Programowanie procesorów graficznych GPU

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UMCS

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Programowanie procesorów graficznych GPU

Marcin Denkowski
Paweł Mikołajczak

Lublin 2012

Instytut Informatyki UMCS
Lublin 2012

Marcin Denkowski
Paweł Mikołajczak

Programowanie procesorów graficznych GPU

Recenzent: Michał Chlebiej

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
FIGARO Group Sp. z o.o. z siedzibą w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl

ISBN: 978-83-62773-21-3

Spis treści

Spis listingów

Przedmowa

Rozwój procesorów wielordzeniowych CPU oraz procesorów graficznych GPU sprawił, że nawet niedrogie komputery osobiste posiadły moce obliczeniowe rzędu teraflopów umożliwiające przeprowadzanie skomplikowanych obliczeń dostępnych do tej pory jedynie dla superkomputerów. Olbrzymie zapotrzebowanie rynku na wysoce wydajne karty graficzne zwróciły uwagę naukowców i inżynierów, którzy zaczęli wykorzystywać ich moc do przeprowadzania obliczeń ogólnego przeznaczenia (GPGPU – ang. *General-Purpose Computing on Graphics Processing Units*), a to w konsekwencji doprowadziło do powstania pierwszych architektur programowania heterogenicznego łączącego klasyczne podejście wykonywania obliczeń za pomocą procesora centralnego CPU z asynchronicznym wykonywaniem wysoce zrównoleglonych algorytmów za pomocą procesora graficznego GPU.

Skrypt ten prezentuje ogólne wprowadzenie do zagadnień wykorzystania kart graficznych do obliczeń dowolnego typu i zawiera jedynie podstawowe informacje, które są niezbędne aby tworzyć programy heterogeniczne, skupiając się na opisie dwóch głównych architektur NVIDIA CUDA$^{\text{TM}}$oraz OpenCL$^{\text{TM}}$. Skrypt ten nie stanowi podręcznika do nauki programowania równoległego na ogólnym poziomie. W niektórych punktach zrezygnowano z dogłębnego omawiania problemu, aby nie rozpraszać uwagi czytelnika od głównych zagadnień.

Ponieważ architektura CUDA jako pierwsza umożliwiła przeprowadzanie obliczeń ogólnego przeznaczenia za pomocą kart graficznych w języku wysokiego poziomu i w dalszym ciągu wyznacza kierunki rozwoju technik GPGPU niniejszy podręcznik w dużej mierze opiera się właśnie na tej architekturze oraz kartach graficznych firmy NVIDIA. Standard OpenCL jako swego rodzaju generalizacja technik GPGPU na dowolne urządzenia obliczeniowe jest w swej specyfikacji bardzo podobny do architektury CUDA i jego omówienie zostało często ograniczone do wskazania różnic pomiędzy tymi oboma architekturami.

Niniejszy skrypt jest równocześnie pomyślany jako zestawienie i porównianie obu środowisk programowania heterogenicznego. W przypadku złożonych problemów lub dużych różnic pomiędzy architekturami, zagadnienia

ich realizacji w konkretnym środowisku zostały umieszczone w osobnych podrozdziałach. W przypadku omawiania danego zagadnienia, przy niewielkich różnicach pomiędzy oboma środowiskami, ich kody źródłowe zostały umieszczone w tym samym podrozdziale a konkretne środowisko zostało zaznaczone poprzez adnotację na marginesie w postaci nazwy CUDA lub OpenCL. Kod w języku CUDA został dodatkowo otoczony ramką, natomiast kod w języku OpenCL został otoczony ramką i drukowany na szarym tle.

OpenCL

Układ książki odpowiada kolejności w jakiej powinno się czytać niniejszy skrypt. Pierwszy rozdział zawiera wprowadzenie do programowania w środowiskach CUDA/OpenCL, opisuje sposób instalacji odpowiednich bibliotek oraz przedstawia pierwszy program, łącznie z jego omówieniem. Drugi rozdział koncentruje się na opisie modelu wykonania programu heterogenicznego obserwowanego z punktu widzenia *hosta*. Rozdział trzeci opisuje typy pamięci znajdujące się na karcie grafiki oraz zawiera parę wskazówek na temat optymalizacji dotępu do takiej pamięci. Rozdziały czwarty oraz piąty stanowią przegląd rozszerzeń jakie wprowadzają języki CUDA C oraz OpenCL C do klasycznego języka C. Rozdział szósty zawiera opis możliwości i technik współpracy potoku graficznego z programowaniem GPGPU. W pierwszym dodatku zawarto implementacje kilku przydatnych funkcji, często używanych w listingach tego skryptu, a w dodatku drugim zawarto specyfikację potencjału obliczeniowego (*Compute Capabilities*) kart graficznych firmy NVIDIA.

Książka stanowi podręcznik dla studentów kierunku informatyka specjalizujących się w zagadnieniach programowania równoległego z wykorzystaniem kart graficznych, choć może być przydatna również dla studentów innych kierunków naukowych lub technicznych oraz dla innych osób wykorzystujących w swojej pracy obliczenia GPGPU. Do poprawnego zrozumienia podręcznik wymaga przynajmniej podstawowego doświadczenia w programowaniu w języku C.

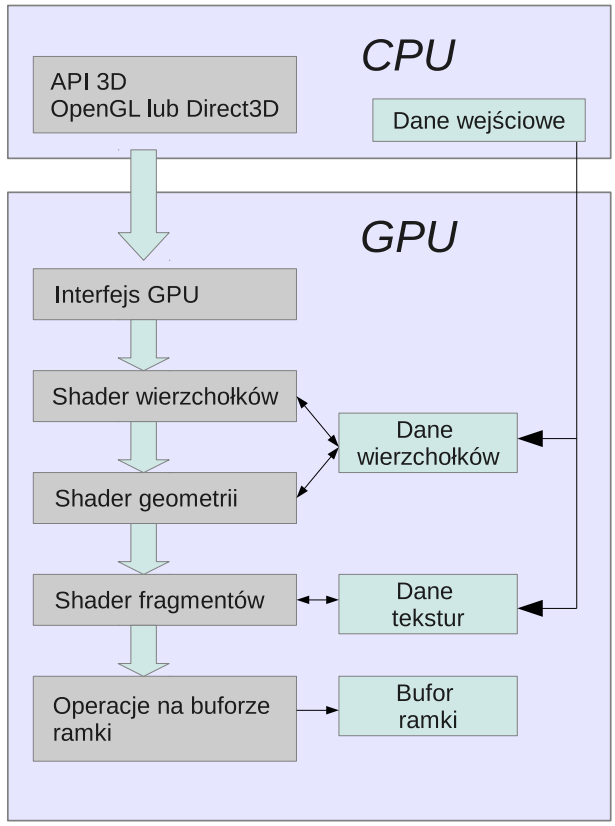
Rozdział 1

Wprowadzenie do Nvidia CUDA i OpenCL

1.1. Architektura urządzeń GPU

Burzliwy rozwój kart graficznych, a w szczególności procesorów graficznych uczynił z nich bardzo wydajne urządzenia obliczeniowe, umożliwiające wykonywanie wysoce zrównoleglonych algorytmów ogólnego przeznaczenia. Warto jednak mieć na uwadze pochodzenie i główne przeznaczenie tych urządzeń, tak aby tworząc programy heterogeniczne ogólnego przeznaczenia móc wykorzystać ich pełnie możliwości.

Współczesne procesory graficzne umożliwiają generowanie, w czasie rzeczywistym, realistycznej grafiki dzięki wprowadzeniu programowalnych jednostek realizujących potok graficzny w miejsce ich statycznych odpowiedników. Uproszczony model takiego potoku został pokazany na rysunku 1.1.



Rysunek 1.1. Model potoku graficznego współczesnych kart graficznych.

Do komunikacji *hosta* z kartą graficzną służy specjalnie zaprojektowane API (ang. *Application Programming Interface*), umożliwiające realizację typowych zadań związanych z generowaniem grafiki. Sam potok graficzny składa się z kilku etapów. Dane wierzchołków, z których składają się pod-

stawowe prymitywy graficzne, po skopiowaniu z pamięci *hosta* są najpierw przetwarzane przez specjalny program w obrębie tzw. shadera wierzchołków. Jego celem jest obliczenie odpowiednio przetransformowanej pozycji każdego z wierzchołków w przestrzeni trójwymiarowej oraz ich oświetlenie i pokolorowanie. Tak przekształcone wierzchołki trafiają następnie do programu przetwarzającego geometrię, którego celem jest konstrukcja większych prymitywów (w obecnych kartach graficznych są to zazwyczaj trójkąty). Po rasteryzacji powstałych w ten sposób prymitywów, do pracy rusza kolejna jednostka cieniująca, zwana shaderem fragmentów, której celem jest obliczenie koloru każdego punktu danego prymitywu. W tym shaderze możliwe jest wykorzystanie innej porcji danych pochodzących z pamięci *hosta*, tzw. tekstur, stanowiących dwu- lub trójwymiarowe obrazy. Przygotowany w ten sposób obraz zapisywany jest w pamięci bufora ramki i zazwyczaj wyświetlany na ekranie.

Zapotrzebowanie na zdolność przetwarzania coraz większej ilości wierzchołków i cieniowania coraz większej ilości punktów wymusiły specyficzną konstrukcję procesorów graficznych. Każdy z programowalnych kroków potoku wymagał bowiem przeprowadzenia bardzo podobnych (zwykle prostych algorytmicznie) obliczeń dla olbrzymiej ilości danych. Takiemu zadaniu mogły sprostać tylko konstrukcje zbudowane z dużej ilości prostych jednostek obliczeniowych. Początkowo, dla każdego typu shadera, wewnątrz GPU, znajdowały się dedykowane jednostki obliczeniowe. Począwszy od wprowadzonego w 2006 roku, procesora GeForce 8800 wszystkie jednostki zostały zunifikowane a potok programowy był realizowany dla każdego typu shadera na wszystkich dostępnych jednostkach obliczeniowych. Coraz bardziej zaawansowane możliwości jednostek cieniujących wraz z wprowadzeniem operacji arytmetycznych na liczbach zmiennoprzecinkowych otworzyły drogę do wykorzystania procesorów graficznych do rozwiązywania bardziej ogólnych problemów, często nie związanych w żaden sposób z generowaniem grafiki. Wtedy też powstał termin GPGPU (ang. *General-Purpose Computing on Graphics Processing Units*) oznaczający przeprowadzanie dowolnych obliczeń za pomocą procesora graficznego. Jednakże, poważnym problemem był brak bezpośredniego dostępu do karty graficznej, realizowanego jak do tej pory jedynie za pomocą dedykowanego API takiego jak OpenGL czy DirectX. Oznaczało to, że dany problem obliczeniowy trzeba było najpierw przekształcić tak, aby odpowiadał w formie bibliotecznym operacjom graficznym, które mogły być wykonane poprzez odpowiednie wywołania API graficznego. Zatem, wszelkie dane wejściowe należało albo przedstawić w formie wierzchołków albo tekstur 2D/3D. Wynik obliczeń również musiał być zapisany w postaci bufora ramki lub tekstury. Odpowiedzią na coraz większe zapotrzebowanie na obliczenia tego typu było opracowanie przez firmę NVIDIA architektury Tesla^{TM}oraz dedykowanego API o nazwie CU-

DA w 2007 roku. W roku 2008 grupa Khronos publikuje również pierwszą specyfikację architektury OpenCL 1.0 [4].

W stosunku do ówczesnych kart graficznych, wprowadzenie możliwości obsługi algorytmów dowolnego przeznaczenia, wymagało jednak pewnych modyfikacji sprzętu, takich jak dodanie lokalnej pamięci do jednostek obliczeniowych, dodatkowej pamięci podręcznej czy dedykowanej logiki kontrolującej wykonywanie instrukcji. Niezbędne okazało się również dodanie możliwości swobodnego dostępu do pamięci globalnej dla każdej jednostki obliczeniowej, opracowanie bardziej ogólnego modelu programowania umożliwiającego hierarchizację wątków, ich synchronizację czy dodanie operacji atomowych. Przykład tak skonstruowanego procesora GPU jest przedstawiony na rysunku 1.2

Dalszy rozwój GPGPU prowadził do dalszych udogodnień a co za tym idzie do wprowadzania coraz większej ilości obsługiwanych właściwości. Istotnym posunięciem wydawało się jednak zachowanie wstecznej kompatybilności, tak aby implementacje algorytmów pisane dla starszych architektur, bez wprowadzania jakichkolwiek zmian działały również na nowym sprzęcie. Dla rozróżnienia konkretnych architektur NVIDIA wprowadziła określenie *Compute Capability* opisujące możliwości obliczeniowe danego procesora graficznego. Pełne zestawienie *Compute Capabilities* oraz spis obsługiwanych funkcjonalności obliczeniowych zebrane zostały w Dodatku B.

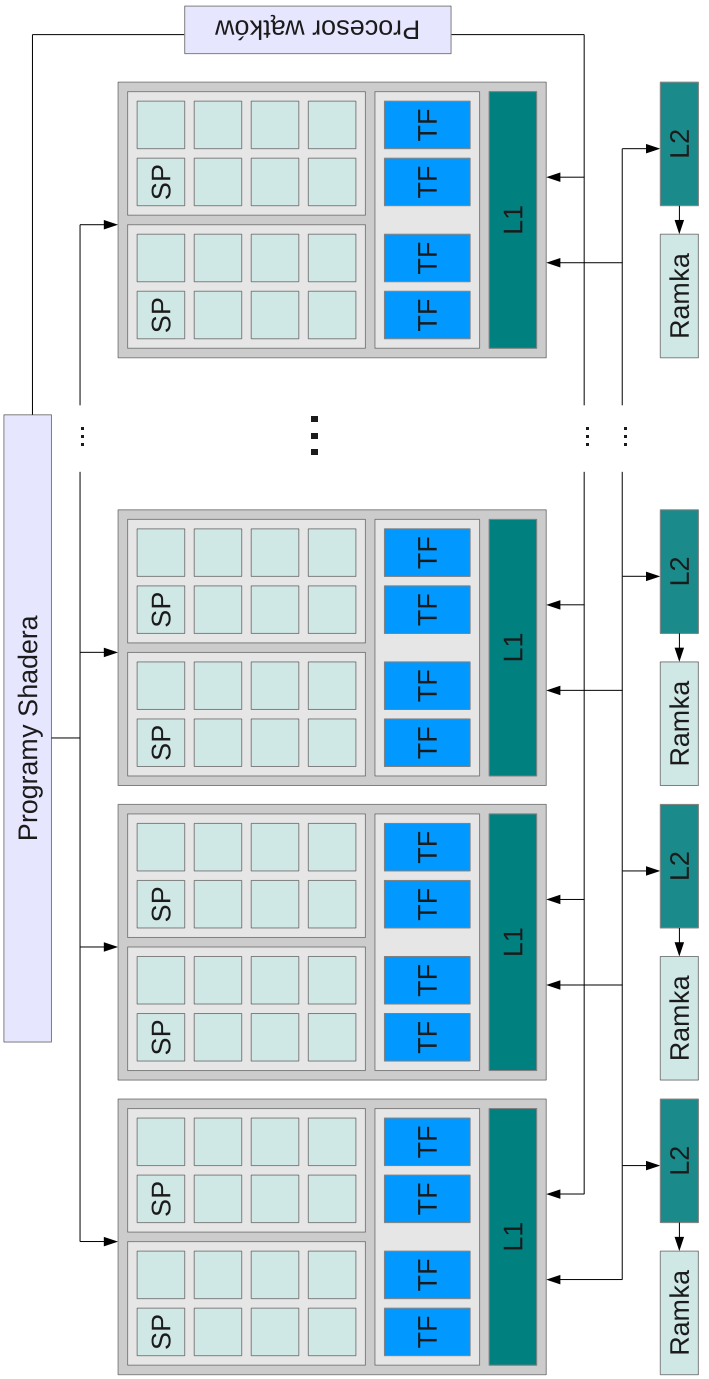
Architektura OpenCL jeszcze bardziej uogólniła przetwarzanie heterogeniczne umożliwiając wykonywanie obliczeń na dowolnym *urządzaniu obliczeniowym*, którym może być z równym powodzeniem karta grafiki, procesor centralny CPU czy dedykowana karta obliczeniowa.

1.2. Instalacja środowiska

W niniejszym podrozdziale przedyskutowany zostanie proces instalacji środowisk programistycznych do przetwarzania równoległego na GPU z podziałem na dwóch głównych producentów procesorów graficznych NVIDIA i AMD.

1.2.1. NVIDIA

NVIDIA, jako jeden z prekursorów *GPU computing*, stworzyła najpierw środowisko CUDA, a później na bazie tego środowiska wprowadziła obsługę standardu OpenCL. Oba środowiska dostarczane są w jednym pakiecie o nazwie *CUDA Toolkit*. W chwili pisania niniejszego podręcznika dostępna była wersja 4.1 tego pakietu. W osobnym pakiecie o nazwie *GPU Computing SDK* NVIDIA dostarcza przykładowe programy dla CUDA, OpenCL,



Rysunek 1.2. Model zunifikowanej architektury współczesnego procesora graficznego.

DirectCompute oraz szereg dodatkowych bibliotek. Oba pakiety są dostępne za darmo do pobrania ze strony *http://developer.nvidia.com*.

W celu wykorzystania możliwości kart graficznych niezbędna jest odpowiednia karta graficzna oparta na układzie GeForce serii conajmniej 8000, Quadro lub Tesla (pełną listę obsługiwanych urządzeń można znaleźć na *http://www.nvidia.com/object/cuda_gpus.html*) oraz zainstalowane sterowniki w wersji co najmniej 270.

Linux

W przypadku systemu Linux, w wielu dystrybucjach są dostępne pakiety oprogramowania CUDA przygotowane specjalnie dla danej dystrybucji. W takim przypadku, w celu zainstalowania środowiska CUDA należy posłużyć się dedykowanym managerem pakietów. Poniższy sposób będzie dotyczył jedynie ręcznej instalacji w oparciu o wersję 4.1.28 tego środowiska.

Ze stron developerskich (*http://developer.nvidia.com/cuda-downloads*) należy pobrać plik instalacyjny *CUDA Toolkit*, w wersji dla danej dystrybucji Linuxa (lub zbliżonej). Przykładowo dla dystrybucji *Ubuntu* będzie to plik o nazwie:

`cudatoolkit_4.1.28_linux_64_ubuntu11.04.run` – dla 64-bitowego systemu

lub

`cudatoolkit_4.1.28_linux_32_ubuntu11.04.run` – dla 32-bitowego systemu.

Pobrany plik należy uruchomić i postępować zgodnie z zaleceniami instalatora. Domyślnie, instalator skopiuje niezbędne pliki do katalogu `/usr/local/cuda`. Po instalacji należy ustawić zmienne środowiskowe:

```
export PATH=$PATH:/usr/local/cuda/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib:
                       /usr/local/cuda/lib64
```

lub dodać te zmienne w globalnym pliku `/etc/profile` lub lokalnie dla danego użytkownika w `~/.bash_profile`.

Przykładowe kody źródłowe oraz dodatkowe biblioteki zawarte w pakiecie *GPU Computing SDK* można zainstalować poprzez uruchomienie pobranego pliku o nazwie `gpucomputingsdk_4.1.28_linux.run`. Domyślnie wybrana lokalizacja instalacji `$(HOME)/NVIDIA_GPU_Computing_SDK` wydaje się rozsądnym rozwiązaniem.

