt;

cl_mem device_out_txt;

cl_program cl_kernel_program;

cl_kernel kernel = NULL;

size_t globalWorkSize;

int main(int argc, char* argv[])

{

     char host_in_txt[] = {’M’, ’j’, ’q’, ’q’, ’t’, ’%’, ’\\’, ’t’, ’w’, ’q’, ’i’,
’&’, 0};

    char host_out_txt[]={0,0,0,0,0,0,0,0,0,0,0,0,0};

    const int N = (sizeof(host_in_txt) / sizeof(char)) – 1;

    size_t dataBytes = sizeof(char)*N;

    cli_err_num = CL_SUCCESS;

    cli_err_num = clGetPlatformIDs(0, NULL, &clui_num_platforms);

    if (cli_err_num != CL_SUCCESS)

    {

        printf(”Brak dostępu OpenCL!!!\n”);

        return 0;

    }

    printf(”Dostępna liczba platform OpenCL: (%d).\n”, clui_num_platforms);

            opencl_platforms   =   (cl_platform_id*)malloc(   clui_num_   platforms   *
sizeof(cl_platform_id) );

    if ( opencl_platforms == NULL )

    {
```

```
        printf(”Brak alokacji pamięci dla platform OpenCL!!!\n” );

        return 0;

    }

    cli_err_num = clGetPlatformIDs (clui_num_platforms, opencl_platforms, NULL);

    if (cli_err_num != CL_SUCCESS) {

        free((void*)opencl_platforms);

        opencl_platforms = NULL;

        return 0;

    }

      cli_err_num = clGetDeviceIDs( opencl_platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL,
&clui_num_devices);

    if (cli_err_num != CL_SUCCESS) {

        free((void*)opencl_platforms);

        opencl_platforms = NULL;

        printf(”Błąd w odczycie liczby urządzeń obliczeniowych!\n”);

        return 0;

    }

    printf(”Dostępna liczba urządzeń obliczeniowych: (%d).\n”, clui_num_devices);

    platform_devices = (cl_device_id*)malloc(clui_num_devices* sizeof(cl_device_id));

          cli_err_num = clGetDeviceIDs( opencl_platforms[0], CL_DEVICE_TYPE_ALL,
clui_num_devices, platform_devices, NULL);

    if (cli_err_num != CL_SUCCESS) {

                printf(”Błąd: Nie udało się odczytać informacji o urządzeniach
obliczeniowych!\n”);

        return EXIT_FAILURE;

    }

             cl_compute_context = clCreateContext( NULL, clui_num_ devices,
&platform_devices[0], NULL, NULL, &cli_err_num);

    if (!cl_compute_context)

    {

        printf(”Błąd: Nie udało się utworzyć kontekstu obliczeniowego!\n”);
```

```
        return EXIT_FAILURE;

    }

            cl_compute_command_queue   =   clCreateCommandQueue(   cl_   compute_context,
platform_devices[0], 0, &cli_err_num);

    if (!cl_compute_command_queue)

    {

        printf(”Błąd: Nie udało się utworzyć kolejki poleceń!\n”);

        return EXIT_FAILURE;

    }

      device_in_txt  =  clCreateBuffer(  cl_compute_context,  CL_MEM_READ_ONLY,  dataBytes,
NULL, &cli_err_num);

     device_out_txt = clCreateBuffer( cl_compute_context, CL_MEM_WRITE_ONLY, dataBytes,
NULL, &cli_err_num);

       cli_err_num  =  clEnqueueWriteBuffer(  cl_compute_command_  queue,  device_in_txt,
CL_TRUE, 0, dataBytes,

                                                        &host_in_txt[0], 0, NULL, NULL);

       cl_kernel_program  =  clCreateProgramWithSource(  cl_compute_  context,  1,  (const
char**)&kernelSource, NULL, &cli_err_num);

    if (!cl_kernel_program)

    {

          printf(”Błąd: Nie udało się utworzyć programu obliczeniowego! Kod = %d\n”,
cli_err_num);

        return EXIT_FAILURE;

    }

            cli_err_num   =   clBuildProgram(   cl_kernel_program,   clui_num_devices,
platform_devices, NULL, NULL, NULL);

    kernel = clCreateKernel(cl_kernel_program, ”helloDecode”, &cli_err_num );

    if (!kernel || cli_err_num != CL_SUCCESS)

    {

        printf(”Błąd: Nie udało się utworzyć jądra obliczeniowego!\n”);

        exit(1);

    }
```

```
    cli_err_num = clSetKernelArg( kernel, 0, sizeof(cl_mem), &device_in_txt );

    cli_err_num |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &device_out_txt );

    cli_err_num |= clSetKernelArg( kernel, 2, sizeof(int), &N );

    globalWorkSize = N;

     cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_ queue, kernel, 1, NULL,
&globalWorkSize, NULL, 0, NULL, NULL );

    if (cli_err_num)

    {

        printf(”Błąd: nie udało się uruchomić jądra obliczeniowego!\n”);

        return EXIT_FAILURE;

    }

       cli_err_num = clEnqueueReadBuffer( cl_compute_command_ queue, device_out_txt,
CL_TRUE, 0, dataBytes,

                                                                host_out_txt, 0, NULL, NULL);

    if (cli_err_num != CL_SUCCESS)

    {

        printf(”Błąd: Nie udało się odczytać danych wyjściowych! %d\n”, cli_err_num);

        exit(1);

    }

    printf(”Odczytana wartość [%s]\n”, &host_out_txt[0]);

    clReleaseKernel(kernel);

    clReleaseProgram(cl_kernel_program);

    clReleaseCommandQueue(cl_compute_command_queue);

    clReleaseMemObject(device_out_txt);

    clReleaseMemObject(device_in_txt);

    clReleaseContext(cl_compute_context);

    free((void*)platform_devices);

    free((void*)opencl_platforms);

    return 0;

}
```

4.3. Dodawanie wektorów

W tym przykładzie będziemy obliczać sumę dwóch ciągów liczb, czyli wektorów. Z punktu widzenia OpenCL wektory będą reprezentowane przez jednowymiarowe tablice.

W języku C czy C++ operację dodawania dwóch tablic możemy zapisać np. za pomocą jednej pętli `for`, co przedstawia się to następująco:

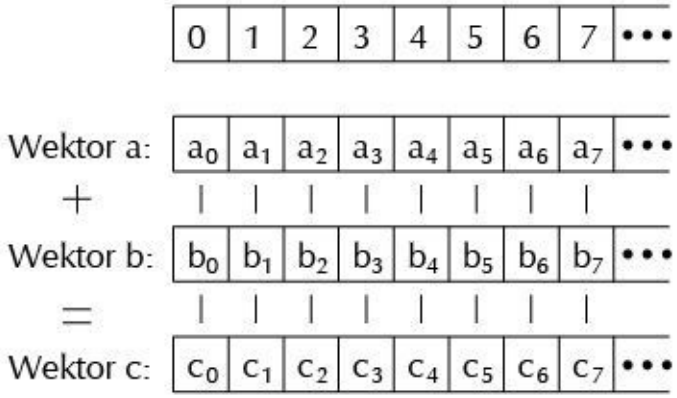
```
int i;

float a[N],b[N],c[N];

for( i=0 ; i<N ; i++ ) {

    c[i] = a[i] + b[i];

}
```

Poszczególne elementy tablicy o nazwie `c` stanowią rezultat sumy elementów tablic `a` oraz `b`. Podstawowym wymaganiem jest, aby wszystkie trzy tablice miały takie same wymiary. Zamiana powyższego kodu na równoważny kod w OpenCL nie jest trudna, a więcej troski będzie wymagać przygotowanie danych i obsługa jądra obliczeniowego.

Operacja dodawania wektorów charakteryzuje się łatwością podania programu równoległego, co bardzo sprzyja implementacji równoległej. Poszczególne elementy tablicy mogą być przetwarzane w sposób niezależny. Dysponując wieloma jednostkami obliczeniowymi, można dodawać różne fragmenty wektorów, nie martwiąc się o spójność danych. Obliczanie sumy np. siódmego elementu zapiszemy jako

```
a[7] = b[7] + c[7];
```

obliczenie tego częściowego wyniku nie ma żadnego wpływu na obliczenia pozostałych elementów wektora. Na rysunku 4.3 przedstawiono schemat obliczania sumy wektora. Każdy element jest dodawany niezależnie od pozostałych.



Rys. 4.3. Ogólna idea przetwarzania równoległego wektora za pomocą wielu jednostek obliczeniowych

4.3.1. Jądro obliczeniowe

Jądro obliczeniowe OpenCL w dwóch wersjach zostało przedstawione na listingu 4.5. Przekazujemy trzy lub cztery argumenty. W argumencie pierwszym o nazwie `a` podajemy wskaźnik do wektora wynikowego, a dwa pozostałe parametry o nazwach `b` oraz `c` stanowią wektory wejściowe. Ostatni argument o nazwie `N` określa wielkość wektorów.

W pierwszym argumencie nie pojawia się słowo `const`, bowiem jest to wektor, do którego będziemy kierować obliczone elementy sumy. Słowo `const` pojawia się w dwóch pozostałych wektorach, bowiem z tych zmiennych tylko odczytujemy dane.

Nie występuje także pętla `for`, gdyż jej rolę przejęły poszczególne jednostki obliczeniowe. Dlatego w pierwszej kolejności odczytujemy indeks, jakim będziemy się zajmować:

```
int i = get_global_id(0);
```

Wartości funkcji `get_global_id` są związane z siatką obliczeniową. W naszym przypadku będzie ona miała kształt i wielkość wektora, krótko mówiąc, będzie to liczba z zakresu od zera do `N-1`, gdzie `N` to liczba elementów, co oznacza iż `get_global_id` będzie zwracać numery indeksów dla naszych tablic reprezentujących wektory. Wykorzystujemy tu ideę pokazaną na rys. 4.3. Proces sumowania możemy zrealizować na dwa sposoby: pojedyncza jednostka może realizować tylko jedną sumę lub pojedyncza jednostka obliczać będzie kilka sum. W tym pierwszym przypadku, jeden indeks/element tablicy jest przetwarzany tylko przez jeden rdzeń obliczeniowy. Nie jest to efektywne rozwiązanie, bowiem lepiej, aby pojedyncza jednostka obliczeniowa przetwarzała więcej elementów wektora. W naszym przypadku na razie pozostaniemy przy prostszym rozwiązaniu, tzn. będziemy obliczać tylko sumę jednego elementu bezpośrednio wskazywanego przez indeks siatki obliczeniowej.

Jeśli wielkość siatki jest równa wielkości przetwarzanego wektora, to można usunąć instrukcję warunkową odpowiedzialną za sprawdzanie, czy został przekroczony zakres rozmiaru wektora. W takim przypadku do obowiązku programisty należy precyzyjne określenie rozmiaru siatki obliczeniowej na równy co do liczby elementów zawartych w wektorze. Z drugiej strony, obecność instrukcji `if` pozwala na zastosowanie większej siatki obliczeniowej o wielkości dopasowanej do liczby dostępnych rdzeni obliczeniowych, co może się przyczynić do większej wydajności końcowych obliczeń. Należy jednak podkreślić raz jeszcze, że obecność instrukcji `if` zabezpiecza przed przekroczeniem zakresu przydzielonej pamięci, jeśli rozmiar siatki byłby większy niż przetwarzany wektor.

Listing 4.5. Suma dwóch wektorów w OpenCL (wersja pierwsza `sum1` pozbawiona instrukcji warunkowej oraz wersja `sum2` z instrukcją `if` sprawdzającą zakres danych)

```
__kernel void sum1( __global const float *a, __global const float *b, __global float
*c ) {

    int gid = get_global_id(0);

    c[gid] = a[gid] + b[gid];

}

__kernel void sum2( __global const float* a, __global const float *b, __global float
*c, const int N) {

    int gid = get_global_id(0);
```

```
    if(gid < N)

        c[gid] = a[gid] + b[gid];

}
```

Przed uruchomieniem obliczeń należy przygotować dane i przesłać je do urządzenia obliczeniowego. Deklaracja zmiennych po stronie gospodarza może zostać zrealizowana w postaci statycznych tablic:

```
float host_in_a[N];

float host_in_b[N];

float host_out_c[N];
```

gdzie: `N` – stała oznaczająca wielkość wektora, np.:

```
const int N = 16;
```

Niezbędne jest też obliczenie rozmiaru danych, jakie będą kopiowane pomiędzy pamięcią urządzenia a pamięcią gospodarza. Ponieważ mamy do czynienia z prostą tablicą, to wystarczy pomnożyć liczbę elementów przez wielkość pojedynczego elementu:

```
size_t dataBytes = sizeof(float)*N;
```

Należy też przygotować zmienne tablicowe po stronie hosta. W naszym przykładzie tablica `host_in_a` zostanie wypełniona elementami w kolejności rosnącej, a tablica `host_in_b` w kolejności malejącej. W zmiennej `host_out_c` umieszczone zostaną zera, a właściwe dane zostaną umieszczone w tej zmiennej dopiero po zakończeniu obliczeń przez jądro obliczeniowe. Zamieniając te objaśnienia na odpowiednią pętlę `for`, otrzymujemy następujące linie kodu:

```
for(i=0;i<N;i++) {

    host_in_a[i]=static_cast<float>(i);

    host_in_b[i]=static_cast<float>(N-i);

    host_out_c[i]=0.0f;

}
```

Kolejne czynności są identyczne jak omówione w poprzednim rozdziale, dlatego nie będziemy ich w tym miejscu powtarzać, możemy od razu przejść do utworzenia odpowiednich buforów w pamięci urządzenia:

```
device_in_a = clCreateBuffer( cl_compute_context, CL_MEM_READ_ONLY, dataBytes, NULL, &cli_err_num);

device_in_b = clCreateBuffer( cl_compute_context, CL_MEM_READ_ONLY, dataBytes, NULL, &cli_err_num);

device_out_c = clCreateBuffer( cl_compute_context, CL_MEM_WRITE_ONLY, dataBytes, NULL, &cli_err_num);
```

Dwa pierwsze bufory reprezentują wektory a oraz b, a trzeci bufor – tylko do zapisu – to bufor wektora c, gdzie zapiszemy wynik obliczeń. Deklaracje są identycznie jak w poprzednim przykładzie.

Bufory dla wektorów a oraz b wymagają przesłania danych z pamięci gospodarza, co zrealizujemy w sposób podobny jak w poprzednim przykładzie:

```
cli_err_num = clEnqueueWriteBuffer( cl_compute_command_queue, device_in_a, CL_TRUE, 0, dataBytes, &host_in_a[0], 0, NULL, NULL);

cli_err_num = clEnqueueWriteBuffer( cl_compute_command_queue, device_in_b, CL_TRUE, 0, dataBytes, &host_in_b[0], 0, NULL, NULL);
```

Do zmiennej c nie warto przesyłać danych, bowiem po wywołaniu jądra obliczeniowego będziemy odczytywać dane z urządzenia obliczeniowego. Kolejnym zadaniem jest utworzenie obiektu jądra oraz kompilacja, ale te czynności są podobne do poprzednio omówionych, więc możemy przejść do ustalenia parametrów jądra. W zależności od tego, jaką wersję będziemy implementować, mamy trzy lub cztery parametry. W obu przypadkach trzy pierwsze argumenty określamy w podobny sposób:

```
cli_err_num = clSetKernelArg( kernel, 0, sizeof(cl_mem), &device_in_a );

cli_err_num |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &device_in_b );

cli_err_num |= clSetKernelArg( kernel, 2, sizeof(cl_mem), &device_out_c );
```

Różnica tkwi w ostatnim parametrze. Brakujący czwarty parametr dla drugiej wersji jądra, gdzie podajemy dodatkowy parametr N określający wielkość tablicy, jest definiowany w następujący sposób:

```
cli_err_num |= clSetKernelArg( kernel, 3, sizeof(int), &N );
```

Mimo obecności dodatkowego argumentu dla jądra, jeśli wykorzystamy tylko globalną siatkę obliczeniową, to wywołanie jądra ma w obu przypadkach identyczną postać:

```
globalWorkSize = N;

cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 1, NULL, &globalWorkSize, NULL, 0, NULL, NULL );
```

Do zmiennej globalWorkSize przepisywany jest wymiar wektora i siatka obliczeniowa ma kształt 1 x N. Jednakże wyznaczony wektor, będący sumą dwóch innych wektorów, znajduje się nadal w urządzeniu obliczeniowym, toteż należy go przesłać do pamięci hosta. Proces odczytania danych będzie podobny do podanego w poprzednim przykładzie w rozdz. 4.2:

```
cli_err_num = clEnqueueReadBuffer( cl_compute_command_queue, device_out_c, CL_TRUE, 0, dataBytes, &host_out_c[0], 0, NULL, NULL);
```

Pozostało już tylko wyświetlić dane, które zostały przeniesione z pamięci urządzenia OpenCL do pamięci gospodarza. Najłatwiej wykorzystać do tego printf oraz pętlę for:

```
printf("Obliczone wartości:\n");

for(i=0;i<N;i++)
```

```
    printf("%f ", host_out_c[i]);

printf("\n");
```

4.3.2. Wersja dla C++

Poprzedni przykład został opracowany za pomocą API OpenCL dla języka C. W tym przykładzie zostanie użyte API dla języka C++. Można oczekiwać uproszczenia wykonywania praktycznie wszystkich czynności, jakie należy wykonać, aby uruchomić jądro obliczeniowe. Widać to po długości listingów przedstawiających pełne kody źródłowe – listing 4.6 dla C++ jest krótszy niż listing 4.2, co prawda są to różne przykłady, ale i tak różnica w długości jest widoczna, szczególnie jeśli weźmiemy pod uwagę tylko czynności wykonywane w kontekście OpenCL.

Pominiemy utworzenie obiektów platformy, urządzenia oraz kontekstu obliczeniowego, bowiem było to już omówione w rozdz. 4.1.2, a także przygotowanie danych po stronie hosta, gdyż korzystamy ze zwykłych wskaźników `float`, aby przechowywać dane. Kolejnym ważnym zadaniem jest budowa jądra obliczeniowego.

Pierwszą czynnością, jaką trzeba wykonać, jest utworzenie obiektu, w którym znajdą się źródła jądra obliczeniowego. Należy też utworzyć parę, której pierwszym elementem jest wskaźnik typu `char*`, a drugim liczba znaków obliczona za pomocą funkcji `strlen`:

```
cl::Program::Sources source(1, std::make_pair(kernelSource, strlen(kernelSource)));
```

Na podstawie obiektu zawierającego kod źródłowy tworzony jest obiekt programu:

```
cl::Program program = cl::Program(context, source);
```

A następnie można utworzyć postać binarną programu dla podanych urządzeń obliczeniowych:

```
program.build(devices);
```

Odczytanie właściwego jądra, czyli funkcji `sum`, również realizuje się jedną linią kodu, tworząc odpowiedni obiekt:

```
cl::Kernel kernel(program, "sum", &cli_err);
```

Przed uruchomieniem jądra obliczeniowego trzeba jeszcze utworzyć bufory dla danych. Bufor wejściowy wektora a tworzy się następująco:

```
cl::Buffer device_in_a( context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
N*sizeof(float), host_in_a);
```

Dodatkowo, chcemy od razu skopiować zawartość `host_in_a` do pamięci urządzenia obliczeniowego, dlatego za pomocą operatora bitowego oraz flagi `CL_MEM_COPY_HOST_PTR` zleca się wykonanie operacji kopiowania[4] danych.

Podobnie postępujemy w przypadku bufora dla wektora `b`. Bufor wyjściowy nie wymaga kopiowania danych wejściowych, dlatego nie podajemy czwartego argumentu:

```
cl::Buffer device_out_c( context, CL_MEM_WRITE_ONLY, N*sizeof(float));
```

Możemy skopiować dane samodzielnie, tworząc oczywiście wcześniej kolejkę poleceń:

```
cl::CommandQueue queue(context, devices[0], 0, &cli_err);
```

i dokonać kopiowania danych z pamięci hosta do pamięci urządzenia obliczeniowego:

```
queue.enqueueWriteBuffer(device_in_a, true, 0, N*sizeof(float), (void*)host_in_a);
```

Ustalenie parametrów jądra obliczeniowego, np. o indeksie zerowym oraz trzecim, sprowadza się do użycia metody `setArg`:

```
kernel.setArg(0, device_in_a );

kernel.setArg(3, sizeof(int), (void*)&N);
```

Pozostało już tylko uruchomić jądro obliczeniowe, jednak skorzystamy z dodatkowego mechanizmu, a mianowicie utworzymy zmienną zdarzenia:

```
cl::Event event;
```

Wywołanie jądra obliczeniowego jest następujące:

```
queue.enqueueNDRangeKernel( kernel, cl::NullRange, cl::NDRange(N,1), cl::NullRange,
NULL, &event);
```

Trzeci parametr `cl::NDRange(N,1)` określa wymiar siatki globalnej, jednak w ostatnim argumencie otrzymano obiekt zdarzenia, więc dla pewności, że jądro zakończyło swoje obliczenia, wystarczy wywołać metodę `wait`, aby program sterujący po stronie gospodarza zaczekał na zakończenie obliczeń:

```
event.wait();
```

Pozostaje tylko odczytać wektor wynikowy:

```
queue.enqueueReadBuffer( device_out_c, true, 0, N*sizeof(float), (void*)host_out_c);
```

Listing 4.6 zawiera kompletny kod źródłowy. Warto zwrócić jeszcze uwagę na to, iż poszczególne czynności związane z tworzeniem obiektów, kompilacją jądra i jego wykonaniem są objęte sekcją `try`, a ewentualne błędy zostaną zgłoszone jako wyjątki. Choć można odczytywać informacje o kodach, podobnie jak w zwykłym API, np. podczas tworzenia jądra obliczeniowego, czy kolejki poleceń.

Listing 4.6. Kompletny kod źródłowy dodawania wektorów wykorzystujący API C++ dla technologii OpenCL

```
#include <ctime>

#include <fstream>

#include <iostream>

#include <exception>

#define __CL_ENABLE_EXCEPTIONS
```

```
#define __NO_STD_VECTOR
#if defined (__APPLE__) || defined(MACOSX)
    #include <OpenCL/cl.hpp>
#else
    #include <CL/opencl.h>
    #include <CL/cl.hpp>
#endif
const int N = 16;
const char* kernelSource =
    "__kernel void sum(\n"
    " __global const float *a, __global const float *b,\n"
    " __global float *c, const int N ) {\n"
    "\n"
    " int id = get_global_id(0); \n"
    " if ( id < N ) \n"
    " c[id] = a[id] + b[id];\n"
    "\n"
    "}\n"
    "\n";
int i;
cl::vector<cl::Platform> platforms;
cl::vector<cl::Device> devices;
cl::vector<cl::Kernel> allKernels;
cl_int cli_err;
float *host_in_a;
float *host_in_b;
float *host_out_c;
void printArray( std::string header, const float *vec)
{
```

```
    std::cout<<"\n"<<header<<"\n";
    for(int i=0 ; i<N ; i++)
    {
        std::cout<<vec[i]<<" ";
    }
    std::cout<<"\n";
}
void fillArray( float *vec, int _seed )
{
    int seed = (unsigned int)time(NULL) % _seed;
    srand(seed);
    for(int i = 0; i < N; i++)
    {
        vec[i] = float(10*rand()/(RAND_MAX + 1.0));
    }
}
int main(int argc, char* argv[])
{
    try {
        host_in_a = new float[N];
        fillArray(host_in_a, 1000);
        printArray("inputA:", host_in_a);
        host_in_b = new float[N];
        fillArray(host_in_b, 2123);
        printArray("inputB:", host_in_b);
        host_out_c = new float[N];
        cl::Platform::get(&platforms);
        platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);
                    cl_context_properties  properties[]  =  {  CL_CONTEXT_  PLATFORM,
```

```
(cl_context_properties)(platforms[0])(), 0};

        cl::Context context(CL_DEVICE_TYPE_GPU, properties);

        cl::vector<cl::Device> devices = context.getInfo <CL_CONTEXT_DEVICES>();

                        cl::Program::Sources   source(1,   std::make_pair   (kernelSource,
strlen(kernelSource)));

        cl::Program program = cl::Program(context, source);

        program.build(devices);

        cl::Kernel kernel(program, ”sum”, &cli_err);

        cl::Buffer device_in_a( context, CL_MEM_READ_ONLY, N*sizeof(float));

        cl::Buffer device_in_b( context, CL_MEM_READ_ONLY, N*sizeof(float));

        cl::Buffer device_out_c( context, CL_MEM_WRITE_ONLY, N*sizeof(float));

        cl::Event event;

        cl::CommandQueue queue(context, devices[0], 0, &cli_err);

        queue.enqueueWriteBuffer(device_in_a, true, 0, N*sizeof(N), (void*)host_in_a);

                        queue.enqueueWriteBuffer(device_in_b,   true,   0,   N*sizeof(N),
(void*)host_in_b);

        kernel.setArg(0, device_in_a );

        kernel.setArg(1, device_in_b );

        kernel.setArg(2, device_out_c );

        kernel.setArg(3, sizeof(int), (void*)&N);

                queue.enqueueNDRangeKernel(  kernel,  cl::NullRange,  cl::NDRange(N,1),
cl::NullRange, NULL, &event);

        event.wait();

                  queue.enqueueReadBuffer(  device_out_c,  true,  0,  N*sizeof(float),
(void*)host_out_c);

        printArray(”output:”, host_out_c);

    }

    catch(cl::Error e) {

        std::cout << e.what() << ”: Error code ” << e.err() << std::endl;

    }

    return 0;
```

```
}
```

4.3.3. Inne operacje na wektorach do ćwiczeń

W ramach dodatkowych ćwiczeń warto się pokusić o implementację kilkunastu innych przypadków operacji na wektorach. Od najprostszych, jak np. kwadrat poszczególnych elementów wektora:

```
c[i] = a[i] * a[i]
```

bądź też dodanie wartości skalarnej `s` do wektora:

```
c[i] = a[i] + s
```

Możemy operacje te skomplikować, stosując kilka wektorów `a`, `b`, `c` oraz kilka wartości skalarnych `k`, `l`, `m` i wykonując operację typu:

```
d[i] = k*a[i] + l*b[i] + m*c[i].
```

Jądro obliczeniowe dla globalnej siatki obliczeniowej będzie miało postać podobną jak w poprzednio podanych przykładach, co potwierdza listing 4.7.

Listing 4.7. Operacja na kilku wektorach i skalarach

```
__kernel void sum1( __global const float *a,

    __global const float *b,

    __global const float *c,

    __global float *d,

const float k, const float l, const float m ) {

    int gid = get_global_id(0);

    cd[gid] = k*a[gid] + l*b[gid] + m*c[gid];

}
```

Dobrymi przykładami do ćwiczeń są operacje na wektorach znajdujące się w bibliotece BLAS[5]. Na początek warto (i trzeba) wybierać prostsze operacje, jak np. `saxpy`. Jej opis jest następujący:

```
a[i] = k * b[i] + a[i]
```

Implementacja naturalnie będzie podobna do kodu z listingu 4.7, co potwierdza listing 4.8.

Listing 4.8. Jądro obliczeniowe dla operacji typu `saxpy`.

```
__kernel void saxpy(const unsigned int N, const float k,

        __global const float*b, __global float*a)

{
```

```
    int gid = get_global_id(0);

    if( gid < N)

    a[gid] = k* b[gid] + a[gid];

}
```

Zastosowanie tylko globalnej siatki obliczeniowej sprawia, że jądro obliczeniowe ma bardzo prostą postać, ale poszczególne wątki robocze wykonują tylko proste dodawania i mnożenia. Warto przekształcić procedurę obliczeniową, aby można było wykonywać więcej operacji. Jeśli zastosujemy także siatkę lokalną, postać procedury obliczeniowej będzie nieco bardziej skomplikowana, ale za to jej wydajność będzie większa. Przyczyni się to do lepszego wykorzystania poszczególnych jednostek obliczeniowych.

4.4. Siatka lokalna oraz globalna

W przykładzie dodawania wektorów wykorzystywana była tylko globalna siatka obliczeniowa. Upraszcza to bardzo kod jądra obliczeniowego, bo wielkość oraz kształt siatki odpowiada dokładnie wielkości danych, które są przetwarzane. Poniżej podano przykład, który pozwoli nam lepiej poznać pojęcie globalnej oraz lokalnej siatki obliczeniowej.

Zadanie dla jądra obliczeniowego to przygotowanie wektora danych w bardzo specyficzny sposób. Wartości elementów wektora będą odzwierciedlać ich miejsca w siatce obliczeniowej z uwzględnieniem parametrów siatki globalnej, lokalnej oraz przesunięcia. Ogólnie pojedyncza wartość będzie miała postać:

```
XYxy
```

gdzie: X oraz Y – współrzędne siatki globalnej, x oraz y – współrzędne siatki lokalnej.

Wartość danego elementu będzie wyznaczana według następującego kodu:

```
float value = gid0 * 10.0f + gid1 * 1.0f;

value += lid0 * 0.1f + lid1 * 0.01f;
```

gdzie: `gid0` oraz `gid1` – odpowiednio pierwszy i drugi identyfikator dla siatki obliczeniowej.

Użyte zostały dwa identyfikatory, więc najlepiej konstrukcja siatki będzie widoczna w przypadku dwuwymiarowym, jednak pierwsza linia wyznaczy tylko współrzędne całkowite. Druga linia mnoży współrzędne lokalne przez wartość 0.1 dla pierwszej współrzędnej oraz 0.01 dla drugiej. Inaczej mówiąc, będziemy się posługiwać dziesiętnymi oraz setnymi częściami. W finalnej wersji, ze względu na zaokrąglenia, najlepiej będzie przemnożyć tak wyznaczoną wartość np. przez 100, aby otrzymać pierwotnie założoną wielkość oraz odrzucić powstałą część ułamkową w wyniku ograniczonej reprezentcji dla liczby `float`. Dodatkowa wartość ułamkowa będzie tylko utrudniać odczytanie współrzędnych, więc aby zrealizować tę poprawkę, podczas zapisu końcowej wartości należy dodatkowo użyć funkcji `trunc`, która odrzuca część ułamkową:

```
output[index] = trunc(value * 100);
```

Innym sposobem jest zastosowanie funkcji `round`, która zaokrągla wyniki:

```
output[index] = round(value * 100);
```

Aby otrzymać poprawne wartości, należy przygotowywać odpowiednio małe siatki obliczeniowe, tak aby numery indeksów miały nie więcej niż dwie cyfry.

Przykładowo, jeśli chcemy wygenerować współrzędne dla siatki globalnej o wymiarach 4 × 4, gdzie nie będzie stosowana siatka lokalna (wymiar siatki lokalnej w takim przypadku to 1 × 1), to otrzymamy następujący ciąg wartości:

```
0, 1000, 2000, 3000, 100, 1100, 2100, 3100, 200, 1200, 2200, 3200, 300, 1300, 2300,
3300
```

Na przysłowiowy pierwszy rzut oka nie sugeruje on żadnego interesującego układu, ale jeśli dodatkowo podzielimy otrzymane dane na cztery wiersze, po cztery elementy w każdym i poszczególne wartości zostaną wyrównane i uzupełnione o nieznaczące zera:

```
0000 1000 2000 3000

0100 1100 2100 3100

0200 1200 2200 3200

0300 1300 2300 3300
```

to otrzymany zestaw danych uwypukla wartości poszczególnych identyfikatorów.

Łatwo zaobserwować zastosowanie parametru `offset`, jeśli np. zostanie użyta para (1,1):

```
1100 2100 3100 4100

1200 2200 3200 4200

1300 2300 3300 4300

1400 2400 3400 4400
```

Zgodnie z oczekiwaniami do pierwszej i drugiej współrzędnej globalnej dodana została wartość 1. Wykorzystanie offsetu w postaci (1,2) oznacza, że druga współrzędna zostanie zwiększona o wartość 2:

```
1200 2200 3200 4200

1300 2300 3300 4300

1400 2400 3400 4400

1500 2500 3500 4500
```

Zastosowanie siatki lokalnej, np. o wymiarach 2 × 2 (bez dodatkowego przesunięcia) oznacza, że nadal mamy 16 elementów, ale tym razem indeksy będą przedstawiały się następująco:

```
0000 1010 2000 3010

0101 1111 2101 3111

0200 1210 2200 3210

0301 1311 2301 3311
```

Mamy cztery grupy elementów, w których trzecia i czwarta cyfra przyjmuje wartości `00`, `01`, `10`, `11` i są to współrzędne w ramach siatki lokalnej. Będzie to bardzo widoczne, jeśli siatka globalna zostanie zwiększona do wymiarów np. 6 × 8, a siatka lokalna 3 × 4:

```
0000 1010 2020 3000 4010 5020

0101 1111 2121 3101 4111 5121

0202 1212 2222 3202 4212 5222

0303 1313 2323 3303 4313 5323

0400 1410 2420 3400 4410 5420

0501 1511 2521 3501 4511 5521

0602 1612 2622 3602 4612 5622

0703 1713 2723 3703 4713 5723
```

Kod źródłowy jądra wykorzystanego w tym przykładzie został przedstawiony w listingu 4.9. Istotny jest sposób wywołania tego jądra obliczeniowego, który naturalnie wymaga konfiguracji zmiennych odpowiedzialnych za ostateczny kształt siatki obliczeniowej.

Ponieważ generowane wcześniej przykłady to struktury dwuwymiarowe, dlatego ustalmy, że siatka jest dwuwymiarowa:

```
size_t dim = 2;
```

Następnie przygotujemy trzy zmienne opisujące wymiary siatki globalnej, lokalnej oraz wartość przesunięcia globalnego:

```
size_t global_size[] = {6, 8};

size_t local_size[] = {3, 4};

size_t global_offset[] = {0, 0};
```

Uruchomienie jądra obliczeniowego sprowadza się do wywołania funkcji `clEnqueueNDRangeKernel` w następujący sposób:

```
cli_err_num = clEnqueueNDRangeKernel(cl_compute_command_queue, prepare_vector_kernel,
dim, global_offset, global_size, local_size, 0, NULL, NULL);
```

Ważny jest sposób odpowiedniego przetworzenia otrzymanego wektora, jednak ten problem będzie tematem przykładu podanego w rozdz. 6.1.3, gdzie procedura obliczeniowa z listingu 4.9 będzie

uruchamiana z poziomu języka Python.

Listing 4.9. Jądro obliczeniowe wyznaczające wektor o wartościach reprezentujących zastosowaną siatkę obliczeniową

```
__kernel void prepare_vector(__global float *output)

{

    size_t gid0 = get_global_id(0);

    size_t gid1 = get_global_id(1);

    size_t gs0 = get_global_size(0);

    size_t lid0 = get_local_id(0);

    size_t lid1 = get_local_id(1);

    int outX0 = gid0 – get_global_offset(0);

    int outY1 = gid1 – get_global_offset(1);

    int index = outY1 * gs0 + outX0;

    float value = gid0 * 10.0f + gid1 * 1.0f;

    value += lid0 * 0.1f + lid1 * 0.01f;

    output[index] = round(value * 100);

}
```

4.5. Zlecanie i wykonywanie zadań

Większość algorytmów, jakie implementuje się w ramach OpenCL, wykorzystuje siatkę obliczeniową (przestrzeń indeksową) do realizacji obliczeń. OpenCL oferuje też inną możliwość, a mianowicie zlecanie tzw. zadań, które można wykonać za pomocą API OpenCL. Zadania te powinny realizować mniej czasochłonne obliczenia, bowiem nie wykorzystują przetwarzania równoległego, które uzyskuje się dzięki przestrzeni indeksowej.

Mimo braku obsługi siatki obliczeniowej, procedura zaprezentowana w listingu 4.10 jest praktycznie identyczna jak w poprzednio podanych przykładach. Brakuje odczytu identyfikatora jednostki roboczej, ale pozostałe elementy, jak np. deklaracje parametrów wejściowych pozostają niezmienne.

W procedurze obliczeniowej realizujemy dodawanie zawartości tablic `A` oraz `B` element po elemencie, a wynik umieszczamy w tablicy `C`. Inaczej mówiąc, jest to typowe dodawanie do siebie dwóch wektorów, przy czym dodawane są tylko cztery elementy. Dodatkowo stosowana jest wartość umieszczona w zmiennej `offset`, aby w ten sposób przesunąć się na inny obszar danych do przetworzenia.

Listing 4.10. Procedura obliczeniowa do realizacji w ramach tzw. zdania OpenCL

```
__kernel void taskAdd(__global float* A, __global float* B, __global float* C, const
```

```
int offset) {

    C[offset + 0] = A[offset + 0] + B[offset + 0];

    C[offset + 1] = A[offset + 1] + B[offset + 1];

    C[offset + 2] = A[offset + 2] + B[offset + 2];

    C[offset + 3] = A[offset + 3] + B[offset + 3];

}
```

Sposób działania tej procedury obliczeniowej łatwo pokazać, zakładając że wektory `A` oraz `B` mają następujące elementy:

```
A: 0 1 2 3 1 2 3 8 4 5 1 5 6

B: 2 4 1 4 5 6 7 3 1 3 3 2 5
```

Jeśli jądro obliczeniowe zostałoby uruchomione z parametrem `offset` równym `0`, to wykonane zostaną tylko pierwsze cztery sumy i wynikowy wektor przyjmie następującą postać:

```
C: 2 5 3 7 ? ? ? ? ? ? ? ? ?
```

Jeśli parametr `offset` miałby wartość równą `5`, to wynikowy wektor będzie miał postać:

```
C: ? ? ? ? ? 8 10 11 5 ? ? ? ?
```

Obecne znaki zapytania w obu przypadkach oznaczają, iż elementy nimi oznaczone nie zostaną obliczone, więc ich wartość zależy od wcześniejszej zawartości wektora `C`.

Czynności zawiązane z przygotowaniem procedury obliczeniowej są identyczne jak w poprzednich przykładach. Począwszy od utworzenia obiektu reprezentującego platformę, urządzenia obliczeniowe, kontekst, aż po proces alokacji pamięci po stronie urządzenia obliczeniowego, która przyjmuje następującą postać:

```
device_in_a = clCreateBuffer( cl_compute_context, CL_MEM_READ_ONLY, dataBytes, NULL, &cli_err_num);

device_in_b = clCreateBuffer( cl_compute_context, CL_MEM_READ_ONLY, dataBytes, NULL, &cli_err_num);

device_out_c = clCreateBuffer( cl_compute_context, CL_MEM_WRITE_ONLY, dataBytes, NULL, &cli_err_num);
```

Zmienna `dataBytes` określa wielkość pojedynczego wektora w bajtach. Przy czym, jeśli porównać podane trzy linie kodu np. z procesem przydzielania pamięci w przykładzie z rozdz. 4.3, to okaże się on identyczny. Analogiczne są też inne zadania związanie z przygotowaniem jądra obliczeniowego, jednak sposób jego wywołania wymaga zastosowania innej funkcji niż poprzednio. Wcześniej należy przygotować parametry przekazywane do jądra obliczeniowego:

```
cli_err_num = clSetKernelArg( kernel, 0, sizeof(cl_mem), &device_in_a );
```

```
cli_err_num |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &device_in_b );

cli_err_num |= clSetKernelArg( kernel, 2, sizeof(cl_mem), &device_out_c );

cli_err_num |= clSetKernelArg( kernel, 3, sizeof(int), &offset );
```

Ważny jest parametr `offset`, stanowiący bazę, od którego elementu będziemy wykonywać operację sumy. Wywołanie jądra obliczeniowego wymaga zastosowania funkcji `clEnqueueTask`:

```
cli_err_num = clEnqueueTask( cl_compute_command_queue, kernel, 0, NULL, NULL );
```

Pierwszy parametr to kolejka poleceń, w drugim podawany jest obiekt reprezentujący jądro obliczeniowe. Kolejne trzy parametry można wykorzystać do przekazania dodatkowych wartości do obsługi zdarzeń związanych z wywołaniem jądra oraz zakończeniem prac odnoszących się do tego jądra obliczeniowego.

4.6. Uwagi o kompilacji jądra obliczeniowego

W poprzednio podanych przykładach podczas tworzenia obiektu reprezentującego procedurę obliczeniową nie były wykorzystywane dodatkowe możliwości związane z zastosowaniem preprocesora, czy też odczytem dokładniejszych informacji o błędach, jakie pojawiły się w trakcie kompilacji jądra obliczeniowego.

Skorzystanie z preprocesora, np. w celu zdefiniowania dodatkowych wielkości, jest możliwe dzięki funkcji `clBuildProgram`, w której w czwartym atrybucie można przekazać dodatkowe parametry, np. wspominane definicje stałych preprocesora za pomocą parametru `–D`. W rozdziale 5.7 podajemy w ten sposób wymiary obrazu wykorzystywane w procedurze obliczeniowej. Aby to zrealizować, należy przygotować w odpowiedni sposób ciąg znaków opisujących potrzebne opcje, np. zdefiniować tablicę typu `char` o odpowiedniej długości:

```
char preprocesorOptions[512];
```

a następnie, korzystając z funkcji `sprintf`, przygotować ciąg znaków:

```
sprintf(preprocesorOptions, ”-D WIDTH=%d – D HEIGHT=%d”, WIDTH, HEIGHT);
```

który, jeśli `WIDTH=512`, a np. `HEIGHT=256`, przyjmie ostatecznie postać:

```
-D WIDTH=512 –D HEIGHT=256
```

Pozostało już tylko przekazać zawartość tablicy `preprocesorOptions` do funkcji tworzącej obiekt procedury obliczeniowej.

Ważnym problemem mogą być także błędy mimowolnie popełnione w procedurze obliczeniowej, np. brak średnika czy nawiasu. Mimo swojej trywialności, uniemożliwią one kompilację jądra obliczeniowego. API technologii OpenCL pozwala na odczytanie komunikatów z kompilacji za pomocą dodatkowej funkcji o nazwie `clGetProgramBuildInfo`. Wcześniej należy rozpocząć proces budowy procedury obliczeniowej za pomocą `clBuildProgram` w typowy dla OpenCL sposób:

```
cli_err_num = clBuildProgram( cl_kernel_program, clui_num_devices, platform_devices,
preprocesorOptions, NULL, NULL);
```

Jeśli wartość zmiennej `cli_err_num` jest równa `CL_SUCESS`, to procedura obliczeniowa została pomyślnie zbudowana, wartość stałej `CL_SUCESS` to zero, więc dowolna inna wartość oznacza błąd. Łatwo to sprawdzić za pomocą następującej instrukcji warunkowej:

```
if (cli_err_num)

{

    printf(”Błąd: budowanie programu nie powiodło się!\n”);

    // pozostałe instrukcje

}
```

Informacja o treści komunikatu odnoszącej się do błędu wymaga deklaracji zmiennej znakowej o odpowiedniej wielkości:

```
size_t len;

char buffer[2048];
```

Odczyt treści komunikatu wymaga podania obiektu `cl_kernel_program` oraz określonego urządzenia, bowiem jądro obliczeniowe będzie kompilowane dla wszystkich urządzeń obliczeniowych. Należy też podać stałą `CL_PROGRAM_BUILD_LOG`, w ten sposób informacje od kompilatora zostaną umieszczone w zmiennej `buffer`:

```
clGetProgramBuildInfo(cl_kernel_program,   platform_devices[0],   CL_PROGRAM_BUILD_LOG,
sizeof(buffer), buffer, &len);
```

Ponieważ jest to tradycyjny ciąg znaków zakończony zerem, to komunikat możemy wyświetlić np. za pomocą `printf`:

```
printf(”%s\n”, buffer);
```

Działanie procedury bądź nawet całego programu należy zakończyć za pomocą słowa `return` lub `exit`:

```
return EXIT_KERNEL_COMPILATION_FAILURE;
```

albo

```
exit( EXIT_KERNEL_COMPILATION_FAILURE );
```

Gdzie `EXIT_KERNEL_COMPILATION_FAILURE` to stała zdefiniowana przez programistę (np. `-10`) oznaczająca błąd kompilacji procedury obliczeniowej.

Przypisy

[1] Pełny kod źródłowy wszystkich omawianych w tej książce przykładów można odszukać na stronach wydawnictwa PWN: http://www.pwn.pl

[2] Urządzenia obliczeniowe OpenCL to przede wszystkim karty graficzne, czyli obliczenia są przeprowadzane za pomocą GPU. Jednakże OpenCL może wykorzystywać także tradycyjne procesory (CPU) oraz układy FPGA, które także oferują wsparcie

w realizacji obliczeń.

[3] Czerwiec, lipiec roku 2014, w przyszłości akceleratorami obliczeń będą również rozwiązania oparte na DSP.

[4] W podobny sposób można także postąpić w podstawowym API dla języka C.

[5] Pakiet (czy też biblioteka) BLAS to zestaw funkcji, pierwotnie opracowany w języku Fortran, do realizacji podstawowych operacji na wartościach skalarnych, wektorowych oraz macierzach. Więcej informacji o pakiecie BLAS można znaleźć pod adresem: http://www.netlib.org/blas/.

5 Przykłady obliczeń w OpenCL

W tym rozdziale przedstawiono różne przykłady obliczeń, jakie można realizować w sposób bardziej efektywny za pomocą technologii OpenCL. Przykłady dotyczą mniej skomplikowanych zagadnień, jak np. obsługa `printf` czy obliczenia związane z przybliżaniem wartości liczby π, czy też operacje redukcji. Podano także bardziej zaawansowane implementacje, jak np. sortowanie bitoniczne. Omówione zostały także problemy pojawiajace się we współpracy pomiędzy standardem OpenCL a np. standardem OpenGL.

Przedstawione zostaną także możliwości przetwarzania obrazów 2D przez wbudowane w OpenCL funkcje, które upraszczają niektóre algorytmy. Przykładem może być realizacja różnego rodzaju filtrów dla obrazów graficznych, np. do wykrywania krawędzi. Podamy także wersję filtru do wykrywania krawędzi bez wykorzystywania dostępnych w OpenCL funkcji do przetwarzania obrazu.

Oto przykłady, które zostaną omówione w tym rozdziale:

• obsługa printf w OpenCL,

• operacja redukcji równoległej,

• wyznaczanie histogramu,

• algorytm szukania „prawie” binarnego,

• przybliżanie wartości liczby π za pomocą metody Monte Carlo,

• symulator gry w życie,

• rysowanie zbioru Mandelbrota i Julii,

• implementacja algorytmu sortowania bitonicznego,

• przetwarzanie obrazów, w tym filtr rozmywający Gaussa, wykrywania krawędzi oraz filtr konwolucyjny,

• współpraca OpenCL z OpenGL.

5.1. Obsługa printf w OpenCL

Tworzenie i testowanie jądra obliczeniowego jest w przypadku OpenCL w pewnym sensie utrudnione, ponieważ urządzenia OpenCL to oddzielne układy obliczeniowe, czasem zupełnie inne niż system gospodarza. Choć istnieją odmiany OpenCL, które działają też na CPU. Wersja standardu 1.2 przyniosła możliwość używania funkcji `printf`, która stanowi bezpośredni odpowiednik funkcji `printf` znanej z ANSI C.

W starszej wersji standardu 1.0 i 1.1 może okazać się, że obsługę `printf` należy uruchomić za pomocą odpowiedniego wyrażenia `#pragma`. W przypadku sterowników OpenCL firmy AMD

wykonamy to w następujący sposób:

```
#pragma OPENCL EXTENSION cl_amd_printf : enable
```

Dla sterownika Intel OpenCL składania polecenia jest taka sama, różni się tylko nazwą rozszerzenia:

```
#pragma OPENCL EXTENSION cl_intel_printf : enable
```

Używanie `printf` w jądrze obliczeniowym jest identyczne jak w języku C:

```
__kernel void test1_kernel( … ) {

    int i = get_global_id(0);

    printf(”ciag znaków z jądra obliczeniowego\n”);

    printf(”kernel id = %d ”, i);

}
```

Najczęściej spotykana jest wersja OpenCL 1.2, w której obsługa `printf` jest dostępna bezpośrednio. Drugim istotnym elementem, jaki należy brać pod uwagę, jest fakt, że wykonywanie wątków nie jest uporządkowane. Przekłada się to na fakt, że wykonanie funkcji `printf`, ale przede wszystkim kolejność wyświetlanych komunikatów, nie mają odgórnie ustalonej kolejności.

Niemniej sposób posługiwania się `printf` jest analogiczny jak w standardzie C99. Elementy poprzedzone symbolem procentu „%” są traktowane jako wyrażenia odpowiedzialne za wykonanie konwersji typu do postaci znakowej. Po tym symbolu wyróżnia się następujące elementy:

• tzw. pole flagi wprowadzające dodatkowe znaki, np. w celu wyrównania do lewej strony lub dopisana dodatkowych symboli,

• minimalna szerokość pola, w którym prezentowana jest informacja,

• opisanie precyzji dla pól liczbowych, zarówno całkowitych, jak też zmiennoprzecinkowych,

• specyfikator opisujący przypadek dla typów wektorowych,

• modyfikator długości pola, w którym prezentowana będzie informacja po konwersji,

• specyfikator konwersji dokonujący interpretacji wartości i realizujący jej zamianę na postać znakową.

Znaki opisujące flagę to symbol „–” oznaczający wyrównanie do lewej strony (domyślnie wyrównanie jest do prawej strony). Symbol „+” oznacza dołączenie symbolu plus lub minus do reprezentacji wartości liczbowej. Znak spacji oznacza zastąpienie znaku pustą spacją. Znak hash „#” oznacza, iż zostanie wyświetlona alternatywna postać wartości, a znak „0” pozwala dodać wiodące zera.

Specyfika typów OpenCL C wymaga jednak wprowadzenia nowych oznaczeń konwersji, więc oprócz doskonale znanych `%d`, `%f`, `%s` dodano nowe, odnoszące się np. do typów wektorowych. Oznaczenie

kodu dla typów wektorowych to `vN`, gdzie `N` przybiera wartości 2, 3, 4, 8, 16, czyli typowe długości wektorów. Użycie specyfikatora lub modyfikatora `vN` oznacza, że wyświetlane wartości zostaną przedstawione w następującej formie:

```
wartość1 S wartość2 S wartość3 S … S wartośćN
```

gdzie: wyrażenia `wartość1`, `wartość2` – kolejne wartości wektora, `S` – symbol separatora, którym domyślnie jest przecinek.

Nowe oznaczenia dotyczą też możliwości stosowania modyfikatora długości pola, inaczej mówiąc, podana wartość zostanie zamieniona i dopasowania do pola charakterystycznego np. dla typu `char`. Zastosowanie „`hh`” oznacza, iż wartość będzie konwertowana do typu `char` lub `uchar`. Pojedyncze „`h`” oznacza konwersję do typu `short` lub `ushort`, a „`l`” konwersję do typu `long` lub `ulong`, Powyższe specyfikatory mogą być stosowane także w połączeniu z typami wektorowymi. Dostępny też jest dodatkowy specyfikator „`hl`” oznaczający wartość `int`, jeśli konwertujemy z typów całkowitych, lub `float`, jeśli z typów zmiennoprzecinkowych.

Właściwe specyfikatory konwersji to „`d`” oraz „`i`”, które zapisują wartość w postaci `[-]dddd`. Inaczej mówiąc, mamy do czynienia z liczbami całkowitymi. Podobnie działają specyfikatory: „`o`”, „`u`”, „`x`”, „`X`”, choć w przypadku „`x`” wartość jest zapisywana w postaci liczb szesnastkowych (`o` – ósemkowych). Konwerter „`f`” oraz „`F`” to liczba zmiennoprzecinkowa zapisana w postaci `[-]ddd.ddd`. Odmiana `printf` dla OpenCL obsługuje naturalnie typy `float`, `double` oraz `half`.

Konwerter „`e`” pozwala na zapis liczby w formacie wykładniczym `[-]d.ddd e±dd`. Podobnie „`g`”, w którym, jeśli wykładnik jest mały, to stosowany jest zapis „`f`”. Obecny jest też specyfikator „`a`”, odpowiedzialny za zapis w formacie `[−]0xh.hhhh p±d` przy użyciu wartości szesnastkowych.

Można naturalnie korzystać też ze specyfikatorów „`c`”, „`s`” oraz „`p`”, czyli wskaźnika, choć tym razem wyświetlana wartość zależy od producenta sterownika. Analogicznie jak w zwykłym `printf` wyrażenie „`%%`” pozwala uzyskać symbol procentu.

Podstawowy przykład wykorzystania `printf` może odnosić się do przypadku, gdy chcemy odczytać identyfikator globalny:

```
int i = get_global_id(0);

printf(”kernel id = %d ”, i);
```

Ze względu na to, że istnieje wiele jednostek roboczych, nie zostanie zachowana kolejność wyświetlania komunikatu przez różne wątki. Mimo to można zrealizować śledzenie potrzebnych wartości. Aby upewnić się co do poprawności implementacji jądra obliczeniowego, należy dodać do wyświetlanych komunikatorów informację o identyfikatorach.

Łatwo wyświetlić zawartość typów wektorowych, choć można to zrobić bezpośrednio po polach dostępowych do każdego elementu wektora, definiując następujące dwa wektory:

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

uchar4 uc4 = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);
```

Pierwsza zmienna `f` zawiera wektor o czterech wartościach zmiennoprzecinkowych typu `float`, a druga ze zmiennych `uc4` to wektor o czterech znakach. Wartości znaków zostały podane jako liczby szesnastkowe.

Użycie `printf` w następujący sposób:

```
printf(”%v4f\n”, f);
```

oznacza wyświetlenie czterech elementów wektora za pomocą specyfikatora `%f`, czyli na ekranie zobaczymy:

```
1.000000,2.000000,3.000000,4.000000
```

Poszczególne wartości są rozdzielone przecinkami. Można też wymusić formatowanie. Jeśli chcemy, aby wartości były wyświetlane w ograniczonym polu o dwóch cyfrach przed przecinkiem i dwóch po przecinku, to należy użyć następującego ciągu formatującego:

```
printf(”f4 = %2.2v4hlf\n”, f);
```

wynik będzie następujący:

```
f4 = 1.00,2.00,3.00,4.00
```

Dodatkowo wartości `float` zostały skonwertowane do typu `long`. Wyświetlenie zawartości zmiennej `uc4` w postaci liczb szesnastkowych wymaga zastosowania specyfikatora „`x`” oraz konwertera „`hh`”, który oznacza, że wartość jest konwertowana do typu `unsigned char`:

```
printf(”uc4 = %#v4hhx\n”, uc4);
```

w odpowiedzi uzyskamy:

```
uc4 = 0xfa,0xfb,0xfc,0xfd
```

Stosowanie[1] `printf` w jądrach obliczeniowych ma sens tylko podczas testów, ponieważ używanie tej konstrukcji wymaga dodatkowych nakładów obliczeniowych, które urządzenie obliczeniowe musi ponieść, aby wykonać konwersję oraz przygotować ciąg znaków do przesłania na urządzenie wyjściowe.

Pewne różnice nie przeszkadzają w uzyskaniu podstawowej funkcjonalności. Jednakże na zakończenie tego rozdziału, w ramach podsumowania wymienione zostaną najważniejsze różnice między OpenCL C oraz tradycyjnym `printf` ze standardu C99:

• Modyfikator „`l`” w połączeniu z specyfikatorem konwersji „`c`” lub „`s`” nie jest obsługiwany przez OpenCL C.

• Modyfikatory długości pola „`ll`”, „`j`”, „`z`”, „`t`”, „`L`” również nie są obsługiwane przez OpenCL C.

• Specyfikator konwersji „`n`[2]” jest zastrzeżony, ale obecnie nie jest obsługiwany przez OpenCL C.

• Specyfikatory konwersji „`f`”, „`F`”, „`e`”, „`E`”, „`g`”, „`G`”, „`a`”, „`A`”, konwertujące argumenty

zmiennoprzecinkowe do wartości typu `double` są dostępne tylko wówczas, gdy istnieje wsparcie dla liczb podwójnej precyzji. Jeśli typ `double` nie jest dostępny, wynikiem będzie liczba typu `float`, tj. liczba o pojedynczej precyzji.

• Modyfikator długości „`l`” jest obsługiwany dla profilu osadzonego (ang. *embedded profile*) tylko wówczas, gdy obsługiwane są liczby 64-bitowe.

• W przypadku implementacji dla OpenCL, wywołanie `printf` zwraca wartość `0`, gdy polecenie `printf` z sukcesem zakończy pracę lub `-1` w przypadku błędu. Jest to inne zachowanie niż w standardzie C99, gdzie wartość ujemna oznacza błąd, a dodatnia liczbę znaków, jaka została wysłana do standardowego wyjścia.

• Operator konwersji „`s`” w przypadku OpenCL C może być stosowany tylko dla argumentów, które są reprezentowane jako ciąg znaków.

Dodatkowe informacje o `printf` można odszukać także w dokumencie referencyjnym opisującym OpenCL i raz jeszcze należy podkreślić, iż `printf` można stosować bez dodatkowych zabiegów na urządzeniach OpenCL zgodnych ze standardem 1.2.

5.2. Operacja redukcji

Operacja redukcji to jedna z podstawowych czynności, jaką się wykonuje w wielu algorytmach równoległych i szeregowych. Prawidłowa implementacja operacji redukcji, szczególnie w środowisku równoległym, ma zasadnicze znaczenie dla wydajności całego algorytmu. Obecnie dla OpenCL można na stronach WWW głównych producentów procesorów czy kart graficznych znaleźć gotowe rozwiązania, które oferują w wielu przypadkach najwyższą wydajność, jaką można osiągnąć za pomocą OpenCL. W tym rozdziale omówimy tylko kilka przypadków implementacji operacji redukcji, które pozwolą zapoznać się z możliwościami OpenCL. Aby zwiększyć wydajność operacji redukcji, należy zastosować pamięć dzieloną, a w dotychczasowych przykładach nie korzystaliśmy z tej możliwości.

5.2.1. Operacja redukcji – wersja szeregowa

Operacja redukcji może charakteryzować się bardzo prostą implementacją – można ją zapisać za pomocą pętli `for`, jeśli np. redukcja to operacja sumy na określonym zbiorze danych. Odpowiedni kod jest następujący:

```
r = 0;

for( i = 0 ; i < NDATA ; i++) {

    r = r + data[i];

}
```

Zakładamy, że zmienna `r` będzie zawierać końcowy wynik, ilość danych została określona w zmiennej `NDATA`. Dane czerpiemy z tablicy `data`. Powyższy kod, składający się z czterech linii, stanowi kompletną operację redukcji, w przypadku, gdy chcemy obliczyć sumę wszystkich elementów tablicy.

Do implementacji możemy także wykorzystać funkcje pochodzące z biblioteki standardowej języka C++. Dane w tym przypadku zostaną zapisane do wektora STL:

```
vector<int> vdata(NDATA);
```

Operacja redukcji przy wykorzystaniu funkcji `accumulate` sprowadza się do następującej funkcji

```
int make_reduction(const vector<int>& source) {

    return accumulate(source.begin(), source.end(), 0);

}
```

Argumenty `accumulate` to początek wektora, jego koniec oraz wartość początkowa, do której będą dodawane kolejne elementy ze źródła danych, tj. z wektora `source`.

W podanym przykładzie warto jeszcze zauważyć jedną bardzo przydatną własność, proces redukcji można podzielić np. na dwie części. Można wyznaczyć sumę elementów np. lewej części tablicy `data` czy też wektora `source`, a następnie prawej części badanego zbioru danych. Otrzymane sumy częściowe wystarczy dodać, aby otrzymać poprawną wartość końcową.

Własność ta sugeruje, iż w kontekście środowiska równoległego obliczanie wartości sumy częściowej dla lewej i prawej części tablicy można wykonywać niezależnie od siebie, co oznacza, że istnieje możliwość zrealizowania wersji równoległej tego algorytmu.

5.2.2. Operacja redukcji – jądro obliczeniowe podejście pierwsze

Listing 5.1 przedstawia pierwszą postać jądra obliczeniowego realizującego operację redukcji. Dla ułatwienia zakładamy, że realizujemy tylko sumę elementów tablicy wejściowej. Typem pojedynczego elementu jest `float`. Choć podajemy kilka wersji jądra obliczeniowego, to za każdym razem nagłówek funkcji jądra jest taki sam:

```
__kernel void add_reduction(__global float *in_data,

    __global float *out_data,

    unsinged int N,

    __local float* shared_data);
```

Pierwszy argument o nazwie `in_data` to dane wejściowe, będą one reprezentowane jako tablica. Drugi argument `out_data` zgodnie z nazwą będzie zawierał rezultat operacji redukcji. Od razu należy zwrócić uwagę, iż nie zawsze będzie to jedna liczba stanowiąca ostateczny wynik. W pierwszym przykładzie otrzymamy kilka wartości, które należy do siebie dodać, aby otrzymać ostateczną sumę. W trzecim parametrze o nazwie `N` podajemy rozmiar (inaczej liczbę elementów) tablicy wejściowej. W ostatnim argumencie, ze względu na wymagania OpenCL, musimy podać wcześniej przygotowany wskaźnik do pamięci dzielonej. Jest to parametr charakterystyczny dla OpenCL, a sposób przydzielania takiej pamięci zaprezentowano w dalszej części tego rozdziału.

Procedura obliczeniowa, jaką omawiamy w tym rozdziale, może zostać podzielona na cztery główne części:

• część pierwsza – odczytanie wartości identyfikatora,

• część druga – wypełnienie pamięci dzielonej,

• część trzecia – wykonanie częściowej redukcji na pamięci dzielonej,

• część czwarta – zapisanie częściowego wyniku w tablicy wyjściowej.

Będziemy sumować 512 elementów i do tej wartości dobierzemy odpowiednio rozmiary tablic umieszczonych w pamięci dzielonej oraz liczbę grup roboczych.

Czynności wykonywane w procedurze redukcyjnej rozpoczyna odczytanie identyfikatora lokalnego oraz globalnego:

```
unsigned int lid = get_local_id(0);

unsigned int gid = get_global_id(0);
```

Identyfikator globalny będzie różny dla każdej jednostki roboczej, można go przyrównać też do indeksu tablicy wejściowej. Jednak identyfikator lokalny jest unikatowy tylko w ramach jednej grupy roboczej.

Pierwsza część procedury obliczeniowej polega na tym, iż element wskazany przez identyfikator globalny jest przenoszony do zmiennej dzielonej `shared_data` pod indeks wskazany przez identyfikator lokalny grupy. Ponieważ mamy wiele grup roboczych, a każda z nich ma swój własny obszar, to w ramach całej grupy pewien zestaw elementów z `in_data` zostanie przeniesiony do wspólnej pamięci grupy roboczej:

```
shared_data[lid] = (gid < N) ? in_data[ gid ] : 0;
```

Należy pamiętać, iż wiele jednostek roboczych wykonuje tę operację, jednak zawsze odczytują one inny element z tablicy `in_data`, i w ramach każdej z grup zapisują do innego fragmentu tablicy `shared_data`.

Aby jednak dane pozostały spójne, należy wstawić tzw. barierę w następujący sposób:

```
barrier(CLK_LOCAL_MEM_FENCE);
```

Powyższa instrukcja w ramach całej grupy nie pozwoli na to, aby poszczególne jednostki robocze rozpoczęły drugi etap redukcji, jeśli inne jednostki wykonują jeszcze operację kopiowania danych do pamięci dzielonej. Postawiona bariera działa tylko w ramach grupy. Różne grupy mogą przejść do kolejnego etapu szybciej lub też wolnej od innych grup.

Etap trzeci jest reprezentowany przez pętlę `for` – w pamięci dzielonej wykonujemy operację redukcji. Wielkość pamięci dzielonej musi zostać poprawnie określona przed wywołaniem jądra, gdyż wartość odczytywana przez funkcję `get_local_size` będzie wskazywać wartość o połowę mniejszą od wielkości tablicy w pamięci roboczej. Sam proces obliczenia sumy nie jest zaskakujący:

```
shared_data[lid] = shared_data[lid] + shared_data[lid + local_size];
```

Indeks `lid` wskazuje element, do którego dodamy element wskazany przez wartość `lid + local_size`, a rezultat ponownie zapiszemy pod indeksem `lid`. Zapis w pamięci dzielonej również wymaga synchronizacji za pomocą bariery.

Kluczowym elementem jest wyznaczanie wartości poszczególnych indeksów. Najpierw należy się upewnić, czy aktualna wartość indeksu docelowego `lid` jest podzielona przez podwojoną wartość `local_size`. Sposób konstrukcji pętli `for` spowoduje, iż wartości `lid` będą liczbami całkowitymi, np. od zera do ośmiu dla założonej wielkości 512 elementów. Należy też zaznaczyć, iż elementy generowane przez pętlę `for` to kolejne potęgi liczby 2.

Aby ostatecznie wyjaśnić ten proces, warto dla jednej grypy roboczej pokazać, jak wygladają wartości zmiennej `local_size` oraz `LID` w ramach jednej grupy roboczej (`sd` to skrót od `shared_data`):

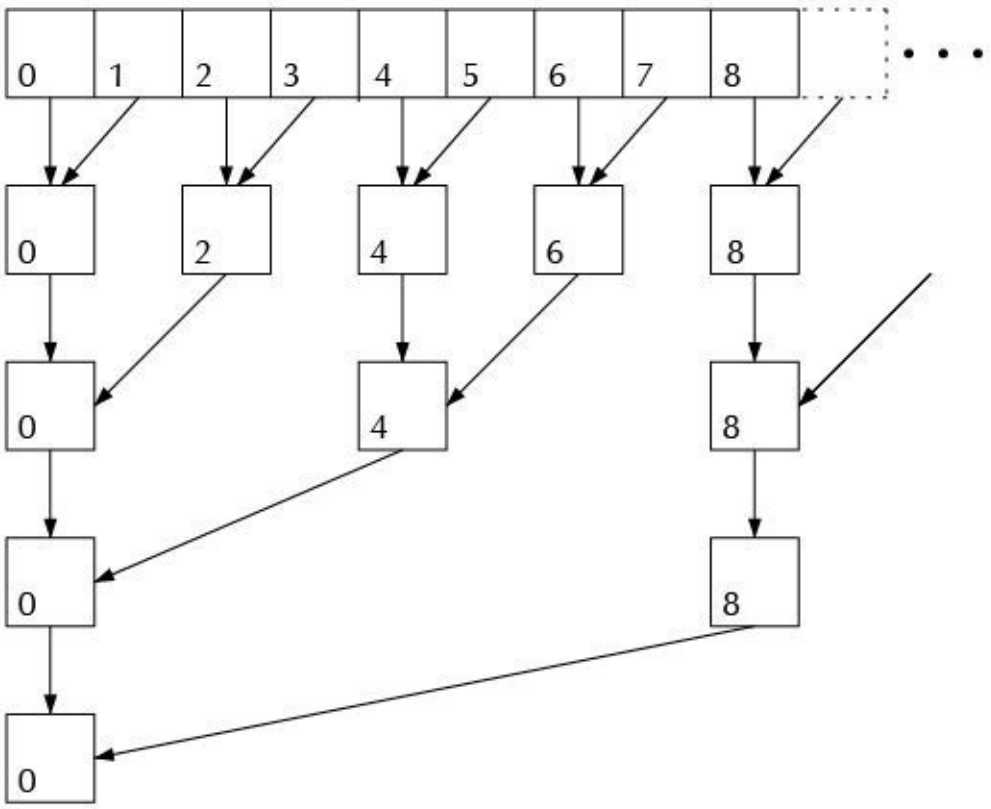
```
local_size=1, LID= 0, sd[0]=sd[0]+sd[1]

local_size=1, LID= 2, sd[2]=sd[2]+sd[3]

local_size=1, LID= 4, sd[4]=sd[4]+sd[5]

local_size=1, LID= 6, sd[6]=sd[6]+sd[7]

local_size=1, LID= 8, sd[8]=sd[8]+sd[9]

local_size=1, LID=10, sd[10]=sd[10]+sd[11]

local_size=1, LID=12, sd[12]=sd[12]+sd[13]

local_size=1, LID=14, sd[14]=sd[14]+sd[15]

local_size=2, LID= 0, sd[0]=sd[0]+sd[2]

local_size=2, LID= 4, sd[4]=sd[4]+sd[6]

local_size=2, LID= 8, sd[8]=sd[8]+sd[10]

local_size=2, LID=12, sd[12]=sd[12]+sd[14]

local_size=4, LID= 0, sd[0]=sd[0]+sd[4]

local_size=4, LID= 8, sd[8]=sd[8]+sd[12]

local_size=8, LID= 0, sd[0]=sd[0]+sd[8]
```

Należy pamiętać, że w ramach grupy roboczej pętla `for` jest wykonywana przez wszystkie jednostki robocze. Oznacza to, że dla `local_size=1` osiem wątków wykona operacje sumowania, dla `local_size=2` już tylko cztery, następnie dwa i na końcu – dla `local_size=8` – tylko jeden z wątków wykona sumę na wskazanych elementach. Proces ten pokazano także na rysunku 5.1, gdzie ostateczna suma zostanie wstawiona pod indeks 0.

Wykonanie redukcji w pamięci dzielonej doprowadza do tego, że pod zerowym indeksem zmiennej

shared_data mamy obliczoną aktualną wartość sumy, ale tylko dla jednego fragmentu. Niemniej, ostatnia czynność wykonywana tylko przez jedną jednostkę roboczą, np. o indeksie lokalnym 0, przepisuje wyznaczoną wartość redukcji do tablicy wyjściowej pod indeks grupy:

```
if ( lid == 0 )

    out_data[get_group_id(0)] = shared_data[0];
```



Rys. 5.1. Schemat operacji redukcji wykonywanej w ramach grupy roboczej przez jądro obliczeniowe z listingu 5.1 (wartości w dolnym rogu to indeks danego elementu)

Na tym kończy się wykonywanie jądra obliczeniowego, a po stronie gospodarza należy jeszcze obliczyć sumę elementów z tablicy out_data, aby otrzymać ostateczny wynik.

Z przedstawionego opisu wynika też sposób, w jaki należy rozdzielić operację redukcji na poszczególne grupy robocze. Dla tablicy wejściowej o 512 elementach, wymiar tablicy dzielonej to np. 16 elementów. Każda grupa robocza wykona 16 operacji sumy na poszczególnych elementach, powstaje zatem pytanie: ile trzeba utworzyć grup roboczych, aby poprawnie zrealizować redukcję? Jeśli redukujemy 512 elementów, to należy utworzyć 512 / 16, tj. 32 grupy robocze.

Listing 5.1. Procedura obliczeniowa OpenCL wykonująca operację redukcji – podejście bezpośrednie (rozwiązanie analogiczne do przykładu zawartego w NVIDIA OpenCL SDK)

```
__kernel void add_reduction(__global float *in_data, __global float *out_data, unsigned int N, __local float* shared_data) {

    unsigned int lid = get_local_id(0);

    unsigned int gid = get_global_id(0);
```

```
    shared_data[lid] = (gid < N) ? in_data[ gid ] : 0;

    barrier(CLK_LOCAL_MEM_FENCE);

    unsigned int local_size;

    for (local_size=1; local_size < get_local_size(0); local_size *= 2)

    {

        if ( (lid % (2*local_size)) == 0 )

        {

            shared_data[lid] = shared_data[lid] + shared_data[lid + local_size];

        }

        barrier(CLK_LOCAL_MEM_FENCE);

    }

    if ( lid == 0 )

        out_data[get_group_id(0)] = shared_data[0];

}
```

Omówiona procedura odzwierciedla założenie co do realizacji operacji redukcji, jednak nie robi tego w najlepszy sposób. Jednym z powodów jest brak uporządkowania operacji na pamięci dzielonej. Sposób, jak to naprawić, omówiono w rozdz. 5.2.3.