W celu weryfikacji poprawności instalacji można wykonać polecenie

```
nvcc --version
```

wypisujące na terminalu aktualną wersję środowiska. Warto również skompilować przykładowe programy znajdujące się w katalogu

`$(HOME)/NVIDIA_GPU_Computing_SDK/C/src`. Po poprawnej kompilacji uruchomienie programu `deviceQuery`, powinno wypisać na terminalu najważniejsze parametry urządzeń zgodnych z technologią CUDA.

Poprawność działania środowiska OpenCL można sprawdzić kompilując przykładowe programy znajdujące się w katalogu `$(HOME)/NVIDIA_GPU_Computing_SDK/OpenCL/src`. Uruchomienie programu `oclDevileQuery` powinno dać podobny rezultat jak powyżej wylistowując na terminalu wszystkie urządzenia zgodne z technologią OpenCL.

Windows XP / Vista / Windows 7

Wersja środowiska CUDA dla systemu Windows wymaga zainstalowanego pakietu *MS Visual Studio* w wersji 2005, 2008 lub 2010 (lub odpowiadającej wersji *MS Visual C++ Express*).

Ze stron developerskich (*http://developer.nvidia.com/cuda-downloads*) należy pobrać plik instalacyjny *CUDA Toolkit* o nazwie:

`cudatoolkit_4.1.28_win_64.msi` – dla 64-bitowego systemu

lub

`cudatoolkit_4.1.28_win_32.msi` – dla 32-bitowego systemu.

Pobrany plik należy uruchomić i postępować zgodnie z zaleceniami instalatora. Domyślnie, środowisko zostanie zainstalowane w katalogu `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA`

Przykładowe kody źródłowe oraz dodatkowe biblioteki pakietu *GPU Computing SDK* można zainstalować poprzez uruchomienie pobranego pliku o nazwie `gpucomputingsdk_4.1.28_win_64.exe` (lub `gpucomputingsdk_4.1.28_win_32.exe` dla systemu 32-bitowego). Pliki tego pakietu zostaną skopiowane do katalogu `%ProgramData%\NVIDIA Corporation\NVIDIA GPU Computing SDK`.

Weryfikacja poprawności instalacji środowiska może polegać na kompilacji i uruchomieniu przykładowych programów pakietu *GPU Computing SDK*. Pakiet ten dostarcza przykładowe rozwiązania zarówno w wersji źródłowej jak i skompilowanej.

Przykładowo, uruchomienie programu `deviceQuery` znajdującego się w katalogu `%ProgramData%\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\bin\win64\Release` (w wersji 32-bitowej Windows w `..\win32\Release`) wypisze na ekranie konsoli wszystkie urządzenia zgodne z technologią CUDA oraz ich najważniejsze parametry.

1.2.2. AMD

AMD/ATI nie tworzyła nigdy własnego środowiska ale aktywnie włączyła się w rozwój otwartego standardu OpenCL. Na stronach deweloperskich zostało udostępnione środowisko programistyczne w postaci SDK (ang. *Software Development Kit*) o nazwie *AMD Accelerated Parallel Processing (APP)*, zastępujące, znane pod pod nazwą *ATI Stream*, starsze *SDK*.

Aktualna wersja *APP* o numerze 2.6 zawiera wsparcie dla standardu OpenCL 1.1. Możliwość przeniesienia obliczeń na kartę graficzną z procesorem ATI wymaga procesora conajmniej serii 5 dla OpenCL 1.1 lub serii 4 dla OpenCL 1.0 oraz sterownika *ATI Catalyst* w wersji co najmniej 11.7. W przeciwieństwie do pakietu NVIDII, AMD *APP* umożliwia uruchomienie obliczeń równoległych na klasycznym procesorze CPU zgodnym z architekturą *x86* z obsługą *SSE 2*.

Poniżej została przedstawiona procedura instalacji tego środowiska z podziałem na system operacyjny:

Linux

Wersja Linuxowa środowiska *SDK* wymaga kompilatora *GCC* w wersji co najmniej 4.1 lub kompilatora *Intel C Compiler (ICC)* w wersji co najmniej 11.x.

Na stronach deweloperskich AMD (*http://developer.amd.com/sdks/AMDAPPSDK/downloads*) należy pobrać plik o nazwie:

`AMD-APP-SDK-v2.6-lnx64.tgz` – dla 64-bitowego systemu
lub
`AMD-APP-SDK-v2.6-lnx32.tgz` – dla 32-bitowego systemu.

Proces instalacji środowiska przebiega w następujący sposób (niezbędne są uprawnienia `roota`):

1. Pobrany plik należy rozpakować poleceniem:

   ```
   tar xfzv AMD-APP-SDK-v2.6-lnx64.tgz
   ```

2. Za pomocą polecenia:

   ```
   tar xfzv AMD-APP-SDK-v2.6-RC3-lnx64.tgz
   ```

   w aktualnym katalogu zostanie utworzony katalog o nazwie `AMD-APP-SDK-v2.6-RC2-lnx64` zawierający wszystkie niezbędne pliki środowiska. Domyślnie zawartość tego katalogu należy skopiować do katalogu `/opt/AMDAPP`.

3. Polecenie

```
tar -xvzf icd-registration.tgz
```

rozpakuje zawartość archiwum `icd-registration.tgz` tworząc katalog `etc/OpenCL` wewnątrz aktualnego katalogu. Powstały katalog `OpenCL` należy skopiować do katalogu `/etc/`.

4. Ustawić zmienne środowiskowe:

```
export AMDAPPSDKROOT=/opt/AMDAPP/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/AMDAPP/lib/
                       x86_64:/opt/AMDAPP/lib/x86
```

lub dodać te zmienne w globalnym pliku `/etc/profile` lub lokalnie dla danego użytkownika w `~/.bash_profile`.
Dla 32-bitowego środowiska należy zamienić w odpowiednich nazwach liczbę 64 na 32.

Windows Vista / Windows 7

Wersja *SDK* środowiska OpenCL dla systemu Windows wymaga kompilatora *Microsoft Visual Studio (MSVS)* w wersji 2008 lub 2010 lub kompilatora *Intel C Compiler (ICC)* w wersji co najmniej 11.x lub kompilatora *Minimalist GNU for Windows (MinGW)* w wersji co najmniej 4.4.

Na stronach deweloperskich AMD (*http://developer.amd.com/sdks/AMDAPPSDK/downloads*) należy pobrać plik o nazwie:

`AMD-APP-SDK-v2.6-Windows-64.exe` – dla 64-bitowego systemu
lub
`AMD-APP-SDK-v2.6-Windows-32.exe` – dla 32-bitowego systemu.

W celu zainstalowania środowiska *SDK* należy uruchomić pobrany plik i postępować według wskazówek instalatora. Domyślnie, cały pakiet oprogramowania zostanie zainstalowany w katalogu `C:\Program Files\AMD APP`.

1.3. Pierwszy program

1.3.1. Rozwiązanie klasyczne

Przeanalizujmy program, który podnosi do kwadratu wszystkie elementy wektora. Na początek program napisany klasycznie w języku C++:

Listing 1.1. Klasyczny program – kwadrat wektora.

```
1 #include <iostream>
2 #include <stdlib.h>
3
4 void pow2(float *vec, int size);
5
6 int main(int argc, char* argv[])
7 {
8   const int size = 1024*1024*128;
9   float *vec = new float[size];
10
11  for(int i=0;i<size; ++i)
12    vec[i] = rand()/(float)RAND_MAX;
13
14  double time = timeStamp();
15  pow2(vec, size);
16  std::cout<< "Time: " << timeStamp() - time << std::endl;
17 }
```

Dla potrzeb testu w linii 9 tworzony jest wektor składający się z $1024 \times 1024 \times 128 = 134217728$ elementów typu `float` (około 500MB), wypełniany następnie liczbami pseudolosowymi za pomocą bibliotecznej funkcji `rand()`. Sama funkcja `pow2()` przeprowadzająca mnożenie została zdefiniowana następująco:

Listing 1.2. Klasyczna funkcja podnosząca elementy wektora do kwadratu.

```
1 void pow2(float *vec, int size)
2 {
3   for(int i=0; i<size; ++i)
4     vec[i] = vec[i] * vec[i];
5 }
```

Do pomiaru czasu została wykorzystana funkcja `double timeStamp()` zdefiniowana na listingu A.1 dla systemu Linux oraz na listingu A.2 dla systemu Windows w Dodatku A. W obu przypadkach pomiar czasu jest z dokładnością co do około 1 milisekundy.

Tak napisany program jest klasycznym przykładem programowania sekwencyjnego, szeregowego, w którym, w danym momencie wykonuje się tylko jedna instrukcja. Tymczasem mnożenia kolejnych elementów potęgowanego wektora są zupełnie niezależne od siebie, zatem mogą być wykonywane w tym samym czasie. Powyższy program można by zatem przepisać zastępując funkcję `pow2()` jej nową wersją:

Listing 1.3. Klasyczny program – kwadrat wektora równolegle.

```
1 void pow2(float *a, int i)
2 {
3     a[i] = a[i] * a[i];
4 }
```

która wykonuje działanie tylko dla jednego konkretnego `i`-tego elementu tablicy. Tę funkcję należy teraz w programie głównym wywołać `size` razy równolegle, po jednym razie dla każdego elementu wektora. W nomenklaturze architektur GPGPU tego typu funkcja, wywoływana wielokrotnie i równolegle dla elementów pewnego zbioru danych nazywana jest rdzeniem (ang. *kernel*) lub zazwyczaj po prostu *kernelem*.

Poniżej przedstawiony zostanie sposób na realizację takiego równoległego mnożenia dla obu rozważanych architektur.

1.3.2. Program w CUDA

Program realizujący taką funkcjonalność w środowisku NVIDIA CUDA będzie wyglądał następująco (przy wykorzystaniu wysokopoziomowego API):

Listing 1.4. CUDA – Kwadrat wektora – plik `hello_cuda.cu`.

```
1  #include <cuda_runtime_api.h>
2  #include <iostream>
3
4  __global__ void pow2(float *vec)
5  {
6    int i = gridDim.x*blockDim.x*blockIdx.y +
7            blockIdx.x*blockDim.x + threadIdx.x;
8
9    vec[i] = vec[i] * vec[i];
10 };
11
12 int main(int argc, char *argv[])
13 {
14   const int size = 1024*1024*128;
15   float *cpuVec = new float[size];
16
17   for(int i=0;i<size; ++i)
18     cpuVec[i] = rand()/(float)RAND_MAX;
19
20   double time = timeStamp();
21
22   float *gpuVec;
23   cudaMalloc((void**)&gpuVec, sizeof(float)*size);
24
25   cudaMemcpy(gpuVec, cpuVec, sizeof(float)*size,
```

```
26              cudaMemcpyHostToDevice);
27
28   dim3 blocks(1024,1024,1);
29   dim3 threads(128,1,1);
30   pow2<<<blocks, threads>>>(gpuVec);
31
32   cudaMemcpy(cpuVec, gpuVec, sizeof(float)*size,
33              cudaMemcpyDeviceToHost);
34
35   delete[] cpuVec;
36   cudaFree(gpuVec);
37
38   cout << "Time on GPU: " << timeStamp() - time << endl;
39 }
```

Przy poprawnie zainstalowanym środowisku developerskim kompilacja tego programu odbywa się za pomocą dedykowanego kompilatora `nvcc`. W powyższym przykładzie wystarczy wykonanie polecenia:

```
#nvcc hello_cuda.cu
```

gdzie `hello_cuda.cu` jest nazwą pliku zawierającego kod źródłowy tego programu. W samym kodzie programu:

- W liniach 4–10 została zdefiniowana funkcja `__global__ void pow2(float*)`. Specyfikator `__global__` informuje, że ta funkcja będzie kernelem CUDA wykonującym się na procesorze GPU.

- W liniach 6–7 obliczany jest indeks aktualnie przetwarzanego elementu wektora za pomocą wbudowanych zmiennych `gridDim`, `blockDim`, `blockIdx` i `threadIdx`. Znaczenie tych zmiennych zostanie wyjaśnione w trochę później.

- W linii 9 funkcji `pow()` następuje właściwe mnożenie elementu wektora. W funkcji tej nie występują żadne pętle iterujące po elementach a dzięki zrównolegleniu wykona się ona po jednym razie dla każdego elementu wektora.

- W liniach 22–36 zawarta została cała logika przeniesienia obliczeń na procesor GPU. Najpierw w linii 23 alokowana jest pamięć na karcie grafiki za pomocą funkcji:

  ```
  cudaError_t cudaMalloc(void** devPtr, size_t size)
  ```

  Pomimo, że obszar tej pamięci jest wskazywany zwykłym wskaźnikiem `float* gpuVec`, to wskazuje on na obszary pamięci grafiki, a dostęp do tej pamięci jest możliwy jedynie przez dedykowane funkcje.

- W liniach 25–26 za pomocą funkcji:

```
cudaError_t cudaMemcpy(void *dst, const void *src,
            size_t count, enum cudaMemcpyKind kind)
```

obszar pamięci znajdujący się w pamięci komputera (nazywanym *hostem*) wskazywany wskaźnikiem `src` jest kopiowany do pamięci karty grafiki (nazywanej *urządzeniem*, ang. *device*) wskazywanej przez wskaźnik `dst`. Typ wyliczeniowy `enum cudaMemcpyKind` decyduje o kierunku i urządzeniach biorących udział w kopiowaniu. W powyższym przykładzie parametr ten ma wartość `cudaMemcpyHostToDevice` i zapewnia kopiowanie z *hosta* do *urządzenia*.

- Linie 28–30 zawierają wywołanie kernela CUDA. Wprowadzona została tutaj nowa składnia wywołania funkcji rdzenia o postaci:

```
kernel<<<dim3 Dg, dim3 Db>>>(...)
```

Jest to składnia akceptowana jedynie przez kompilator `nvcc` i nie jest zgodna ze standardem C/C++. Parametr `dim3 Dg` jest trójwymiarowym wektorem wyznaczającym rozmiar siatki (ang. *grid*) bloków (ang. *block*). Parametr `dim3 Db` jest trójwymiarowym wektorem wyznaczającym rozmiar bloku składającego się z pojedynczych wątków (ang. *thread*).
W obliczeniach GPGPU zwykło się organizować pojedyncze wątki w trójwymiarowe struktury grupujące. Podstawową strukturą jest blok będący jedno-, dwu- lub trójwymiarową strukturą wątków. Liczba wątków przypadających na blok jest z góry ograniczona, ponieważ wszystkie wątki powinny znajdować się na pojedynczym rdzeniu procesora GPU. Maksymalna liczba jest zależna od danego urządzenia i aktualnie oscyluje w granicach 1024. Bloki można organizować w jedno-, dwu- lub trójwymiarową strukturę zwaną siatką (ang. *grid*). Liczba bloków w siatce jest z reguły podyktowana rozmiarem przetwarzanych danych. Maksymalny rozmiar siatki jest także ograniczony.
W powyższym przykładzie zastosowano dwuwymiarową siatkę składającą się z 1024×1024 bloków. Każdy blok składa się ze 128 wątków. Wewnątrz rdzenia informacje o rozmiarze siatki i bloku są dostępne przez wspomniane już wbudowane zmienne `gridDim`, `blockDim`.

- W liniach 32–33 za pomocą funkcji `cudaMemcpy()` wektor z pamięci karty grafiki (*device*) jest kopiowany z powrotem do pamięci RAM komputera (*host*).

- W linii 35 za pomocą klasycznego operatora `delete` usuwany jest z pamię-

ci *hosta* wektor `cpuVec`, a w linii 36 usuwany jest z pamięci karty grafiki (*device*) wektor `gpuVec`.

1.3.3. Program w OpenCL

W środowisku OpenCL program o analogicznej funkcjonalności będzie miał postać:

Listing 1.5. OpenCL – Kwadrat wektora – plik `hello_opencl.cpp`.

```
1  #include <CL/opencl.h>
2  #include <iostream>
3
4  char* loadProgSource(const char*, const char*, size_t*);
5
6  int main(int argv, char *argc[])
7  {
8    const int size = 1024*1024*128;
9    float* cpuVec = new float[size];
10
11   for (int i = 0; i < size; i++)
12     cpuVec[i] = rand()/(float)RAND_MAX;
13
14   double time = timeStamp();
15
16   cl_platform_id platform;
17   cl_device_id devices;
18   clGetPlatformIDs(1, &platform, NULL);
19   clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &devices,
20                  NULL);
21   cl_context context = clCreateContext(0, 1, &devices,
22                        NULL, NULL, NULL);
23
24   cl_command_queue cmdQueue = clCreateCommandQueue(context,
25                                     devices, 0, &errcode);
26
27   size_t kernelLength;
28   char* programSource = loadProgSource("pow2.cl", "",
29                                        &kernelLength);
30   cl_program program = clCreateProgramWithSource(context,
31                        1, (const char**)&programSource,
32                        &kernelLength, &errcode);
33   clBuildProgram(program, 0, 0, 0, 0, 0);
34   cl_kernel kernel = clCreateKernel(program, "pow2",
35                                     &errcode);
36
37   cl_mem clVec = clCreateBuffer(context, CL_MEM_READ_WRITE,
38                        size*sizeof(float),  0,   &errcode);
39   clEnqueueWriteBuffer(cmdQueue, clVec,  CL_FALSE, 0,
40                  size*sizeof(float), cpuVec, 0, NULL, NULL);
41
```

```
42   clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&clVec);
43   size_t dims[3] = {size,1,1};
44   size_t localDims[3] = {128,1,1};
45   clEnqueueNDRangeKernel(cmdQueue, kernel, 1, 0,
46                          dims, localDims, 0, 0, 0);
47
48   clEnqueueReadBuffer(cmdQueue, clVec, CL_TRUE, 0,
49                   size*sizeof(int), cpuVec, 0, NULL, NULL);
50   clFinish(cmdQueue);
51
52   delete[] cpuVec;
53   clReleaseMemObject(clVec);
54   std::cout << "Time on GPU: " << timeStamp() - time << std::endl;
55 }
```

W przeciwieństwie do środowiska CUDA, kompilacja tego programu odbywa się za pomocą klasycznego kompilatora C/C++. W przypadku pakietu GCC wystarcza wydanie polecenia:

```
#g++ -lOpenCL hello_opencl.cpp
```

gdzie `hello_opencl.cpp` jest nazwą pliku zawierającego kod źródłowy tego programu. Budowa samego kernela odbywa się już podczas działania programu a odpowiedzialny za nią jest wbudowany w bibliotekę OpenCL kompilator. W samym programie:

- Linie 16–22 zawierają wywołania funkcji tworzących kontekst (ang. *context*) obliczeń dla danej platformy i urządzenia obliczeniowego. Procedura ta jest niezbędna ponieważ, w odróżnieniu od architektury CUDA, OpenCL może działać na wielu platformach obliczeniowych. Z równym powodzeniem może to być karta grafiki, zwykły procesor czy dedykowana karta obliczeniowa.

- W linii 24 za pomocą funkcji `clCreateCommandQueue()` tworzona jest dla kontekstu `context` i urządzenia `device` kolejka poleceń do wykonania na urządzeniu obliczeniowym. Kolejne wywołania API funkcji OpenCL są jedynie dodawane do wykonania do takiej kolejki a o czasie wykonania tych rozkazów decyduje środowisko. Można jednakże wymusić wykonanie znajdujących się w kolejce poleceń funkcją:

  ```
  cl_int clFinish(cl_command_queue queue)
  ```

  która blokuje aktualny wątek do czasu zakończenia wszystkich zakolejkowanych poleceń.

- W liniach 27-33 wczytywany, tworzony i kompilowany jest tzw. program w kontekście OpenCL. W odróżnieniu od środowiska CUDA, tutaj funk-

cje wykonywalne na urządzeniu, zbierane są w postaci programu, którego kod źródłowy jest reprezentowany w formie napisu (c-stringu) lub w formie binarnej. W linii 28 wczytywany jest do stringu program zawierający definicję funkcji rdzenia o nazwie `pow2` za pomocą funkcji `loadProgSource()` zdefiniowanej na listingu A.3 w Dodatku A. Kod programu zapisany jest w pliku o nazwie `pow2.cl`. Zawartość pliku z definicją rdzenia przedstawia poniższy listing:

Listing 1.6. OpenCL – Kwadrat wektora – plik `pow2.cl` z definicją rdzenia.

```
1 __kernel void pow2(__global  float* vec)
2 {
3     int i = get_global_id(0);
4     vec[i] = vec[i] * vec[i];
5 }
```

- W liniach 30–32 tworzony jest program za pomocą funkcji `clCreateProgramWithSource()`. Dany program może dowolną ilość definicji funkcji rdzeni. Stworzony program jest następnie budowany za pomocą funkcji `clBuildProgram()`. Na tym etapie dany program jest kompilowany i łączony za pomocą kompilatora OpenCL.
- W liniach 34–35 z podanego programu `program` tworzona jest funkcja kernela `pow2()` za pomocą funkcji `clCreateKernel()`. Funkcja rdzenia jest identyfikowana po nazwie przekazanej w drugim parametrze funkcji `clCreateKernel()`.
- W liniach 37–40 tworzony jest obiekt pamięciowy (*Memory Object*) reprezentujący obiekt w pamięci urządzenia obliczeniowego. Użyta została do tego celu funkcja `clCreateBuffer()` alokująca dla danego kontekstu odpowiednią porcję pamięci. Drugi parametr tej funkcji decyduje o sposobie dostępu do tej pamięci. W powyższym przykładzie będzie to obiekt, który można zarówno czytać jak i zapisywać (flaga o wartości `CL_MEM_READ_WRITE`).
- W liniach 39-40 wywołana jest funkcja `clEnqueueReadBuffer()`, która kolejkuje polecenie kopiowania pamięci *hosta* (`cpuVec`) do obiektu buforowego (`clVec`).
- W liniach 42–46 następuje właściwe wywołanie funkcji rdzenia. Najpierw w linii 42 za pomocą funkcji `clSetKernelArg()` ustawiana jest, dla danego kernela, wartość parametru jego wywołania.

- W linii 45 następuje zakolejkowanie właściwego wywołania rdzenia za pomocą funkcji `clEnqueueNDRangeKernel()` z podaniem rozmiarów globalnej i lokalnej grupy wątków (ang. *work group*), będących odpowiednikiem bloków CUDA.
- W linii 48 następuje zakolejkowanie kopiowania danych z obiektu buforowego z powrotem do pamięci *hosta* za pomocą funkcji `clEnqueueReadBuffer()`.
- W linii 50 funkcja `clFinish()` blokuje aktualny wątek *hosta* do czasu wykonania wszystkich zakolejkowanych poleceń.
- W linii 52 usuwany jest za pomocą klasycznego operatora **`delete`** wektor `cpuVec` a w linii 53 usuwany jest obiekt buforowy `clVec` za pomocą funkcji `clReleaseMemObject()`.

1.3.4. Analiza czasu wykonania

Przeanalizujmy czasy wykonania programów we wszystkich trzech środowiskach. Nie jest tu istotny czas inicjalizacji wektora i wypełnienie go liczbami pseudolosowymi. Istotny będzie jedynie czas obliczeń, czas tworzenia kontekstu i czas transferu pamięci dla programów GPGPU. Czasy te dla rozważanych środowisk zostały zebrane w Tabeli 1.1. Test został przeprowadzony na procesorze Intel QuadCore Q8200 2.33GHz oraz karcie grafiki NVidia GeForce GTS250.

Tabela 1.1. Czasy wykonania programu *Kwadrat wektora* dla poszczególnych środowisk

CPU[s]	CUDA[s]	OpenCL[s]
0.75	0.67	0.69

Wyraźnie widać, że zysk wykorzystania procesora GPU jest niewielki a różnica pomiędzy środowiskami CUDA i OpenCL jest w granicach błędu pomiarowego.

Analizując poszczególne kroki realizacji obliczeń GPGPU można dokładniej określić przyczynę niewielkiego zysku czasowego. Tabela 1.2 zawiera zestawienie czasowe dla inicjalizacji środowiska, transferu danych pomiędzy *hostem* a kartą grafiki oraz samego wykonania rdzenia. Inicjalizacja środowiska CUDA jest ukryta w pierwszym wywołaniu jakiejkolwiek funkcji API CUDA.

Po przeanalizowaniu tabeli jasnym staje się fakt, że cały zysk z równoległego wykonania obliczeń na GPU trwających około 0.02 sek (w porównaniu do 0.75 sek dla CPU) został stracony przez czas transferu danych do kar-

Tabela 1.2. Czasy wykonania poszczególnych etapów programu *Kwadrat wektora*

Funkcja	CUDA[s]	OpenCL[s]
Inicjalizacja		0.03
Kopiowanie *host→device*	0.33	0.32
Wykonanie kernela	0.019	0.02
Kopiowanie *device→host*	0.032	0.32

ty grafiki i z powrotem. Dla 500MB danych trwało to łącznie ponad 0.64 sek co stanowi około 95% całkowitego czasu wykonania. W specyficznych przypadkach czas ten można skrócić używając pamięci zabezpieczonej przed stronicowaniem (ang. *page-locked lub pinned*), jednakże jej użycie wiąże się z dodatkowymi kosztami i nie zawsze będzie możliwe.

1.3.5. Podsumowanie

Omawiany przykład, pomimo swojej prostoty, dosyć wyraźnie uwypukla zalety, wady oraz problemy z jakimi spotyka się programista podczas programowania równoległego z wykorzystaniem procesorów graficznych. Bardzo prosty rdzeń obliczeniowy, wysoce zrównoleglony został wykonany prawie 40 razy szybciej w porównaniu z wersją szeregową, nawet jeżeli weźmiemy pod uwagę, że pojedynczy rdzeń procesora *hosta* ma dużo większą moc obliczeniową w porównaniu z pojedynczym elementem realizującym wątek na procesorze graficznym. Z przykładu wyraźnie widać również główny problem programowania GPGPU, a mianowicie koszt transferu danych pomiędzy *hostem* i urządzeniem obliczeniowym. Generalnie, dla uzyskania jak najlepszej wydajności, ważnym jest aby jak najbardziej minimalizować operacje alokacji i kopiowania pamięci.

Nie bez znaczenia jest również zwiększony stopień skomplikowania samego programu oraz wykorzystywanych narzędzi. Poza samym problemem zrównoleglenia algorytmu liczącego, zakodowanego w języku c-podobnym, dochodzą dodatkowe funkcje inicjalizacji urządzenia, alokacji odpowiednich struktur po stronie GPU, funkcje transferu danych oraz samego wykonania rdzenia. O ile kod programu napisanego szeregowo w języku C zawierał się w około 20 liniach, to ta sama funkcjonalność w środowisku CUDA zajęła już prawie 40 linii a w środowisku OpenCL już ponad 60 linii. Środowisko CUDA dzięki temu, że jest dedykowane do konkretnej platformy sprzętowej może w dużym stopniu odciążyć programistę z samego procesu inicjalizacji i obsługi dedykowanego sprzętu. OpenCL, jednakże poprzez skomplikowanie tego procesu zyskuje na uniwersalności i elastyczności. Dodatkowo kod źródłowy CUDA, a przynajmniej funkcjonalność wywoływania rdzenia, mu-

si być kompilowany przy użyciu dedykowanego kompilatora *nvcc*, co może utrudniać proces integracji z resztą oprogramowania pisaną w jednym z popularnych języków (C/C++, JAVA, C#).

1.4. Proces kompilacji

Dla obu środowisk kod aplikacji jest dzielony na część CPU (*hosta*) i część GPU (*urządzenia*). Kompilacja części CPU jest realizowana za pomocą systemowego kompilatora natomiast część GPU, czyli kernel, w obu przypadkach jest najpierw tłumaczona na kod maszyny wirtualnej a w następnym kroku do postaci kodu binarnego zrozumiałego dla konkretnego urządzenia. Ten swoisty „wirtualny assembler" został wprowadzony w celu odseparowania sposobu realizacji obliczeń od ich sprzętowej realizacji. Karty graficzne ewoluują bowiem w szybkim tempie, często nie zachowując binarnej kompatybilności wstecznej, dodawane są również nowe funkcjonalności.

Sam proces kompilacji części GPU jest przeprowadzany w następujący sposób:

CUDA

NVIDIA wraz ze swoim pakietem dostarcza kompilator o nazwie *nvcc*. Pliki źródłowe kompilowane za pomocą tego narzędzia mogą zawierać mieszankę kodu *hosta* i kodu *urządzenia*.

Podczas kompilacji *nvcc* oddziela kod *hosta* od kodu wykonywanego na GPU a cały proces przebiega w kilku krokach:

— w kodzie *hosta* nowa składnia (<<<...>>>) zastępowana jest przez szereg wywołań funkcji, które ładują i uruchamiają kernel. Tak zmodyfikowany kod jest następnie kompilowany przez systemowy kompilator;
— kod *urządzenia* jest kompilowany do do postaci asemblera, zwanej *PTX (Parallel Thread Execution)* lub do postaci binarnej;
— postać binarna programu *urządzenia* jest linkowana do kodu *hosta*.

W przypadku kompilacji części GPU do postaci *PTX*, kod asemblera jest kompilowany w czasie działania programu do postaci binarnej przez sterownik urządzenia. Ten proces jest nazywany *just-in-time compilation*. Taka kompilacja zwiększa co prawda czas uruchomienia programu ale za cenę wykorzystania nowych możliwości sterownika.

Sam wirtualny asembler *PTX* nie jest w pełni przenośny i jest kompatybilny w obrębie danej wersji *Compute Capability* oraz wersji wyższych.

Przez podanie opcji `-arch` kompilatora *nvcc* można wymusić kompilację pod konkretną wersję *Compute Capability*.

OpenCL

Środowisko OpenCL do kompilacji całości kodu źródłowego do postaci wykonywalnej wymaga jedynie systemowego kompilatora i dołączenia biblioteki `OpenCL` w procesie linkowania. Kod programu zawierający funkcje *urządzenia* jest kompilowany już w momencie wykonywania się programu, za pomocą wbudowanego kompilatora OpenCL, do binarnej postaci zależnej od urządzenia. Ten dynamiczny proces kompilacji (ang. *runtime compilation*) składa się z dwóch etapów:

1) Kod źródłowy jest kompilowany do postaci *IR (Intermediate Representation)*, będącej asemblerem maszyny wirtualnej przez tzw. *Front-End compiler*. Ten etap jest nazywany kompilacją offline (ang. *offline compilation*);
2) *IR* jest kompilowany do postaci wykonywalnej danego urządzenia przez tzw. *Back-End compiler*. Tan etap jest nazywany kompilacją online (ang. *online compilation*).

Proces kompilacji offline może być przeprowadzony wcześniej, przed uruchomieniem właściwej aplikacji, a podczas jej działania może być załadowany plik zawierający już skompilowany do *IR* kod kerneli. Funkcjonalność tą realizuje funkcja:

```
cl_program clCreateProgramWithBinary(cl_context context,
           cl_uint num_devices,
           const cl_device_id *device_list,
           const size_t *lengths,
           const unsigned char **binaries,
           cl_int *binary_status, cl_int *errcode_ret)
```

Program do postaci *IR* musi jednak zostać wcześniej skompilowany w klasyczny sposób. Kod binarny można uzyskać posługując się funkcją:

```
cl_int clGetProgramInfo(cl_program program,
                        cl_program_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```

po uprzednim skompilowaniu programu.

Należy pamiętać, że kod *IR* mimo wszystko, wciąż nie jest przenośny i w dużej mierze jest zależny od urządzenia, na którym został wygenerowany.

1.5. Obsługa błędów

W przypadku obu środowisk obsługa błędów leży po stronie programisty. Zastosowany został tu standardowy sposób informowania o stanie wywołania funkcji poprzez zwracanie kodu błędu w postaci wartości zwracanej przez funkcję lub w postaci referencji do zmiennej przekazanej w parametrze wywołania.

1.5.1. CUDA

Praktycznie każda funkcja API CUDA zwraca wartość błędu typu:

```
cudaError_t errcode
```

Typ `cudaError_t` jest typem wyliczeniowym. W przypadku sukcesu danej funkcji przyjmuje on wartość `cudaSuccess`, w przypadku niepowodzenia jedną z pozostałych wartości (pełna lista możliwych wartości kodów zobacz [**6**]).

Dodatkowo, funkcja:

```
cudaError_t cudaGetLastError(void)
```

zwraca ostatni status, który został zwrócony przez którąkolwiek z wywołanych funkcji dla danego wątku hosta. Funkcja ta jednocześnie ustawia aktualną wartość statusu błędu na `cudaSuccess`. Bliźniacza funkcja:

```
cudaError_t cudaPeekAtLastError(void)
```

realizuje analogiczną funkcjonalność nie resetując jednak wartości aktualnego statusu błędu.

Kod błędu można przedstawić w formie napisu dzięki funkcji:

```
const char* cudaGetErrorString(cudaError_t)
```

zwracającej string z opisem danego błędu.

Przykładowe użycie powyższych funkcji zostało przedstawione na listingu 1.7.

Listing 1.7. CUDA – Obsługa błędów.

```
1 #include <cuda_runtime_api.h>
2 #include <iostream>
3
4 int main(int argv, char *argc[])
5 {
6   ...
7   cudaError_t status;
8   float *mem;
```

```
9
10   status = cudaMalloc((void**)mem, sizeof(float)*size);
11
12   if(status != cudaSuccess)
13     std::cout << cudaGetErrorString(status) << std::endl;
14   ...
15 }
```

1.5.2. OpenCL

Funkcje API środowiska OpenCL informacje o stausie własnego wykonania zwracają w postaci wartości zwracanej przez daną funkcję lub zapisując kod błędu w przekazanym przez wskaźnik parametrze. Status wykonania funkcji jest typu:

```
cl_int errcode
```

Wartość odpowiadająca prawidłowemu wykonaniu funkcji została zdefiniowana przez nazwę `CL_SUCCESS` i ma numeryczną wartość 0. Wszystkie wartości błędów wraz z ich kodami zostały zdefiniowane w pliku nagłówkowym `cl.h`. OpenCL nie oferuje żadnego mechanizmu pamiętania otatniego zwróconego statusu i nie posiada funkcji konwertującej kod błędu na jej opisową formę. Na listingu A.4 w dodatku A została zdefiniowana metoda:

```
const char* clErrorString(cl_int)
```

realizująca taką funkcjonalność.

Przykłady wykorzystania kodów błędów zostały zobrazowane na listingu 1.8.

Listing 1.8. OpenCL – Obsługa błędów.

```
1  #include <CL/opencl.h>
2
3  int main(int argv, char *argc[])
4  {
5    cl_int errcode;
6    cl_platform_id platform;
7    errcode = clGetPlatformIDs(1, &platform, NULL);
8    cout << "Platform: " << clErrorString(errcode) << endl;
9
10   cl_device_id devices;
11   cl_uint num_dev;
12   errcode = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
13                            1, &devices, &num_dev);
14   cout << "Device: " << clErrorString(errcode) << endl;
15
```

```
16   cl_context context = clCreateContext(0, 1, &devices,
17                         NULL, NULL, &errcode);
18   cout << "Context: " << clErrorString(errcode) << endl;
19   ...
20 }
```

W przypadku funkcji asynchronicznych kod błędu nie jest zwracany zaraz po powrocie z funkcji a dopiero po jej faktycznym wykonaniu.

1.6. Uzyskiwanie informacji o urządzeniach, obiektach i stanie kompilacji

1.6.1. CUDA

Większość informacji o dostępnych urządzeniach zgodnych z technologią CUDA oraz z samą platformą jest dostępna z wysokopoziomowego API (*Runtime API*) poprzez kilka funkcji:

1) `cudaError_t cudaRuntimeGetVersion(int* runtimeVersion)`
2) `cudaError_t cudaDriverGetVersion(int* driverVersion)`
3) `cudaError_t cudaGetDeviceCount(int* count)`
4) `cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp* prop, int device)`
5) `cudaError_t cudaMemGetInfo(size_t* free, size_t* total)`
6) `cudaError_t cudaFuncGetAttributes(struct cudaFuncAttributes* attr, const char* func)`
7) `cudaError_t cudaPointerGetAttributes(struct cudaPointerAttributes* attributes, void* ptr)`

Dwie pierwsze funkcje zwracają numer wersji, odpowiednio wysokopoziomowego API (*runtime API*) i niskopoziomowego API (*driver API*).

Trzecia funkcja zwraca ilość urządzeń kompatybilnych z dowolną wersją CUDA.

Czwarta funkcja, dla konkretnego urządzenia, zwraca strukturę `cudaDeviceProp` zawierającą szereg składowych opisujących własności tego urządzenia (pełna lista tych własności zobacz w [6]).

Piąta funkcja zwraca rozmiar całkowitej (`total`) oraz dostępnej (`free`) pamięci aktualnego urządzania.

Szósta i siódma funkcja umożliwiają uzyskanie dodatkowych informacji odpowiednio o danym kernelu lub obiekcie znajdującym się w pamięci urządzenia.

Listing 1.9 pokazuje sposób uzyskania podstawowych danych o urządzeniu za pomocą tych funkcji.

Listing 1.9. CUDA – Uzyskiwanie informacji o urządzaniach.