5.2.3. Operacja redukcji – jądro obliczeniowe podejście drugie

Redukcja przedstawiona w poprzednim przykładzie nie jest najlepszym rozwiązaniem, m.in. ze względu na sposób, w jaki jest ona przeprowadzana. Sprawdzanie, czy wartość LID ma poprawną wartość z użyciem reszty z dzielenia zajmuje cenny czas, a problem ten można łatwiej rozwiązać, inaczej generując wartości opisujące indeksy elementów do sumowania.

Listing 5.2 przedstawia nową wersję procedury obliczeniowej, która okazała się też nieco krótsza od poprzedniej. Początek procedury jest naturalnie identyczny – należy odczytać fragment danych w pamięci lokalnej, a następnie można przystąpić do przeprowadzenia operacji redukcji w pamięci lokalnej (dzielonej).

W pierwszej kolejności definiuje się zmienną `step`:

```
unsigned int step;
```

Będzie ona stosowana do wyznaczenia odstępu między sumowanymi elementami. Inaczej mówiąc, będziemy sumować np. elementy o indeksie 0 oraz 8, bo wartość odstępu jest równa 8. Wyznaczanie poszczególnych wartości odstępu to zadanie dla pętli `for`, którą tu dla wygody raz jeszcze przytoczymy:

```
for(step=get_local_size(0)/2; step>0; step>>=1)

{

    if (lid < step)

    {

        shared_data[lid] += shared_data[lid + step];

    }

    barrier(CLK_LOCAL_MEM_FENCE);

}
```

Początkowa wartość kroku jest równa połowie wielkości grupy roboczej. Jest to warunek niezbędny, bo każda większa wartość zmiennej `step` będzie powodować przekroczenie zakresu zmiennej lokalnej, nie musi to od razu oznaczać przerwania obliczeń, ale wynik końcowy może być całkowicie błędny.

Pamiętać należy, że w grupie roboczej wszystkie jednostki robocze wykonują tę samą pętlę równolegle, ale suma lokalnego identyfikatora roboczego oraz wartości skoku zawsze będzie dawać unikatowe indeksy, a dodatkowa bariera zapewniać spójność odczytu i zapisu danych. W pętli wyrażenie: `step>>=1` to przesunięcie bitowe w lewo o jedną pozycję, które realizuje dzielenie przez dwa, co oznacza też, że wartość pamięci lokalnej, gdzie wykonuje się redukcję, musi być obowiązkowo podzielna przez dwa. Za każdą iteracją pętli krok jest coraz mniejszy i dodatkowa instrukcja sprawdzająca wartość `LID` (a dokładnie czy indeks lokalny jest mniejszy niż krok) powoduje, że wykonywać będziemy coraz mniej operacji sumowania.

Podobnie jak w poprzednim przypadku, można pokazać operacje sumowania w ramach jednej grupy roboczej:

```
step=8, LID=0, sd[0]=sd[0]+sd[8]

step=8, LID=1, sd[1]=sd[1]+sd[9]

step=8, LID=2, sd[2]=sd[2]+sd[10]

step=8, LID=3, sd[3]=sd[3]+sd[11]

step=8, LID=4, sd[4]=sd[4]+sd[12]

step=8, LID=5, sd[5]=sd[5]+sd[13]

step=8, LID=6, sd[6]=sd[6]+sd[14]

step=8, LID=7, sd[7]=sd[7]+sd[15]

step=4, LID=0, sd[0]=sd[0]+sd[4]

step=4, LID=1, sd[1]=sd[1]+sd[5]

step=4, LID=2, sd[2]=sd[2]+sd[6]
```

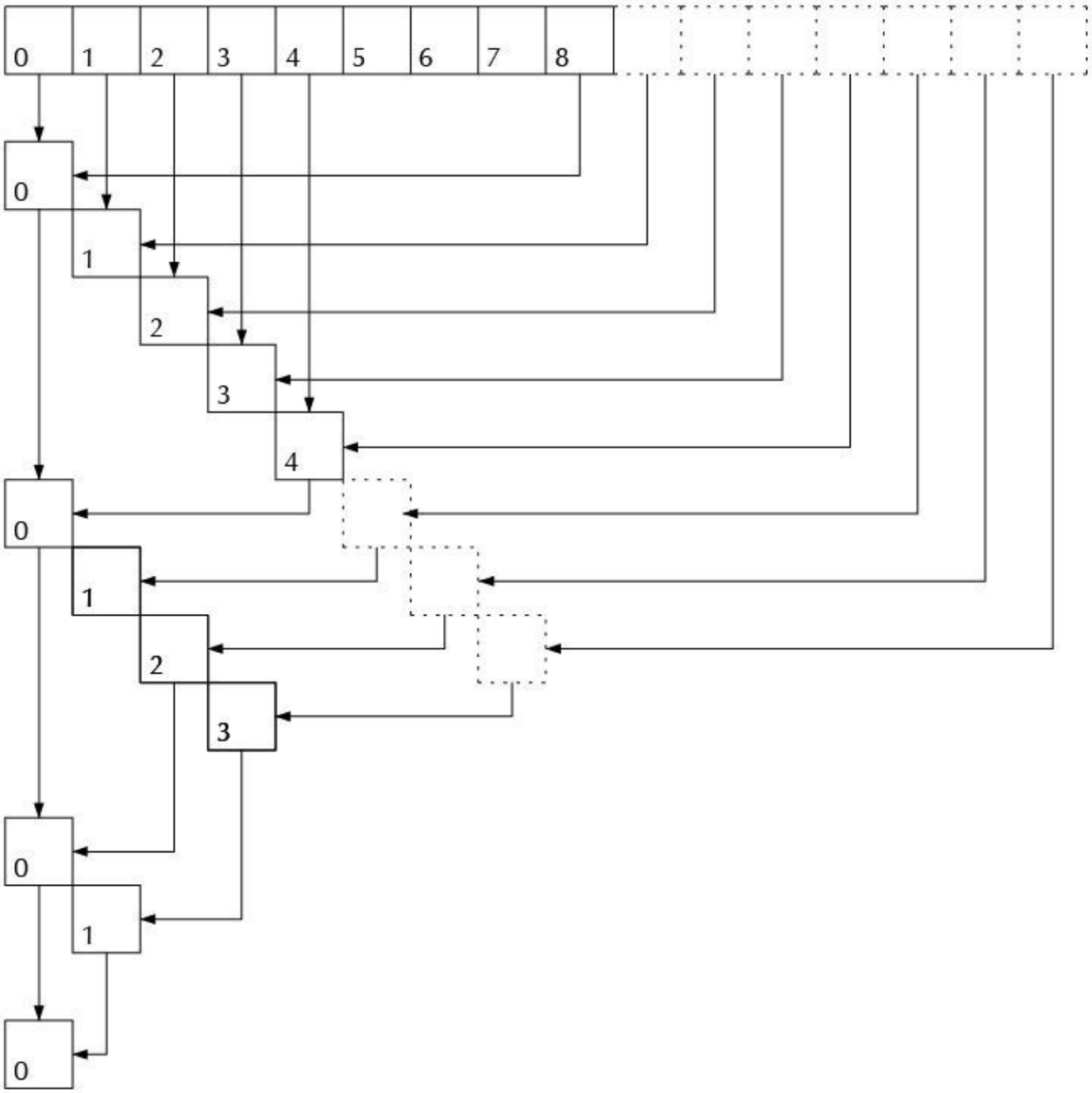
```
step=4, LID=3, sd[3]=sd[3]+sd[7]

step=2, LID=0, sd[0]=sd[0]+sd[2]

step=2, LID=1, sd[1]=sd[1]+sd[3]

step=1, LID=0, sd[0]=sd[0]+sd[1]
```

Jak widać, tym razem operacje są zorganizowane ponownie w drzewo o wysokości logarytmicznej (logarytmicznie zależnej od długości tablicy), co dodatkowo zostało pokazane na rysunku 5.2. Dla kroku największego (w przykładzie wartość 8), wykonujemy osiem sum pomocniczych i otrzymujemy osiem wyników umieszczonych w indeksach od 0 do 7. Następna grupa operacji (z krokiem 4) wykona cztery sumowania. Ostania operacja to dodanie indeksów 0 i 1. W ten sposób, pod adresem zero otrzymano sumę dla całej grupy roboczej. Ostatnia grupa, analogicznie jak w poprzednim przykładzie, zapisuje tę sumę w pamięci globalnej pod indeksem grupy roboczej. Operacje na pamięci są tym razem inaczej uporządkowane i odczyt oraz zapis są realizowane w coraz mniejszej liczbie sąsiadujących ze sobą komórek. Poprzednie podejście w każdym etapie powiększało odstęp między zapisywanymi komórkami.

Listing 5.2. Poprawiona procedura obliczeniowa OpenCL wykonująca operację redukcji – operacje na pamięci dzielonej zostały uporządkowane

```
__kernel void add_reduction_better(__global float *in_data, __global float *out_data,
unsigned int N, __local float*

                                                                    shared_data) {

    unsigned int lid = get_local_id(0);

    unsigned int gid = get_global_id(0);

    shared_data[lid] = (gid < N) ? in_data[gid] : 0;

    barrier(CLK_LOCAL_MEM_FENCE);

    unsigned int step;

    for(step=get_local_size(0)/2; step>0; step>>=1)

    {

        if (lid < step)

        {

            shared_data[lid] += shared_data[lid + step];

        }

        barrier(CLK_LOCAL_MEM_FENCE);

    }

    if (lid == 0)
```

```
        out_data[get_group_id(0)] = shared_data[0];

}
```



Rys. 5.2. Schemat operacji redukcji wykonywanej w ramach grupy roboczej przez jądro obliczeniowe z listingu 5.2 (wartość w dolnym rogu to indeks danego elementu)

Ważną kwestią jest też dobór liczby grup roboczych, a także rozmiaru danych wejściowych, dla których wykonujemy redukcję. Dwie powyższe procedury obliczeniowe, a także następne, wymagają, aby wielkości danych wejściowych były potęgą liczby dwa. Determinuje to także podział na grupy robocze, np. jeśli chcemy wykonać redukcję dla 512 elementów, przy zastosowaniu 16 grup roboczych, to całkowita liczba jednostek roboczych powinna być równa 32 * 16, gdzie 16 to rozmiar grupy roboczej. Iloczyn ten jest równy 512. Skoro więc rozmiar grupy roboczej wynosi 16, wartość 512 / 16 będzie równa liczbie grup roboczych. Inaczej mówiąc, przedstawione jądra obliczeniowe wykonają redukcję za pomocą 32 grup roboczych. Obliczenia te są ważne również dlatego, że

pozwalają określić rozmiary parametrów jądra obliczeniowego. Zmienna `out_data` powinna mieć rozmiar nie mniejszy niż 32, a wielkość pamięci dzielonej jest równa wielkości grupy roboczej – w naszym przypadku ma 16 elementów.

Można sobie zadać pytanie, czy można w jakiś łatwy sposób przyspieszyć realizację redukcji. Odpowiedź jest twierdząca – w łatwy sposób możemy zmniejszyć liczbę operacji na danych, stosując typ wektorowy. W poprzednich dwóch przykładach stosowaliśmy wartości skalarne, ale wykorzystując OpenCL C i typ `float4`, możemy w dość prosty sposób przekształcić jądro obliczeniowe z listingu 5.2 i dostosować je do pracy z typem wektorowym, co zostało pokazane w listingu 5.3.

Na pierwszy rzut oka zmian nie widać, dotyczą one bowiem tylko typów parametrów jądra obliczeniowego. Pozostałe czynności pozostają niezmienione. Nowa wersja z listingu 5.3 wymaga uwagi przy wywoływaniu jądra – choć ilość przydzielonej pamięci musi być taka sama, jak w poprzednich przykładach, to parametr `N` musi zostać podzielony przez 4, gdyż jedna porcja informacji pobrana np. z `in_data` zawiera 4 liczby. Tablicę `in_data` należy zatem indeksować wartościami od zera do `N/4`, gdzie `N` to całkowita ilość liczb, na których wykonujemy redukcję.

Listing 5.3. Wersja listingu 5.2, w której zamiast tablicy zastosowano typ wektorowy – istotna zmiana dotyczy tylko typów danych w parametrach jądra obliczeniowego

```
__kernel void add_reduction_float4(__global float4 *in_data, __global float4 *out_data, unsigned int N, __local float4*
                                                                                                                shared_data)
{
    unsigned int lid = get_local_id(0);
    unsigned int gid = get_global_id(0);
    shared_data[lid] = (gid < N) ? in_data[gid] : 0;
    barrier(CLK_LOCAL_MEM_FENCE);
    unsigned int step;
    for(unsigned int step=get_local_size(0)/2; step>0; step>>=1)
    {
        if (lid < step)
        {
            shared_data[lid] += shared_data[lid + step];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
```

```
    if (lid == 0)

        out_data[get_group_id(0)] = shared_data[0];

}
```

5.2.4. Operacja redukcji – jądro obliczeniowe poddane optymalizacji

Kończąc omawianie operacji redukcji, zastanowimy się ponownie nad zwiększeniem wydajności. Dotyczy to redukcji realizowanej przy użyciu pamięci dzielonej. W poprzednich odsłonach procedury obliczeniowej była stosowana pętla `for` do wyznaczania indeksów elementów, jakie należy do siebie dodać. Zamiast stosowania kosztownej obliczeniowo pętli `for` można jawnie podać wszystkie możliwe indeksy, jakie trzeba wyznaczyć. Inaczej mówiąc, należy rozwinąć pętlę, aby uzyskać dodatkowe przyspieszenie.

Rozwinięcie pętli należy poprzedzić następującą istotną obserwacją – w jądrze obliczeniowym z listingu 5.2 wykorzystaliśmy wartość `step`, aby uporządkowywać operacje na pamięci. Wykorzystaliśmy pętlę `for`, aby rozproszyć operację sumy i uzyskać jak najwięcej równoległych operacji sumowania. Należy zwrócić uwagę, że w pierwszej grupie operacji dla największego kroku:

```
step=8, LID=0, sd[0]=sd[0]+sd[8]

step=8, LID=1, sd[1]=sd[1]+sd[9]

step=8, LID=2, sd[2]=sd[2]+sd[10]

step=8, LID=3, sd[3]=sd[3]+sd[11]

step=8, LID=4, sd[4]=sd[4]+sd[12]

step=8, LID=5, sd[5]=sd[5]+sd[13]

step=8, LID=6, sd[6]=sd[6]+sd[14]

step=8, LID=7, sd[7]=sd[7]+sd[15]
```

wszystkie operacje sumy można wykonać bez kłopotów, nie występują bowiem żadne kolizje między elementami. Jednak nie można przejść do następnej grupy operacji, gdyż pierwsze cztery elementy zapamiętują wyniki pod indeksami od 0 do 3, które są używane w następnej grupie elementów:

```
step=4, LID=0, sd[0]=sd[0]+sd[4]

step=4, LID=1, sd[1]=sd[1]+sd[5]

step=4, LID=2, sd[2]=sd[2]+sd[6]

step=4, LID=3, sd[3]=sd[3]+sd[7]
```

Podobnie w grupie trzeciej:

```
step=2, LID=0, sd[0]=sd[0]+sd[2]

step=2, LID=1, sd[1]=sd[1]+sd[3]
```

a także w czwartej, choć dla rozmiaru grupy roboczej 16 mamy już tylko jedną operację:

```
step=1, LID=0, sd[0]=sd[0]+sd[1]
```

Również i tu są indeksy kolidujące z wartościami z poprzednich grup. Ponieważ za każdym razem używany jest indeks zerowy i wyższe, to problem kolizji wymaga zastosowania bariery, aby utrzymać spójność danych. Rozwinięcie pętli polega na tym, że należy podzielić operacje sumowania na dwie grupy. W pierwszej wykonywana będzie suma, po której będzie stawiana bariera, a następnie druga grupa wykona sumy, które nie powodują kolizji indeksów. W ostatecznym rozrachunku w indeksie zero znajdzie się wartość sumy wyznaczonej w ramach grupy roboczej. Całość tych uwag można zamienić na kod przedstawiony w listingu 5.4. Bliźniacze rozwiązanie znajduje się też w przykładach firmy NVIDIA. Warto także sprawdzić, jak operację redukcji zaimplementowano w przykładach firm AMD oraz Apple.

Listing 5.4. Procedura obliczająca redukcję w wersji poddanej optymalizacji wraz z typem wektorowym

```
__kernel void add_reduction_unroll(__global float4 *in_data, __global float4 *out_data, unsigned int N, __local volatile float4* shared_data)

{

    unsigned int lid = get_local_id(0);

    unsigned int idx = get_group_id(0)*(get_local_size(0)*2) + get_local_id(0);

    shared_data[lid] = (idx < N) ? in_data[idx] : 0;

    if (idx + __DWSize__ < N)

        shared_data[lid] += in_data[idx + __DWSize__];

    barrier(CLK_LOCAL_MEM_FENCE);

     if (__DWSize__ >= 512) { if (lid < 256) { shared_data[lid] += shared_data[lid + 256]; } barrier(CLK_LOCAL_MEM_FENCE); }

     if (__DWSize__ >= 256) { if (lid < 128) { shared_data[lid] += shared_data[lid + 128]; } barrier(CLK_LOCAL_MEM_FENCE); }

     if (__DWSize__ >= 128) { if (lid < 64) { shared_data[lid] += shared_data[lid + 64]; } barrier(CLK_LOCAL_MEM_FENCE); }

    if (lid < 32)

    {

        if (__DWSize__ >= 64) { shared_data[lid] += shared_data[lid + 32]; }

        if (__DWSize__ >= 32) { shared_data[lid] += shared_data[lid + 16]; }

        if (__DWSize__ >= 16) { shared_data[lid] += shared_data[lid + 8]; }

        if (__DWSize__ >= 8) { shared_data[lid] += shared_data[lid + 4]; }

        if (__DWSize__ >= 4) { shared_data[lid] += shared_data[lid + 2]; }
```

```
        if (__DWSize__ >= 2) { shared_data[lid] += shared_data[lid + 1]; }

    }

    if (lid == 0)

        out_data[get_group_id(0)] = shared_data[0];

}
```

5.3. Wyznaczanie histogramu

Wyznaczanie histogramu, czyli częstości występowania poszczególnych typów danych w zbiorze danych, to również zadanie, które względnie łatwo daje się implementować w środowisku OpenCL. Co ważne, możemy osiągnąć znaczne przyspieszenie wykonywania tego zadania względem implementacji sekwencyjnej wykonywanej na pojedynczym procesorze.

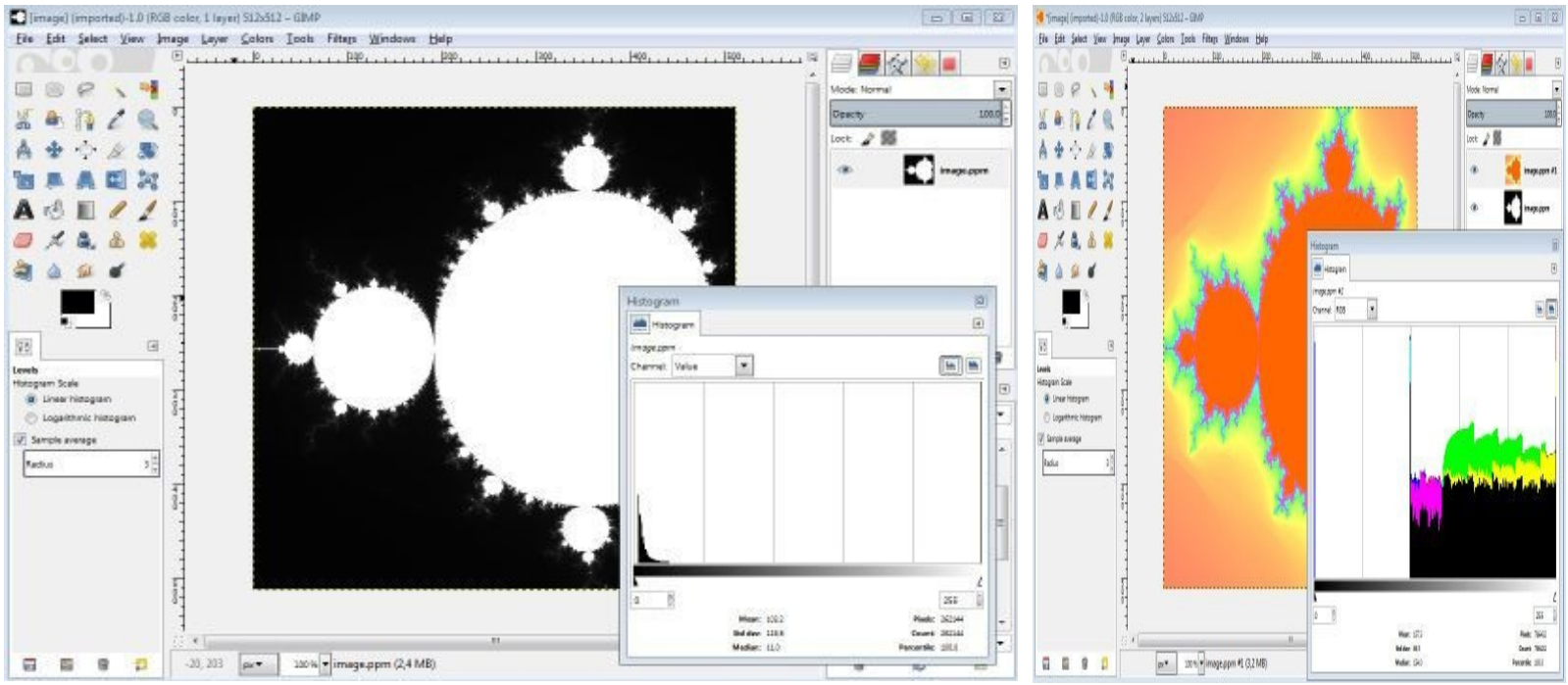
Samo pojęcie histogramu łatwo pokazać na prostym przykładzie, zakładając że mamy następujący zbiór danych:

0, 1, 2, 6, 1, 0, 2, 3, 4, 2, 1, 8, 9

Przygotowanie histogramu polega na określeniu, ile razy występują poszczególne wartości. W powyższym przykładzie zero powtarza się dwa razy, jedynka trzy razy, ale dziewiątka tyko raz. Bardzo często jednak histogram przygotowuje się w postaci rysunku. Na przykład histogram na rys. 5.3 został przygotowany przez program GIMP dla obrazu fraktalnego wygenerowanego w rozdziale 5.7.3. Nasz przykład ograniczymy jednak tylko do wyznaczania histogramu z ciągu liczb, którego wersja szeregowa jest dość prosta do napisania, ale implementacja w OpenCL wymaga włożenia pewnego wysiłku, aby była zgodna z architekturą OpenCL.

5.3.1. Wersja szeregowa

Za nim zaczniemy omawiać implementację równoległą, postąpimy w sposób podobny do poprzedniego przykładu dotyczącego redukcji i przedstawimy implementację szeregową. Histogram, dla uproszczenia, będziemy wykonywać na liczbach całkowitych. Zarówno w odmianie szeregowej, jak i równoległej.



Rys. 5.3. Histogramy dla obrazu fraktalnego, jaki generujemy w rozdz. 5.7 – wersja dla obrazu w szarościach oraz kolorach RGB

Zakładamy, że mamy tablicę elementów typu `int` o rozmiarze `NDATA`:

```
int tab[NDATA];
```

i wskaźnik, który wskazuje na tę tablicę:

```
int *p;
```

Wykonanie histogramu będzie wymagać dodatkowej tablicy o rozmiarze dopasowanym do zbioru danych, jaki będziemy analizować. Założymy, że będziemy analizować elementy z tablicy `tab` lub ze wskaźnika `p`, a wartości, jakie będziemy otrzymywać, będą się znajdować w zakresie od 0 do 15. Histogram będzie powstawał w innej tablicy, np. zadeklarowanej w następujący sposób:

```
int hist[16];
```

lub ponownie możemy użyć tylko wskaźnika:

```
int *phist;
```

Przed rozpoczęciem budowy histogramu należy w przypadku wskaźników przydzielić pamięć:

```
data = new int[NDATA];

hist = new int[DATA_MAX_VALUE];
```

Wystarczy tylko, aby otrzymane wartości potraktować jako indeks do innej tablicy, np. otrzymaną wartość pięć pod indeksem pięć w innej tablicy zwiększamy o jeden. Podobnie postępujemy z kolejnymi odczytanymi elementami, używamy ich jako indeksów do tablicy, w której budujemy histogram. Po przejrzeniu całej tablicy `data` będziemy dysponować już pełnymi informacjami o częstości pojawiania się badanego zbioru danych.

Utworzenie histogramu sprowadza się dosłownie do jednej pętli `for`:

```
int i;

for( i = 0 ; i < NDATA ; i++) {

    hist[data[i]] = hist[data[i]] + 1;

}
```

Pętla realizuje dokładnie algorytmy, które podaliśmy w akapicie powyżej. Wyznaczenie histogramu w najprostszej wersji jest, jak widać, niezwykle proste do realizacji, podobnie jak operacja redukcji.

Histogram ma jedną bardzo przydatną cechę, która nie jest widoczna w pętli podanej powyżej. Jeśli jeden zbiór danych, dla którego wyznaczamy histogram, zostanie podzielony np. na dwie części, to powstaną dwa histogramy, jednak ze względu na to, że na oba zbiory danych współdzielą ten sam typ wartości, można dodać do siebie dwa histogramy, aby otrzymać wynikowy histogram dla całości zbioru. Oznacza to, że możemy badany zbiór danych podzielić na wiele części, a dla każdej z nich przygotować oddzielny histogram. Następnie dodajemy do siebie histogramy (lub też nakładamy na siebie histogramy), tak aby otrzymać ostateczny histogram dla całego zbioru danych.

5.3.2. Wersja dla OpenCL

Implementacja OpenCL niestety nie jest już tak elementarnie prosta, jak w przypadku procedury szeregowej. Listingi 5.5 oraz 5.6 przedstawiają cały kod źródłowy dwóch procedur, które realizują wyznaczanie histogramu. Zakładamy, że obie procedury otrzymują dane w tablicy `in_data`, ale wartości tych danych znajdują się w zakresie od 0 do 255. Względnie łatwo przerobić listingi napisane dla jednowymiarowych tablic i operujące na liczbach od 0 do 255 na programy wyznaczające histogramy dla obrazów, czyli tablic dwuwymiarowych, których każda „komórka” składa się z trzech wartości od 0 do 255 (inaczej trójka RGB).

Listing 5.5 przedstawia główną procedurę obliczeniową do wyznaczania histogramu częściowego, bowiem w zależności od wymiarów siatki roboczej, powstanie kilkanaście histogramów częściowych, które dopiero po połączeniu pozwolą uzyskać ostateczną postać histogramu badanego zbioru danych.

Histogramy częściowe można dodać na poziomie systemu gospodarza lub dość łatwo zbudować jądro obliczeniowe (listing 5.6), które wykona tę operację za pomocą OpenCL. W ten sposób oszczędzamy nieco czasu na kopiowaniu danych, a także (jeśli liczba histogramów częściowych jest dość duża) nadal wykorzystujemy moc obliczeniową oferowaną przez OpenCL.

Pierwszym elementem procedury obliczeniowej w listingu 5.5 jest włączenie obsługi operacji atomowych (rozdz. 3.7), które zapewnią spójność operacji na pamięci. Nastepnie odczytujemy potrzebne identyfikatory oraz czyścimy lokalny histogram zerami:

```
for( j=256 ; j > 0 ; j-=local_size, indx+=local_size)

{

    if (LID < j)
```

```
        tmp_histogram[indx+LID] = 0;

}
```

Wykorzystujemy identyfikator lokalny, aby zera były wstawione bez kolizcji przy zapisie do pamięci w ramach całej grupy roboczej. Następny element to bariera, bowiem wszystkie jednostki robocze w grupie muszą wpisać zera, aby można było przejść do kolejnego etapu.

Następnie cały zbiór danych dzielimy na grupy równe rozmiarowi siatki lokalnej. Jeżeli obliczamy histogram dla 512 elementów, to możemy założyć, że siatka lokalna będzie obliczać histogram dla 128 elementów, co od razu pozwala wyznaczyć liczbę histogramów częściowych. W tym przypadku powstaną cztery histogramy częściowe. Każda z czterech grup roboczych będzie obliczać swój histogram na wydzielonej grupie danych. Histogram ten jest reprezentowany przez zmienną `tmp_histogram`.

Pierwsza część tego zadania jest realizowana za pomocą następującego fragmentu kodu:

```
if (GID < N)

{

    uint value = in_data[GID];

    atom_inc(&tmp_histogram[value]);

}

barrier(CLK_LOCAL_MEM_FENCE);
```

Wszystkie jednostki robocze w danej grupie uaktualniają wartość histogramu lokalnego za pomocą funkcji atomowej `atom_inc`. Zastosowanie funkcji atomowej jest niezbędne w przypadku, gdy pojawią się dwie takie same wartości w zbiorze danych, aby poprawnie zwiększyć wartość w budowanym histogramie.

Ostatnim elementem jest przeniesienie histogramu częściowego do pamięci globalnej, gdzie składowane są histogramy częściowe. Tym zadaniem zajmuje się ostatnia pętla `for`. Kod tej pętli jest bardzo podobny do pierwszej pętli `for`, gdzie lokalny histogram był wypełniany zerami, ale tym razem uaktualniany jest histogram częściowy znajdujący się w pamięci globalnej.

Listing 5.5. Procedura obliczeniowa obliczająca histogramy częściowe – liczba histogramów jest wyznaczana przez wielkość grupy roboczej

```
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable

__kernel void histogram(__global uint* in_data,

    __global uint* partial_histogram,

    unsigned int N )

{

    int LID = get_local_id(0);
```

```
    int GID = get_global_id(0);

    int local_size = get_local_size(0);

    int GRPIDX = get_group_id(0) * 256;

    local uint tmp_histogram[256];

    int j = 256;

    int indx = 0;

    for( j=256 ; j > 0 ; j-=local_size, indx+=local_size)

    {

        if (LID < j)

            tmp_histogram[indx+LID] = 0;

    }

    barrier(CLK_LOCAL_MEM_FENCE);

    if (GID < N)

    {

        uint value = in_data[GID];

        atom_inc(&tmp_histogram[value]);

    }

    barrier(CLK_LOCAL_MEM_FENCE);

    for( j=256, indx=0 ; j > 0 ; j-=local_size, indx+=local_size)

    {

        if (LID < j)

        partial_histogram[GRPIDX + indx + LID] += tmp_histogram[indx + LID];

    }

}
```

Obliczenie histogramu ostatecznego (za pomocą funkcji obliczeniowej `final_histogram`) jest już dość proste. Tworzona jest zmienna lokalna `tmp_histogram`, gdzie dana grupa robocza będzie sumować elementy z każdego histogramu częściowego. Liczba elementów, jaką trzeba do siebie dodać, zależy od liczby grup roboczych, gdyż liczba grup roboczych odpowiada liczbie histogramów częściowch. Aby nie powodować kolizji, indeksy adresujące zawsze będą zwiększane o wartość 256. W ten sposób kolejne iteracje pętli `while` odczytują różne elementy w różnych wątkach, zapewnia to także wartość `GID`, gdyż każdy element histogramu będzie miał swój własny

identyfikator, co w przypadku kodu przedstawia się następująco:

```
tmp_histogram[GID] += partial_histogram[group_indx + GID];

group_indx += 256;
```

Uruchomienie tej procedury ponownie wymaga zwrócenia uwagi na siatkę indeksującą. Jeśli pierwotnie zbiór danych miał 512 elementów i obliczono cztery histogramy częściowe (ze 128 elementów), to rozmiar siatki globalnej jest równy 256, bo z tylu elementów zbudowany jest histogram końcowy. Rozmiar siatki lokalnej to także 256 elementów (histogramy częściowe także mają po 256 elementów), ale wartość parametru `N` będzie równa 4, bo poprzednio wyznaczono tylko cztery histogramy częściowe.

Listing 5.6. Procedura obliczeniowa sumująca wyznaczone histogramy w jeden histogram końcowy (parametr `num_of_hist` oznacza liczbę histogramów, które należy do siebie dodać, aby otrzymać ostateczny histogram)

```
__kernel void final_histogram(__global uint *partial_ histogram, __global uint *histogram, unsigned int num_of_hist)

{

    int GID = get_global_id(0);

    int group_indx;

    int n = num_of_hist;

    local uint tmp_histogram[256];

    tmp_histogram[GID] = partial_histogram[GID];

    group_indx = 256;

    while (n > 0)

    {

        tmp_histogram[GID] += partial_histogram[group_indx + GID];

        group_indx += 256;

        n--;

    }

    histogram[GID] = tmp_histogram[GID];

}
```

5.4. Równoległe szukanie prawie binarne

Algorytm szukania binarnego to dobrze znany i szybki algorytm do znajdowania wartości w zbiorze danych. Aby przyspieszyć szukanie danego elementu, zbiór danych musi być zbiorem uporządkowanym.

Ogólnie algorym szukania binarnego ma złożoność logarytmiczną, a dokładnie jego złożoność można opisać jako $O(\log_2 n)$, gdyż w każdej iteracji poszukiwany zbiór jest o połowę mniejszy. Algorytm szukania binarnego można implementować w wersji rekurencyjnej lub iteracyjnej. Listing 5.7 przedstawia typową szeregową wersję rekurencyjną.

Główna idea w listingu 5.7 polega na sprawdzeniu, czy indeks `idx_max` jest większy niż `idx_min`. Jeśli tak, to można wyznaczyć indeks środkowy, a następnie w zależności od tego, czy element środkowy jest większy, czy mniejszy, możemy ponownie wywołać funkcję szukania rekurencyjnego tylko dla odpowiedniej połowy zbioru danych.

Listing 5.7. Szeregowa wersja algorytmu szukania binarnego w odmianie rekurencyjnej

```
int binary_search_rec(int *A, int elem_to_find, int idx_min, int idx_max)

{

    int idx_center;

    if (idx_max >= idx_min)

    {

        idx_center = (idx_min + idx_max) / 2 ;

        if (A[idx_center] > elem_to_find)

            return binary_search_rec(A, elem_to_find, idx_min, idx_center-1);

        else

            if (A[idx_center] < elem_to_find)

                return binary_search_rec(A, elem_to_find, idx_center+1, idx_max);

            else

                return idx_center;

    }

    else

    {

        return – 1;

    }

}
```

Jeśli wartości indeksów `idx_max` oraz `idx_min` nie spełniają relacji, to oznacza, iż poszukiwanego elementu nie ma w podanej tablicy. Łatwo zamienić też wersję rekurencyjną na iteracyjną, ale naszym zadaniem jest podanie wersji dla OpenCL.

5.4.1. Wersja równoległa dla OpenCL

Może to się wydawać zaskakujące, ale nie można bezpośrednio przenieść algorytmu szukania binarnego z listingu 5.7 na poziom jądra obliczeniowego. Pierwszy powód to rekurencja – nie można rekurencyjnie wywoływać jądra obliczeniowego[3]. Drugim powodem jest sposób adresowania pamięci – funkcja szukania odczytuje praktycznie w sposób losowy pojedyncze elementy, więc taki dostęp do pamięci nie jest dobrym rozwiazaniem w przypadku OpenCL, gdzie odczyt losowy zabiera najwięcej czasu.

Można zaproponować takie rozwiązanie, które wykorzysta dostępne rdzenie obliczeniowe, jednak wbrew pozorom nie musi to być szybsze rozwiązane niż oryginalne szukanie binarne. Listing 5.8 przedstwia kod źródłowy procedury obliczeniowej, która naśladuje proces szukania binarnego. Podobne rozwiązanie zostało zaprezentowane w pakiecie SDK dla OpenCL, dostarczane przez firmę AMD.

Argumenty tej procedury są następujące: `in_data` to zbiór danych, który będziemy przeszukiwać. Celowo został zastosowny typ wektorowy. Parametr `out_data` to wektor o dwóch elementach, gdzie znajdzie się informacja, czy udało się odszukać dany element, ostatni parametr to wartość elementu poszukiwanego.

Treść prodcedury `binarySearch` jest dość krótka – odczytujemy indeks aktualnie przetwarzanego elementu, a następnie sprawdzamy, czy aktualne wartości w wektorze są większe czy mniejsze niż szukany element. Jeśli tak jest, to nie udało się nam wskazać elementu. Jeśli warunek zwiódł, to wiadomo, iż udało się nam odszukać element lub wskazać dwa elementy, pomiędzy którymi znajduje się szukany element. Do zmiennej `out_data` zapisujemy indeks odszukanego elementu oraz jedność.

Sposób uruchomienia tej procedury wykaże, że nie dokońca jest to szukanie binarne, a raczej liniowe, które wykorzystuje uporządkowanie elementów. Dla zbioru o 64 elementach siatka globalna będzie mieć 32 elementy, wykorzystujemy bowiem fakt, że jeden element tablicy wejściowej zawiera dwa elementy. Rozmiar siatki lokalnej nie odgrywa dużej roli, może ona mieć wielkość 2, 4, 8, choć przełoży się to tylko na większą liczbę wątków w jednej grupie i przyspieszy proces szukania.

W odpowiedzi otrzymamy dwie liczby – pierwsza będzie oznaczać indeks odszukanego elementu, a druga liczba to jedność, oznaczająca, że udało się dany element odszukać. Jednak trzeba pamiętać, że zwrócony indeks wskazuje na pozycję szukanego elementu w tablicy, której elementy to wektory dwuelementowe. Dlatego otrzymany indeks należy pomnożyć przez dwa i sprawdzić dwa sąsiadujące ze sobą elementy, aby otrzymać dokładną wartość indeksu wskazującego, gdzie znajduje się szukany element. Może się też okazać, że danego elementu nie ma, bo procedura w listingu 5.8 sprawdza tylko, czy wartość szukanego elementu mieści się między wartościami wektora.

W przeciwieństwie do oryginalnego szukania binarnego, zaprezentowana procedura obliczeniowa wykonuje bardzo dużo porównań. Pierwotnie w szeregowej implementacji szukania binarnego zbiór był dzielony na połowę i połowa elementów była odrzucana. Jednak zaleta tej implementacji jest taka, że można podzielić zbiór na kilka sekcji, aby wykorzystać wiele jednostek obliczeniowych, przeszukiwać np. kilka sekcji badanego zbioru równolegle.

Listing 5.8. Procedura naśladująca proces szukania binarnego

```
__kernel void binarySearch(

    const __global uint2 *in_data,

    __global uint2 *out_data,

    const unsigned int elem_to_find)

{

    unsigned int GID = get_global_id(0);

    uint2 element = in_data[GID];

    if( (element.x > elem_to_find) || (element.y < elem_to_find))

    {

        return;

    }

    else

    {

        out_data[0].x = GID;

        out_data[0].y = 1;

    }

}
```

Procedurę obliczeniową z listingu 5.8 można także w dość prosty sposób przyspieszyć, stosując większy wektor w deklaracji danych wejściowych:

```
const __global uint4 *in_data
```

Należy też zmienić definicję zmiennej prywatnej `element`:

```
uint4 element = in_data[GID];
```

Instrukcja `if` musi porównywać inne składniki wektora:

```
if( (element.x > elem_to_find) || (element.w < elem_to_find))

{ … }
```

Tym razem sprawdzamy składniki `x`, czyli pierwszy element wektora oraz `w`, czyli czwarty element wektora. Rozmiar siatki obliczeniowej tym razem musi być dzielony przez 4, a odszukany indeks również będzie wskazywał na cztery kolejne elementy, które mogą zawierać poszukiwany element (ale nie muszą). Wiadomo tylko, że szukany element mieści się w zakresie wskazywanym przez elementy wektora `x` i `w`.

5.5. Przybliżanie wartości liczby π

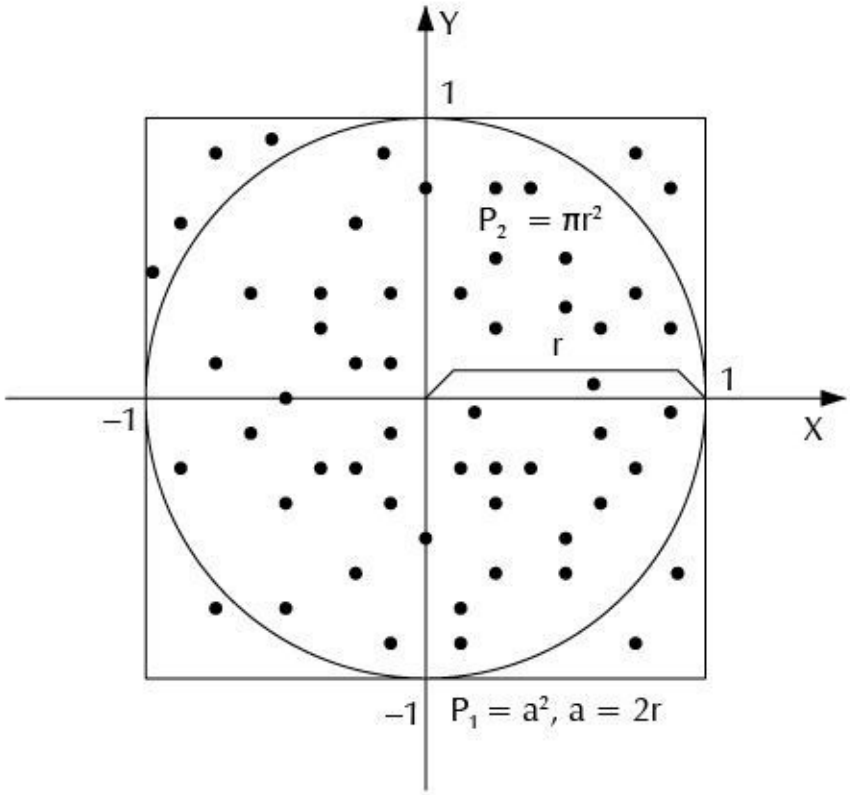
Przybliżanie wartości liczby π za pomocą metody Monte Carlo to dobrze znany przykład wykorzystania obliczeń równoległych do przyspieszania obliczeń. Również w przypadku OpenCL, bardzo często przybliżanie wartości liczby π jest wykorzystywane jako przykład możliwości obliczeniowych kart graficznych. My również poświęcimy mu nieco więcej uwagi, bo jest to dość prosty sposób przybliżania wartości liczby π, ale jego wersja równoległa wymaga nieco wysiłku, aby ją poprawnie opracować. Ze względu na walory edukacyjne warto dokonać implementacji tej metody w OpenCL. Wyjątkowo procedura nie będzie implementowana w API dla języka C, ale wykorzystamy pakiet PyOpenCL dla języka Python, o którym więcej informacji podano w rozdz. 6.1

5.5.1. Wersja szeregowa

Za nim przejdziemy do wersji równoległej opracowanej w ramach OpenCL, pokażemy wersję szeregową, aby lepiej poznać sposób wyznaczania wartości liczby π w ramach techniki Monte Carlo[4]. Na rysunku 5.4 przedstawiono ideę metody Monte Carlo w odniesieniu do wyznaczania wartości liczby π. Rysunek przedstawia okrąg o promieniu `r`, na którym został opisany kwadrat o długości boku `2r`.

Idea metody Monte Carlo przybliżająca wartość liczby n sprowadza się do tego, że będziemy losować po dwa punkty `x` oraz `y` znajdujące się wewnątrz kwadratu. Dla każdego wylosowanego punktu sprawdzamy, czy mieści się wewnątrz okręgu, tj. warunek $x^2 + y^2 < 1$. Wszystkie punkty, które znajdują się wewnątrz okręgu, należy naturalnie policzyć. Ponieważ stosunek pola okręgu do pola kwadratu wynosi `Pi/4`, to łatwo podać wzór, który posłuży nam do obliczenia liczby π, a będzie to `4 * cnt / mc`. Wielkość `cnt` to liczba punktów o współrzęnych `(x, y)`, które znajdują się wewnątrz okręgu, natomiast wartość `mc` to całkowita liczba wylosowanych punktów.

Zamiana powyższego opisu na odpowiedni program jest dość prosta, ale musimy mieć możliwość generacji punktów losowych, czy raczej pseudolosowych. Listing 5.9 przedstawia główną pętlę, która generuje punkty za pomocą generatora liczb pseudolosowych o nazwie TinyMT – *Tiny Mersenne Twister*[5], którego autorami są Mutsuo Saito oraz Makoto Matsumoto. Co prawda języki, takie jak C/C++, posiadają wbudowane generatory liczb pseudolosowych, ale wspominany generator TinyMT (oprócz niewielkiego rozmiaru) oferuje odpowiednią do naszych potrzeb funkcję. Nieco uogólniając, generatory wbudowane do bibliotek standardowych mogą wykazywać się słabymi własnościami statystycznymi, więc choć nasze obliczenia są mało skomplikowane, to lepiej do nich używać lepszych rozwiązań, jak np. TinyMT.



Rys. 5.4. Idea wyznaczania przybliżonej wartości liczby π za pomocą metody Monte Carlo

Listing 5.9. Szeregowa wersja przybliżania wartości liczby π

```
tinymt32_t tinymt;

int seed = 2;

tinymt32_init(&tinymt, seed);

float x, y, epi;

long i, counter = 0, max_counter = 1000000;

for( i=0 ; i < max_counter ; i++) {

    x = tinymt32_generate_float(&tinymt);

    y = tinymt32_generate_float(&tinymt);

    if( (x*x + y*y) < 1) counter++;

}

epi = 4.0 * (float)counter / (float)max_counter;

printf(”przybliżona wartość pi = %3.7f ”, epi);

printf(”błąd = %3.7f ”, fabs((float)M_PI – epi));
```

Poszczególne elementy naszego programu nie są skomplikowane. Na początku zainicjujemy generator liczb losowych za pomocą jakiejś liczby całkowitej – jest to tzw. **zalążek** lub **ziarno** (ang.

seed) – wpisując ją do zmiennej `seed`, może to być np. wartość 1, 2, 3 i tak dalej. Używanie tej samej wartości `seed` zawsze w efekcie będzie dawać ten sam ciąg liczb pseudolosowych. Wewnętrzny stan generatora będzie przechowywany w zmiennej `tinymt`. Proces inicjowania wykonamy za pomocą jednej linii kodu:

```
tinymt32_init(&tinymt, seed);
```

Po tym krótkim przygotowaniu pętla `for` generuje liczby `x`, `y` za pomocą funkcji `tinymt32_generate_float`, a naszym zadaniem pozostaje tylko sprawdzać, czy przynależą one do koła oraz zliczać, ile punktów trafiło do jego wnętrza. Zmienna `max_counter` określa liczbę wszystkich punktów, jakie zostaną wygenerowane.

Po zakończeniu pętli mnożymy przez cztery liczbę trafień, a w mianowniku podajemy całkowitą liczbę wygenerowanych punktów. W naszym programie wartość ta jest obliczana w następujący sposób:

```
epi = 4.0 * (float)counter / (float)max_counter;
```

Zamiast typu `float` można zastosować typ `double`, aby zwiększyć ewentualną dokładność obliczeń.

Tak wygląda cała metoda wyznaczenia przybliżonej wartości liczby π. Mimo że łatwo dokonać jej implementacji, to aby otrzymać wysoką dokładność, wymagane jest wylosowanie wielu punktów, np. dla miliona punktów otrzymamy liczbę π z dokładnością do około trzech miejsc po przecinku, co naturalnie oznacza niską wydajność tej metody.

Można się teraz zastanowić, w jaki sposób poprawić program z listingu 5.9, aby można było wykorzystać obliczenia równoległe. Kluczem jest sposób losowania punktów – program w listingu 5.9 losuje punkty kolejno za pomocą pętli, a można to zrobić równolegle. Temu sposobowi poświęcony jest rozdz. 5.5.2.

5.5.2. Wersja OpenCL – podejście bezpośrednie

Równoległe obliczanie przybliżenia wartości liczby π wykorzystuje niezależne badanie, czy współrzędne wylosowanego punktu należą do wnętrza okręgu jednostkowego. Rezultat testu zostaje zapamiętany w oddzielnej tablicy. Po zakończeniu tych badań pozostaje policzyć (możemy to także zrobić równolegle, wykorzystując operację redukcji), ile punktów znalazło się we wnętrzu okręgu, aby wyznaczyć przybliżoną wartość liczby π. Na początek w sposób równoległy wykonamy tylko sprawdzanie przynależności punktu do okręgu za pomocą OpenCL – równolegle wykonamy tylko test przynależności.

Aby zrealizować w OpenCL rozwiązanie opisane w rozdz. 5.5.1, możemy szybko skonstruować procedurę obliczeniową OpenCL przedstawioną w listingu 5.10. Poszczególne argumenty mają następujące znaczenie: dwa pierwsze (`a` oraz `b`) to tablice, które zawierają wartości współrzędnych poszczególnych punktów. Odczytując wartości z tablic, będziemy badać przynależność danego punktu do wnętrza okręgu o promieniu `r`. W naszym przypadku będzie to wartość `1.0`. Trzecim parametrem `inside_in_circle` jest tablica, w której umieścimy 0, jeśli badany punkt nie należy do okręgu, a 1, jeśli należy. Oznacza to, że liczba elementów, jaką przechowuje każda tablica `a`, `b` oraz `inside_in_circle` musi być taka sama. Ostatnie dwa parametry to wielkość promienia okręgu `r` oraz

całkowita liczba próbek `N`.

Listing 5.10. Równoległa wersja operacji obliczającej, czy badany punkt należy do wnętrza okręgu

```
__kernel void piEstimation( __global float *a,

    __global float *b,

    __global unsigned int *inside_in_circle,

    const float r,

    const int N)

{

    const int idx = get_global_id(0);

    if(idx < N)

    {

        if( (a[idx]*a[idx] + b[idx]*b[idx]) < r )

            inside_in_circle[idx]=1;

        else

            inside_in_circle[idx]=0;

    }

}
```

Sprawdzenie działania jądra obliczeniowego z listingu 5.10 najłatwiej wykonać za pomocą pakietu Python OpenCL. Omówimy teraz poszczególne czynności, jakie są realizowane w skrypcie.

Pierwsza czynność to włączenie odpowiednich plików z modułami:

```
import math

import pyopencl

import numpy
```

Pierwszy moduł to standardowy moduł Pythona z podstawowymi definicjami matematycznymi, w tym także z liczbą π. Po zakończeniu obliczeń porównamy jakość otrzymanego przybliżenia.

Nastepnie należy przygotować dwa parametry: liczbę próbek `N` oraz wartość promienia `r`:

```
N = 2 ** 15

r= numpy.float32(1.0)
```

Musimy od razu założyć liczbę próbek, co uprości postać siatki obliczeniowej do linii o wymiarach `1`

× N, gdzie N – liczba próbek (maksymlna wartość N wynosi 65536). To zasadnicze ograniczenie jest konsekwencją przyjętej postaci jądra obliczeniowego podanego w listingu 5.10. Nie jest to jednak dobre rozwiązanie pod względem wydajności. Znacznie lepsze zaproponujemy w następnym rozdziale.

Pomijając rozważania na temat wydajności, przed wywołaniem jądra obliczeniowego musimy przygotować zestaw liczb pseudolosowych. Najlepiej, aby (zgodnie z rys. 5.4) były to wartości z przedziału od –1 do 1. Jeśli wykorzystamy pakiet numpy, sprowadzi się to do dwóch następujących linii kodu w programie Python:

```
a = (numpy.random.random_sample(N)*(2.0)+(-1.0)).astype(numpy.float32)

b = (numpy.random.random_sample(N)*(2.0)+(-1.0)).astype(numpy.float32)
```

Korzystamy z metody random_sample, która generuje liczby pseudolosowe z przedziału [0.0, 1.0). Nie jest to założony wcześniej przedział, więc trzeba niejako przesunąć liczby do pożądanego przedziału, co zrealizujemy za pomocą mnożenia przez 2 oraz odejmując 1. Po takiej operacji otrzymamy zbiór wartości w założonym wcześniej przedziale. Za wielkość tablicy odpowiada parametr N.

Warto pamiętać, że wygenerowanie zestawu wartości losowych z przedziału a, b zrealizujemy w następujący sposób:

```
wektor = (numpy.random.random_sample(N)*(a-(b))+(b)).astype(numpy.float32)
```

Przygotowanie tablicy na wyniki również jest łatwe, gdy korzysta się z pakietu numpy:

```
inc_host = numpy.zeros(N).astype(numpy.uint32)
```

Tym razem tablica o długości N zostanie wypełniona zerami. W tym przypadku zamiast typu float stosujemy typ całkowity.

Po przygotowaniu tablic z danymi, kolejnym etapem jest przygotowanie obiektów współpracujących w OpenCL – tworzymy kontekst oraz kolejkę, co zawiera się w dwóch liniach kodu:

```
ctx = pyopencl.create_some_context()

q = pyopencl.CommandQueue(ctx)
```

Kolejne zadanie to utworzenie buforów OpenCL – dla zmiennych a oraz b będą to zmienne tylko do odczytu:

```
a_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=a)

b_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=b)
```

W ostatnim parametrze podajemy obiekt, z którego zostaną pobrane informacje o punktach. Fizyczne przeniesienie danych z pamięci gospodarza do pamięci urządzenia obliczeniowego zostanie wykonane już samodzielnie przez pakiet PyOpenCL.

Przygotowanie zmiennej do zapisu informacji o punktach znajdujących się wewnątrz okręgu to także jedna linia kodu:

```
inc_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.WRITE_ONLY, inc_host.nbytes)
```

Ze względu na charakter tej funkcji nie musimy kopiować do niej danych, ale po wykonaniu obliczeń musimy odczytać zawartość `inc_buf`, aby stwierdzić, ile punktów trafiło do wnętrza okręgu.

Kolejne zadanie – kompilację jądra obliczeniowego – wykonujemy za pomocą dwóch konstrukcji. W pierwszej tworzymy tylko obiekt programu:

```
prgobj = pyopencl.Program(ctx, """

    // treść kodu w OpenCL

    """)
```

a jego kompilacja oraz budowa następuje po wydaniu polecenia:

```
prg = prgobj.build()
```

Jesli nie pojawią się błędy, to możemy przystąpić do określenia parametrów wywołania funkcji obliczeniowej. Odczytujemy obiekt reprezentujący jądro obliczeniowe:

```
piEstimation = prg.piEstimation
```

a następnie określamy poszczególne parametry zgodnie z ich kolejnością, jaka wystepuje w kodzie OpenCL w Listingu 5.10:

```
piEstimation.set_scalar_arg_dtypes([None, None, None, numpy.float32, numpy.uint32])
```

Wartość `None` jest stosowana do oznaczenia typów wskaźnikowych, a typy skalarne, czyli parametry `r` oraz `N`, muszą mieć typ określony precyzyjnie. Dlatego dwa ostatnie elementy to nazwy typów `float32` oraz `uint32`.

Dopiero teraz można uruchomić procedurę obliczeniową OpenCL. Wykorzystamy do tego uzyskany wcześniej obiekt `piEstimation`:

```
piEstimation(q, a.shape, None, a_buf, b_buf, inc_buf, r, N)
```

Znaczenie poszczególnych parametrów jest następujące: zmienna `q` – kolejka, `a.shape` – kształt globalnej siatki obliczeniowej (w naszym przypadku to wektor o wymiarach `1 × N`, bo zakładamy, że zostanie uruchomionych `N` wątków równocześnie). Nie określamy postaci siatki lokalnej, dlatego w trzecim argumencie podana została wartość `None`. Następne argumenty to już właściwe dane dla naszej procedury obliczeniowej.

Aby poznać liczbę punktów, które trafiły do wnętrza okręgu, musimy odczytać dane z pamięci urządzenia obliczeniowego. Do tego potrzebna jest dodatkowa zmienna po stronie gospodarza:

```
inc_host_2 = numpy.empty_like(inc_host)
```

Metoda `empty_like` tworzy zmienną identycznego typu i rozmiaru co zmienna podana w argumencie. Odczyt danych z pamięci urządzenia to zadanie dla metody `enqueue_copy`:

```
pyopencl.enqueue_copy(q, inc_host_2, inc_buf )
```

Ostatnie obliczenie, jakie trzeba wykonać, to policzenie liczby punktów należących do okręgu. W zmiennej `inc_host_2`, będącej wektorem, znajdują się tylko zera i jedności, dlatego wystarczy obliczyć sumę elementów tego wektora:

```
nic = 0

for i in inc_host_2:

    nic = nic + i piest = nic*4.0 / N
```

Na koniec możemy wyświetlić wartość przybliżonej liczby π:

```
print ”pi approx = ”, piest
```

Niestety, jakość naszego przybliżenia nie będzie dość dobra, wystarczy kilka razy uruchomić skrypt np. dla `N` = 32768, aby się przekonać, że otrzymane przybliżenia zazwyczaj mieszczą się w przedziale od 3.13 do 3.15 i nie jest to zbyt dobry rezultat, choć łatwo trafić dobre przybliżenia 3.141. O tym nie najlepszym wyniku decyduje zarówno mała liczba próbek, jak też nie najlepszy generator liczb pseudolosowych. Nasza implementacja w OpenCL nie wykorzystuje optymalnie mocy urządzenia obliczeniowego, choć proces testowania przynależności jest znacznie szybszy niż wersja szeregowa.

Listing 5.11. Skrypt w programie Python wyznaczający przybliżoną wartość liczby π, gdzie test przynależności jest wykonywany przez jądro obliczeniowe OpenCL z listingu 5.10

```
#! /usr/bin/python


import math


import pyopencl


import numpy


N = 2 ** 4


r= numpy.float32(1.0)
```

```
a = (numpy.random.random_sample(N)*(1.0-(-1.0))+(-1.0)).astype(numpy.float32)

b = (numpy.random.random_sample(N)*(1.0-(-1.0))+(-1.0)).astype(numpy.float32)

inc_host = numpy.zeros(N).astype(numpy.uint32)

ctx = pyopencl.create_some_context()

q = pyopencl.CommandQueue(ctx)

a_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=a)

b_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=b)

inc_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.WRITE_ONLY, inc_host.nbytes)

prgobj = pyopencl.Program(ctx, """

// treść kodu OpenCL taka jak na Listingu 5.5.2

    """)

prg = prgobj.build()

piEstimation = prg.piEstimation

piEstimation.set_scalar_arg_dtypes([None, None, None, numpy.float32, numpy.uint32])
```

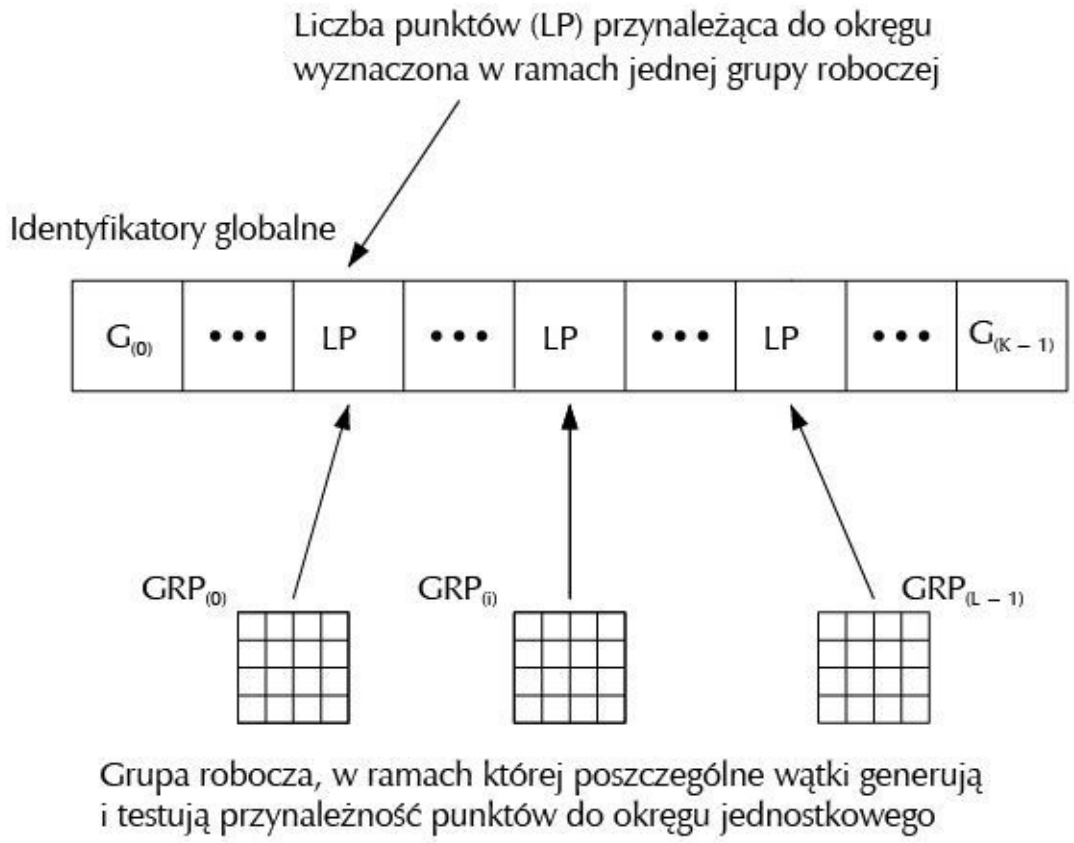
```
piEstimation(q, a.shape, None, a_buf, b_buf, inc_buf, r, N)



inc_host_2 = numpy.empty_like(inc_host)

pyopencl.enqueue_copy(q, inc_host_2, inc_buf )

nic = 0

for i in inc_host_2:

    nic = nic + i piest = nic*4.0 / N

print ”pi = ”, math.pi

print ”pi approx = ”, piest

print ”error = ”, math.fabs(math.pi – piest)
```

5.5.3. Wersja OpenCL – podejście o wyższej wydajności

Zaprezentowane w rozdz. 5.5.3 rozwiązanie nie jest poprawnym rozwiązaniem z punktu widzenia architektury obliczeniowej oferowanej przez OpenCL. Aby uzyskać wyższą wydajność, należy poprawnie wykorzystać lokalną oraz globalną siatkę obliczeniową. Wbrew pozorom, nie to będzie najtrudniejsze w implementacji lepszej metody wyznaczania liczby π.

Poprawne wykorzystanie siatki obliczeniowej będzie polegać na losowaniu współrzędnych punktu, testowaniu jego przynależności do wnętrza okręgu, zapamiętywaniu liczby punktów spełniających test w lokalnej zmiennej, np. `local_hit_tbl`, która jest współdzielona przez wszystkie wątki w danej grupie.

Wszystkie wątki można rozróżnić za pomocą identyfikatora lokalnego oraz globalnego przydzielonego dla każdego z nich. Identyfikator lokalny będzie używany do zapamiętania liczby punktów, które spełniły test w jednym wątku. Można zdefiniować tzw. **zmienną lokalną**, która jest dostępna dla wszystkich wątków z danej grupy. Każdy wątek zapisze liczbę punktów, które przeszły pomyślnie test przynależności do tablicy tylko pod jednym indeksem – numer tego indeksu to identyfikator lokalny. Po zakończeniu pracy przez wszystkie wątki w pojedynczej grupie, zadaniem jednego z nich będzie obliczenie sumy wszystkich trafień lokalnych i zapisanie wyników do globalnej tablicy trafień. Wynikiem całej procedury będzie globalna tablica, po której odczytaniu, podobnie jak poprzednio, procesor obliczy ostateczną sumę.



Rys. 5.5. Podział obliczeń w ulepszonej procedurze wyznaczania przybliżonej wartości liczby π

Listing 5.12. Wydajniejsza wersja badania przynależności punktów do wnętrza okręgu

```
__kernel void estimate_pi_value(__global tinymt32wp_t *d_status,
            __global uint *global_hit_tbl,
            __local uint *local_hit_tbl,
            uint seed,
            uint N)
{
    const size_t gid = get_group_id(0);
    const size_t lid = get_local_id(0);
    const size_t local_hit_tbl_size = get_local_size(0);
    const size_t tinymt_id = tinymt_get_sequential_id();
    tinymt32wp_t tiny_prng;
    float x, y;
    uint i, tmp_sum = 0;
    tinymt32_status_read(&tiny_prng, d_status);
```

```
    tinymt32_init(&tiny_prng, seed + tinymt_id);

    for (i = 0; i < N; i++)

    {

        x = tinymt32_single01(&tiny_prng);

        y = tinymt32_single01(&tiny_prng);

        if (x * x + y * y < 1.0f)

        {

            tmp_sum++;

        }

    }

    local_hit_tbl[lid] = tmp_sum;

    barrier(CLK_LOCAL_MEM_FENCE);

    tmp_sum = 0;

    if (lid == 0)

    {

        for (i = 0; i < local_hit_tbl_size; i++)

        {

            tmp_sum += local_hit_tbl[i];

        }

        global_hit_tbl[gid] = tmp_sum;

    }

}
```

Opisane zadanie w całości realizuje procedura obliczeniowa z listingu 5.12. Pierwsza pętla `for` generuje punkty przy wykorzystaniu generatora TinyMT. Wartość parametru `N` określa liczbę tych punktów, nie oznacza całkowitej liczby próbek. Dokładną liczbę tych punktów przedstawimy, opisując sposób wywołania tej funkcji.

Po wyznaczeniu odpowiedniej liczby punktów zapisujemy ją do zmiennej `local_hit_tbl`, a następnie stawiamy barierę synchronizującą. Zablokuje ona działanie pojedynczego wątku dopóki wszystkie inne wątki w danej grupie nie zakończą testowania punktów i obliczenia nie dotrą do punktu wyznaczonego przez barierę.

Gdy tak się stanie, wątek o indeksie 0 obliczy sumę danych zgromadzonych w lokalnej zmiennej

`local_hit_tbl`, a wynik zostanie zapisany do zmiennej `global_hit_tbl`.

Większy problemem stanowi generator TinyMT – można się pokusić o jego samodzielną implementację, ale ze względu na bardzo ścisłą postać algorytmu generatora TinyMT, otrzymany kod będzie bardzo podobny do oryginalnego rozwiązania. Dodatkowo, samodzielna implementacja jest utrudniona przez to, że generator będzie działał w środowisku równoległym, więc trzeba przygotować możliwość obsługi wielu strumieni, będących źródłem przyszłych wartości pseudolosowych. Dlatego, skorzystamy z wersji TinyMT przygotowanej nie tylko dla pojedynczego procesora, z której skorzystaliśmy już w rozdz. 5.5.1, ale też dla technologii OpenCL.

Nas interesuje archiwum[6] *TinyMT-opencl-src-1.0.tar.gz* (ewentualnie wersja w formacie ZIP). Mimo że mamy tam wszystkie niezbędne pliki, a nawet podobny do naszego przykład przybliżający wartość liczby π, musimy odpowiednio przygotować źródła generatora TinyMT dla naszych potrzeb.

Pliki: *tinymt32.clh*, *tinymt.clh*, *tinymt32def.h* musimy połączyć w jedną całość, aby otrzymać zestaw definicji do obsługi generatora TinyMT. W wersji dla CPU kompilator wykonał za nas to zadanie, gdy użyliśmy instrukcji preprocesora `#include`, włączając odpowiedni plik. Tym razem zrobimy to samodzielnie, przy czym kolejność dołączania poszczególnych plików ma znaczenie. Ogólnie na początku w finalnym pliku musi się znaleźć treść pliku *tinymt.clh*, następnie dołączamy zawartość *tinymt32def.h*, a na samym końcu zawartość pliku *tinymt32.clh*. Podczas łączenia można sprawdzić, czy da się usunąć nieużywane funkcje. Proces łączenia plików możemy zlecić preprocesorowi języka C/C++, np. wykorzystując preprocesor cpp dostępny w kompilatorze GCC, wydajemy następujące polecenie:

```
cpp –DKERNEL_PROGRAM tinymt32.clh –o tinymt-afterpreproc.i
```

Zawartość otrzymanego pliku *tinymt-afterpreproc.i* dołączamy do źródeł jądra obliczeniowego przed funkcją `estimate_pi_value`. Definicja `KERNEL_PROGRAM` jest bardzo istotna, bowiem bez jej zastosowania preprocesor wygeneruje nam zestaw funkcji bez kilku definicji niezbędnych, aby kod kompilował się poprawnie w środowisku OpenCL.

Teraz możemy uruchomić jądro obliczeniowe w pakiecie Python OpenCL. Większość przygotowań jest identyczna jak w poprzednim przykładzie. W inny sposób określamy liczbę próbek (zmienna `data_size`), mnożąc wielkości `group_num` przez `local_num` oraz przez `1024`. Musimy pamiętać, aby przy zwiększaniu liczby wątków przypadających na grupę lokalną nie przekroczyć ilości pamięci lokalnej dostępnej w danym urządzeniu OpenCL. Dodajmy też, że pojedynczy wątek wyznaczy `data_size/total_num` próbek.

W odróżnieniu od poprzedniego przykładu, mamy zmienną umieszczoną w pamięci lokalnej. Przydział bufora dla danych tego typu zmiennych wykonujemy w następujący sposób:

```
lcl_sum = pyopencl.LocalMemory(numpy.dtype(numpy.uint32).itemsize * local_num)
```

Procedura obliczeniowa korzysta też z typu `tinymt32wp_t`, co oznacza, że musimy z poziomu języka Python zdefiniować odpowiednią zmienną dostępową, aby poprawnie zainicjować zestaw parametrów startowych poszczególnych instancji generatorów TinyMT.

Nie będziemy w tym miejscu dokładnie opisywać, w jaki sposób utworzyć odpowiedni most pomiędzy strukturą w języku C a środowiskiem Pythona, ponieważ zostało to szerzej przedstawione

w rozdz. 6.1.4. Od razu tworzymy typ pośredniczący w następujący spób:

```
tinymt32wp_t_proxy = numpy.empty(total_num, tinymt32wp_t)
```

Niestety, to nadal nie jest jeszcze koniec przygotowań. Należy teraz, wykorzystując plik *tinymt32dc.0.2000.txt* dostępny w oryginalnym archiwum TinyMT, wczytać parametry do zainicjowania generatorów TinyMT. Odpowiedni kod odczytujący dane został podany w listingu 5.13. Wykorzystujemy moduł `csv`, co upraszcza proces odczytu.

Kolejne czynności są podobne do opisanych w poprzednim rozdziale, więc możemy je tu pominąć. Wywołanie procedury obliczeniowej również jest podobne, ale tym razem musimy określić postać siatki obliczeniowej:

```
estimate_pi_value(q, ((total_num),), ((local_num),), tinymt32wp, glb_sum, lcl_sum,
seed, (data_size / total_num))
```

Wymiary siatki globalnej i lokalnej to odpowiednio drugi oraz trzeci parametr.

Podobnie jak w poprzednim przykładzie, musimy odczytać dane, aby już za pomocą kodu w Pythonie obliczyć ostateczną wartość sumy. Tym razem elementów do sumowania jest znacznie mniej, bo zmienna `total_num` określa tę wielkość.

W porównaniu do poprzedniego przykładu, dokładność będzie lepsza, błąd przybliżenia nie występuje już na drugim, trzecim, ale dopiero na piątym miejscu po przecinku.