```
#include <cuda_runtime.h>
#include <iostream>

int main()
{
  int ver;
  cudaDriverGetVersion(&ver);
  cout << "Driver version: " << ver << endl;
  cudaRuntimeGetVersion(&ver);
  cout << "Runtime version: " << ver << endl;

  int dev_co;
  cudaGetDeviceCount(&dev_co);
  cout << "Device count: " << dev_co << endl;

  int dev_no = 0;
  cudaSetDevice(dev_no);
  cudaDeviceProp prop;
  cudaGetDeviceProperties (&prop, dev_no);

  cout << "Device name: " << prop.name << endl;
  cout << "Device compute capability: " << prop.major <<
          "." << prop.minor << endl;
  cout << "Multiprocessor count: " <<
          prop.multiProcessorCount << endl;
  cout << "Total global mem: " <<
          prop.totalGlobalMem/1024/1024 << " MB" << endl;
  cout << "Max threads per MProcessor: " <<
          prop.maxThreadsPerMultiProcessor << endl;

  size_t total_mem, free_mem;
  cudaMemGetInfo(&free_mem, &total_mem);
  cout << "Total mem: " << total_mem/1024/1024 << " MB" <<
          endl << "Free mem: " << free_mem/1024/1024 <<
          " MB" << endl;

  return 0;
}
```

W wyniku działania powyższy program wypisze na konsoli następujące informacje (lub odpowiednie dla danego urządzenia):

```
Driver version: 4000
Runtime version: 4000
Device count: 1
Device name: GeForce GTS 250
Device compute capability: 1.1
Multiprocessor count: 16
Total global mem: 1023 MB
```

```
Max threads per MProcessor: 768
Total mem: 1023 MB
Free mem: 785 MB
```

Niskopoziomowe API (*Driver API*) dostarcza analogicznych funkcji umożliwiających uzyskanie informacji o urządzeniach, obiektach i funkcjach:

1) `CUresult cuDriverGetVersion(int* driverVersion)`
2) `CUresult cuDeviceGetAttribute(int* pi, CUdevice_attribute attrib, CUdevice dev)`
3) `CUresult cuDeviceGetCount(int* count)`
4) `CUresult cuDeviceGetName(char* name, int len, CUdevice dev)`
5) `CUresult cuDeviceGetProperties(CUdevprop* prop, CUdevice dev)`
6) `CUresult cuDeviceTotalMem(size_t* bytes, CUdevice dev)`
7) `CUresult cuMemGetInfo(size_t* free, size_t* total)`
8) `CUresult cuPointerGetAttribute(void* data, CUpointer_attribute attribute, CUdeviceptr ptr)`
9) `CUresult cuFuncGetAttribute(int* pi, CUfunction_attribute attrib, CUfunction hfunc)`

1.6.2. OpenCL

W środowisku OpenCL istnieje szereg funkcji, który nazwy zakończone są słowem `Info`, służących do uzyskiwania informacji o używanych urządzeniach lub obiektach. Jest to również jedyny sposób uzyskania informacji o stanie i ewentualnych błędach kompilacji programu OpenCL. Pełna lista tych funkcji obejmuje:

1) `clGetPlatformInfo()`
2) `clGetDeviceInfo()`
3) `clGetContextInfo()`
4) `clGetMemObjectInfo()`
5) `clGetImageInfo()`
6) `clGetSamplerInfo()`
7) `clGetProgramInfo()`
8) `clGetProgramBuildInfo()`
9) `clGetKernelInfo()`
10) `clGetKernelWorkGroupInfo()`
11) `clGetEventInfo()`
12) `clGetEventProfilingInfo()`

W każdym przypadku, do funkcji podany zostaje w parametrze badany obiekt, nazwa badanego parametru i jego dopuszczalny rozmiar w bajtach, natomiast zwracane są przez referencje: wartość tego parametru oraz jego wielkość rzeczywista w bajtach.

Poniższy przykład ilustruje sposób wykorzystania kilku takich funkcji do uzyskania informacji o platformie, używanym urządzeniu oraz statusie kompilacji programu kernela:

Listing 1.10. OpenCL – Uzyskiwanie informacji o obiektach.

```
#include <CL/opencl.h>
#include <iostream>

int main(int argv, char *argc[])
{
  cl_int errcode;
  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);

  const int info_size = 10240;
  char info[info_size];
  clGetPlatformInfo(platform, CL_PLATFORM_PROFILE,
                    info_size, info, NULL);
  cout << "Platform profile: " << info << endl;
  clGetPlatformInfo(platform, CL_PLATFORM_VERSION,
                    info_size, info, NULL);
  cout << "Platform version: " << info << endl;
  clGetPlatformInfo(platform, CL_PLATFORM_NAME,
                    info_size, info, NULL);
  cout << "Platform name: " << info << endl;
  clGetPlatformInfo(platform, CL_PLATFORM_VENDOR,
                    info_size, info, NULL);
  cout << "Platform vendor: " << info << endl;
  clGetPlatformInfo(platform, CL_PLATFORM_EXTENSIONS,
                    info_size, info, NULL);
  cout << "Platform extensions: " << info << endl;

  cl_device_id devices;
  cl_uint num_dev;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
                           1, &devices, &num_dev);

  clGetDeviceInfo(devices, CL_DEVICE_NAME,
                  info_size, info, NULL);
  cout << "Device name: " << info << endl;
  clGetDeviceInfo(devices, CL_DEVICE_VENDOR,
                  info_size, info, NULL);
  cout << "Device vendor: " << info << endl;
  clGetDeviceInfo(devices, CL_DEVICE_VERSION,
                  info_size, info, NULL);
  cout << "Device version: " << info << endl;
  cl_uint comp_units;
  clGetDeviceInfo(devices, CL_DEVICE_MAX_COMPUTE_UNITS,
                  sizeof(cl_uint), &comp_units, NULL);
  cout << "Device compute units: " << comp_units << endl;
```

```
46  cl_ulong mem_size;
47  clGetDeviceInfo(devices, CL_DEVICE_GLOBAL_MEM_SIZE,
48                  sizeof(cl_ulong), &mem_size, NULL);
49  cout << "Device global memory size: " <<
50          mem_size/1024/1024 << "MB" << endl;
51  clGetDeviceInfo(devices, CL_DEVICE_NAME,
52                  info_size, info, NULL);
53  cout << "Device name: " << info << endl;
54
55  ...
56
57  errcode = clBuildProgram(program, 0, 0, 0, 0, 0);
58  cout << "Program build:"<< clErrorString(errcode)<< endl;
59
60  clGetProgramBuildInfo(program, devices,
61                        CL_PROGRAM_BUILD_LOG, info_size,
62                        info, NULL);
63  cout << "Program build log: " << info << endl;
64  ...
```

W liniach 10–26 korzystając z funkcji `clGetPlatformInfo()` zostały wypisane istotne parametry platformy, takie jak jej profil, wersja, nazwa, dostawca oraz możliwe rozszerzenia.

Analogicznie, w liniach 28–53 tworzony jest obiekt urządzenia i za pomocą funkcji `clGetDeviceInfo()` pobierane są wybrane parametry tego urządzenia.

W liniach 57–63 pokazany jest sposób uzyskania informacji o stanie kompilacji i budowy programu. Funkcja `clGetProgramBuildInfo()` jest jedną z najczęściej wykorzystywanych funkcji informacyjnych ponieważ kompilator OpenCL w żaden inny sposób nie może powiadomić o ewentualnych błędach podczas budowy programu.

W wyniku działania powyższy program wypisze na konsoli następujące informacje (lub odpowiednie dla danego urządzenia):

```
Platform: CL_SUCCESS
Platform profile: FULL_PROFILE
Platform version: OpenCL 1.0 CUDA 4.0.1
Platform name: NVIDIA CUDA
Platform vendor: NVIDIA Corporation
Platform extensions: cl_khr_byte_addressable_store cl_khr_icd
                     cl_khr_gl_sharing cl_nv_compiler_options
                     cl_nv_device_attribute_query
                     cl_nv_pragma_unroll

Device: CL_SUCCESS
Device name: GeForce GTS 250
Device vendor: NVIDIA Corporation
Device version: OpenCL 1.0 CUDA
```

```
Device compute units: 16
Device global memory size: 1023MB
Device name: GeForce GTS 250

Program build: CL_BUILD_PROGRAM_FAILURE
Build log: <program source>:4:25: error: use of undeclared
identifier 'vect'
    vec[i] = vec[i] * vect[i];
                      ^
```

W budowanym kernelu (z listingu 1.6) celowo została popełniona literówka zamieniająca nazwę zmiennej `vec` na nieistniejącą nazwę `vect` w linii 4.

1.7. Integracja CUDA z językiem C/C++

Specyficzne własności środowiska CUDA wymagają użycia dedykowanego kompilatora kodu GPU o nazwie *nvcc* dostarczanego w pakiecie NVIDIA CUDA. Kod CPU jest kompilowany przy użyciu standardowego kompilatora systemowego. NVIDIA wspiera tu kompilatory: GCC dla platformy Linux, Microsoft Visual C compiler dla platformy MS Windows oraz GCC/Xcode dla Mac OS X.

Kompilator *nvcc* potrafi kompilować obie części kodu źródłowego używając systemowego kompilatora dla części CPU. O ile przy prostszych rozwiązaniach użycie tego narzędzia jest wystarczające o tyle przy większych projektach niezbędna jest kompilacja całości kodu CPU przez właściwy kompilator i osobno część GPU przez kompilator *nvcc* do postaci binarnej a następnie linkowanie poszczególnych części za pomocą linkera. Poniższy przykład pokazuje sposób postępowania przy mieszaniu standardowego oprogramowania, kompilowanego przez GCC z kodem GPU.

Listing 1.11. CUDA – Integracja, część CPU, plik main.cpp.

```
1  #include <iostream>
2
3  extern void cuda_function(float* a, int b);
4
5  int main(int argv, char **argc)
6  {
7    int size = 256;
8    float *pmem = new float[size];
9
10   cuda_function(pmem, size);
11
12   for(int i=0; i<size; i++)
13     std::cout << pmem[i] << "; ";
14 }
```

Celem tego prostego programu jest wywołanie funkcji wykonywanej na GPU wypełniającej 256 elementową tablicą typu `float` wartościami równymi indeksom tablicy. Funkcja ta została zadeklarowana w linii 3 jako zewnętrzna i została zdefiniowana na listingu 1.12.

Listing 1.12. CUDA – Integracja, część GPU - cuda.cu.

```
1 #include <cuda_runtime_api.h>
2 #include "kernel.cu"
3
4 extern void cuda_function(float* hmem, int size)
5 {
6   float * dmem;
7   cudaMalloc((void**)&dmem, size*sizeof(float));
8   kernel<<<1, size>>>(dmem);
9   cudaMemcpy(hmem, dmem, size*sizeof(float),
10              cudaMemcpyDeviceToHost);
11  cudaFree(dmem);
12 }
```

W pliku `cuda.cu` zawarta została część kodu realizowana przez GPU, włącznie z wywołaniem funkcji kernela zdefiniowanej w pliku `kernel.cu` na listingu 1.13.

Listing 1.13. CUDA – Integracja, część GPU, plik kernel.cu.

```
1 __global__ void kernel(float* a)
2 {
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4     a[i] = i;
5 }
```

Część CPU stanowi tylko plik `main.cpp` i tylko on może zostać skompilowany przy użyciu systemowego kompilatora. Dla GCC będzie to równoznaczne z wydaniem polecenia:

```
g++ -c main.cpp
```

tworzącym odpowiedni plik obiektowy `main.o`.

Część GPU stanowi plik `cuda.cu` oraz plik `kernel.cu` zawierający definicję rdzenia dołączany do pliku `cuda.cu`. Ten plik musi zostać skompilowany przy użyciu kompilatora *nvcc* przez wydanie polecenia:

```
nvcc -c cuda.cu
```

generującego plik binarny `cuda.o`. W celu linkowania obu plików binarnych należy użyć polecenia:

```
g++ -o cpp_integration -lcudart main.o cuda.o
```

generującego wykonywalny plik o nazwie `cpp_integration`.

Używając niskopoziomowego API (*Driver API*) można cały program skompilować przy użyciu systemowego kompilatora. Niezbędne jest jednak odpowiednie przygotowanie funkcji kernela w postaci modułu. Moduł akceptowany przez niskopoziomowe funkcje API musi zostać skompilowany przez *nvcc* do postaci *cubin* lub *PTX*. Poniższy listing przedstawia program o identycznej funkcjonalności z programem z listingu 1.11.

Listing 1.14. CUDA – Niskopoziomowa część CPU, plik main.cpp.

```
1  #include <iostream>
2  #include <cuda_runtime.h>
3  #include <cuda.h>
4
5  int main(int argv, char **argc)
6  {
7    int size = 256;
8    float *pmem = new float[size];
9
10   CUdevice   hDevice;
11   CUcontext  hContext;
12   CUmodule   hModule;
13   CUfunction hFunction;
14   CUdeviceptr pDeviceMem;
15
16   cuInit(0);
17   cuDeviceGet(&hDevice, 0);
18   cuCtxCreate(&hContext, 0, hDevice);
19
20   cuModuleLoad(&hModule, "kernel.cubin");
21   cuModuleGetFunction(&hFunction, hModule, "kernel");
22
23   cuMemAlloc(&pDeviceMem, size*sizeof(float));
24   cuMemcpyHtoD(pDeviceMem, pmem, size*sizeof(float));
25
26   void *args[] = {&pDeviceMem};
27   cuLaunchKernel(hFunction, 1, 1, 1, size, 1, 1,
28                  0, NULL, args, NULL);
29
30   cuMemcpyDtoH((void*)pmem, pDeviceMem,
31                size*sizeof(float));
32   cuMemFree(pDeviceMem);
33
34   for(int i=0; i<size; i++)
35     std::cout << pmem[i] << "; ";
36
```

```
37   return 0;
38 }
```

Programując CUDA w niskopoziomowym API, na wzór OpenCL, niezbędna jest inicjalizacja sterownika funkcją `cuInit()` w linii 16 oraz utworzenie urządzenia i kontekstu w liniach 17-18.

Moduł kernela jest ładowany przez funkcję `cuModuleLoad()` w linii 20 z pliku `"kernel.cubin"` i będzie wskazywany uchwytem `hModule`. W następnej linii z modułu wyłuskiwana jest konkretna funkcja kernela o nazwie `"kernel"`.

Samo wywołanie kernela odbywa się w linii 27 przez wywołanie funkcji `cuLaunchKernel()`. Funkcja ta potrzebuje uchwytu do funkcji kernelu `hFunction`, określenia wielkości siatki i bloku wątków, wielkości pamięci współdzielonej, ewentualnego strumienia oraz zestawu parametrów kernela. W powyższym przykładzie wielkość pamięci współdzielonej została ustalona na 0 bajtów. Nie został również podany żaden strumień. Zestaw parametrów można podać na dwa sposoby: albo za pomocą parametru `kernelParams` albo parametru `extra`. W prostszym przypadku, tj. za pomocą `kernelParams`, wszystkie parametry kernelu muszą zostać zebrane w tablicę wskaźników ustawionych na kolejne parametry funkcji kernela. W powyższym przykładzie zostało to wykonane w linii 26. Dokładne omówienie tej funkcji będzie w rozdziale 2.3.

Pozostaje jeszcze przygotowanie modułu. Kod źródłowy modułu jest przedstawiony na poniższym listingu:

Listing 1.15. CUDA – Kod źródłowy modułu, plik kernel.cu.

```
1 extern "C" __global__ void kernel(float* a)
2 {
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4     a[i] = i;
5 }
```

Kompilacja części głównej CPU odbywa się za pomocą systemowego kompilatora:

```
g++ -lcudart -lcuda main.cpp
```

Natomiast przygotowanie modułu odbywa się za pomocą kompilatora *nvcc* tworzącego postać *cubin*:

```
nvcc --cubin kernel.cu
```

lub postać *PTX*:

```
nvcc --ptx kernel.cu
```

i generującego odpowiednio plik `kernel.cubin` lub `kernel.ptx`.

W ogólności, integracja kodu wykonywanego na GPU z istniejącym oprogramowaniem polega na oddelegowaniu całej funkcjonalności CUDA do osobnego modułu kompilowanego przy użyciu kompilatora *nvcc* a następnie linkowaniu tego modułu z resztą oprogramowania przy pomocy standardowych narzędzi.

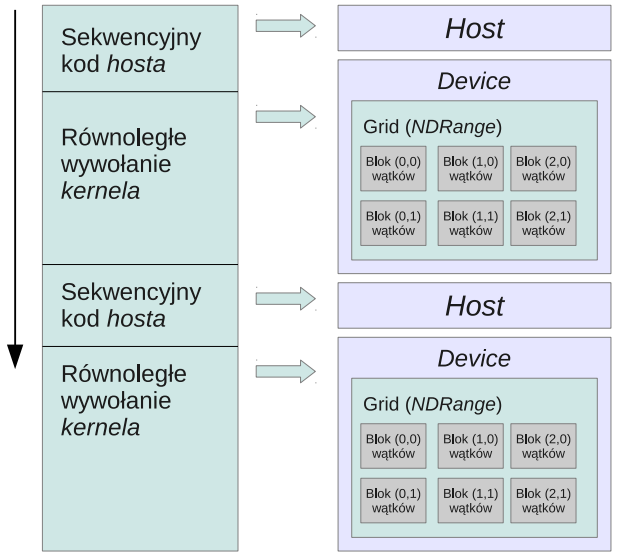
Rozdział 2

Architektura środowisk CUDA i OpenCL

2.1. Model wykonania

2.1.1. Kernel

W programowaniu heterogenicznym zrównoleglonym realizowanym za pomocą komputera klasy PC wyposażonego w dedykowaną kartę graficzną, część programu wykonywana na procesorze CPU jest nazywana częścią *hosta*, natomiast część wykonywana na karcie graficznej nazywanej *urządzeniem obliczeniowym* (ang. *computing device*) jest określana mianem *kernela* (zobacz Rysunek 2.1). Ściśle mówiąc, *kernel* jest pewną funkcją wykonywaną na dedykowanym urządzeniu (*device*) działającym w obrębie kontekstu na określonej porcji pamięci. Program *hosta* definiuje kontekst dla kernela i zarządza jego wykonywaniem oraz transferami danych pomiędzy pamięcią *hosta* a pamięcią *urządzenia*.



Rysunek 2.1. Wykonanie kodu sekwencyjnego na *hoście* i równoległego na *device*.

Funkcja kernela z reguły tworzy olbrzymią ilość wątków wykorzystując możliwość zrównoleglenia danych. W odróżnieniu od pojęcia wątku procesora CPU, wątki GPU są znacznie prostszymi tworami a ich tworzenie i zarządzanie zabiera niewielkie ilości cykli w porównaniu do ciężkich wątków CPU.

Typowe wykonanie programu CUDA/OpenCL jest przedstawione na rysunku 2.1. Całym procesem wykonywania programu steruje procesor *hosta* (CPU). Kiedy zostaje uruchomiona funkcja kernela, wykonywanie przenosi

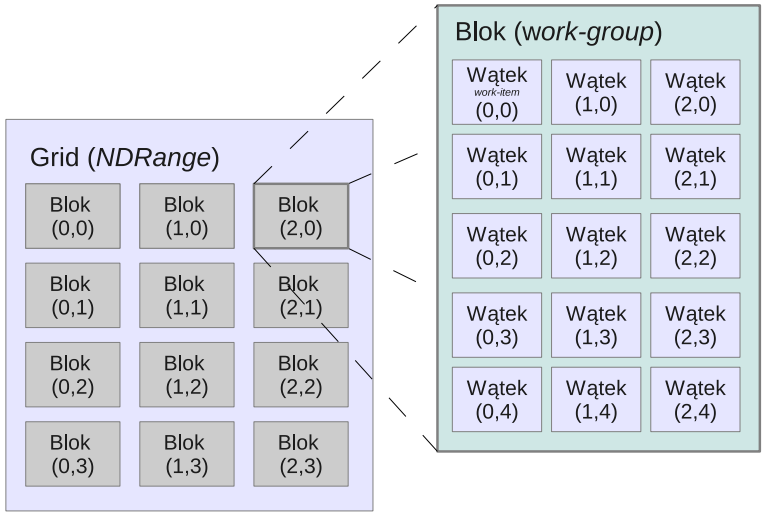
się do urządzenia GPU. W sytuacji, gdy wywołanie funkcji rdzenia odbywa się asynchronicznie, sterowanie powraca od razu do wątku *hosta*. W innym przypadku, wątek *hosta* jest blokowany i czeka na powrót sterowania do czasu zakończenia obliczeń wykonywanych na *urządzeniu*. Po uruchomieniu kernela, na karcie grafiki, tworzony jest zbiór wątków nazywanych *Siatką* (ang. *Grid*) w CUDA lub *NDRange* w OpenCL wykonywanych równolegle porcjami, w zależności od możliwości danego urządzenia obliczeniowego. Po wykonaniu funkcji kernela przez każdy z wątków *Grid/NDRange* jest usuwany.

2.1.2. Organizacja wątków

Pojęcie wątku w programowaniu GPGPU jest ściśle związane z pojęciem kernela. Dla obu rozważanych architektur pojedynczy wątek (w danej grupie wątków) wykonuje jednakowy kod funkcji rdzenia. W środowisku CUDA wątek nie ma specjalnej nazwy i jest nazywany po prostu wątkiem (ang. *thread*), natomiast w środowisku OpenCL pojedynczy wątek jest nazywany *work-item*.

Podczas wykonywania danego kernela tworzona jest przestrzeń indeksów wątków. Dana instancja kernela wykona się dokładnie jeden raz dla każdego punktu w tej przestrzeni. Sam indeks jest jedno-, dwu- lub trójwymiarowym wektorem definiującym całkowitą liczbę wątków, które zostaną wykonane podczas pojedynczego uruchomienia kernela. W środowisku CUDA przestrzeń ta została nazwana *Siatką* (ang. *Grid*), natomiast w środowisku OpenCL ma nazwę *NDRange*.

Pojedyncze wątki są organizowane w jedno-, dwu- lub trójwymiarowe grupy wątków o identycznym rozmiarze zwane blokami (ang. *blocks*) w CUDA lub *work-groups* w OpenCL. Pozwala to na dodatkową segmentację przestrzeni indeksów. Blok/*work-group* również posiada unikalny indeks w ogólnej przestrzeni wątków a poszczególne wątki/*work-items* mają przypisane unikalne lokalne indeksy wewnątrz danego bloku/*work-group*. Ilość bloków jest zwykle podyktowana ilością danych do przetworzenia. Rysunek 2.2 obrazuje omawianą organizację wątków dla przypadku dwuwymiarowego. Ilość wątków w pojedynczym bloku jest z góry ograniczona i oscyluje dla obecnej generacji kart graficznych w okolicy 1024. Ograniczenie to wynika głównie z założenia, że wszystkie wątki danego bloku muszą być wykonywane równocześnie w obrębie pojedynczej jednostki obliczeniowej (*computing unit/core*) oraz współdzielić ograniczoną ilość pamięci przypisanej do tej jednostki obliczeniowej. Tylko wątki należące do tego samego bloku mogą podlegać wzajemnej synchronizacji. Z kolei różne bloki mogą wykonywać się niezależnie, w dowolnej kolejności, równolegle lub szeregowo.



Rysunek 2.2. Schemat organizacji wątków dla dwuwymiarowego przypadku. Cała przestrzeń indeksów została podzielona na (3×4) bloków, w każdym bloku znajduje się (3×5) wątków.

Numer pojedynczego wątku z całej puli wątków w danej przestrzeni indeksów jest również wektorem jedno-, dwu- lub trójwymiarowym.

W środowisku OpenCL dostęp do takiej liczby identyfikującej jednoznacznie wątek (*work-item*) jest możliwy dzięki adresowaniu globalnemu, bowiem każdy *work-item* ma przypisany globalny identyfikator zwany *global ID* dostępny za pomocą funkcji `get_global_id(uint dim)` zwracającej indeks dla konkretnego wymiaru `dim`.

Rozmiar całej przestrzeni indeksów *NDRange* jest dostępny za pomocą funkcji `get_global_size(uint dim)`, ilość *work-groups* za pomocą funkcji `get_num_groups(uint dim)` a ilość *work-items* w *work-group* za pomocą funkcji `get_local_size(uint dim)`. Lokalny indeks *work-item* wewnątrz jego *work-group* jest zwracany przez funkcję `get_local_id(uint dim)`.

W środowisku CUDA obliczenie całkowitej ilości indeksów wymaga przemnożenia ilości bloków w siatce (zmienna wbudowana `gridDim`) przez ilość wątków w bloku, czyli rozmiar bloku (zmienna wbudowana `blockDim`). Na przykład dla współrzędnej x będzie to wyrażenie:

$$n\_index.x = gridDim.x \times blockDim.x$$

Obliczenie globalnego numeru wątku w CUDA wymaga przemnożenia indeksu aktualnego bloku (zmienna wbudowana `blockIdx`) przez rozmiar blo-

ku (`blockDim`) i dodanie aktualnego indeksu wątku w tym bloku (zmienna wbudowana `threadIdx`). Dla współrzędnej x będzie to wyrażenie:

$$index.x = blockIdx.x \times blockDim.x + threadIdx.x$$

Tabela 2.1 zawiera zestawienie zmiennych/funkcji indeksujących wątki dla obu środowisk.

Tabela 2.1. Odpowiedniki indeksacji wątków w środowisku CUDA i OpenCL. Przykład jedynie dla współrzędnej x.

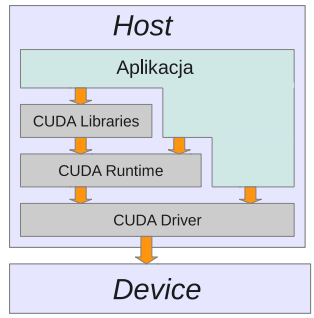
Opis	CUDA	OpenCL
Rozmiar przestrzeni indeksów	$gridDim.x \times blockDim.x$	`get_global_size(0)`
Globalny indeks wątku/*work-item*	$blockIdx.x \times blockDim.x + threadIdx.x$	`get_global_id(0)`
Rozmiar pojedynczego bloku/*work-group*	$blockDim.x$	`get_local_size(0)`
Lokalny indeks wątku/*work-item*	$threadIdx.x$	`get_local_id(0)`
Ilość bloków/ *work-goups*	$gridDim.x$	`get_num_groups(0)`
Numer bloku/ *work-goup*	$blockIdx.x$	`get_group_id(0)`

2.2. Programowanie wysokopoziomowe CUDA

Środowisko CUDA dostarcza dwóch interfejsów programistycznych: niskopoziomowego, nazywanego *CUDA driver API* oraz wysokopoziomowego zwanego *CUDA runtime API* (zobacz rysunek 2.3).

Driver API jest zbiorem funkcji języka C dających pełną kontrolę nad przebiegiem sterowania programem CUDA. W tym przypadku to programista dba o poprawną inicjalizację środowiska, utworzenie niezbędnych kontekstów i urządzeń oraz zarządza modułami kerneli. *Runtime API* stanowi nadbudowę wyższego poziomu uproszczając standardowe procedury zarządzania środowiskiem. Programista może korzystać dowolnie z obu interfejsów oddzielnie lub mieszając ich wywołania ze sobą.

Implementacja funkcji wysokopoziomowych *runtime API* została zebrana w bibliotece `cudart`. Wszystkie wywołania tego API rozpoczynają się przedrostkiem `cuda`.



Rysunek 2.3. Budowa warstwowa architektury CUDA API.

2.2.1. Konfiguracja urządzeń

Inicjalizacja środowiska zachodzi niejawnie podczas pierwszego wywołania dowolnej funkcji API. Podczas inicjalizacji jest tworzony kontekst dla każdego urządzenia znajdującego się w systemie zgodnego z technologią CUDA. Kontekst ten staje się głównym kontekstem CUDA i w jego obrębie następuje kolejkowanie wywołań funkcji API. Wywołanie funkcji

```
cudaError_t cudaDeviceReset()
```

niszczy aktualny główny kontekst.

Wybór konkretnego urządzenia, na którym będą przeprowadzane obliczenia, jest możliwy za pomocą funkcji:

```
cudaError_t cudaSetDevice(int device)
```

ustawiającego urządzenie `device` jako aktualnie wybrane. Od momentu wywołania tej funkcji wszystkie funkcje zarządzające pamięcią urządzenia oraz wywołujące funkcje rdzenia będą realizowane za pomocą tego urządzenia. W przypadku gdy nie zostanie wywołana powyższa funkcja, urządzenie o numerze 0 staje się aktualnie wybranym.

Informacji o urządzeniach zainstalowanych w systemie dostarczają funkcje:

```
cudaError_t cudaGetDeviceCount(int* count)
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp*
                                    prop, int device)
```

zwracające odpowiednio ilość urządzeń kompatybilnych z CUDA oraz strukturę opisującą wybrane urządzenie. Dokładna specyfikacja struktury `cudaDeviceProp` jest dostępna w specyfikacji języka CUDA [6].

2.2.2. Wywołanie kernela

CUDA rozszerza składnię ANSI C o kilka dodatkowych słów kluczowych. Każda deklarowana funkcja musi być poprzedzona specyfikatorem `__global__`, `__device__` lub `__host__`.

Funkcja zadeklarowana jako `__host__` jest klasyczną funkcją wykonywaną tylko po stronie *hosta* i wywoływaną z innej funkcji *hosta*. Domyślnie wszystkie funkcje, które nie mają jawnie podanego specyfikatora traktowane są jako funkcje `__host__`. Ma to niewątpliwie uzasadnienie w przenośności i migracji istniejącego oprogramowania do środowiska CUDA.

Funkcje zadeklarowane jako `__global__` wskazują właściwą funkcję kernela, która może być wykonana tylko na *urządzeniu* i może być wywołana tylko z poziomu *hosta*, tworząc jednocześnie siatkę wątków na urządzeniu obliczeniowym.

Funkcja zadeklarowana jako `__device__` jest funkcją, która może być wykonana tylko na *urządzeniu* i może być wywołana tylko z poziomu funkcji rdzenia lub innej funkcji zadeklarowanej jako `__device__`.

Istnieje również możliwość zadeklarowania funkcji jako `__host__ __device__`. W takim przypadku kompilator generuje dwie wersje funkcji mogącej wykonywać się na *hoście* i na *urządzeniu*.

Funkcja zadeklarowana jako `__global__` może zostać wywołana z poziomu *hosta* za pomocą nowej składni:

```
kernel_function<<< Dg, Db, Ns, S >>>(...)
```

gdzie:

— `kernel_function` jest nazwą wywoływanej funkcji ze specyfikatorem `__global__`
— `Dg` jest typu `dim3` i określa rozmiar siatki, tj. ilość bloków składających się na tę siatkę,
— `Db` jest typu `dim3` i określa wielkość bloku,
— `Ns` jest typu `size_t` i określa ilość pamięci współdzielonej alokowanej dynamicznie dla każdego bloku. Ten parametr jest opcjonalny i ma wartość domyślną równą 0,
— `S` jest typu `cudaStream_t` i określa skojarzony strumień. Ten parametr jest opcjonalny i ma wartość domyślną równą 0.

W nawiasach okrągłych `(...)` wywołania funkcji rdzenia podawane są klasyczne parametry wywołania funkcji.

Listing 2.1 ilustruje deklarowanie wielkości siatki i bloku oraz wywołanie funkcji kernela za pomocą powyższej składni na przykładzie programu sumującego dwie macierze. Przykład ten obrazuje również sposób użycia struktur w funkcjach kernela.

Listing 2.1. CUDA – Program sumujący macierze w wysokopoziomowym API.

```
1  #include <cuda_runtime_api.h>
2
3  struct Mat
4  {
5    int w;
6    int h;
7    float* elem;
8  };
9
10 __host__ void initMat(Mat& m, int w, int h)
11 {
12   m.w = w;
13   m.h = h;
14   cudaMalloc(&m.elem, w*h*sizeof(*m.elem));
15 }
16
17 __device__ __host__ float add(float v1, float v2)
18 {
19   return v1+v2;
20 }
21
22 __global__ void matAdd(const Mat A, const Mat B, Mat C)
23 {
24   int x = blockDim.x*blockIdx.x + threadIdx.x;
25   int y = blockDim.y*blockIdx.y + threadIdx.y;
26   int idx = x + C.w*y;
27   C.elem[idx] = add(A.elem[idx], B.elem[idx]);
28 }
29
30 int main()
31 {
32   const int size = 2048;
33   size_t matsize = size*size*sizeof(float);
34
35   Mat A, B, C;
36   initMat(A, size, size);
37   initMat(B, size, size);
38   initMat(C, size, size);
39
40   float *data = new float[size*size];
41   for(int i=0; i<size*size; ++i)
42     data[i] = 10;
43   cudaMemcpy(A.elem, data, matsize, cudaMemcpyHostToDevice);
44   for(int i=0; i<size*size; ++i)
45     data[i] = 13;
46   cudaMemcpy(B.elem, data, matsize, cudaMemcpyHostToDevice);
47
48   dim3 block(16,16);
49   dim3 grid(size/16, size/16);
```

```
50
51    matAdd<<<grid, block>>>(A, B, C);
52
53    cudaMemcpy(data, C.elem, matsize, cudaMemcpyDeviceToHost);
54
55    ...
56    cudaFree(A.elem);
57    cudaFree(B.elem);
58    cudaFree(C.elem);
59    delete[] data;
60 }
```

W programie założono, że macierze będą miały rozmiar zawsze podzielny przez 16. W samym programie:

- W liniach 3–8 zdefiniowana jest prosta struktura `Mat` opisująca macierz, zawierająca jedynie ilość kolumn i wierszy oraz wskaźnik na liniowy obszar pamięci przechowujący poszczególne elementy macierzy.
- W liniach 10–15 zdefiniowana została funkcja inicjalizująca macierz i alokująca dla niej pamięć po stronie GPU.
- W liniach 17–20 zdefiniowana została prosta funkcja `add()` zwracająca sumę dwóch liczb typu `float`. Dzięki użyciu specyfikatorów `__device__ __host__` kompilator wygeneruje dwie wersje tej funkcji – jedną klasyczną i drugą dostępną tylko z poziomu *urządzenia*.
- W liniach 22–28 jest właściwa definicja kernela realizującego dodawanie dwóch macierzy a wynik zapisująca do trzeciej macierzy. W linii 27 użyta została w funkcja `add()` zdefiniowana powyżej.
- W liniach 35–38 tworzone i inicjalizowane są trzy testowe macierze.
- W liniach 40–46 dwie pierwsze macierze wypełniane są konkretnymi wartościami. Użyta została do tego celu pomocnicza tablica zdefiniowana po stronie *hosta* oraz funkcja `cudaMemcpy()` do przesłania wartości macierzy do pamięci GPU.
- W linii 48 stworzony został obiekt typu `dim3` reprezentujący wielkość bloku. Rozmiar 16×16 daje w sumie 256 wątków przypadających na blok. Jest to dla tego przypadku i danego urządzenia optymalna wartość.
- W linii 49 tworzony jest obiekt reprezentujący rozmiar siatki, tak aby ilość bloków przypadająca na siatkę była odpowiednia i pokrywała całą przestrzeń utworzonych powyżej macierzy.

- W linii 51 następuje wywołanie funkcji kernela `matAdd()` z właściwymi parametrami.

Wysokopoziomowe API umożliwia również bardziej klasyczne wywołanie funkcji kernela za pomocą zestawu funkcji:

```
1) cudaError_t cudaConfigureCall(dim3 gridDim, dim3 blockDim,
                size_t sharedMem=0, cudaStream_t stream=0)
2) cudaError_t cudaSetupArgument(const void* arg, size_t size, size_t offset)
3) cudaError_t cudaLaunch(const char* entry)
```

Pierwsza funkcja `cudaConfigureCall()` specyfikuje rozmiar przestrzeni indeksów poprzez jawne podanie rozmiaru siatki i bloku. Opcjonalnie można podać wielkość pamięci współdzielonej oraz numer skojarzonego strumienia.

Druga funkcja `cudaSetupArgument()` odkłada na stosie wywołania funkcji `size` bajtów argumentu wskazywanego przez `arg` na pozycji poczynając od `offset` od początku stosu. Ta funkcja musi być poprzedzona wywołaniem `cudaConfigureCall()`.

Trzecia funkcja `cudaLaunch()` rozpoczyna wykonanie kernela. W tym przypadku funkcję rdzenia można podać w dwojaki sposób: (1) albo w postaci adresu funkcji, (2) albo w postaci c-stringu. W przypadku podania nazwy funkcji w postaci napisu, deklaracja kernela musi być zgodna z konwencją nazewniczą języka C, co oznacza, że w przypadku programów C++ musi być poprzedzona deklaracją `extern "C"`.

Odnosząc sie do wywołania funkcji rdzenia z listingu 2.1, linię 51, tj:

```
51      matAdd<<<grid, block>>>(A, B, C);
```

można zamienić na szereg następujących wywołań:

Listing 2.2. CUDA – Wywołanie kernela w wysokopoziomowym API – wersja 2.

```
51 cudaConfigureCall(grid, block);
52
53 int offset = 0;
54 cudaSetupArgument((void*)&A, sizeof(Mat), offset);
55 offset+=sizeof(Mat);
56 cudaSetupArgument((void*)&B, sizeof(Mat), offset);
57 offset+=sizeof(Mat);
58 cudaSetupArgument((void*)&C, sizeof(Mat), offset);
59
60 cudaLaunch("matMul");
61  // lub
62 cudaLaunch(matMul);
```

2.3. Programowanie niskopoziomowe CUDA

Implementacja niskopoziomowych funkcji *driver API* została zebrana w bibliotece `cuda`. Wszystkie wywołania tego API rozpoczynają się przedrostkiem `cu` a odpowiednie struktury przedrostkiem `CU`. W odróżnieniu od *runtime API* biblioteka niskopoziomowa jest instalowana razem ze sterownikiem do urządzenia NVIDIA.

W większości przypadków obiekty tego API są wskazywane przez uchwyty (ang. *handles*), którymi manipulują odpowiednie funkcje.

2.3.1. Inicjalizacja i kontekst

Przed wywołaniem jakiejkolwiek funkcji API środowisko musi zostać zainicjalizowane funkcją:

```
CUresult cuInit(unsigned int flags)
```

Parametr `flags` musi mieć wartość 0.

W następnym kroku musi zostać utworzony kontekst przypisany do wybranego urządzenia. Jest on odpowiednikiem procesu po stronie *hosta*. Kontekst ten musi stać się aktualnym kontekstem dla wywołującego funkcje API wątku *hosta*. Kontekst jest tworzony funkcją:

```
CUresult cuCtxCreate(CUcontext* pctx, unsigned int flags,
                     CUdevice dev)
```

W rezultacie wywołania w parametrze `pctx` zwrócony zostanie uchwyt do utworzonego kontekstu dla urządzenia `dev`. Parametr `flags` definiuje sposób w jaki *host* kontroluje wywołania API. Stała `CU_CTX_SCHED_AUTO` o wartości 0 jest dobrą wartością domyślną dla standardowego kontekstu. Tak utworzony kontekst staje się automatycznie aktualnym kontekstem dla danego wątku *hosta*.

Kontekst jest niszczony wywołaniem funkcji:

```
CUresult cuCtxDestroy(CUcontext ctx)
```

przyjmującej w parametrze uchwyt do niszczonego kontekstu. Wywołanie tej funkcji niszczy dany kontekst bez względu na to czy jest on przypisany do jakiegoś wątku czy nie.

Aktualny kontekst jest dostępny poprzez wywołanie funkcji:

```
CUresult cuCtxGetCurrent(CUcontext* pctx)
```

a aktualne urządzenie skojarzone z kontekstem jest dostępne poprzez funkcję:

```
CUresult cuCtxGetDevice(CUdevice* device)
```

Ponieważ standardowe wywołania funkcji rdzenia odbywają się asynchroniczne, przydatna do synchronizacji jest funkcja:

```
CUresult cuCtxSynchronize (void)
```

która blokuje aktualny wątek *hosta* do czasu zakończenia wszystkich wywołań dla danego kontekstu.

2.3.2. Konfiguracja urządzeń

Obiekt urządzenia jest tworzony za pomocą funkcji:

```
CUresult cuDeviceGet(CUdevice* device, int ndev)
```

która w parametrze `device` zwraca uchwyt do urządzenia o numerze `ndev`. Informacji ilości urządzeń zainstalowanych w systemie oraz ich parametrach dostarczają funkcje:

```
CUresult cuDeviceGetCount(int* count)
CUresult cuDeviceGetProperties(CUdevprop* prop,
                               CUdevice dev)
```

2.3.3. Wywołanie kernela

W obrębie danego kontekstu CUDA kernele są jawnie ładowane z poziomu *hosta* w postaci plików *PTX* lub obiektów binarnych. Taka forma kernela musi zostać najpierw przygotowana za pomocą kompilatora `nvcc` z opcją `--ptx` dla plików *PTX* lub z opcją `--cubin` w przypadku generacji pliku binarnego. Przykład utworzenia tego typu plików przedstawiony został w rozdziale 1.7.

Tak utworzone pliki kerneli wczytywane są do programu *hosta* w postaci modułów za pomocą funkcji:

```
CUresult cuModuleLoad(CUmodule* module, const char* fname)
```

Funkcja ta wczytuje plik o nazwie `fname` i w parametrze `module` zwraca uchwyt do utworzonego modułu w obrębie aktualnego kontekstu. Plik `fname` musi zawierać kod *PTX* lub kod binarny *cubin* lub *fatbin*.

Nieużywany moduł może być usunięty za pomocą funkcji:

```
CUresult cuModuleUnload(CUmodule module)
```

Uchwyt do konkretnej funkcji w module można uzyskać za pomocą funkcji:

```
CUresult cuModuleGetFunction(CUfunction* hfunc,
              CUmodule hmod, const char* name)
```

Uchwyt będzie zwrócony poprzez parametr `hfunc` dla modułu `hmod`. Parametr `name` jest pełną nazwą szukanej funkcji w postaci c-stringu.

Mając uchwyt do funkcji można wywołać kernel za pomocą funkcji:

```
CUresult cuLaunchKernel(CUfunction f, unsigned int gridDimX,
             unsigned int gridDimY, unsigned int gridDimZ,
             unsigned int blockDimX, unsigned int blockDimY,
             unsigned int blockDimZ,
             unsigned int sharedMemBytes, CUstream hStream,
             void ** kernelParams, void ** extra)
```

która rozpoczyna wykonanie funkcji rdzenia `f` na siatce `gridDimX` × `gridDimY` × `gridDimZ`. Każdy blok zawiera `blockDimX` × `blockDimY` × `blockDimZ` wątków. Parametr `sharedMemBytes` zawiera ilość bajtów współdzielonej pamięci dla każdego bloku a parametr `hStream` skojarzony strumień. Parametry wywołania funkcji kernela mogą być podane w dwojaki sposób za pomocą parametrów `kernelParams` lub `extra`.

1. W pierwszym przypadku wszystkie parametry muszą być zebrane w tablicy wskaźników typu `void* params[]`. Ilość elementów tablicy musi być równa ilości parametrów funkcji kernela.
2. W drugim przypadku parametr `void* extra[]` jest 5–elementową tablicą zbudowaną następująco:

   ```
   void *extra[] = {
     CU_LAUNCH_PARAM_BUFFER_POINTER, params,
     CU_LAUNCH_PARAM_BUFFER_SIZE, &paramsSize,
     CU_LAUNCH_PARAM_END
   };
   ```

   Zmienna `params` jest liniowym buforem zawierającym wartości wszystkich parametrów wywołania kernela a zmienna `paramSize` jest sumą rozmiarów w bajtach wszystkich parametrów.

Na listingu 2.3 znajduje się przykładowe wywołanie funkcji `cuLaunchKernel` dla pierwszego przypadku a na listingu 2.4 dla przypadku drugiego. Program realizuje podobną funkcjonalność do programu z listingu 2.1 sumującego dwie macierze.

Listing 2.3. CUDA – Program sumujący macierze w niskopoziomowym API.

```
1  #include <cuda.h>
2
3  int main()
4  {
5    CUdevice    hDevice;
6    CUcontext   hContext;
7    CUmodule    hModule;
8    CUfunction  hFunction;
9
10   cuInit(0);
11   cuDeviceGet(&hDevice, 0);
12   cuCtxCreate(&hContext, 0, hDevice);
13   cuModuleLoad(&hModule, "matAdd.ptx");
14   cuModuleGetFunction(&hFunction, hModule, "matAdd");
15
16   const int size = 4096;
17   size_t matsize = size*size*sizeof(float);
18
19   CUdeviceptr dA, dB, dC;
20   cuMemAlloc(&dA, matsize);
21   cuMemAlloc(&dB, matsize);
22   cuMemAlloc(&dC, matsize);
23
24   float *data = new float[size*size];
25   for(int i=0; i<size*size; ++i) data[i] = 10;
26   cuMemcpyHtoD(dA, data, matsize);
27   for(int i=0; i<size*size; ++i) data[i] = 13;
28   cuMemcpyHtoD(dB, data, matsize);
29
30   dim3 block(16, 16);
31   dim3 grid(size/16, size/16);
32
33   void* args[] = {(void*)&dA, (void*)&dB, (void*)&dC};
34   cuLaunchKernel(hFunction, grid.x, grid.y, grid.z,
35                  block.x, block.y, block.z,
36                  0, NULL, args, NULL);
37
38   cuCtxSynchronize();
39   cuMemcpyDtoH(data, dC, matsize);
40   ..
41   cuMemFree(dA);
42   cuMemFree(dB);
43   cuMemFree(dC);
44   delete[] data;
45 }
```

W programie:

- W linii 10 jest inicjalizowane środowisko a w linii 11 tworzony jest uchwyt do pierwszego urządzenia zgodnego z CUDA.

- W linii 12 tworzony jest kontekst dla tego urządzenia. Ten kontekst staje się aktualnym dla danego wątku *hosta*.
- W linii 13 wczytywany jest moduł z funkcjami kernela a w następnej linii uzyskiwany jest uchwyt do kernela o nazwie `"matAdd"`.
  W odróżnieniu od *runtime API* w niskopoziomowym interfejsie dostęp do pamięci GPU jest realizowany przy pomocy uchwytu typu `CUdeviceptr` a nie bezpośrednio za pomocą typowych wskaźników. Stąd też, w liniach 19–28 tworzone są odpowiednie uchwyty `CUdeviceptr` do danych macierzy i za ich pomocą alokowana jest pamięć na *device* oraz kopiowane są dane z pamięci *hosta* funkcją `cuMemcpyHtoD()` w linii 26 dla macierzy A i w linii 28 dla macierzy B.
- W linii 34 wywoływana jest funkcja kernela (wskazywanego przez uchwyt `hFunction`) za pomocą funkcji `cuLaunchKernel()`. Argumenty wywołania funkcji rdzenia zostały podane w pierwszy sposób za pomocą parametru `args`, który został przygotowany w linii 33 przez zebranie wszystkich argumentów w tablicy.

Drugi sposób przekazania argumentów do funkcji rdzenia zobrazowany jest na listingu 2.4. Kod całego programu pozostaje identyczny z kodem listingu 2.3, zmianie ulegają linie 33–36:

```
32   ...
33   void* args[] = {(void*)&dA, (void*)&dB, (void*)&dC};
34   cuLaunchKernel(hFunction, grid.x, grid.y, grid.z,
35                  block.x, block.y, block.z,
36                  0, NULL, args, NULL);
37  ...
```

odpowiednio na:

Listing 2.4. CUDA – Wywołanie funkcji rdzenia w niskopoziomowym API – wersja druga.

```
32 ...
33 char argbuff[256];
34 int offset = 0;
35
36 *((CUdeviceptr*)(argbuff+offset)) = dA;
37 offset += sizeof(dA);
38 *((CUdeviceptr*)(argbuff+offset)) = dB;
39 offset += sizeof(dB);
40 *((CUdeviceptr*)(argbuff+offset)) = dC;
41 offset += sizeof(dC);
42
43 void *extra[] = {
44   CU_LAUNCH_PARAM_BUFFER_POINTER, argbuff,
```

```
45   CU_LAUNCH_PARAM_BUFFER_SIZE, &offset,
46   CU_LAUNCH_PARAM_END
47 };
48
49 cuLaunchKernel(hFunction, grid.x, grid.y, grid.z,
50                block.x, block.y, block.z,
51                0, NULL, NULL, extra);
52 ...
```

W tym przypadku wszystkie argumenty wywołania kernela zostały zebrane w tablicę bajtową `argbuff[]`, zajmując `offset` jej początkowych bajtów. Tablica ta została umieszczona w drugim elemencie tablicy `void* extra[]` wymaganej przez funkcję `cuLaunchKernel()`.

Poniżej został zamieszczony kod funkcji rdzenia, na podstawie którego został wygenerowany plik `"matAdd.ptx"` użyty na listingu 2.3 w linii 13.

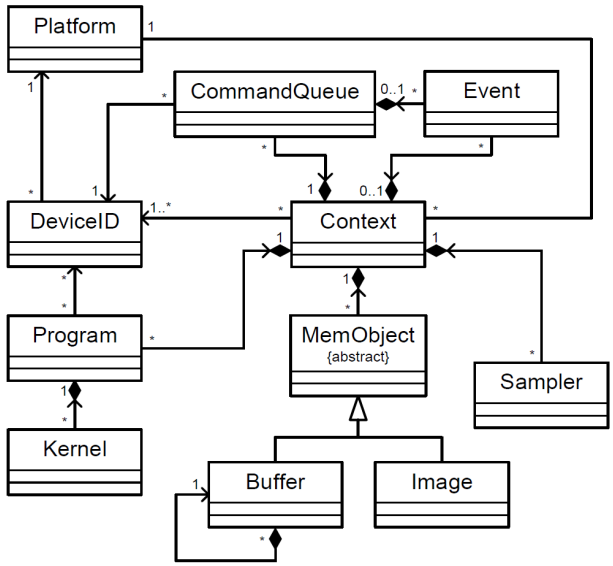
Listing 2.5. CUDA – Plik `"matAdd.cu"` funkcji kernela w niskopoziomowym API.

```
32 extern "C"
33 __device__ float add(float v1, float v2)
34 {
35   return v1+v2;
36 }
37
38 extern "C" __global__
39 void matAdd(const float* A, const float* B, float* C)
40 {
41   int x = blockDim.x*blockIdx.x + threadIdx.x;
42   int y = blockDim.y*blockIdx.y + threadIdx.y;
43   int idx = x + gridDim.x*blockDim.x*y;
44
45   C[idx] = add(A[idx], B[idx]);
46 }
```

2.4. Programowanie OpenCL

Rysunek 2.4 przedstawia diagram klas w notacji UML specyfikujący najważniejsze klasy obiektów w środowisku OpenCL.

Klasa Platformy jest najwyższą hierarchicznie klasą agregującą pozostałe klasy. Cały model platformy składa się z jednego *hosta* oraz jednego lub więcej urządzeń *device* OpenCL . Urządzenie OpenCL jest podzielone na jedno lub kilka jednostek obliczeniowych (CU - ang. *Computing Unit*), które są dalej podzielone na elementy przetwarzające (PE - ang. *processing elements*). Obliczenia na urządzeniu są realizowane przez elementy PE. Sa-



Rysunek 2.4. Diagram klas UML architektury OpenCL (źródło [4]).

ma aplikacja OpenCL jest uruchamiana na *hoście*. Z *hosta* są przekazywane do urządzenia polecenia wykonujące obliczenia za pomocą PE.

2.4.1. Inicjalizacja środowiska

Obiekt platformy OpenCL jest jest uzyskiwany za pomocą funkcji:

```
cl_int clGetPlatformIDs(cl_uint num_entries,
                        cl_platform_id *platforms,
                        cl_uint *num_platforms)
```

która zwraca listę dostępnych platform `platforms` oraz ich liczbę `num_platforms`. W następnym kroku niezbędne jest uzyskanie co najmniej jednego urządzenia OpenCL za pomocą funkcji:

```
cl_int clGetDeviceIDs(cl_platform_id platform,
                      cl_device_type device_type,
                      cl_uint num_entries,
                      cl_device_id *devices,
                      cl_uint *num_devices)
```

zwracającej dla platformy `platform` listę kompatybilnych urządzeń `devices` oraz ich liczbę `num_devices` podanego typu `device_type`. Możliwe typy urządzeń w rozważanej specyfikacji OpenCL 1.1 obejmują:
— `CL_DEVICE_TYPE_CPU` – urządzeniem obliczeniowym będzie procesor *hosta*,

— `CL_DEVICE_TYPE_GPU` – urządzeniem obliczeniowym będzie procesor graficzny GPU,
— `CL_DEVICE_TYPE_ACCELERATOR` – urządzeniem obliczeniowym będzie dedykowany akcelerator (np. IBM CELL Blade),
— `CL_DEVICE_TYPE_DEFAULT` – domyślne urządzenie obliczeniowe zainstalowane w systemie,
— `CL_DEVICE_TYPE_ALL` - wszystkie możliwe urządzenia obliczeniowe dostępne w systemie.

Konkretna implementacja biblioteki OpenCL nie musi obsługiwać wszystkich typów urządzeń. W rozważanych dwóch przypadkach, tj. implementacji NVIDIA i ATI, sterowniki obsługują odpowiednio `CL_DEVICE_TYPE_GPU` dla NVIDII oraz `CL_DEVICE_TYPE_CPU` i `CL_DEVICE_TYPE_GPU` dla ATI.

Po uzyskaniu obiektów urządzeń należy stworzyć kontekst OpenCL dla jednego lub wielu urządzeń, zarządzający obiektami takimi jak kolejka poleceń, obiekty pamięciowe, programy czy kernele. Specyfikacja OpenCL udostępnia dwie funkcje tworzące kontekst:

```
cl_context clCreateContext(
             const cl_context_properties *props,
             cl_uint num_devices,
             const cl_device_id *devices,
             void (CL_CALLBACK  *pfn_notify) (
                   const char *errinfo, const void*
                   info, size_t cb, void*user_data),
             void *user_data,
             cl_int *errcode_ret
)
cl_context clCreateContextFromType(
             const cl_context_properties *props,
             cl_device_type  device_type,
             void  (CL_CALLBACK *pfn_notify) (
                   const char *errinfo, const void*
                   info, size_t  cb, void*user_data),
             void  *user_data,
             cl_int  *errcode_ret
)
```

W pierwszym przypadku tworzony jest kontekst dla wszystkich urządzeń podanych w parametrze `devices`. W drugim przypadku kontekst jest tworzony dla wszystkich urządzeń podanego typu `device_type`. Parametr `cl_context_properties* props` jest listą dodatkowych parametrów kontekstu oraz ich wartości. Przykład użycia tego parametru będzie pokazany w rozdziale 6 podczas tworzenia kontekstu współdzielonego z kontekstem OpenGL. Parametr `pfn_notify` jest wskaźnikiem na funkcję zwrotną umożli-

wiającą uzyskiwanie informacji o błędach występujących podczas działania programu w obrębie tego kontekstu.

Na koniec pozostaje jeszcze utworzenie co najmniej jednej kolejki poleceń za pomocą funkcji:

```
cl_command_queue clCreateCommandQueue(
                   cl_context context,
                   cl_device_id device,
                   cl_command_queue_properties properties,
                   cl_int *errcode_ret
)
```

Kolejka ta będzie przechowywać listę poleceń do wykonania w obrębie kontekstu `context` na urządzeniu `device`.

2.4.2. Zarządzanie programem

Funkcje wykonywane na urządzeniu obliczeniowym są zebrane w obiekcie programu odpowiedzialnym za przechowywanie oraz kompilację kodu tych funkcji. Za utworzenie obiektu programu są odpowiedzialne dwie funkcje:

```
cl_program clCreateProgramWithSource(cl_context context,
             cl_uint count, const char **strings,
             const size_t *lengths, cl_int *errcode)
cl_program clCreateProgramWithBinary(cl_context context,
             cl_uint num_devs, const cl_device_id *device_list,
             const size_t *lengths, const unsigned char **
             binaries, cl_int *binary_status, cl_int *errcode)
```

W pierwszym przypadku, tworzony jest program z kodu źródłowego, przekazywanego w tablicy c-stringów `strings`. Program zostanie utworzony dla wszystkich urządzeń skojarzonych z kontekstem `context`. W drugim przypadku, program zostanie utworzony dla podanej w parametrze `binaries` listy prekompilowanych do postaci binarnej kodów. Program zostanie utworzony jedynie dla podanej w parametrze `device_list` listy urządzeń skojarzonych z kontekstem `context`.

Program OpenCL składa się ze zbioru funkcji rdzeni zadeklarowanych ze specyfikatorem `__kernel` oraz innych pomocniczych funkcji i stałych, które mogą być użyte wewnątrz kerneli.

Za kompilacje i linkowanie programu do wersji wykonywalnej odpowiada funkcja:

```
cl_int clBuildProgram(cl_program program,
         cl_uint num_devices, const cl_device_id *
         device_list, const char *options,
         void (CL_CALLBACK *pfn_notify)(
```

```
                cl_program program, void *user_data),
            void *user_data)
```

budująca `program` dla podanej listy urządzeń `device_list`. Parametr `options` umożliwia wyspecyfikowanie dodatkowych parametrów kompilacji i linkowania, wśród których znajdują się opcje preprocesora, opcje kontrolujące funkcje matematyczne, funkcje optymalizacji, ostrzeżeń czy wersji OpenCL. Pełna lista możliwych opcji znajduje się w specyfikacji OpenCL [4]. Kontrolę poprawności budowy programu umożliwiają dwie funkcje: `clGetProgramInfo()` oraz `clGetProgramBuildInfo()` omawiane szerzej w rozdziale 1.6.2.

Na tym etapie możliwe jest już utworzenie obiektów kerneli za pomocą funkcji:

```
cl_kernel clCreateKernel(cl_program  program,
                         const char *kernel_name,
                         cl_int *errcode_ret)
```

Funkcje rdzeni zawarte w programie `program` identyfikowane są poprzez ich nazwę `kernel_name` przekazywaną do funkcji w postaci c-stringu.

2.4.3. Wykonanie programu

Operacje składające się na wykonanie programu będą kolejkowane w kolejce poleceń `cl_command_queue`. Za dodawanie poleceń do kolejki odpowiadają funkcje o nazwie rozpoczynającej się na `clEnqueue...()` i obejmują one funkcjonalność odczytu i zapisu z/do bufora pamięci, mapowania pamięci, wstawiania markerów zdarzeniowych, synchronizacji za pomocą barier oraz samego wykonywania funkcji rdzeni.

Polecenie kolejkujące wykonanie kernela na urządzeniu ma następującą postać:

```
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 newl, const cl_event *ew_list,
          cl_event *event)
```

Parametr `command_queue` określa kolejkę, do której zostanie wstawiony `kernel`. Parametr `work_dim` określa ilość wymiarów użytych przy specyfikowaniu globalnej ilości *work-items*. Możliwe wartości tego parametru to 1, 2 lub 3. Parametr `global_work_offset` umożliwia podanie ewentualnych przesunięć w przestrzeni indeksów wątków. Parametry `global_work_size`

oraz `local_work_size` specyfikują odpowiednio globalną oraz lokalną ilość *work-items*.

Argumenty wywołania funkcji kernela są przekazywane za pomocą funkcji:

```
cl_int clSetKernelArg(cl_kernel kernel, cl_uint arg_index,
                      size_t arg_size, const void *arg_value)
```

Każde wywołanie tej funkcji specyfikuje dokładnie jeden argument rdzenia `kernel` o indeksie `arg_index`. Parametr `arg_size` określa wielkość typu przekazywanego argumentu w bajtach a parametr `arg_value` wskaźnik na jego wartość. Wartość argumentu jest kopiowana do pamięci urządzenia. W przypadku obiektów buforowych typu `cl_mem`, jako wielkość podaje się wielkość typu bufora, tj. `sizeof(cl_mem)`.

W sytuacji gdy obiekt, który ma być przetwarzany przez funkcję rdzenia jest wskaźnikiem na typ prosty lub zdefiniowaną strukturę, wtedy należy najpierw utworzyć odpowiedni obiekt buforowy za pomocą funkcji `clCreateBuffer()`, skopiować do niego dane za pomocą funkcji `clEnqueueWriteBuffer()` lub odpowiednika a następnie do funkcji `clSetKernelArg()` przekazać jako argument ten obiekt buforowy.

Listing 2.6 ilustruje sposób konfiguracji środowiska OpenCL, budowy programu OpenCL, deklarowania *NDRange* oraz wywołania kernela na przykładzie programu sumującego dwie macierze.

Listing 2.6. OpenCL – Program sumujący macierze.