Listing 5.13. Skrypt w Pythonie wyznaczający przybliżoną wartość liczby π, gdzie test przynależności jest wykonywany przez ulepszone jądro obliczeniowe OpenCL wykorzystujące generator TinyMT

```
#! /usr/bin/python

import math

import pyopencl

import pyopencl.tools

import numpy

import time
```

```
import csv

kernel_source = """

deficja generatora

funkcja estimate_pi_value

"""

seed = numpy.uint32(time.time())

group_num = 1024;

local_num = 256;

total_num = group_num * local_num;

data_size = total_num * 1024;

r= numpy.float32(1.0)

ctx = pyopencl.create_some_context()

q = pyopencl.CommandQueue(ctx)

glb_sum_host = numpy.zeros( numpy.dtype(numpy.uint32).itemsize * total_num / local_num
).astype(numpy.uint32)
```

```
lcl_sum = pyopencl.LocalMemory(numpy.dtype(numpy.uint32).itemsize * local_num)

glb_sum = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_WRITE | pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=glb_sum_host)

tinymt32wp_t = numpy.dtype([

(”s0”, numpy.uint32),

(”s1”, numpy.uint32),

(”s2”, numpy.uint32),

(”s3”, numpy.uint32),

(”mat1”, numpy.uint32),

(”mat2”, numpy.uint32),

(”tmat”, numpy.uint32) ])

tinymt32wp_t, tinymt32wp_t_c_decl = pyopencl.tools.match_dtype_to_c_struct(ctx.devices[0], ”tinymt32wp_t”, tinymt32wp_t)

tinymt32wp_t_proxy = numpy.empty(total_num, tinymt32wp_t)

tinymt32wp_file = open(‘tinymt32dc.0.2000.txt’,”r”)

csv_reader = csv.reader(tinymt32wp_file)
```

```
idx=0

for row in csv_reader:

    tinymt32wp_t_proxy[idx][”s0”]=0

    tinymt32wp_t_proxy[idx][”s1”]=0

    tinymt32wp_t_proxy[idx][”s2”]=0

    tinymt32wp_t_proxy[idx][”s3”]=0

    tinymt32wp_t_proxy[idx][”mat1”]=int(row[3], 16)

    tinymt32wp_t_proxy[idx][”mat2”]=int(row[4], 16)

    tinymt32wp_t_proxy[idx][”tmat”]=int(row[5], 16)

     idx = idx + 1

    if idx >= total_num:

        break

tinymt32wp = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=tinymt32wp_t_proxy)

prgobj = pyopencl.Program(ctx, kernel_source)

prg = prgobj.build()

estimate_pi_value = prg.estimate_pi_value
```

```
estimate_pi_value.set_scalar_arg_dtypes([None, None, None, numpy.uint32, numpy.uint32])

estimate_pi_value(q, ((total_num),), ((local_num),), tinymt32wp, glb_sum, lcl_sum, seed, (data_size / total_num))

glb_sum_2 = numpy.empty_like(glb_sum_host)

pyopencl.enqueue_copy(q, glb_sum_2, glb_sum )

nic = 0

for i in glb_sum_2:

    nic = nic + i piest = nic*4.0 / data_size

print ”pi = ”, math.pi

print ”pi przybliżenie = ”, piest

print ”Błąd = ”, math.fabs(math.pi – piest)
```

Zaprezentowane implementacje metody Monte Carlo warto podsumować, badając ich wydajność. Można zmierzyć czas działania całego programu oraz czas działania głównej procedury obliczeniowej, tj. testowania przynależności punktów do wnętrza okręgu. Sposób, w jaki to można zrobić, pokazano w rozdz. 7, gdzie za pomocą narzędzia CodeXL badana jest wydajność procedury do symulacji „Gry w życie”.

5.6. Gra w życie

Gra w życie to jeden z najsłynniejszych przykładów automatu komórkowego, którego reguły zostały podane przez Johna Conwaya. Rozgrywka (choć nie jest to gra w typowym rozumieniu) toczy się na nieskończonej płaszczyźnie. W przypadku symulacji komputerowej płaszczyzną tą jest macierz o wymiarach $n \times m$, często stososuje się macierz kwadratową $n \times n$.

Poszczególne elementy macierzy są nazywane **komórkami**. Elementy macierzy mogą przyjmować tylko dwie wartości: 1 − gdy komórka jest zamieszkana / żywa albo 0 − gdy komórka jest niezamieszkana / martwa. Ponieważ gra toczy się na płaszczyźnie, to każda komórka ma ośmiu sąsiadów. Reguły gry w życie są następujące :

• komórka sąsiadująca z trzema lub dwiema zamieszkanymi komórkami nadal będzie żywa, inaczej mówiąc, pozostanie zamieszkana do następnej iteracji,

• komórka sąsiadująca z czterema lub więcej zamieszkanymi komórkami staje się niezamieszkana, inaczej mówiąc, ginie z przeludnienia,

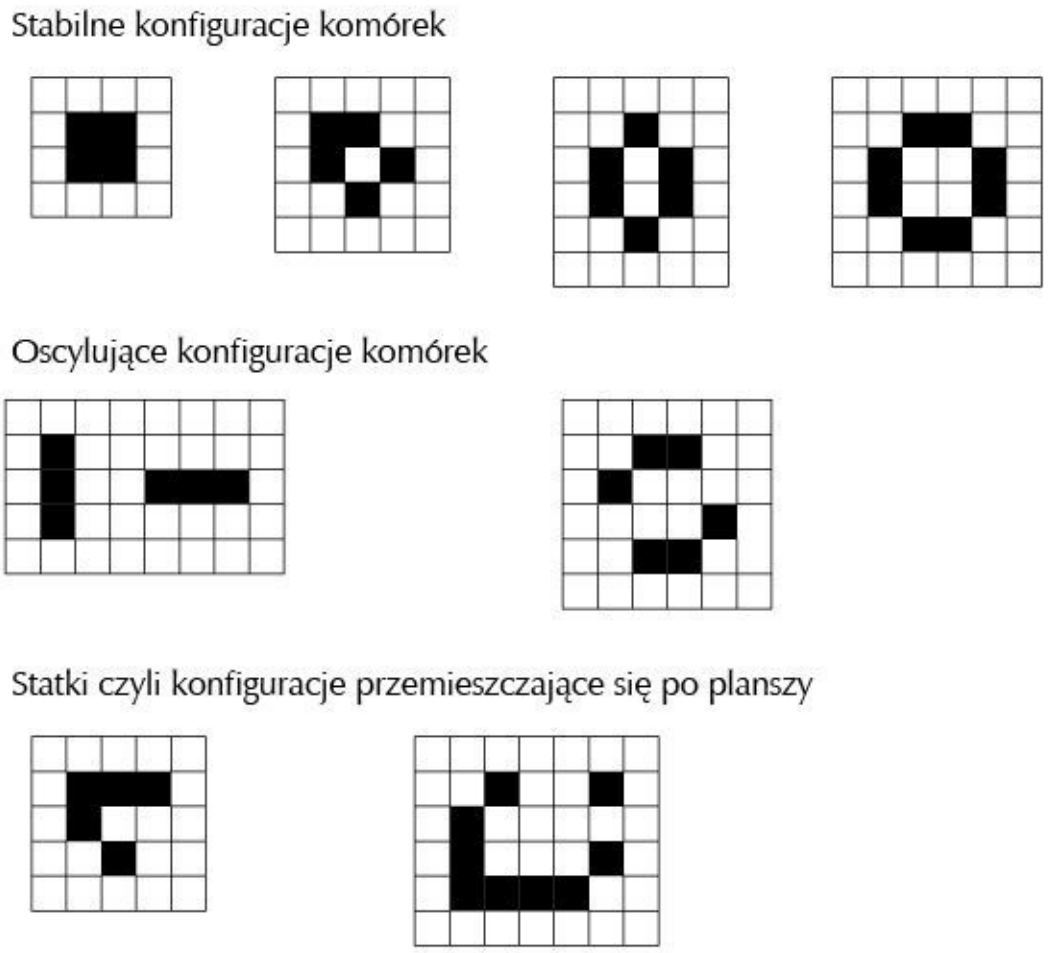
• komórka sąsiadująca z mniej niż dwiema zamieszkanymi komórkami ginie z powodu braku sąsiadów,

• każda komórka martwa (niezamieszkana/pusta) sąsiadująca z trzema zamieszkanymi staje się komórką żywą.

Pojedyncza iteracja gry sprowadza się do przejrzenia wszystkich komórek w macierzy i zastosowaniu do nich reguł opisanych powyżej. Płaszczyzna, na której zaczyna się gra, powinna zawierać jakieś żywe komórki. Omówienie postaci komórek, ich rodzajów, jest tematem na oddzielną książkę.

Na rysunku 5.6 przedstawiono kilka podstawowych układów, które potrafią pozostać stabilne (nieśmiertelne) bądź oscylować, a nawet tworzyć nowe komórki czy przemieszczać się po planszy.

Gra w życie jest dość prostym procesem do symulacji, więc dość szybko można opracować własny program symulujący tę grę − zarówno w wersji szeregowej, jak i wykorzytując możliwości równoległego przetwarzania w technologii OpenCL.



Rys. 5.6. Przykładowe rodziny komórek w ”Grze w życie”

5.6.1. Gra w życie – procedura obliczeniowa

Technologia OpenCL ułatwia opracowanie prostego symulatora gry w życie, z wykorzystaniem obiektów do przetwarzania obrazów. Bardziej szczegółowe informacje o przetwarzaniu obrazów podano w rozdz. 5.9. Tu wykorzystamy jedynie fakt, że obraz to struktura dwuwymiarowa, a stan początkowy płaszczyzny będziemy wczytywać jak rysunek w formacie PNG. Rysunek ten może zawierać tylko dwa kolory, np. biały oznaczający komórkę żywą oraz czarny − komórkę martwą. Kolory będą kodowanie w modelu RGB, czarny (0,0,0) i biały (255,255,255), co pozwala także kolorować planszę, oznaczając różnymi kolorami komórki, które żyją, umierają, bądź się przesuwają.

Zastanowimy się, w jaki sposób wykorzystać dostępne rdzenie obliczeniowe, aby uzyskać wysoką wydajność. Gra wymaga obecności macierzy zawierającej aktualny stan − w naszym przypadku będzie to obraz. Zadanie polega na odczytaniu informacji o pikselach, które traktujemy jak komórki, i utworzeniu drugiego obrazu na podstawie tych wartości. W nowym obrazie będą umieszczane nowe komórki na podstawie stanu komórek z pierwszego obrazu. W ten sposób wykonamy jedną iterację symulacji gry w życie. Kolejna iteracja sprowadza się do zamiany obrazów miejscami, nie trzeba nawet kasować zawartości obrazu, na miejscu którego zostanie umieszczana nowa iteracja, bowiem przeglądamy wszystkie komórki obrazu źródłowego i każda komórka obrazu docelowego otrzyma nową wartość.

Listing 5.14 zawiera główne elementy jądra obliczeniowego. Podstawową procedurą, którą wywołujemy jest `next_iteration`, gdzie podajemy dwa obrazy: wejściowy (parametr `in_image`) oraz wyjściowy (`out_image`). Istotną rolę pełni także parametr `rules`, gdzie znajduje się opis reguł stosowanych do symulacji gry w życie. Tablica `rules` umożliwia stosowanie różnych reguł, a szczegóły dotyczące jej wypełnienia zostaną podane dalej. Tablica jest opatrzona słowem `__constant`, co oznacza, że jest to zmienna przeznaczona wyłącznie do odczytu.

Czynności wykonywane w procedurze obliczeniowej `next_iteration` są krótkie. Odczytujemy aktualne współrzędne w obrazie na podstawie identyfikatorów pozyskanych z siatki obliczeniowej:

```
int2 coord = (int2)(get_global_id(0),get_global_id(1));
```

Za pomocą funkcji `read_imageui` do odczytu danych z obrazu odczytujemy wartość danej komórki:

```
uint4 state = read_imageui(in_image, sampler, coord);
```

Następnie odczytujemy liczbę komórek żywych za pomocą wywołania funkcji pomocnicznej `num_of_neighbour_cells`.

```
uchar nnc = num_of_neighbour_cells(state, coord, in_image);
```

Idea tej funkcji jest bardzo prosta − za pomocą dwóch pętli `for` przeglądane jest otoczenie aktualnie badanej komórki, a następnie odczytana liczba żywych komórek jest wykorzystywana do budowy indeksu, za pomocą którego odczytujemy z tablicy `rules`, czy dana komórka ma być komórką żywą, czy też martwą.

Ponieważ nie sprawdzamy zakresu współrzędnych siatki globalnej, to musimy zadbać o to, aby jądro obliczeniowe otrzymało poprawne rozmiary przetwarzanego obrazu. Ważny jest także odczyt danych z krawędzi obrazu. Procedura `num_of_neighbour_cells` nie sprawdza zakresu, gdyż robi to samodzielnie procedura `read_imageui`, dzięki obecności tzw. **samplera**. Jego definicja została podana na początku listingu 5.14. Mamy tam podaną wartość `CLK_ADDRESS_CLAMP` (więcej podstawowych informacji o obrazach w rozdz. 3.8 oraz 5.9), co oznacza, że w przypadku odczytu danych spoza obszaru obrazu zwracany jest tzw. kolor domyślny, zazwyczaj czarny. W naszym przypadku kolor czarny oznacza martwą komórkę.

Ostatnim elementem głównej procedury obliczeniowej jest zapis informacji o komórce do obrazu wyjściowego:

```
write_imageui(out_image, coord, (uint4)(rules[i],rules[i],rules[i],1));
```

Należy podkreślić, że każdy punkt obrazu, czyli jedna komórka, jest w ramach jednej iteracji przetwarzana niezależnie od pozostałych, dlatego postać jądra obliczeniowego jest stosunkowa krótka.

Listing 5.14. Jądro obliczeniowe do symulacji gry w życie

```
constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;

inline uchar num_of_neighbour_cells(

        uint4 state,

        int2 coord,

        __read_only image2d_t image)

{

    uchar counter = 0;

    uint4 neighbour_cell;

    for (int i=-1; i<=1; i++)

    {

        for (int k=-1; k<=1; k++)

        {

                        neighbour_cell = read_imageui(image, sampler, (int2)(coord.x+i,coord.y+k));

            counter += (neighbour_cell.x >> 7);

        }

    }

    return (counter – (state.x >> 7));

}

__kernel void next_iteration(

        __read_only image2d_t in_image,

        __write_only image2d_t out_image,

        __constant uchar *rules )

{

    int2 coord = (int2)(get_global_id(0),get_global_id(1));
```

```
    uint4 state = read_imageui(in_image, sampler, coord);

    uchar nnc = num_of_neighbour_cells(state, coord, in_image);

    uchar i = nnc + 9*(state.x >> 7);

    write_imageui(out_image, coord,

(uint4)(rules[i],rules[i],rules[i],1));

}
```

5.6.2. Gra w życie – obsługa symulacji

Obsługa jądra obliczeniowego z listingu 5.14 jest podobna jak w poprzednich przykładach, chociaż nie tworzymy typowych buforów danych, ale obrazy, co wymaga deklaracji ich formatu:

```
image_format.image_channel_data_type = CL_UNSIGNED_INT8;

image_format.image_channel_order = CL_RGBA;
```

Obiekt pamięciowy dla obrazu wejściowego tworzony jest w następujący sposób:

```
input_image = clCreateImage2D(cl_compute_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, &image_format, WIDTH, HEIGHT, 0, (void*)bytes, &cli_err_num);
```

Zmienna `bytes` zawiera stan początkowy symulacji, jaką będziemy przeprowadzać. Utworzenie obrazu wyjściowego jest podobne:

```
output_image = clCreateImage2D(cl_compute_context, CL_MEM_READ_WRITE, &image_format, WIDTH, HEIGHT, 0, NULL, &cli_err_num);
```

Dane opisujace reguły gry w życie, mimo iż jest to niewielka tabela, również należy utworzyć w postaci bufora pamięciowego:

```
deviceRules = clCreateBuffer(cl_compute_context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, rulesSizeBytes, rules, &cli_err_num);
```

Są to dane przeznaczone tylko do odczytu, o czym świadczy atrybut `constant` umieszczony w argumencie `rules` jądra obliczeniowego `next_iteration`.

Inne czynności odnoszące się do podanej procedury obliczeniowej są analogiczne jak w poprzedniej, tylko sposób wywołania jądra obliczeniowego tym razem jest bardziej rozbudowany. Program do symulacji za pomocą pętli `for` generuje podaną liczbę klatek symulacji, które są zapisywane jako obrazy w formacie PPM (można też zastosować format PNG).

Listing 5.15 zawiera całość tego procesu. Na początku określamy wymiary siatki lokalnej oraz globalnej. W pętli `for` pojawiają się instrukcje warunkowe `if` o postaci:

```
if(frame % 2 == 0) { … }
```

W zależności od tego, czy numer iteracji jest parzysty, czy nieparzysty, odpowiednio zamieniamy parametry dla jądra obliczeniowego oraz kopiujemy dane z właściwego obrazu do pliku w drugiej

części kodu w listingu 5.15.

Wywołanie jądra obliczeniowego jest nieco inne niż to było w poprzednich przykładach:

```
cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 2, NULL, globalWorkSize, localWorkSize, 0, NULL, &kernelEvent );
```

Korzystamy z siatki lokalnej oraz globalnej, a także ze zdarzenia, które sygnalizuje, że jądro obliczeniowe zakończyło obliczenia, i będziemy odczytywać informacje o czasie działania procedury obliczeniowej. Aby możliwe było profilowanie kodu, w trakcie tworzenia kolejki, należy dodać stałą `CL_QUEUE_PROFILING_ENABLE`:

```
cl_compute_command_queue = clCreateCommandQueue( cl_compute_context, platform_devices[0], CL_QUEUE_PROFILING_ENABLE, &cli_err_num);
```

Odczyt wartości czasu, w jakim procedura obliczeniowa została uruchomiona i zakończona, wykonujemy, używając obiektu zdarzenia:

```
cli_err_num = clGetEventProfilingInfo(kernelEvent, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);

cli_err_num |= clGetEventProfilingInfo(kernelEvent, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
```

Znalezienie czasu trwania obliczeń sprowadza się do obliczenia prostej różnicy:

```
executionTime = (end – start) * 1.0e-6f;
```

Mnożenie zmienia jednostkę czasu, która domyślnie jest równa jednej milionowej sekundy (tj. jednej mikrosekundzie) − po pomnożeniu przez milion otrzymujemy czas w sekundach. Listing 5.15 zawiera również kod dotyczący kopiowania obrazu, który zostanie omówiony w dalszych przykładach.

Listing 5.15. Uruchamianie procesu symulacji gry w życie

```
globalWorkSize[0]=WIDTH;

globalWorkSize[1]=HEIGHT;

localWorkSize[0]=16;

localWorkSize[1]=16;

int frame;

for( frame = 0; frame < 64 ; frame++)

{

    printf(”ramka = %d\n”, frame);

    if(frame % 2 == 0)

    {
```

```
        cli_err_num = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_image);

        cli_err_num = clSetKernelArg(kernel, 1, sizeof(cl_mem), &output_image);

        cli_err_num = clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceRules);

        if(cli_err_num != CL_SUCCESS) {

            printf(”Nie udało się ustalić parametrów jądra!\n”);

            return EXIT_FAILURE;

        }

    } else {

        cli_err_num = clSetKernelArg(kernel, 0, sizeof(cl_mem), &output_image);

        cli_err_num = clSetKernelArg(kernel, 1, sizeof(cl_mem), &input_image);

        cli_err_num = clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceRules);

        if(cli_err_num != CL_SUCCESS) {

            printf(”Nie udało się ustalić parametrów jądra!\n”);

            return EXIT_FAILURE;

        };

    }

     cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_ queue, kernel, 2, NULL,
globalWorkSize, localWorkSize,

                                                                    0, NULL, &kernelEvent );

    clWaitForEvents(1, &kernelEvent);

    if (cli_err_num != CL_SUCCESS)

    {

        printf(”Błąd: wywołanie jądra nie powiodło się!\n”);

        return EXIT_FAILURE;

    } else {

        if (copyEvent == NULL)

        {

            cl_ulong start, end;

                                cli_err_num   =   clGetEventProfilingInfo(kernelEvent,
CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
```

```
                                cli_err_num   |=   clGetEventProfilingInfo(kernelEvent,
CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);

            if(cli_err_num != CL_SUCCESS)

            {

                printf(”Bład w odczycie informacji o profilowaniu!\n”);

            }

            executionTime = (end – start) * 1.0e-6f;

            printf(”Czas wykonania: %f sekundy.

\n”, executionTime);

        }

        clReleaseEvent(kernelEvent);

        kernelEvent= NULL;

    }

    clFinish(cl_compute_command_queue);

    origin[0] = 0; origin[1] = 0; origin[2] = 0;

    region[0] = WIDTH ; region[1] = HEIGHT; region[2] = 1;

    if(frame % 2 == 0)

    {

             cli_err_num  =  clEnqueueReadImage(cl_compute_command_queue,  output_image,
CL_TRUE, origin, region, 0, 0, (void*)out_bytes, 0, NULL, NULL);

    } else {

             cli_err_num  =  clEnqueueReadImage(cl_compute_command_queue,  input_image,
CL_TRUE, origin, region, 0, 0, (void*)out_bytes, 0, NULL, NULL);

    }

    clFinish(cl_compute_command_queue);

    if (cli_err_num != CL_SUCCESS)

    {

        printf(”Błąd: nie udało się odczytać obrazu! %d\n”, cli_err_num);

            return EXIT_FAILURE;

    }
```

```
    sprintf(&fileName[0],"image-%02d.ppm", frame);

    save_as_ppm( &fileName[0] );

}
```

Jeżeli siatka obliczeniowa zostanie poprawnie skonfigurowana, otrzymamy dość wydajną i krótką procedurę obliczeniową, która wykorzysta szybkość oferowaną przez technologię OpenCL. W rozdziale 7 przedstawiono krótką analizę wydajności tej procedury, a także sugestie, jakich poprawek należy dokonać, aby uzyskać dalsze zwiększenie wydajności.

5.7. Zbiór Mandelbrota

Algorytmy wizualizacji zbiorów fraktalnych można łatwo przenieść do środowiska równoległego, co oznacza też, że procedury rysujące osiągną znaczną wydajność, która umożliwi nie tylko szybkie narysowanie obrazu zbioru fraktalnego, ale też jego animację.

Zbiór fraktalny Mandelbrota (od jego odkrywcy Benito Mandelbrota) powstaje z jednej niezwykle krótkiej funkcji o następującej postaci:

$$f(z) = z^2 + c,$$

gdzie: z oraz c − liczby zespolone.

Jeżeli zapiszemy tę funkcję w nieco innej postaci:

$$z_{n+1} = z_n^2 + c \text{ dla } n \geq 0$$

widzimy, że jest to funkcja iteracyjna wymagająca powtarzania obliczeń w celu otrzymania nowej wartości zmiennej z. Algorytm, który służy do generacji obrazu zbioru fraktalnego, składa się z sześciu kroków:

(1) Start;

(2) zmienna c pełni rolę stałej i przyjmuje przekształcone współrzędne ekranowe;

(3) zmienna z w pierwszej iteracji przyjmuje wartość 0;

(4) realizacja iteracji, tj. obliczanie kolejnych wartości z: $z_{n+1} = z_n^2 + c$,

(5) krok (4) powtarzany jest tak długo aż:

> (a) wartość zmiennej z przekroczy z góry ustalaną wielkość (a dokładnie jest to promień okręgu o wartości dwa), co skutkuje przerwaniem realizacji kroku (4) i przejściem do (6); lub
>
> (b) liczba iteracji osiągnie założoną wartość, np. można założyć, że obliczenia w kroku (4) zostaną powtórzone maksymalnie 64 razy, po czym nastąpi przejście do (6);

(6) Stop.

Krok (4) powyższego algorytmu realizuje tzw. **funkcję ucieczki**, która sprawdza, czy wartość *z* przekracza założoną wartość normy liczby zespolonej z, tj $|z| = \sqrt{a^2 + b^2}$, gdzie *a* to część rzeczywista liczby *z*, natomiast wartość *b* to część urojona. Zamiast „norma” dość często używa się „wartość promienia” i „przekroczenie obszaru okręgu”. W wielu przykładach tworzenia zbioru Mandelborta sprawdza się, czy norma przekracza wartość 2:

$$\text{obliczaj } z_{n+1} = z_n^2 + c \text{ tak długo, jak } |z| < 2.$$

Norma liczby zespolonej *z* wymaga policzenia pierwiastka, które zajmuje cenny czas, dlatego lepiej warunek iteracji zapisać za pomocą wzoru:

$$\text{obliczaj } z_{n+1} = z_n^2 + c \text{ tak długo, jak } (a^2 + b^2) < 4,$$

w którym pominięto pierwiastek, ale za to zwiększono do 4 wartość, której nie możemy przekroczyć.

W wyniku realizacji funkcji ucieczki dla danych współrzędnych otrzymujemy wartość zespoloną *z* oraz numer iteracji. Wielkości te można wykorzystać do określenia koloru piksela, jaki postawimy w danym miejscu. Odpowiedni dobór funkcji kolorującej poszczególne piksele ma bardzo duże znaczenie dla ostatecznego efektu. Najprościej jest utworzyć obraz fraktala w szarościach (informacje o kolorowaniu fraktali omówionych w tym rozdziale podano w rozdz. 5.7.5).

5.7.1. Zbiór Julii

Drugim zbiorem fraktalnym, o którym zawsze się mówi w kontekście zbioru Mandelbrota, jest zbiór Julii. Sposób jego tworzenia jest bardzo podobny. Funkcja opisująca poszczególne iteracje jest pozornie identyczna:

$$z_{n+1} = z_n^2 + c \text{ dla } n \geq 0$$

jednak w tym przypadku stała *c* nie jest równa współrzędnym ekranowym (jak w zbiorze Mandelbrota), ale jest zawsze stała w trakcie obliczania wartości funkcji ucieczki, tj. wartości zmiennej z. Klasyczna wartość stałej *c* to:

$$c = -0.74543 + 0.11301i.$$

Zbiór Julii jest bardzo wrażliwy na wartość parametru *c*, dzięki czemu bardzo łatwo przygotować animację. Wystarczy uzależnić wartość *c* od czasu, aby otrzymać płynną animację zmiany kształtu.

5.7.2. Tworzenie obrazu zbioru Mandelbrota – wersja szeregowa

Obraz graficzny zbioru Mandelbrota można uzyskać, wykorzystując algorytm opisany na początku rozdz. 5.7. Wykonując obliczenia bezpośrednio, otrzymamy kod podobny do podanego w listingu 5.16. Uzyskane rozwiązanie będzie bardzo podobne do wielu procedur obliczeniowych, jakie można napotkać w internecie czy innych pozycjach książkowych.

Drugim bardzo popularnym rozwiązaniem jest stosowanie tylko jednej współrzędnej, na której podstawie wyznacza się współrzędne `x` oraz `y`. Na początek jednak wykorzystamy podejście oparte na dwóch współrzędnych, co ułatwi przeniesienie wersji szeregowej na poziom jądra obliczeniowego.

Procedura zaprezentowana w listingu 5.16 wymaga podania kilku dodatkowych informacji: wymiarów obrazu w pikselach (tj. wysokość `IMAGE_HEIGHT` oraz szerokość `IMAGE_WIDTH`), a także współrzędnych lewego górnego rogu obszaru, od którego rozpoczyna się generowanie obrazu fraktalnego. W przykładzie są to zmienne o nazwach `left_x` oraz `top_y`.

Istnieją też dwie pętle `for` generujące kolejne wartości tzw. współrzędnych ekranowych, czyli współrzędnych pikseli odnoszących się bezpośrednio do obrazu. Jednak nie są to współrzędne dla zbioru Mandelbrota, trzeba je bowiem dodać do lewego i górnego brzegu okna, dla którego tworzymy obraz. Należy także przemnożyć współrzędne ekranowe przez wartość kroku, który jest wyznaczany następująco:

```
step_x = 2.0 / WIDTH ;

step_y = 2.0 / HEIGHT;
```

Obliczenie właściwych współrzędnych nie jest już trudne:

```
picx = left_x + (x * step_x);

picy = top_y – (y * step_y);
```

Następnie oblicza się wartości funkcji ucieczki. Zmienna `maxIter` ogranicza maksymalną liczbę iteracji. Zmienne `zx`, `zy` to inaczej wartość zespolona `z`, która jest używana do wyznaczenia wartości funkcji ucieczki. Można się zastanawiać, dlaczego wyznaczamy nową wartość `zx` i `zy`, stosując dodatkowo wartość `ztmp`. Wynika to ze sposobu, w jaki oblicza się iloczyn dwóch liczb zespolonych, a w naszym przykładzie jest to kwadrat liczby `z`. Jeśli liczba `z` to:

```
z = zx + i *zy
```

gdzie: `i` − czynnik zespolony (kwadrat `z`):

$$z^2=(zx + i *zy)(zx + i *zy) = zx^2 + zy^2 + (zx*zy + zx*zy)*i$$

Dlatego wyznaczamy pomocniczą wartość `ztmp` oraz wcześniej kwadraty wartości `zx` oraz `zy`. Nowe wartości `zx`, `zy` są obliczane na podstawie `picx`, `picy` oraz obliczonego kwadratu liczby `z`. Pozostałe części procedury zależą od koloru piksela. Najłatwiej uzyskać obraz w szarościach, stosując tylko numer iteracji. Jeśli zmienna `maxIter` zostanie ograniczona do wartości `255`, to bardzo łatwo zbudować trójkę RGB, gdzie wszystkie składniki przyjmą aktualną wartość iteracji, i w ten sposób otrzymamy obraz zbioru Mandelbrota w szarościach.

Technicznym problemem jest postawienie piksela na ekranie. W listingu 5.16 zakładamy, że dostępna jest funkcja stawiająca piksel określonego koloru pod podanymi współrzędnymi:

```
set_pixel(x, y, color);
```

W technologii OpenCL nie stanowi to problemu, jeśli bowiem wykorzystamy możliwości w zakresie

przetwarzania obrazów, nie musimy się martwić technicznymi aspektami formatu obrazu czy ułożenia kolorów.

Listing 5.16. Wersja szeregowa procedury odpowiedzialnej za rysowanie zbioru Mandelbrota

```
float ztmp;

float picx, picy, zx, zy, zx_sqr, zy_sqr;

for (int y=0;y<IMAGE_HEIGHT;y++)

{

    for (int x=0;x<IMAGE_WIDTH;x++)

    {

        picx = left_x + (x * step_x);

        picy = top_y – (y * step_y);

        zx = 0;

        zy = 0;

        int i=0;

        for (i=0 ; i < maxIter ; i++)

        {

            zx_sqr = zx*zx;

            zy_sqr = zy*zy;

            if ( (zx_sqr + zy_sqr) > 4.0)

                break;

            ztmp = 2.0*zx*zy;

            zx = picx + (zx_sqr – zy_sqr);

            zy = picy + ztmp ;

        }

        unsigned int color = calculate_color(i, maxIter, zx, zy);

        set_pixel(x, y, color);

    }

}
```

5.7.3. Tworzenie obrazu zbioru Mandelbrota – wersja OpenCL

Kod szeregowy łatwo przekształcić na OpenCL, zwłaszcza gdy skorzystamy z możliwości przetwarzania obrazów. Dlatego w tym momencie zastosujemy możliwości OpenCL w tej dziedzinie (więcej informacji o przetwarzaniu obrazów podano w rozdz. 5.11).

OpenCL dostarcza typ `image2d_t` reprezentujący obraz dwuwymiarowy. Ponieważ fraktal również jest dwuwymiarowy, więc zastosowanie typu `image2d_t` uprości zadanie przeniesienia procedury z listingu 5.16 do środowiska OpenCL. Rolę dwóch pętli `for` z poprzedniego przykładu zastąpi siatka obliczeniowa, której wymiary będą odpowiadać liczbie pikseli składających się na obraz zbioru Mandelbrota. Pierwszą czynnością jest zatem odczytanie aktualnych identyfikatorów globalnych jednostki roboczej, która zajmuje się pojedynczym pikselem:

```
int x = get_global_id(0);

int y = get_global_id(1);
```

Następnie przygotowujemy pomocnicze zmienne w sposób podobny do wersji szeregowej. Również pętla `for` obliczająca wartość funkcji ucieczki została przeniesiona bezpośrednio z kodu szeregowego.

Różnica polega na zapisie piksela, który został obliczony przez wcześniejszy kod. Niemniej, dzięki możliwościom OpenCL, jest on realizowany zaledwie w dwóch liniach. W pierwszej tworzony jest wektor `uint4`, który zawiera cztery składowe piksela, tj. kolory RGB oraz kanał alfa (kanał przezroczystości).

```
uint4 pixel= (uint4)(it, it, it, 0);
```

Zapis piksela do obrazu wyjściowego jest bardzo prosty, korzystamy bowiem z gotowej funkcji `write_imageui`:

```
write_imageui(dst_image, (int2)(x, y), pixel);
```

gdzie: pierwszy parametr − zmienna reprezentująca obraz wyjściowy, następne − współrzędne piksela, trzeci parametr − wartość piksela.

Ponieważ każda jednostka robocza zajmuje się innym pikselem, uzyskamy dość duże przyspieszenie praktycznie bez większego wysiłku, bo listingi procedur obliczeniowych są prawie identyczne z kodem szeregowym. Co więcej, dzięki zastosowaniu typu `image2d_t`, zapis danych do zmiennej reprezentującej obraz został uproszczony, a funkcje OpenCL troszczą się o techniczne aspekty związane z formatem obrazu.

Czynności, jakie trzeba wykonać w API OpenCL, są takie same jak w poprzednich przykładach, dodatkowo trzeba tylko zarezerwować pamięć dla obrazu. Wymaga to określenia formatu obrazu; przyjmiemy tu format RGBA (trzy kolory RGB oraz kanał alfa[7]), a poszczególne składowe będą zapisywane jako liczby ośmiobitowe. Wymaga to następującej deklaracji zmiennej:

```
cl_image_format image_format;
```

a także wpisania odpowiednich stałych:

```
image_format.image_channel_data_type = CL_UNSIGNED_INT8;
```

```
image_format.image_channel_order = CL_RGBA;
```

Następnie można utworzyć bufor na obraz. Do przykładu wystarczy utworzyć tylko jeden bufor wyjściowy:

```
output_image = clCreateImage2D(cl_compute_context, CL_MEM_WRITE_ONLY, &image_format, WIDTH, HEIGHT, 0, NULL, &cli_err_num);
```

Czynności związane z budową jądra są podobne jak w poprzednich przykładach, choć w listingu 5.17 mamy stałe `WIDTH` oraz `HEIGHT`, których wartości podamy jako stałe preprocesora. W tym celu budowany jest ciąg znaków w zmiennej `preprocesorOptions` za pomocą `sprintf`:

```
char preprocesorOptions[512];

sprintf(preprocesorOptions, ”-D WIDTH=%d –D HEIGHT=%d”, WIDTH, HEIGHT);
```

a następnie ciąg ten jest podawany jako argument podczas budowy jądra obliczeniowego:

```
cli_err_num = clBuildProgram( cl_kernel_program, clui_num_devices, platform_devices, preprocesorOptions, NULL, NULL);
```

Parametr dla procedury obliczeniowej, czyli bufor wyjściowy, tworzymy podobnie jak poprzednio:

```
cli_err_num = clSetKernelArg(kernel, 0, sizeof(cl_mem), &output_image);
```

Uruchomienie procedury obliczeniowej wymaga przygotowania siatki obliczeniowej. Wykorzystamy tylko siatkę globalną. Wielkość siatki odpowiada liczbie pikseli, z których budujemy ostateczny obraz, mający strukturę dwuwymiarową. Dlatego zmienna opisująca siatkę globalną została zdefiniowana następująco:

```
size_t globalWorkSize[2];
```

Dwuelementowa tablica zawiera wysokość oraz szerokość obrazu (np. 512 ×512 punktów):

```
globalWorkSize[0]=HEIGHT;

globalWorkSize[1]=WIDTH;
```

Ostateczne uruchomienie procedury obliczeniowej sprowadza się do następującej linii kodu:

```
cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 2, NULL, globalWorkSize, NULL, 0, NULL, NULL );
```

Obraz zbioru Mandelbrota znajduje się w pamięci urządzenia, więc należy go przenieść do pamięci gospodarza, aby zapisać dane do pliku. Potrzebne są w tym celu dwie dodatkowe zmienne − współrzędne górnego rogu obrazu źródłowego (zmienna `origin`) oraz prawego dolnego rogu (zmienna `region`):

```
size_t origin[3], region[3];
```

W ten sposób po wpisaniu następujących wartości:

```
origin[0] = 0; origin[1] = 0; origin[2] = 0;
```

```
region[0] = WIDTH; region[1] = HEIGHT; region[2] = 1;
```

wyznaczony został prostokąt, stanowiący obszar, jaki zostanie przeniesiony do pamięci gospodarza. Podane wartości oznaczają cały obszar (dokładniejszy opis znaczenia zmiennych `origin` oraz `region` przedstawiono w rozdz. 5.9). Proces kopiowania jest realizowany za pomocą następującej linii kodu:

```
cli_err_num = clEnqueueReadImage(cl_compute_command_queue, output_image, CL_TRUE, origin, region, 0, 0, (void*)bytes, 0, NULL, NULL);
```

Zawartość obrazu z pamięci urządzenia zostanie umieszczona w obszarze wskazanym przez wskaźnik `bytes`. Można teraz zapisać dane do pliku, np. do formatu PPM (procedura zapisu została przedstawiona w Dodatku C):

```
save_as_ppm( ”image.ppm”);
```

Warto się zastanowić nad możliwością użycia siatki lokalnej. Pojedyncza jednostka robocza może wyznaczyć wartość więcej niż jednego piksela. Poszczególne fragmenty obrazu fraktalnego można umieścić w pamięci dzielonej. Następnie jeden z wątków może przenieść obliczone dane do pamięci globalnej.

Listing 5.17. Wersja OpenCL procedury odpowiedzialnej za rysowanie zbioru Mandelbrota wykorzystująca typ `image2d_t`

```
__kernel void imageKernel( write_only image2d_t dst_image)

{

    int x = get_global_id(0);

    int y = get_global_id(1);

    float left_x = – 1.5;

    float top_y = 1.0;

    float step_x = 2.0 / WIDTH;

    float step_y = 2.0 / HEIGHT;

    float atmp;

    float cx = left_x + x * step_x;

    float cy = top_y – y * step_y;

    float ax = 0;

    float ay = 0;

    unsigned int it;

    for (it=0;it<256;it++)

    {

        float ax2 = ax*ax;
```

```
        float ay2 = ay*ay;

        if (ax2 + ay2 > 4.0)

            break;

        atmp = 2.0*ax*ay;

        ax = ax2 – ay2 + cx;

        ay = atmp + cy;

    }

    uint4 pixel= (uint4)(it, it, it, 0);

    write_imageui(dst_image, (int2)(x, y), pixel);

}
```

5.7.4. Tworzenie obrazu zbioru Julii

Drugi typ obrazu fraktalnego, tj. zbiór Julii, jest tworzony przez bardzo podobną procedurę obliczeniową jak zbiór Mandelbrota. Pełny kod źródłowy jądra obliczeniowego został przedstawiony w listingu 5.18.

Główne zmiany dotyczą tylko sposobu, w jaki obliczana jest wartość funkcji ucieczki. Dodajemy do kwadratu liczby *z* stałą `c`, która ma wartość `–0.74543f + i0.11301f`. W kodzie realizują to następujące linie:

```
nar = ar * ar – ai * ai;

nai = ai * ar + ar * ai;

nar = nar + cr;

nai = nai + ci;
```

Inaczej wyznaczamy wartość kroku, np. dla współrzędnej `x`:

```
float step_x = 3.0 / WIDTH;
```

Zostały też zmienione nazwy zmiennych, jednak jest to zmiana tylko kosmetyczna. Pozostałe czynności, jakie wykonujemy po stronie hosta, są identyczne jak w przypadku zbioru Mandelbrota. Można zwyczajnie podmienić kod jądra obliczeniowego, bowiem wielkość siatki czy sposób kopiowania danych z pamięci urządzenia do pamięci gospodarza pozostają takie same.

Tworzony obraz zbioru Julii będzie kolorowany za pomocą numeru iteracji. Po zakończeniu pętli `for` wykonywane są dwie linie:

```
uint4 pixel= (uint4)(it, it, it, 0);

write_imageui(dst_image, (int2)(x, y), pixel);
```

W ten sposób uzyskamy obraz zbioru Julii w odcieniach szarości. Jeden z możliwych sposobów kolorowania został przedstawiony w rozdz. 5.7.5.

Listing 5.18. Procedura rysowania zbioru Julii − wersja w OpenCL

```
__kernel void imageKernel( write_only image2d_t dst_image )

{

    int x = get_global_id(0);

    int y = get_global_id(1);

    float left_x = – 1.5;

    float top_y = 1.5;

    float step_x = 3.0 / WIDTH;

    float step_y = 3.0 / HEIGHT;

    float cx = left_x + x * step_x;

    float cy = top_y – y * step_y;

    float cr = – 0.74543f;

    float ci = 0.11301f;

    float nar, ar = cx;

    float nai, ai = cy;

    unsigned int it;

    for (it=0;it<256;it++)

    {

        nar = ar * ar – ai * ai;

        nai = ai * ar + ar * ai;

        nar = nar + cr;

        nai = nai + ci;

        if (nar * nar + nai * nai > 4)

            break;

        ar = nar;

        ai = nai;

    }
```

```
    uint4 pixel= (uint4)(it, it, it, 0);

    write_imageui(dst_image, (int2)(x, y), pixel);

}
```

5.7.5. Kolorowanie zbioru fraktalnego

Do kolorowania zbioru fraktalnego można zastosować różne rozwiązania, np.:

− numer iteracji może kodować jeden konkretny kolor, skoro mamy ograniczoną liczbę iteracji,

− algorytm płynnego kolorowania − oprócz numeru iteracji, po którym przekroczono zadaną wartość, uwzględnia aktualną wartość zmiennej *z* przynależącą do zbioru fraktalnego (w internecie łatwo odszukać informacje o tym algorytmie),

− tworzenie histogramu numerów iteracji dla poszczególnych pikseli,

− optymalizacja kolorowania.

Utworzenie kolorowego obrazu zbioru Mandelbrota wymaga przygotowania odpowiedniej funkcji tworzącej opis koloru piksela, nie tylko za pomocą numeru iteracji, ale także przy wykorzystaniu współrzędnych. Kolor będzie kodowany w modelu HSV, a następnie konwertowany do trybu RGB.

Algorytm płynnego kolorowania można zastosować do zbioru Mandelbrota oraz Julii, wystarczy bowiem tuż po pętli, która realizuje funkcję ucieczki, odczytać interesujący nas numer iteracji, zmienną `it` oraz aktualne współrzędne `ax`, `ay` (wartość zmiennej `z`). Następnie trzeba zadeklarować dwie zmienne opisujące piksel obrazu oraz składowe RGB:

```
uint4 pixel;

int r, g, b;
```

Potem sprawdzamy, czy liczba iteracji jest większa od zera – jeśli nie, to zakłada się, że piksel jest czarny (wpisujemy do poszczególnych składowych zera):

```
pixel= (uint4)(0, 0, 0, 0);
```

Liczba iteracji większa od zera oznacza, że kolor piksela wyznacza następującey wzór:

$$|z| = \sqrt{a^2 + b^2},$$

mu = iteration – log(log(*z*))/log(2)

W naszym przypadku pominięto obliczanie pierwiastka kwadratowego, a numer iteracji to `it`. Po obliczeniu wartości `mu`, która jest stosowana do wyznaczenia wartości składowej odcienia w modelu HSV, obliczamy wartość `index` w następujący sposób:

```
float index = ( (0.95f + 10 * mu) );
```

Dwie pozostałe wielkości − jasność (ang. *brightness*) oraz nasycenie (ang. *saturation*, w skrócie sat) to stałe, które warto eksperymentalnie zmieniać, aby przekonać się, jak wypływają na ostateczną postać obrazu.

Zamiana składowych HSV na kolory w trybie RGB wykonywana jest za pomocą funkcji `hsv2rgb` (patrz Dodatek C). Wcześniej zostały zdefiniowane zmienne `pixel` oraz składowe kolorów `r`, `g`, `b`.

```
uint4 pixel;

int r, g, b;
```

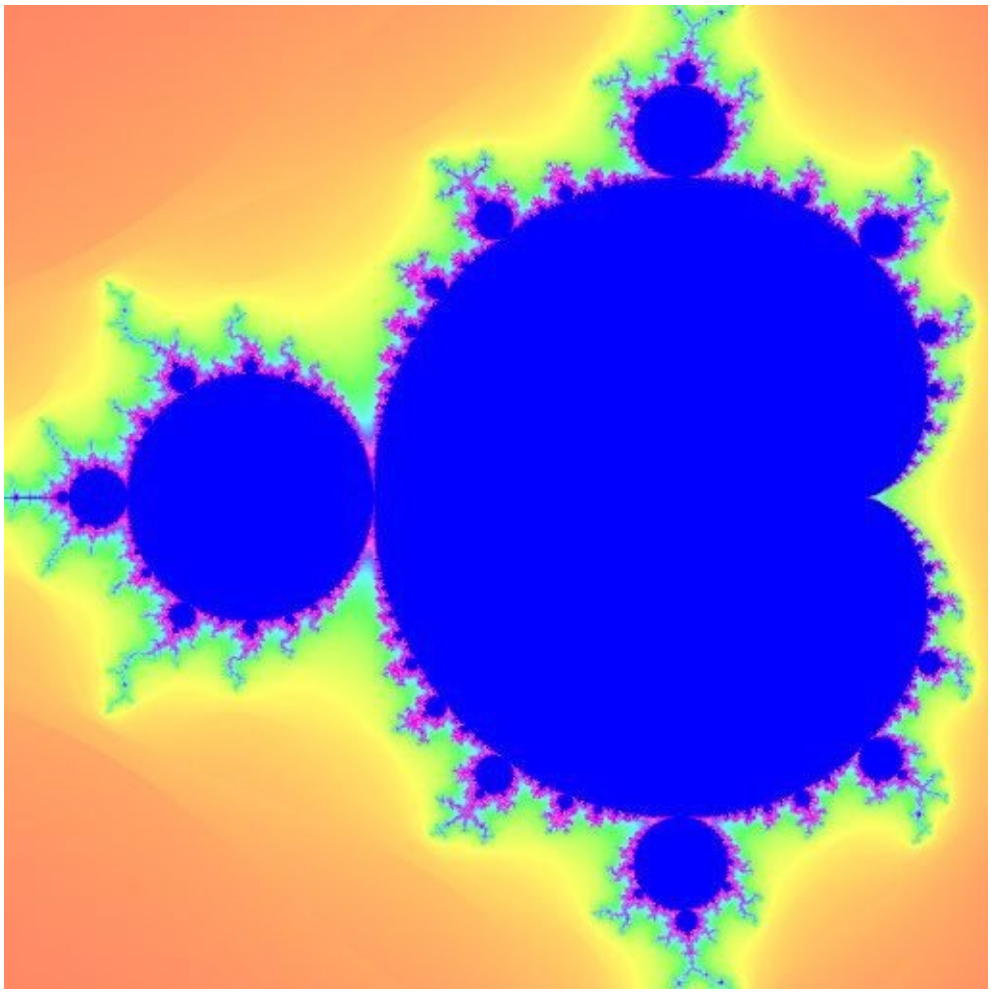
Zamiana składowych HSB na kolory RGB i wpisanie obliczonych wartości do zmiennej `pixel` realizują dwie następujące linie:

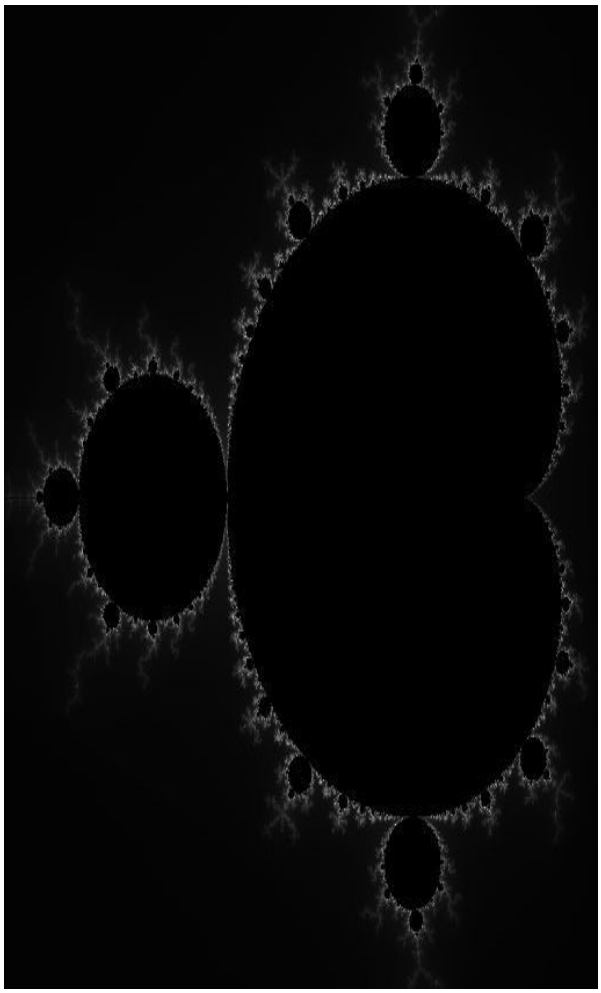
```
hsv2rgb(index, sat, bright, &r,&g,&b );

pixel= (uint4)(r, g, b, 0);
```

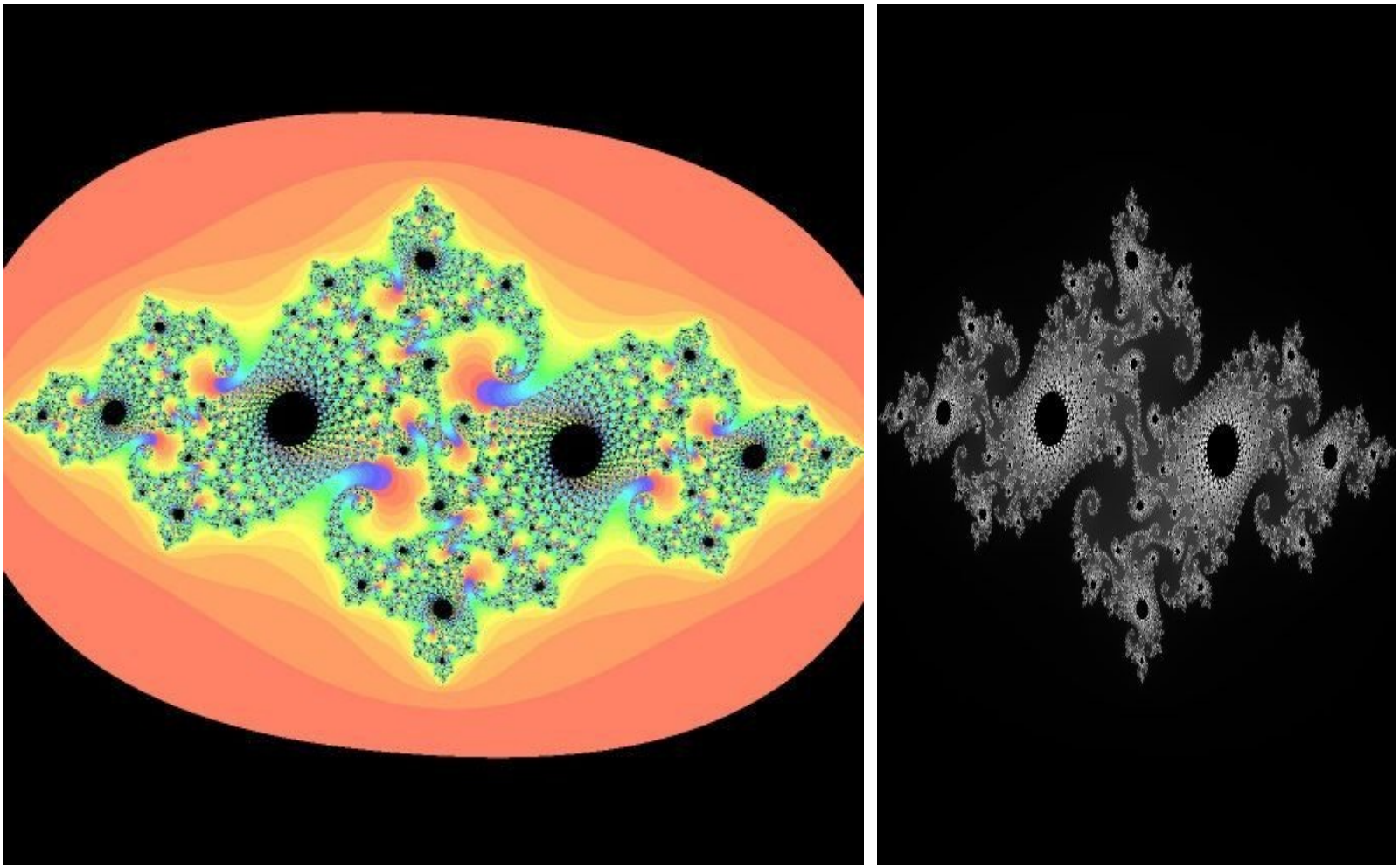
Następnie można wyznaczony kolor piksela zapisać do obrazu wyjściowego za pomocą kolejnej linii kodu:

```
write_imageui(dst_image, (int2)(ix, iy), pixel);
```

Obrazy utworzone w trybach szarości oraz techniką płynnego kolorowania zostały pokazane na rys. 5.7.







Rys. 5.7. Przykładowe rysunki zbiorów fraktalnych Julii oraz Mandelbrota w szarościach oraz po zastosowaniu płynnego kolorowania (wersja kolorowa dostępna w wydaniu elektronicznym)

Listing 5.19. Kolorowanie zbioru Mandelbrota oraz Julii

```
if (it > 0) {

    float mu = it – log(log(ax*ax + ay*ay)) / log(2.0f);

    float index = ( (0.95f + 10 * mu) );

    float sat = (0.6f);

    float bright = (1.0f);

    hsv2rgb(index, sat, bright, &r,&g,&b );

    pixel= (uint4)(r, g, b, 0);

}

else

    pixel= (uint4)(0, 0, 0, 0);

write_imageui(dst_image, (int2)(ix, iy), pixel);
```

5.8. Algorytm sortowania bitonicznego

Do implementacji w ramach OpenCL nie nadają się algorytmy sortowania wartości liczbowych zaprojektowane do pracy w środowisku wyposażonym w jeden procesor. Są to np. sortowanie szybkie, sortowanie bąbelkowe czy sortowanie przez wstawianie. Powód jest bardzo prozaiczny – nie są one przystosowane do pracy w środowisku równoległym. Jest to bezpośrednia konsekwencja faktu, iż są to rozwiązania zaprojektowane pierwotnie do pracy w środowisku wyposażonym w jeden procesor.

W przypadku OpenCL i modelu programowania, jaki on oferuje, należy stosować algorytmy, które mogą wykorzystać wiele dostępnych jednostek przetwarzających. Dużą wydajność możemy uzyskać za pomocą tzw. sieci sortujących, np. algorytm sortowania bitonicznego pozwoli osiągnąć wysoką wydajność, przy czym sam nie jest zbyt skomplikowany do implementacji.

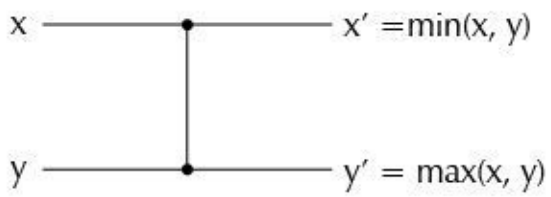
5.8.1. Sieci sortujące

Podstawowym elementem każdej sieci sortującej jest **komparator**, tj. element, który porównuje dwie wartości wejściowe. Komparator ma dwa wejścia (x oraz y) oraz dwa wyjścia (x' oraz y'). Na rysunku 5.8 przedstawiono graficzną interpretację komparatora.

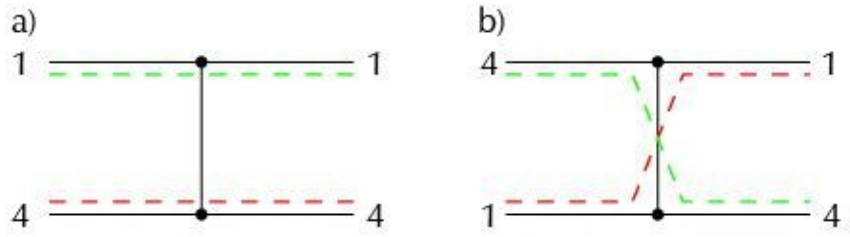
Zasada działania komparatora polega na tym, że na wyjście x' zawsze jest przekazywana wartość mniejsza z pary x i y, natomiast na wyjście y' – wartość większa. Opisane zachowanie można przedstawić za pomocą funkcji `max` oraz `min`, które zostały użyte do opisu wyjść komparatora na rys. 5.8.

Sposób działania komparatora przedstawiono także na rys. 5.9. Widzimy dwa przypadki, gdy na wejście x i y podano wartości 1 i 4. W przypadku a) elementy są przekazywane wprost na wyjście bez dodatkowych zmian, a w przypadku b) elementy zostały zamienione zgodnie z zasadą działania komparatora.

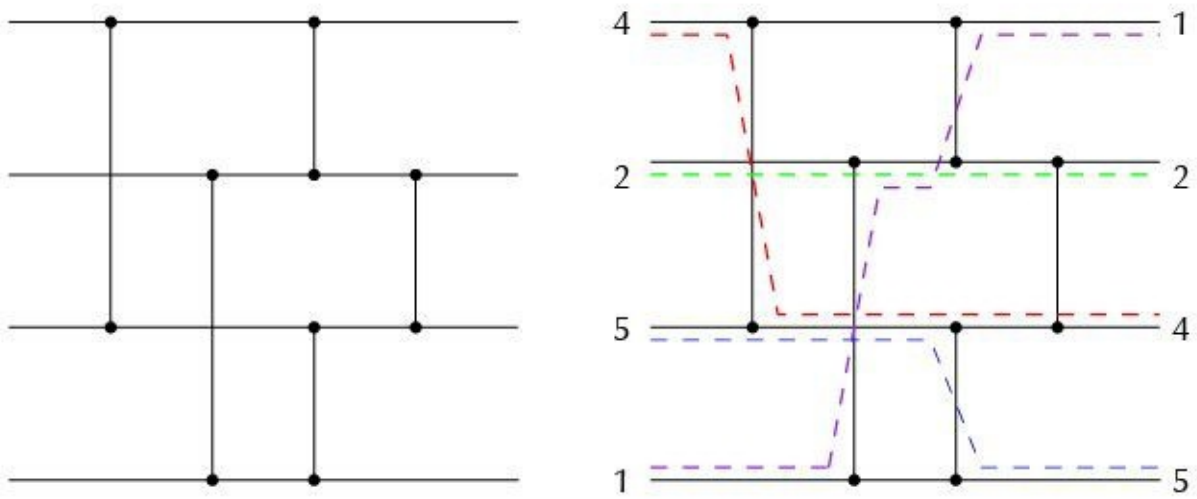
Można określić dwa podstawowe typy komparatorów w zależności od tego, czy na wyjście x' wstawiany jest mniejszy element (jak na rys. 5.9) czy większy. Typ komparatora użyty do budowy sieci sortującej determinuje, czy będziemy sortować ciąg w porządku rosnącym, czy malejącym.



Rys. 5.8. Graficzna interpretacja zasady działania komparatora



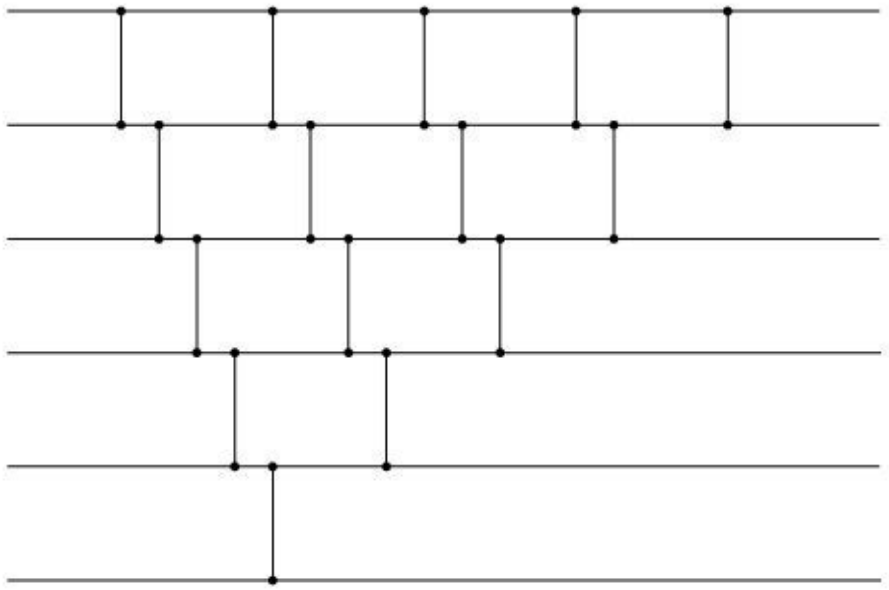
Rys. 5.9. Działanie komparatora dla przykładowych wartości: a) na wejście x podano mniejszy element, b) na wejście x podano większy element



Rys. 5.10. Przykładowa sieć sortująca dla 4 elementów

Przykład sieci sortującej dla czterech elementów przedstawiono na rys. 5.10. W sieci mamy pięć komparatorów, przy czym dwa komparatory w trzeciej kolumnie mogą funkcjonować równolegle niezależnie od siebie. Podobnie dwa pierwsze również mogą funkcjonować równolegle. Jest to podstawowa zaleta sieci sortujących, dzięki której względnie łatwo zaimplementować ją w środowisku równoległym, gdzie mamy do dyspozycji wiele niezależnych od siebie jednostek obliczeniowych.

Nie każdą sieć sortującą można efektywnie zaimplementować w środowisku równoległym. Przykładem może być sieć bazująca na algorytmie bąbelkowym. Jej interpretację graficzną pokazano na rys. 5.11.



Rys. 5.11. Przykładowa sieć sortująca opierająca się na algorytmie sortowania bąbelkowego

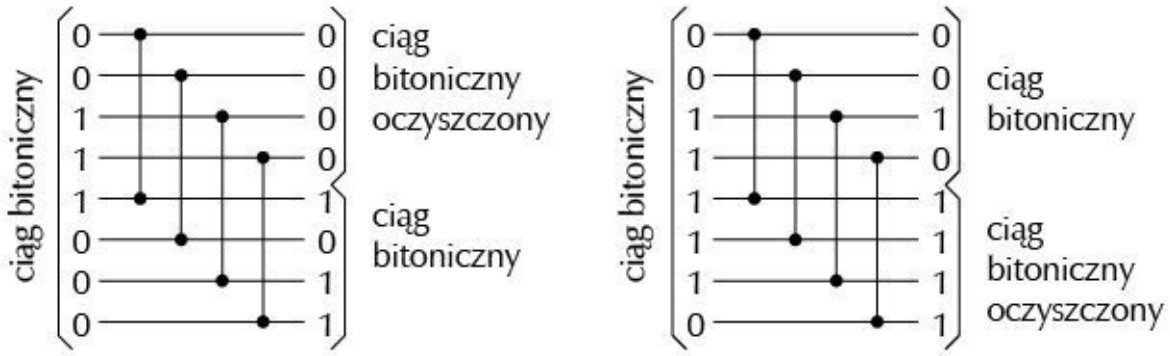
Nie można tu wskazać zbioru komparatorów mogących działać równolegle. Nie zamyka to jednak drogi do otrzymania wydajnej sieci sortującej. Sieci sortujące bitoniczne oraz scalające są przystosowane do implementacji w środowisku równoległym i w środowisku OpenCL.

5.8.2. Bitoniczna sieć sortująca

Bitoniczna sieć sortująca jest bardzo często wybierana do projektowania algorytmów sortujących, które mają pracować w środowisku równoległym wykorzystującym wiele jednostek obliczeniowych. Oryginalna sieć bitoniczna sortuje poprawnie tylko ciągi bitoniczne, czyli takie, których elementy są ułożone najpierw w sposób nierosnący, a potem niemalejący, bądź najpierw w sposób niemalejący, a następnie nierosnący. Algorytm sortujący musi być tak skonstruowany, aby bitoniczna sieć sortująca była zdolna do sortowania dowolnego ciągu.

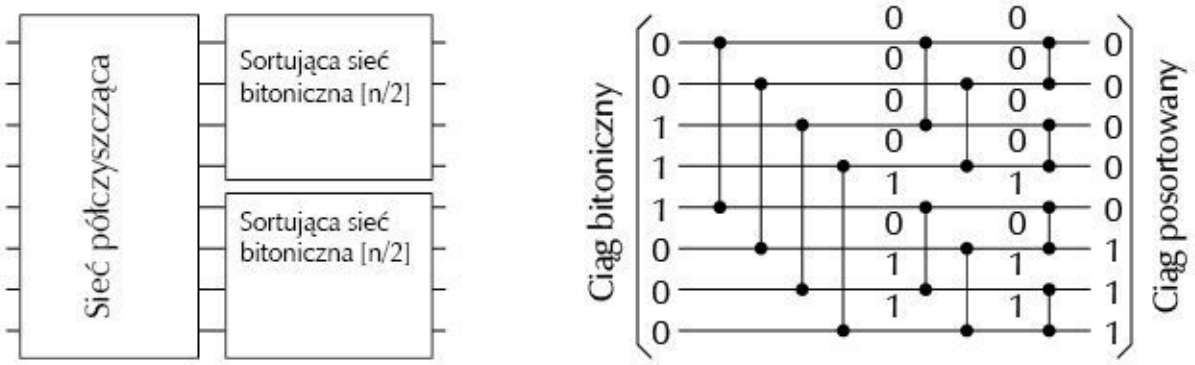
Do konstrukcji uniwersalnej sieci sortującej wykorzystamy tzw. **lemat o sieciach półczyszczących** [Cormen2012]. Zasada działania sieci półczyszczącej jest następująca: jeśli na wejście podamy ciąg bitoniczny (np. ciąg zer i jedynek), to ciąg wyjściowy ma następujące własności: obie połowy – górna i dolna – są bitoniczne, każdy element w górnej (lub w dolnej) połowie nie będzie większy niż dowolny element w dolnej (lub w górnej) połowie oraz przynajmniej jedna połowa jest oczyszczona.

Zasadę działania sieci półczyszczącej pokazano na rys. 5.12. Jak widać, wejściowy ciąg bitoniczny został podzielony na ciąg oczyszczony zawierający same zera lub jedynki oraz podciąg bitoniczny.



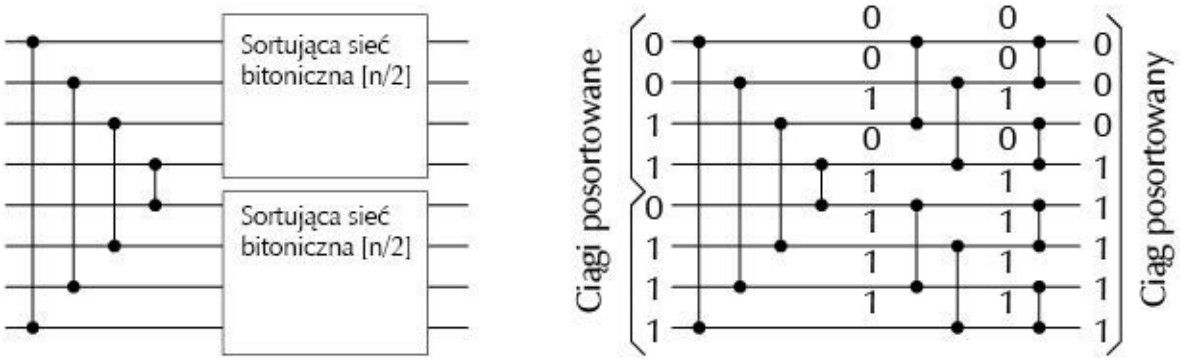
Rys. 5.12. Zasada działania sieci półczyszczącej

Do skonstruowania sieci sortującej ciągi bitoniczne stosujemy **rekurencyjnie sieci półczyszczące**. Pierwsze użycie sieci półczyszczącej dotyczy całego zbioru. Następnie będziemy ją stosować niezależnie dla górnej i dolnej połowy zbioru (górna czy dolna połowa zbioru to określenia umowne – można mówić też o lewej i prawej połowie sortowanego zbioru). Kolejne rekurencyjne wywołania dzielą aktualnie przetwarzany zbiór danych ponownie na połowę, tak długo, aż sortowane będą tylko pary elementów. Sugeruje to także wielkość zbioru, która musi być liczbą parzystą, a ściślej mówiąc, wartością równą odpowiedniej potędze liczby dwa. Na rysunku 5.13 przedstawiono rekurencyjną konstrukcję bitonicznej sieci sortującej.



Rys. 5.13. Rekurencyjna konstrukcja bitonicznej sieci sortującej

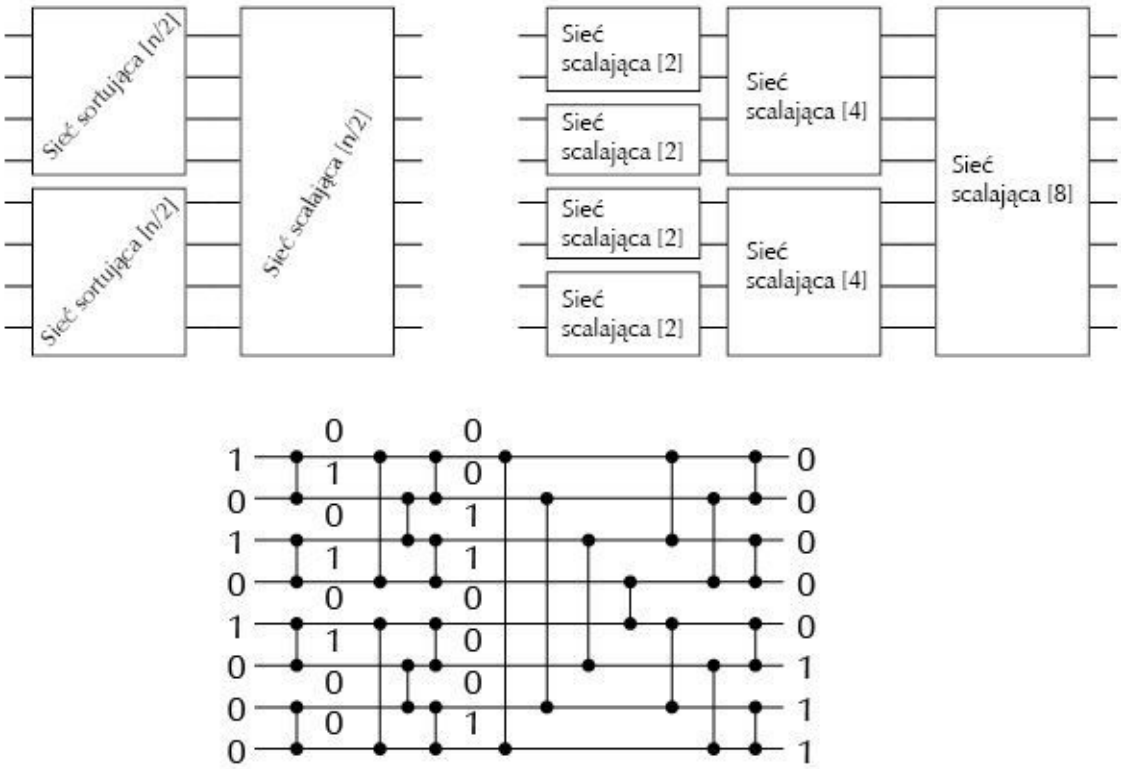
Drugim krokiem do uzyskania uniwersalnej sieci sortującej jest wprowadzenie operacji **scalania**, aby z dwóch podciągów posortowanych podanych na wejście, otrzymać jeden ciąg scalony z dwóch podciągów. Na rysunku 5.14 przedstawiono tak zbudowaną sieć. Jak widać, mimo że na wejście podano dwa ciągi posortowane po etapie scalania, na wyjściu nie otrzymano ciągu posortowanego. Możliwe jest natomiast podzielenie całego zbioru na dwie w pełni niezależne części, tak aby sortując bitonicznie każdą część otrzymać w pełni posortowany zbiór. Procedury sortownia bitonicznego mogą w tym przypadku działać równolegle. Na rysunku 5.14 widać, że proces scalania również może być wykonywany równolegle.