```
1  #include <CL/opencl.h>
2
3  const int SIZE = 4096;
4
5  cl_platform_id platform;
6  cl_device_id device;
7  cl_context context;
8  cl_command_queue cmdQueue;
9  cl_program hProgram;
10 cl_kernel hKernel;
11
12 struct Mat
13 {
14   int w, h;
15   float* data;
16 };
17
18 void initMat(Mat& m, int w, int h)
19 {
20   m.w = w;
21   m.h = h;
```

```
22   m.data = new float[w*h];
23
24   for (int i=0; i<w*h; i++)
25     m.data[i] = rand()/1024/256;
26 }
27
28 int main(int argv, char *argc[])
29 {
30   Mat matA, matB, matC;
31   initMat(matA, SIZE, SIZE);
32   initMat(matB, SIZE, SIZE);
33   initMat(matC, SIZE, SIZE);
34
35   clGetPlatformIDs(1, &platform, NULL)
36   cl_uint num_dev;
37   clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device,
38                  &num_dev);
39   context = clCreateContext(0, 1, &device, 0,0,0);
40   cmdQueue = clCreateCommandQueue(context, device, 0,0);
41
42   size_t kernelLength;
43   char* programSource = loadProgSource("matadd.cl", "",
44                                        &kernelLength);
45   hProgram = clCreateProgramWithSource(context, 1,
46              (const char**)&programSource, &kernelLength, 0);
47   clBuildProgram(hProgram, 0, 0, 0, 0, 0);
48
49   hKernel = clCreateKernel(hProgram, "matadd", 0);
50
51   cl_mem cl_matA, cl_matB, cl_matC;
52
53   cl_matA = clCreateBuffer(context, CL_MEM_READ_ONLY,
54                            matA.w*matA.h*sizeof(float),0,0);
55   cl_matB = clCreateBuffer(context, CL_MEM_READ_ONLY,
56                            matB.w*matB.h*sizeof(float),0,0);
57   cl_matC = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
58                            matC.w*matC.h*sizeof(float),0,0);
59
60   clEnqueueWriteBuffer(cmdQueue, cl_matA, CL_FALSE, 0,
61     matA.w*matA.h*sizeof(float), matA.data, 0,0,0);
62   clEnqueueWriteBuffer(cmdQueue, cl_matB, CL_FALSE, 0,
63     matB.w*matB.h*sizeof(float), matB.data, 0,0,0);
64
65   clSetKernelArg(hKernel, 0, sizeof(cl_mem), &cl_matC);
66   clSetKernelArg(hKernel, 1, sizeof(cl_mem), &cl_matA);
67   clSetKernelArg(hKernel, 2, sizeof(cl_mem), &cl_matB);
68   int len = matC.w*matC.h;
69   clSetKernelArg(hKernel, 3, sizeof(int), &len);
70
71   size_t GLOBAL_WS[] = {matC.w, matC.h};
72   size_t LOCAL_WS[] = {16, 16};
```

```
73   clEnqueueNDRangeKernel(cmdQueue, hKernel, 2, 0,
74                          GLOBAL_WS, LOCAL_WS, 0, 0, 0);
75
76   clEnqueueReadBuffer(cmdQueue, cl_matC, CL_FALSE, 0,
77     matC.w*matC.h*sizeof(float), matC.data, 0,0,0);
78
79   clFinish(cmdQueue);
80
81   ...
82   clReleaseMemObject(cl_matA);
83   clReleaseMemObject(cl_matB);
84   clReleaseMemObject(cl_matC);
85
86   return 0;
87 }
```

W programie:

- W liniach 12–26 zdefiniowana została prosta struktura `Mat` opisująca macierz, zawierająca jedynie ilość kolumn i wierszy oraz wskaźnik na liniowy obszar pamięci przechowujący poszczególne elementy macierzy. Struktura ta reprezentuje dane po stronie *hosta*. Macierz będzie inicjalizowana wartościami pseudolosowymi.
- W liniach 35–40 uzyskane zostały obiekty platformy `platform` oraz urządzenia `device`. Na aktualne zostało wybrane pierwsze urządzenie typu `CL_DEVICE_TYPE_GPU` zainstalowane w systemie. Z tym urządzeniem został skojarzony kontekst `context`, a następnie została dla niego stworzona kolejka poleceń `cmdQueue`.
- W liniach 42–47 wczytywany jest z pliku `matadd.cl` kod źródłowy kernela (przedstawiony na listingu 2.7). Na podstawie tego kodu tworzony jest w linii 47 obiekt programu `hProgram`, który jest następnie budowany w linii 49.
- W liniach 51–58 za pomocą funkcji `clCreateBuffer()` utworzone zostały 3 obiekty pamięciowe `matA, matB, matC`, które będą reprezentować wartości macierzy po stronie urządzenia OpenCL.
- W liniach 60–63 dane z macierzy *hosta* są kopiowane do pamięci GPU.
- W liniach 65–69 zostały wyspecyfikowane argumenty wywołania funkcji kernela.
- W liniach 71–74 następuje właściwe zakolejkowanie funkcji kernela za pomocą funkcji `clEnqueueNDRangeKernel()`. W tym przypadku globalna ilość

work-items `GLOBAL_WS` została ustawiona na wielkość macierzy docelowej `matC` a lokalna `LOCAL_WS` na 16×16.

Na koniec pozostaje jeszcze przedstawienie samej funkcji rdzenia liczącej sumy poszczególnych elementów macierzy:

Listing 2.7. OpenCL – Dodawanie macierzy – funkcja rdzenia.

```
1  __kernel void matadd(__global float* matDest,
2                       __global float* matA,
3                       __global float* matB,
4                       int len)
5  {
6    int i = get_global_id(0) + get_global_id(1) *
7                               get_global_size(0);
8    if(i<len)
9      matDest[i] = matA[i] + matB[i];
10 }
```

2.5. Pomiar czasu za pomocą zdarzeń GPU

Dotychczas stosowanym sposobem pomiaru czasu była dosyć nieprecyzyjna metoda oparta na pobieraniu czasu systemowego, którego dokładność wynosiła ok. 1 milisekundy. Ta dokładność w dużej mierze zależy od programowanego sprzętu i systemu operacyjnego. Co więcej, problematyczny może okazać się pomiar czasu funkcji asynchronicznych wykonywanych na GPU. Rozwiązaniem tego problemu może być użycie dedykowanych funkcji pomiaru czasu zdefiniowanych w obu środowiskach CUDA/OpenCL. W obu przypadkach pomiar czasu jest oparty na tzw. obiektach zdarzeniowych (ang. *event obects*). Czas jest w tym przypadku odmierzany przez urządzenie obliczeniowe (GPU) z dokładnością do nanosekund.

CUDA

Obiekt zdarzeniowy w CUDA `cudaEvent_t event` jest tworzony za pomocą funkcji konstruktora:

```
cudaError_t cudaEventCreate(cudaEvent_t* event)
```

i niszczony za pomocą funkcji:

```
cudaError_t cudaEventDestroy(cudaEvent_t event)
```

Takie obiekty zdarzeniowe mogą zostać wykorzystane do pomiaru czasu przez ich rejestrację w danym strumieniu za pomocą funkcji:

```
cudaError_t cudaEventRecord(cudaEvent_t event,
```

```
                                    cudaStream_t stream = 0)
```

Tak zarejestrowane zdarzenie zostanie uznane za zakończone, gdy zostaną wykonane wszystkie polecenia z danego strumienia poprzedzające wywołanie funkcji rejestrującej zdarzenie. Zakończone zdarzenia mogą posłużyć do pomiaru czasu dzięki funkcji:

```
cudaError_t cudaEventElapsedTime(float* ms, cudaEvent_t start,
                                 cudaEvent_t end)
```

określającej, z bardzo dużą dokładnością, czas jaki upłyną po stronie GPU pomiędzy tymi dwoma zdarzeniami. Dokładnie, funkcja `cudaElapsedTime()` zwraca czas jaki upłyną pomiędzy zakończeniem zdarzenia `end` a zakończeniem zdarzenia `start`.

Poniżej znajduje się fragment prostego kodu obrazującego sposób pomiaru czasu metodą zdarzeniową:

Listing 2.8. CUDA – Metoda pomiaru czasu za pomocą zdarzeń.

```
32 float time;
33 cudaEvent_t start, end;
34
35 cudaEventCreate(&start)
36 cudaEventCreate(&end)
37
38 cudaEventRecord(start, 0);
39
40   cudaMemcpy(...)
41   kernel<<<...>>>(...);
42   cudaMemcpy(...)
43
44 cudaEventRecord(end, 0);
45 cudaEventSynchronize(end);
46 cudaEventElapsedTime(&time, start, end);
```

Przed samym pomiarem czasu została wywołana funkcja `cudaEventSynchronize(cudaEvent_t)`. Jest to jedna z wielu metod synchronizacji strumienia CUDA po stronie *hosta*, która blokuje aktualny wątek *hosta*, aż do czasu wykonania wszystkich poleceń w danym strumieniu poprzedzających podane w parametrze zdarzenie.

W OpenCL, aby możliwy był pomiar czasu GPU, należy włączyć opcję profilowania kolejki rozkazów podczas jej tworzenia:

OpenCL

```
cl_command_queue queue = clCreateCommandQueue(context, devices,
                         CL_QUEUE_PROFILING_ENABLE, 0);
```

W powyższym przykładzie, w trzecim parametrze została podana flaga `CL_QUEUE_PROFILING_ENABLE` włączająca profilowanie. W przypadku istniejących kolejek rozkazów można włączać lub wyłączać konkretne opcje za pomocą funkcji:

```
cl_int clSetCommandQueueProperty(cl_command_queue command_queue,
       cl_command_queue_properties properties,
       cl_bool enable,
       cl_command_queue_properties *old_properties)
```

Listing 2.9 przedstawia sposób wykorzystania obiektów zdarzeniowych `cl_events` do pomiaru czasu wykonania kolejki poleceń oraz czasu wykonania pojedynczego polecenia.

Listing 2.9. OpenCL – Metoda pomiaru czasu za pomocą zdarzeń.

```
32 cl_event start, end, kernel_ev;
33 cl_ulong time, time2;
34
35 clEnqueueMarker(queue, &start);
36
37   clEnqueueWriteBuffer(...);
38   clSetKernelArg(...);
39   clEnqueueNDRangeKernel(queue, kernel, 1, 0,
40                          GLOBAL_WS, LOCAL_WS, 0, 0,
41                          &kernel_ev);
42   clEnqueueReadBuffer(...);
43
44 clEnqueueMarker(queue, &end);
45 clFinish(queue);
46
47
48 clGetEventProfilingInfo(start, CL_PROFILING_COMMAND_START,
49                         sizeof(cl_ulong), &time, 0);
50 clGetEventProfilingInfo(stop, CL_PROFILING_COMMAND_END,
51                         sizeof(cl_ulong), &time2, 0);
52 cout<<"Total time: "<<(time2-time)*1e-6<<"[ms]"<<endl;
53
54 clGetEventProfilingInfo(kernel_ev, CL_PROFILING_COMMAND_START,
55                         sizeof(cl_ulong), &time, 0);
56 clGetEventProfilingInfo(kernel_ev, CL_PROFILING_COMMAND_END,
57                         sizeof(cl_ulong), &time2, 0);
58 cout<<"Kernel time: "<<(time2-time)*1e-6<<"[ms]"<<endl;
```

Funkcja użyta w linii 35 i 44, tj:

```
cl_int clEnqueueMarker(cl_command_queue queue, &cl_event*event)
```

wstawia do kolejki znacznik i zwraca związane z tym znacznikiem zdarzenie

`event`. Znacznik w kolejce zostanie uznany za zakończony, w momencie gdy wszystkie polecenia wstawione do tej kolejki przed nim zostaną zakończone. W ten sposób w linii 35 do kolejki `queue` został wstawiony znacznik opisany zdarzeniem `start`, który posłuży do określenia rozpoczęcia pomiaru czasu a w linii 44 został wstawiony znacznik opisany zdarzeniem `end` wykorzystany następnie do określenia czasu zakończenia pomiaru. Dodatkowe zdarzenie `kernel_ev`, przekazane w ostatnim parametrze wywołania funkcji kernela `clEnqueueNDRangeKernel()` w linii 39, będzie identyfikowało to konkretne jego wywołanie.

Znaczniki czasowe można uzyskać z danego zdarzenia za pomocą funkcji:

```
cl_int clGetEventProfilingInfo(cl_event event,
       cl_profiling_info param_name, size_t param_value_size,
       void *param_value, size_t *param_value_size_ret)
```

która w parametrach przyjmuje konkretny obiekt zdarzeniowy `event`, nazwę parametru, który chcemy uzyskać i jego rozmiar w bajtach oraz aktualną wartość tego parametru. W przypadku profilowania czasu wywołania istotne są dwa parametry: `CL_PROFILING_COMMAND_START` oraz `CL_PROFILING_COMMAND_END` zwracające 64-bitową liczbę całkowitą `cl_ulong` zawierającą czas w nanosekundach na danym urządzeniu, kiedy odpowiednio, dane polecenie rozpoczęło swoje wykonanie i zakończyło swoje wykonanie.

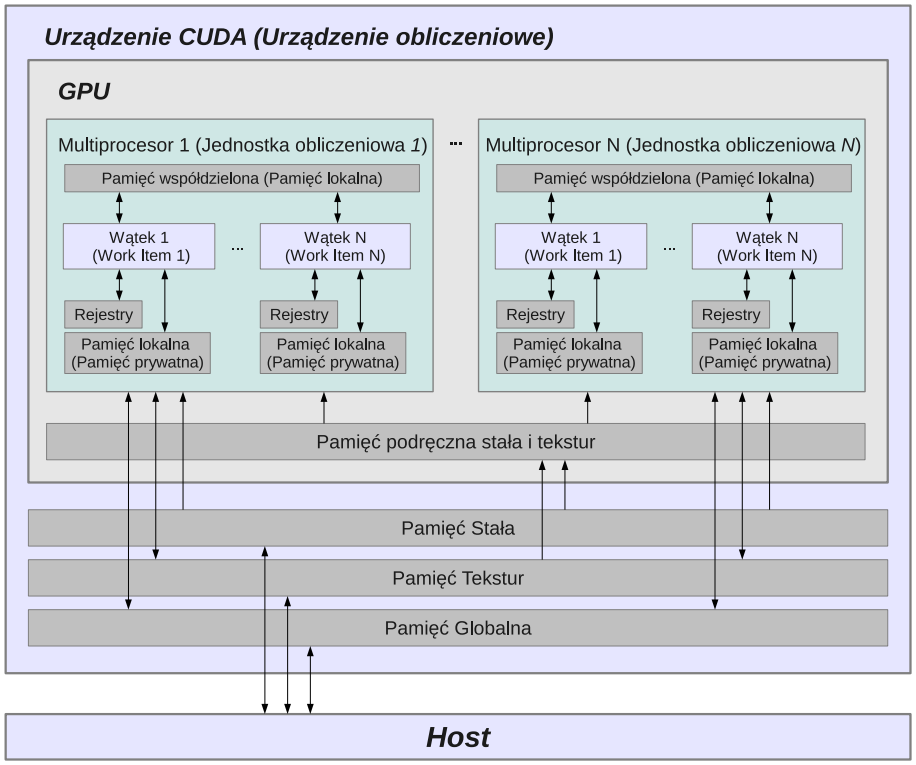
Rozdział 3

Model pamięci GPGPU

3.1. Typy pamięci

W przypadku obu rozważanych środowisk heterogenicznych, typy pamięci można ogólnie podzielić na pamięć *hosta* (*host memory*) znajdującą się fizycznie w przestrzeni CPU oraz pamięć karty graficznej (*device memory*) rezydującą w przestrzeni GPU. Transfer pomiędzy tymi dwoma typami pamięciami jest możliwy tylko przy pomocy dedykowanych funkcji API danego środowiska.

Pamięć urządzenia jest jednak zdecydowanie bardziej złożona niż w przypadku *hosta* i ogólnie składa się z pamięci globalnej, lokalnej, stałej, współdzielonej, pamięci tekstur oraz rejestrów. Rysunek 3.1 obrazuje podział pamięci urządzenia obliczeniowego oraz możliwe przepływy danych pomiędzy poszczególnymi typami pamięci.



Rysunek 3.1. Model pamięci kart graficznych dla środowiska CUDA oraz OpenCL (w nawiasach podano nazwy OpenCL).

Takie zróżnicowanie jest wynikiem kompromisu pomiędzy szybkością transferu/czasem dostępu do pamięci a wielkością danego typu pamięci. Część typów pamięci (współdzielona, rejestry, podręczna) zostały fizycz-

nie umieszczone wewnątrz procesora GPU zapewniając maksymalną wydajność, natomiast pozostałe, tj. pamięć globalna, tekstur, stała oraz lokalna zostały umieszczone na zewnątrz kości procesora ale są najbardziej pojemne i jednocześnie najwolniejsze. W niektórych architekturach sprzętowych te zewnętrzne pamięci mogą być buforowane w pamięci podręcznej znajdującej się wewnątrz procesora GPU.

W przypadku obu środowisk wszystkie typy pamięci mają swoje dokładne odpowiedniki. Rozbieżności występują tylko w warstwie nazewniczej, gdzie pamięci współdzielonej (*shared memory*) w CUDA odpowiada nazwa pamięć lokalna (*local memory*) w OpenCL oraz w nazwie pamięci prywatnej (*private memory*) w CUDA, której odpowiada nazwa pamięć lokalna (*local memory*) w OpenCL. W tabeli 3.1 zebrano odpowiadające sobie nazwy dla wszystkich typów pamięci dla CUDA i OpenCL.

Tabela 3.1. Odpowiedniki nazw pamięci urządzenia obliczeniowego dla środowisk CUDA i OpenCL

CUDA	OpenCL
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

3.1.1. Pamięć globalna

Pamięć globalna jest najbardziej pojemnym typem pamięci, fizycznie z reguły umieszczonym na karcie graficznej (lub współdzielonej z pamięcią *hosta* dla zintegrowanych układów graficznych) o dostępie swobodnym (DRAM - ang. Dynamic Random Access Memory). Jest to równocześnie najwolniejszy typ pamięci obsługiwanej przez urządzenia CUDA/OpenCL, którego opóźnienia w dostępnie sięgają setek cykli procesora GPU.

Ten typ pamięci może być odczytywany (read) i zapisywany (write) zrówno przez urządzenie jak i *hosta* dzięki dedykowanym funkcjom. Z poziomu kernela, w obrębie danego *Grida*/*NDRange*, cała zaalokowana pamięć globalna jest dostępna dla każdego działającego wątku urządzenia.

Poza pamięcią stałą jest to jedyny rodzaj pamięci, który jest dostępny z poziomu *hosta*.

W przypadku środowiska CUDA za alokację/dealokację pamięci odpowiadają funkcje:

CUDA