Rys. 5.14. Schemat działania sieci sortującej − pierwszy etap to operacja scalania, drugi to sortowanie niezależne obu części zbioru przez dwie bitoniczne sieci sortujące

Budowa uniwersalnej sieci sortującej wymaga definicji rekurencyjnej, aby można było ją wielokrotnie zastosować (rys. 5.14), począwszy od par elementów, aż do całego zbioru danych na samym końcu procesu sortowania. Zasadę działania uniwersalnej sieci sortującej pokazano w sposób graficzny na rys. 5.15.

Uniwersalna sieć sortująca może zostać efektywnie zaimplementowana w środowisku równoległym. Na rysunku 5.15 pokazano, że w każdym etapie sortowania wiele komparatorów może funkcjonować równolegle, wykonując swoje działanie niezależnie od siebie. Skraca to znacznie czas działania algorytmu, jednak wysoka wydajność jest możliwa, gdy mamy do dyspozycji wiele jednostek przetwarzających. Tak jest w przypadku kart graficznych, więc implementacja sortowania za pomocą sieci w przypadku OpenCL w efekcie da nam wysoką wydajność.



Rys. 5.15. Zasada działania uniwersalnej sieci sortującej stosowanej rekurencyjnie na poszczególnych fragmentach zbioru danych

Więcej informacji o sieciach sortujących, w tym obszerną dyskusję o poprawności sortowania bitonicznego, a także szczegółowe omówienie aspektów teoretycznych, w tym złożoności

obliczeniowej, można znaleźć w książce [Cormen2012].

5.8.3. Szeregowa wersja algorytmu sortowania bitonicznego

Algorytm sortowania bitonicznego można implementować w wersji szeregowej, więc wykorzystajmy tę możliwość, aby lepiej go poznać. Listing 5.20 przedstawia główną procedurę sortującą o nazwie `bitonic_sort`. Mamy trzy parametry, pierwszy `s` oznacza indeks elementu, od którego rozpoczniemy sortowanie. Przy pierwszym uruchomieniu procedury sortującej będziemy podawać wartość zero, czyli od początku tablicy. Sortować będziemy tablicę globalną o nazwie `a`, którą zdefiniujemy w następujący sposób:

```
#define N 32

int a[N];
```

Drugi parametr głównej procedury o nazwie `ncount` określa liczbę elementów. W naszym przypadku dla uproszczenia przyjmiemy, że będzie to zawsze wielkość podzielna przez dwa, inaczej mówiąc liczba elementów będzie parzysta, a dokładnie będzie potęgą dwójki. Istotną rolę pełni także trzeci argument − określa, w jakim porządku ma się odbywać sortowanie − jeśli w porządku rosnącym, należy podać stałą `ASCENDING_ORDER`, a jeśli w malejącym − `DESCENDING_ORDER`.

Listing 5.20. Sortowanie bitoniczne − procedura główna

```
void bitonic_sort(int s, int ncount, int direction)

{

    if ( ncount > 1 )

    {

        int k = ncount / 2;

        bitonic_sort(s, k, ASCENDING_ORDER);

        bitonic_sort(s+k, k, DESCENDING_ORDER);

        bitonic_merge(s, ncount, direction);

    }

}
```

Treść głównej procedury przedstawiono graficznie na rys. 5.15. Sprawdzamy, czy sortujemy więcej niż jeden element (co jest konsekwencją rekurencji, którą zaraz wyjaśnimy). Pierwszą istotną czynnością jest podzielenie liczby elementów przez dwa. Zgodnie z rys. 5.15 najpierw sortujemy elementy od `s` do `k`, inaczej mówiąc, od zera do połowy, a potem za pomocą drugiej procedury sortujemy drugą część zbioru danych. Następnie uruchamiamy proces scalania. Jak widać na listingu 5.20, procedury sortujące sortują jedną połowę zbioru rosnąco, a drugą malejąco.

Procedury sortujące `bitonic_sort` mogą zostać uruchomione równolegle, jednak kolejna procedura scalania musi poczekać, aż poprzedni proces sortowania zostanie zakończony. Procedura `bitonic_sort` jest procedurą rekurencyjną − wyznaczamy zbiór elementów, które należy posortować w odpowiedniej kolejności, ale główne zadanie sortowania będzie się odbywać w procedurze scalającej `bitonic_merge`, którą teraz krótko przedstawimy.

Całość procedury scalającej przedstawiono na listingu 5.21. Jest to procedura rekurencyjna, której główne zadanie wykonywane w pętli `for` polega na wywołaniu `k` komparatorów działających zawsze na parze wskazanej przez `i` oraz `i+k`. Gdzie `k`, analogicznie jak w poprzedniej procedurze, oznacza połowę pierwotnej wielkości zbioru, który sortujemy. Podobnie jak poprzednio, sprawdzamy, czy zbiór zawiera więcej niż jeden element, aby można było wywołać przynajmniej jeden komparator dla pary elementów.

Listing 5.21. Sortowanie bitoniczne − procedura scalania

```
void bitonic_merge(int s, int count, int direction)

{

    if (count > 1)

    {

        int k = count / 2;

        int i;

        for (i=s; i<s+k; i++)

            comparator(i, i+k, direction);

        bitonic_merge(s, k, direction);

        bitonic_merge(s+k, k, direction);

    }

}
```

Jak widać, to komparator przedstawiony w listingu 5.22 wykonuje główne zadanie − porównuje dwa elementy i, jeśli rezultatem warunku w instrukcji `if` będzie prawda, to elementy zostaną przestawione. Ta specyficzna postać warunku `if` pozwala w dość „sprytny” sposób zapisać operację testującą, czy mamy przestawić elementy, jeśli nie są zgodne z aktualnym kierunkiem sortowania. Wymaga to naturalnie odpowiedniego zdefiniowania wartości określających kierunek sortowania:

```
const int ASCENDING_ORDER = 1;

const int DESCENDING_ORDER = 0;
```

Ponieważ nasz program do sortowania bitonicznego jest tworzony w języku C, to można wykorzystać fakt, że wartość zero oznacza fałsz, a jedność prawdę (w języku C wszystkie wartości różne od zera są traktowane jako prawda). Sortując dany zbiór np. rosnąco, komparator otrzymuje

w parametrze `direction` wartość jeden, a jeśli wynik testu `a[i] > a[j]` jest prawdziwy, to również wynik testu dla całej instrukcji warunkowej `direction == (a[i] > a[j])` także jest prawdziwy, i elementy zostaną zamienione miejscami zgodnie z zasadą działania komparatora (i-ty element jest mniejszy niż j-ty element). Taki sposób działania jest konsekwencją sposobu działania sieci sortującej.

Listing 5.22. Sortowanie bitoniczne – implementacja komparatora

```
void comparator(int i, int j, int direction)

{

    int tmp;

    if ( direction == (a[i] > a[j]) )

    {

        tmp=a[i];

        a[i]=a[j];

        a[j]=tmp;

    }

}
```

Jeżeli sortujemy rosnąco, kierunek jest oznaczony jednością. Jeśli zatem test `a[i] > a[j]` daje w wyniku prawdę, to zamienimy elementy miejscami, bowiem pierwszy element był większy niż drugi, a nas interesuje kolejność rosnąca.

Przedstawione trzy procedury, w tym test w instrukcji `if`, umożliwiają dość zwartą implementację sieci sortującej w środowisku szeregowym − zaledwie trzy dość krótkie funkcje. Przygotowanie wydajnej wersji algorytmu sortującego w OpenCL omówiono w rozdz. 5.8.4.

5.8.4. Równoległa wersja algorytmu sortowania bitonicznego dla OpenCL

W sortowaniu bitonicznym jądro obliczeniowe należy uruchamiać wielokrotnie, aby otrzymać posortowany zbiór danych. Procedura obliczeniowa jest dość krótka, co widać w listingu 5.23.

Sam proces przestawienia elementów realizuje instrukcja `if`, która została podana pod koniec procedury obliczeniowej `bitonic_stage`. Wykorzystuje ona to samo rozwiązanie co procedura szeregowa, aby skrócić kod źródłowy. Istotne są początkowe definicje identyfikatorów oraz zmiennych. Element wiodący jest oznaczany jako identyfikator `GID`, jednak w zależności od etapu sortowania, określa się też kierunek sortowania. Poprzednia procedura szeregowa zmieniała kierunek sortowania w kolejnych wywołaniach rekurencyjnych. Należy też wyznaczyć odległość pomiędzy elementami, jakie będą porównywane:

```
int pair_dist = 1 << (stage – pass);
```

Za pomocą tej wielkości należy wskazać, jakie elementy będziemy porównywać, obliczając: `left_id`

oraz right_id. Oznacza to, że GID nie musi wskazywać elementu, jaki będziemy sortować, tylko raczej element pomocniczy, wskazujący element, na którym trzeba działać na danym etapie sortowania bitonicznego, zgodnie z ideą przedstawioną na rys. 5.15.

Listing 5.23. Procedura obliczeniowa realizująca jeden etap sortowania bitonicznego

```
__kernel void bitonic_stage(__global uint *data,
        const uint stage,
        const uint pass,
        const uint _direction)
{
    uint GID = get_global_id(0);
    uint direction = _direction;
    uint pair_dist = 1 << (stage – pass);

    uint left_id = (GID % pair_dist) + (GID / pair_dist) * (2 * pair_dist);
    uint right_id = left_id + pair_dist;
    uint block_width = 1 << stage;

    if((GID/block_width) % 2 == 1)
        direction = 1 – direction;

    uint tmp;

    if ( direction == (data[left_id] > data[right_id]) )
    {
        tmp = data[left_id];
        data[left_id] = data[right_id];
        data[right_id] = tmp;
    }
}
```

Kluczowy jest sposób wywołania jądra obliczeniowego. Zakładając, że mamy do sortowania szesnaście elementów umieszczonych w zmiennej `in_data_dev`, należy określić kierunek sortowania, np. rosnący:

```
orderASC = 1
```

Zakładamy wobec tego liczbę elementów `totalN` i wielkość siatki lokalnej `localN`. Dla szesnastu elementów zmienne te przyjmują następujące wartości:

```
totalN = 16

localN = 4
```

Aby poprawić czytelność, wywołanie jądra obliczeniowego zapiszemy w pseudokodzie. Liczba elementów wymaga, aby procedurę wywołać w czterech etapach numerowanych 0, 1, 2, 3. W każdym etapie wykonujemy dodatkowe przebiegi − w pierwszym tylko jeden, w drugim dwa i tak dalej. Daje to następujący ciąg wywołań sortowania bitonicznego:

```
bitonic_stage(q, [totalN], [localN], in_data_dev, 0, 0, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 1, 0, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 1, 1, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 2, 0, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 2, 1, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 2, 2, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 3, 0, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 3, 1, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 3, 2, orderASC )

bitonic_stage(q, [totalN], [localN], in_data_dev, 3, 3, orderASC )
```

Jak widać, jest to dość duża liczba, ale liczba wywołań ogólnie zależy logarytmicznie od liczby sortowanych elementów, dlatego w przypadku większych ilości danych do sortowania końcowy czas będzie znacznie krótszy niż w wersji szeregowej.

5.9. Przetwarzanie obrazów

Standard OpenCL oferuje ułatwione przetwarzanie obrazów. Dzięki obecności typów `image1d_t`, `image2d_t` czy `image3d_t` oraz dodatkowego obiektu, tzw. **samplera** (obiektu próbkującego) w dość prosty sposób można zrealizować implementacje wielu algorytmów do przetwarzania obrazu, np. filtrów wyostrzających czy rozmywających obraz. W dalszych punktach tego rozdziału przedstawiono rozwiązania podstawowych problemów dotyczących przetwarzania obrazu, jak skalowanie obrazu, stosowanie filtrów, a także algorytm wykrywania krawędzi. W tym ostatnim przypadku dane o obrazie są przekazywane bez pośrednictwa typów dedykowanych dla obrazów.

5.9.1. Podstawy – wypełnienie obrazu

Pierwszy przykład polega na nałożeniu gradientu na istniejący obraz. Na podstawie tego przykładu pokażemy podstawowe elementy OpenCL, jakie wykorzystuje się w przetwarzaniu obrazów.

Pierwszy problem, jaki pojawia się nawet w najprostszym zadaniu, to wybór formatu przechowywania informacji o obrazie. W tym przykładzie będziemy stosować format RGBA[8], czyli trójkę kolorów, tj. czerwony, zielony, niebieski oraz kanał alfa. Wymaga to naturalnie przygotowania odpowiednich buforów danych, dla których trzeba określić format obrazu. W tym celu stosowany jest typ `cl_image_format`

```
cl_image_format image_format;
```

Typ `cl_image_format` ma dwa pola. W przykładzie poszczególne kanały kolorów oraz kanał alfa są opisywane przez liczby całkowite oraz układ kanałów − wspominany już układ RGBA:

```
image_format.image_channel_data_type = CL_UNSIGNED_INT8;

image_format.image_channel_order = CL_RGBA;
```

Ponieważ mamy bufor wejściowy oraz wyjściowy, to potrzebne są dwa obiekty do buforów:

```
cl_mem input_image;

cl_mem output_image;
```

Utworzenie obiektów reprezentujących obrazy to zadanie funkcji `clCreateImage2D`, gdzie z punktu widzenia przetwarzanego obrazu najważniejsze parametry to format oraz szerokość (parametr `WIDTH`) i wysokość (`HEIGHT`) obrazu. W przypadku obrazu wejściowego:

```
input_image = clCreateImage2D(cl_compute_context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, &image_format, WIDTH, HEIGHT, 0, (void*)bytes, &cli_err_num);
```

dodatkowo podana została zmienna `bytes`, z której kopiowane są dane do bufora wejściowego. Należy podkreślić, iż dane są kopiowane ze zmiennej umieszczonej w pamięci gospodarza do pamięci urządzenia obliczeniowego.

Zmienna reprezentująca bufor wyjściowy jest tworzona w następujący sposób:

```
output_image = clCreateImage2D( cl_compute_context, CL_MEM_WRITE_ONLY, &image_format, WIDTH, HEIGHT, 0, NULL, &cli_err_num );
```

Poszczególne etapy przygotowawcze do uruchomienia jądra obliczeniowego są podobne jak w innych przykładach. Jądro obliczeniowe z listingu 5.24 używa dwóch parametrów, w których należy podać obiekty reprezentujące bufor wejściowy oraz wyjściowy:

```
cli_err_num = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_image);

cli_err_num = clSetKernelArg(kernel, 1, sizeof(cl_mem), &output_image);
```

Po ustaleniu parametrów procedury obliczeniowej należy określić postać siatki obliczeniowej.

Ponieważ przetwarzany będzie obraz dwuwymiarowy, to deklaracja zmiennej opisującej siatkę jest następująca:

```
size_t globalWorkSize[2];
```

Będzie ona zawierać wymiary tworzonego obrazu, dlatego umieszczamy tu informacje o szerokości i wysokości:

```
globalWorkSize[0]=HEIGHT;

globalWorkSize[1]=WIDTH;
```

W tym momencie można uruchomić jądro obliczeniowe za pomocą `clEnqueueNDRangeKernel` w następujący sposób:

```
cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 2, NULL, globalWorkSize, NULL, 0, NULL, NULL );
```

Sposób, w jaki realizujemy nałożenie gradientu na istniejący obraz, został zawarty w kodzie procedury obliczeniowej pokazanej w listingu 5.24, jednak opis zostanie przedstawiony później, bowiem po wykonaniu obliczeń przez jądro obliczeniowe, należy odebrać dane. Wymaga to zdefiniowania dwóch dodatkowych zmiennych, w których podaje się współrzędne źródłowe (wskazujące na początek obrazu) `origin` oraz współrzędne docelowe `region`:

```
size_t origin[3], region[3];
```

Wpisanie wartości określających początek obrazu w przypadku źródła przedstawia się następująco:

```
origin[0] = 0; origin[1] = 0; origin[2] = 0;
```

Tradycyjnie początek jest umieszczony we współrzędnych (0, 0), ale mamy też trzeci parametr, który jest stosowany dla obrazów trójwymiarowych. W przykładzie zajmujemy się obrazem dwuwymiarowym, więc trzeci parametr przyjmuje wartość zero.

Dla zmiennej `region` dwa pierwsze parametry to lewy dolny wierzchołek obrazu, trzeci parametr znajduje zastosowanie w przypadku obrazu trójwymiarowego. Jednak, jeśli `region` odnosi się do obrazu dwuwymiarowego, standard OpenCL wymaga, aby była wstawiona tam wartość jeden:

```
region[0] = WIDTH; region[1] = HEIGHT; region[2] = 1;
```

Ostatecznie odczyt danych z przetworzonego obrazu do zmiennej `bytes` (umieszczonej w pamięci gospodarza) zrealizujemy w następujący sposób:

```
cli_err_num = clEnqueueReadImage(cl_compute_command_queue, output_image, CL_TRUE, origin, region, 0, 0, (void*)bytes, 0, NULL, NULL);
```

Wartość `CL_TRUE` umieszczona w trzecim parametrze oznacza, że proces kopiowania będzie tzw. procesem blokowanym, inaczej mówiąc, funkcja `clEnqueueReadImage` nie zwróci sterowania do następnej instrukcji do czasu zakończenia procesu kopiowania danych. Naturalnie można to zmienić za pomocą odpowiednio zbudowanej procedury do obsługi zdarzeń.

Po omówieniu niezbędnych czynności, jakie należy wykonać, aby uruchomić jądro obliczeniowe, możemy omówić samo jądro obliczeniowe. Funkcja `imageKernel` przyjmuje dwa argumenty: `src_image` − reprezentuje obraz wejściowy oraz `dst_image` − obraz wyjściowy.

Bardzo istotną rolę ogrywa deklaracja tzw. samplera (obiekt `sampler`). Obiekt ten opisuje sposób, w jaki odczytywane są piksele z obrazu wejściowego. Listing 5.24 definiuje `sampler` jako stałą. Poszczególne flagi, jakie zostały użyte, mają następujące znaczenie: `CLK_NORMALIZED_COORDS_FALSE` oznacza, że współrzędne poszczególnych pikseli obrazu nie będą znormalizowane, co pozwala zapisać wartości współrzędnych np. od 0 do 768. Normalizacja oznacza, że będą stosowane współrzędne od zera do jedności.

Druga zastosowana flaga `CLK_ADDRESS_CLAMP` oznacza, iż wszystkie piksele odczytywane spoza zakresu obrazu będą posiadały wartości pikseli znajdujących się na brzegu oryginalnego obrazu.

Ostatnia, trzecia flaga dotyczy sposobu odczytywania pikseli. Dla wartości `CLK_FILTER_NEAREST` odczyt pikseli polega na stosowaniu tzw. filtrowania bezpośredniego. Łatwo to wytłumaczyć, posługując się np. jedną linią obrazu. Jeśli odczytuje się piksel np. numer dwa, to można zwrócić informację o nim, podobnie w przypadku piksela numer trzy. Jednakże, gdyby obraz jednej linii został powiększony dwukrotnie, to np. piksel numer dwa zostanie powielony. Podobnie będzie z pikselem numer trzy. Jeśli obraz nie jest powiekszany o całkowitą wielokrotność, to trzeba wybrać piksel, który zostanie powielony (numer dwa, czy też numer trzy). Niewątpliwie pogorszy to jakość obrazu. Jeszcze gorszy przypadek to zmniejszenie obrazu, trzeba bowiem część pikseli usunąć. Dlatego lepszym rozwiązaniem jest stosowanie filtru liniowego `CLK_FILTER_LINEAR`, który poprawia znacznie jakość odczytywanych pikseli przez uśrednienie wartości piksela za pomocą pikseli sąsiednich.

Jednak w naszym przypadku wystarczy odczyt wartości piksela sąsiedniego, dodatkowo nie są stosowane współrzędne znormalizowane, więc poszczególne piksele są adresowane bezpośrednio.

Po definicji samplera jądro wykonuje operację odczytania piksela z obrazu źródłowego:

```
uint4 pixel = read_imageui(src_image, sampler, coord);
```

Następnie współrzędne zapisane w zmiennej `coord` są odczytywane przez globalne identyfikatory uzyskane za pomocą siatki globalnej:

```
int2 coord = (int2)(get_global_id(0), get_global_id(1));
```

Oczywiście, czynność budowy zmiennej `coord` wykonuje się przed odczytem wartości piksela. Kolejna linia jądra obliczeniowego do zmiennej `pixel2` wpisuje nową wartość. Polega to na tym, iż wartości współrzędnych `x` oraz `y` stanowić będą składowe kolorów czerwonego (składowa R) oraz zielonego (składowa G) pozostałe dwie składowe zostają ustalone na zero:

```
pixel2 = (uint4)(coord.x, coord.y, 0, 0);
```

Teraz dodajemy do siebie dwa piksele:

```
pixel = pixel + pixel2;
```

Może się okazać, iż wartości dwóch pierwszych składowych przekroczą wielkość 255, dlatego dodajemy dwie instrukcje warunkowe sprawdzające zakres wartości w dwóch pierwszych składowych piksela, co widać na listingu 5.24.

Ostatnia czynność wykonywana w jądrze obliczeniowym to zapis piksela do obrazu wyjściowego:

```
write_imageui(dst_image, coord, pixel);
```

Jeśli chcemy tylko wygenerować sam gradient bez łączenia go z obrazem wejściowym, to procedura obliczeniowa jest bardzo krótka:

```
__kernel void imageKernel(write_only image2d_t dst_image) {

    int2 coord = (int2)(get_global_id(0), get_global_id(1));\

    uint4 pixel = (uint4)(coord.x, coord.y, 0, 0);\

    write_imageui(dst_image, coord, pixel);

}
```

Nie ma potrzeby stosowania samplera, bowiem dane o pikselach do obrazu wyjściowego są zapisywane bezpośrednio bez stosowania dodatkowych filtrów.

Listing 5.24. Jądro obliczeniowe nakładające gradient na obraz znajdujący się w obrazie wejściowym `src_image` – obraz wyjściowy jest zapisywany w zmiennej `dst_image`

```
const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;

__kernel void imageKernel(read_only image2d_t src_image, write_only image2d_t dst_image)

{

    int2 coord = (int2)(get_global_id(0), get_global_id(1));

    uint4 pixel = read_imageui(src_image, sampler, coord);

    uint4 pixel2 =(uint4)(coord.x, coord.y, 0, 0);

    pixel=pixel + pixel2;

    if(pixel.x > 255) pixel.x=255;

    if(pixel.y > 255) pixel.y=255;

    write_imageui(dst_image, coord, pixel);

}
```

5.9.2. Skalowanie obrazu

W tym przykładzie prześledzimy operację skalowania obrazu, a dokładniej obraz o mniejszej

rozdzielczości przeskalujemy do wyższej rozdzielczości. Zwrócimy też uwagę na jakość obrazu powiększonego przez zastosowanie odpowiedniego filtru podczas odczytu informacji.

Czynności związane z przygotowaniem danych są bardzo podobne jak w poprzednim przykładzie, to znaczy tworzymy obraz wejściowy o szerokości `WIDTH` oraz wysokości `HEIGHT`, format obrazu będzie taki sam jak w poprzednim przykładzie, tj.:

```
image_format.image_channel_data_type = CL_UNSIGNED_INT8;

image_format.image_channel_order = CL_RGBA;
```

Utworzenie obrazu wejściowego wykonuje następująca jedna linia:

```
input_image = clCreateImage2D(cl_compute_context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, &image_format, WIDTH, HEIGHT, 0, (void*)bytes, &cli_err_num);
```

Utworzenie obrazu wyjściowego wymaga większej uwagi, bowiem po wykonaniu skalowania obraz wyjściowy będzie większy niż obraz wejściowy. Wymiary należy pomnożyć przez współczynnik skalowania, aby otrzymać nową wielkość. Dla uproszczenia obydwa wymiary obrazu zostaną pomnożone przez ten sam współczynnik:

```
output_image = clCreateImage2D(cl_compute_context, CL_MEM_WRITE_ONLY, &image_format, (WIDTH * scale_factor), (HEIGHT * scale_factor), 0, NULL, &cli_err_num);
```

Kolejne czynności związane z budowaniem procedury obliczeniowej pominiemy, możemy zająć się wywołaniem jądra. Należy ustalić trzy parametry: obraz wejściowy, obraz wyjściowy oraz wartość współczynnika skalowania. Wykonamy to w następujący sposób:

```
cli_err_num = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_image);

cli_err_num = clSetKernelArg(kernel, 1, sizeof(cl_mem), &output_image);

cli_err_num = clSetKernelArg(kernel, 2, sizeof(int), &scale_factor);
```

W procedurze uruchamienia jądra obliczeniowego należy zwrócić uwagę na rozmiar siatki obliczeniowej. Działanie procedury skalowania przedstawionej w listingu 5.25 polega na pobieraniu pikseli z obrazu wejściowego, a następnie wstawianiu ich do obrazu wyjściowego. Potrzebna do tego jest siatka obliczeniowa o wymiarach obrazu wejściowego, aby odczytać wszystkie piksele wejściowe:

```
globalWorkSize[0]=WIDTH;

globalWorkSize[1]=HEIGHT;
```

Uruchomienie jądra realizuje poniższa linia kodu:

```
cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 2, NULL, globalWorkSize, NULL, 0, NULL, NULL );
```

Pozostaje do zrealizowania odczyt przeskalowanego obrazu, którego wymiary będą większe, bo pomnożone przez współczynnik skalowania. Ogólna postać kodu do przeniesienia danych obrazu z urządzenia obliczeniowego do pamięci hosta jest analogiczna jak w poprzednim przykładzie:

```
origin[0] = 0; origin[1] = 0; origin[2] = 0;

region[0] = WIDTH * scale_factor; region[1] = HEIGHT * scale_factor; region[2] = 1;

cli_err_num = clEnqueueReadImage(cl_compute_command_queue, output_image, CL_TRUE, origin, region, 0, 0, (void*)out_bytes, 0, NULL, NULL);
```

W tym miejscu należy przejść do opisu procedury obliczeniowej z listingu 5.25. W pierwszej kolejności zwrócimy uwagę na obiekt próbkujący, tj. sampler. Jest on taki sam jak w poprzednim przykładzie, choć w dalszej części zostanie lekko zmodyfikowany, aby poprawić (i to znacznie) jakość skalowania.

Drugim istotnym elementem jest trzeci parametr jądra obliczeniowego − współczynnik skalowania:

```
const int scale_factor
```

Jak widać, jest to liczba całkowita, co oznacza, że możemy tylko powiększać obraz, a dokładnie zwielokrotniać jego wymiary, np. dwu-, trzy- czy czterokrotnie.

Pierwsza czynność wykonywana w jądrze obliczeniowym to odczytanie współrzędnych za pomocą `get_global_id` i ustalenie od razu współrzędnych docelowych (pomnożenie aktualnych współrzędnych przez współczynnik skalowania).

Główne zadanie skalowania wykonują dwie pętle `for`. Przed pętlami odczytujemy wartość piksela z obrazu źródłowego:

```
pixel = read_imageui(src_image, sampler, input_coord );
```

a następnie odpowiednia ilość wywołań:

```
write_imageui(dst_image, output_coord + (int2)(i, j), pixel);
```

zapisuje kilkanaście pikseli do obrazu wyjściowego. Piksele będą powielane, dlatego powiększony obraz nie będzie miał najwyższej jakości. Jeśli wartość `scale_factor` wynosi dwa, to jeden piksel źródłowy zostanie w obrazie docelowym powiększony za pomocą czterech takich samych pikseli. Powiększony obraz będzie, mówiąc kolokwialnie, „kwadratowy”.

Listing 5.25. Procedura obliczeniowa odpowiedzialna za proste skalowanie obrazu

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;

__kernel void imageKernel(read_only image2d_t src_image, write_only image2d_t dst_image, const int scale_factor)

{

    uint4 pixel;

    int2 input_coord = (int2) (get_global_id(0), get_global_id(1));

                int2 output_coord = (int2)(scale_factor*get_global_id(0), scale_factor*get_global_id(1));
```

```
    pixel = read_imageui(src_image, sampler, input_coord );

    for(int i=0; i<scale_factor; i++)

    {

        for(int j=0; j<scale_factor; j++)

        {

            write_imageui(dst_image, output_coord + (int2)(i, j), pixel);

        }

    }

}
```

Poprawę procedury skalowania z listingu 5.25 można osiągnąć przez usunięcie pętli `for` oraz zmianę wymiarów siatki obliczeniowej. Siatka została powiększona do wymiarów obrazu po procedurze skalowania. Zmieniony został także sampler, a dokładnie zamiast flagi `CLK_FILTER_NEAREST` zastosowano flagę oznaczającą filtrowanie liniowe: `CLK_FILTER_LINEAR`. Zastosowano także współrzędne znormalizowane `CLK_NORMALIZED_COORDS_TRUE`, tj. od zera do jedności.

Listing 5.26. Skalowanie obrazu – uproszczona procedura obliczeniowa

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_TRUE | CLK_ADDRESS_CLAMP | CLK_FILTER_LINEAR;

__kernel void imageKernel(read_only image2d_t src_image, write_only image2d_t dst_image)

{

    int outputX = get_global_id(0);

    int outputY = get_global_id(1);

    int width = get_image_width(dst_image);

    int height = get_image_height(dst_image);

    float sourceX = outputX / (float)width;

    float sourceY = outputY / (float)height;

    uint4 pixel = read_imageui(src_image, sampler, (float2) (sourceX, sourceY));

    write_imageui(dst_image, (int2)(outputX, outputY), pixel);

}
```

Całość nowej postaci procedury pokazana w listingu 5.26 jest krótsza od poprzedniej. Jej sposób wywołania nie różni się znacznie od poprzedniej wersji, mamy nawet mniej parametrów do podania: tylko obrazy wejściowy oraz wyjściowy. Należy inaczej skonfigurować siatkę obliczeniową,

a dokładnie mówiąc, trzeba ją powiększyć, wykorzystując współczynnik skalowania:

```
globalWorkSize[0]=WIDTH * scale_factor;

globalWorkSize[1]=HEIGHT * scale_factor;
```

Wywołanie jądra obliczeniowego to taka sama linia kodu jak poprzednio. Istotny element w procedurze jądra to odczytanie współrzędnych:

```
int outputX = get_global_id(0);

int outputY = get_global_id(1);
```

Tym razem są to współrzędne obrazu docelowego, więc aby uzyskać współrzędne źródłowe, trzeba otrzymane wartości podzielić przez docelową szerokość oraz wysokość:

```
float sourceX = outputX / width;

float sourceY = outputY / height;
```

Informacje o wysokości i szerokości można przekazać jako parametry do procedury obliczeniowej, jednak łatwo odczytać informacje o wymiarach obrazu docelowego:

```
int width = get_image_width(dst_image);

int height = get_image_height(dst_image);
```

Dzięki odpowiedniej definicji samplera udało się podać krótką procedurę obliczeniową. Piksele źródłowe są odczytywane ze współrzędnymi od zera do jedności, ale nowe piksele sa zapisywane pod współrzędnymi całkowitymi wygenerowanymi przez siatkę obliczeniową. W ten sposób wykorzystane zostały możliwości OpenCL. Bez jawnego tworzenia kodu do skalowania obrazu można go powiększyć, stosując nie tylko całkowity współczynnik skalowania, np. 2-krotnie, ale także np. 1.5-krotnie. Bez kłopotu zmniejszymy też obraz, stosujac wartość 0.5 − współczynnik skalowania nie musi być liczbą całkowitą, ale np. `float`:

```
const float scale_factor = 0.5;
```

Procedurę skalowania obrazu możemy dalej modyfikować. Inną odmianę pokazano w listingu 5.27. Podobnie jak w poprzednim przykładzie, siatka obliczeniowa ma wymiary obrazu wyjściowego. Jednak w parametrach jawnie musimy podać odwrotność współczynników skalujących:

```
fltWidthNorm = 1.0f / (WIDTH * scale_factor);

fltHeightNorm = 1.0f / (HEIGHT * scale_factor);
```

Robimy to także po to, aby pominąć operację dzielenia w jądrze obliczeniowym podczas wyznaczania współrzędnych źródłowych:

```
float2 sourceNormalXY = convert_float2(outputXY) * (float2)(widthNormFactor,
heightNormFactor);
```

Wystarczy teraz pomnożyć docelowe współrzędne przez podane współczynniki i za pomocą funkcji

convert_float2 wykonać odpowiednią konwersję do typu float.

Listing 5.27. Skalowanie obrazu − inna wersja procedury skalującej bez operacji dzielenia

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_TRUE | CLK_ADDRESS_CLAMP | CLK_FILTER_LINEAR;

__kernel void imageKernel(read_only image2d_t src_image, write_only image2d_t dst_image, const float widthNormFactor, const float heightNormFactor)

{

    int2 outputXY = (int2)(get_global_id(0), get_global_id(1));

      float2 sourceNormalXY = convert_float2(outputXY) * (float2) (widthNormFactor, heightNormFactor);

    uint4 pixel = read_imageui(src_image, sampler, sourceNormalXY);

    write_imageui(dst_image, outputXY, pixel);

}
```

5.9.3. Rozmywanie obrazu – filtr Gaussa

W tym przykładzie pokażemy działanie filtru realizującego rozmycie obrazu. Może on zostać zastosowany do obrazu, który zawiera szum. Filtr rozmywający poprawi jakość obrazu poprzez rozmycie szumu. W przykładzie zastosujemy filtr oparty na funkcji Gaussa (G) rozkładu normalnego. Funkcja G jest określona jako funkcja o dwóch argumentach, bowiem będziemy ją stosować dla obrazu dwuwymiarowego:

$$G(x, y) = \frac{1}{\sigma^2 2\pi} e^{-\frac{(x^2 + y^2)}{2\sigma^2}}$$

gdzie: σ − wartość rzeczywista.

Będziemy stosować wartości równe np. 0.6, 0.8 czy też 1.0.

Warto sprawdzić, jak wartość tego parametru wpływa na poziom rozmycia obrazu. Drugim ważnym elementem jest rozmiar tego filtru. Za pomocą funkcji G generujemy macierz, którą niejako przykłada się do poszczególnych pikseli. Wielkość tej macierzy nie jest naturalnie stała, możemy tworzyć filtry o różnych wymiarach. Warto samodzielnie sprawdzić, jak filtr o wielkości 3 × 3, czy też 5 × 5, wpływa na ostateczną postać obrazu (czy jest mniej lub bardziej rozmyty).

Elastyczność w doborze parametrów dla filtru Gaussa wymaga zdefiniowania odpowiedniej funkcji, np. o nazwie makeGaussianMatrix (szczegóły implementacyjne tej funkcji zostały podane w Dodatku C). Jej stosowanie jest bardzo proste, należy założyć wielkość filtru, np. 7, i określić wartość parametru σ, np. 0.6, zakładając że mamy utworzoną zmienną np.: filterValues, do której przydzielono odpowiednią ilość pamięci (dla rozmiaru 7 macierz będzie miała 49 elementów typu float). Przygotowanie zmiennej filterValues sprowadza się do następującego wywołania:

```
makeGaussianMatrix( 7, 0.6f, filterValues);
```

W naszym programie będziemy stosować nieco inny zapis wymiaru filtru. Jeśli oczekujemy, że macierz filtrująca ma mieć wymiary 7 × 7, to bazowy wymiar filtru jest określony w następujący sposób:

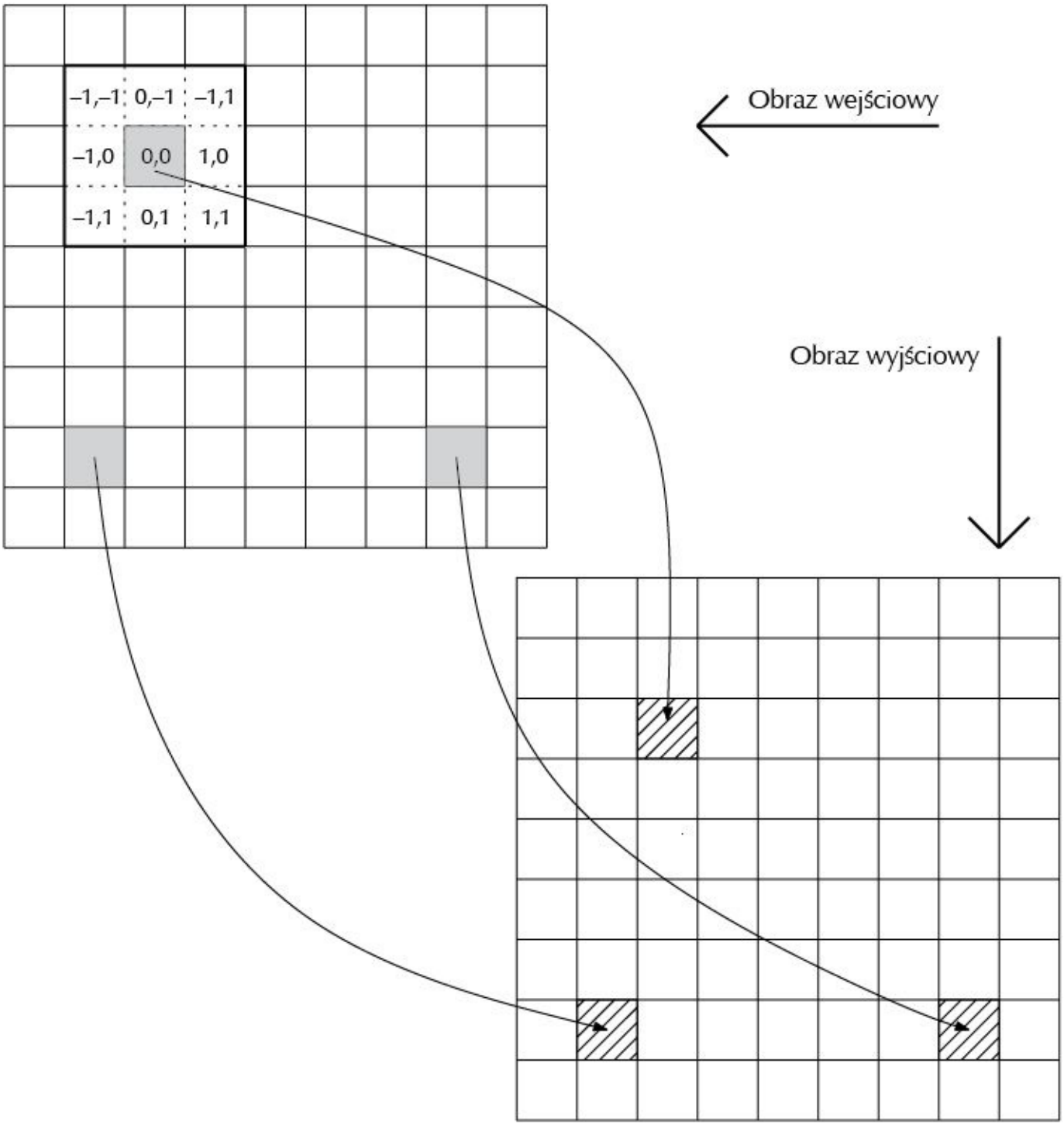
```
const int FILTER_WINDOW_SIZE = 3;
```

Wystarczy pomnożyć tę wartość przez dwa oraz dodać jedność, aby otrzymać właściwy wymiar.

```
makeGaussianMatrix( (2*FILTER_WINDOW_SIZE + 1), sigmaValue, filterValues);
```

Takie rozwiązanie odnosi się do sposobu, w jaki przykładamy macierz opisującą filtr do poszczególnych pikseli. Poglądowo przedstawiono to na rys. 5.16. Choć wyznaczamy wartość jednego piksela, to pobierane są informacje o otaczających go pikselach. Wielkość filtru determinuje, jak duże będzie otoczenie badanego piksela. Ma to zasadnicze duże znacznie dla jakości, czy też stopnia rozmycia obrazu. Dlatego nasza implementacja musi zaoferować możliwość stosowania filtru o zmiennym rozmiarze.

Jądro obliczeniowe z listingu 5.28 jest stosunkowo niewielkie. Deklarujemy sampler, a czynności, jakie wykonujemy w samym jądrze, dzielimy na dwie grupy. W pierwszej odczytujemy informacje o aktualnie przetwarzanym pikselu za pomocą siatki obliczeniowej, co oznacza, że jej wielkość musi odpowiadać rozdzielczości przetwarzanego obrazu.



Rys. 5.16. Stosowanie filtru do poszczególnych pikseli obrazu (choć obliczany jest tylko jeden piksel, to w obliczeniach należy wziąć pod uwagę bezpośrednie otoczenie badanego piksela; wielkość otoczenia w naszym przykładzie to 3 ×3)

Odczytanie współrzędnych piksela sprowadza się do pobrania odpowiednich identyfikatorów dla aktualnej jednostki roboczej:

```
int outputX = get_global_id( 0 );

int outputY = get_global_id( 1 );
```

Następnie tworzona jest zmienna pomocnicza `pixel`, której składowe są ustalane na wartość zero:

```
float4 pixel = (float4)(0.0f);
```

Dwie pętle `for` wykonują główne zadanie: odczytują informacje o otoczeniu piksela, następnie mnożą je przez wartości umieszczone w macierzy filtru, a uzyskana wartość jest dodawana do zmiennej `pixel`. Po odczytaniu wszystkich pikseli, obliczona wartość piksela może zostać zapisana w obrazie wyjściowym. Wykonuje się to za pomocą jednej linii:

```
write_imagef(dst_image, (int2)(outputX, outputY), pixel);
```

Procedura obliczeniowa okazała się dość prosta w implementacji, ale dane opisujące filtr są liczbami rzeczywistymi, natomiast dane obrazu mogą być reprezentowane za pomocą trójek RGB o wartościach w zakresie <0, 255>. Wbrew pozorom nie trzeba jednak konwertować danych, bo jak widać w listingu 5.28 nie ma operacji konwersji. Informacje o pikselach należy odczytywać za pomocą `read_imagef` oraz zapisywać `write_imagef`. Obie funkcje dostarczają informacje o pikselu w postaci znormalizowanych liczb rzeczywistych, co jest niezbędne, aby filtr Gaussa funkcjonował poprawnie.

Obiekty reprezentujące obrazy, gdzie kolory będą opisywane za pomocą liczb 8-btiowych, a ich ułożenie to model RGB z kanałem alfa:

```
image_format.image_channel_data_type = CL_UNORM_INT8;

image_format.image_channel_order = CL_RGBA;
```

Utworzenie obrazów jest podobne jak w poprzednich przykładach, więc obraz wejściowy:

```
input_image = clCreateImage2D(cl_compute_context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, &image_format, WIDTH, HEIGHT, 0, (void*)bytes, &cli_err_num);
```

a obraz wyjściowy jest tworzony następująco:

```
output_image = clCreateImage2D(cl_compute_context, CL_MEM_WRITE_ONLY, &image_format,
WIDTH, HEIGHT, 0, NULL, &cli_err_num);
```

Dane do obrazu wejściowego zostały wcześniej wczytane do zmiennej `bytes`. Pomijamy w tej części wiele innych czynności, jakie należy wykonać, bowiem są one podobne do wcześniejszych przykładów.

Nieco inaczej postępujemy w przypadku wywołania jądra obliczeniowego, gdyż wymagane jest przekazanie danych opisujących filtr. W przypadku tego jądra dane filtru zostały umieszczone w tzw. pamięci stałej, opatrzonej atrybutem `__constant`. Tego typu dane są przeznaczone tylko do odczytu. Mimo to przygotowanie bufora dla nich jest podobne jak w przypadku bufora tworzonego w pamięci urządzenia:

```
filter_values_constant = clCreateBuffer(cl_compute_context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, filterBytes, (void*)filterValues, &cli_err_num);
```

Otrzymany obiekt można przekazać bezpośrednio do jądra obliczeniowego za pomocą `clSetKernelArg`, tak samo, jak zwykły bufor w pamięci głównej urządzenia obliczeniowego.

Przekazanie rozmiaru filtru jest wykonywane inaczej, bowiem nie jest to parametr jądra obliczeniowego, ale stała dla preprocesora, więc należy w pierwszej kolejności zbudować ciąg znaków z opcją `-D`:

```
char preprocesor_options[512];

sprintf(preprocesor_options, ”-DFILTER_WINDOW=%d”, FILTER_WINDOW_SIZE);
```

Sam proces budowy jądra z dodatkową opcją dla preprocesora jest taki sam jak w wielu innych przykładach podawanych w rozdz. 5:

```
cli_err_num = clBuildProgram( cl_kernel_program, clui_num_devices, platform_devices, preprocesor_options, NULL, NULL);
```

Istotnym elementem jest uruchomienie procedury obliczeniowej z poprawnie określoną siatką roboczą. Stosować można tylko siatkę globalną, której wymiary powinny odpowiadać rozdzielczości przetwarzanego obrazu. Sprowadza się to do utworzenia tablicy `globalWorkSize` i podania szerokości i wysokości obrazu:

```
globalWorkSize[0]=WIDTH;

globalWorkSize[1]=HEIGHT;
```

Uruchomienie jądra to jedna linia kodu:

```
cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 2, NULL, globalWorkSize, NULL, 0, NULL, NULL );
```

Po wydaniu polecenia uruchomienia jądra, warto zaczekać na zakończenie tej operacji. Można to zrobić za pomocą zdarzeń lub skorzystać z funkcji `clFinish`:

```
clFinish(cl_compute_command_queue);
```

Odczytanie obrazu wyjściowego po zastosowaniu filtru rozmywającego sprowadza się do ustalenia źródła i regionu obrazu, jaki chcemy skopiować do pamięci gospodarza:

```
origin[0] = 0; origin[1] = 0; origin[2] = 0;

region[0] = WIDTH ; region[1] = HEIGHT ; region[2] = 1;

cli_err_num = clEnqueueReadImage(cl_compute_command_queue, output_image, CL_TRUE, origin, region, 0, 0, (void*)out_bytes, 0, NULL, NULL);
```

Inne czynności zawiązane z obrazem to np. zapisanie danych do pliku PPM albo PNG, ale nie będą one tu opisywane, jako że nie są to zadania wykonywane za pomocą OpenCL.

Listing 5.28. Jądro obliczeniowe realizujące filtr Gaussa do rozmywania obrazu

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

__kernel void imageKernel(read_only image2d_t src_image, write_only image2d_t dst_image, __constant float *filterWeights)

{

    int outputX = get_global_id( 0 );
```

```
    int outputY = get_global_id( 1 );

    float4 pixel = (float4)(0.0f );

    for(int y = – FILTER_WINDOW; y <= FILTER_WINDOW; y++)

    {

        for(int x = – FILTER_WINDOW; x <= FILTER_WINDOW; x++)

        {

                    pixel  +=  filterWeights[(x+FILTER_WINDOW)  +  (y+FILTER_  WINDOW)*
(FILTER_WINDOW*2  +  1)]  *  read_imagef(  src_image,  sampler,  (int2)(outputX,  outputY)  +
(int2)(x,y) );

        }

    }

    write_imagef(dst_image, (int2)(outputX, outputY), pixel);

}
```

5.9.4. Wykrywanie krawędzi w obrazie za pomocą filtru Sobela

Ten przykład dotyczy implementacji procedury obliczeniowej filtru Sobela do wykrywania krawędzi obrazu, ale tym razem opracowanej w podstawowym API standardu OpenCL, czyli pisanej bezpośrednio w języku OpenCL C.

Opracujemy dwie odmiany tego filtru: jedną do wykrywania krawędzi w obrazie w trybie szarości, a drugą w pełnym trybie RGB. Odmiany te różnią się pewnymi szczegółami, więc pokażemy dwie pełne wersje procedur obliczeniowych − oddzielnie dla obrazu w trybie szarości, oddzielnie dla obrazu w pełnym trybie RGB.

W poprzednim przykładzie, do każdego piksela obrazu była stosowana odpowiednia macierz, a w tym przypadku zastosujemy dwie macierze − dla współrzędnych `x` i `y`. Piksel wynikowy otrzymamy, mnożąc piksele otaczające piksel wejściowy przez elementy tych macierzy. Macierz dla współrzędnych `x` ma następującą postać:

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

a macierz dla współrzęnych `y`:

$$M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Obydwie macierze stosuje się podczas wyznaczania wartości piksela.

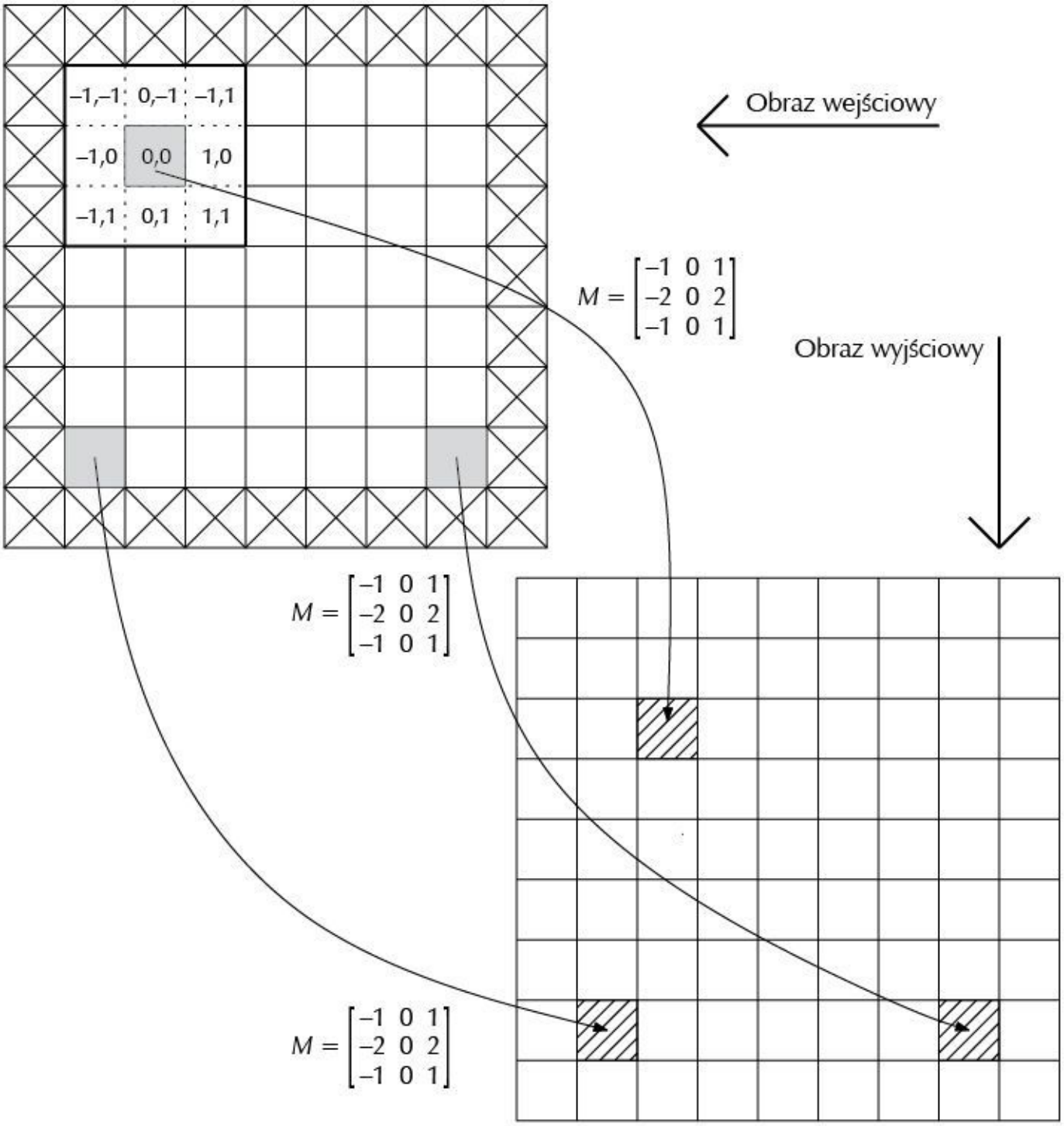
Pojawia się problem z przetwarzaniem pikseli leżących np. w pierwszej kolumnie albo w pierwszym wierszu. W tym przypadku nie możemy zastosować możliwości OpenCL w dziedzinie przetwarzania obrazu. Problem ten można rozwiązać, analizując piksele z pominięciem kłopotliwego pierwszego i ostatniego wiersza oraz pierwszej i ostatniej kolumny.

Ogólna zasada takiego przetwarzania obrazu oraz postaci macierzy reprezentujących filtr Sobela zostały przedstawione na rys. 5.17. Otrzymane wartości danego piksela dla współrzędnej `x` oraz `y` należy odpowiednio przetworzyć z uwzględnieniem opisu kolorów. Jeśli stosujemy kolory RGB lub filtr jest stosowany tylko dla obrazu w skali szarości, to obliczenie nowej wartości piksela o współrzędnych np. `(1,1)` przedstawia się następująco:

```
output_pixel[1,1] = (Img[0,0] * – 1) + (Img[0,1] * 0) + (Img[0,2] * 1) + (Img[1,0] *
– 2) + (Img[1,1] * 0) + (Img[1,2] * 2) + (Img[2,0] * – 1) + (Img[2,1] * 0) +
(Img[2,2] * 1)
```

Wyrażenie `Img` oznacza obraz reprezentowany przez tablicę dwuwymiarową. Podsumowując, przeglądamy każdy piksel obrazu, do każdego piksela oddzielnie stosujemy macierze `x` oraz `y`, otrzymując dwie wartości opisujące piksel źródłowy. Te dwie wyznaczone wartości sumujemy, otrzymując nową wartość piksela, który trzeba odwzorować na wartościach w zakresie od 0 do 255.

W przypadku obrazów w trybie RGB operacje trzeba powtórzyć oddzielnie dla każdej składowej koloru przetwarzanego piksela. Inaczej niż w przypadku filtru Gaussa, rozmiar filtru Sobela dla tego przykładu jest stały, tj. mamy macierz 3×3. Jeżeli zmienimy wartości macierzy, otrzymamy inne filtry.



Rys. 5.17. Obliczanie wartości poszczególnych pikseli dla filtru Sobela (krawędzie obrazu zostają pominięte, a do obrazu docelowego wpisane wartości zerowe; każdy piksel przemnażamy przez elementy filtru oraz obrazu źródłowego w podoby sposób, jak w przypadku filtru Gaussa)

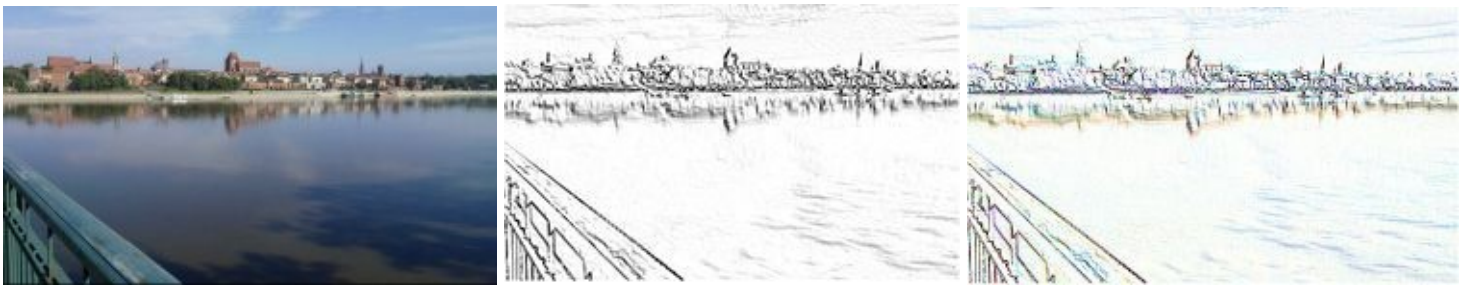
Listing 5.29. Szeregowa procedura realizująca filtr Sobela w trybie szarości napisana w języku C

```
void sobel_grey_filter(unsigned char *in_image, unsigned char *out_image, int sx, int sy)

{

    int ix, iy ;

    for (iy = 0; iy < sy; iy++)
```

```
{
    out_image[iy * sx] = 255;
    out_image[((iy+1) * sx) – 1] = 255;
}
for (ix = 1; ix < sx-1; ix++)
{
    out_image[ix] = 255;
    out_image[ix + ((sy-1) * sx)] = 255;
}
for (iy = 1; iy < sy-1; iy++)
{
    for (ix = 1; ix < sx-1; ix++)
    {
        int sum_x=0, sum_y=0, sum=0;
        sum_x += (int)( in_image[ ix – 1 + (iy – 1)*sy] * – 1);
        sum_x += (int)( in_image[ ix + (iy – 1)*sy] * – 2);
        sum_x += (int)( in_image[ ix + 1 + (iy – 1)*sy] * – 1);
        sum_x += (int)( in_image[ ix – 1 + (iy + 1)*sy] * 1);
        sum_x += (int)( in_image[ ix + (iy + 1)*sy] * 2);
        sum_x += (int)( in_image[ ix + 1 + (iy + 1)*sy] * 1);
        sum_x = min(255, max( 0, sum_x ));
        sum_y += (int)( in_image[ ix – 1 + (iy – 1)*sy] * 1);
        sum_y += (int)( in_image[ ix + 1 + (iy – 1)*sy] * – 1);
        sum_y += (int)( in_image[ ix – 1 + (iy)*sy] * 2);
        sum_y += (int)( in_image[ ix + 1 + (iy)*sy] * – 2);
        sum_y+= (int)( in_image[ ix – 1 + (iy + 1)*sy] * 1);
        sum_y+= (int)( in_image[ ix + 1 + (iy + 1)*sy] * – 1);
        sum_y = min(255, max( 0, sum_y ));
        sum = abs(sum_x) + abs(sum_y);
```

```
            // lub za pomocą pierwiastka

            // sum = (int)(sqrt(sum_x) + sqrtf(sum_y));

            out_image[ ix + iy * sx] = 255 – (unsigned char)(sum);

        }

    }

}
```

Rezultat działania filtru Sobela dla przykładowego obrazu (widok na stare miasto Torunia z mostu na Wiśle) pokazano na rys. 5.18. W przypadku obrazu w trybie szarości filtr Sobela funkcjonuje tak, jak opisano to powyżej. Dla obrazu w kolorach RGB możemy zastosować identyczne podejście, ale, jak już to wspomniano, oddzielnie dla każdej składowej.



Oryginalny obraz | Krawędzie w trybie szarości | Krawędzie w trybie RGB

Rys. 5.18. Wykryte krawędzie dla obrazu testowego w trybie szarości oraz w pełnym trybie RGB (wersja kolorowa dostępna w wydaniu elektronicznym)

Listing 5.30 przedstawia przykładową procedurę obliczeniową, która realizuje filtr Sobela w trybie szarości. W trybie kolorowym opisanym przez trójkę RGB realizuje się go podobnie. Zasada konstruowana procedury obliczeniowej jest podobna do pierwszego przykładu, ale tym razem przetwarzana struktura danych to nie wektor, ale macierz, bo możemy potraktować obraz 2D jako macierz. Nie ma pętli `for`, gdyż podobnie jak w przykładzie z wektorem, indeksy poszczególnych elementów są odczytywane za pomocą funkcji `get_global_id` i, co warto przypomnieć, z argumentami zero oraz jeden, gdyż siatka obliczeniowa jest dwuwymiarowa.

W jądrze obliczeniowym, zanim przystąpimy do obliczania wartości wskazanego punktu, pierwsze dwie instrukcje warunkowe zajmują się ramką obrazu, gdyż nie możemy obliczać wartości punktów leżących np. w pierwszym wierszu. Zajmuje się tym nasza główna instrukcja warunkowa odpowiedzialna za sprawdzenie, czy podany punkt należy do wnętrza obrazu, jeśli tak jest, to przeprowadzane są odpowiednie obliczenia na danym pikselu.

Przygotowanie obiektów platformy, urządzenia, kontekstu, kolejki czy buforów jest podobne jak w poprzednich przykładach, dlatego zajmiemy się tylko tworzeniem jądra obliczeniowego. Rozpoczynamy ten proces od kompilacji kodu źródłowego:

```
program = clCreateProgramWithSource(context, 1, (const char **) &KernelSource, NULL,
```

```
&err);
```

Następnie trzeba utworzyć obiekt programu:

```
err = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

i jądro obliczeniowe:

```
kernel = clCreateKernel(program, ”sobel_grey”, &err);
```

Po utworzeniu jądra obliczeniowego należy ustalić parametry jego wywołania. Poszczególne parametry: obraz wejściowy i obraz wyjściowy, a także wymiary obrazu ustalamy w następujący sposób:

```
err = 0;

err = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&input);

err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&output);

err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &hsx);

err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &hsy);
```

Jeśli nie uda się nam uzyskać wartości `CL_SUCCESS`, to naturalnie nie udało się poprawnie ustalić parametrów jądra obliczeniowego, co można sprawdzić za pomocą pojedynczej instrukcji `if`:

```
if (err != CL_SUCCESS) { …. }
```

W tym momencie możemy już uruchomić procedurę obliczeniową, musimy jednak określić rozmiary siatki obliczeniowej globalnej oraz lokalnej. Mimo że nasz przykład nie korzysta z lokalnej siatki do obliczeń, to jednak będzie działał szybciej, jeśli grupa jednostek roboczych będzie zawierać więcej niż jedną jednostkę roboczą.

```
global[0] = hsx; global[1] = hsy;

local[0]=16; local[1]=16;
```

Samo uruchomienie procedury to zadanie dla polecenia `clEnqueueNDRangeKernel`, które tym razem wykorzystuje globalną oraz lokalną przestrzeń indeksową:

```
err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, &global, &local, 0, NULL,
NULL);

if (err) { …. }
```

Określenie wymiarów jest dokonywane w trzecim parametrze. Odczytywanie przetworzonych danych odbywa się podobnie jak w poprzednich przykładach.

Listing 5.30. Jądro obliczeniowe dla filtru Sobela w trybie szarości

```
__kernel void sobel_grey( __global uchar* input, __global uchar* output,
```

```
unsigned int sx, unsigned int sy) {
int ix = get_global_id(0);
int iy = get_global_id(1);
if ( ix == 0 || ix == (sx – 1) ) {
    output[ iy * sx ] = 255;
    output[ (( iy + 1) * sx) – 1 ] = 255;
}
if ( iy == 1 || iy == (sy – 2) ) {
    output[ ix ] = 255;
    output[ ix + ((sy-1) * sx) ] = 255;
}
if ( (iy > 0 && iy < ( sy-1)) && (ix>0 && ix<(sx-1)) ) {
    int sum_x=0, sum_y=0, sum=0;
    sum_x += (int)( input[ ix – 1 + (iy – 1)*sy] * – 1);
    sum_x += (int)( input[ ix + (iy – 1)*sy] * – 2);
    sum_x += (int)( input[ ix + 1 + (iy – 1)*sy] * – 1);
    sum_x += (int)( input[ ix – 1 + (iy + 1)*sy] * 1);
    sum_x += (int)( input[ ix + (iy + 1)*sy] * 2);
    sum_x += (int)( input[ ix + 1 + (iy + 1)*sy] * 1);
    sum_x = min(255, max( 0, sum_x ));
    sum_y += (int)( input[ ix – 1 + (iy – 1)*sy] * 1);
    sum_y += (int)( input[ ix + 1 + (iy – 1)*sy] * – 1);
    sum_y += (int)( input[ ix – 1 + (iy)*sy] * 2);
    sum_y += (int)( input[ ix + 1 + (iy)*sy] * – 2);
    sum_y+= (int)( input[ ix – 1 + (iy + 1)*sy] * 1);
    sum_y+= (int)( input[ ix + 1 + (iy + 1)*sy] * – 1);
    sum_y = min(255, max( 0, sum_y ));
    sum = abs(sum_x) + abs(sum_y);
    output[ ix + iy * sx] = 255 – (unsigned char)(sum);
```

```
    }

}
```

Zaprezentowana implementacja filtru Sobela może zostać dodatkowo przyspieszona przez poprawną konfigurację lokalnej siatki obliczeniowej. Warto w tym momencie spróbować zaimplementować tzw. **filtr konwolucyjny** (ang. *convolution filter*). Na przykład warto zastosować macierz o następującej postaci:

$$E = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

aby otrzymać obraz wytłoczony (ang. *emboss*).

Warto również wykonać testy wydajności procedury bezpośredniej oraz opartej na typach obrazów, aby się przekonać, ile zyskamy, realizując implementację filtru do wykrywania krawędzi za pomocą OpenCL w porównaniu do zwykłej szeregowej jego odmiany. Do analizy wydajności można wykorzystać pakiet CodeXL, który został przedstawiony w rozdz. 7.

5.10. Współpraca OpenCL z OpenGL

Dotychczasowe przykłady skupiały się na różnych aspektach obliczeniowych. Tymczasem OpenCL to technologia, która nie jest ograniczona tylko do realizacji samych obliczeń. Standard OpenCL oferuje możliwość współpracy ze standardem OpenGL[9]. Można za jego pomocą wykonać odpowiednie przekształcenia wierzchołków, a obliczone dane przekazać do wizualizacji za pomocą OpenGL. Również za pomocą procedur obliczeniowych OpenCL można generować tekstury.

W dalszej części tego rozdziału przedstawione zostaną sposoby łączenia OpenCL oraz OpenGL. Program będzie tworzył wykres funkcji `sin` na zielonym tle. Przy czym obliczenia dotyczące wartości funkcji `sin` będą przeprowadzane na poziomie jądra obliczeniowego, również tekstura będzie generowana przez procedurę obliczeniową OpenCL.

5.10.1. Procedury obliczeniowe OpenCL

Omawianie rozpoczniemy od przedstawienia dwóch procedur obliczeniowych pokazanych w listingu 5.31. Pierwsza z nich `create_texture_kernel` odpowiada za wypełnienie podanego obrazu kolorem zielonym. Warto ją tak przerobić, aby np. na podstawie dodatkowych argumentów można było zmieniać kolor, np. w zależności wartości parametru `sequence`, tak jak to robi pierwsza z procedur obliczeniowych: `create_vbo_kernel`. Wykonuje swoje główne zadanie, czyli generuje wartości funkcji `sin`, w zależności od wymiarów obrazu (parametry `w` oraz `h`) oraz wartości wspomianego parametru `sequence`, który pozwala na przesunięcie w poziomie wykresu funkcji `sin`.

Listing 5.31. Jądra obliczeniowe OpenCL generujące wartości wykresu funkcji `sin` oraz teksturę stanowiącą tło wykresu

```
__kernel void create_vbo_kernel(__global float4 *in_data, const int w, const int h,
const int sequence)

{
```

```
    int GID = get_global_id(0);

    float4 line;

    float f = 0.75f;

    float a = (float)h / 3.0f;

    float b = w/2.0f;

    line.x = GID;

    line.y = b + a*sin(M_PI_F*4.0*((float)GID/(float)w*f + (float)sequence/(float)w));

    line.z = GID+1.0f;

              line.w    =    b    +    a*sin(M_PI_F*4.0*((float)(GID+1.0f)/(float)w*f    +
(float)sequence/(float)w));

    in_data[GID] = line;

}

__kernel void create_texture_kernel(__write_only image2d_t in_image )

{

    int2 coord = { get_global_id(0), get_global_id(1) };

    float4 color = {0.20f, 0.40f, 0.0f, 0.0f};

    write_imagef( in_image, coord, color );

}
```

5.10.2. Czynności wykonywane w programie gospodarza

Postaci procedur z listingu 5.31 są bardzo typowe, o wiele więcej czynności należy wykonać w programie gospodarza. Trzeba zadeklarować dwie zmienne `GLuint`, które będą reprezentować teksturę oraz bufor wierzchołków:

```
GLuint tex = 0;

GLuint vbo = 0;
```

Przygotowanie tekstury po stronie OpenGL wykonuje funkcja `prepare_texture`. Kod źródłowy tej funkcji został przedstawiony w listingu 5.32. Funkcja ta tylko przygotowuje indeks oraz określa parametry tekstury. Podobnie jest w przypadku bufora wierzchołków. Funkcja `prepare_vbo` jest odpowiedzialna za przygotowanie indeksu oraz określenie poprawnego rozmiaru bufora.

Listing 5.32. Przygotowanie indeksu tekstury oraz indeksu dla bufora wierzchołków

```
void prepare_texture( int width, int height )

{
```

```
    glGenTextures(1, &tex);

    glTexEnvi( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE );

    glBindTexture(GL_TEXTURE_RECTANGLE_ARB, tex);

       glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,  0,  GL_RGBA32F_ARB,  width,  height,  0,
GL_LUMINANCE, GL_FLOAT, NULL );

}

void prepare_vbo (int vbo_len )

{

    GLint buf_size;

    glGenBuffers(1, &vbo);

    glBindBuffer(GL_ARRAY_BUFFER, vbo);

    if( glGetError() != GL_NO_ERROR ) { … }

    glBufferData(GL_ARRAY_BUFFER, vbo_len*sizeof(float)*4, NULL, GL_STREAM_DRAW);

    if( glGetError() != GL_NO_ERROR ) { … }

    glGetBufferParameteriv(GL_ARRAY_BUFFER, GL_BUFFER_SIZE, &buf_size);

    if ((GLuint)buf_size != (vbolen*sizeof(float)*4)) { … }

glBindBuffer(GL_ARRAY_BUFFER, 0);

    if( glGetError() != GL_NO_ERROR ) { … }

}
```

Przygotowania po stronie OpenGL oraz OpenCL przebiegają w typowy dla siebie sposób. Jednakże utworzenie kontekstu obliczeniowego wymaga wskazania, że OpenCL będzie współpracował z OpenGL np. za pomocą protokołu X. Wymaga to utworzenia dodatkowego obiektu opisującego własności tego kontekstu:

```
cl_context_properties context_props[] = {

    CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id,

    CL_GL_CONTEXT_KHR, (cl_context_properties)glXGetCurrentContext(),

    CL_GLX_DISPLAY_KHR, (cl_context_properties)glXGetCurrentDisplay(),

    0};
```

Oznacza to też, iż tworzony program będzie funkcjonował w środowisku X, np. w systemie Linux.

Utworzenie kontekstu obliczeniowego wymaga podania opisu własności:

```
context = clCreateContext(context_props, 1, &devies[0], NULL, NULL, &cli_err_num);
```

W przypadku systemu Windows, dodatkowe własności dla kontekstu są określane w następujący sposób:

```
cl_context_properties context_props[] = {

    CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id,

    CL_GL_CONTEXT_KHR, (cl_context_properties)wglGetCurrentContext(),

    CL_WGL_HDC_KHR, (cl_context_properties)wglGetCurrentDC(),

    0};
```

Kolejna zmiana dotyczy sposobu, w jaki tworzone są obiekty pamięciowe. Utworzenie bufora w pamięci urządzenia obliczeniowego na podstawie indeksu `vbo` odbywa się następująco:

```
cl_mem cl_vbo_mem;

cl_vbo_mem = clCreateFromGLBuffer(context, CL_MEM_READ_WRITE, vbo, &cli_err_num );
```

Przydział pamięci dla tekstury również odbywa się za pomocą odpowiedniej funkcji:

```
cl_mem cl_tex_mem;

cl_tex_mem = clCreateFromGLTexture2D(context, CL_MEM_READ_WRITE,
GL_TEXTURE_RECTANGLE_ARB, 0, tex, &cli_err_num );
```

Listing 5.33 przedstawia główne etapy uruchomienia jądra obliczeniowego generującego teksturę. Istotne elementy to wywołanie funkcji `clEnqueueAcquireGLObjects` oraz `clEnqueueReleaseGLObjects`, które zapewniają, iż podczas generowania tekstury, dostęp do niej będzie zablokowany i procedura OpenCL poprawnie wykona wszystkie obliczenia związane z generacją tekstury.

Listing 5.33. Kolejne etapy wywołania jądra obliczeniowego tworzącego teksturę

```
cli_err_num = clSetKernelArg(tex_kernel, 0, sizeof(cl_mem), &cl_tex_mem);

size_t tex_global[2] = { image_width, image_height };

size_t tex_local[2] = { 16, 2 } ;

glFinish();

cli_err_num = clEnqueueAcquireGLObjects(command_queue, 1, &cl_tex_mem, 0, NULL, NULL );

cli_err_num = clEnqueueNDRangeKernel(command_queue, tex_kernel, 2, NULL,
tex_globalWorkSize, tex_localWorkSize, 0, NULL, NULL);

cli_err_num = clEnqueueReleaseGLObjects(command_queue, 1, &cl_tex_mem, 0, NULL, NULL );
```

```
clFinish(command_queue);
```

W analogiczny sposób postępujemy w przypadku jądra obliczeniowego generującego wykres funkcji, również wymagane jest zablokowanie dostępu do obiektów, które aktualnie są przetwarzane. Dodatkowo, parametr `sequence` należy sukcesywnie zwiększać oraz, korzystając z reszty dzielenia modulo, zawęzić wartości tego parametru do szerokości generowanego obrazu:

```
sequence = (sequence+1) % (image_width);
```

Po zakończeniu procesu obliczeniowego, obliczoną teksturę oraz wykres funkcji należy wyświetlić za pomocą OpenGL. Listing 5.34 przedstawia fragmenty procedur do wyświetlania tekstu oraz bufora wierzchołków. W tym drugim przypadku korzystamy z funkcji `glDrawArrays` do wyświetlania linii opisującej wykres funkcji `sin`.

Pozostało już tylko podać, jak wygląda funkcja główna, odpowiadająca za generowanie sceny w ramach OpenGL:

```
glClearColor(0.0f, 0.0f, 1.0f, 1.0f );

glClear( GL_COLOR_BUFFER_BIT );

compute_kernel_for_texture();

compute_kernel_for_vbo();

display_texture(imWidth,imHeight);

render_vbo( vbo_len );

glutSwapBuffers();
```

W ten sposób udało się połączyć przetwarzanie OpenCL z generowaniem grafiki przez OpenGL, co umożliwia wykorzystanie możliwości obu standardów do tworzenia wydajnych aplikacji graficznych.

Listing 5.34. Procedury odpowiedzialne za wyświetlenie tekstury oraz wykresu funkcji obliczonej przez OpenCL

```
void display_texture(int width, int height)

{

    glEnable(GL_TEXTURE_RECTANGLE_ARB);

    glBindTexture(GL_TEXTURE_RECTANGLE_ARB, tex );

    glBegin(GL_QUADS);

    glTexCoord2f(0, 0); glVertex2f(0, 0);

    glTexCoord2f(0, height); glVertex2f(0, height);

    glTexCoord2f(width, height); glVertex2f(width, height);

    glTexCoord2f(width, 0); glVertex2f(width, 0);
```

```
    glEnd();

    glDisable(GL_TEXTURE_RECTANGLE_ARB);

}

void render_vbo( int vbo_len )

{

    glColor3f(1.0f, 0.0f, 0.0f);

    glLineWidth(2.0f);

    glBindBufferARB(GL_ARRAY_BUFFER_ARB, vbo);

    glEnableClientState(GL_VERTEX_ARRAY);

    glVertexPointer( 2, GL_FLOAT, 0, 0 );

    glDrawArrays(GL_LINES, 0, vbo_len*2);

    glDisableClientState(GL_VERTEX_ARRAY);

    glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);

}
```

Przypisy

[1] Jeśli `printf` jest stosowany w procedurze obliczeniowej wywoływanej za pośrednictwem pakietu PyOpenCL, może się okazać, iż niezbędne jest ustawienie zmiennej środowiskowej PYOPENCL_COMPILER_OUTPUT=1, aby zobaczyć komunikaty na poziomie konsoli języka Python.

[2] Specyfikator konwersji `%n` pozwala odczytać, ile znaków zostało przesłane na wyście do momentu wystąpienia tego specyfikatora, np. `printf("Jakis tekst!!!%n", &ile_znakow);`, w zmiennej `ile_znakow` umieszczona zostanie wartość 14.

[3] Nowa, obecnie przygotowywana, wersja standardu OpenCL 2.0 będzie już zezwalać na uruchamianie jądra obliczeniowego z poziomu innego jądra obliczeniowego.

[4] Technika Monte Carlo poraz pierwszy została opracowana m.in. przez polskiego matematyka Stanisława Ulama. Zazwyczaj polega na losowaniu wartości i sprawdzaniu, czy spełniają pewne założone warunki. Wylosowanie odpowiednio dużej liczby punktów pozwala przybliżyć się do właściwego rozwiązania, bez przeprowadzania skomplikowanych obliczeń analitycznych, czy numerycznych.

[5] Adres projektu TinyMT jest następujący: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html`.

[6] Plik, który nas interesuje, znajduje się pod adresem: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/OPENCL/index.html`.

[7] Dalsze informacje dotyczące przetwarzania obrazów zostały podane w rozdziale 5.11.

[8] W przykładzie można również zastosować tryb RGB bez kanału alfa, jednak warto sprawdzić, jakie typy obrazu są obsługiwane przez dane urządzenie OpenCL. W większości przypadków obsługiwany jest tryb RGBA, jednak można się spotkać z sytuacją, że typ bez kanału alfa, tj. RGB, nie będzie obsługiwany.

[9] Współpraca nie ogranicza się tylko do standardu OpenGL, system DirectX i inne rozwiązania, jak np. protokół X, także są obsługiwane.

6 OpenCL w innych językach programowania

Technologia OpenCL jest względnie łatwa do przeniesienia do innych języków programowania. Istnieją także dodatkowe pakiety, które upraszczają korzystanie z OpenCL. Takim pakietem jest PythonCL, który ułatwia korzystanie z technologii OCL w ramach języka Python. Jest to bardzo dobre rozwiązanie do testowania poprawności działania określonej procedury obliczeniowej. Python oraz wygodne środowisko pracy, np. Spyder, umożliwią generowanie interaktywnych testów, które w języku o pełnej kompilacji, jak np. C czy C++, będą naturalnie utrudnione. Informacje o tym, jak skorzystać z tych możliwości podano w rozdz. 6.1.

Nieco inną rolę pełni pakiet APARAPI. Jest to biblioteka przeznaczona dla języka Java, współpracująca ze standardem OpenCL. Poszczególne procedury obliczeniowe są tworzone w Javie i tłumaczone przez APARAPI do OpenCL. Mimo iż część programu jest wykonywana na urządzeniu obliczeniowym, to pierwotnie cały program jest napisany w języku Java. W ten sposób bardzo łatwo implementować niektóre algorytmy do obliczeń numerycznych. Przykładem jest algorytm Blacka-Sholesa odnoszący się do wyceny instrumentów finansowych. Jego wersja w Javie jest niemal bezpośrednim przeniesieniem oryginalnego zapisu, a mimo to uzyskuje się spore przyspieszenie względem podstawowej wersji wykonywanej na procesorze szeregowym. Dodatkowo, dzięki APARAPI, proces tworzenia jądra obliczeniowego został znacząco uproszczony.

Odnośnie do języka C++ warto także wspomnieć o pakiecie VexCL, szczególnie w aspekcie obliczeń. Oprócz ukrycia detali związanych z zainicjowaniem OpenCL, VexCL oferuje kilkanaście dodatkowych typów ułatwiających obsługę wektorów oraz macierzy. Jeśli obliczenia, które są realizowane, wykorzystują macierze oraz wektory, to VexCL może znacząco przyspieszyć i ułatwić realizację postawionego zadania.

6.1. Język Python

Tworząc nawet niewielki prototyp funkcji obliczeniowej, niestety trzeba przygotować sporo kodu pomocniczego, który dla różnych funkcji nie musi być za każdym razem identyczny, bowiem szczegóły wywołania testowanej funkcji mogą wymagać większych bądź mniejszych zmian. Dotyczy to szczególnie API w języku C, stosowanie C++ rozwiązuje kilka problemów.

Najwygodniej jednak byłoby stosować bardziej abstrakcyjne rozwiązanie, i takim jest wykorzystanie języka Python oraz pakietu PyOpenCL[1] autorstwa Andreasa Kloecknera[2] [PyOpenCL]. Tworząc prototyp funkcji, możemy się skupić tylko na jądrze obliczeniowym, bowiem API PyOpenCL oferuje kilkanaście wygodnych rozwiązań, które sprowadzają zadanie związane z OpenCL do pojedynczych linii kodu.

W rozdziale 3.1 pokazaliśmy prosty przykład przedstawiający informacje o platformie obliczeniowej. Opracowanie podobnego skryptu w języku Python jest zadaniem znacznie łatwiejszym do realizji. Poprzednio, mimo iż interesował nas tylko odczyt informacji o platformie, wymagało to wielu linii kodu.

Uzyskanie informacji o dostępnych platformach wymaga skorzystania z metody `get_platforms`, która znajduje się w module `pyopencl`. Inaczej mówiąc, pisząc:

```
plat = pyopencl.get_platforms()
```

otrzymujemy obiekt, który możemy potraktować jako tablicę i odczytywać informacje o platformach, pisząc wprost np.:

```
print plat[0].name

print plat[1].name
```

lub korzystając z pętli `for`:

```
for p in pyopencl.get_platforms():

    print p.name
```

W ten sposób zobaczymy nazwy dwóch platform, a liczbę dostępnych platform OpenCL sprawdzimy za pomocą `size`:

```
platcount = size(plat)
```

Każda platforma ma jednakże metodę `get_info`, dzięki której odczytujemy potrzebne informacje, np. o nazwie platformy, czy też ciąg znaków, gdzie znajdują się wszystkie dostępne rozszerzenia. Metoda `get_info` wymaga podania argumentu określającego, jakie informacje chcemy odczytać. Listing 6.1 przedstawia prosty przykład wyświetlania nazwy wszystkich dostępnych platform oraz informacji o dostępnych rozszerzeniach.

Listing 6.1. Informacje o dostępnych platformach OpenCL oraz spis dostępnych rozszerzeń

```
import pyopencl

for platform in pyopencl.get_platforms():

    platName = platform.get_info( pyopencl.platform_info.NAME )

    platExt = platform.get_info( pyopencl.platform_info.EXTENSIONS )

    print ”Platform:”, platName

    print

    print ”\t”,platExt

    print
```

6.1.1. Przegląd API pakietu PyOpenCL

W tabeli 6.1 przedstawiono podstawowe moduły, klasy oraz najważniejsze metody, jakie są dostępne w pakiecie PyOpenCL. Ich nazwy odzwierciedlają funkcje podstawowego API dostępne w C/C++. Dzięki ukryciu zarządzania pamięcią, wykonywanie typowych czynności, jak np. odczyt platformy, kolejki, czy kompilacja jądra obliczeniowego, zostały znacząco uproszczone.

Tabela 6.1. Podstawowe klasy, moduły oraz metody dostępne w PyOpenCL

Nazwa klasy/modułu	Istotne metody
pyopencl	get_platforms, create_some_context, enqueue_read_buffer, enqueue_write_buffer
Platform	info, get_info, get_devices
Device	info, get_info
Context	info, get_info, konstruktor Context
Program	info, get_info, build, get_build_info, kernel_name, all_kernels, konstruktor Program
Kernel	info, get_info, set_arg, set_args, konstruktor Kernel
CommandQueue	info, get_info, set_property, flush, finish, konstruktor CommandQueue
MemoryObject	info, get_info, release, get_host_array
Buffer	get_sub_region, __getitem__, konstruktor Buffer
Image	info, get_image_info, shape, release, konstruktor Image

Uruchomienie jądra obliczeniowego przy wykorzystaniu pakietu PyOpenCL jest bardzo proste, ale w pierwszej kolejności musimy utworzyć kolejkę poleceń. W tym celu musimy uzyskać dostęp do platformy, pobrać listę urządzeń z platformy, utworzyć kontekst, a następnie, używając tego kontekstu oraz urządzenia, utworzyć kolejkę poleceń.

Platformę, jak już wcześniej pokazaliśmy, możemy odczytać bezpośrednio[3], pierwsza dostępna

platforma jest oznaczona indeksem zero:

```
platform = pyopencl.get_platforms()[0]
```

Następnie odczytamy obiekt reprezentujący urządzenia:

```
devices = platform.get_devices()
```

Bez wskazywania konkretnego urządzenia, możemy utworzyć obiekt kontekstu:

```
context = pyopencl.Context(devices)
```

Utworzenie kolejki wymaga wskazania konkretnego urządzenia, więc wykorzystamy fakt, że jest to tablica, i wybierzemy element znajdujący się pod indeksem zero:

```
queue = pyopencl.CommandQueue( context, devices[0] )
```

Pakiet PyOpenCL bardzo uprościł wykonywanie podstawowych czynności w porównaniu zwłaszcza do języka C. Odczytanie nazwy urządzenia obliczeniowego to tylko trzy linie kodu:

```
dev_name = queue.get_info(pyopencl.command_queue_info.DEVICE)
.get_info(pyopencl.device_info.NAME)

dev_name = queue.get_info(

    pyopencl.command_queue_info.DEVICE).get_info(

        pyopencl.device_info.NAME)
```

Wykorzystujemy obiekt kolejki, aby odczytać informacje o urządzeniu, z którego korzysta kolejka, a następnie odczytujemy nazwę urządzenia.

Listing 6.2. Utworzenie kolejki dla pierwszej platformy oraz odczytanie nazwy urządzenia obliczeniowego

```
import pyopencl



platform = pyopencl.get_platforms()[0]

devices = platform.get_devices()

context = pyopencl.Context(devices)



queue = pyopencl.CommandQueue( context, devices[0] )

dev_name =
queue.get_info(pyopencl.command_queue_info.DEVICE).get_info(pyopencl.device_info.NAME)

print(”Nazwa urządzenia: %s” % (dev_name))
```

Kolejne zadanie to utworzenie jądra obliczeniowego. Wczytanie kodu jądra OpenCL z zewnętrznego

pliku sprowadza się do dwóch poleceń:

```
fileHandle = open('kernelSource.cl', 'r')

kernelSource_text = fileHandle.read()
```

W pierwszej linii otwieramy plik ze źródłami jądra obliczeniowego w trybie tylko do odczytu, a w drugiej linii wczytujemy jego zawartość do zmiennej tekstowej. Oznacza to, iż możemy treść jądra obliczeniowego podać wprost jako ciąg znaków i przypisać do wybranej zmiennej, np. w taki sposób:

```
kernelSource_text = """

__kernel void piEstimation( __global float *a, __global float *b, const int N) {

    const int idx = get_global_id(0);

    // kod jądra obliczeniowego OpenCL

}

"""
```

Utworzenie obiektu reprezentującego procedurę obliczeniową, którą będziemy mogli poddać kompilacji, wymaga dostępu do kontekstu oraz samego tekstu źródłowego:

```
program = pyopencl.Program(context, kernelSource_text )
```

Słowo `Program` oznacza w tym momencie konstruktor obiektu reprezentującego procedurę obliczeniową (możemy nazywać ją także programem). Po uzyskaniu obiektu dla jądra obliczeniowego można od razu przystąpić do kompilacji, wywołując metodę `build`:

```
program.build()
```

Jednak warto zastosować sekcję `try`, aby przechwycić ewentualne wyjątki, jakie mogą się pojawić, jeśli w kodzie źródłowym znajdować się będą błędy np. syntaktyczne w postaci brakującego przecinka, średnika czy też nawiasu.

Informację o błędach wykrytych w trakcie kompilacji łatwo odczytać, korzystając z metody `get_build_info`:

```
try:

    program.build()

except:

    logString = program.get_build_info( devices[0], pyopencl.program_build_info.LOG) )

    raise
```

Musimy podać urządzenie oraz wskazać, że chcemy odczytać informacje z logu powstałego w trakcie kompilacji. W podobny sposób odczytamy informacje o statusie i dodatkowych opcjach, jakie zostały

użyte w trakcie kompilacji określonego jądra obliczeniowego:

```
print(program.get_build_info(devices[0], pyopencl.program_build_info.OPTIONS))

print(program.get_build_info(devices[0], pyopencl.program_build_info.STATUS))
```

Kompilacja jądra obliczeniowego to jeszcze nie koniec przygotowań. Należy odczytać obiekt reprezentujący skompilowaną procedurę obliczeniową. W podstawowym API jądro było wywoływane za pomocą funkcji `clEnqueueNDRangeKernel`, w Pythonie możemy odczytać odpowiedni obiekt np. w taki sposób:

```
kernelName = pyopencl.Kernel(program, 'kernelName')
```

Odczytany obiekt `kernelName` reprezentuje funkcję z jądra o takiej samej nazwie. Nazwa obiektu nie musi być taka sama jak nazwa funkcji, możemy przyjąć inną. Inny sposób uzyskania odniesienia do jądra sprowadza się do odczytu odpowiedniej własności z obiektu reprezentującego skompilowane jądro obliczeniowe. Jeśli mamy dwie procedury: wymienioną już `kernelName` oraz `nameone`, to dostęp do tej drugiej procedury obliczeniowej możemy uzyskać w następujący sposób:

```
nameOne_kernel = program.nameone
```

Uzyskane obiekty możemy uruchamiać prawie jak zwykłe funkcje Pythona − oprócz argumentów zadeklarowanych w jądrze obliczeniowym, musimy także przekazać obiekt kolejki oraz kształt siatki obliczeniowej:

```
kernelName( queue, globalGrid, localgrid, par1, Par2, N)
```

Ten sposób wywołania jądra może zostać uproszczony, co pokażemy w rozdz. 6.1.2.

Z jądra obliczeniowego możemy odczytywać dalsze informacje, np. o nazwie danej funkcji, liczbie jej argumentów czy też liczbie odniesień do wybranej przez nas funkcji obliczeniowej:

```
kernelName.get_info( pyopencl.kernel_info.FUNCTION_NAME )

kernelName.get_info( pyopencl.kernel_info.NUM_ARGS )

kernelName.get_info( pyopencl.kernel_info.REFERENCE_COUNT)
```

Całość omówionych dotychczas zadań podsumowuje listing 6.3, w którym znajduje się kod źródłowy tworzący jądro obliczeniowe.

Listing 6.3. Utworzenie jądra obliczeniowego oraz uzyskanie dostępu do wybranych funkcji obliczeniowych

```
import pyopencl



platform = pyopencl.get_platforms()[0]

devices = platform.get_devices()

context = pyopencl.Context(devices)
```

```
fileHandle = open(’kernelSource.cl’, ’r’)

kernelSource_text = fileHandle.read()


program = pyopencl.Program(context, kernelSource_text )


try:

    program.build()

except:

    print(”Informacje o bledzie:”)

    print(program.get_build_info(devices[0], pyopencl.program_build_info.LOG))

    print(program.get_build_info(devices[0], pyopencl.program_build_info.OPTIONS))

    print(program.get_build_info(devices[0], pyopencl.program_build_info.STATUS))

    raise


kernelName = pyopencl.Kernel(program, ’kernelName’)


nameOne_kernel = program.nameone


print(”Nazwa jadra:”)

print( kernelName.get_info( pyopencl.kernel_info.FUNCTION_NAME ) )

print( nameOne_kernel.get_info( pyopencl.kernel_info.FUNCTION_NAME ) )
```

6.1.2. Znów dodawanie wektorów, ale w Pythonie

W tym przykładzie pokażemy ponownie dodawanie wektorów, ale najważniejszym elementem będzie w nim przekazanie argumentów do jądra obliczeniowego. To zadanie również jest obsługiwane przez pakiet PyOpenCL, więc można się spodziewać dalszych ułatwień. Szczególnie, iż PyOpenCL współpracuje z pakietem Numerical Python (w skrócie NumPy, więcej informacji w [NumPy2014]), który stał się bardzo popularnym rozwiązaniem do obliczeń numerycznych w języku Python.

Najpierw utworzymy jądro obliczeniowe. Przyjmie ono taką postać, jak w przykładzie omówionym

w rozdz. 4.3 listing 4.5:

```
__kernel void sum(

    __global const float *a, __global const float *b,

    __global float *c ) {



        int id = get_global_id(0);

        c[id] = a[id] + b[id];



}
```

Dwa pierwsze argumenty reprezentują dane do odczytu, a w trzecim argumencie znajduje się wektor wynikowy, do którego tylko zapisujemy dane.

Działania, jakie należy wykonać w skrypcie, można podzielić na kilka etapów. Na początek należy włączyć dwa pakiety: wymieniony wcześniej NumPy oraz PyOpenCL ułatwiający korzystanie z OpenCL, ale też w pełni współpracujący z pakietem NumPy.

Pierwszą ważną czynnością w kontekście OpenCL, jest wybór urządzenia. W przypadku PyOpenCL, ten krok jest automatycznie wykonywany poprzez wybór pierwszej dostępnej platformy oraz urządzenia, więc możemy go pominąć.

Trzeba przygotować dane − niezbędne jest określenie ich wielkości, np. za pomocą zmiennej `VECTOR_SIZE`. Tworzymy tablice o nazwach `a` oraz `b`, które są tablicami liczb zmiennoprzecinkowych typu `float`. Zaletą pakietu NumPy jest zgodność typów z typami OpenCL, co znacząco ułatwia tworzenie procedur obliczeniowych, dlatego wystarczą dwie linie kodu, aby utworzyć odpowiednie zmienne:

```
a = numpy.random.rand(VECTOR_SIZE).astype(numpy.float32)

b = numpy.random.rand(VECTOR_SIZE).astype(numpy.float32)
```

Dodatkowo, stosując metodę `rand`, od razu wypełniamy tablicę wartościami losowymi, a za pomocą `astype` wymuszamy, aby został zastosowany typ o pojedynczej precyzji.

Następnie tworzymy kontekst obliczeniowy reprezentowany przez zmienną `ctx` oraz kolejkę poleceń – zmienną `q`.

Najważniejszym zadaniem jest przeniesienie danych z wektorów `a` i `b` do pamięci urządzenia obliczeniowego. Ponieważ zmienne `a` oraz `b` reprezentują dane przeznaczone tylko do odczytu, to stosujemy flagę `READ_ONLY`, co jest ważne dla pakietu PyOpenCL, aby poprawnie utworzył obiekty reprezentujące parametry jądra:

```
a_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY | pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=a)
```

```
b_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=b)
```

Powyższe linie kodu utworzą w pamięci urządzenia obliczeniowego nowe bufory w trybie tylko do odczytu oraz samodzielnie przeniosą zawartości zmiennych `a` oraz `b`. Wykorzystujemy też obiekt kontekstu obliczeń, który został podany jako pierwszy argument.

Trzeci bufor, jaki tworzymy, będzie zawierał rezultat dodawania:

```
dest_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.WRITE_ONLY, b.nbytes)
```

Utworzony bufor jest przeznaczony wyłącznie do zapisu, a jego wielkość jest określona bezpośrednio w bajtach. Ponieważ rozmiar wyjścia jest taki sam, jak np. zmiennej `b`, to wykorzystując tę zmienną i pole `nbytes`, podajemy rozmiar danych wyznaczony wcześniej.

Po przygotowaniu bufora w pamięci urządzenia obliczeniowego, od razu można przystąpić do utworzenia obiektu reprezentującego procedurę obliczeniową. Zadanie to wykonuje się w dwóch krokach: w pierwszym jest tworzony obiekt programu (zmienna `prgobj`), a w drugim jest budowany obiekt procedury obliczeniowej (zmienna `prg`):

```
prgobj = pyopencl.Program(ctx, ”treść jądra obliczeniowego”)

prg = prgobj.build()
```

Za pomocą zmiennej `prg` można uruchomić procedurę obliczeniową poprzez wywołanie metody o takiej samej nazwie, jak procedura obliczeniowa opatrzona słowem kluczowym `kernel`:

```
prg.sum( q, a.shape, None, a_buf, b_buf, dest_buf )
```

Pierwszy argument w powyższej linii kodu to kolejka, a drugi określa kształt siatki obliczeniowej. Zgodnie z wcześniejszymi ustaleniami, siatka jest reprezentowana przez wektor indeksowany wartościami od `0` do `n-1`. Trzeci argument, wartość `None`, określa postać lokalnej siatki obliczeniowej. Ostatnie trzy argumenty to obiekty pamięciowe zawierające dane naszych wektorów.

Pozostałe czynności do wykonania w programie polegają na weryfikacji, czy obliczenia zostały przeprowadzone poprawnie. Wcześniej odczytujemy rezultat za pomocą `enqueue_copy`:

```
pyopencl.enqueue_copy(q, a_plus_b, dest_buf)
```

co polega na kopiowaniu danych z bufora urządzenia obliczeniowego do zmiennej znajdującej się w pamięci RAM gospodarza, do tego zadania wykorzystujemy kolejkę poleceń.

Warto wykonać jeszcze jeden test, a mianowicie obliczyć normę różnicy między sumą obliczoną przez jądro obliczeniowe oraz sumą wyznaczoną przez pakiet NumPy. Wystarczy do tego tylko jedna linia kodu:

```
print ”wartość błędu:”, numpy.linalg.norm(a_plus_b – (a+b))
```

Wartość zero oznacza, że w obliczeniach nie ma żadnych błędów. Wartość inna niż zero ogólnie nie musi od razu oznaczać zepsutego sprzętu, ale np. zły rozmiar siatki obliczeniowej bądź niepoprawne zaimplementowanie jądra obliczeniowego.

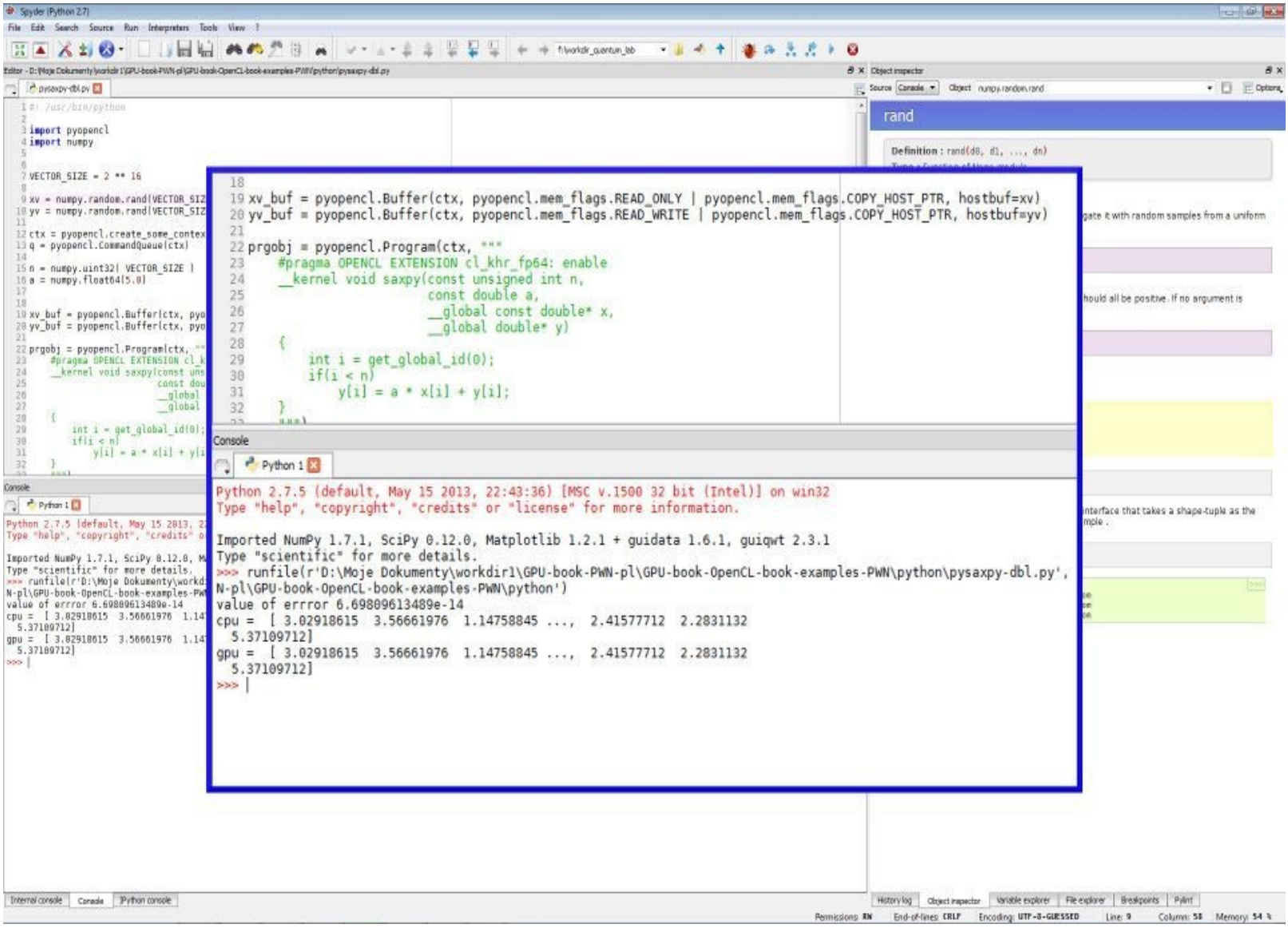
Całość kodu źródłowego omówionego przykładu przedstawiono w listingu 6.4, co ponownie pokazuje, iż PyOpenCL pomaga ukryć wiele detali z obsługą OpenCL i nadaje się do testów procedur obliczeniowych, łatwiej bowiem uruchomić skrypt w Pythonie, niż ponownie kompilować cały program w języku C, czy też C++.

Listing 6.4. Kompletny skrypt do obliczania sumy dwóch wektorów w PyOpenCL

```
#! /usr/bin/python

import pyopencl

import numpy

VECTOR_SIZE = 2 ** 3

a = numpy.random.rand(VECTOR_SIZE).astype(numpy.float32)

b = numpy.random.rand(VECTOR_SIZE).astype(numpy.float32)

ctx = pyopencl.create_some_context()

q = pyopencl.CommandQueue(ctx)

a_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=a)

b_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=b)

dest_buf = pyopencl.Buffer(ctx, pyopencl.mem_flags.WRITE_ONLY, b.nbytes)

prgobj = pyopencl.Program(ctx, """

__kernel void sum( __global const float *a,

        __global const float *b,

        __global float *c ) {

    int id = get_global_id(0);

    c[id] = a[id] + b[id];

}

""")

prg = prgobj.build()

prg.sum(q, a.shape, None, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)

pyopencl.enqueue_copy(q, a_plus_b, dest_buf)
```

```
print ”wartość błędu:”, numpy.linalg.norm(a_plus_b – (a+b))
```

Warto wspomnieć, że pierwsze wersje kart graficznych nie obsługiwały liczb o podwójnej precyzji, jednakże od dłuższego czasu nie ma takiego ogranicznia. W przykładzie z listingu 6.4, aby uzyskać obsługę tego typu liczb, należy do procedury obliczeniowej dodać odpowiednią pragmę:



Rys. 6.1. Środowisko Spyder oraz skrypt do sumy dwóch wektorów dla liczb o podwójnej precyzji

```
#pragma OPENCL EXTENSION cl_khr_fp64: enable
```

a także poprawić nagłówek funkcji w następujący sposób:

```
__kernel void sum(__global const double *a,

    __global const double *b,

    __global double *c)

{ … }
```

W samym skrypcie należy wykonać tylko zmianę typu generowanych wartości, np.:

```
a = numpy.random.rand(VECTOR_SIZE).astype(numpy.float64)
```

Stosujemy typ `float64`, który jest bezpośrednim odpowiednikiem typu `double`.

6.1.3. Skrypt do testowania siatki obliczeniowej

W rozdziale 4.4 podano procedurę obliczeniową tworzącą wektor, którego wartości odpowiadały konfiguracji siatki obliczeniowej. Listing 6.5 pokazuje skrypt w Pythonie, który ma za zadanie uruchomić tę procedurę, aby łatwo było zauważyć relacje między parametrami siatki globalnej oraz lokalnej.

Parametry `X` oraz `Y`, jakie znajdują się na początku skryptu w listingu 6.5 to wymiary siatki globalnej, a parametry `lx` oraz `ly` to wymiary siatki lokalnej. Wartości `X`, `Y` są używane do określenia wielkości wektora, gdyż istotnie wyznaczają liczbę elementów, jaka musi zostać utworzona:

```
N = numpy.uint32( (X)*(Y) );
```

Kolejne czynności są takie, jak w poprzednio omawianych przykładach − tworzone są obiekty platformy, urzadzeń, kontekstu i kolejki poleceń. Ponieważ jądro obliczeniowe generuje tylko wektor wyjściowy, dlatego tworzony jest tylko jeden obiekt pamięciowy:

```
out_data_dev = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_WRITE |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=out_data)
```

Istotne jest utworzenie jądra obliczeniowego, należy bowiem przekazać wymiary siatki obliczeniowej globalnej, lokalnej, a także wartość offsetu globalnego:

```
prg.prepare_vector(q, (X, Y), (lx, ly), out_data_dev, global_offset=(0, 0))
```

Odczytanie danych z urządzenia obliczeniowego to ponownie typowe wykorzystanie API, jakie oferuje pakiet PyOpenCL. Jednak otrzymane dane to tylko wektor, gdy chcemy pokazać współrzędne w postaci macierzy. Pakiet NumPy oferuje funkcję `reshape`, która otrzymany wektor zamienia w macierz. Wymiary macierzy to oczywiście wartości `X`, `Y`, czyli wymiary siatki globalnej:

```
out_data_shadow_reshape = numpy.reshape(out_data_shadow, (Y, X))
```

Argumenty `X`, `Y` zostały zamienione miejscami, ponieważ `reshape` oczekuje podania liczby wierszy w pierwszym argumencie, a tę rolę pełni wartość `Y` oznaczająca niejako wysokość struktury, jaką tworzymy. Liczba `X` to inaczej liczba kolumn, czyli elementów w jednym wierszu. Wyświetlenie tak przetworzonego wektora sprowadza się do użycia typowej instrukcji `print` w ramach języka Python.

Listing 6.5. Skrypt w Pythonie do testowania konfiguracji siatki obliczeniowej

```
#! /usr/bin/python

import pyopencl

import numpy



X=6
```

```
Y=8

lx=3

ly=4

N = numpy.uint32( (X)*(Y) );

out_data = numpy.zeros(N).astype(numpy.float32)

platform = pyopencl.get_platforms()[0]

platName = platform.get_info( pyopencl.platform_info.NAME )

platExt = platform.get_info( pyopencl.platform_info.EXTENSIONS )
print ”Platform:”, platName
print ”\t”,platExt
print

devices = platform.get_devices()

ctx = pyopencl.Context(devices)

q = pyopencl.CommandQueue(ctx)

out_data_dev = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_WRITE |
pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=out_data)
```

```
prgobj = pyopencl.Program(ctx, """

__kernel void prepare_vector(__global float *output)

{

// usunięte fragmenty procedury obliczeniowej

// kod procedury znajduje się na Listingu 4.4.1

}

""")



prg = prgobj.build()



prg.prepare_vector(q, (X,Y), (lx,ly), out_data_dev, global_offset=(0,0))



out_data_shadow = numpy.empty_like(out_data)

pyopencl.enqueue_copy(q, out_data_shadow, out_data_dev)



out_data_shadow_reshape = numpy.reshape(out_data_shadow, (Y, X))

print out_data_shadow

print out_data_shadow_reshape
```

6.1.4. Stosowanie typów użytkownika

Pakiet NumPy wspomaga pakiet PyOpenCL w zakresie typów podstawowych, takich jak `float`, `double`, a także typów całkowitych jak choćby `int`. Jednakże język OpenCL C dopuszcza stosowanie struktur, które pozwolą na zwiększenie czytelności kodu. W rozdziale 5.5 przedstawiono przykład generowania liczby π za pomocą metody Monte Carlo. Zastosowany generator liczb pseudolosowych wymagał stosowania struktury do przekazywania dodatkowych danych do jądra obliczeniowego.

Środowisko Python oferuje odpowiednie rozwiazania (pakiet NumPy), ułatwiające przekazywanie danych z poziomu skryptu do funkcji w języku C, a nawet procedury obliczeniowej OpenCL danych zapisanych w odpowiednio przygotowanej strukturze. Wykorzystuje się pakiet `numpy.dtype`, aby utworzyć odpowiednie obiekty dostępowe między Pythonem a językiem C, czy też API OpenCL.

W przykładzie z rozdziału 5.5.3 stosowano strukturę o nazwie `tinymt32wp_t`, która jest zdefiniowana w następujący sposób:

```
typedef struct TINYMT32WP_T {
```

```
    uint s0;

    uint s1;

    uint s2;

    uint s3;

    uint mat1;

    uint mat2;

    uint tmat;

} tinymt32wp_t;
```

Definicja zmiennej dostępowej rozpoczyna się od określenia typu dla poszczególnych pól oryginalnej struktury `tinymt32wp_t`:

```
tinymt32wp_t = numpy.dtype([

    (”s0”, numpy.uint32),

    (”s1”, numpy.uint32),

    (”s2”, numpy.uint32),

    (”s3”, numpy.uint32),

    (”mat1”, numpy.uint32),

    (”mat2”, numpy.uint32),

    (”tmat”, numpy.uint32) ])
```

W ten sposób otrzymujemy reprezentację typu strukturalnego z języka C/C++ w języku Python. Należy jeszcze dokonać jej konwersji na typ zgodny z określonym urządzeniem obliczeniowym OpenCL:

```
tinymt32wp_t,                            tinymt32wp_t_c_decl                            =
pyopencl.tools.match_dtype_to_c_struct(ctx.devices[0], ”tinymt32wp_t”, tinymt32wp_t)
```

Obiekt `ctx` to kontekst obliczenieniowy. Utworzenie np. tablicy opartej na strukturze `tinymt32wp_t` sprowadza się do zastosowania metody `numpy.empty`, przy czym w pierwszym argumencie podajemy liczbę elementów, a w drugim typ pojedynczego elementu:

```
tinymt32wp_t_proxy = numpy.empty(total_num, tinymt32wp_t)
```

Powstała struktura nie jest jednak zainicjowana. Wpisanie wartości do poszczególnych elementów jest bardzo proste, np. wpisanie na pozycję `idx` wartości do pola `s0`, czy też `tmat` jest realizowane w następujący sposób:

```
tinymt32wp_t_proxy[idx][”s0”]=0
```

```
tinymt32wp_t_proxy[idx][”tmat”]=0xcc3b75ff
```

Obiekt pamięciowy tworzy się podobne jak obiekt dla liczb typu `float`. Deklaracja obiektu tylko do odczytu, którego elementami są struktury wraz z przeniesieniem zawartości tablicy `tinymt32wp_t_proxy` do pamięci urządzenia obliczeniowego, jest realizowane przez następującą linię kodu:

```
tinymt32wp = pyopencl.Buffer(ctx, pyopencl.mem_flags.READ_ONLY | pyopencl.mem_flags.COPY_HOST_PTR, hostbuf=tinymt32wp_t_proxy)
```

6.1.5. Klasa w Pythonie dla łatwiejszej współpracy z OpenCL

Rozdział o pakiecie PyOpenCL warto podsumować, tworząc nieskomplikowaną klasę, która jeszcze bardziej ułatwi korzystanie z OpenCL. Dla podstawowych czynności, jak wczytanie jądra obliczeniowego, jego kompilacja, czy też wywołanie oraz odczyt rezultatów, warto przygotować kilka metod, pozwalających na szybką integrację jądra obliczeniowego z resztą programu bez konieczności przeplatania czynności związanych z OpenCL z kodem sterującym dla podstawowej aplikacji.

Zbudujemy klasę, która ułatwi ładowanie oraz uruchamianie określonego jądra obliczeniowego. Metody pakietu PyOpenCL upraszczają wiele czynności, ale chcemy uzyskać możliwość pracy z procedurą obliczeniową poprzez proste tworzenie obiektu, a następnie ładowanie jądra obliczeniowego realizowane jedną metodą, podobnie budowę. Na koniec trzecia metoda doprowadzi do uruchomienia procedury obliczeniowej i odczyta otrzymane wyniki. Może to wyglądać następująco:

```
example = OCLExample()

example.loadKernel(”testKernel.cl”)

example.prepareKernel(10)
```

Uruchomienie jądra i odczyt informacji wykonamy w następujący sposób:

```
[a,b,c] = example.executeKernel(”testKernel”)

print ”a”, a print ”b”, b

print ”c”, c
```

Zakładamy, że kod źródłowy jądra podany w pliku *testKernel.cl* ma procedurę o nazwie `testKernel`. Nie ustalono parametrów, ale nasza klasa zakłada, że choć procedury mogą mieć różne nazwy, to zestaw parametrów jest taki sam.

Cała klasa OCLExample została przedstawiona w listingu 6.6. Mamy tu kilka metod. Pierwsza `__init__` to naturalnie konstruktor klasy, który odczytuje kontekst oraz kolejkę, nie odczytuje natomiast platformy i urządzenia. Metody PyOpenCL dostarczą nam obiekty wskazujące na domyślne urządzenie obliczeniowe, zazwyczaj jest to karta graficzna.

Druga metoda `loadKernel` wczytuje jądro obliczeniowe i dokonuje jego kompilacji, ponieważ dajemy możliwość wskazania nazwy jądra obliczeniowego, co może być przydatne, gdy chcemy

wybrać jedno z kilku dostępnych. Obiekt programu budujemy w następujący sposób:

```
self.program = ocl.Program(self.ctx, fstr).build()
```

Parametr `fstr` to źródłowa postać jądra obliczeniowego.

Kolejna metoda `prepareKernel` przygotowuje bufory dla danych. W naszym przypadku jedynym parametrem tej metody jest liczba elementów, które będziemy przechowywać w buforze. Ostatnia metoda `executeKernel` ma za zadanie uruchomić wskazane jądro obliczeniowe. Przyjmuje za parametr nazwę metody, jaka ma zostać uruchomiona. Dlatego przed uruchomieniem należy utworzyć obiekt jądra obliczeniowego w następujący sposób:

```
self.kernel = ocl.Kernel(self.program, kernelName)
```

Uruchomienie jądra w takim przypadku można zrealizować w następujący sposób:

```
self.kernel(self.queue, self.a.shape, None, self.a_buf, self.b_buf, self.dest_buf)
```

Nie musimy tu podawać nazwy funkcji obliczeniowej, jak np. w rozdz. 6.1.2.

Listing 6.6. Przykładowa klasa w języku Python wykorzystująca pakiet OpenCL do podstawowego zarządzania jądrem obliczeniowym

```
#! /usr/bin/python


import pyopencl as ocl

import numpy


class OCLExample:

    def __init__(self):

        self.ctx = ocl.create_some_context()

        self.queue = ocl.CommandQueue(self.ctx)


    def loadKernel(self, filename, showKernel = False):

        f = open(filename)


        fstr = f.read()

        if showKernel == True:

            print fstr
```

```
        self.program = ocl.Program(self.ctx, fstr).build()



    def prepareKernel(self, N):

        self.a = (numpy.random.rand(N)*10).astype(numpy.uint32);

        self.b = (numpy.random.rand(N)*10).astype(numpy.uint32);



                    self.a_buf  =  ocl.Buffer(self.ctx,  ocl.mem_flags.READ_  ONLY  |
ocl.mem_flags.COPY_HOST_PTR, hostbuf=self.a)

                    self.b_buf  =  ocl.Buffer(self.ctx,  ocl.mem_flags.READ_  ONLY  |
ocl.mem_flags.COPY_HOST_PTR, hostbuf=self.b)

        self.dest_buf = ocl.Buffer(self.ctx, ocl.mem_flags.WRITE_ONLY, self.b.nbytes)



    def executeKernel(self, kernelName):

        self.kernel = ocl.Kernel(self.program, kernelName)

                self.kernel(self.queue,  self.a.shape,  None,  self.a_buf,  self.b_buf,
self.dest_buf)

        c = numpy.empty_like(self.a)

        ocl.enqueue_read_buffer(self.queue, self.dest_buf, c).wait()

        return [self.a, self.b, c]
```

6.2. Pakiet APARAPI dla języka Java

APARAPI to najbardziej zaawansowany pakiet przedstawiany w kontekście OpenCL. Sama nazwa APARAPI jest skrótem od ang. „*A PARallel API*”, w bezpośrednim tłumaczeniu to równoległe API.

Korzystanie z APARAPI jest najłatwiejsze w porównaniu do innych zaprezentowanych rozwiązań, gdyż procedury OpenCL, w tym także pisanie jądra obliczeniowego, zostały w całości przeniesione do języka Java. Nie musimy korzystać z API OpenCL, aby tworzyć kolejkę, obiekt jądra, czy też alokować pamięć po stronie urządzenia. Wszystkie te zdania są wykonywane niejako w tle, lub też zostały znacznie uproszczone.

Największym ułatwieniem jest tworzenie treści jądra obliczeniowego również w języku Java. Obowiązują pewne obostrzenia, jednak programy zaprezentowane w tej części w całości będą pisane w języku Java. Na pierwszy rzut oka nie widać, że część kodu jest wykonywana przez zewnętrzne urządzenie obliczeniowe.

6.2.1. Dodawanie wektorów

Powrócimy raz jeszcze do skryptu realizującego dodawanie dwóch wektorów `a` i `b`, zapisującego wynik do wektora `c`:

```
c[i] = a[i] + b[i];
```

Przykład zapisany w Javie, dzięki pakietowi APARAPI będzie jednym z krótszych przykładów, gdzie stosujemy OpenCL. Potwierdza to listing 6.7, szczególnie jeśli porównamy ten przykład z poprzednimi przykładami − jest krótszy oraz w całości napisany w Javie, bez oddzielnego pliku bądź zmiennej tekstowej, w której podano treść jądra obliczeniowego przeznaczonego do kompilacji.

Oddzielny problem to kompilacja przykładu, a także instalacja pakietu APARAPI. Nie będzie ona omawiana w tym miejscu, szczegóły można znaleźć w Dodatku A.

W przykładzie mamy tylko jedną klasę o nazwie `VectorSum`. Jądro obliczeniowe jest tworzone dynamicznie za pomocą klasy `Kernel`. Jednak nim do tego dojdziemy, w funkcji głównej definiujemy potrzebne tablice reprezentujące wektory:

```
final float[] a = new float[N];

final float[] b = new float[N];

final float[] c = new float[N];
```

Są to typowe wyrażenia dla języka Java. Zdefiniowaliśmy też zmienną `N`, która reprezentuje wielkość naszego wektora. Kolejne czynności to przygotowanie zawartości wektorów `a` oraz `b` − wystarczy pętla `for`, która wypełni je wartościami losowymi.

Przygotowanie jądra obliczeniowego również jest znacznie uproszczone w porównaniu do podstawowego API OpenCL, wystarczy bowiem utworzyć obiekt typu `Kernel`, a następnie podać implementację metody `run`, która zostanie wykonana przez urządzenie obliczeniowe. W naszym przykładzie przedstawia się to następująco:

```
Kernel sum_kernel = new Kernel() {

    @Override public void run() {

        int gid = getGlobalId();

    c[gid] = a[gid] + b[gid];

    }

};
```

Nie ma potrzeby odczytywać kontekstu, kolejki, tworzyć obiektów dla programu, jądra czy przeprowadzać kompilacji danej procedury obliczeniowej. Wszystkie te czynności są wykonywane samodzielnie przez pakiet APARAPI.

Metoda `getGlobalId` to naturalnie bezpośredni odpowiednik funkcji `get_global_id`, czyli odczytuje numer wątku za pomocą globalnej siatki obliczeniowej. Jednak w odróżnieniu od zwykłego API, nie ma potrzeby przekazywania argumentów do jądra obliczeniowego. Skoro w jądrze, czyli w metodzie

`run`, pojawiają się odwołania do zmiennych `a`, `b`, `c`, to oznacza, iż należy przekazać do klasy `Kernel` odwołania do tych zmiennych. APARAPI zadba o przesłanie danych do urządzenia, wykonanie obliczeń oraz odczyt danych z urządzenia. Niewątpliwie, takie podejście jest w duchu języka Java, w którym nie zwalniamy pamięci samodzielnie, bo zajmuje się tym odśmiecacz (ang. *garbage collector*) − nie przekazujemy argumentów, bo system wykona to samodzielnie.

Uruchomienie jądra również jest dość proste, należy jedynie przygotować obiekt opisujący siatkę obliczeniową. W kontekście omawianego przykładu korzystamy tylko z typu `Range`, aby nadać siatce obliczeniowej kształt linii:

```
sum_kernel.execute(Range.create( N ));
```

To prawie wszystkie zadania, jakie odnoszą się do OpenCL − po zakończeniu obliczeń należy usunąć jądro obliczeniowe za pomocą metody `dispose`:

```
sum_kernel.dispose();
```

Pakiet APARAPI nie usuwa jądra obliczeniowego, tę czynność należy wykonać samodzielnie.

Dodatkowo w przykładzie przed usunięciem procedury obliczeniowej za pomocą metody:

```
sum_kernel.getExecutionMode()
```

można odczytać, gdzie zostało wykonane jądro obliczeniowe − czy na układzie graficznym GPU, czy na tradycyjnym procesorze CPU.

Listing 6.7. Program w Javie obliczający sumę dwóch wektorów, wykorzystujący pakiet APARAPI

```
import com.amd.aparapi.Kernel;

import com.amd.aparapi.Range;



import java.io.*;

import java.util.Random;



public class VectorSum {



    public static void main(String[] args) {



        final int N = 16;
```

```
        final float[] a = new float[N];

        final float[] b = new float[N];

        final float[] c = new float[N];


        for (int i = 0; i < N; i++) {

            a[i] = (float)(Math.random());

            b[i] = (float)(Math.random());

        }


        Kernel sum_kernel = new Kernel()

        {

            @Override public void run()

            {

                int gid = getGlobalId();

                c[gid] = a[gid] + b[gid];

            }

        };


        sum_kernel.execute( Range.create( N ) );

        for (int i = 0; i < N; i++) {

              System.out.printf(”%d : %3.2f + %3.2f = CPU %3.2f GPU %3.2f\n”, i, a[i],
b[i],a[i]+b[i], c[i]);

        }

                        System.out.println(”Tryb   wykonania   dla   jądra   =   ”   +
sum_kernel.getExecutionMode());

        sum_kernel.dispose();

    }

}
```

6.2.2. Przegląd API pakietu APARAPI

APARAPI oferuje bogaty zestaw dodatkowych klas i metod, mimo że celem jego powstania była daleko posunięta integracja z Javą. Istnieje kilkanaście dodatkowych obiektów, które warto omówić, szczególnie, jeśli utworzone procedury obliczeniowe mają charakteryzować się wysoką wydajnością. Można też odrzucić mechanizmy dynamicznego tłumaczenia kodu w Javie do OpenCL i bezpośrednio podać treść procedur obliczeniowych w OpenCL C. Więcej informacji na ten temat podano w rozdz. 6.2.4.

Najłatwiejszy sposób budowy jądra obliczeniowego sprowadza się do utworzenia obiektu typu `Kernel`, a następnie w miejscu utworzenia obiektu możemy od razu przeciążyć metodę `run`. Przedstawia się to w następujący sposób:

```
Kernel name_of_kernel = new Kernel() {

    @Override public void run() {

    // treść jądra obliczeniowego

    }

};
```

Takie rozwiązanie zostało wykorzystane w poprzednim rozdziale, w którym realizowano sumę dwóch wektorów.

Uruchomienie jądra obliczeniowego wykonamy za pomocą metody `execute`. Głównym argumentem w przypadku APARAPI jest obiekt opisujący rozmiar i kształt siatki obliczeniowej. Wcześniej jednak można określić typ urządzenia, na jakim jądro obliczeniowe ma zostać uruchomione. Pakiet APARAPI określa cztery typy urządzeń:

• układ klasy GPU – `Kernel.EXECUTION_MODE.GPU` (urządzenie zgodne z OpenCL),

• tradycyjny procesor – `Kernel.EXECUTION_MODE.CPU` (urządzenie zgodne z OpenCL),

• za pomocą puli wątków Java – `Kernel.EXECUTION_MODE.JTP`,

• szeregowo za pomocą jednej pętli – `Kernel.EXECUTION_MODE.SEQ`.

Uruchamiając jądro obliczeniowe, system APARAPI stara się uruchomić podany kod na urządzeniu GPU, szukając w systemie bibliotek dla standardu OpenCL. Jeśli nie uda się zlokalizować obsługi OpenCL lub nie ma urządzenia obliczeniowego zgodnego ze standardem OpenCL, to zlecone jądro obliczeniowe zostanie wykonane za pomocą wątków Javy.

Możemy zawsze sprawdzić, w jakim trybie nastąpiło wykonanie jądra obliczeniowego, korzystając z metody `getExecutionMode`, która zwróci jedną z czterech podanych powyżej stałych. Można naturalnie wymusić tryb uruchomienia jądra obliczeniowego za pomocą tej metody:

```
computation_kernel.setExecutionMode( Kernel.EXECUTION_MODE.CPU );
```

Powyższa linia określa, iż kod jądra zostanie wykonany przez urządzenie OpenCL przy użyciu tradycyjnego procesora.

W przypadku Open CL, jak zawsze, ważne jest, aby odczytać identyfikatory grupy i jednostki roboczej oraz wielkości grupy. APARAPI ma następujące metody (funkcje), które odczytują potrzebne nam informacje:

• `getGlobalId()` – identyfikator globalny jednostki roboczej,

• `getGlobalSize()` – całkowita liczba jednostek roboczych,

• `getGroupId()` – identyfikator grupy roboczej,

• `getLocalId()` – identyfikator pojedynczej jednostki w grupie roboczej,

• `getLocalSize()` – liczba jednostek roboczych w grupie roboczej,

• `getNumGroups()` – liczba grup roboczych.

Ogólnie powyższe funkcje stanowią odpowiedniki funkcji z OpenCL C, przy czym należy podkreślić, iż starsze wydania APARAPI obsługiwały tylko jednowymiarową siatkę obliczeniową. Najnowsza wersja (w połowie roku 2014 na etapie wersji testowej) oferuje już pełną obsługę siatek obliczeniowych.

Nie ma potrzeby też określać funkcji do przenoszenia danych między urządzeniem obliczeniowym a pamięcią gospodarza, proces ten jest całkowicie przezroczysty dla programisty.

6.2.3. Algorytm Blacka-Sholesa

Algorym Blacka-Sholesa opracowany przez Fishera Blacka i Myrona Scholesa (nazwa algorytmu/metody pochodzi od nazwisk Autorów) to metoda oceny instrumentów finansowych, którą w bezpośredni sposób można implementować w środowisku równoległym, gdzie jest dostępnych wiele jednostek obliczeniowych. Jej implementacja w OpenCL to praktycznie bezpośrednie przeniesienie formuł modelu Blacka-Sholesa na poziom OpenCL, czy w przypadku APARAPI na poziom języka Java.

Model jest opisany przez następujące cząstkowe równanie różniczkowe:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

gdzie: S − cena danego instrumentu (wartość losowa), V − funkcja zależna od czasu i ceny instrumentu, t − czas.

Nie jest naszym zadaniem dokładne omawianie tego równania, powiemy tylko, że pokazuje ono, że cena zależy tylko od ceny podanej w czasie t.

Nie stosuje się oryginalnych równań, ale przybliżenia modelu Blacka-Sholesa, w których wprowadza się więcej parametrów, jak K − cena wykonania, r − stopa procentowa pozbawiona ryzyka, σ − zmienność.

$$C(S, t) = N(d_1)S - N(d_2)Ke^{-r(T-t)}$$

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)\left(T-t\right)\right]$$

$$d_2 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)\left(T-t\right)\right] = d_1 - \sigma\sqrt{T-t}$$

gdzie: C − wycena danego instrumentu finansowego (opcja kupna), funkcja N − rozkład prawdopodobieństwa.

Można też podać opcję sprzedaży:

$$P(S, t) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)S$$

Powyższe wzory są bezpośrednio zamieniane na wyrażenia języka Java lub OpenCL C, jeśli bezpośrednio implementuje się algorytm Blacka-Sholesa. Na przykład obliczenie wartość `d1` oraz `d2` wykonujemy następująco:

```
d1 = (log(S / K) + (R + sigma_val * sigma_val / two) * T) / sigma_sqrt_t;

d2 = d1 – sigma_sqrt_t;
```

Listing 6.8 zawiera także wywołanie funkcji `phi`, która oblicza przybliżenie wartości skumulowanego rozkładu normalnego za pomocą metody Abromowitza-Steguna.

Przygotowanie jądra obliczeniowego wymaga sporządzenia tablicy wartości losowych `rand_array`, za pomocą której będą zainicjowane ceny akcji ograniczone stałymi `S_LOWER_LIMIT` oraz `S_UPPER_LIMIT` (określającymi dolną i górną granicę ceny danego instrumentu finansowego). Tworzone są też tabele opcji kupna i sprzedaży:

```
rand_array = new float[size];

call_option = new float[size];

put_option = new float[size];

for (int i = 0; i < size; i++) {

    rand_array[i] = i * 1.0f / size;

}
```

Uruchomienie jądra obliczeniowego wymaga przygotowania siatki obliczeniowej:

```
Range range = Range.create(size);
```

i utworzenia obiektu reprezentującego jądro:

```
BlackScholesKernel kernel = new BlackScholesKernel(size);
```

Teraz można uruchomić jądro obliczeniowe dla podanej siatki obliczeniowej:

```
kernel.execute( range );
```

Procedura obliczająca wycenę jest dość prosta z punktu widzenia implementacji równoległej. Wydajność uzyskujemy dzięki temu, że poszczególne wyceny mogą być prowadzone niezależnie od siebie, a dzięki obecności wielu jednostek przetwarzających można uzyskać znaczące przyspieszenie obliczeń.

Listing 6.8. Metoda `run` obejmująca główne obliczenia w modelu Blacka-Sholesa, uruchamiana na poziomie urządzenia obliczeniowego jako jądro obliczeniowe

```
@Override public void run() {

    float d1, d2;

    float phi_d1, phi_d2;

    float sigma_sqrt_t;

    float k_exp_minus_rt;



    int GID = getGlobalId();



    float two = 2.0f;

    float in_rand = rand_array[GID];

    float S = S_LOWER_LIMIT * in_rand + S_UPPER_LIMIT * (1.0f – in_rand);

    float K = K_LOWER_LIMIT * in_rand + K_UPPER_LIMIT * (1.0f – inRand);

    float T = T_LOWER_LIMIT * in_rand + T_UPPER_LIMIT * (1.0f – inRand);

    float R = R_LOWER_LIMIT * in_rand + R_UPPER_LIMIT * (1.0f – in_rand);

      float sigma_val = SIGMA_LOWER_LIMIT * in_rand + SIGMA_UPPER_LIMIT * (1.0f –
in_rand);



    sigma_sqrt_t = sigma_val * sqrt(T);



    d1 = (log(S / K) + (R + sigma_val * sigma_val / two) * T) / sigma_sqrt_t;

    d2 = d1 – sigma_sqrt_t;
```

```
    k_exp_minus_rt = K * exp(-R * T);



    phiD1 = phi(d1);

    phiD2 = phi(d2);



    call_option[GID] = S * phi_d1 – k_exp_minus_rt * phi_d2;



    phiD1 = phi(-d1);

    phiD2 = phi(-d2);

    put_option[GID] = k_exp_minus_rt * phi_d2 – S * phi_d1;

}
```

6.2.4. Mechanizm rozszerzeń

Implementacja algorytmu Blacka-Sholesa pokazuje, że dzięki APARAPI można uzyskać bardzo czytelny kod, który jest w pełni zintegrowany z językiem Java. Nie zawsze jednak tłumaczenie automatyczne z języka Java do OpenCL C daje dobre wyniki, często lepiej podać kod jądra obliczeniowego bezpośrednio w OpenCL C. Pakiet APARAPI pozwala na tego typu rozwiązanie.

Listing 6.9 przedstawia znaną z wcześniejszych przykładów procedurę `saxpy`. Nie będziemy tłumaczyć kodu jądra na Javę (ale warto takie ćwiczenie wykonać samemu), bo tworząc jądro obliczeniowe podamy tylko kod źródłowy. Mimo to, nadal będziemy korzystać z możliwości APARAPI, np. nie będziemy samodzielnie kopiować obszarów pamięci między obszarem pamięci gospodarza a urządzeniem obliczeniowym.

Listing 6.9. Procedura obliczeniowa `saxpy`

```
__kernel void saxpy( const float a, __global const float* x, __global float* y, const int N)

{

    const size_t i = get_global_id(0);



    if(i < N)

    {

        y[i] = a * x[i] + y[i];

    }
```

```
}
```

Listing 6.10 przedstawia kompletny kod w języku Java korzystający z metody obliczeniowej `saxpy` poprzez mechanizm rozszerzeń APARAPI. Kod źródłowy jądra obliczeniowego został umieszczony w oddzielnym pliku o nazwie *kernelSource.cl*. Deklaracja zewnętrznego jądra obliczeniowego jest wykonywana za pomocą interfejsu zdefiniowanego w następujący sposób:

```
@OpenCL.Resource(”./kernelSource.cl”)      interface      SaxpyWithResource      extends
OpenCL<SaxpyWithResource> {

    public SaxpyWithResource saxpy(//

        Range _range,

            @Arg(”a”) float a,

            @GlobalReadWrite(”x”) float[] x,

            @GlobalReadWrite(”y”) float[] y,

            @Arg(”N”) int N);

    }
```

Kluczowym elementem jest deklaracja metody `saxpy`, która musi mieć taką samą nazwę jak jądro obliczeniowe. Atrybuty `@Arg`, `@GlobalReadWrite` oznaczają typy argumentów stosowane w procedurze obliczeniowej, przy czym wartość prywatna to `@Arg`, a wartość globalna to `@GlobalReadWrite`.

W tej metodzie wywołania jądra obliczeniowego należy utworzyć urządzenie, choć nie trzeba przeprowadzać kompilacji procedury obliczeniowej. Odszukanie urządzenia obliczeniowego jest bardzo proste:

```
Device device = Device.best();
```

Podłączenie jądra do urządzenia realizuje kolejna pojedyncza linia kodu:

```
SaxpyWithResource saxpy = openclDevice.bind(SaxpyWithResource.class);
```

Uruchomienie jądra odbywa się inaczej niż w poprzednich przykładach APARAPI:

```
saxpy.saxpy(range, a, x, y, size);
```

Można powiedzieć, że mechanizm rozszerzeń naśladuje pakiet PyOpenCL, z tą różnicą, że korzystamy z języka Java. Liczba ułatwień jest jednak większa, bowiem, jak widać w listingu 6.10, zarządzanie pamięcią pozostało takie samo, jak w przykładach w rozdz. 6.2.1 czy 6.2.3.

Listing 6.10. Przykład wykorzystania jądra obliczeniowego za pomocą mechanizmu rozszerzeń

```
import com.amd.aparapi.Device;

import com.amd.aparapi.OpenCL;
```

```
import com.amd.aparapi.OpenCLDevice;

import com.amd.aparapi.Range;

public class SaxpyExample {

@OpenCL.Resource(”./kernelSource.cl”)      interface      SaxpyWithResource      extends
OpenCL<SaxpyWithResource> {

    public SaxpyWithResource saxpy(//

        Range _range,

        @Arg(”a”) float a,

        @GlobalReadWrite(”x”) float[] x,

        @GlobalReadWrite(”y”) float[] y,

        @Arg(”N”) int N);

    }



    public static void main(String[] args) {



        int size = 32;

        final float a = 5;

        float[] x = new float[size];

        float[] y = new float[size];

        for (int i = 0; i < size; i++) {

            x[i] = i;

            y[i] = 0;

        }

        Range range = Range.create(size);

        Device device = Device.best();

        if (device instanceof OpenCLDevice) {

            OpenCLDevice openclDevice = (OpenCLDevice) device;

            SaxpyWithResource saxpy = openclDevice.bind (SaxpyWithResource.class);

            saxpy.saxpy(range, a, x, y, size);
```

```
            for (int i = 0; i < size; i++) {

                System.out.println(”a=” + a + ” ” + x[i] + ” = ” + y[i]);

            }

        }

    }

}
```

6.3. Inne rozwiązania, VexCL i nie tylko

Inne biblioteki wykorzystujące OpenCL to np. ViennaCL oraz Magma, są to jednak biblioteki do realizacji obliczeń numerycznych. Podobnę rolę odgrywa także VexCL − pakiet wzorców dla języka C++, współpracujący z OpenCL i ułatwiający korzystanie z niego w wielu aspektach. Można go przyrównać do pakietu PyOpenCL. Jest to nowoczesny pakiet wymagający kompilatora zgodnego ze standardem C++11. Kompilator GCC również oferuje wszystkie niezbędne funkcje już od wersji 4.6.x, co pozwala na korzystanie z tego pakietu. Należy też dodać, iż VexCL wymaga OpenCL w wersji 1.2.

Nie ma potrzeby instalacji pakietu VexCL, ponieważ jest to pakiet wzorców, zawierający tylko pliki nagłówkowe. Po ściągnięciu aktualnej wersji z repozytorium GIT, wystarczy jedynie skopiować zawartość katalogów CL oraz VexCL do katalogu, gdzie znajdują się źródła naszego własnego projektu.

Listing 6.11 zawiera przykład sprawdzający, jakie urządzenia OpenCL są dostępne w systemie. Uzyskanie informacji o dostępnych urządzeniach obliczeniowych sprowadza się tylko do jednej linii kodu:

```
auto dev = device_list(Filter::All);
```

Obiekt `dev` zawierać będzie listę wszystkich urządzeń, które za pomocą pętli `for` możemy przejrzeć, a następnie wyświetlić informacje o urządzeniach. Co ważne, możliwe jest wskazanie opisowe, jakie urządzenie nas interesuje, np. utworzenie kontekstu dla kart produkowanych przez firmę NVIDIA z obsługą liczb o podwójnej precyzji przedstawia się następująco:

Listing 6.11. Lista dostępnych urządzeń OpenCL

```
vex::Context ctx(Filter::Name(”NVIDIA”) && Filter::DoublePrecision);

#include <iostream>

#include <vexcl/vexcl.hpp>



using namespace vex;
```

```
int main() {

    std::cout << ”OpenCL devices:” << std::endl << std::endl;

    auto dev = device_list(Filter::All);

    for (auto d = dev.begin(); d != dev.end(); d++) {

    std:cout << ” ” << d->getInfo<CL_DEVICE_NAME>() << std::endl

    << ” CL_PLATFORM_NAME = ”

        << cl::Platform(d->getInfo<CL_DEVICE_PLATFORM>()).

            getInfo<CL_PLATFORM_NAME>() << std::endl

        << ” CL_DEVICE_OPENCL_C_VERSION = ”

        << d->getInfo<CL_DEVICE_OPENCL_C_VERSION>() << std::endl

        << ” CL_DEVICE_MAX_COMPUTE_UNITS = ”

        << d->getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>() << std::endl

        << ” CL_DEVICE_HOST_UNIFIED_MEMORY = ”

        << d->getInfo<CL_DEVICE_HOST_UNIFIED_MEMORY>() << std::endl

        << ” CL_DEVICE_GLOBAL_MEM_SIZE = ”

        << d->getInfo<CL_DEVICE_GLOBAL_MEM_SIZE>() << std::endl

        << ” CL_DEVICE_LOCAL_MEM_SIZE = ”

        << d->getInfo<CL_DEVICE_LOCAL_MEM_SIZE>() << std::endl

        << ” CL_DEVICE_MAX_CLOCK_FREQUENCY = ”

        << d->getInfo<CL_DEVICE_MAX_CLOCK_FREQUENCY>() << std::endl

        << std::endl;

    }

}
```

Utworzenie kontekstu zawierającego m.in. kolejkę do realizacji obliczeń jest również uproszczone:

```
vex::Context ctx(Filter::All);
```

Sprawdzenie, czy istotnie odnaleziono odpowiednie urządzenie obliczeniowe, wykonujemy przez stwierdzenie, czy kolejka jest pusta:

```
if (ctx.queue().empty()) {

    // brak urządzeń OpenCL
```

```
}
```

Zarządzanie obiektami pamięciowymi także zostało uproszczone, np. utworzenie zmiennej o `N` elementach typu `double` w pamięci operacyjnej CPU oraz wypełnienie jej wartościami losowymi realizujemy, wykorzystując lambda wyrażenie w następujący sposób:

```
std::vector<double> host_vec1(N);

std::generate(host_vec1.begin(), host_vec1.end(), [ ]( ) {

    return double(rand()) / RAND_MAX;

});
```

Umieszczenie nowo powołanej zmiennej w pamięci urządzenia obliczeniowego OpenCL można zrealizować, deklarując potrzebny obiekt:

```
vex::vector<double> x ( ctx.queue(), host_vec1 );
```

Realizacja wielu typowych operacji, np. dodawania dwóch wektorów, sprowadza się do następującego wyrażenia:

```
x = y + z;
```

co jest możliwe dzięki wzorcom oraz operatorom, które zostały odpowiednio przeciążone. Proces kopiowania zawartości zmiennej z urządzenia OCL do pamięci operacyjnej CPU również został sprowadzony do prostego wywołania `copy`:

```
copy(x, host_vec1);
```

W dość prosty sposób można też deklarować własne funkcje, np. funkcję do obliczania sumy kwadratu elementów:

```
VEX_FUNCTION(double, squared_fnc, (double, x)(double, y),

    return x * x + y * y;

    );
```

Użycie tej funkcji na zmiennych `X` oraz `Y` umieszczonych w pamięci urządzenia sprowadza się do następującego kodu:

```
Z = sin(squared_fnc(X, Y));
```

Jak widać, funkcja `squared_fnc` została użyta jak każda inna funkcja w programie. Choć trzeba pamiętać, że treść funkcji określanej przez makro `VEX_FUNCTION` musi być zgodna z językiem OpenCL C.

Oprócz pakietu VexCL istnieje wiele dodatkowych rozwiązań. W internecie można znaleźć wiele mniejszych i większych pakietów, które z pewnością ułatwią rozwiązywanie problemów i tworzenie programów w OpenCL. Warto zwrócić uwagę na następujące rozwiązania[4], jakie obecnie są dostępne dla standardu OpenCL:

• pakiet ViennaCL, http://viennacl.sourceforge.net/

• pakiet do algebry liniowej Magma, http://icl.cs.utk.edu/magma/

• CUDAfy, mimo swojej nazwy, pakiet ten obsługuje także OpenCL i jest rozwiązaniem podobnym do APARAPI, ale dla języka C#, https://cudafy.codeplex.com/

• Boost.Compute podobne rozwiązanie jak VexCL, ale silnie zintegrowane z pakietem Boost: http://kylelutz.github.io/compute/, w przyszłości być może stanie się elementem pakietu Boost dla C++,

• POCL, otwarta implementacja OpenCL, tylko dla procesorów: http://portablecl.org/

Przypisy

[1] Techniczne informacje, jak zainstalować język Python oraz pakiet PyOpenCL można odszukać w „Dodatku B. Instalacja pakietu OpenCL dla języka Python”.

[2] Pakiet PyOpenCL został udostępniony na licencji o nazwie „MIT/XConsortium License”. Jest to bardzo liberalna licencja, która zezwala na wiele rzeczy, min. powielanie, łączenie, czy nawet licencjonowanie pakietu PyOpenCL. Jedynym wymaganiem jest oczywiście jawne wskazanie Autora PyOpenCL oraz wskazanie dokumentu o prawach co do pakietu PyOpenCL.

[3] Jeśli mamy więcej niż jedną kartę graficzną, bądź też mamy zainstalowaną obsługę OpenCL z poziomu procesora głównego komputera, może się okazać, iż platforma o indeksie zero to właśnie platforma OpenCL obsługiwana przez procesor główny.

[4] Lista podanych bibliotek powstała w czerwcu roku 2014, i może się okazać iż część adresów WWW została zmieniona lub niestety nie jest już aktualna.

7 Zamiast zakończenia

Materiał przedstawiony we wcześniejszych rozdziałach nie opisuje wszystkich elementów OpenCL. Pozostał do omówienia jeszcze dość istotny element standardu, mechanizm zdarzeń. Nie zostały też omówione wszystkie elementy języka OpenCL C. Można byłoby przedstawić więcej algorytmów równoległych, np. sumowanie prefiksowe. Standard OpenCL jest dostępny też dla urządzeń mobilnych. Jednakże, zamiast wymieniać na co warto zwrócić jeszcze uwagę, w ostatnim rozdziale zostanie krótko przedstawione narzędzie CodeXL oferowane przez firmę AMD.

Program CodeXL jest dostępny dla systemów operacyjnych Windows oraz Linux. Jest to program, a raczej środowisko do śledzenia programów napisanych dla CPU. CodeXL to także narzędzie do analizy procedur obliczeniowych OpenCL. Możliwe jest profilowanie, czyli przeprowadzenie analizy wydajności pracy programu dla CPU oraz dla procedur obliczeniowych uruchomionych w GPU.

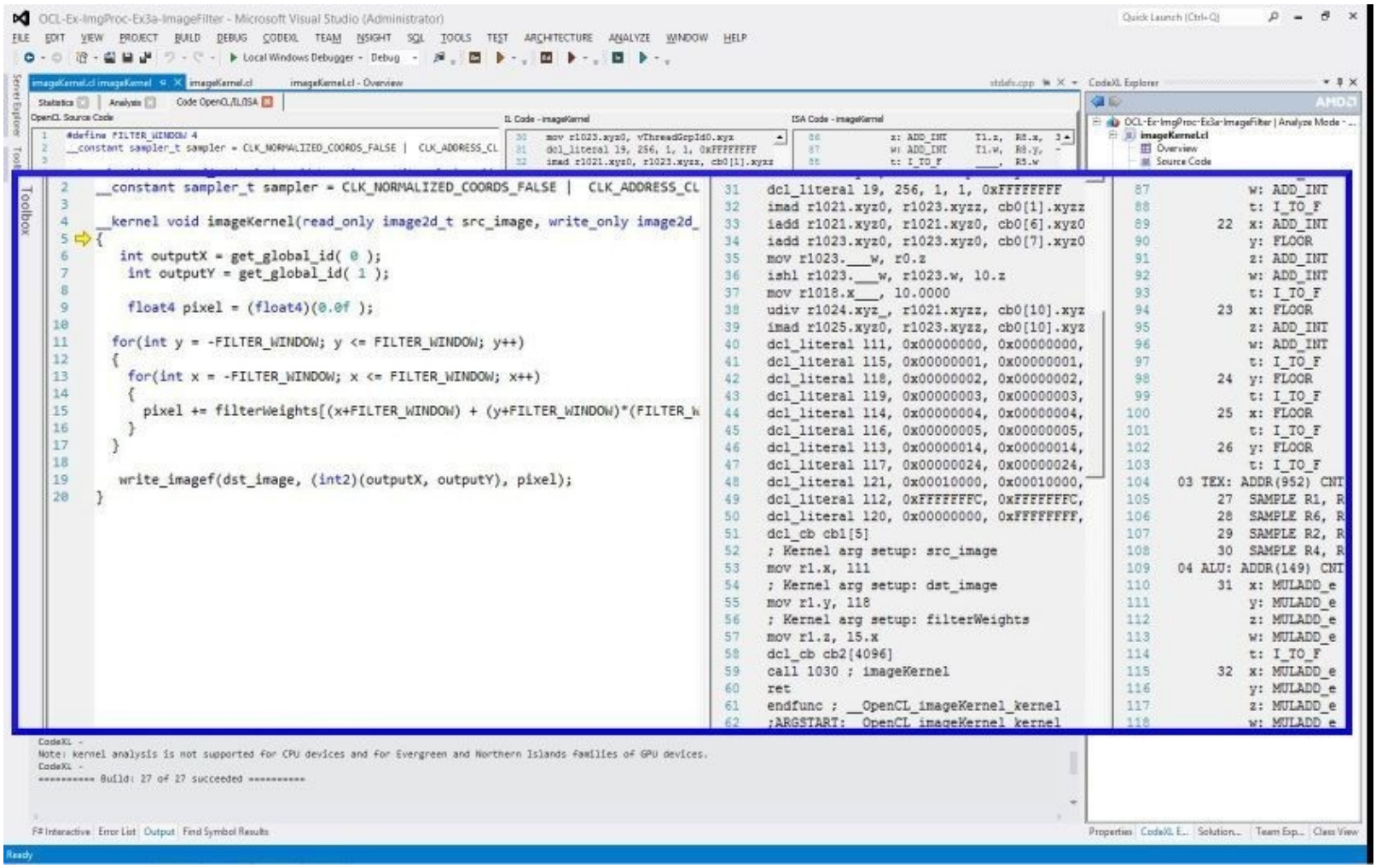
Środowisko CodeXL potrafi działać samodzielnie oraz jako tzw. wtyczka w środowisku Visual Studio. Jednak jest to oprogramowanie AMD i pracuje tylko z kartami graficznymi tego producenta. W dalszej części wykorzystamy to narzędzie, aby sprawdzić wydajność pracy przykładu „Gra w życie” (rozdz. 5.6), postaramy się też przyspieszyć działanie tego prosesu poprzez zmianę parametrów siatki obliczeniowej.

Ogólne spojrzenie na program CodeXL uruchomiony jako wtyczka do programu Visual Studio przedstawiono na rys. 7.1. Widać tam jądro obliczeniowe, pokazane w rozdz. 5.9 o przetwarzaniu obrazów, w postaci kodu w OpenCL C, ale także jako procedurę po kompilacji do tzw. kodu pośredniego oraz kodu maszynowego, który jest wykonywany fizycznie przez kartę graficzną.

1.1.1. Analiza wydajności symulacji przykładu „Gra w życie”

Możliwości programu CodeXL jest zbyt wiele, aby je tu dokładnie opisywać. Podany dalej przykład to podstawowy scenariusz, który nie zabierze dużo czasu i pozwoli sprawdzić wydajność, a dokładnie szybkość wybranego fragmentu tworzonej aplikacji.

Naszą analizę zaczniemy od uruchomienia programu do symulacji gry w życie w trybie Profile Mode i np. wygenerujemy 64 klatki symulacji gry. Pozwoli to sprawdzić czasy obliczeń poszczególnych etapów. Rezultat przedstawiono na rys. 7.2. Jak widać, pierwsze uruchomienie pokazało od razu, że można dokonać pewnych zmian w konfiguracji siatki obliczeniowej. CodeXL sugeruje tę zmianę (rys. 7.2a), gdyż w wyniku analizy wydajności, poszczególne wywołania jądra obliczeniowego nie dały poprawnego obciążenia jednostek roboczych.

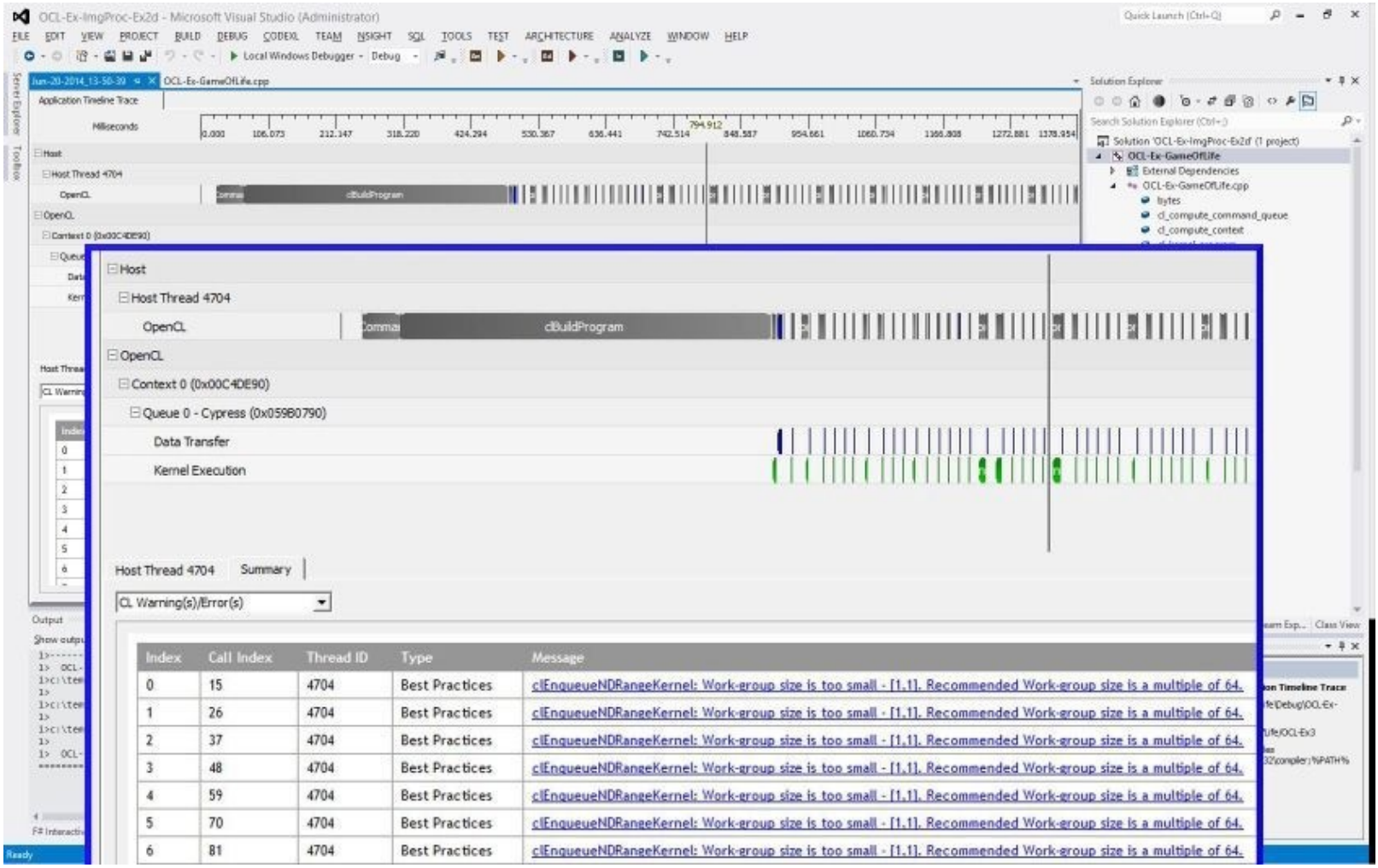


Rys. 7.1. Analiza pliku źródłowego jądra obliczeniowego realizującego filtr dla obrazu dwuwymiarowego (wersja kolorowa dostępna w wydaniu elektronicznym)

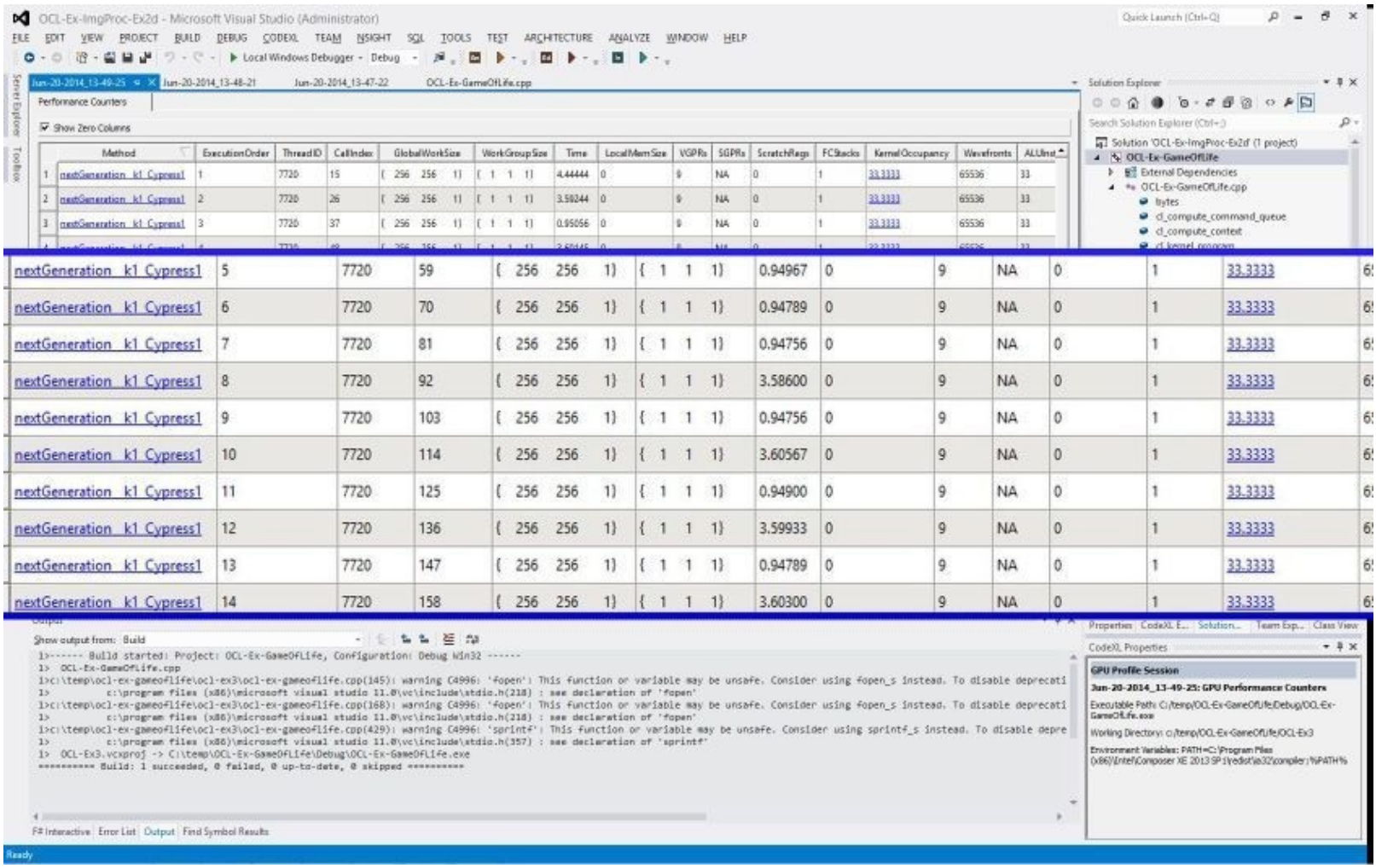
Na rysunku 7.2b pokazano też czasy działania symulacji dla poszczególnych wywołań jądra obliczeniowego. Czas ten, jak widać nie jest stały, choć symulacja jednej klatki o wymiarach 256 × 256 trwa około 1 ms (milisekundy). Zdarzają się jednak przypadki, kiedy zabiera to 3 ms, a nawet 7 ms. Widać też, że jądro obliczeniowe nie wykorzystuje poprawnie całej mocy obliczeniowej oferowanej przez GPU. Świadczy o tym wartość w kolumnie KernelOccupancy, czyli procentowe obciążenie poszczególnych jednostek roboczych. Wartość 33.33% oznacza, że zaledwie jedna trzecia mocy obliczeniowej została wykorzystana.

Zmiana konfiguracji siatki obliczeniowej pozwoli na poprawę wydajności i skrócenie czasu działania symulacji gry w życie. Ponowne uruchomienie dla siatki o wymiarze 2 × 2 pozwoli ustabilizować czas symulacji każdej iteracji do około 1 ms (milisekundy), lecz dopiero zwiększenie siatki lokalnej do wymiarów 4 × 4, skróci czas do około 0.3 ms, a dla siatki 8 × 8 czas symulacji jednej klatki wynosi 0.1 ms. Choć skrócił się czas obliczeń już prawie dziesięciokrotnie, to nadal wartość KernelOccupancy sugeruje, że jądro obliczeniowe nie wykorzystuje do końca całej dostępnej mocy obliczeniowej.

a)


OCL-Ex-ImgProc-Ex2d - Microsoft Visual Studio (Administrator)
Application Timeline Trace
Milliseconds
Host
Host Thread 4704
OpenCL
clBuildProgram
Context 0 (0x00C4DE90)
Queue 0 - Cypress (0x059B0790)
Data Transfer
Kernel Execution
Host Thread 4704
Summary
CL Warning(s)/Error(s)
Index | Call Index | Thread ID | Type | Message
0 | 15 | 4704 | Best Practices | clEnqueueNDRangeKernel: Work-group size is too small - [1,1]. Recommended Work-group size is a multiple of 64.
1 | 26 | 4704 | Best Practices | clEnqueueNDRangeKernel: Work-group size is too small - [1,1]. Recommended Work-group size is a multiple of 64.
2 | 37 | 4704 | Best Practices | clEnqueueNDRangeKernel: Work-group size is too small - [1,1]. Recommended Work-group size is a multiple of 64.
3 | 48 | 4704 | Best Practices | clEnqueueNDRangeKernel: Work-group size is too small - [1,1]. Recommended Work-group size is a multiple of 64.
4 | 59 | 4704 | Best Practices | clEnqueueNDRangeKernel: Work-group size is too small - [1,1]. Recommended Work-group size is a multiple of 64.
5 | 70 | 4704 | Best Practices | clEnqueueNDRangeKernel: Work-group size is too small - [1,1]. Recommended Work-group size is a multiple of 64.
6 | 81 | 4704 | Best Practices | clEnqueueNDRangeKernel: Work-group size is too small - [1,1]. Recommended Work-group size is a multiple of 64.

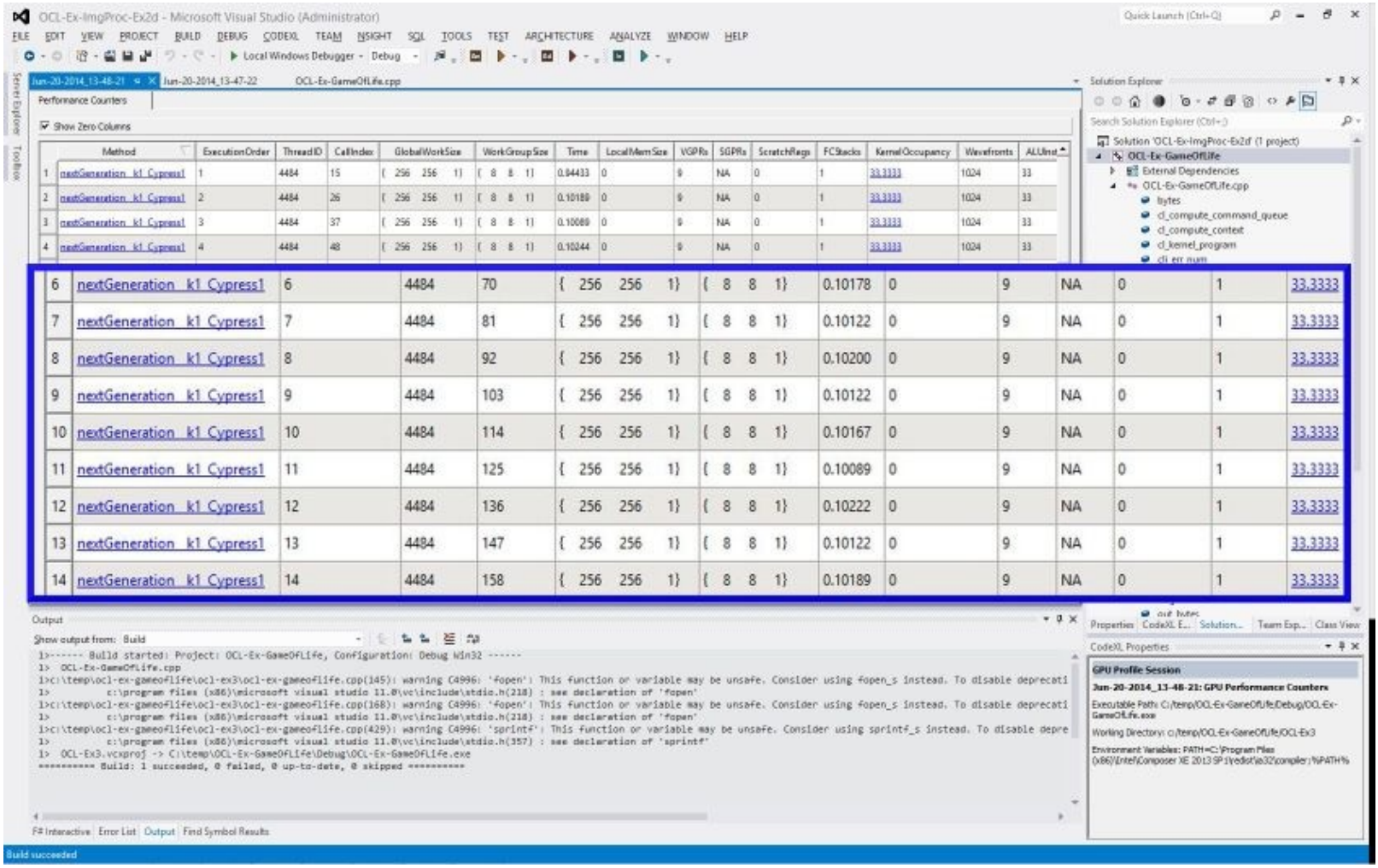
b)



	Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy	Wavefronts	ALUInst
1	nextGeneration_k1_Cypress1	1	7720	15	{ 256 256 1}	{ 1 1 1}	4.44444	0	9	NA	0	1	33.3333	65536	33
2	nextGeneration_k1_Cypress1	2	7720	26	{ 256 256 1}	{ 1 1 1}	3.58244	0	9	NA	0	1	33.3333	65536	33
3	nextGeneration_k1_Cypress1	3	7720	37	{ 256 256 1}	{ 1 1 1}	0.95056	0	9	NA	0	1	33.3333	65536	33
	nextGeneration_k1_Cypress1	5	7720	59	{ 256 256 1}	{ 1 1 1}	0.94967	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	6	7720	70	{ 256 256 1}	{ 1 1 1}	0.94789	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	7	7720	81	{ 256 256 1}	{ 1 1 1}	0.94756	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	8	7720	92	{ 256 256 1}	{ 1 1 1}	3.58600	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	9	7720	103	{ 256 256 1}	{ 1 1 1}	0.94756	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	10	7720	114	{ 256 256 1}	{ 1 1 1}	3.60567	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	11	7720	125	{ 256 256 1}	{ 1 1 1}	0.94900	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	12	7720	136	{ 256 256 1}	{ 1 1 1}	3.59933	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	13	7720	147	{ 256 256 1}	{ 1 1 1}	0.94789	0	9	NA	0	1	33.3333	65	
	nextGeneration_k1_Cypress1	14	7720	158	{ 256 256 1}	{ 1 1 1}	3.60300	0	9	NA	0	1	33.3333	65	

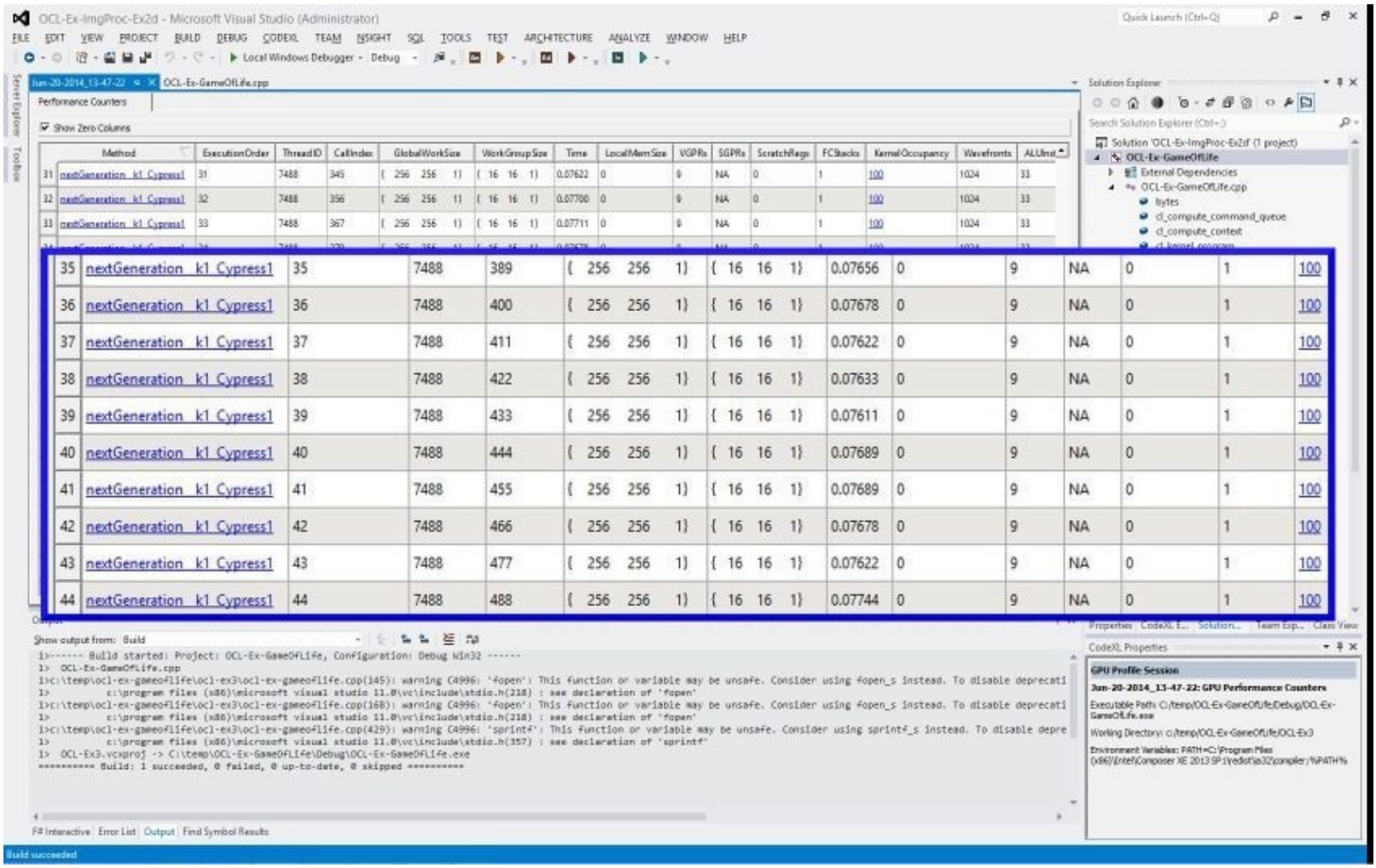
Rys. 7.2. Analiza jądra obliczeniowego symulującego „Grę w życie”: a) dla siatki lokalnej 1 × 1, b) czasy działania dla poszczególnych wywołań jądra obliczeniowego (wersje kolorowe dostępne w wydaniu elektronicznym)

a)



	Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy	Wavefronts	ALUInst
1	nextGeneration_k1_Cypress1	1	4484	15	{ 256 256 1}	{ 8 8 1}	0.04433	0	9	NA	0	1	33.3333	1024	33
2	nextGeneration_k1_Cypress1	2	4484	26	{ 256 256 1}	{ 8 8 1}	0.10189	0	9	NA	0	1	33.3333	1024	33
3	nextGeneration_k1_Cypress1	3	4484	37	{ 256 256 1}	{ 8 8 1}	0.10089	0	9	NA	0	1	33.3333	1024	33
4	nextGeneration_k1_Cypress1	4	4484	48	{ 256 256 1}	{ 8 8 1}	0.10244	0	9	NA	0	1	33.3333	1024	33
6	nextGeneration_k1_Cypress1	6	4484	70	{ 256 256 1}	{ 8 8 1}	0.10178	0	9	NA	0	1	33.3333		
7	nextGeneration_k1_Cypress1	7	4484	81	{ 256 256 1}	{ 8 8 1}	0.10122	0	9	NA	0	1	33.3333		
8	nextGeneration_k1_Cypress1	8	4484	92	{ 256 256 1}	{ 8 8 1}	0.10200	0	9	NA	0	1	33.3333		
9	nextGeneration_k1_Cypress1	9	4484	103	{ 256 256 1}	{ 8 8 1}	0.10122	0	9	NA	0	1	33.3333		
10	nextGeneration_k1_Cypress1	10	4484	114	{ 256 256 1}	{ 8 8 1}	0.10167	0	9	NA	0	1	33.3333		
11	nextGeneration_k1_Cypress1	11	4484	125	{ 256 256 1}	{ 8 8 1}	0.10089	0	9	NA	0	1	33.3333		
12	nextGeneration_k1_Cypress1	12	4484	136	{ 256 256 1}	{ 8 8 1}	0.10222	0	9	NA	0	1	33.3333		
13	nextGeneration_k1_Cypress1	13	4484	147	{ 256 256 1}	{ 8 8 1}	0.10122	0	9	NA	0	1	33.3333		
14	nextGeneration_k1_Cypress1	14	4484	158	{ 256 256 1}	{ 8 8 1}	0.10189	0	9	NA	0	1	33.3333		

b)



Rys. 7.3. Analiza jądra obliczeniowego symulującego „Grę w życie”: a) czas generowania następnej iteracji dla siatki lokalnej 8 × 8, b) rozmair siatki lokalnej 16 × 16 (wersje kolorowe dostępne w wydaniu elektronicznym)

Wystarczy jednak zwiększyć siatkę lokalną do rozmiaru 16 × 16 (wyniki wydajności zostały przedstawione na rys. 7.3b), aby skrócić czas obliczeń do 0.07 ms. Modyfikacja, co ważne, podniosła wartość KernelOccupancy do 100%. Ten prosty zabieg dotyczący poprawności konfiguracji siatki obliczeniowej pozwolił przyspieszyć obliczenia piętnastokrotnie, bez dodatkowego wysiłku, gdyż w programie źródłowym zmienione zostały zaledwie dwie wartości odnoszące się do wymiaru siatki lokalnej. Użycie programu CodeXL umożliwia weryfikację zmian w dość dokładny sposób.

Za pomocą CodeXL można sprawdzić wiele innych parametrów, np. czasy trwania poszczególnych operacji kopiowania danych z pamięci urządzenia do pamięci gospodarza. Umożliwi to dokładne obliczenie szybkości wykonywania poszczególnych etapów realizacji tego zadania. Pozostało już tylko zachęcić Czytelnika do wybróbowania tego narzędzia we własnych, nawet bardzo prostych projektach, które wykorzystują technologię OpenCL, aby przetestować daną aplikację.

Dodatek A. Kompilacja przykładów

Większość kodów źródłowych przykładów zaprezentowanych w tej książce to pojedyncze pliki źródłowe w języku C/C++, więc sposób ich kompilacji można sprowadzić do jednego polecenia wydanego z poziomu konsoli. Za przykład może posłużyć kod z rozdz. 5.10, gdzie podany został przykład współpracy OpenCL wraz OpenGL. Jeśli plik źródłowy nosi nazwę *OCLandOGL.cpp*, to jego kompilacja może zostać wykonana za pomocą polecenia:

```
g++ –o GLinterop OCLandOGL.cpp –lOpenCL –lfreeglut –lOpenGL32 –lglew32 –lglu32
```

Powyższy przykład zakłada obecność dodatkowych bibliotek, takich jak Freeglut, Glew32, jednak kompilacja będzie taka sama w systemie Linux, MacOS, a nawet Windows. Kompilator z dystrybucji MinGW, nawet jeśli wskaże się bibliotekę OpenCL.lib np. z SDK OpenCL od producentów NVIDIA czy AMD, poprawnie zbuduje postać binarną.

Wygodnie jednak skorzystać ze skryptu dla programu makefile, jego postać źródłowa została zaprezentowana w listingu A.1. Głównym elementem jest zmienna o nazwie `PROJ`. Powinna ona zawierać nazwę pliku źródłowego, ale bez rozszerzenia. Rozszerzenie `cpp` zostanie dodane przez regułę:

```
$(PROJ): $(PROJ).cpp

    $(CC) $(CFLAGS) –o $@ $^ $(LIBS)
```

Reguła wykona również kompilację kodu źródłowego oraz utworzy plik wynikowy. Stosowane są dwie zmienne o nazwach `$@` oraz `$^`. Zmienna `$@` reprezentuje docelową nazwę pliku wynikowego, a zmienna `$^` to aktualnie przetwarzany plik, czyli kod źródłowy przykładu.

Nazwę kompilatora można ustalić za pomocą zmiennej `CC`, a dodatkowe opcje w zmiennej `CFLAGS`, natomiast dodatkowe biblioteki powinny zostać wymienione w zmiennej `LIBS`.

Podany skrypt jest naturalnie bardzo prosty, jednak spełnia swoje podstawowe zadanie, czyli kompilację przykładów podanych w rozdz. 4 oraz 5.

Listing A.1. Podstawowy skrypt makefile do przeprowadzania kompilacji

```
PROJ=OCL-Ex-GameOfLife



CC=g++

CFLAGS=-O2

LIBS=-lOpenCL




.PHONY: clean
```

```
$(PROJ): $(PROJ).cpp

    $(CC) $(CFLAGS) –o $@ $^ $(LIBS)

clean:

    rm $(PROJ)
```

Kompilacja programów w języku Java może również zostać wykonana za pomocą skryptu zaprezentowanego w listingu A.1. Można też wydać polecenie kompilacji z poziomu konsoli. Jeśli chcemy skompilować przykład o nazwie *VectorSum.java*, to wystarczy wydać jedno polecenie:

```
javac –g –cp .;aparapi.jar VectorSum.java
```

Pakiet `aparapi.jar` naturalnie musi się znaleźć w tym samym katalogu co plik źródłowy. Uruchomienie przykładów w Javie również można wykonać z poziomu konsoli, wystarczy wydać polecenie:

```
java –cp aparapi.jar;. VectorSum
```

Przykłady podane w języku Python nie wymagają kompilacji, można je uruchamiać wprost z poziomu konsoli:

```
python fileex1.py
```

Wygodnie jest też zastosować środowisko Spyder lub dowolne inne IDE dla języka Python.

Dodatek B. Instalacja pakietu OpenCL dla języka Python

Sposób instalacji pakietu Python OpenCL zależy bezpośrednio od używanego systemu operacyjnego. W przypadku Windows najwygodniej będzie skorzystać z pakietu dystrybucji o nazwie PythonXY (dostępnego pod adresem: *https://code.google.com/p/pythonxy/*). Pakiet ten jest dystrybuowany również jako plik instalacyjny. Pakiet PythonXY oferuje, co prawda, starszą odmianę języka Python w wersji 2.7, która nadal jest główną wersją tego języka, mimo iż już od dłuższego czasu dostępna jest wersja 3.x. Dostępne jest też bardzo dobre środowisko pracy Spyder oraz wiele dodatkowych pakietów, w tym także pakiet NumPy (PyOpenCL korzysta bezpośrednio z tego pakietu). W podstawowej wersji nie ma dostępnego pakietu PyOpenCL, ale znajduje się on na stronie projektu PythonXY w grupie pakietów dodatkowych. Instalując w pierwszej kolejności pakiet PythonXY, a później PyOpenCL, otrzymuje się kompletne środowisko do pracy z językiem Python wraz z obsługą standardu OpenCL.

W przypadku systemu Linux warto przeprowadzić kompilację PyOpenCL samodzielnie, bo wbrew pozorom nie jest to trudny proces. Po ściągnięciu źródeł należy rozpakować archiwum:

```
tar xvfz pyopencl-2014.1.tar.gz
```

Archiwum można ściągnąć ze strony domowej projektu PyOpenCL lub ze strony *https://pypi.python.org/pypi/pyopencl*. Po tym można od razu przejść do katalogu z rozpakowanym PyOpenCL i dokonać konfiguracji oraz instalacji pakietu PyOpenCL:

```
python configure.py

su – c ”make install”
```

Jeżeli podczas konfiguracji okaże się, że nie można odszukać plików nagłówkowych oraz bibliotek OpenCL, należy wskazać katalog, w którym znajdują się odpowiednie pliki. Jeśli na przykład korzysta się z pakietu CUDA firmy NVIDIA, zwiera on także odpowiednie pliki nagłówkowe OpenCL. W poniższym przykładzie założono, iż pakiet CUDA został zainstalowany w katalogu */opt/cuda*:

```
python configure.py \

    –-cl-inc-dir=/opt/cuda/include \

    –-cl-lib-dir=/opt/cuda/lib \

    –-cl-libname=OpenCL
```

Może też się okazać, że nie mamy pakietu NumPy, wtedy należy zainstalować pomocniczy pakiet `setuptools` dla Pythona oraz sam pakiet NumPy, co powinno się udać za pomocą dwóch poleceń:

```
su –c ”python ez_setup.py”

su –c ”easy_install numpy”
```

W przypadku pakietu NumPy warto sprawdzić, czy dystrybucja systemu Linux, której używamy, nie

oferuje gotowego pakietu, zazwyczaj bowiem odpowiedni pakiet jest dostępny i nie ma potrzeby samodzielnej kompilacji.

W procesie instalacji oraz kompilacji PyOpenCL pomocna może się także okazać strona dokumentacji pakietu PyOpenCL, którą znajdziemy pod adresem: *http://wiki.tiker.net/PyOpenCL/Installation*. Warto też sprawdzić przykłady zawarte w kodzie źródłowym PyOpenCL.

Dodatek C. Opis funkcji pomocniczych

W tym dodatku przedstawione zostały funkcje, które mają charakter pomocniczy, jak np. zapis bądź też odczyt danych do określonego formatu pliku, a także funkcje pomocnicze konwertujące stałe oznaczające typy kolorów do wartości znakowych.

C.1. Zapis do formatu PPM

Listing C.1 przedstawia krótką funkcję, która zawartość globalnej tablicy interpretuje jako informacje o kolorze w formacie RGB i zapisuje je do pliku w formacie PPM. Choć postać formatu PPM jest niezwykle prosta i pozbawiona kompresji danych, to można pliki tego formatu otwierać np. w programie GIMP, czy też IrfanView.

Ogólnie, plik w formacie PPM można podzielić na dwie części: nagłówek oraz dane o pikselach. Nagłówek to trzy linie tekstu np.:

```
P6

512 512

255
```

gdzie: P6 – identyfikator formatu; dwie liczby zapisane jako wyrażenie tekstowe w linii drugiej – szerokość oraz wysokość; trzecia liczba – maksymalna wartość składowej koloru (255 oznacza, że poszczególne składowe koloru będą kodowane jako wartości 8-bitowe); pozostała część pliku to wartości binarne w postaci trójek RGB, gdzie R oznacza kolor czerwony, G – zielony oraz B – niebieski.

Zapis danych polega na tym, aby przenieść trójki RGB wprost jako wartości binarne do pliku, liczba trójek RGB musi naturalnie odpowiadać rozmiarowi obrazu, tj. iloczynowi wysokości i szerokości.

Listing C.1. Zapis danych do pliku graficznego w formacie PPM

```
void save_a_as_ppm(char *fname)

{

    FILE *fp = fopen(fname, ”w”);

    fprintf(fp, ”P6\n%d %d\n255\n”, WIDTH, HEIGHT);

    for (int j = 0; j < WIDTH * HEIGHT; ++j)

    {

        static unsigned char color[3];



        color[0] = get_r( a[j] ); /* składowa czerwona */
```

```
        color[1] = get_g( a[j] ); /* składowa zielona */

        color[2] = get_b( a[j] ); /* składowa niebieska */

        fwrite(color, 1, 3, fp);

    }

    fclose(fp);

}
```

Wartości tablicy globalnej w naszym przypadku są liczbami typu `int`, więc są to wartości 32-bitowe. W funkcji do zapisu wykorzystywane są jeszcze dodatkowe funkcje o nazwach np. `get_r`, `get_g`, `get_b`, których zadaniem będzie odczytanie składowej wskazanego koloru, np. `get_r` odczytuje kolor czerwony.

Odczyt zawsze polega na zastosowaniu odpowiedniej maski bitowej (dla koloru czerwonego mamy 0xFF0000) i przesunięciu bitowym (dla składowej czerwonej wymagane jest przesunięcie o szesnaście bitów). Podane funkcje oczywiście mają dość uproszczoną postać, nie rozróżniają, czy bit najmniej znaczący znajduje się po lewej, czy prawej stronie zapisu bitowego.

Listing C.2. Odczyt składowych koloru z piksela opisywanego liczbą 32-bitową

```
unsigned char get_r(unsigned int v){

    unsigned int r;

    r = v & 0xFF0000;

    r>>=16;

    return (unsigned char)r;

}



unsigned char get_g(unsigned int v){

    unsigned int r;

    r = v & 0x00FF00;

    r>>=8;

    return (unsigned char)r;

}



unsigned char get_b(unsigned int v){

    unsigned int r;
```

```
    r = v & 0x0000FF;

    return (unsigned char)r;

}
```

C.2. Konwersja kodów formatu koloru do wartości znakowych

Funkcja przedstawiona w listingu C.3 jest wykorzystywana w rozdz. 4.1 do konwersji stałych opisujących rodzaje kanałów stosowane do opisu kolorów (a także same typy kanałów) na odpowiednie ciągi znaków. Sprowadza się to tylko do kilkunastu instrukcji if, które sprawdzają podany argument i zwracają odpowiedni ciąg znaków.

Listing C.3. Konwersja stałych opisujących dostępne typy formatów obrazu do ciągów znakowych

```
const char* imgFormatToChars(cl_uint imgFormatCode) {

    if (imgFormatCode == CL_R) return ”CL_R”;

    if (imgFormatCode == CL_A) return ”CL_A”;

    if (imgFormatCode == CL_RG) return ”CL_RG”;

    if (imgFormatCode == CL_RA) return ”CL_RA”;

    if (imgFormatCode == CL_RGB) return ”CL_RGB”;

    if (imgFormatCode == CL_RGBA) return ”CL_RGBA”;

    if (imgFormatCode == CL_BGRA) return ”CL_BGRA”;

    if (imgFormatCode == CL_ARGB) return ”CL_ARGB”;

    if (imgFormatCode == CL_INTENSITY) return ”CL_INTENSITY”;

    if (imgFormatCode == CL_LUMINANCE) return ”CL_LUMINANCE”;



    if (imgFormatCode == CL_SNORM_INT8) return ”CL_SNORM_INT8”;

    if (imgFormatCode == CL_SNORM_INT16) return ”CL_SNORM_INT16”;

    if (imgFormatCode == CL_UNORM_INT8) return ”CL_UNORM_INT8”;

    if (imgFormatCode == CL_UNORM_INT16) return ”CL_UNORM_INT16”;

    if (imgFormatCode == CL_UNORM_SHORT_565) return ”CL_UNORM_SHORT_565”;

    if (imgFormatCode == CL_UNORM_SHORT_555) return ”CL_UNORM_SHORT_555”;

    if (imgFormatCode == CL_UNORM_INT_101010) return ”CL_UNORM_INT_101010”;

    if (imgFormatCode == CL_SIGNED_INT8) return ”CL_SIGNED_INT8”;

    if (imgFormatCode == CL_SIGNED_INT16) return ”CL_SIGNED_INT16”;
```

```
    if (imgFormatCode == CL_SIGNED_INT32) return ”CL_SIGNED_INT32”;

    if (imgFormatCode == CL_UNSIGNED_INT8) return ”CL_UNSIGNED_INT8”;

    if (imgFormatCode == CL_UNSIGNED_INT16) return ”CL_UNSIGNED_INT16”;

    if (imgFormatCode == CL_UNSIGNED_INT32) return ”CL_UNSIGNED_INT32”;

    if (imgFormatCode == CL_HALF_FLOAT) return ”CL_HALF_FLOAT”;

    if (imgFormatCode == CL_FLOAT) return ”CL_FLOAT”;



    return ”format nieznany”;

}
```

C.3. Zapis i odczyt danych w formacie PNG

Przykłady dotyczące przetwarzania obrazów, choć nie tylko, bo również przykład odnoszący się do „Gry w życie” oraz generowanie zbioru Mandelbrota, korzystają z prostego formatu graficznego o nazwie PPM, który został już krótko przedstawiony w rozdziale C.1. Jest możliwe, dzięki obecności pakietu libpng, dodanie do przykładów możliwości obsługi plików w formacie PNG.

Odczyt oraz zapis tego typu plików, zwłaszcza w formacie RGB, wymaga tylko kilku linii kodu.

Listing C.4 przedstawia funkcję `load_png_data`, która odczytuje dane z pliku PNG. Do jej wywołania potrzeba deklaracji trzech zmiennych:

```
unsigned char *input_data;

size_t _tmp_width;

size_t _tmp_height;
```

Pierwsza oznacza bufor, do którego zostanie przydzielona pamięć oraz umieszczone informacje o obrazie jako trójki RGB bądź czwórki RGB z kanałem alfa, tj. RGBA (kolejność składowych można zmienić). Dwie pozostałe to szerokość oraz wysokość obrazu, jaki zostanie wczytany. Uruchomienie procedury może przebiegać w następujący sposób:

```
load_png_data(”InputImage.png”, &(png_bytep)input_data, &_tmp_width, &_tmp_height);
```

Wszystkie czynności, jakie są wykonywane w tej funkcji, zostały przedstawione w listingu C.4

Listing C.4. Odczyt danych graficznych reprezentujących obraz z pliku w formacie PNG

```
int load_png_data(const char* filename, png_bytep* input, size_t* w, size_t* h)

{

    int i;
```

```
    unsigned char chn_colors;

    FILE *png_input;

    png_input = fopen(filename, ”rb”);

    if( png_input == NULL) {

        return – 1;

    }



     png_structp  png_ptr  =  png_create_read_struct  (PNG_LIBPNG_VER_STRING,  NULL,  NULL,
NULL);

    png_infop info_ptr = png_create_info_struct(png_ptr);

    png_init_io(png_ptr, png_input);

    png_read_info(png_ptr, info_ptr);



    *w = png_get_image_width(png_ptr, info_ptr);

    *h = png_get_image_height(png_ptr, info_ptr);




    chn_colors = png_get_channels(png_ptr, info_ptr);



    *input = (png_bytep)malloc(*h * png_get_rowbytes(png_ptr, info_ptr));

    for(i=0; i<*h; i++) {

           png_read_row(png_ptr,  *input  +  i  *  png_get_rowbytes  (png_ptr,  info_ptr),
NULL);

    }



    png_read_end(png_ptr, info_ptr);

    png_destroy_read_struct(&png_ptr, &info_ptr, (png_infopp)NULL);

    fclose(png_input);



    return 0;
```

```
}
```

W bardzo podobny sposób przedstawia się zapis danych do formatu PNG. Przykładowe wywołanie odpowiedniej funkcji, która zapisze dane bez kanału alfa jest następujące:

```
save_as_png_without_alpha_channel( ”image_output.png”, data_for_png, WIDTH , HEIGHT);
```

Tym razem wskaźnik `data_for_png` musi zawierać już wcześniej przygotowane dane zgodnie ze sposobem zapisu danych do PNG, ale w naszym przypadku zazwyczaj jest to format RGB z opcjonalnym kanałem alfa. Funkcja przedstawiona w listingu C.5 nie zapisuje kanału alfa. Zmiana, jaką trzeba wykonać, aby dodać kanał alfa, to wywołanie funkcji opisującej format danych:

```
png_set_IHDR(png_ptr, info_ptr, w, h, 8,

        PNG_COLOR_TYPE_RGB_ALPHA , PNG_INTERLACE_NONE,

        PNG_COMPRESSION_TYPE_BASE, PNG_FILTER_TYPE_BASE);
```

Dane wskazywane przez wskaźnik `data_for_png` czy parametr funkcji `data` muszą zawierać poprawnie zbudowane trójki dla danych RGB lub czwórki dla RGBA.

Listing C.5. Zapis danych graficznych do pliku w formacie PNG

```
void save_as_png_without_alpha_channel(const char* filename, png_bytep data, size_t w,
size_t h)

{



    int i;



    FILE *png_output;

    png_output = fopen(filename, ”wb”);

    if(( png_output == NULL) {

        return – 1;

    }



     png_structp png_ptr = png_create_write_struct (PNG_LIBPNG_VER_STRING, NULL, NULL,
NULL);

    png_infop info_ptr = png_create_info_struct(png_ptr);

    png_init_io(png_ptr, png_output);

    png_set_IHDR(png_ptr, info_ptr, w, h, 8,
```

```
        PNG_COLOR_TYPE_RGB , PNG_INTERLACE_NONE,

        PNG_COMPRESSION_TYPE_BASE, PNG_FILTER_TYPE_BASE);

    png_write_info(png_ptr, info_ptr);

    for(i=0; i<h; i++) {

        png_write_row(png_ptr, data + i * png_get_rowbytes (png_ptr, info_ptr));

    }



    png_write_end(png_ptr, NULL);

    png_destroy_write_struct(&png_ptr, &info_ptr);

    fclose(png_output);

}
```

C.4. Konwersja z modelu kolorów HSV do RGB

Listing C.6[1] przedstawia typową implementację metody konwersji koloru opisanego w modelu HSV do modelu RGB. Implementacja ta zakłada, że podaje się trzy wartości HSV (wartość pierwsza to liczba całkowita w zakresie <0, 359>, następne dwie to liczby rzeczywiste <0, 1>) oraz adresy, gdzie zostaną umieszczone wartości koloru RGB jako liczby w zakresie <0, 255>.

Listing C.6. Typowa implementacja konwersji koloru opisanego w modelu HSV do modelu RGB

```
void hsv2rgb(float hue, float sat, float val, int &_red, int &_green, int &_blue)

{

    float red, grn, blu;

    float i, f, p, q, t;



    if(val==0) {

        red = 0;

        grn = 0;

        blu = 0;

    } else {

        hue = hue / 60;

        i = floor(hue);

        f = hue-i;
```

```
        p = val*(1-sat);

        q = val*(1-(sat*f));

        t = val*(1-(sat*(1-f)));

        if (i==0) {red=val; grn=t; blu=p;}

            else if (i==1) {red=q; grn=val; blu=p;}

            else if (i==2) {red=p; grn=val; blu=t;}

            else if (i==3) {red=p; grn=q; blu=val;}

            else if (i==4) {red=t; grn=p; blu=val;}

            else if (i==5) {red=val; grn=p; blu=q;}

    }

    _red = min((int)(red * 255f), 255);

    _green = min((int)(grn * 255f), 255);

    _blue = min((int)(blu * 255f), 255);

}
```

C.5. Utworzenie macierzy dla filtru rozmywającego

Funkcja `makeGaussianMatrix` tworzy macierz opisującą filtr Gaussa. Cały kod źródłowy tej funkcji został przedstawiony w listingu C.7. Jej parametry mają następujące znaczenie: `base_matrix_size` oznacza podstawowy wymiar macierzy, np. podanie wartości trzy oznacza, że macierz będzie miała wymiar `3 × 3`, należy podkreślić, że wielkość macierzy powinna być nieparzysta. Jest to konieczne, aby badany piksel znajdował się w centrum maski filtru. Wartość parametru sigma podajemy w drugim argumencie, a w trzecim podajemy zmienną wskaźnikową wskazującą na odpowiednio duży obszar pamięci, w którym zostaną umieszczone poszczególne elementy macierzy filtru.

Funkcja z listingu C.7 została zbudowana z dwóch części. Pierwsza to dwie pętle `for` dla zmiennych `x` oraz `y`, których zadaniem jest wskazanie poszczególnych elementów macierzy. Jednak trzeci parametr jest pojedynczym wskaźnikiem, czyli tablicą jednowymiarową. Wykorzystując fakt, że tworzona macierz jest kwadratowa, łatwo obliczyć, jaki element wskazujemy, za pomocą `base_matrix_size`:

```
y*base_matrix_size+x
```

Pozostałe obliczenia realizowane w tych pętlach bezpośrednio wyznaczają wartości filtru. Ze względu na fakt, że macierz musi być macierzą znormalizowaną, należy obliczyć sumę wszystkich elementów, jakie zostaną wyznaczone, a następnie wszystkie elementy macierzy podzielić przez wartość sumy i tym zajmuje się druga część funkcji z pojedynczą pętlą `for`.

Wartości macierzy to liczby typu `float`, tymczasem piksele obrazu mogą być innego typu, np. wartościami od `0` do `255`. Nie można wprost zastosować macierzy utworzonej przez tę funkcję.

Jednak, jak pokazano w rozdz. 5.10.3, łatwo wymusić na OpenCL, aby piksele były konwertowane do odpowiedniego typu.

Listing C.7. Tworzenie macierzy reprezentującej filtr rozmywający (Gaussa)

```
void makeGaussianMatrix(int base_matrix_size, float sigma, float *filter_values)

{

    int x,y;

    float center = base_matrix_size / 2.0f;

    float total_sum = 0.0f;



    for(x = 0; x < base_matrix_size; x++)

    {

        for(y=0; y < base_matrix_size; y++)

        {

              filter_values[ y*base_matrix_size+x ] = exp( (((x-center)*(x-center)+(y-
center)*(y-center))/(2.0f*sigma*sigma))*-1.0f ) / (2.0f * M_PI_F *sigma*sigma);

             total_sum+=filter_values[ y*base_matrix_size+x];

        }

    }



    for(x = 0; x < base_matrix_size*base_matrix_size;x++)

    {

        filter_values[x] = filter_values[x]/total_sum;

    }

}
```

Utworzony filtr, mimo że zapisany w jednowymiarowej tablicy, daje się łatwo zaprezentować, np. jako macierz, choćby za pomocą instrukcji `printf` pod następującym kodem:

```
printf(”Macierz filtru Gaussowskiego:\n”);

for(x = 0; x < base_mtarix_size; x++) {

    for(y=0; y < base_matrix_size; y++) {

        printf(”%f ”,filter_values[ y*base_matrix_size+x]);
```

```
    }

    printf(”\n”);

}
```

W przykładzie przedstawionym w rozdz. 5.9.3 była stosowana macierz o wymiarach 5 × 5, której elementy są następujące:

```
0.000014 0.000327 0.001562 0.001562 0.000327

0.000327 0.007450 0.035544 0.035544 0.007450

0.001562 0.035544 0.169570 0.169570 0.035544

0.001562 0.035544 0.169570 0.169570 0.035544

0.000327 0.007450 0.035544 0.035544 0.007450
```

Przypisy

[1] W internecie bez trudu można znaleźć wiele przykładów podobnych do tego z listingu C.6, np. *http://www.cs.rit.edu/~ncs/color/t_convert.html*. Ogólne informacje o konwersji znajdują się np. na stronach Wikipedii: *http://en.wikipedia.org/wiki/HSL_and_HSV*.

Dodatek D. Zestawianie używanych funkcji API OpenCL

Dodatek zwiera referencyjny spis funkcji z API OpenCL, jakie występują w przykładach podawanych w tej książce. Nie jest to pełny spis funkcji OpenCL. Pełną dokumentację można znaleźć na stronach WWW organizacji Khronos. Przedstawione wyjaśnienia i komentarze odnoszą się do standardu 1.2. Każda funkcja została opatrzona krótkim opisem jej przeznaczenia, nagłówkiem, opisem parametrów, jakie przyjmuje, oraz dodatkowymi uwagami o wartościach zwracanych. Podane zostały też informacje o kodach błędów, jakie są zwracane przez poszczególne funkcje API OpenCL. Do każdej funkcji podano przykłady jej użycia.

D.1. Funkcja clGetPlatformIDs

Funkcja `clGetPlatformIDs` jest używana do otrzymania liczby dostępnych platform bądź informacji o platformach. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clGetPlatformIDs( cl_uint num_entries,

cl_platform_id *platforms,

 cl_uint *num_platforms);
```

Znaczenie poszczególnych parametrów:

• `num_entries` – liczba elementów typu `cl_platform_id`, które są wskazywane przez wskaźnik `platforms`. Jeśli parametr `platforms` nie jest wartością `NULL`, to wartość `num_entries` musi być wartością większą niż zero;

• `platforms` – lista dostępnych platform OpenCL. Elementy `cl_platform_id` zawarte w `platforms` są używane do wskazania, jaka platforma ma być wykorzystywana w programie. Jeśli argument `platforms` jest równy `NULL`, to jest on ignorowany. Liczba odczytanych platform jest określona przez wartość `num_entries`;

• `num_platforms` – całkowita liczba dostępnych platform OpenCL. W przypadku podania wartości `NULL`, argument jest ignorowany.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Jeśli wynikiem funkcji (wartością końcową) jest wartość `CL_SUCCESS`, to wykonanie funkcji zakończyło się sukcesem. W przeciwnym przypadku zwracana jest wartość `CL_INVALID_VALUE`: jeśli parametr `num_entries` jest równy zero oraz argument `platforms` jest różny od wartości `NULL`, lub jeśli dla parametrów `num_platforms` oraz `platforms` także podano wartość `NULL`. Funkcja zwraca też stałą `CL_OUT_OF_HOST_MEMORY`, jeśli po stronie gospodarza nie uda się przydzielić zasobów, jakie są niezbędne do jej działania.

Przykłady użycia

• odczyt informacji o liczbie dostępnych platform obliczeniowych:

```
cli_err_num = clGetPlatformIDs(0, NULL, &clui_num_platforms);
```

• odczyt dostępnych platform obliczeniowych:

```
cli_err_num = clGetPlatformIDs (clui_num_platforms, opencl_platforms, NULL);
```

D.2. Funkcja clGetPlatformInfo

Funkcja pobiera informacje o wybranej platformie OpenCL. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clGetPlatformInfo( cl_platform_id platform,

cl_platform_info param_name,

size_t param_value_size,

void *param_value,

size_t *param_value_size_ret);
```

Znaczenie poszczególnych parametrów:

• `platform` – identyfikator platformy otrzymany z funkcji `clGetPlatformIDs`. Możliwe jest podanie wartości `NULL`, lecz w takim przypadku zachowanie się funkcji zależy od konkretnej implementacji OpenCL dla danej platformy sprzętowo-programowej;

• `param_name` – identyfikator informacji, która ma zostać odczytana, w tab. D.1 przedstawiono zestaw dopuszczalnych argumentów;

• `param_value_size` – parametr pozwala określić wielkość pamięci wskazywanej przez `param_value`. Należy zadbać, aby przydzielona pamięć miała odpowiednią wielkość, zgodnie z parametrami określonymi w tab. D.1;

• `param_value` – wskaźnik do obszaru pamięci gospodarza, gdzie zostaną umieszczone odczytane informacje. Można podać wartość `NULL`, co powoduje, że funkcja zignoruje prośbę o odczyt określonych informacji;

• `param_value_size_ret` – argument pozwala odczytać aktualną ilość (w bajtach) danych, które zostaną przesłane do argumentu `param_value`. W przypadku podania wartości `NULL`, argument jest ignorowany.

Tabela D.1. Możliwe wartości parametru `param_name` dla funkcji `clGetPlatformInfo`

Stała cl_platform_info	Typ	Opis
		Ciąg znaków reprezentujący tzw. profil OpenCL. Obecnie stosowane są dwa profile: profil pełny

CL_PLATFORM_PROFILE	char[]	FULL_PROFILE, gdy mamy do czynienia z pełną implementacją funkcjonalności OpenCL opisanej w standardzie, a także profil osadzony EMBEDDED_PROFILE dla urządzeń mobilnych, zazwyczaj jest to wybrany podzbiór możliwości pełnej wersji OpenCL.
CL_PLATFORM_VERSION	char[]	Ciąg znaków reprezentujący numer wersji OpenCL mający następujący format: OpenCL <znak spacji> <główny numer>.<pomocniczy numer> <informacje specyficzne dla platformy> Numery wersji <główny numer>.<pomocniczy numer> obecnie są zapisane jako: 1.0, 1.1, a także 1.2.
CL_PLATFORM_NAME	char[]	Ciąg znaków reprezentujący nazwę platformy.
CL_PLATFORM_VENDOR	char[]	Ciąg znaków reprezentujący dostawcę platformy.
CL_PLATFORM_EXTENSIONS	char[]	Rezultatem jest ciąg znaków, w którym umieszczone są nazwy dostępnych rozszerzeń dla określonej platformy. Nazwy są rozdzielone pojedynczą spacją (nazwy rozszerzeń nie zawierają spacji).

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Jeśli zawracaną wartością dla zapytania jest typ `char[]`, to otrzymuje się ciąg znaków zakończony wartością `NULL`. Jeśli wynikiem funkcji (wartością końcową) `clGetPlatformInfo` jest `CL_SUCCESS`, to wykonanie funkcji zakończyło się sukcesem. W każdym innym przypadku funkcja zwraca jedną z następujących wartości:

• `CL_INVALID_PLATFORM` – wskazana platforma nie jest poprawna;

• `CL_INVALID_VALUE` – `param_name` nie otrzymał predefiniowanej stałej (podanej w tab. D.1), lub podana ilość pamięci nie jest wystarczająca, aby zapisać poprawnie zwracane informacje, co oznacza, że `param_value` w tym przypadku nie ma wartości `NULL`.

• `CL_OUT_OF_HOST_MEMORY` – nie udało przydzielić potrzebnych zasobów do poprawnej pracy funkcji po stronie gospodarza.

Przykłady użycia:

• odczytanie nazwy platformy obliczeniowej:

```
cli_err_num = clGetPlatformInfo( opencl_platforms[ocl_plat_idx], CL_PLATFORM_NAME, 4096, buffer1, NULL);
```

D.3. Funkcja clGetDeviceInfo

Funkcja `clGetDeviceInfo` umożliwia otrzymanie informacji dotyczących urządzeń obliczeniowych OpenCL. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clGetDeviceInfo( cl_device_id device,

cl_device_info param_name,

 size_t param_value_size,

 void *param_value,

 size_t *param_value_size_ret);
```

Znaczenie poszczególnych parametrów:

• `device` – obiekt reprezentujący urządzenie, uzyskany za pomocą funkcji `clGetDeviceIDs`;

• `param_name` – stała identyfikująca informacje, które ma zwrócić funkcja `clGetDeviceInfo` (spis stałych podano w tab. D.2);

• `param_value_size` – ilość przydzielonej pamięci (w bajtach) do parametru `param_value`. Należy zwrócić baczną uwagę na to, aby ilość przydzielonej pamięci przekraczała rzeczywisty rozmiar zwracanej informacji lub przynajmniej była jej równa;

• `param_value` – wskaźnik do obszaru pamięci, w którym zostaną umieszczone odczytane informacje. W przypadku podania wartości `NULL` argument zostanie zignorowany;

• `param_value_size_ret` – zwraca aktualny rozmiar informacji umieszczonej w parametrze `param_value`. Podanie wartości `NULL` oznacza zignorowanie tego parametru.

Tabela D.2. Stałe dla funkcji `clGetDeviceIDs`

Stała `cl_device_info`	Typ	Opis
`CL_DEVICE_ADDRESS_BITS`	`cl_uint`	Wartość domyślna odnosząca się do wielkości przestrzeni adresowej podawana w bitach. Obecnie są to wartości 32 lub 64.
`CL_DEVICE_AVAILABLE`	`cl_bool`	Otrzymanie wartości `CL_TRUE` oznacza, iż określone urządzenie jest dostępne, otrzymanie `CL_FALSE` oznacza, że urządzenie nie jest dostępne do dalszych działań.
		Otrzymanie wartości `CL_FALSE` oznacza, że sterownik

CL_DEVICE_COMPILER_AVAILABLE	cl_bool	OpenCL dla danej implementacji nie oferuje wbudowanego kompilatora dla źródeł jądra obliczeniowego. Taka sytuacja dość często pojawia się w przypadku platform mobilnych, gdzie sterownik nie ma wbudowanego kompilatora i źródła jądra obliczeniowego muszą być kompilowane zewnętrznym kompilatorem OpenCL C.
CL_DEVICE_DOUBLE_FP_CONFIG	cl_device_fp_config	Opis wsparcia dla liczb o podwójnej precyzji, zwrócone zostanie pole, gdzie poszczególne bity pozwalają odczytać, jakie możliwości obliczeniowe są wspierane (obsługiwane) przez urządzenie obliczeniowe: • CL_FP_DENORM – liczby denormalne są wspierane, • CL_FP_INF_NAN – oznaczenia INF oraz NaN są wspierane, • CL_FP_ROUND_TO_NEAREST – zaokrąglenie w kierunku liczby parzystej jest wspierane, • CL_FP_ROUND_TO_ZERO – zaokrąglenie w kierunku zera jest wspierane, • CL_FP_ROUND_TO_INF – zaokrąglanie w kierunku „+ nieskończoność” oraz „– nieskończoność” są wspierane, • CP_FP_FMA – operacja FMA (tzw. rozmyte mnożenie i dodawanie) w standardzie IEEE754-2008 jest wspierane, • CL_FP_SOFT_FLOAT – podstawowe operacje, jak np. dodawanie i mnożenie są zaimplementowane w sposób programowy. Należy podkreślić, iż wsparcie dla liczb podwójnej precyzji jest opcjonalne, przy czym, jeśli urządzenie obliczeniowe obsługuje tego typu liczby, wymaga się, aby dostępne były funkcje opisane przez następujące flagi: CL_FP_FMA, CL_FP_ROUND_TO_NEAREST, CL_FP_ROUND_TO_ZERO, CL_FP_ROUND_TO_INF, CL_FP_INF_NAN, CL_FP_DENORM.
CL_DEVICE_ENDIAN_LITTLE	cl_bool	Jeśli zwrócona zostanie wartość CL_TRUE, to urządzenie OpenCL jest urządzeniem o organizacji bajtów typu „little endian”.
CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool	Otrzymanie wartości CL_TRUE oznacza, że urządzenie obliczeniowe oferuje korekcję błędów dla układów pamięci, pamięci cache oraz rejestrów. Nie jest to wymagana funkcjonalność.

CL_DEVICE_EXECUTION_CAPABILITIES	cl_device_exec_capabilities	Opis zdolności wykonawczych urządzenia obliczeniowego. Dostępne są dodatkowe flagi: • CL_EXEC_KERNEL – urządzenie OpenCL może wykonywać dowolne jądra obliczeniowe, • CL_EXEC_NATIVE_KERNEL – urządzenie OpenCL może wykonywać tylko tzw. jądra natywne. Podstawowym warunkiem jest obsługa wykonywania dowolnych jąder obliczeniowych, co opisuje flaga – CL_EXEC_KERNEL.
CL_DEVICE_EXTENSIONS	char[]	Wynikiem jest lista nazw rozszerzeń w postaci ciągu znaków, w którym poszczególne rozszerzenia są rozdzielone spacją. Nie ma ustalonej listy nazwy rozszerzeń, choć organizacja standaryzująca aprobuje następujące nazwy rozszerzeń, jakie mogą być obsługiwane w sterowniku OpenCL: • cl_khr_int64_base_atomics, • cl_khr_int64_extended_atomics, • cl_khr_fp16, • cl_khr_gl_sharing, • cl_khr_gl_event, • cl_khr_d3d10_sharing, • cl_khr_dx9_media_sharing, • cl_khr_d3d11_sharing. W przypadku standardu OpenCL wersja 1.2, organizacja tworząca standard wymaga, aby obsługiwane były następujące rozszerzenia: • cl_khr_global_int32_base_atomics, • cl_khr_global_int32_extended_atomics, • cl_khr_local_int32_base_atomics, • cl_khr_local_int32_extended_atomics, • cl_khr_byte_addressable_store, • cl_khr_fp64.

CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	cl_ulong	Wielkość pamięci cache dla pamięci globalnej w bajtach.
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	cl_device_mem_cache_type	Typ pamięci cache dla pamięci globalnej; obsługiwane są następujące stałe: CL_NONE, CL_READ_ONLY_CACHE, CL_READ_WRITE_CACHE.
CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE	cl_uint	Wielkość pojedynczej linii cache dla pamięci globalnej.
CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong	Wielkość pamięci globalnej w bajtach.
CL_DEVICE_HALF_FP_CONFIG	cl_device_fp_config	Opis wsparcia dla liczb o precyzji połówkowej / ograniczonej (typ half), zwrócone zostanie pole, gdzie poszczególne bity pozwalają odczytać, jakie możliwości obliczeniowe są wspierane (obsługiwane) przez urządzenie obliczeniowe: • CL_FP_DENORM – liczby denormalne są wspierane, • CL_FP_INF_NAN – oznaczenia INF oraz NaN są wspierane, • CL_FP_ROUND_TO_NEAREST – zaokrąglenie w kierunku liczby parzystej jest wspierane, • CL_FP_ROUND_TO_ZERO – zaokrąglenie w kierunku zera jest wspierane, • CL_FP_ROUND_TO_INF – zaokrąglanie w kierunku „+nieskończoność” oraz „–nieskończoność” są wspierane, • CP_FP_FMA – operacja FMA (tzw. rozmyte mnożenie i dodawanie) w standardzie IEEE754-2008 jest wspierane. Należy podkreślić, iż wsparcie dla liczb half jest opcjonalne, przy czym, jeśli urządzenie obliczeniowe obsługuje tego typu liczby, wymaga się, aby dostępne były funkcje opisane przez następujące flagi: CL_FP_ROUND_TO_ZERO, CL_FP_ROUND_TO_INF, CL_FP_INF_NAN.
CL_DEVICE_HOST_UNIFIED_MEMORY	cl_bool	Jeśli zwrócona została wartość CL_TRUE, to urządzenie obliczeniowe OpenCL oraz system gospodarza mają zunifikowany system obsługi pamięci.

CL_DEVICE_IMAGE_SUPPORT	cl_bool	Jeśli została zwrócona wartość CL_TRUE, to urządzenie OpenCL wspiera obsługę obrazów.
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t	Maksymalna wysokość obrazu 2D w pikselach. Wartość minimalna wynosi 8192. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t	Maksymalna szerokość obrazu 2D w pikselach. Wartość minimalna wynosi 8192. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.
CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t	Maksymalna głębokość obrazu 3D w pikselach. Wartość minimalna wynosi 2048. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t	Maksymalna wysokość obrazu 3D w pikselach. Wartość minimalna wynosi 2048. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t	Maksymalna szerokość obrazu 3D w pikselach. Wartość minimalna wynosi 2048. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.
CL_DEVICE_IMAGE_MAX_BUFFER_SIZE	size_t	Maksymalna liczba pikseli dla obrazu 1D. Wartość minimalna wynosi 65536. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.
CL_DEVICE_IMAGE_MAX_ARRAY_SIZE	size_t	Maksymalna liczba obrazów w tablicy obrazów 1D lub 2D. Wartość minimalna wynosi 2048. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.
CL_DEVICE_LINKER_AVAILABLE	cl_bool	Zwrócenie wartości CL_FALSE oznacza, że dany sterownik OpenCL nie oferuje programu konsolidującego (linker). Taka sytuacja może się pojawić w przypadku platform mobilnych. Zwrócenie wartości CL_TRUE oznacza też konieczność obecności kompilatora. Inaczej mówiąc, dla flagi

		CL_DEVICE_COMPILER_AVAILABLE również należy uzyskać wartość CL_TRUE.
CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong	Wielkość pamięci lokalnej w bajtach. Wartość minimalna wynosi 32 kb.
CL_DEVICE_LOCAL_MEM_TYPE	cl_device_local_mem_type	Rodzaj pamięci lokalnej. Flaga CL_LOCAL oznacza, że pamięć lokalna została zaimplementowana np. za pomocą dedykowanej pamięci typu SRAM. Wartość CL_GLOBAL oznacza, że użyto pamięci globalnej do utworzenia obszaru dla pamięci lokalnej.
CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint	Maksymalna częstotliwość pracy urządzenia obliczeniowego w Mhz.
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint	Liczba jednostek obliczeniowych w urządzeniu OpenCL. Minimalna liczba jednostek to 1 jednostka.
CL_DEVICE_MAX_CONSTANT_ARGS	cl_uint	Maksymalna liczba argumentów, jaka może zostać zadeklarowana z kwalifikatorem __constant w jądrze obliczeniowym. Minimalna wielkość to 8.
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	cl_ulong	Maksymalna wielkość w bajtach bufora stałego, jaka może zostać przydzielona. Minimalna wielkość to 64 kb.
CL_DEVICE_MAX_MEM_ALLOC_SIZE	cl_ulong	Maksymalny rozmiar alokowanego obiektu w bajtach. Minimalna wartość to ¼ rozmiaru pamięci globalnej.
CL_DEVICE_MAX_PARAMETER_SIZE	size_t	Maksymalny rozmiar w bajtach argumentów, jakie mogą zostać przekazane do jądra obliczeniowego. Minimalna wartość tego parametru to 1024, co przekłada się na 128 argumentów.
CL_DEVICE_MAX_READ_IMAGE_ARGS	cl_uint	Maksymalna liczba obiektów reprezentujących obrazy, jaka może być odczytywana przez jądro obliczeniowe. Obecnie minimalna liczba to 128. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.

CL_DEVICE_MAX_SAMPLERS	cl_uint	Maksymalna liczba obiektów próbkujących, jakiej można użyć w pojedynczym jądrze obliczeniowym. Minimalna liczba to 16. Wymagane jest wsparcie dla przetwarzania obrazów przez urządzenie obliczeniowe.
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t	Maksymalna liczba jednostek roboczych w grupie roboczej dla jądra obliczeniowego w pojedynczej jednostce obliczeniowej (inaczej wielkość grupy roboczej).
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	cl_uint	Maksymalna wymiarowość, jaką można zastosować dla globalnej oraz lokalnej siatki obliczeniowej. Minimalna wartość wynosi 3.
CL_DEVICE_MAX_WORK_ITEM_SIZES	size_t[]	Maksymalna liczba jednostek roboczych, jaka może zostać podana do grupy roboczej w każdym dostępnym wymiarze. Informacje są zwracane w tablicy typu size_t, a liczbę elementów można odczytać, korzystając ze stałej CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, zwracającej liczbę wymiarów, jaka jest dostępna w opisie siatki obliczeniowej.
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint	Maksymalna liczba obrazów, jaką może zapisać jądro obliczeniowe, minimalna liczba obrazów wynosi 8.
CL_DEVICE_MEM_BASE_ADDR_ALIGN	cl_uint	Minimalna wielkość w bitach największego typu wbudowanego OpenCL, obsługiwanego przez podane urządzenie obliczeniowe. Zazwyczaj jest to typ long16, choć w przypadku urządzeń mobilnych może to być również typ int16.
CL_DEVICE_NAME	char[]	Nazwa urządzenia obliczeniowego zapisana jako ciąg znaków.
CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT CL_DEVICE_NATIVE_VECTOR_WIDTH_INT		Stałe pozwalają odczytać natywną długość typów wektorowych, jakie są dostępne w urządzeniu obliczeniowym. Przez długość rozumie się liczbę elementów, które można zapisać do danego typu.

CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF	cl_uint	W przypadku braku obsługi liczb podwójnej precyzji, czy też typu `half`, stałe, `CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE` oraz `CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF` zwracają wartość zero.
CL_DEVICE_OPENCL_C_VERSION	char[]	Stała zwraca numer wersji języka OpenCL C w postaci ciągu znaków. Zawsze zwracany jest numer najwyższej wersji języka, jaka jest obsługiwana przez kompilator (kompilator może również obsługiwać starsze odmiany OpenCL C). Format zapisu numeru wersji ma następującą postać: `OpenCL<spacja>C<spacja> <numer_główny.numer_pomocniczy><spacja> <informacje_od_sprzedawcy>` Numer wersji `numer_główny.numer_pomocniczy` zawsze zwraca numer wersji języka OpenCL C, np. jeśli język ma numer 1.1, to postać numeru wersji to OpenCL C 1.1.
CL_DEVICE_PARENT_DEVICE	cl_device_id	Zwracane jest urządzenie nadrzędne do podanego urządzenia. Jeśli podano urządzenie główne, zwracana jest wartość `NULL`.
CL_DEVICE_PARTITION_MAX_SUB_DEVICES	cl_uint	Maksymalna liczba pod-urządzeń, jakie można wydzielić z urządzenia obliczeniowego. Wartość, wg standardu OpenCL, nie może przekroczyć wartości otrzymywanej za pomocą stałej `CL_DEVICE_MAX_COMPUTE_UNITS`.
CL_DEVICE_PLATFORM	cl_platform_id	Platforma OpenCL, do której przynależy podane urządzenie obliczeniowe.
CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF	cl_uint	Stałe pozwalają odczytać preferowaną natywną długość typów wektorowych, jakie są dostępne w urządzeniu obliczeniowym. Przez długość rozumie się liczbę elementów, które można zapisać do danego typu. W przypadku braku obsługi liczb podwójnej precyzji, czy też typu `half`, stałe: `CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE oraz` `CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF` zwracają wartość zero.

CL_DEVICE_PRINTF_BUFFER_SIZE	size_t	Maksymalna wielkość wewnętrznego bufora dla instrukcji printf przechowująca przekazywane komunikaty. Minimalna wielkość to 1 MB.
CL_DEVICE_PREFERRED_INTEROP_USER_SYNC	cl_bool	Otrzymanie wartości CL_TRUE oznacza, iż urządzenie obliczeniowe wymaga, aby synchronizacja obiektów pamięciowych pomiędzy OpenCL a np. OpenGL była wykonywana przez użytkownika. Otrzymanie wartość CL_FALSE oznacza, że to urządzenie będzie odpowiedzialne za synchronizację obiektów.
CL_DEVICE_PROFILE	char[]	Stała pozwala odczytać tzw. profil urządzenia w postaci ciągu znaków. Obecnie dostępne są dwa profile w ramach standardu OpenCL: • ciąg „FULL_PROFILE” – urządzenie obsługuje pełną wersję standardu OpenCL. Dostępne są wszystkie funkcje OpenCL, które w specyfikacji zostały określone jako wymagane, mogą być też dostępne inne funkcje poprzez mechanizm rozszerzeń uruchamianych za pomocą polecenia #pragma, • ciąg „ EMBEDDED_PROFILE” – dane urządzenie obsługuje profil osadzony, który obecnie jest stosowany w przypadku urządzeń mobilnych.
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t	Stała zwraca opis rozdzielczości licznika / timera dostępnego w urządzeniu obliczeniowym. Wielkość jest zwracana w nanosekundach.
CL_DEVICE_QUEUE_PROPERTIES	cl_command_queue_properties	Stała opisuje własności kolejki poleceń wspieranej przez urządzenie. Zwrócona wartość to pole bitowe, które można odczytać za pomocą poniższych flag: • CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE • CL_QUEUE_PROFILING_ENABLE Wymaga się, aby kolejka posiadała własność: CL_QUEUE_PROFILING_ENABLE.
CL_DEVICE_SINGLE_FP_CONFIG	cl_device_fp_config	Opis wsparcia dla liczb o pojedynczej precyzji, zwrócone zostanie pole, którego poszczególne bity

pozwalają odczytać, jakie możliwości obliczeniowe są

		wspierane (obsługiwane) przez urządzenie obliczeniowe: • `CL_FP_DENORM` – liczby denormalne są wspierane, • `CL_FP_INF_NAN` – oznaczenia INF oraz NaN są wspierane, • `CL_FP_ROUND_TO_NEAREST` – zaokrąglenie w kierunku liczby parzystej jest wspierane, • `CL_FP_ROUND_TO_ZERO` – zaokrąglenie w kierunku zera jest wspierane, • `CL_FP_ROUND_TO_INF` – zaokrąglanie w kierunku „+ nieskończoność” oraz „– nieskończoność” jest wspierane, • `CP_FP_FMA` – operacja FMA (tzw. rozmyte mnożenie i dodawanie) w standardzie IEEE754-2008 jest wspierane, • `CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT` – operacja dzielenia oraz funkcja pierwiastka kwadratowego zwracają poprawnie zaokrąglony rezultat zgodny ze specyfikacją IEEE-754, • `CL_FP_SOFT_FLOAT` – podstawowe operacje, jak np. dodawanie, mnożenie są zaimplementowane w sposób programowy. Wymaga się, aby dostępne były funkcje opisane przez następujące flagi: `CL_FP_ROUND_TO_NEAREST`, `CL_FP_INF_NAN`.
`CL_DEVICE_TYPE`	`cl_device_type`	Stała pozwala odczytać typ urządzenia obliczeniowego. Dostępne są następujące typy: `CL_DEVICE_TYPE_CPU`, `CL_DEVICE_TYPE_GPU`, `CL_DEVICE_TYPE_ACCELERATOR`, lub `CL_DEVICE_TYPE_DEFAULT`. Dopuszcza się kombinację powyższych typów, a także wartość `CL_DEVICE_TYPE_CUSTOM`.
`CL_DEVICE_VENDOR`	`char[]`	Ciąg znaków reprezentujący dostawcę.
`CL_DEVICE_VENDOR_ID`	`cl_uint`	Unikalny identyfikator urządzenia obliczeniowego dostarczony przez producenta.
		Stała zwraca numer obsługiwanego standardu OpenCL w postaci ciągu znaków. Format zapisu numeru wersji ma następującą postać:

CL_DEVICE_VERSION	char[]	OpenCL<spacja><numer_główny.numer_pomocniczy><spacja><informacje_od_sprzedawcy> Numer wersji numer_główny.numer_pomocniczy zawsze zwraca numer standardu OpenCL. Jeśli obsługiwany standard ma numer 1.1, to postać numeru wersji to OpenCL 1.1.
CL_DRIVER_VERSION	char[]	Ciąg znaków reprezentujący wersję sterownika OpenCL. Postać numeru jest następująca: numer_główny.numer_pomocniczy, czyli numer główny poprzedza numer pomocniczy.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Jeśli wynikiem funkcji (wartością końcową) jest wartość CL_SUCCESS, to wykonanie funkcji zakończyło się sukcesem. W przeciwnym przypadku zwracany jest jeden z poniższych kodów:

• CL_INVALID_DEVICE – podano niepoprawny obiekt reprezentujący urządzenie obliczeniowe,

• CL_INVALID_VALUE – param_name ma inną wartość niż wymieniona w tab. D.2 (lista podanych wartości nie jest pełna, dokładne informacje można znaleźć w specyfikacji), lub liczba bajtów podana w param_value_size nie jest wystarczająca, aby poprawnie odczytać wskazane informacje, lub wartość param_name jest stałą opisującą własność, która jest dostępna jako rozszerzenie, a określone urządzenie nie obsługuje tej własności,

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL,

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• odczytanie nazwy urządzenia obliczeniowego:

```
cli_err_num = clGetDeviceInfo(platform_devices[ocl_dev_idx], CL_DEVICE_NAME, 4096, buffer1, NULL);
```

• odczytanie liczby jednostek przetwarzających:

```
cli_err_num = clGetDeviceInfo(platform_devices[ocl_dev_idx], CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &tmp, NULL);
```

• odczyt informacji o regułach przetwarzania liczb zmiennoprzecinkowych w pojedynczej precyzji:

```
cli_err_num = clGetDeviceInfo(platform_devices[ocl_dev_idx], CL_DEVICE_SINGLE_FP_CONFIG, sizeof(dfpc_tmp), &dfpc_tmp, NULL);
```

D.4. Funkcja clGetSupportedImageFormats

Funkcja `clGetSupportedImageFormats` jest używana do uzyskania informacji o tym, jakie formaty obrazów są wspierane w ramach dostępnych urządzeń obliczeniowych OpenCL. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clGetSupportedImageFormats ( cl_context context,

cl_mem_flags flags,

    cl_mem_object_type image_type,

    cl_uint num_entries,

    cl_image_format *image_formats,

    cl_uint *num_image_formats);
```

Znaczenie poszczególnych parametrów:

• `context` – kontekst obliczeniowy OpenCL, dla którego będą tworzone obiekty reprezentujące obrazy;

• `flags` – pole bitowe, analogiczne jak dla funkcji `clCreateBuffer`, do opisu dodatkowych informacji o sposobie traktowania obiektu reprezentującego obraz;

• `image_type` – opis formatu obrazu, parametr przyjmuje następujące stałe: `CL_MEM_OBJECT_IMAGE1D`, `CL_MEM_OBJECT_IMAGE1D_BUFFER`, `CL_MEM_OBJECT_IMAGE2D`, `CL_MEM_OBJECT_IMAGE3D`, `CL_MEM_OBJECT_IMAGE1D_ARRAY`, `CL_MEM_OBJECT_IMAGE2D_ARRAY`, oznaczające typ obrazu, dla jakiego chcemy otrzymać listę formatów (może to być pojedynczy obraz, ale także tablica obrazów);

• `num_entries` – liczba elementów w tablicy opisu formatów, jaka zostanie zwrócona i umieszczona w parametrze `image_formats`;

• `image_formats` – wskaźnik do pamięci, w której zostaną umieszczone elementy opisujące dostępne formaty obrazu, zwracane są tylko te formaty, które są obsługiwane przez używaną implementację OpenCL, podanie wartości `NULL` powoduje zignorowanie argumentu;

• `num_image_formats` – liczba obsługiwanych formatów dla podanego typu obrazu w parametrze `image_type`, podanie wartości `NULL` w tym parametrze oznacza jego zignorowanie.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Minimalna liczba formatów, jaka jest wymagana do obsługi przez standard OpenCL (jeśli zadeklarowano wsparcie obsługi obrazów) została podana w tab. D.3.

Tabela D.3. Podstawowy zestaw formatów obrazu, jaki jest obsługiwany przez OpenCL

Liczba kanałów dla kolorów	Sposób ułożenia kanałów	Typ opisu pojedynczego kanału
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
4	CL_BGRA	CL_UNORM_INT8

Jeśli wynikiem wykonania funkcji (wartością końcową) jest wartość `CL_SUCCESS`, to zakończyło się ono sukcesem. W przeciwnym przypadku zwracany jest jeden z poniższych kodów:

• `CL_INVALID_CONTEXT` – podany kontekst obliczeniowy nie jest poprawny;

• `CL_INVALID_VALUE` – podano błędną wartość w parametrze `flags` lub `image_type`, lub jeśli w parametrze `num_entries` podano zero, ale wartość `image_formats` jest różna od `NULL`;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• odczyt informacji o liczbie obsługiwanych formatów dla obrazów 2D:

```
clGetSupportedImageFormats(ctx, CL_MEM_READ_ONLY, CL_MEM_OBJECT_IMAGE2D, NULL, NULL, &frmtCount);
```

• odczyt informacji o formatach obrazów 2D obsługiwanych przez dane urządzenie obliczeniowe:

```
clGetSupportedImageFormats(ctx, CL_MEM_READ_ONLY, CL_MEM_OBJECT_IMAGE2D, tmp, img_formats, NULL);
```

D.5. Funkcja clGetDeviceIDs

Funkcja `clGetDeviceIDs` jest używana do otrzymania listy dostępnych urządzeń obliczeniowych w ramach wybranej platformy OpenCL. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clGetDeviceIDs( cl_platform_id platform,

cl_device_type device_type,

    cl_uint num_entries,

    cl_device_id *devices,

 cl_uint *num_devices);
```

Znaczenie poszczególnych parametrów:

• `platform` – obiekt reprezentujący platformę zwrócony przez funkcję `clGetPlatformIDs`, może mieć wartość `NULL`, jednak zachowanie się funkcji w tym przypadku zależy od implementacji konkretnego sterownika OpenCL;

• `device_type` – poszczególne bity tego pola pozwalają na wyszczególnienie, jaki typ urządzenia OpenCL jest poszukiwany. Dopuszczalne wartości dla tego argumentu zostały podane w tab. D.4;

• `num_entries` – liczba elementów, która zostanie wpisana do parametru `devices`, jeśli parametr `devices` nie jest równy `NULL`, wartość tego parametru musi być większa niż zero;

• `devices` – lista odczytanych urządzeń obliczeniowych, wartość parametru może być równa `NULL`, w takim przypadku jest on ignorowany, liczba odczytanych urządzeń jest ograniczona przez wartość `num_entries`;

• `num_devices` – liczba urządzeń OpenCL zgodnych z wartością parametru `device_type`, jeśli podano wartość `NULL` argument jest ignorowany.

Tabela D.4. Możliwe wartości typu `cl_device_type`

Stałe typu cl_device_type	Opis
`CL_DEVICE_TYPE_CPU`	Urządzenie OpenCL jest reprezentowane przez tradycyjny procesor (jedno- bądź wielordzeniowy).
`CL_DEVICE_TYPE_GPU`	Urządzenie OpenCL jest reprezentowane przez procesor typu GPU, inaczej mówiąc jest to karta graficzna.

CL_DEVICE_TYPE_ACCELERATOR	Stała oznacza, iż urządzenie OpenCL jest wyspecjalizowanym urządzeniem obliczeniowym.
CL_DEVICE_TYPE_CUSTOM	Specjalne urządzenie obliczeniowe, które nie obsługuje programów napisanych w języku OpenCL C.
CL_DEVICE_TYPE_DEFAULT	Domyślne urządzenie OpenCL dostępne w systemie użytkownika. Domyślnym urządzeniem nie mogą być urządzenia typu CL_DEVICE_TYPE_CUSTOM.
CL_DEVICE_TYPE_ALL	Stała określa wszystkie urządzenia OpenCL dostępne w systemie z pominięciem urządzeń typu CL_DEVICE_TYPE_CUSTOM.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Funkcja clGetDeviceIDs zwraca wartość CL_SUCCESS, jeśli operacja pobrania urządzeń zakończyła się poprawnie. W przeciwnym przypadku, można otrzymać jeden z następujących kodów:

• CL_INVALID_PLATFORM – podany obiekt reprezentujący platformę nie jest poprawnym obiektem;

• CL_INVALID_DEVICE_TYPE – typ podany w parametrze device_type nie jest poprawną wartością;

• CL_INVALID_VALUE – parametr num_devices jest równy zero, ale parametr devices jest różny od wartości NULL, bądź parametr num_devices oraz parametr devices przyjmują wartość NULL;

• CL_DEVICE_NOT_FOUND – nie ma urządzeń OpenCL zgodnych z typem podanym w parametrze device_type;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• liczba urządzeń obliczeniowych dostępnych w ramach platformy:

```
cli_err_num = clGetDeviceIDs ( opencl_platforms[ocl_plat_idx], CL_DEVICE_TYPE_ALL, 0, NULL, &clui_num_devices);
```

• odczyt obiektów reprezentujących urządzenia obliczeniowe:

```
cli_err_num = clGetDeviceIDs ( opencl_platforms[ocl_plat_idx], CL_DEVICE_TYPE_ALL, clui_num_devices, platform_devices, &clui_num_devices);
```

D.6. Funkcja clCreateContext

Funkcja `clCreateContext` jest stosowana do utworzenia kontekstu obliczeniowego dla podanych urządzeń obliczeniowych OpenCL. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_context clCreateContext( cl_context_properties *properties,

cl_uint num_devices,

    const cl_device_id *devices,

    void *pfn_notify (

        const char *errinfo,

        const void *private_info,

        size_t cb,

        void *user_data),

    void *user_data,

 cl_int *errcode_ret);
```

Znaczenie poszczególnych parametrów:

• `properties` – pozwala podać ciąg własności w postaci tablicy zakończonej wartością zero. Tworzony kontekst uwzględni własności podane w tym parametrze, np. jeśli chcemy, aby procedury obliczeniowe OpenCL współpracowały ze standardem OpenGL. Każda z własności może nakładać na urządzenie obliczeniowe dodatkowe wymagania, np. aby były dostępne specyficzne formaty obrazów;

• `num_devices` – liczba urządzeń podanych w argumencie `devices`;

• `devices` – wskaźnik do danych zwróconych przez funkcję `clGetDeviceIDs`, czyli listy dostępnych urządzeń obliczeniowych;

• `pfn_notify` – adres do funkcji zwrotnej, która zostanie wywołania przez sterownik OpenCL w momencie wystąpienia błędu. Wywołanie tej funkcji może być całkowicie asynchroniczne, więc należy zapewnić, aby była to funkcja przystosowana do pracy w oddzielnym wątku. Parametry funkcji są następujące;

– `errinfo` – wskaźnik do ciągu znaków opisujących błąd,

– `private_info` – wskaźnik do danych binarnych zwracanych przez określoną implementację sterownika OpenCL, gdzie wystąpił błąd,

– `cb` – wielkość danych przekazywanych jako `private_info`,

– `user_data` – wykorzystywany do przekazania danych użytkownika przekazanych w argumencie `user_data` funkcji `clCreateContext`. W przypadku podania wartości `NULL` funkcja zwrotna nie zostanie wywołana.

• `user_data` – dodatkowe dane użytkownika, które zostaną przekazane do argumentu `user_data` podczas wywołania funkcji `pfn_notify`. Wartością tego pola może być też `NULL`,

• `errcode_ret` – zwraca wskaźnik zmiennej zawierającej ewentualny kod błędu, podanie wartość `NULL` oznacza, że kod nie zostanie zwrócony.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Należy podkreślić fakt, iż kontekst obliczeniowy jest tworzony nie tylko dla jednego urządzenia, ale dla wszystkich dostępnych urządzeń obliczeniowych, jakie oferuje dana platforma. Oprócz funkcji `clCreateContext` dostępna jest też funkcja `clCreateContextFromType`, (z dodatkowym parametrem określającym typ urządzenia), pozwalająca ściślej określić typ urządzenia, dla którego ma zostać utworzony kontekst obliczeniowy.

Funkcja zwraca obiekt reprezentujący kontekst obliczeniowy oraz wartość `CL_SUCCESS` umieszczoną w zmiennej wskazanej przez `errcode_ret`, jeśli operacja zakończyła się poprawnie. W przeciwnym przypadku zwrócona zostanie wartość `NULL`, natomiast w parametrze `errcode_ret` jest umieszczany jeden z następujących kodów oznaczających błąd:

• `CL_INVALID_PLATFORM` – argument `properties` otrzymał wartość `NULL` oraz nie wybrano platformy lub wskazana platforma nie jest poprawna;

• `CL_INVALID_VALUE` – podano błędną nazwę własności lub argument `devices` jest równy `NULL`, lub liczba urządzeń (`num_devies`) jest równa zero, lub parametr `pnf_notify` jest równy `NULL`, ale wartość `user_data` jest różna od `NULL`;

• `CL_INVALID_DEVICE` – istnieje urządzenie, które nie jest poprawnym urządzeniem lub nie jest urządzeniem podanej platformy;

• `CL_DEVICE_NOT_AVAILABLE` – jedno z urządzeń obliczeniowych nie jest aktualnie dostępne;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Podane tu informacje nie uwzględniają kodów charakterystycznych dla określonych implementacji, np. jeśli oczekujemy, że kontekst obliczeniowy OpenCL ma współpracować z interfejsem DirectX, protokołem X, czy też ze standardem OpenGL. Dokładniejsze informacje można znaleźć w oryginalnych materiałach standardu OpenCL.

Przykłady użycia:

• utworzenie kontekstu obliczeniowego:

```
cl_compute_context = clCreateContext( NULL, clui_num_devices, &platform_devices[0], NULL, NULL, &cli_err_num);
```

D.7. Funkcja clBuildProgram

Funkcja `clBuildProgram` jest odpowiedzialna za zbudowanie programu obliczeniowego z podanego wcześniej kodu źródłowego lub postaci binarnej. Przeprowadzana jest kompilacja oraz konsolidacja programu dla dostępnych urządzeń obliczeniowych. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clBuildProgram ( cl_program program,

cl_uint num_devices,

    const cl_device_id *device_list,

    const char *options,

    void (*pfn_notify)(cl_program, void *user_data),

 void *user_data);
```

Znaczenie poszczególnych parametrów:

• `program` – parametr reprezentuje obiekt programu;

• `num_devices` – liczba urządzeń podanych w parametrze `device_list`;

• `device_list` – lista urządzeń, dla których ma zostać zbudowany program obliczeniowy. Podanie wartości `NULL` oznacza, iż program zostanie zbudowany dla wszystkich dostępnych urządzeń obliczeniowych;

• `options` – wskaźnik do ciągu znaków, w którym podano dodatkowe opcje, jakie zostaną użyte w procesie kompilacji oraz konsolidacji procedury obliczeniowej;

• `pfn_notify` – adres do funkcji zwrotnej, która zostanie wywołania przez sterownik OpenCL w momencie zakończenia budowy programu jądra obliczeniowego, zarówno w przypadku sukcesu, jak też, jeśli w procesie budowy wystąpią błędy. Wywołanie tej funkcji może być całkowicie asynchroniczne, więc należy zapewnić, aby była to funkcja przystosowana do pracy w oddzielnym wątku. Funkcja zwrotna przyjmuje dwa argumenty – obiekt programu oraz dodatkowe dane od użytkownika. W przypadku podania wartości `NULL`, funkcja zwrotna nie zostanie wywołana, a po zleceniu procesu budowania programu funkcja `clBuildProgram` zakończy działanie dopiero po zakończonym procesie budowy programu. W przypadku podania adresu do funkcji zwrotnej, funkcja `clBuildProgram` rozpocznie proces budowy programu, a następnie przekaże sterowanie do następnej funkcji w programie gospodarza;

• `user_data` – dodatkowe dane użytkownika, które zostaną przekazane do argumentu `user_data` podczas wywołania funkcji `pfn_notify`. Pole to może też mieć wartość `NULL`.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Główne zadanie tej funkcji to utworzenie postaci binarnej jądra obliczeniowego lub biblioteki, która może zostać wykorzystana później. Opcje przekazywane do `clBuildProgram` są podobne do opcji kompilatorów języka C/C++, szczególne dla kompilatora gcc. Poniżej podano wybrany zestaw dodatkowych przełączników, które na mocy standardu OpenCL muszą być wspierane przez kompilator procedur obliczeniowych dostępny w ramach funkcji `clBuildProgram` (możliwe jest też stosowanie zewnętrznego kompilatora, np. w przypadku urządzeń / platform mobilnych). Producenci konkretnego sterownika OpenCL mogą naturalnie oferować dodatkowe opcje obsługujące proces kompilacji, lecz nie będą to rozwiązania uniwersalne.

Lista poszczególnych opcji zgodnych ze specyfikacją OpenCL jest następująca:

• `-D name` – definicja stałej dla preprocesora o nazwie `name` i wartości `1`, odpowiednik `#define name 1`;

• `-D name=value` – bezpośredni odpowiednik dyrektywy `#define name value`

• `-I dir` – dodanie katalogu o nazwie `dir` do listy przeszukiwanych katalogów dla plików nagłówkowych;

• `-cl-single-precision-constant` – stałe liczbowe typu `double` będą traktowane jak wartości `float`;

• `-cl-denorms-are-zero` – użycie tej opcji oznacza, iż wartości, w przypadku liczb `double` oraz `float`, będą automatycznie zamieniane na zera. Jeśli dana implementacja OpenCL nie obsługuje wartości denormalnych (informacje o tym można uzyskać za pomocą `CL_DEVICE_SINGLE_FP_CONFIG`), to opcja jest ignorowana. Należy też podkreślić, iż dane w przypadku obrazów nie są objęte zamianą wartości denormalnych na zera, dotyczy to tylko operacji na skalarach oraz wektorach;

• `-cl-fp32-correctly-rounded-divide-sqrt` – użycie tej opcji oznacza, iż operacje dzielenia oraz obliczania wartości pierwiastka kwadratowego będą poprawnie zaokrąglane, w przeciwnym przypadku stosowane będą metody o niższej dokładności. Wymagane jest także, aby dane urządzenie obliczeniowe oferowało taką możliwość, co oznacza, iż flaga `CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT`, jest ustawiona w informacji odczytanej za pomocą stałej `CL_DEVICE_SINGLE_FP_CONFIG`. W przypadku, gdy dane urządzenie obliczeniowe nie oferuje dostępu do tej możliwości, próba kompilacji z tą opcją spowoduje jej przerwanie i zwrócenie błędu;

• `-cl-opt-disable` – wyłącznie optymalizacji, domyślnie optymalizacja kodu jest włączona;

• `-cl-strict-aliasing` – włączenie stosowania reguł aliasingu;

• `-cl-mad-enable` – wyrażenia typu `a * b + c` zostaną zastąpione wywołaniem funkcji `mad`, która oblicza wartość `a * b + c` w niższej dokładności, szczególnie dotyczy to wartości iloczynu `a * b`;

• `-cl-no-signed-zeros` – opcja uruchamia optymalizację arytmetyki zmiennoprzecinkowej, w której w przypadku operacji z zerem wartość znaku dla zera jest ignorowana. Należy podkreślić, że standard IEEE 754 określa zachowanie się operacji w przypadku, gdy zero zawiera informację o znaku. Użycie opcji `– cl-no-signed-zeros` wyłącza to zachowanie;

• -cl-unsafe-math-optimizations – opcja umożliwia stosowanie tzw. niebezpiecznych optymalizacji (obejmuje to opcje – cl-no-signed-zeros oraz – cl-mad-enable-options. Należy podkreślić, iż uruchomienie tej opcji jest polecane tylko wówczas, gdy obliczenia zawsze generują poprawne rezultaty. Zastosowanie tej opcji oznacza, iż nie będą obowiązywać reguły standardu IEEE-754 do przetwarzania wartości zmiennoprzecinkowych, a także reguły przetwarzania wartości zmiennoprzecinkowych wymagane przez standard OpenCL;

• -cl-finite-math-only – użycie tej opcji zezwala na optymalizację operacji zmiennoprzecinkowych, gdzie zakłada się, iż nie wystąpią wartości oznaczające nieskończoność bądź kod NaN oznaczający, iż wartość nie jest liczbą. Analogicznie jak opcja –cl-unsafe-math-optimizations, zastosowanie opcji –cl-finite-math-only może złamać reguły przetwarzania wartości zmiennoprzecinkowych zapisane w standardzie OpenCL;

• -cl-fast-relaxed-math – użycie tej opcji oznacza, iż zostaną użyte również opcje –cl-finite-math-only, a także –cl-unsafe-math-optimizations. Analogicznie jak poprzednio, powoduje to złamanie reguł przetwarzania wartości zmiennoprzecinkowych w ramach standardu IEEE-754. Zdefiniowane zostanie także makro __FAST_RELAXED_MATH__ dostępne z poziomu jądra obliczeniowego;

• -w – ukrycie wszystkich ostrzeżeń;

• -Werror – wszystkie ostrzeżenia są traktowane jak błędy;

• -create-library – opcja pozwala utworzyć bibliotekę z plików binarnych, która później może być stosowana za pomocą m.in. funkcji clLinkProgram;

• -enable-link-options – opcja pozwala na aktywację funkcji linkera (programu konsolidującego) w trakcie budowy postaci binarnej programu obliczeniowego. Opcja ta powinna być stosowana razem z opcją –create-library.

Jeśli wynikiem funkcji (wartością końcową) clBuildProgram jest wartość CL_SUCCESS, to wykonanie funkcji zakończyło się sukcesem. W każdym innym przypadku funkcja zwraca jedną z następujących wartości:

• CL_INVALID_PROGRAM – podany obiekt programu nie jest poprawnym obiektem;

• CL_INVALID_VALUE – parametr pfn_notify ma wartość NULL, podczas gdy parametr user_data jest różny od NULL, lub argument device_list jest równy NULL, podczas gdy liczba urządzeń (num_devies) jest większa od zera, bądź też liczba urządzeń jest równa zeru, ale wartość device_list jest różna od NULL;

• CL_INVALID_DEVICE – urządzenie lub urządzenia obliczeniowe wskazane w parametrze device_list nie jest lub nie są urządzeniemi, dla których utworzono program obliczeniowy wskazany w parametrze program,

• CL_INVALID_BINARY – program został utworzony za pomocą clCreateProgramWithBinary, jednak istnieje urządzenie podane w parametrze device_list, dla którego nie podano poprawnej postaci programu binarnego;

• `CL_INVALID_BUILD_OPTIONS` – opcje podane do budowy programu nie są poprawne;

• `CL_COMPILER_NOT_AVAILABLE` – podano program utworzony za pomocą funkcji `clCreateProgramWithSource`, a dla danej platformy i urządzenia nie ma dostępnego wbudowanego kompilatora języka OpenCL C;

• `CL_BUILD_PROGRAM_FAILURE` – proces budowy programu zakończył się błędem;

• `CL_INVALID_OPERATION` – zamiast programu podano obiekt jądra obliczeniowego lub wskazano obiekt programu, dla którego nie zakończono procesu budowy programu, lub też zakończył się on błędem;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• budowa programu dla podanego obiektu jądra obliczeniowego:

```
cli_err_num = clBuildProgram( cl_kernel_program, clui_num_devices, platform_devices, NULL, NULL, NULL);
```

• budowa programu wraz z przekazaniem dodatkowych opcji dla preprocesora:

```
cli_err_num = clBuildProgram( cl_kernel_program, clui_num_devices, platform_devices, preprocesor_options, NULL, NULL);
```

D.8. Funkcja clGetProgramBuildInfo

Funkcja `clGetProgramBuildInfo` umożliwia odczytanie informacji o programie po procesie budowania wersji wykonawczej. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clGetProgramBuildInfo ( cl_program program,

cl_device_id device,

    cl_program_build_info param_name,

    size_t param_value_size,

    void *param_value,

    size_t *param_value_size_ret);
```

Znaczenie poszczególnych parametrów:

• `program` – obiekt programu, dla którego będą odczytywane dodatkowe informacje;

• `device` – obiekt reprezentuje urządzenie, dla którego zbudowano program, toteż musi to być poprawne urządzenie, które zostało wcześniej skojarzone z programem;

• `param_name` – parametr określa, jaki typ informacji zostanie odczytany. Lista poprawnych wartości tego parametru została przedstawiona w tab. D.5, zawierającej także informacje o typie, jaki jest zwracany dla danego parametru;

• `param_value_size` – w parametrze należy określić wielkość bufora pamięci, w której zostaną umieszczone odczytane informacje, wielkość ta powinna być większa lub przynajmniej równa liczbie zwracanych danych, określonych w tab. D.5;

• `param_value` – wskaźnik do pamięci gospodarza, w której zostaną umieszczone odczytane informacje, podanie wartości `NULL` oznacza pominięcie tego parametru;

• `param_value_size_ret` – parametr pozwala odczytać rozmiar informacji, która zostanie umieszczona w parametrze `param_value`, ilość informacji jest określana w bajtach, podanie wartości `NULL` oznacza pominięcie tego parametru;

Tabela D.5. Stałe dla parametru `param_name` dla funkcji `clGetProgramBuildInfo`

cl_program_buid_info	Typ zwracanej informacji	Opis
`CL_PROGRAM_BUILD_STATUS`	cl_build_status	Stała pozwala odczytać stan programu – czy został zbudowany, poddany kompilacji – oraz stan procesu konsolidacji. W tym celu stosowane są następujące stałe: • `CL_BUILD_NONE` – nie były wykonywane akcje zwiazane z funkcjami `clBuildProgram`, `clCompileProgram` lub też `clLinkProgram`, na obiekcie reprezentującym program; • `CL_BUILD_ERROR` – wystąpił błąd podczas realizacji jednej z funkcji `clBuildProgram`, `clCompileProgram` lub też `clLinkProgram` na obiekcie reprezentującym program; • `CL_BUILD_SUCCESS` – wywołanie funkcji `clBuildProgram`, `clCompileProgram` lub też `clLinkProgram` zostało zakończone pełnym sukcesem; • `CL_BUILD_IN_PROGRESS` – akcje wykonywane za pomocą funkcji `clBuildProgram`, `clCompileProgram` lub też `clLinkProgram` nie zostały jeszcze zakończone.
`CL_PROGRAM_BUILD_OPTIONS`	char[]	Stała pozwala odczytać, jakie opcje zostały przekazane do funkcji `clBuildProgram`, `clCompileProgram` lub też `clLinkProgram`. W przypadku, gdy status budowy programu to `CL_BUILD_NONE`, zwracany jest pusty ciąg znaków.
`CL_PROGRAM_BUILD_LOG`	char[]	Stała pozwala odczytać log zawierający informacje zwrócone przez funkcje `clBuildProgram`, `clCompileProgram` lub też `clLinkProgram`. Mogą tam zostać umieszczone informacje np. o błędach składniowych typu brakujący

		przecinek. W przypadku, gdy status budowy programu to CL_BUILD_NONE, zwracany jest pusty ciąg znaków.
CL_PROGRAM_BINARY_TYPE	cl_program_binary_type	Stała odczytuje, jaki typ binarnego programu został umieszczony w urządzeniu. Wartość CL_PROGRAM_BINARY_TYPE_NONE oznacza, iż postać binarna nie jest dołączona do urządzenia. Wartość CL_PROGRAM_BINARY_TYPE_COMPILED_OBJECT oznacza, iż program binarny powstał za pomocą clCreateProgramWithSource oraz funkcji clCompileProgram lub został załadowany za pomocą funkcji clCreateProgramWithBinary. Wartość CL_PROGRAM_BINARY_TYPE_LIBRARY oznacza, że do urządzenia dołączono bibliotekę binarną. Wartość CL_PROGRAM_BINARY_TYPE_EXECUTABLE oznacza, iż dołączono wykonywalną postać programu, co pozwala stwierdzić, że nie dołączono biblioteki binarnej.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Należy dodać, iż standard OpenCL pozwala, aby urządzenia obliczeniowe były zorganizowane w drzewo. Jednak w takim przypadku, program dla urządzenia rodzicielskiego musi być także zbudowany dla wszystkich urządzeń podległych urządzeniu rodzicielskiemu.

Funkcja zwraca wartość CL_SUCCESS, jeśli operacja zakończyła się poprawnie. W przeciwnym przypadku można otrzymać jeden z następujących kodów:

• CL_INVALID_DEVICE – urządzenie nie jest na liście urządzeń odnoszących się do obiektu programu;

• CL_INVALID_VALUE – param_name nie otrzymał predefiniowanej stałej (podanej w tab. D.5) lub podana ilość pamięci nie jest wystarczająca, aby zapisać poprawne zwracane informacje, co oznacza, że param_value w tym przypadku nie jest wartością NULL.

• CL_INVALID_PROGRAM – podany obiekt nie reprezentuje poprawnego programu;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• odczytanie informacji o błędach (z logu kompilatora OpenCL C) np. składniowych popełnionych w kodzie źródłowym jądra obliczeniowego:

```
clGetProgramBuildInfo(cl_kernel_program, platform_devices[0], CL_PROGRAM_BUILD_LOG,
sizeof(buffer), buffer, &len);
```

D.9. Funkcja clCreateCommandQueue

Funkcja `clCreateCommandQueue` tworzy kolejkę zadań dla podanego kontekstu obliczeniowego. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_command_queue clCreateCommandQueue( cl_context context,

cl_device_id device,

    cl_command_queue_properties properties,

 cl_int *errcode_ret);
```

Znaczenie poszczególnych parametrów:

• `context` – obiekt reprezentujący kontekst obliczeniowy OpenCL;

• `device` – urządzenie obliczeniowe (takie samo jak w kontekście obliczeniowym);

• `properties` – w parametrze można podać dodatkowe wymagane własności, jakie ma posiadać tworzona kolejka poleceń, obecnie dopuszczalne są dwie stałe (mogą być łączone za pomocą operatora bitowego `lub`): `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`, która określa sposób przetwarzania w tzw. trybie out-order, polecenia nie muszą być wykonywane w kolejności ich umieszczenia w kolejce. Jeśli nie użyto tej własności, to polecenia są wykonywane w kolejności umieszczenia ich w kolejce (in-order). Druga stała to `CL_QUEUE_PROFILING_ENABLE`, oznaczająca, że kolejka zbierać będzie dane o wydajności, np. czasy działania poszczególnych operacji kopiowania, wykonania jądra etc.;

• `errcode_ret` – parametr zwraca ewentualny kod błędu, podanie wartość `NULL` oznacza, iż kod nie zostanie zwrócony.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Kolejka poleceń wykonuje zadania na obiektach pamięciowych, programach i jądrach obliczeniowych. Zastosowanie kolejki to także synchronizacja operacji wykonywanych na wymienionych obiektach. Dopuszcza się także obecność kilku kolejek poleceń, choć w takim przypadku należy zwrócić uwagę na synchronizację poszczególnych operacji na obiektach.

Funkcja zwraca obiekt reprezentujący kolejkę obliczeniową oraz wartość `CL_SUCCESS` umieszczoną w parametrze `errcode_ret`, jeśli operacja zakończyła się poprawnie. W przeciwnym przypadku zwrócona zostanie wartość `NULL`, natomiast w parametrze `errcode_ret` jest umieszczany jeden z następujących kodów oznaczających błąd:

• `CL_INVALID_CONTEXT` – podany obiekt nie jest poprawnym kontekstem obliczeniowym;

• `CL_INVALID_DEVICE` – obiekt reprezentujący urządzenie nie jest poprawny lub nie jest powiązany z kontekstem;

• `CL_INVALID_VALUE` – stała opisująca własność nie jest poprawna;

• CL_INVALID_QUEUE_PROPERTIES – własności podane w momencie tworzenia kolejki nie są wspierane (obsługiwane) przez dane urządzenie obliczeniowe;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• utworzenie kolejki poleceń:

```
cl_compute_command_queue = clCreateCommandQueue( cl_compute_context, platform_devices[0], 0, &cli_err_num);
```

D.10. Funkcja clCreateBuffer

Funkcja tworzy bufor w pamięci urządzenia obliczeniowego. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_mem clCreateBuffer ( cl_context context,

    cl_mem_flags flags,

    size_t size,

    void *host_ptr,

    cl_int *errcode_ret);
```

Znaczenie poszczególnych parametrów:

• context – obiekt reprezentujący kontekst obliczeniowy;

• flags – parametr reprezentuje pole bitowe, gdzie można podać dodatkowe atrybuty co do przeznaczenia bufora umieszczonego w pamięci urządzenia obliczeniowego, np. można określić, że dany bufor jest przeznaczony tylko do zapisu. Informacje o tym, jakie flagi można stosować, podano w tab. D.6. Podanie wartości zero oznacza, iż bufor można zapisywać oraz odczytywać, co odpowiada stałej CL_MEM_READ_WRITE;

Tabela D.6. Stała opisująca dodatkowe własności obiektów pamięciowych

Stała cl_mem_flags	Opis

`CL_MEM_READ_WRITE`	Stała oznacza, iż przydzielony obszar będzie można zarówno zapisywać, jak też odczytywać.
`CL_MEM_WRITE_ONLY`	Stała oznacza, iż na obiekcie pamięciowym procedura obliczeniowa będzie tylko zapisywać dane. Zachowanie się systemu obliczeniowego przy próbie odczytu danych nie jest sprecyzowane, podobnie zachowanie się sterownika OpenCL po stronie gospodarza nie jest określone
`CL_MEM_READ_ONLY`	Stała oznacza, iż z obiektu pamięciowego (lecz nie dla obiektu zawierającego obraz) procedura obliczeniowa będzie tylko odczytywać dane. Zachowanie się systemu obliczeniowego przy próbie zapisu danych nie jest sprecyzowane, podobnie zachowanie się sterownika OpenCL po stronie gospodarza nie jest określone.
`CL_MEM_USE_HOST_PTR`	Użycie flagi jest poprawne tylko wówczas, gdy parametr `host_ptr` jest różny od `NULL`. Flaga oznacza, że obiekt pamięciowy OpenCL będzie również używał pamięci gospodarza. Oznacza to, że sterownik OpenCL będzie dynamicznie tworzył kopię „cache” w pamięci urządzenia, w przypadku gdy będzie uruchamiane jądro obliczeniowe, które korzysta z obiektu pamięciowego.
`CL_MEM_ALLOC_HOST_PTR`	Flaga zezwala na alokację pamięci po stronie gospodarza przez funkcje API OpenCL.
`CL_MEM_COPY_HOST_PTR`	Flaga może zostać zastosowana tylko wówczas, gdy parametr `host_ptr` jest różny od `NULL`. Oznacza to, iż sterownik OpenCL będzie kopiował wskazany obszar pamięci gospodarza do pamięci urządzenia obliczeniowego.
`CL_MEM_HOST_WRITE_ONLY`	Flaga oznacza, że gospodarz będzie tylko zapisywał dany obiekt pamięciowy.
`CL_MEM_HOST_READ_ONLY`	Flaga oznacza, że gospodarz będzie tylko odczytywał dany obiekt pamięciowy.
`CL_MEM_HOST_NO_ACCESS`	Flaga oznacza, iż dostęp do obiektu pamięciowego zostanie zablokowany.

• `size` – rozmiar obszaru pamięci, jaka zostanie przydzielona do obiektu;

• `host_ptr` – wskaźnik do danych po stronie gospodarza, wielkość bufora hosta powinna być równa lub większa niż wielkość podana w parametrze `size`;

• `errcode_ret` – parametr zwraca ewentualny kod błędu, podanie wartość `NULL` oznacza, iż kod nie zostanie zwrócony.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Funkcja tworzy poprawny obiekt pamięciowy oraz wstawia do parametru `errcode_ret` wartość `CL_SUCCESS`, jeśli udało się ten proces ukończyć z sukcesem. W przeciwnym przypadku zwrócona zostanie wartość `NULL`, a parametr `errcode_ret` otrzyma jeden z poniższych kodów:

• `CL_INVALID_CONTEXT` – podany obiekt kontekstu nie jest poprawnym kontekstem;

• `CL_INVALID_VALUE` – stałe podane w parametrze `flags` nie są poprawnymi wartościami;

• `CL_INVALID_BUFFER_SIZE` – parametr `size` jest równy zeru, lub też wartość podana w `size` przekracza wartość określoną przez stałą `CL_DEVICE_MAX_MEM_ALLOC_SIZE`;

• `CL_INVALID_HOST_PTR` – `host_ptr` jest równy wartości `NULL` oraz użyto flagi `CL_MEM_USE_HOST_PTR` lub `CL_MEM_COPY_HOST_PTR`, bądź parametr `host_ptr` jest różny od `NULL`, ale pominięto użycie flag `CL_MEM_COPY_HOST_PTR`, lub `CL_MEM_USE_HOST_PTR`;

• `CL_MEM_OBJECT_ALLOCATION_FAILURE` – nie udało się przydzielić pamięci dla obiektu pamięciowego;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• utworzenie bufora (obiektu pamięciowego) tylko do odczytu:

```
device_in_a = clCreateBuffer( cl_compute_context, CL_MEM_READ_ONLY, dataBytes, NULL, &cli_err_num);
```

• utworzenie bufora (obiektu pamięciowego) tylko do zapisu:

```
device_out_c = clCreateBuffer( cl_compute_context, CL_MEM_WRITE_ONLY, dataBytes, NULL, &cli_err_num);
```

D.11. Funkcja clEnqueueReadBuffer

Funkcja `clEnqueueReadBuffer` kopiuje wskazany obszar pamięci z urządzenia obliczeniowego do pamięci po stronie gospodarza. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clEnqueueReadBuffer ( cl_command_queue command_queue,

cl_mem buffer,

    cl_bool blocking_read,

    size_t offset,
```

```
    size_t size,

    void *ptr,

    cl_uint num_events_in_wait_list,

    const cl_event *event_wait_list,

cl_event *event);
```

Znaczenie poszczególnych parametrów:

• `command_queue` – obiekt wskazujący na kolejkę poleceń; należy podkreślić, że wskazana kolejka oraz bufor muszą być utworzone w ramach tego samego kontekstu;

• `buffer` – poprawny obiekt pamięciowy;

• `blocking_read` – parametr pozwalający stwierdzić, czy operacja odczytu danych jest operacją blokującą (należy podać wartość `CL_TRUE`), co oznacza iż funkcja `clEnqueueReadBuffer` nie przekaże sterowania tak długo, jak długo będą kopiowane dane. W przypadku podania wartości `CL_FALSE`, proces kopiowania danych jest procesem nieblokującym i należy skorzystać z mechanizmu zdarzeń, aby wykryć jego zakończenie. Należy też zwrócić uwagę na to, aby nie używać wskaźnika po stronie gospodarza oraz obiektu pamięciowego dopóki proces kopiowania danych nie zostanie zakończony;

• `offset` – wartość dodatkowego przesunięcia;

• `size` – wielkość w bajtach danych, jakie zostaną odczytane;

• `ptr` – wskaźnik do obszaru pamięci gospodarza, gdzie zostaną zapisane dane;

• `event_wait_list` oraz `num_events_in_wait_list` – w parametrze `event_wait_list` można umieścić obiekty, które reprezentują zdarzenia, jakie muszą zostać zrealizowane przed rozpoczęciem procesu odczytu danych. Wartość `NULL` oznacza, iż proces odczytu danych może zostać rozpoczęty natychmiast – pociąga to za sobą konieczność podania zera w parametrze `num_events_in_wait_list`. Wartość większa niż zero jest interpretowana jako liczba zdarzeń umieszczona w `event_wait_list`. Należy zadbać o to, aby wszystkie zdarzenia były utworzenie za pomocą tego samego kontekstu obliczeniowego co wskazana kolejka poleceń;

• `event` – parametr pozwala odczytać obiekt zdarzenia oznaczający zakończenie procesu odczytu danych. Podanie wartości `NULL` oznacza, że nie będziemy odczytywać obiektu reprezentującego zdarzenie odczytu danych. Należy zwrócić uwagę, że jeśli parametry `event_wait_list` oraz `num_events_in_wait_list` nie mają wartości `NULL`, to obiekt `event` nie może być elementem, który w jakiś sposób odnosi się do zdarzeń umieszczonych w `event_wait_list`.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

W przypadku, gdy funkcja `clEnqueueReadBuffer` odczytuje dane z bufora przy użyciu przesunięcia, tj. parametru `offset`, należy zwrócić szczególną uwagę na to, aby inne polecenia nie korzystały z obszaru danych, z którego następuje odczyt, a także jest wymagane, aby obszar pamięci nie był odwzorowany[1] za pomocą mechanizmów OpenCL.

Funkcja clEnqueueReadBuffer zwraca wartość CL_SUCCESS, jeśli operacja odczytu danych zakończyła się poprawnie. W przeciwnym przypadku można otrzymać jeden z następujących kodów:

• CL_INVALID_COMMAND_QUEUE – podany obiekt reprezentujący kolejkę poleceń nie jest poprawnym obiektem;

• CL_INVALID_CONTEXT – kontekst obliczeniowy dla kolejki oraz bufora z danymi nie jest taki sam lub kontekst i kolejka są różne w zdarzeniach umieszczonych w parametrze event_wait_list;

• CL_INVALID_MEM_OBJECT – podany obiekt reprezentujący bufor nie jest poprawny;

• CL_INVALID_VALUE – wartości wskazywane przez offset oraz size są zbyt duże lub wartość ptr jest równa NULL, lub wartość size jest równa zero;

• CL_INVALID_EVENT_WAIT_LIST – liczba zdarzeń w parametrze num_events_in_wait_list jest większa od zera, ale parametr event_wait_list jest równy NULL, lub na odwrót – parametr event_wait_list jest różny od NULL, ale liczba zdarzeń została ustalona na zero, bądź też obiekty reprezentujące zdarzenia podane w argumencie event_wait_list nie są poprawnymi obiektami zdarzeń;

• CL_MISALIGNED_SUB_BUFFER_OFFSET – bufor (bądź pod-bufor), a także wartość offset nie są wyrównane zgodnie z wartością, jaką określono za pomocą CL_DEVICE_MEM_BASE_ADDR_ALIGN dla urządzenia stowarzyszonego z daną kolejką poleceń;

• CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST – operacje odczytu i zapisu są operacjami blokującymi oraz status jakiegokolwiek zdarzenia umieszczonego w parametrze event_wait_list jest wartością ujemną;

• CL_MEM_OBJECT_ALLOCATION_FAILURE – nie udało się przydzielić pamięci dla bufora;

• CL_INVALID_OPERATION – wywołano funkcję clEnqueueReadBuffer dla bufora, którego własności są opisane przez stałe CL_MEM_HOST_WRITE_ONLY lub CL_MEM_HOST_NO_ACCESS;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• odczyt danych z urządzenia obliczeniowego do pamięci gospodarza:

```
cli_err_num = clEnqueueReadBuffer( cl_compute_command_queue, device_out_c, CL_TRUE, 0,
dataBytes, &host_out_c[0], 0, NULL, NULL);
```

D.12. Funkcja clEnqueueWriteBuffer

Funkcja clEnqueueWriteBuffer kopiuje wskazany obszar pamięci z pamięci gospodarza do obszaru

pamięci urządzenia obliczeniowego. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clEnqueueWriteBuffer ( cl_command_queue command_queue,

cl_mem buffer,

    cl_bool blocking_write,

    size_t offset,

    size_t size,

    const void *ptr,

    cl_uint num_events_in_wait_list,

    const cl_event *event_wait_list,

 cl_event *event);
```

Znaczenie poszczególnych parametrów:

• `command_queue` – obiekt wskazujący na kolejkę poleceń; należy podkreślić, że wskazana kolejka oraz bufor muszą być utworzone w ramach tego samego kontekstu;

• `buffer` – poprawny obiekt pamięciowy;

• `blocking_write` – parametr pozwala stwierdzić, czy operacja zapisu danych jest operacją blokującą (należy podać wartość `CL_TRUE`), co oznacza, iż funkcja `clEnqueueWriteBuffer` nie przekaże sterowania tak długo, jak długo będą kopiowane dane. W przypadku podania wartości `CL_FALSE`, proces kopiowania danych jest procesem nieblokującym i należy skorzystać z mechanizmu zdarzeń, aby wykryć jego zakończenie. Należy też zwrócić uwagę na to, aby nie używać wskaźnika po stronie gospodarza oraz obiektu pamięciowego dopóki proces kopiowania danych nie zostanie zakończony;

• `offset` – wartość dodatkowego przesunięcia;

• `size` – wielkość w bajtach danych, które zostaną zapisane;

• `ptr` – wskaźnik do obszaru pamięci gospodarza, reprezentuje obszar danych źródłowych, które zostaną przeniesione do pamięci urządzenia obliczeniowego;

• `event_wait_list` oraz `num_events_in_wait_list` – w parametrze `event_wait_list` można umieścić obiekty, które reprezentują zdarzenia, jakie muszą zostać zrealizowane przed rozpoczęciem procesu zapisu danych. Wartość `NULL` oznacza, iż proces odczytu danych może zostać rozpoczęty natychmiast – pociąga to za sobą konieczność podania zera w parametrze `num_events_in_wait_list`. Wartość większa niż zero jest interpretowana jako liczba zdarzeń umieszczona w `event_wait_list`. Należy zadbać o to, aby wszystkie zdarzenia były utworzone za pomocą tego samego kontekstu obliczeniowego co wskazana kolejka poleceń;

• event – parametr pozwala odczytać obiekt zdarzenia oznaczający zakończenie procesu zapisu danych. Podanie wartości NULL oznacza, że nie będziemy odczytywać obiektu reprezentującego zdarzenie zapisu danych. Należy zwrócić uwagę, że jeśli parametry event_wait_list oraz num_events_in_wait_list nie mają wartości NULL, to obiekt event nie może być elementem, który w jakiś sposób odnosi się do zdarzeń umieszczonych w event_wait_list.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

W przypadku, gdy funkcja clEnqueueWriteBuffer zapisuje dane do bufora umieszczonego w pamięci urządzenia obliczeniowego przy użyciu przesunięcia, tj. parametru offset, należy zwrócić dodatkową uwagę na to, aby inne polecenia nie korzystały z obszaru danych, z którego następuje odczyt, a także jest wymagane, aby obszar pamięci nie był odwzorowany za pomocą mechanizmów OpenCL.

Funkcja clEnqueueWriteBuffer zwraca wartość CL_SUCCESS, jeśli operacja zapisu danych zakończyła się poprawnie. W przeciwnym przypadku można otrzymać jeden z następujących kodów:

• CL_INVALID_COMMAND_QUEUE – podany obiekt reprezentujący kolejkę poleceń nie jest poprawnym obiektem;

• CL_INVALID_CONTEXT – kontekst obliczeniowy dla kolejki oraz bufora z danymi nie jest taki sam lub kontekst i kolejka są różne w zdarzeniach umieszczonych w parametrze event_wait_list;

• CL_INVALID_MEM_OBJECT – podany obiekt reprezentujący bufor nie jest poprawny;

• CL_INVALID_VALUE – wartości wskazywane przez offset oraz size są zbyt duże lub wartość ptr jest równa NULL, lub wartość size jest równa zero;

• CL_INVALID_EVENT_WAIT_LIST – liczba zdarzeń w parametrze num_events_in_wait_list jest większa od zera, ale parametr event_wait_list jest równy NULL, lub na odwrót – parametr event_wait_list jest różny od NULL, ale liczba zdarzeń została ustalona na zero, bądź też obiekty reprezentujące zdarzenia podane w argumencie event_wait_list nie są poprawnymi obiektami zdarzeń;

• CL_MISALIGNED_SUB_BUFFER_OFFSET – bufor (bądź pod-bufor), a także wartość offset, nie są wyrównane zgodnie z wartością, jaką określono za pomocą CL_DEVICE_MEM_BASE_ADDR_ALIGN dla urządzenia stowarzyszonego z daną kolejką poleceń;

• CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST – operacje odczytu i zapisu są operacjami blokującymi oraz status jakiegokolwiek zdarzenia umieszczonego w parametrze event_wait_list jest wartością ujemną;

• CL_MEM_OBJECT_ALLOCATION_FAILURE – nie udało się przydzielić pamięci dla bufora;

• CL_INVALID_OPERATION – nastąpiło wywołanie clEnqueueWriteBuffer dla bufora, który został utworzony z własnościami opisanymi za pomocą następujących stałych: CL_MEM_HOST_READ_ONLY lub CL_MEM_HOST_NO_ACCESS;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie

urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• zapis danych do pamięci urządzenia obliczeniowego:

```
cli_err_num = clEnqueueWriteBuffer( cl_compute_command_queue, device_in_a, CL_TRUE, 0, dataBytes, &host_in_a[0], 0, NULL, NULL);
```

D.13. Funkcja clCreateProgramWithSource

Funkcja `clCreateProgramWithSource` tworzy obiekt programu dla podanego kontekstu obliczeniowego, a następnie przekazuje podany kod źródłowy w postaci ciągu znaków do utworzonego obiektu. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_program clCreateProgramWithSource ( cl_context context,

cl_uint count,

    const char **strings,

    const size_t *lengths,

    cl_int *errcode_ret);
```

Znaczenie poszczególnych parametrów:

• `context` – obiekt reprezentujący poprawny kontekst obliczeniowy OpenCL;

• `count` – liczba wskaźników podanych w parametrze `strings`;

• `strings` – tablica ciągów znaków zawierająca kod źródłowy jądra obliczeniowego;

• `lengths` – tablica z długościami każdego ciągu znaków podanego w parametrze `strings`; podanie wartości zero oznacza, iż ciągi znaków będą zakończone wartością `NULL`;

• `errcode_ret` – parametr zwraca ewentualny kod błędu; podanie wartość `NULL` oznacza, iż kod nie zostanie zwrócony.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Funkcja tworzy obiekt program dla podanego kontekstu oraz wczytuje kod źródłowy do obiektu programu. Urządzenie stowarzyszone z programem będzie tym samym urządzeniem, które zostało podane w kontekście obliczeniowym.

Funkcja tworzy poprawny obiekt programu oraz wstawia do parametru `errcode_ret` wartość

CL_SUCCESS, jeśli udało się ten proces ukończyć z sukcesem. W przeciwnym przypadku zwrócona zostanie wartość NULL, a parametr errcode_ret otrzyma jeden z poniższych kodów:

• CL_INVALID_CONTEXT – obiekt reprezentujący kontekst nie jest poprawny;

• CL_INVALID_VALUE – wartość count jest równa zero lub podano wartość NULL do parametru strings, lub wartość tablicy strings to wartości NULL;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• utworzenie obiektu programu obliczeniowego na podstawie podanego ciągu znaków zakończonych wartością NULL:

```
cl_kernel_program = clCreateProgramWithSource( cl_compute_context, 1, (const char**)&kernelSource, NULL, &cli_err_num);
```

D.14. Funkcja clCreateKernel

Funkcja clCreateKernel tworzy obiekt jądra obliczeniowego, który jest zwracany jako rezultat funkcji. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_kernel clCreateKernel ( cl_program program,

const char *kernel_name,

    cl_int *errcode_ret);
```

Znaczenie poszczególnych parametrów:

• program – obiekt reprezentujący program, dla którego poprawnie zbudowano postać binarną;

• kernel_name – nazwa funkcji, którą opatrzono kwalifikatorem kernel lub __kernel;

• errcode_ret – parametr zwraca ewentualny kod błędu, podanie wartości NULL oznacza, że kod nie zostanie zwrócony.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Jądro obliczeniowe w kodzie źródłowym zawsze jest opatrzone kwalifikatorem __kernel. W danym kodzie źródłowym można umieścić kilka funkcji opatrzonych kwalifikatorem __kernel, a dostęp do każdej z nich można uzyskać za pomocą clCreateKernel, tworząc kilka

obiektów reprezentujących poszczególne procedury obliczeniowe.

Funkcja zwraca obiekt reprezentujący jądro obliczeniowe oraz wartość `CL_SUCCESS` umieszczoną w parametrze `errcode_ret`, jeśli operacja zakończyła się poprawnie. W przeciwnym przypadku zwrócona zostanie wartość `NULL`, a w parametrze `errcode_ret` będzie umieszczony jeden z następujących kodów oznaczających błąd:

• `CL_INVALID_PROGRAM` – podany obiekt programu nie jest poprawny;

• `CL_INVALID_PROGRAM_EXECUTABLE` – postać wykonywalna programu z jądrem obliczeniowym nie jest poprawna;

• `CL_INVALID_KERNEL_NAME` – w programie nie występuje jądro o nazwie podanej w parametrze `kernel_name`;

• `CL_INVALID_KERNEL_DEFINITION` – istnieje kilka funkcji opatrzonych atrybutem `__kernel`, np. dla kilku urządzeń obliczeniowych, jednak w zależności od urządzenia obliczeniowego poszczególne jądra obliczeniowe posiadają argumenty o różnych typach;

• `CL_INVALID_VALUE` – parametr `kernel_name` otrzymał wartość `NULL`;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• utworzenie obiektu jądra obliczeniowego:

```
kernel = clCreateKernel(cl_kernel_program, ”sum”, &cli_err_num );
```

D.15. Funkcja clSetKernelArg

Funkcja ustala wartości argumentów dla jądra obliczeniowego. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clSetKernelArg ( cl_kernel kernel,

    cl_uint arg_index,

    size_t arg_size,

 const void *arg_value);
```

Znaczenie poszczególnych parametrów:

• kernel – obiekt reprezentujący jądro obliczeniowe;

• arg_index – indeks argumentu, dopuszczalne są wartości w zakresie od zera do n-1, gdzie n to całkowita liczba argumentów, jaka została zadeklarowana w jądrze obliczeniowym;

• arg_size – wielkość argumentu, przy czym nie jest to rzeczywista wielkość, np. obszaru pamięci reprezentującego bufor, ale tylko wielkość struktury, która ten obszar pamięci reprezentuje, np. w przypadku wskazania obszaru w pamięci globalnej wielkość argumentu to sizeof(cl_mem), a nie rzeczywista wielkość w bajtach przydzielonego bloku pamięci;

• arg_value – wskaźnik do danych, które zostaną przekazane do jądra obliczeniowego; dane mogą zostać zmienione po użyciu funkcji wywołania jądra, tj. clEnqueueNDRangeKernel lub clEnqueueTask, bez konieczności ponownego budowania jądra obliczeniowego, typ wskaźnika danych ściśle zależy od typu podanego w jądrze obliczeniowym, np. cl_mem określa bufor danych umieszczonych w pamięci urządzenia obliczeniowego.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Choć funkcje OpenCL należy rozpatrywać jako bezpieczne w środowisku wielowątkowym, to jednak należy zwrócić uwagę na przekazywane wartości parametrów oraz obiekt jądra obliczeniowego, jeśli ten sam obiekt jądra obliczeniowego jest używany przez więcej niż jeden wątek. W takim przypadku, zachowanie nie jest określone, zwłaszcza gdy kilka wątków zmieni parametry przy wywołaniu tego samego jądra obliczeniowego.

Jeśli wynikiem funkcji (wartością końcową) clSetKernelArg jest wartość CL_SUCCESS, to wykonanie funkcji zakończyło się sukcesem. W każdym innym przypadku funkcja zwraca jedną z następujących wartości:

• CL_INVALID_KERNEL – obiekt jądra obliczeniowego nie jest poprawnym obiektem;

• CL_INVALID_ARG_INDEX – wartość arg_index nie wskazuje poprawnego argumentu;

• CL_INVALID_ARG_VALUE – wartość wskazana przez arg_value nie jest poprawna;

• CL_INVALID_MEM_OBJECT – argument oczekujący obiektu pamięciowego otrzymał niepoprawny obiekt pamięciowy;

• CL_INVALID_SAMPLER – argument oczekujący na obiekt typu sampler_t otrzymał błędny obiekt;

• CL_INVALID_ARG_SIZE – wielkość wskazana przez argument arg_size nie jest zgodna z wielkością typu obiektu niebędącego obiektem pamięciowym lub podana wielkość w przypadku obiektu pamięciowego jest różna od wartości sizeof(cl_mem) lub w przypadku samplera arg_size jest różne od sizeof(cl_sampler), lub też wartością arg_size jest zero, ale podano typ z atrybutem __local;

• CL_INVALID_ARG_VALUE – niezgodność atrybutów i opisu dostępu do danych obrazu, np. gdy obraz został zadeklarowany z atrybutem read_only, a podany obiekt obrazu został utworzony z flagą CL_MEM_WRITE;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie

urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• ustalenie argumentu w postaci obiektu pamięciowego dla jądra obliczeniowego:

```
cli_err_num = clSetKernelArg( kernel, 0, sizeof(cl_mem), &device_in_a );
```

• ustalenie argumentu w postaci zmiennej typu `int` dla jądra obliczeniowego:

```
cli_err_num = clSetKernelArg( kernel, 2, sizeof(int), &N );
```

• ustalenie obiektu pamięciowego reprezentującego obraz:

```
cli_err_num = clSetKernelArg(kernel, 1, sizeof(cl_mem), &output_image);
```

D.16. Funkcja clEnqueueNDRangeKernel

Funkcja `clEnqueueNDRangeKernel` jest odpowiedzialna za wykonanie jądra obliczeniowego za pomocą przestrzeni indeksującej (siatki obliczeniowej). **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clEnqueueNDRangeKernel ( cl_command_queue command_queue,

cl_kernel kernel,

    cl_uint work_dim,

    const size_t *global_work_offset,

    const size_t *global_work_size,

    const size_t *local_work_size,

    cl_uint num_events_in_wait_list,

    const cl_event *event_wait_list,

 cl_event *event);
```

Znaczenie poszczególnych parametrów:

• `command_queue` – parametr reprezentuje kolejkę, w której zostanie umieszczone polecenie wykonania jądra obliczeniowego;

• `kernel` – poprawny obiekt reprezentujący jądro obliczeniowe; kontekst obliczeniowy dla jądra oraz kolejki musi być taki sam;

• work_dim – liczba wymiarów, która opisuje siatkę obliczeniową, podana wartość nie może przekraczać wartości, jaka jest dopuszczalna dla danego urządzenia obliczeniowego (co można odczytać za pomocą stałej CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS);

• global_work_offset – wartość przesunięcia dodawana do globalnych identyfikatorów dla każdej jednostki roboczej, podanie wartości NULL oznacza, że wartość przesunięcia wynosi zero dla każdego wymiaru;

• global_work_size – parametr zawiera wymiary globalnej siatki obliczeniowej;

• local_work_size – parametr zawiera wymiary lokalnej grupy/siatki obliczeniowej;

• event_wait_list oraz num_events_in_wait_list – w parametrze event_wait_list można umieścić obiekty reprezentujące zdarzenia, jakie muszą zostać zrealizowane przed wykonaniem jądra obliczeniowego. Podanie wartości NULL pociąga za sobą konieczność podania zera w parametrze num_events_in_wait_list. Oznacza to także, że jądro może zostać wykonane bez oczekiwania na zakończenie innych zdarzeń. Wartość większa niż zero w parametrze num_events_in_wait_list jest interpretowana jako liczba zdarzeń umieszczona w event_wait_list. Należy zadbać o to, aby wszystkie zdarzenia były utworzenie za pomocą tego samego kontekstu obliczeniowego co wskazana kolejka poleceń.

• event – parametr pozwala odczytać obiekt zdarzenia umożliwiający identyfikację wykonywanego jądra obliczeniowego. Odczytany obiekt pozwala sprawdzić m.in., czy dane jądro obliczeniowe zakończyło swoje działanie. Podanie wartości NULL oznacza, że nie będziemy odczytywać obiektu identyfikującego jądro obliczeniowe.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Wymagane wymiary siatki roboczej można także podać w kodzie źródłowym jądra obliczeniowego za pomocą atrybutu __attribute__ ((reqd_work_group_size(X, Y, Z))), przedstawionego w rozdz. 3.4.

Funkcja zwraca wartość CL_SUCCESS, jeśli operacja zakończyła się poprawnie. W przeciwnym przypadku można otrzymać jeden z następujących kodów:

• CL_INVALID_PROGRAM_EXECUTABLE – program przekazany do urządzenia nie ma poprawnie zbudowanej postaci wykonywalnej dla urządzeń we wskazanej kolejce poleceń;

• CL_INVALID_COMMAND_QUEUE – podany obiekt kolejki poleceń nie jest poprawny;

• CL_INVALID_KERNEL – podany obiekt jądra nie jest poprawny;

• CL_INVALID_CONTEXT – kontekst obliczeniowy dla kolejki oraz bufora z danymi nie jest taki sam lub kontekst i kolejka są różne w zdarzeniach umieszczonych w parametrze event_wait_list;

• CL_INVALID_KERNEL_ARGS – jedna z wartości dla jądra obliczeniowego nie jest poprawnie podana;

• CL_INVALID_WORK_DIMENSION – niepoprawna wartość wymiaru siatki obliczeniowej, obecnie dopuszczalne wartości to 1, 2, 3;

• CL_INVALID_GLOBAL_WORK_SIZE – wartość global_work_size została określona jako NULL lub jakakolwiek z wartości w tablicy global_work_size jest równa zero, lub przekracza dopuszczalną wartość;

• CL_INVALID_GLOBAL_OFFSET – podana wartość przesunięcia oraz rozmiar siatki obliczeniowej przekraczają wielkość wyznaczoną przez sizeof(size_t);

• CL_INVALID_WORK_GROUP_SIZE – błędna wartość rozmiaru siatki lokalnej, całkowita liczba jednostek nie dzieli się całkowicie przez rozmiar grupy roboczej lub wymiar siatki roboczej nie jest zgodny z wartościami podanymi przez atrybut __attribute__ ((reqd_work_group_size(X, Y, Z))), lub wymiar grupy roboczej przekracza dopuszczalną wartość określoną przez stałą CL_DEVICE_MAX_WORK_GROUP_SIZE;

• CL_INVALID_WORK_ITEM_SIZE – liczba jednostek roboczych aktywnych w grupie roboczej przekracza wartości określone przez stałe CL_DEVICE_MAX_WORK_ITEM_SIZES dla poszczególnych wymiarów siatki roboczej;

• CL_MISALIGNED_SUB_BUFFER_OFFSET – bufor (bądź pod-bufor), a także wartość offset, nie są wyrównane zgodnie z wartością, jaką określono za pomocą CL_DEVICE_MEM_BASE_ADDR_ALIGN dla urządzenia stowarzyszonego z daną kolejką poleceń;

• CL_INVALID_IMAGE_SIZE – wymiary obrazu, tj. wysokość lub szerokość, są równe zero lub przekraczają maksymalne dozwolone wielkości w ramach używanego urządzenia obliczeniowego;

• CL_INVALID_IMAGE_FORMAT – format obrazu, tj. kanały kolorów bądź wartości opisujące poszczególne kolory, nie są obsługiwane przez urządzenie obliczeniowe;

• CL_MEM_OBJECT_ALLOCATION_FAILURE – nie udało się przydzielić pamięci podczas operacji na parametrach jądra opisujących obraz lub bufor;

• CL_INVALID_EVENT_WAIT_LIST – liczba zdarzeń w parametrze num_events_in_wait_list jest większa od zera, ale parametr event_wait_list jest równy NULL lub na odwrót – parametr event_wait_list jest różny od NULL, ale liczba zdarzeń została ustalona na zero, lub obiekty reprezentujące zdarzenia podane w argumencie event_wait_list nie są poprawnymi obiektami zdarzeń;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL, np. podano zbyt wiele obrazów do odczytu lub zapisu, bądź liczba samplerów użytych w jądrze obliczeniowym przekracza dopuszczalną wielkość danego urządzenia obliczeniowego;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• uruchomienie jądra obliczeniowego bez zdarzenia i tylko z globalną siatką obliczeniową:

```
cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 2, NULL,
```

```
globalWorkSize, NULL, 0, NULL, NULL );
```

• uruchomienie jądra obliczeniowego ze zdarzeniem:

```
cli_err_num = clEnqueueNDRangeKernel( cl_compute_command_queue, kernel, 2, NULL, globalWorkSize, localWorkSize, 0, NULL, &kernelEvent );
```

D.17. Funkcja clEnqueueTask

Funkcja `clEnqueueTask` jest odpowiedzialna za wykonanie jądra obliczeniowego w trybie tzw. zadania bez korzystania z siatki obliczeniowej. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clEnqueueTask ( cl_command_queue command_queue,

cl_kernel kernel,

    cl_uint num_events_in_wait_list,

    const cl_event *event_wait_list,

 cl_event *event );
```

Znaczenie poszczególnych parametrów:

• `command_queue` – parametr reprezentuje kolejkę, w której zostanie umieszczone polecenie wykonania jądra obliczeniowego;

• `kernel` – poprawny obiekt reprezentujący jądro obliczeniowe, kontekst obliczeniowy dla jądra oraz kolejki musi być taki sam;

• `event_wait_list` oraz `num_events_in_wait_list` – w parametrze `event_wait_list` można umieścić obiekty, które reprezentują zdarzenia, jakie muszą zostać zrealizowane przed wykonaniem zadania. Podanie wartości `NULL` pociąga za sobą konieczność podania zera w parametrze `num_events_in_wait_list`. Oznacza to także, iż zadanie może zostać wykonane bez oczekiwania na zakończenie innych zdarzeń. Wartość większa niż zero w parametrze `num_events_in_wait_list` jest interpretowana jako liczba zdarzeń umieszczona w `event_wait_list`. Należy zadbać o to, aby wszystkie zdarzenia były utworzenie za pomocą tego samego kontekstu obliczeniowego co wskazana kolejka poleceń.

• `event` – parametr pozwala odczytać obiekt zdarzenia oznaczający zakończenie wykonywania zleconego zadania. Podanie wartości `NULL` oznacza, że nie ma możliwości odczytania obiektu reprezentującego zakończenie zleconego zadania.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Zlecone zadanie do wykonania można przyrównać do wykonania jądra obliczeniowego za pomocą tylko jednej jednostki roboczej. Funkcja `clEnqueueTask` może być utożsamiana z wywołaniem `clEnqueueNDRangeKernel`, gdzie dostępna jest tylko jedna grupa robocza z jedną aktywną jednostką roboczą. Funkcja zwraca wartość `CL_SUCCESS`, jeśli operacja zakończyła się poprawnie.

W przeciwnym przypadku, można otrzymać jeden z następujących kodów:

• `CL_INVALID_PROGRAM_EXECUTABLE` – podany obiekt programu nie został poprawnie zbudowany dla urządzenia w danej kolejce zadań;

• `CL_INVALID_COMMAND_QUEUE` – nie podano poprawnego obiektu reprezentującego kolejkę;

• `CL_INVALID_KERNEL` – podany obiekt jądra obliczeniowego nie jest poprawnym obiektem;

• `CL_INVALID_CONTEXT` – kontekst obliczeniowy w kolejce oraz w podanym jądrze obliczeniowym nie są takie same lub kontekst obliczeniowy oraz konteksty obiektów zdarzeń umieszczone w parametrze `event_wait_list` nie są takie same;

• `CL_INVALID_KERNEL_ARGS` – argumenty jądra obliczeniowego nie zostały poprawnie określone;

• `CL_INVALID_WORK_GROUP_SIZE` – definicja grupy roboczej określonej w kodzie źródłowym za pomocą atrybutu `__attribute__ ((reqd_work_group_size(X, Y, Z)))` nie jest równa (1, 1, 1);

• `CL_MISALIGNED_SUB_BUFFER_OFFSET` – pod-bufor w obiekcie bufora nie jest poprawnie wyrównany zgodnie z wymaganiami dla używanego urządzenia obliczeniowego;

• `CL_INVALID_IMAGE_SIZE` – wymiary obrazu, tj. wysokość lub szerokość, są równe zero lub przekraczają maksymalne dozwolone wielkości w ramach używanego urządzenia obliczeniowego;

• `CL_INVALID_IMAGE_FORMAT` – format obrazu, tj. kanały kolorów bądź wartości opisujące poszczególne kolory, nie są obsługiwane przez urządzenie obliczeniowe;

• `CL_MEM_OBJECT_ALLOCATION_FAILURE` – nie udało się przydzielić pamięci podczas operacji na parametrach jądra opisujących obraz lub bufor;

• `CL_INVALID_EVENT_WAIT_LIST` – liczba zdarzeń w parametrze `num_events_in_wait_list` jest większa od zera, ale parametr `event_wait_list` jest równy `NULL`, lub na odwrót – parametr `event_wait_list` jest różny od `NULL`, ale liczba zdarzeń została ustalona na zero, bądź też obiekty reprezentujące zdarzenia podane w argumencie `event_wait_list` nie są poprawnymi obiektami zdarzeń;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• uruchomienie pojedynczego zadania OpenCL:

```
cli_err_num = clEnqueueTask(command_queue, kernel[i], 0, NULL, NULL);
```

D.18. Funkcja clGetKernelWorkGroupInfo

Funkcja `clGetKernelWorkGroupInfo` jest odpowiedzialna za odczytanie informacji o konfiguracji siatki obliczeniowej dla wskazanego jądra obliczeniowego. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clGetKernelWorkGroupInfo ( cl_kernel kernel,

cl_device_id device,

    cl_kernel_work_group_info param_name,

    size_t param_value_size,

    void *param_value,

    size_t *param_value_size_ret );
```

Znaczenie poszczególnych parametrów:

• `kernel` – obiekt reprezentujący jądro obliczeniowe;

• `device` – pozwala wskazać określone urządzenie, które jest powiązane z jądrem obliczeniowym; urządzenie musi przynależeć do tego samego kontekstu obliczeniowego co jądro. W przypadku, gdy z kontekstem powiązane jest tylko jedno urządzenie obliczeniowe, parametr `device` może przyjąć wartości `NULL`;

• `param_name` – parametr pozwala określić typ informacji, jaki ma zostać odczytany. Lista możliwych stałych została umieszczona w tab. D.7;

• `param_value_size` – w parametrze należy określić, ile pamięci (w bajtach) zostało przydzielone dla parametru `param_value`; wartość ta powinna być równa lub większa od typu danych wskazanych w tab. D.7;

• `param_value` – wskaźnik do pamięci gospodarza, w której zostaną umieszczone odczytane informacje, podanie wartości `NULL` oznacza pominięcie tego parametru;

• `param_value_size_ret` – parametr pozwala odczytać rozmiar informacji, która zostanie umieszczona w parametrze `param_value`; ilość informacji jest określana w bajtach, podanie wartości `NULL` oznacza pominięcie tego parametru;

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Funkcja zwraca wartość `CL_SUCCESS`, jeśli operacja odczytu informacji zakończyła się poprawnie. W przeciwnym przypadku, można otrzymać jeden z następujących kodów:

• `CL_INVALID_DEVICE` – niepoprawne urządzenie – wskazane w parametrze `device` urządzenie nie jest urządzeniem, dla którego utworzone zostało jądro obliczeniowe, inaczej mówiąc nie przynależy do kontekstu obliczeniowego lub podano wartość `NULL`, a kontekst obliczeniowy zawiera kilka urządzeń, dla których utworzono kilka instancji procedury obliczeniowej;

• CL_INVALID_VALUE – wartość dla param_name nie jest poprawna lub ilość przydzielonej pamięci wskazanej w parametrze param_value_size jest mniejsza niż jest to wymagane, a wartość parametru param_value jest różna od NULL, a także gdy wartością param_name jest CL_KERNEL_GLOBAL_WORK_SIZE, a jądro obliczeniowe nie zostało jeszcze zbudowane;

• CL_INVALID_KERNEL – obiekt jądra obliczeniowego nie jest poprawnym obiektem;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• odczytanie informacji o rozmiarze grupy roboczej:

```
cli_err_num = clGetKernelWorkGroupInfo(kernel, platform_devices[0], CL_KERNEL_WORK_GROUP_SIZE, sizeof(size_t), &workgroup_size, NULL);
```

Tabela D.7. Opis znaczenia parametrów param_name, oraz wielkość i typ zwróconej struktury w parametrze param_value

cl_kernel_work_group_info	Typ	Opis
CL_KERNEL_GLOBAL_WORK_SIZE	size_t[3]	Zwrócona informacja to maksymalna wielkość siatki globalnej, jaka może zostać użyta przy uruchomieniu jądra obliczeniowego. Odczytana informacja może zostać przekazana podczas wywoływania jądra obliczeniowego za pomocą clEnqueueNDRangeKernel.
CL_KERNEL_WORK_GROUP_SIZE	size_t	Zwrócona informacja to maksymalna wielkość grupy roboczej, jaka może zostać podana dla urządzenia, które jest używane do wykonania jądra obliczeniowego. Należy zwrócić uwagę, iż wielkość grupy roboczej wpływa na alokację lokalnych zasobów, np. pamięci prywatnej. Dość często zmniejszenie rozmiaru grupy roboczej może w efekcie przyspieszyć działanie procedury obliczeniowej.
CL_KERNEL_COMPILE_WORK_GROUP_SIZE	size_t[3]	Zwrócona informacja opisuje rozmiar grupy roboczej podanej w kodzie źródłowym danego jądra obliczeniowego za pomocą atrybutu: `__attribute__((reqd_work_group_size(X, Y, Z)))` Jeśli powyższy atrybut nie został zastosowany, to zwrócona zostanie informacja (0,0,0).

CL_KERNEL_LOCAL_MEM_SIZE	cl_ulong	Stała pozwala otrzymać informacje o ilości pamięci lokalnej, jaka jest zadeklarowana przez jądro obliczeniowe. Podana wielkość jest niezbędna, aby poprawnie wykonać dane jądro obliczeniowe, obejmuje wszystkie zmienne opatrzone słowem kluczowym `local`, a także ilość pamięci, jaka została zadeklarowana po stronie gospodarza przed wykonaniem jądra obliczeniowego. W przypadku, gdy jądro obliczeniowe nie używa pamięci lokalnej, za pomocą słowa kluczowego `local` bądź po stronie gospodarza nie dokonano alokacji tego typu pamięci, zwrócona zostaje wartość zero.
CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE	size_t	Stała pozwala odczytać sugerowaną przez OpenCL liczbę grup roboczych do równoległego uruchomienia. Wartość ta powinna być stosowana do osiągnięcia najwyższej wydajności. Całkowita liczba grup roboczych powinna być wielokrotnością liczby równolegle działających grup roboczych. Pozwoli to na osiągnięcie wyższej wydajności.
CL_KERNEL_PRIVATE_MEM_SIZE	cl_ulong	Stała pozwala odczytać ilość pamięci prywatnej (w bajtach) jaka jest używana przez pojedynczą jednostkę roboczą dla jądra obliczeniowego. Wartość ta odnosi się do wszystkich zmiennych prywatnych, jakie są niezbędne, aby poprawnie zrealizować obliczenia w ramach jądra obliczeniowego. Obejmuje to także wszystkie używane funkcje, również te wbudowane, które do poprawnej pracy wymagają skorzystania z pamięci prywatnej.

D.19. Funkcja clCreateImage2D

Funkcja `clCreateImage2D` jest odpowiedzialna za utworzenie obrazu dwuwymiarowego i jest obecna w standardzie 1.0 oraz 1.1. Standard 1.2 ma funkcję `clCreateImage`, tworzącą obrazy 1D, 2D oraz 3D. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_mem clCreateImage2D ( cl_context context,

cl_mem_flags flags,

    const cl_image_format *image_format,

    size_t image_width,

    size_t image_height,

    size_t image_row_pitch,

    void *host_ptr,

    cl_int *errcode_ret);
```

Znaczenie poszczególnych parametrów:

• `context` – obiekt reprezentujący kontekst obliczeniowy OpenCL;

• `flags` – parametr pozwala opisać dodatkowe własności tworzonego obiektu do reprezentacji pamięci, są one zgodne z flagami, jakie są używane dla funkcji `clCreateBuffer`;

• `image_format` – wskaźnik do zmiennej opisującej format obrazu;

• `image_width` – szerokość obrazu w pikselach, musi być większa niż zero;

• `image_height` – wysokość obrazu w pikselach, musi być większa niż zero;

• `image_row_pitch` – wielkość wiersza w bajtach musi być równa zero, jeśli parametr `host_ptr` przyjmuje wartość `NULL`. Jeśli parametr `host_ptr` przyjmuje wartość różną od `NULL`, a wartość `image_row_pitch` nadal jest równa zero, to wielkość wiersza zostanie wyznaczona przez OpenCL wg wzoru szerokość wiersza razy wielkość pojedynczego piksela;

• `host_ptr` – wskaźnik do obrazu po stronie gospodarza. Wielkość obrazu musi być taka sama lub większa niż wysokość obrazu pomnożona przez wielkość wiersza. Zakłada się, iż obraz w pamięci gospodarza jest ułożony linia po linii, a w każdej linii piksele są ułożone jeden za drugim;

• `errcode_ret` – wskaźnik do zmiennej, w której zostanie umieszczony ewentualny kod błędu, umieszczenie w nim wartości `NULL` oznacza, iż kod błędu nie będzie zwracany.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Funkcja `clCreateImage2D` zwraca poprawny obiekt reprezentujący obraz, jeśli wartością parametru `errcode_ret` jest wartość `CL_SUCCESS`. W przeciwnym przypadku otrzymuje się wartość `NULL`, a parametr `errcode_ret` wskazuje na jedną z poniższych wartości:

• `CL_INVALID_CONTEXT` – podany kontekst obliczeniowy nie jest poprawnym obiektem;

• `CL_INVALID_VALUE` – wartości podane w parametrze `flags` nie są poprawne;

• `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` – wartości opisujące format obrazu nie są poprawne lub podano wartość `NULL`;

• `CL_INVALID_IMAGE_SIZE` – wymiary obrazu, tj. wysokość lub szerokość, są równe zero lub przekraczają maksymalne dozwolone wielkości;

• `CL_INVALID_HOST_PTR` – parametr `host_ptr` jest równy `NULL` oraz użyto flag `CL_MEM_USE_HOST_PTR` bądź `CL_MEM_COPY_HOST_PTR`, bądź `host_ptr` jest różny od `NULL`, ale flagi `CL_MEM_COPY_HOST_PTR` lub `CL_MEM_USE_HOST_PTR` nie zostały użyte;

• `CL_IMAGE_FORMAT_NOT_SUPPORTED` – podany format obrazu nie jest wspierany przez urządzenie obliczeniowe;

• `CL_MEM_OBJECT_ALLOCATION_FAILURE` – nie udało się przydzielić pamięci dla obiektu

reprezentującego obraz;

• `CL_INVALID_OPERATION` – w danym kontekście nie ma urządzenia obsługującego przetwarzanie obrazu;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• utworzenie obrazu 2D tylko do zapisu:

```
output_image = clCreateImage2D(cl_compute_context, CL_MEM_WRITE_ONLY, &image_format, WIDTH, HEIGHT, 0, NULL, &cli_err_num);
```

• utworzenie obrazu 2D tylko do odczytu wraz kopiowaniem danych z pamięci gospodarza do pamięci urządzenia obliczeniowego:

```
input_image = clCreateImage2D(cl_compute_context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, &image_format, WIDTH, HEIGHT, 0, (void*)bytes, &cli_err_num);
```

D.20. Funkcja clEnqueueReadImage

Głównym zadaniem funkcji `clEnqueueReadImage` jest odczytanie obrazu (pojedynczego wskazanego obrazu lub obrazu z tzw. tablicy obrazów) z pamięci urządzenia i umieszczenie danych w pamięci gospodarza. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clEnqueueReadImage ( cl_command_queue command_queue,

cl_mem image,

    cl_bool blocking_read,

    const size_t *origin,

    const size_t *region,

    size_t row_pitch,

    size_t slice_pitch,

    void *ptr,

    cl_uint num_events_in_wait_list,

    const cl_event *event_wait_list,

 cl_event *event );
```

Znaczenie poszczególnych parametrów:

• `command_queue` – parametr reprezentuje kolejkę, w której umieszczane jest polecenie kopiujące obrazy; wymagane jest; aby kolejka oraz obrazy były utworzone w ramach tego samego kontekstu;

• `image` – parametr reprezentuje obraz bądź tablicę obrazów;

• `blocking_read` – parametr ten pozwala określić, czy operacja odczytu danych ma być operacją blokującą (`CL_TRUE`), czy nieblokującą (`CL_FALSE`). W pierwszym przypadku wywołanie funkcji `clEnqueueReadImage` oznacza, iż sterowanie do kolejnej instrukcji po stronie gospodarza nie zostanie przekazane do momentu zakończenia operacji przesyłania danych. Pozwala to na bezpieczne używanie wskaźnika `ptr` po zakończeniu przesyłu danych. W przypadku operacji nieblokowanej, po zleceniu wykonania przesyłu danych (i umieszczeniu stosownych poleceń w kolejce) sterowanie jest przekazywane do następnych instrukcji po `clEnqueueWriteImage`. Oznacza to, iż program sterujący po stronie hosta może realizować kolejne zadania, jednak wskaźnik `ptr` nie może zostać użyty do czasu zakończenia odczytu danych. Aby wykryć, czy proces odczytu został zakończony, należy skorzystać z mechanizmu zdarzeń;

• `origin` – parametr określa tzw. przesunięcie w przestrzeni obrazu z dokładnością do piksela. Wartość `(0, 0, 0)` oznacza lewy górny róg obrazu. Parametr ten jest zapisywany w postaci tablicy o trzech elementach. Ogólnie są to trzy liczby `(x, y, z)`, można także je zapisywać jako `origin[0]`, `origin[1]`, `origin[2]`. W przypadku obrazu 3D używane są wszystkie trzy parametry, a w przypadku 2D używane są parametry `(x, y)`, a trzeci parametr `z` (albo `origin[2]`) musi być ustawiony na zero. Dla przypadku 1D stosuje się pierwszy parametr, a dwa pozostałe muszą być ustawione na zero. Opisana interpretacja obowiązuje, gdy mamy do czynienia z pojedynczym obrazem. W przypadku, gdy używana jest tablica obrazów interpretacja jest nieco inna. Jeśli mamy do czynienia z tablicą obrazów 2D, trzeci parametr `origin[2]` wskazuje na numer obrazu w tablicy. W przypadku obrazów 1D numer obrazu w tablicy jest reprezentowany przez parametr `y` albo `origin[1]`;

• `region` – parametr określa tzw. prostokąt wg nomenklatury OpenCL, dla obrazów 1D, 2D, 3D – dane są zapisywane w taki sam sposób, jak `origin`, choć tym razem wartości parametru są interpretowane jako `(width, height, depth)`, tj, szerokość, wysokość oraz głębokość. Jeśli zapisujemy dane z pojedynczego obrazu 1D, to wartości `region[1]` oraz `region[2]` muszą być ustawione na jedność, podobnie w trybie 2D, wartością `region[2]` jest jedynka. W trybie tablicy obrazów dla przypadku 2D trzeci parametr oznacza numer obrazu, a dla tablicy 1D parametr `region[1]` oznacza numer obrazu w tablicy, a `region[2]` musi być ustawiony na jedynkę;

• `input_row_pitch` – parametr oznacza wielkość pojedynczego wiersza w bajtach, jeśli podano wartość zero, to na podstawie danych podanych podczas tworzenia obrazu za pomocą `clCreateImage2D`, wielkość wiersza zostanie wyznaczona samodzielnie;

• `slice_pitch` – wielkość w bajtach pojedynczego obszaru 2D w przypadku obrazu 3D lub pojedynczego obrazu w przypadku, gdy mamy tablicę obrazów 1D bądź 2D. W przypadku, gdy podajemy pojedynczy obraz, wartość tego parametru powinna być ustawiona na zero. Wartość parametru powinna odpowiadać wysokości obrazu pomnożonej przez długość wiersza. Wstawienie wartości zero oznacza, że API OpenCL samodzielnie obliczy wartość tego parametru. W przypadku gdy podana zostanie wielkość większa od zera, oznacza to dodatkowe wyrównanie, którego

znaczenie zostało opisane w funkcji clEnqueueWriteImage;

• ptr – wskaźnik do obszaru danych po stronie gospodarza, pod tym adresem zostaną umieszczone dane odczytywane z pamięci urządzenia obliczeniowego;

• event_wait_list oraz num_events_in_wait_list – w parametrze event_wait_list można umieścić obiekty, które reprezentują zdarzenia, jakie muszą zostać zrealizowane przed rozpoczęciem procesu odczytu danych. Wartość NULL oznacza, iż proces odczytu danych może zostać rozpoczęty natychmiast – pociąga to za sobą konieczność podania zera w parametrze num_events_in_wait_list. Wartość większa niż zero jest interpretowana jako liczba zdarzeń umieszczona w event_wait_list. Należy zadbać o to, aby wszystkie zdarzenia były utworzone za pomocą tego samego kontekstu obliczeniowego co wskazana kolejka poleceń;

• event – parametr pozwala odczytać obiekt zdarzenia oznaczający zakończenie procesu przenoszenia danych. Podanie wartości NULL oznacza, że nie będziemy odczytywać obiektu reprezentującego zdarzenie zakończenia kopiowania danych. Należy zwrócić uwagę, iż jeśli parametry event_wait_list oraz num_events_in_wait_list nie są równe NULL, to obiekt event nie może być elementem, który w jakiś sposób odnosi się do zdarzeń umieszczonych w event_wait_list.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

W przypadku, gdy stosujemy clEnqueueReadImage dla pojedynczych obrazów, nie ma istotnych uwag co do wskaźnika ptr po stronie gospodarza. Jednak, jeśli parametr slice_pitch określa dodatkowy odstęp, należy zadbać o to, aby ten parametr był zgodny z obrazem, z którego czerpane są informacje. Szczególnie, jeśli kopiowany jest tylko fragment bądź pojedynczy obraz w tablicy obrazów. Nie można także zmieniać wartości wskaźnika ptr podczas kopiowania danych, szczególnie jeśli mamy do czynienia operacją kopiowania nieblokowanego.

Funkcja clEnqueueReadImage zwraca wartość CL_SUCCESS, jeśli operacja kopiowania danych zakończyła się poprawnie. W przeciwnym przypadku można otrzymać jeden z następujących kodów:

• CL_INVALID_COMMAND_QUEUE – nie podano poprawnego obiektu reprezentującego kolejkę;

• CL_INVALID_CONTEXT – kontekst obliczeniowy kolejki poleceń oraz konteksty obiektów zdarzeń umieszczone w parametrze event_wait_list nie są takie same;

• CL_INVALID_MEM_OBJECT – podany obiekt obrazu nie jest poprawnym obiektem;

• CL_INVALID_VALUE – region, do którego zapisywane są dane, jest poza zakresem obszaru wskazanego pierwotnie przez origin i region lub wskaźnik ptr jest równy NULL, lub wartości parametrów origin i region nie będą poprawnie określone, np. brakujące jedynki w przypadku obrazów 1D czy 2D;

• CL_INVALID_EVENT_WAIT_LIST – liczba zdarzeń w parametrze num_events_in_wait_list jest większa od zera, ale parametr event_wait_list jest równy NULL, lub na odwrót – parametr event_wait_list jest różny od NULL, ale liczba zdarzeń została ustalona na zero, bądź też obiekty reprezentujące zdarzenia podane w argumencie event_wait_list nie są poprawnymi obiektami

zdarzeń;

• CL_INVALID_IMAGE_SIZE – podane wymiary obrazów, a także wielkości wierszy, bądź też pojedynczy obraz 2D, nie są obsługiwane przez dane urządzenie obliczeniowe, do którego podłączona została kolejka poleceń;

• CL_INVALID_IMAGE_FORMAT – format obrazu, tj. kanały kolorów bądź wartości opisujące poszczególne kolory, nie są obsługiwane przez urządzenie obliczeniowe;

• CL_MEM_OBJECT_ALLOCATION_FAILURE – nie udało się przydzielić pamięci podczas zapisywania danych obrazu;

• CL_INVALID_OPERATION – funkcja clEnqueueWriteImage została użyta do obrazu (parametr image), gdzie zastosowano flagi CL_MEM_HOST_WRITE_ONLY (pamięć tylko do zapisu) lub CL_MEM_HOST_NO_ACCESS (brak dostępu do obszaru pamięci);

• CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST – operacja zapisu jest operacją blokującą, a status zdarzeń podanych event_wait_list jest wartością ujemną, tj. wystąpił błąd w przynajmniej jednym zdarzeniu;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• odczyt danych o obrazie z urządzenia obliczeniowego:

```
cli_err_num = clEnqueueReadImage(cl_compute_command_queue, output_image, CL_TRUE, origin, region, 0, 0, (void*)out_bytes, 0, NULL, NULL);
```

D.21. Funkcja clEnqueueWriteImage

Głównym zadaniem funkcji clEnqueueWriteImage jest zapisanie obrazu (pojedynczego wskazanego obrazu lub obrazu z tzw. tablicy obrazów) do pamięci urządzenia, przy czym dane reprezentujące obraz są pobierane z pamięci gospodarza. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clEnqueueWriteImage( cl_command_queue command_queue,

cl_mem image,

    cl_bool blocking_write,

    const size_t *origin,

    const size_t *region,
```

```
    size_t input_row_pitch,

    size_t input_slice_pitch,

    const void *ptr,

    cl_uint num_events_in_wait_list,

    const cl_event *event_wait_list,

cl_event *event);
```

Znaczenie poszczególnych parametrów:

• `command_queue` – parametr reprezentuje kolejkę, w której umieszczane jest polecenie kopiujące obrazy; wymagane jest, aby kolejka oraz obrazy były utworzone w ramach tego samego kontekstu;

• `image` – parametr reprezentuje obraz bądź tablicę obrazów;

• `blocking_write` – parametr ten pozwala określić, czy operacja kopiowania danych ma być operacją blokującą (`CL_TRUE`), czy nieblokującą (`CL_FALSE`). W pierwszym przypadku, wywołanie funkcji `clEnqueueWriteImage` oznacza, iż sterowanie do kolejnej instrukcji po stronie gospodarza nie zostanie przekazane do momentu zakończenia operacji przesyłania danych. Pozwala to na bezpieczne używanie wskaźnika `ptr` po zakończeniu przesyłu danych. W przypadku operacji nieblokowanej, po zleceniu wykonania przesyłu danych (i umieszczeniu odpowiednich poleceń w kolejce) sterowanie jest przekazywane do następnych instrukcji po `clEnqueueWriteImage`. Oznacza to, iż program sterujący po stronie hosta może realizować kolejne zadania, jednak wskaźnik `ptr` nie może zostać użyty do czasu zakończenia kopiowania. Aby wykryć, czy proces kopiowania został zakończony, należy skorzystać z mechanizmu zdarzeń;

• `origin` – parametr określa tzw. przesunięcie w przestrzeni obrazu z dokładnością do piksela. Wartość `(0, 0, 0)` oznacza lewy górny róg obrazu. Parametr ten jest zapisywany w postaci tablicy o trzech elementach. Ogólnie są to trzy liczby `(x, y, z)`, można także je zapisywać jako `origin[0], origin[1], origin[2]`. W przypadku obrazu 3D, używane są wszystkie trzy parametry, natomiast w przypadku 2D, używane są parametry `(x, y)`, a trzeci parametr `z` (albo `origin[2]`) musi być ustawiony na zero. Dla przypadku 1D stosuje się pierwszy parametr, a dwa pozostałe muszą być ustawione na zero. Opisana interpretacja obowiązuje, gdy mamy do czynienia z pojedynczym obrazem. W przypadku, gdy stosowna jest tablica obrazów, interpretacja jest nieco inna. Jeśli mamy do czynienia z tablicą obrazów 2D, trzeci parametr `origin[2]` wskazuje na numer obrazu w tablicy. W przypadku obrazów 1D, numer obrazu w tablicy jest reprezentowany przez parametr `y` albo `origin[1]`;

• `region` – parametr określa tzw. prostokąt wg nomenklatury OpenCL, dla obrazów 1D, 2D, 3D – dane są zapisywane w taki sam sposób jak `origin`, choć tym razem wartości parametru są interpretowane jako `(width, height, depth)`, tj. szerokość, wysokość oraz głębokość. Jeśli zapisujemy dane z pojedynczego obrazu 1D, to wartości `region[1]` oraz `region[2]` muszą być ustawione na jedność, podobnie w trybie 2D wartością `region[2]` jest jedynka. W trybie tablicy obrazów, dla przypadku 2D trzeci parametr oznacza numer obrazu, a dla tablicy 1D parametr `region[1]` oznacza numer obrazu w tablicy, a `region[2]` musi być ustawiony na jedynkę;

• `input_row_pitch` – parametr oznacza wielkość wiersza w bajtach, jeśli podano wartość zero, to na podstawie danych podanych podczas tworzenia obiektu obrazu za pomocą `clCreateImage2D`, wielkość wiersza zostanie wyznaczona samodzielnie;

• `input_slice_pitch` – wielkość w bajtach pojedynczego obszaru 2D w przypadku obrazu 3D lub pojedynczego obrazu w przypadku, gdy mamy tablicę obrazów 1D bądź 2D. Jeżeli podajemy pojedynczy obraz, wartość tego parametru powinna być ustawiona na zero. Wartość parametru powinna odpowiadać wysokości obrazu pomnożonej przez długość wiersza, przy czym może być większa, co oznacza, że poszczególne fragmenty obrazu mają dodatkowe wyrównanie (lub są przedzielone dodatkowym obszarem). Wstawienie wartości zero oznacza, że API OpenCL samodzielnie obliczy wartość tego parametru, co oznacza brak wyrównania (bądź dodatkowego odstępu) pomiędzy poszczególnymi obrazami w tablicy lub regionami w przypadku obrazu 3D;

• `ptr` – wskaźnik do obszaru danych po stronie gospodarza, dane odczytane z pamięci gospodarza zostaną umieszczane w pamięci urządzenia obliczeniowego;

• `event_wait_list` oraz `num_events_in_wait_list` – w parametrze `event_wait_list` można umieścić obiekty reprezentujące zdarzenia, które muszą zostać zrealizowane przed rozpoczęciem procesu kopiowania danych. Wartość `NULL` oznacza, iż proces kopiowania danych może zostać rozpoczęty natychmiast – pociąga to za sobą konieczność podania zera w parametrze `num_events_in_wait_list`. Wartość większa niż zero jest interpretowana jako liczba zdarzeń umieszczona w `event_wait_list`. Należy zadbać o to, aby wszystkie zdarzenia były utworzenie za pomocą tego samego kontekstu obliczeniowego co wskazana kolejka poleceń;

• `event` – parametr pozwalający odczytać obiekt zdarzenia, który oznacza proces kopiowania danych; obiekt ten można wykorzystać do sprawdzenia, czy proces kopiowania został zakończony. Podanie wartości `NULL` oznacza, że nie będziemy odczytywać obiektu reprezentującego zdarzenie zakończenia kopiowania danych.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

W przypadku, gdy stosujemy `clEnqueueWriteImage` dla pojedynczych obrazów, nie ma istotnych uwag co do wskaźnika `ptr` po stronie gospodarza. Jednak, jeśli parametr `input_slice_pitch` określa dodatkowy odstęp, należy zadbać o poprawne przydzielenie obszaru pamięci po stronie gospodarza. Szczególnie, jeśli kopiowany jest tylko fragment lub pojedynczy obraz w tablicy obrazów. Nie można także zmieniać wartości wskaźnika `ptr` podczas kopiowania danych, szczególnie jeśli mamy do czynienia z operacją kopiowania nieblokowanego.

Otrzymanie wartości `CL_SUCCESS` oznacza, iż funkcja zakończyła poprawnie wykonywane czynności. W innym przypadku można otrzymać jeden z następujących kodów:

• `CL_INVALID_COMMAND_QUEUE` – nie podano poprawnego obiektu reprezentującego kolejkę;

• `CL_INVALID_CONTEXT` – kontekst obliczeniowy kolejki poleceń oraz konteksty obiektów zdarzeń umieszczone w parametrze `event_wait_list` nie są takie same;

• `CL_INVALID_MEM_OBJECT` – podany obiekt reprezentujący obraz po stronie urządzenia obliczeniowego nie jest poprawny;

• CL_INVALID_VALUE – kod się pojawia w przypadku, gdy region, do którego zapisywane są dane jest poza zakresem obszaru wskazanego pierwotnie przez origin i region, lub gdy wskaźnik ptr jest równy NULL, lub wartości parametrów origin i region nie są poprawnie określone, np. brakujące jedynki w przypadku obrazów 1D czy 2D;

• CL_INVALID_EVENT_WAIT_LIST – liczba zdarzeń w parametrze num_events_in_wait_list jest większa od zera, ale parametr event_wait_list jest równy NULL, lub na odwrót – parametr event_wait_list jest różny od NULL, ale liczba zdarzeń została ustalona na zero, bądź też obiekty reprezentujące zdarzenia podane w argumencie event_wait_list nie są poprawnymi obiektami zdarzeń;

• CL_INVALID_IMAGE_SIZE – podane wymiary obrazów, a także wielkości wierszy bądź też pojedynczy obraz 2D, nie są obsługiwane przez dane urządzenie obliczeniowe, do którego podłączona została kolejka poleceń;

• CL_INVALID_IMAGE_FORMAT – format obrazu, tj. kanały kolorów bądź wartości opisujące poszczególne kolory, nie są obsługiwane przez urządzenie obliczeniowe;

• CL_MEM_OBJECT_ALLOCATION_FAILURE – nie udało się przydzielić pamięci podczas zapisywania danych obrazu;

• CL_INVALID_OPERATION – określone urządzenie nie obsługuje przetwarzania obrazów;

• CL_INVALID_OPERATION – funkcja clEnqueueWriteImage została użyta do obrazu (parametr image), gdzie zastosowano flagi CL_MEM_HOST_WRITE_ONLY (pamięć tylko do zapisu) lub CL_MEM_HOST_NO_ACCESS (brak dostępu do obszaru pamięci);

• CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST – operacja zapisu jest operacją blokującą, a status zdarzeń podanych event_wait_list jest wartością ujemną, tj. wystąpił błąd w przynajmniej jednym zdarzeniu;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• zapisanie danych obrazu do urządzenia obliczeniowego:

```
cli_err_num = clEnqueueWriteImage( cl_compute_command_queue, input_image, CL_TRUE, origin, region, 0, 0, host_image, 0, NULL, NULL);
```

D.22. Funkcja clFinish

Głównym zadaniem funkcji clFinish jest wprowadzenie swoistej blokady (bądź też synchronizacji), tzn. funkcja clFinish nie zakończy swojego działania tak długo, jak długo w podanej kolejce będą znajdować się zadania do wykonania. Dopiero po zrealizowaniu wszystkich zadań, funkcja clFinish

pozwoli, aby sterowanie w programie po stronie gospodarza zostało przekazane do następnych poleceń występujących po `clFinish`. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clFinish ( cl_command_queue command_queue );
```

Znaczenie poszczególnych parametrów:

• `command_queue` – obiekt reprezentuje kolejkę poleceń.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Otrzymanie wartości `CL_SUCCESS` oznacza, iż funkcja zakończyła poprawnie swoje działania. W innym przypadku otrzymać można jeden z trzech następujących kodów:

• `CL_INVALID_COMMAND_QUEUE` – parametr `command_queue` nie wskazuje poprawnego obiektu kolejki;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• oczekiwanie na zakończenie realizacji zadań w podanej kolejce poleceń:

```
clFinish(cl_compute_command_queue);
```

D.23. Funkcja clReleaseKernel

Funkcja `clReleaseKernel` jest odpowiedzialna za zmniejszenie licznika referencji dla wskazanego obiektu reprezentującego jądro obliczeniowe dla urządzenia obliczeniowego. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clReleaseKernel ( cl_kernel kernel );
```

Znaczenie poszczególnych parametrów:

• `kernel` – obiekt reprezentuje jądro obliczeniowe przeznaczone do usunięcia.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Usunięcie obiektu jądra obliczeniowego następuje w przypadku, gdy licznik referencji jest równy zero oraz gdy wszystkie jądra obliczeniowe zakończą realizację własnych obliczeń.

Jeśli wartością końcową funkcji jest `CL_SUCCESS`, to wykonanie funkcji zakończyło się sukcesem.

W innym przypadku można otrzymać trzy możliwe kody błędów:

• `CL_INVALID_KERNEL` – podany obiekt nie reprezentuje poprawnego jądra obliczeniowego;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• zwolnienie zasobów jądra obliczeniowego:

```
clReleaseKernel(kernel);
```

D.24. Funkcja clReleaseProgram

Funkcja `clReleaseProgram` jest odpowiedzialna za zmniejszenie licznika referencji dla wskazanego obiektu reprezentującego program dla urządzenia obliczeniowego. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clReleaseProgram ( cl_program program );
```

Znaczenie poszczególnych parametrów:

• `program` – obiekt reprezentuje program przeznaczony do usunięcia.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Usunięcie programu następuje w przypadku, gdy licznik referencji jest równy zero oraz gdy wszystkie jądra obliczeniowe korzystające z obiektu programu zostaną usunięte.

Jeśli wartością końcową funkcji jest `CL_SUCCESS`, to wykonanie funkcji zakończyło się sukcesem. Jeśli jest inaczej, można otrzymać jeden z trzech kodów oznaczających błąd:

• `CL_INVALID_PROGRAM` – obiekt programu nie jest poprawnym obiektem reprezentującym program;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• zwolnienie obiektu reprezentującego program:

```
clReleaseProgram(cl_kernel_program);
```

D.25. Funkcja clReleaseCommandQueue

Funkcja `clReleaseCommandQueue` jest odpowiedzialna za zmniejszenie licznika referencji dla wskazanego obiektu reprezentującego kolejkę poleceń. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clReleaseCommandQueue( cl_command_queue command_queue );
```

Znaczenie poszczególnych parametrów:

• `command_queue` – obiekt reprezentuje kolejkę poleceń przeznaczoną do usunięcia.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Usunięcie kolejki następuje w przypadku, gdy licznik referencji jest równy zero oraz gdy wszystkie aktywne polecenia znajdujące się w kolejce zrealizują swoje zadania (np. operacje kopiowania pamięci pomiędzy urządzeniem obliczeniowym a pamięcią gospodarza, bądź zlecenie wykonania procedur obliczeniowych). Dodatkowo, wywołanie funkcji `clReleaseCommandQueue` spowoduje usunięcie oczekujących zadań.

Jeśli wartością końcową funkcji jest `CL_SUCCESS`, to wykonanie funkcji zakończyło się sukcesem. Jeśli jest inaczej, można otrzymać jeden z trzech kodów oznaczających następujące błędy:

• `CL_INVALID_COMMAND_QUEUE` – parametr `command_queue` nie wskazuje poprawnego obiektu kolejki;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• zwolnienie kolejki poleceń:

```
clReleaseCommandQueue(cl_compute_command_queue);
```

D.26. Funkcja clReleaseMemObject

Funkcja `clReleaseContext` jest odpowiedzialna za zmniejszenie licznika referencji dla wskazanego obiektu reprezentującego obszar umieszczony w pamięci urządzenia OpenCL (obiekt pamięciowy). **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clReleaseMemObject ( cl_mem memobj );
```

Znaczenie poszczególnych parametrów:

• `memobj` – obiekt reprezentujący wskaźnik do pamięci urządzenia.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Usunięcie obiektu pamięciowego następuje w przypadku, gdy licznik referencji jest równy zero oraz dodatkowo, jeśli polecenia umieszczone w kolejce odnoszące się do tego obiektu zakończyły wszystkie operacje. Jeśli obiekt pamięciowy jest buforem i zawiera tzw. pod-bufory, nie zostanie usunięty, dopóki wszystkie pod-bufory również nie zostaną usunięte.

Jeśli wartością końcową funkcji jest `CL_SUCCESS`, to jej wykonanie zakończyło się sukcesem. Jeśli jest inaczej, można otrzymać jeden z trzech kodów oznaczających błąd:

• `CL_INVALID_MEM_OBJECT` – parametr `memobj` nie wskazuje poprawnego obiektu pamięciowego;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• usunięcie obiektu pamięciowego z pamięci globalnej urządzenia obliczeniowego:

```
clReleaseMemObject(input_image);
```

D.27. Funkcja clReleaseContext

Funkcja `clReleaseContext` jest odpowiedzialna za zmniejszenie licznika referencji dla wskazanego kontekstu. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clReleaseContext ( cl_context context );
```

Znaczenie poszczególnych parametrów:

• `context` – obiekt reprezentujący kontekst obliczeniowy.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Zmniejszenie wartości licznika referencji nie powoduje zwolnienia przydzielonej pamięci. Stanie się to dopiero, gdy licznik referencji osiągnie wartość zero. W takim przypadku inne obiekty podłączone do kontekstu (np. obiekty reprezentujące wskaźniki do pamięci urządzenia, kolejka) zostaną zwolnione, a sam kontekst usunięty.

Jeśli wynikiem funkcji jest wartość `CL_SUCCESS`, to jej wykonanie zakończyło się sukcesem. W przeciwnym przypadku można otrzymać kod jednego z trzech potencjalnych błędów:

• `CL_INVALID_CONTEXT` – podany kontekst nie jest poprawnym kontekstem OpenCL;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• zwolnienie obiektu reprezentującego kontekst obliczeniowy:

```
clReleaseContext(cl_compute_context);
```

D.28. Funkcja clCreateFromGLBuffer

Funkcja `clCreateFromGLBuffer` tworzy bufor OpenCL (w pamięci urządzenia obliczeniowego) na podstawie podanego bufora OpenGL. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_mem clCreateFromGLBuffer ( cl_context context,

cl_mem_flags flags,

    GLuint bufobj,

 cl_int * errcode_ret);
```

Znaczenie poszczególnych parametrów:

• `context` – obiekt reprezentujący kontekst obliczeniowy dostosowany do współpracy z OpenGL;

• `flags` – parametr reprezentuje pole bitowe, gdzie można podać dodatkowe atrybuty opisujące możliwość dostępu do danych umieszczonych w pamięci urządzenia obliczeniowego, w przypadku tej funkcji można określić, iż dany bufor jest przeznaczony tylko do zapisu, tylko odczytu lub do odczytu i zapisu (co można oznaczyć za pomocą flag: `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` oraz `CL_MEM_READ_WRITE`);

• `bufobj` – identyfikator bufora utworzonego za pomocą funkcji OpenGL; bufor OpenGL musi zostać utworzony, ale nie musi zostać zainicjowany;

• `errcode_ret` – parametr zwraca ewentualny kod błędu, podanie wartość `NULL` oznacza, iż kod nie zostanie zwrócony.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Należy zwrócić uwagę na sytuację, kiedy po wywołaniu funkcji `clCreateFromGLBuffer`, wielkość oryginalnego bufora zostanie zmieniona przez API OpenGL. W takim przypadku korzystanie z obiektu bufora po stronie OpenCL może doprowadzić do nieprzewidzianego zachowania, a w rezultacie do przerwania działania aplikacji.

Funkcja `clCreateFromGLBuffer` zwraca poprawny obiekt pamięciowy w przypadku, gdy utworzenie bufora zakończyło się sukcesem, oznacza to również, iż wartość `CL_SUCCESS` zostaje umieszczona w parametrze `errcode_ret`. W przeciwnym wypadku zwracana jest wartość `NULL`, a w parametrze `errcode_ret` wstawiany jest jeden z następujących kodów:

• `CL_INVALID_CONTEXT` – podany obiekt kontekstu nie jest poprawny lub też nie został utworzony jako kontekst współpracujący z OpenGL;

• `CL_INVALID_VALUE` – podana wartość lub wartości w parametrze `flags` nie są poprawne;

• `CL_INVALID_GL_OBJECT` – parametr `bufobj` nie jest obiektem wskazującym na bufor OpenGL lub też wielkość bufora nie jest poprawna, np. jest równa zero;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• utworzenie obiektu, który będzie reprezentował tablicę wierzchołków:

```
cl_vbo_mem = clCreateFromGLBuffer(context, CL_MEM_READ_WRITE, vbo, &cli_err_num );
```

D.29. Funkcja clCreateFromGLTexture2D

Funkcja `clCreateFromGLTexture2D` tworzy obraz 2D zgodny z OpenCL na podstawie podanego obiektu tekstury utworzonego w ramach OpenGL 2D. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_mem clCreateFromGLTexture2D ( cl_context context,

cl_mem_flags flags,

    GLenum texture_target,

    GLint miplevel,

    GLuint texture,

    cl_int *errcode_ret);
```

Znaczenie poszczególnych parametrów:

• `context` – obiekt reprezentujący kontekst obliczeniowy dostosowany do współpracy z OpenGL,

• `flags` – parametr reprezentuje pole bitowe, gdzie można podać dodatkowe atrybuty opisujące możliwość dostępu do danych umieszczonych w pamięci urządzenia obliczeniowego. W przypadku tej funkcji można określić, iż dany bufor jest przeznaczony tylko do zapisu, tylko odczytu lub do odczytu i zapisu (co można oznaczyć za pomocą flag: `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` oraz

CL_MEM_READ_WRITE);

• texture_target – dopuszczalne wartości to: GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, GL_TEXTURE_RECTANGLE. Parametr texture_target jest stosowany do określania typu tekstury;

• miplevel – poziom tzw. mipmapingu;

• texture – obiekt reprezentujący teksturę po stronie OpenGL;

• errcode_ret – parametr zwraca ewentualny kod błędu; podanie wartość NULL oznacza, iż kod nie zostanie zwrócony.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Funkcja clCreateFromGLBuffer zwraca poprawny obiekt pamięciowy reprezentujący obraz, co oznacza, iż wartość CL_SUCCESS zostaje umieszczona w parametrze errcode_ret. W przeciwnym wypadku, zwracana jest wartość NULL, a w parametrze errcode_ret wstawiany jest jeden z następujących kodów:

• CL_INVALID_CONTEXT – podany obiekt kontekstu nie jest poprawny lub też nie został utworzony jako kontekst współpracujący z OpenGL;

• CL_INVALID_VALUE – podana wartość lub wartości w parametrze flags bądź w texture_target nie są poprawne;

• CL_INVALID_MIP_LEVEL – wartość miplevel jest większa niż zero oraz sterownik OpenGL nie obsługuje tworzenia obrazu z niezerowego obiektu miplevel;

• CL_INVALID_GL_OBJECT – podana tekstura w parametrze texture nie jest zgodna z typem tekstury texture_target bądź poziom miplevel dla tekstury nie został poprawnie określony, np. rozmiary tekstury, tj. szerokość, wysokość dla określonego poziomu miplevel są równe zero;

• CL_INVALID_IMAGE_FORMAT_DESCRIPTOR – format tekstury OpenGL nie jest obsługiwany przez urządzenie obliczeniowe OpenCL;

• CL_INVALID_OPERATION – obiekt tekstury po stronie OpenGL został utworzony z ramką o szerokości większej niż zero;

• CL_OUT_OF_RESOURCES – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• CL_OUT_OF_HOST_MEMORY – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• utworzenie obiektu, który będzie reprezentował teksturę 2D:

```
cl_tex_mem = clCreateFromGLTexture2D(context, CL_MEM_READ_WRITE, GL_TEXTURE_RECTANGLE_ARB, 0, texture, &cli_err_num );
```

D.30. Funkcja clEnqueueAcquireGLObjects

Funkcja pozwala na uzyskanie dostępu do obiektu/obiektów pamięciowych OpenCL, które pierwotnie zostały utworzone z obiektów OpenGL. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clEnqueueAcquireGLObjects ( cl_command_queue command_queue,

cl_uint num_objects,

const cl_mem *mem_objects,

cl_uint num_events_in_wait_list,

const cl_event *event_wait_list,

cl_event *event);
```

Znaczenie poszczególnych parametrów:

• `command_queue` – obiekt reprezentujący poprawną kolejkę, która obsługuje współpracę pomiędzy OpenGL oraz OpenCL,

• `num_objects` – liczba obiektów pamięciowych, jakie zostały umieszczone w parametrze `mem_objects`,

• `mem_objects` – wskaźnik do obiektów pamięciowych odnoszących się do obiektów OpenGL,

• `event_wait_list` oraz `num_events_in_wait_list` – w parametrze `event_wait_list` można umieścić obiekty, które reprezentują zdarzenia, które muszą zostać zrealizowane przed realizacją pozyskania obiektu. Wartość `NULL` oznacza, iż proces pozyskania obiektu może zostać rozpoczęty natychmiast, pociąga to za sobą konieczność podania zera w parametrze `num_events_in_wait_list`. Wartość większa niż zero jest interpretowana jako liczba zdarzeń umieszczona w `event_wait_list`. Należy zadbać o to, aby wszystkie zdarzenia były utworzenie za pomocą tego samego kontekstu obliczeniowego co wskazana kolejka poleceń.

• `event` – parametr pozwala odczytać obiekt zdarzenia oznaczający zakończenie operacji realizowanej przez tę funkcję. Podanie wartości `NULL` oznacza, że nie będziemy odczytywać obiektu zdarzenia.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Należy podkreślić, iż po utworzeniu obiektów OpenCL na podstawie podanych obiektów OpenGL, nie są one jeszcze gotowe do użycia po stronie OpenCL. Należy za pomocą funkcji `clEnqueueAcquireGLObjects` na danym obiekcie wykonać dodatkowe czynności, aby dany obiekt stał się zdatny do użycia po stronie API OpenCL. Wymaga to upewnienia się, iż operacje OpenGL na danym obiekcie zostały ukończone (zazwyczaj zastosowanie `glFlush` może okazać się

wystarczające). Wykonywanie operacji na obiektach OpenCL, które są modyfikowane przez OpenGL, może spowodować trudne do wykrycia błędy i niestabilność pracy sterowników OpenGL, a także OpenCL, zwłaszcza jeśli operacje OpenGL oraz OpenCL są wykonywane w różnych wątkach.

Otrzymanie wartości `CL_SUCCESS` naturalnie oznacza, iż funkcja zakończyła poprawnie swoje działania. W innym przypadku otrzymać można jeden z następujących kodów:

• `CL_INVALID_VALUE` – wartość parametru `num_objects` wynosi zero, ale wartość `mem_objects` jest różna od `NULL`, bądź `mem_objects` jest równe `NULL`, ale `num_objects` jest większa niż zero;

• `CL_INVALID_MEM_OBJECT` – obiekty pamięciowe podane w parametrze `mem_objects` nie są poprawnymi obiektami;

• `CL_INVALID_COMMAND_QUEUE` – nie podano poprawnego obiektu reprezentującego kolejkę;

• `CL_INVALID_CONTEXT` – podany obiekt kontekstu nie jest poprawny lub też nie został utworzony jako kontekst współpracujący z OpenGL;

• `CL_INVALID_GL_OBJECT` – obiekty pamięciowe podane w `mem_objects` nie zostały utworzone na podstawie obiektów OpenGL;

• `CL_INVALID_EVENT_WAIT_LIST` – liczba zdarzeń w parametrze `num_events_in_wait_list` jest większa od zera, ale parametr `event_wait_list` jest równy `NULL` lub na odwrót – parametr `event_wait_list` jest różny od `NULL`, ale liczba zdarzeń została ustalona na zero, bądź też obiekty reprezentujące zdarzenia podane w argumencie `event_wait_list` nie są poprawnymi obiektami zdarzeń;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• uzyskanie dostępu do obiektu reprezentującego dane o wierzchołkach:

```
cli_err_num = clEnqueueAcquireGLObjects(commandQueue, 1, &cl_vbo_mem, 0, NULL, NULL );
```

D.31. Funkcja clEnqueueReleaseGLObjects

Funkcja `clEnqueueReleaseGLObjects` zwalnia obiekty pamięciowe OpenCL, jakie zostały utworzone na podstawie wskazanych obiektów OpenGL. **Ogólna postać nagłówka funkcji** jest następująca:

```
cl_int clEnqueueReleaseGLObjects ( cl_command_queue command_queue ,

cl_uint num_objects ,

    const cl_mem *mem_objects ,
```

```
    cl_uint num_events_in_wait_list ,

    const cl_event *event_wait_list ,

cl_event *event );
```

Znaczenie poszczególnych parametrów:

• `command_queue` – obiekt reprezentujący poprawną kolejkę, która obsługuje współpracę pomiędzy OpenGL oraz OpenCL;

• `num_objects` – liczba obiektów pamięciowych, które zostały umieszczone w parametrze `mem_objects`;

• `mem_objects` – wskaźnik do obiektów pamięciowych odnoszących się do obiektów OpenGL;

• `event_wait_list` oraz `num_events_in_wait_list` – w parametrze `event_wait_list` można umieścić obiekty, które reprezentują zdarzenia, jakie muszą zostać zrealizowane przed realizacją zwolnienia obiektu/obiektów. Wartość `NULL` oznacza, iż proces pozyskania obiektu może zostać rozpoczęty natychmiast, co pociąga za sobą konieczność podania zera w parametrze `num_events_in_wait_list`. Wartość większa niż zero jest interpretowana jako liczba zdarzeń umieszczona w `event_wait_list`. Należy zadbać o to, aby wszystkie zdarzenia były utworzone za pomocą tego samego kontekstu obliczeniowego co wskazana kolejka poleceń;

• `event` – parametr pozwala odczytać obiekt zdarzenia oznaczający zakończenie operacji realizowanej przez tę funkcję. Podanie wartości `NULL` oznacza, że nie będziemy odczytywać obiektu zdarzenia.

Komentarz oraz informacje o wartości zwracanej jako rezultat funkcji

Zwolnienie obiektów OpenCL jest niezbędne, jeśli chcemy usunąć obiekt OpenGL lub wykonać operację na danym obiekcie OpenGL, który był w jakikolwiek sposób przetwarzany przez OpenCL. Wymaga to naturalnie upewnienia się, iż operacje OpenCL na danym obiekcie zostały ukończone (zazwyczaj zastosowanie `clFinish` na danej kolejce, na której zlecano operacje na danym buforze może okazać się wystarczające). Wykonywanie operacji na obiektach OpenGL, które są modyfikowane przez OpenCL, przed ich zwolnieniem może spowodować trudne do wykrycia błędy i niestabilność pracy sterowników OpenGL, a także OpenCL, zwłaszcza, gdy operacje OpenCL oraz OpenGL są wykonywane w różnych wątkach.

Otrzymanie wartości `CL_SUCCESS` oznacza, iż funkcja zakończyła poprawnie swoje działania. W innym przypadku otrzymać można jeden z następujących kodów:

• `CL_INVALID_VALUE` – wartość parametru `num_objects` wynosi zero, ale wartość `mem_objects` jest różna od `NULL`, bądź `mem_objects` jest równe `NULL`, ale `num_objects` jest większa niż zero;

• `CL_INVALID_MEM_OBJECT` – obiekty pamięciowe podane w parametrze `mem_objects` nie są poprawnymi obiektami;

• `CL_INVALID_COMMAND_QUEUE` – nie podano poprawnego obiektu reprezentującego kolejkę;

• `CL_INVALID_CONTEXT` – podany obiekt kontekstu nie jest poprawny lub też nie został utworzony jako kontekst współpracujący z OpenGL;

• `CL_INVALID_GL_OBJECT` – obiekty pamięciowe podane w `mem_objects` nie zostały utworzone na podstawie obiektów OpenGL;

• `CL_INVALID_EVENT_WAIT_LIST` – liczba zdarzeń w parametrze `num_events_in_wait_list` jest większa od zera, ale parametr `event_wait_list` jest równy `NULL`, lub na odwrót – parametr `event_wait_list` jest różny od `NULL`, ale liczba zdarzeń została ustalona na zero, bądź też obiekty reprezentujące zdarzenia podane w argumencie `event_wait_list` nie są poprawnymi obiektami zdarzeń;

• `CL_OUT_OF_RESOURCES` – zabrakło zasobów niezbędnych do przeprowadzenia operacji po stronie urządzenia obliczeniowego OpenCL;

• `CL_OUT_OF_HOST_MEMORY` – nie udało się przydzielić zasobów niezbędnych do wykonania zadania po stronie gospodarza.

Przykłady użycia:

• zwolnienie obiektu po zakończeniu obliczeń przez jądro obliczeniowe:

```
// inne operacje

cli_err_num = clEnqueueNDRangeKernel(commandQueue, kernel,

    1, NULL,globalWorkSize, localWorkSize,

    0, NULL, NULL);

// inne operacje

cli_err_num = clEnqueueReleaseGLObjects(commandQueue, 1, &cl_vbo_mem, 0, NULL, NULL );
```

Przypisy

[1] Mechanizm odwzorowania pamięci za pomocą funkcji np. `clEnqueueMapBuffer` nie był omawiany, jest on nieco rzadziej stosowany.

Dodatek E. Spis kodów błędów API OpenCL

Sprawdzanie kodów zwracanych przez funkcje API, które sygnalizują pojawienie się błędów, jest kluczowym elementem wielu aplikacji. Szczególnie w przypadku API OpenCL, w wielu przykładach zaprezentowanych we wcześniejszych rozdziałach, interesowało nas, jakie potencjalne błędy mogą się pojawić.

Ogólnie, w przypadku OpenCL, informacje o błędach są zwracane na dwa sposoby: w parametrze funkcji, jeśli dana funkcja tworzy nowy obiekt, lub jako wartość końcowa funkcji.

Przykładami funkcji przynależących do pierwszej grupy są `clCreateContext`, tworząca kontekst obliczeniowy, czy też `clCreateBuffer`, tworząca bufor w pamięci urządzenia obliczeniowego. W tych przypadkach ewentualny kod błędu jest umieszczany w ostatnim argumencie tej metody.

Jednak w przypadku funkcji, których wynikiem nie jest nowy obiekt, wartość końcowa (czyli rezultat funkcji) jest źródłem kodu, np. oznaczającego sukces, tj. `CL_SUCESS`, lub innego oznaczającego wystąpienie błędu.

Należy też wskazać trzecią grupę funkcji, np. `clGetProgramBuildInfo`, za pomocą której można odczytać informację o błędach, jakie zostały wykryte w kodzie źródłowym przyszłego jądra obliczeniowego.

Dla wygody Czytelnika w tab. E.1 podano listę błędów, jakie można otrzymać z poziomu funkcji API OpenCL. Jest to tylko sama lista do szybkiego sprawdzenia znaczenia określonego kodu. Dokładniejsze informacje o tym, jakie błędy mogą zostać zwrócone w przypadku konkretnej funkcji API OpenCL, można znaleźć w oryginalnych materiałach referencyjnych, a wybrany zbiór funkcji API został przedstawiony i opatrzony komentarzem w Dodatku D.

Tabela E.1. Zestaw kodów błędów dla API OpenCL

Kod błędu	Opis
`CL_SUCCESS`	Polecenia zostały wykonane z sukcesem, bez żadnych błędów.
`CL_DEVICE_NOT_FOUND`	Nie znaleziono urządzeń zgodnych ze standardem OpenCL.
`CL_DEVICE_NOT_AVAILABLE`	Urządzenie OpenCL nie jest aktualnie dostępne.
`CL_COMPILER_NOT_AVAILABLE`	Kompilator języka OpenCL C nie jest dostępny na danej platformie obliczeniowej.

CL_MEM_OBJECT_ALLOCATION_FAILURE	Błąd w alokacji pamięci dla bufora lub obrazu.
CL_OUT_OF_RESOURCES	Niewystarczająca ilość zasobów (np. pamięci po stronie urządzenia obliczeniowego) do realizacji aktualnego polecenia.
CL_OUT_OF_HOST_MEMORY	Niewystarczająca ilość pamięci dostępnej w systemie gospodarza do poprawnego wykonania aktualnego polecenia API OpenCL.
CL_PROFILING_INFO_NOT_AVAILABLE	Informacje dot. profilowania nie są dostępne lub też kolejka poleceń ma wyłączoną obsługę informacji o profilowaniu.
CL_MEM_COPY_OVERLAP	Dwa bufory współdzielą ten sam obszar pamięci.
CL_IMAGE_FORMAT_MISMATCH	Podane obrazy mają różne formaty.
CL_IMAGE_FORMAT_NOT_SUPPORTED	Podany format obrazu nie jest wspierany.
CL_BUILD_PROGRAM_FAILURE	Nie udało się zbudować procedury obliczeniowej.
CL_MAP_FAILURE	Pamięć nie może zostać odwzorowana (slangowe „zamapowana”) po stronie systemu gospodarza.
CL_INVALID_VALUE	Argumenty dla polecenia mają niepoprawne wartości.
CL_INVALID_DEVICE_TYPE	Podany typ urządzenia obliczeniowego nie jest poprawną wartością.
CL_INVALID_PLATFORM	Przekazany obiekt reprezentujący platformę nie jest poprawnym obiektem.
CL_INVALID_DEVICE	Przekazany obiekt reprezentujący urządzenie nie jest poprawnym obiektem.

CL_INVALID_CONTEXT	Przekazany obiekt reprezentujący kontekst obliczeniowy nie jest poprawnym obiektem.
CL_INVALID_QUEUE_PROPERTIES	Urządzenie nie akceptuje własności dla kolejki poleceń.
CL_INVALID_COMMAND_QUEUE	Przekazany obiekt reprezentujący kolejkę poleceń nie jest poprawnym obiektem.
CL_INVALID_HOST_PTR	Błędny wskaźnik po stronie gospodarza.
CL_INVALID_MEM_OBJECT	Przekazany obiekt reprezentujący obiekt pamięciowy (np. bufor) nie jest poprawnym obiektem.
CL_INVALID_IMAGE_FORMAT_DESCRIPTOR	Opis formatu obrazu nie jest poprawny.
CL_INVALID_IMAGE_SIZE	Nieobsługiwane wymiary obrazu.
CL_INVALID_SAMPLER	Przekazany obiekt reprezentujący sampler nie jest poprawnym obiektem.
CL_INVALID_BINARY	Przekazany obiekt reprezentujący program w postaci binarnej nie jest poprawnym obiektem.
CL_INVALID_BUILD_OPTIONS	Przekazane opcje (przynajmniej jedna) dla budowania programu nie są poprawne.
CL_INVALID_PROGRAM	Przekazany obiekt reprezentujący program nie jest poprawnym obiektem.
CL_INVALID_PROGRAM_EXECUTABLE	Nie udało się przekształcić programu do postaci wykonywalnej dla urządzeń przyłączonych do kolejki poleceń.
CL_INVALID_KERNEL_NAME	Podana nazwa jądra obliczeniowego nie występuje w programie obliczeniowym.

`CL_INVALID_KERNEL_DEFINITION`	Jądro obliczeniowe określone w programie źródłowym nie jest poprawne.
`CL_INVALID_KERNEL`	Przekazany obiekt reprezentujący jądro obliczeniowe nie jest poprawnym obiektem.
`CL_INVALID_ARG_INDEX`	Argument wskazany przez indeks (np. w wywołaniu `clSetKernelArg`) nie jest poprawny dla danego jądra obliczeniowego.
`CL_INVALID_ARG_VALUE`	Argument dla jądra obliczeniowego jest równy `NULL` dla argumentu nielokalnego lub dla argumentu, który nie może przyjąć wartości `NULL`.
`CL_INVALID_ARG_SIZE`	Wielkość argumentu nie odpowiada wielkości argumentu w jądrze obliczeniowym.
`CL_INVALID_KERNEL_ARGS`	Nie przypisano wartości do argumentów jądra obliczeniowego.
`CL_INVALID_WORK_DIMENSION`	Wymiar dla jednostki roboczej nie jest jedną z dopuszczalnych wartości 1, 2 lub 3.
`CL_INVALID_WORK_GROUP_SIZE`	Wielkość siatki lokalnej bądź globalnej nie jest poprawna.
`CL_INVALID_WORK_ITEM_SIZE`	Wielkość jednostki roboczej przekracza wielkość wspieraną przez urządzenie obliczeniowe.
`CL_INVALID_GLOBAL_OFFSET`	Podana wartość przesunięcia globalnego (ang. *global offset*) przekracza wspierany (obsługiwany) zakres.
`CL_INVALID_EVENT_WAIT_LIST`	Lista / Tablica zdarzeń zawiera niepoprawne obiekty zdarzeń lub jest niewłaściwej wielkości.
`CL_INVALID_EVENT`	Przekazany obiekt reprezentujący zdarzenie nie jest poprawnym obiektem.
`CL_INVALID_OPERATION`	Wykonanie polecenia nie jest poprawne, gdyż może doprowadzić do błędnej operacji.

`CL_INVALID_GL_OBJECT`	Niepoprawny obiekt reprezentujący standard OpenGL.
`CL_INVALID_BUFFER_SIZE`	Wielkość bufora jest poza zakresem.
`CL_INVALID_MIP_LEVEL`	Poziom tzw. mipmapy tekstury OpenGL nie jest poprawny.
`CL_INVALID_GLOBAL_WORK_SIZE`	Liczba globalnych jednostek roboczych nie jest poprawna, np. jest równa zero lub przekracza wielkość wspieraną przez urządzenie obliczeniowe.

Dodatek F. Najczęściej spotykane nazwy rozszerzeń

W standardzie opublikowanym przez grupę Khronos można znaleźć dokument dotyczący dostępnych rozszerzeń, które są całkowicie opcjonalne dla programów pisanych w OpenCL C. W wersji standardu OpenCL 1.2, dokument opisujący rozszerzenia to dziewiąty rozdział standardu. Jest to zazwyczaj oddzielny plik, np. o nazwie *opencl-1.2-extensions.pdf*.

Tabela F.1. Rozszerzenia określone i zdefiniowane przez grupę Khronos

Nazwa rozszerzenia	Opis
`cl_khr_int64_base_atomics`	Obsługa 64-bitowych operacji atomowych.
`cl_khr_int64_extended_atomics`	Obsługa 64-bitowych rozszerzonych operacji atomowych.
`cl_khr_3d_image_writes`	Zapis do obrazów 3D.
`cl_khr_fp16`	Obsługa liczb zmiennoprzecinkowych o tzw. połówkowej/ograniczonej precyzji.
`cl_apple_gl_sharing`	Współpraca ze standardem OpenGL w ramach systemu MacOS X.
`cl_khr_gl_sharing`	Współpraca ze standardem OpenGL.
`cl_khr_gl_event`	Obiekty z poziomu OpenGL mogą generować zdarzenia OpenCL.
`cl_khr_d3d10_sharing`	Współdzielenie obiektów pamięciowych z interfejsem Direct3D 10.
`cl_khr_d3d11_sharing`	Współdzielenie obiektów pamięciowych z interfejsem Direct3D 11.
`cl_khr_dx9_media_sharing`	Współdzielenie obiektów pamięciowych z interfejsem Direct3D 9.

`cl_khr_icd`	Dostęp do systemu ładującego ICD (ang. *installable client driver*).

Literatura

Podane poniżej odniesienia zarówno do internetu, jak też do tradycyjnych publikacji, jak artykuły oraz książki, uzupełniają informacje podane w tej książce. Czytelnik znajdzie tam dodatkowe informacje o algorytmach, technikach programowania równoległego oraz technologii OpenCL.

[Cormen2012] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, Wydanie Siódme, (pierwsze w PWN), Wydawnictwo Naukowe PWN, Warszawa 2012.

[Czech2013] Zbigniew J. Czech, *Wprowadzenie do obliczeń równoległych*, Wydanie II, Wydawnictwo Naukowe PWN, Warszawa 2013.

[PyOpenCLWWW] Strona domowa pakietu PyOpenCL, http://mathema.tician.de/software/pyopencl/, 2014.

[PyOpenCL] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, PyCUDA and PyOpenCL: a *scripting-based approach to GPU run-time code generation*, *Parallel Computing*, Volume 38, Issue 3, March 2012, s. 157–174.

[KhronosWWW] Strona organizacji Khronos, http://www.khronos.org, ostatni dostęp czerwiec 2014.

[NETPBM2013] Pakiet Netpbm, zestaw programów do przetwarzania plików graficznych, http://netpbm.sourceforge.net/doc/index.html, ostatni dostęp maj 2014.

[GIMP] Edytor graficzny GIMP, http://www.gimp.org, ostatni dostęp maj 2014.

[IrfanView] Przeglądarka plików graficznych, http://www.irfanview.com/, ostatni dostęp czerwiec 2014.

[NumPy2014] Pakiet do obliczeń numerycznych w ramach środowiska Python, http://www.numpy.org/, ostatni dostęp czerwiec 2014.

[OpenCLProgGuide2012] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg, *OpenCL Programming Guide*, Addison-Wesley Professional, 2012.

[Scarpino2012] Matthew Scarpino, *OpenCL in Action*, Manning Publications Co, 2012.

[Gaster2013] Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa, *Heterogeneeous Computing with OpenCL* (Revised OpenCL 1.2 Edition), Elsevier Inc., 2013.

[Tay2013] Raymond Tay, *OpenCL Parallel Programming Development Cookbook*, Packt Publishing, 2013.

[Banger2013] Ravishekhar Banger, Koushik Bhattacharyya, *OpenCL Programming by Example*, Packt Publishing, 2013.

[APARAPI] Strona główna projektu APARAPI, http://code.google.com/p/aparapi/, ostatni dostęp

czerwiec 2014.

[AparapiAMD] Strona poświęcona technologii AMD: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/aparapi/, ostatni dostęp czerwiec 2014.

[MALIOpenCLSDK] Strona zawierająca przykłady programowania OpenCL dla układu mobilnego Mali600: http://malideveloper.arm.com/downloads/deved/tutorial/SDK/opencl/index.html, ostatni dostęp czerwiec 2014.

[Bealto2011] Strona zwierająca analizę sposobu implementacji sortowania bitoniczengo, http://www.bealto.com/, ostatni dostęp 30 maja 2014.

[WikiMandelBrot2014] Strona Wikipedii poświęcona zbiorowi Mandelbrota, http://en.wikipedia.org/wiki/Mandelbrot_set, ostatni dostęp maj 2014.

[TinyMT2014] Pakiet implementujący generator liczby pseudolosowych TinyMT: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html, ostatni dostęp czerwiec, 2014.

[HerlihyShavit2010] Maurice Herlihy, Nir Shavit, *Sztuka programowania wieloprocesorowego*, Wydawnictwo Naukowe PWN, Warszawa 2010.