```
cudaError_t cudaMalloc(void** devPtr, size_t size)
cudaError_t cudaMalloc3D(struct cudaPitchedPtr* pitchedDevPtr,
```

```
            struct cudaExtent extent)
cudaError_t cudaMallocArray(struct cudaArray** array, const
            struct cudaChannelFormatDesc* desc, size_t width,
            size_t height=0, unsigned int flags=0)
cudaError_t cudaMalloc3DArray(struct cudaArray** array,
            const struct cudaChannelFormatDesc* desc,
            struct cudaExtent extent, unsigned int flags=0)
cudaError_t cudaFree(void* devPtr)
cudaError_t cudaFreeArray(struct cudaArray* array)
```

W każdym przypadku funkcje alokujące w parametrze przyjmują referencję na wskaźnik na dany obiekt pamięci oraz rozmiar obszaru, podany bezpośrednio lub w odpowiednich strukturach. Funkcje te są w stanie zaalokować pamięć liniową lub w postaci tablic. Tablice CUDA są zoptymalizowane pod kątem wykorzystania ich w postaci tekstur. Są one przedmiotem dyskusji rozdziału 6.

Za transfer pomiędzy pamięcią *hosta* a pamięcią globalną odpowiadają rodziny funkcji:

```
cudaError_t cudaMemcpy(void* dst, const void* src,
            size_t count, enum cudaMemcpyKind kind)
cudaError_t cudaMemset(void* devPtr, int value, size_t count)
```

OpenCL

W środowisku OpenCL obiekt w pamięci globalnej są przechowywane w postaci buforów lub obrazów. Za alokację/dealokację pamięci odpowiadają funkcje:

```
cl_mem clCreateBuffer(cl_context context, cl_mem_flags flags,
       size_t size, void *host_ptr, cl_int *errcode_ret)
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)
cl_int clReleaseMemObject(cl_mem memobj)
```

W pierwszym przypadku jest to alokacja `size` bajtów pamięci globalnej. Parametr `flags` decyduje o możliwości dostępu do pamięci z poziomu funkcji kernela i musi przyjmować jedną z wartości: `CL_MEM_READ_WRITE`, `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY`. Możliwa jest również alternatywa bitowa dla alokacji pamięci zabezpieczonej przed stronicowaniem. Ten sposób dostępu do pamięci zostanie opisany w rozdziale 3.3.

Za transfer pomiędzy pamięcią *hosta* a pamięcią globalną odpowiadają funkcje:

```
cl_int clEnqueueReadBuffer(cl_command_queue command_queue,
```

```
        cl_mem buffer, cl_bool blocking_read, size_t offset,
        size_t cb, void *ptr, cl_uint newl, const cl_event* ewt,
        cl_event *event)
cl_int clEnqueueWriteBuffer(cl_command_queue command_queue,
        cl_mem buffer, cl_bool blocking_write, size_t offset,
        size_t cb, const void *ptr, cl_uint newl,
        const cl_event* ewl, cl_event *event)
cl_int clEnqueueReadImage(cl_command_queue command_queue,
        cl_mem image, cl_bool blocking_read, const size_t
        origin[3], const size_t region[3], size_t row_pitch,
        size_t slice_pitch, void *ptr, cl_uint newl,
        const cl_event *ewl, cl_event *event)
cl_int clEnqueueWriteImage(cl_command_queue command_queue,
        cl_mem image, cl_bool blocking_write, const size_t
        origin[3], const size_t region[3], size_t in_row_pitch,
        size_t input_slice_pitch, const void * ptr, cl_uint
        newl, const cl_event *ewl, cl_event *event)
```

kolejno do kopiowania bufora z pamięci globalnej do pamięci *hosta*, do kopiowania bufora z pamięci *hosta* do pamięci globalnej, oraz analogiczne wersje dla kopiowania pamięci obrazów.

Wewnątrz definicji funkcji rdzenia zmienne umieszczone w pamięci globalnej są deklarowane z kwalifikatorem `__global`.

Niewielka przepustowość pamięci globalnej w stosunku do mocy obliczeniowej GPU jest często powodem znacznego spadku wydajności obliczeń. Miarą efektywności wykorzystania pamięci globalnej jest współczynnik CGMA (ang. *Compute to Global Memory Access*) opisujący ilość obliczeń zmiennoprzecinkowych w stosunku do ilości sięgnięć do pamięci globalnej. Dla współczesnych architektur sprzętowych wydajność obliczeniową szacuje się na rząd wielkości większą od przepustowości pamięci globalnej. Zatem optymalnie byłoby gdyby na jedno sięgnięcie do pamięci przypadały przynajmniej 10 operacji zmiennoprzecinkowych. Z tego też powodu powstało wiele technik optymalizacji dostępu do pamięci, które obejmują wykorzystanie pamięci typu `constant` oraz pamięci współdzielonej `shared memory/local memory`.

3.1.2. Pamięć stała

Pamięć stała (ang. *constant memory*) jest wydzielonym obszarem pamięci urządzenia, która jest zoptymalizowana pod kątem szybkości dostępu z poziomu urządzenia ale zezwala jedynie na odczyt wartości poszczególnych komórek. W przypadku, gdy wiele aktywnych wątków (w obrębie

pół-warpa[1]) w danym cyklu próbuje odczytać wartość z tej samej komórki pamięci stałej, kontroler pamięci wykonuje tylko jeden odczyt, który jest rozsyłany (ang. *bradcast*) do wszystkich żądających wątków. Co więcej, pamięć typu stałego jest buforowana, zatem kolejne odczyty z tego samego adresu (nawet spoza danego *pół-warpa*) nie będą wymagały dodatkowego transferu.

Niestety w przypadku gdy wątki z tego samego *pół-warpa* odczytują różne komórki pamięci stałej ich wykonanie jest serializowane, tym samym znacznie wydłużając czas realizacji danego kernela. W takim przypadku na ogół wydajniejsze będzie wykorzystanie pamięci globalnej.

Z poziomu *hosta* pamięć tego typu jest możliwa do zapisu i odczytu. Pojemność tego typu pamięci we współczesnych architekturach to ok. 64KB.

CUDA

W architekturze CUDA zmienne, które mają być umieszczone w pamięci stałej poprzedza się specyfikatorem `__constant__`. Takie zmienne mogą być wykorzystane jedynie w funkcjach kernela. Za transfer pomiędzy pamięcią hosta a pamięcią stałą odpowiadają funkcje:

```
cudaError_t cudaMemcpyToSymbol(const char* symbol,
            const void* src, size_t count, size_t offset=0,
            enum cudaMemcpyKind kind=cudaMemcpyHostToDevice)
cudaError_t cudaMemcpyFromSymbol(void* dst, const char* symbol,
            size_t count, size_t offset = 0,
            enum cudaMemcpyKind kind=cudaMemcpyDeviceToHost)
```

oraz ich asynchroniczne odpowiedniki.

Na listingu 3.1 został umieszczony kod programu wykorzystującego pamięć typu `constant` jako bufor dla zagadnienia typu LookUpTable. W zagadnieniu tym, pewien zbiór liczb całkowitych zawartych w tablicy `data` jest odwzorowywany na nowy zbiór wartości za pomocą tablicy `cu_lut` przechowującej pary indeks-wartość. Ta pomocnicza tablica będzie zaalokowana w pamięci stałej urządzenia.

Listing 3.1. CUDA – Przykład użycia pamięci `constant`.

```
1 #include <cuda_runtime_api.h>
2
3 #define SIZE 1024
4 #define LUT_SIZE 256
5
6 __constant__ int cu_lut[LUT_SIZE];
```

[1] W architekturze NVIDII blok wątków jest podzielony na grupy zwane *warpami* zawierające 32 wątki o sąsiadujących indeksach w przestrzeni indeksów. Poszczególne *warpy* danego bloku nie muszą wykonywać się w tym samym czasie. Przez *pół-warp* należy rozumieć 16 wątków danego *warpa* o indeksach 0–15 lub 16–31.

```
7
8  __global__ void lookUpTable(int* data)
9  {
10   int i = threadIdx.x + blockIdx.x*blockDim.x;
11   data[i] = cu_lut[data[i]];
12 }
13
14 int main(int argc, char* argv[])
15 {
16   int lut[LUT_SIZE];
17   int* data = new int[SIZE];
18   int* odata = new int[SIZE];
19   int* cu_data;
20
21   for(int i=0; i<LUT_SIZE; ++i)
22      lut[i] = LUT_SIZE-1-i;
23
24   for(int i=0; i<SIZE; ++i)
25     data[i] = rand()%LUT_SIZE;
26
27   cudaMalloc(&cu_data, SIZE*sizeof(int));
28   cudaMemcpy(cu_data, data, SIZE*sizeof(int),
29              cudaMemcpyHostToDevice);
30
31   cudaMemcpyToSymbol(cu_lut, lut, LUT_SIZE*sizeof(int), 0,
32                      cudaMemcpyHostToDevice);
33
34   lookUpTable<<<SIZE/256, 256>>>(cu_data);
35
36   cudaMemcpy(odata, cu_data, SIZE*sizeof(int),
37              cudaMemcpyDeviceToHost);
38
39   for(int i=0; i<SIZE; ++i)
40     cout << odata[i] << ", ";
41
42   return 0;
43 }
```

W odróżnieniu od CUDA, w architekturze OpenCL, pamięć stała może być alokowana dynamicznie z poziomu hosta. Ponieważ programy OpenCL mogą działać na różnym sprzęcie, wielkość pamięci `constant` nie jest z góry ograniczona. Konkretną wartość dla danej platformy można uzyskać poprzez wywołanie funkcji `clGetDeviceInfo()`:

OpenCL

```
cl_uint max_const_size;
clGetDeviceInfo(device, CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE,
                sizeof(cl_uint), &max_const_size, NULL);
```

Poniżej przedstawiony został program o analogicznej funkcjonalności do programu z Listingu 3.1. Zmienna zapisana w pamięci stałej jest przekazywana do funkcji rdzenia w parametrze specyfikowanym jako `__constant`.

Listing 3.2. OpenCL – Przykład użycia pamięci `constant` – program kernela.

```
__kernel void lookUpTable(__global int* data,
                          __constant int* lut)
{
  int i = get_global_id(0);
  data[i] = lut[data[i]];
}
```

W programie głównym pamięć typu stałego jest traktowana jak standardowa pamięć globalna. Jedyną różnicą jest określenie sposobu dostępu podczas tworzenia obiektu w pamięci urządzenia w funkcji `clCreateBuffer()`. Drugi parametr tej funkcji musi mieć wartość `CL_MEM_READ_ONLY`.

Listing 3.3. OpenCL – Przykład użycia pamięci `constant`.

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

cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue cmdQueue;
cl_program hProgram;
cl_kernel hKernel;

#define SIZE 1024
#define LUT_SIZE 256

int main(int argv, char *argc[])
{
  int* data  = new int[SIZE];
  int* odata = new int[SIZE];
  int* lut = new int[LUT_SIZE];
  cl_mem cl_data = 0;
  cl_mem cl_lut = 0;

  for(int i=0; i<LUT_SIZE; ++i)
    lut[i] = 255-i;

  for (int i=0; i<SIZE; i++)
    data[i] = rand()%LUT_SIZE;

  clGetPlatformIDs(1, &platform, NULL)
  cl_uint num_dev;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device,
                 &num_dev);
```

```
31   context = clCreateContext(0, 1, &device, 0,0,0);
32   cmdQueue = clCreateCommandQueue(context, device, 0,0);
33
34   size_t kernelLength;
35   char* programSource = loadProgSource("lookUpTable.cl", "",
36                                        &kernelLength);
37   hProgram = clCreateProgramWithSource(context, 1,
38              (const char**)&programSource, &kernelLength, 0);
39   clBuildProgram(hProgram, 0, 0, 0, 0, 0);
40   hKernel = clCreateKernel(hProgram, "lookUpTable", 0);
41   size_t GLOBAL_WS[] = {SIZE};
42   size_t LOCAL_WS[] = {256};
43
44   cl_data = clCreateBuffer(context, CL_MEM_READ_WRITE |
45             CL_MEM_COPY_HOST_PTR, SIZE*sizeof(int),  data, 0);
46   cl_lut = clCreateBuffer(context, CL_MEM_READ_ONLY |
47            CL_MEM_COPY_HOST_PTR, LUT_SIZE*sizeof(int), lut, 0);
48
49   clSetKernelArg(hKernel, 0, sizeof(cl_mem), &cl_data);
50   clSetKernelArg(hKernel, 1, sizeof(cl_mem), &cl_lut);
51   clEnqueueNDRangeKernel(cmdQueue, hKernel, 1, 0,
52                          GLOBAL_WS, LOCAL_WS, 0, 0, 0);
53
54   clEnqueueReadBuffer(cmdQueue, cl_data, CL_FALSE, 0,
55                       SIZE*sizeof(int), odata, 0, NULL, NULL);
56   clFinish(cmdQueue);
57
58   for(int i=0; i<SIZE; ++i)
59     cout << odata[i] << ", ";
60   return 0;
61 }
```

3.1.3. Pamięć współdzielona

Pamięć współdzielona (`shared memory/local memory`) (w środowisku OpenCL jest nazywana pamięcią lokalną) jest bardzo szybką pamięcią znajdującą się wewnątrz procesora obliczeniowego. Czas dostępu do danych jest ok. 100-krotnie mniejszy niż w przypadku dostępu do pamięci globalnej. Pamięć współdzielona fizycznie rezyduje wewnątrz Multiprocesora/Jednostki obliczeniowej (MP). Współczesne architektury sprzętowe posiadają tego typu pamięci kilkadziesiąt kilobajtów, przypadających na każdy Multiprocesor karty grafiki. Na każdym Multiprocesorze całość pamięci współdzielonej jest dzielona na poszczególne bloki działające w danym czasie. Przykładowo, w urządzeniach NVIDIA o *Compute Capability* 2.x na pojedynczy Multiprocesor przypada 48KB pamięci współdzielonej, co przy maksymalnie 8

blokach działających w danym czasie na MP daje 6KB takiej pamięci do wykorzystania w obrębie bloku wątków.

Poza szybkością, drugą istotną cechą pamięci współdzielonej jest jej zakres widoczności przez poszczególne wątki. Dokładniej, wszystkie wątki działające w obrębie bloku mają dostęp do całej zawartości pamięci współdzielonej przydzielonej temu blokowi. Ta cecha sprawia, że jest to idealny sposób synchronizacji danych pomiędzy wątkami, niestety tylko w obrębie własnego bloku.

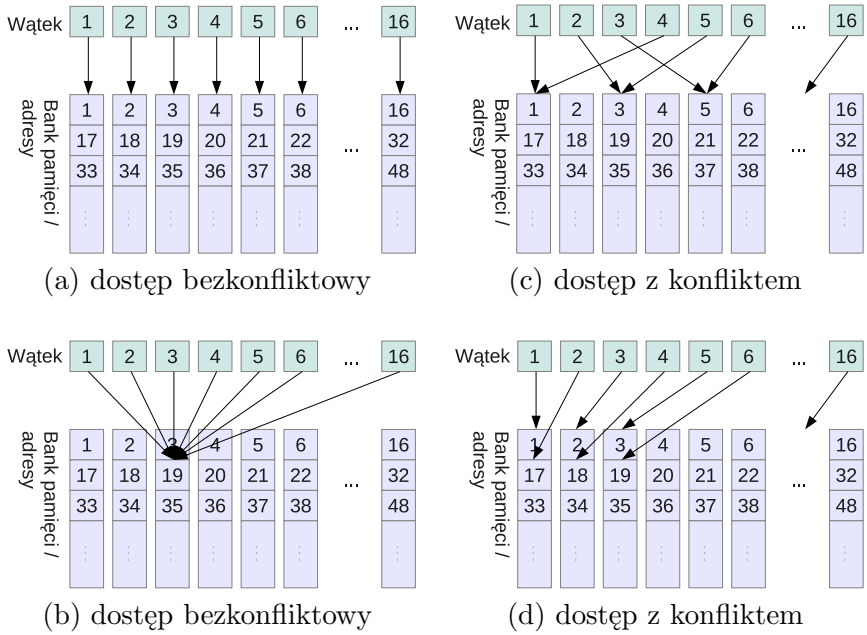
Aby uzyskać wysoką wydajność, pamięć współdzielona podzielona jest na moduły o równej wielkości zwane bankami pamięci (ang. *memory banks*). W sytuacji gdy kilka wątków żąda danych znajdujących się w pamięci współdzielonej ale w różnych bankach, dostęp do tych danych jest równoczesny. Jeżeli natomiast kilka wątków żąda różnych danych z tego samego banku pamięci, dostęp jest serializowany a wątki wykonują się szeregowo. W przypadku gdy kilka wątków żąda tego samego elementu z tego samego banku pamięci, wtedy możliwy jest dostęp poprzez rozgłaszanie (ang. *broadcast*). W urządzeniach NVIDIA zgodnych z *Compute Capability* 1.x jest dokładnie 16 banków a w urządzeniach zgodnych z *Compute Capability* 2.x są 32 banki pamięci współdzielonej. Kolejne adresy pamięci współdzielonej znajdują się w kolejnych bankach pamięci, na przemian. Taki rozkład zabezpiecza przed konfliktem dostępu przy liniowym dostępie do pamięci wewnątrz danego *warpa*. Na rysunku 3.2 przeanalizowane zostały 4 przypadki dostępu do pamięci współdzielonej, po dwa przykłady dla dostępu do banków bez konfliktu i z konfliktem.

Szczegóły wykorzystania pamięci współdzielonej wraz z przykładem zostały umieszczone w rozdziale 3.2

3.1.4. Rejestry i pamięć lokalna

W obrębie funkcji wykonywanych na urządzeniu obliczeniowym lokalne automatyczne zmienne przechowywane są w rejestrach. Jest to bardzo wydajna pamięć o niemalże zerowym czasie dostępu do danych. Niestety jej ilość jest mocno ograniczona. Współczesne architektury posiadają ok. 32k–64k 32-bitowych rejestrów, przypadających na każdy blok. W obrębie bloku może działać równocześnie nawet do tysiąca wątków (w zależności od urządzenia), co w wyraźny sposób ogranicza ilość rejestrów do kilkudziesięciu w danej funkcji kernela. Zastosowanie liczb zmiennoprzecinkowych podwójnej precyzji zmniejsza tę liczbę dwukrotnie.

Pamięć lokalna (`local memory/private memory`) (w środowisku OpenCL jest nazywaną pamięcią prywatną) jest pewnym rozszerzeniem rejestrów



Rysunek 3.2. Organizacja dostępu do pamięci współdzielonej z poziomu *pół-warpa*: (a) dostęp bezkonfliktowy – każdy wątek czyta z innego banku pamięci; (b) dostęp bezkonfliktowy – wszystkie wątki czytają z tego samego adresu, wykonane zostanie rozgłaszanie; (c) dostęp z konfliktem – wątki wykonają się w dwóch grupach, ponieważ po dwa wątki sięgają do tych samych komórek; (d) dostęp z konfliktem – wątki wykonają się w dwóch grupach, ponieważ po dwa wątki sięgają do różnych adresów tych samych banków pamięci.

procesora. Dokładnie mówiąc, jest to wydzielona część pamięci globalnej, która zostanie użyta dla zmiennych automatycznych, w przypadku gdy:
— wszystkie rejestry w danym bloku zostały już wykorzystane,
— alokowana jest struktura lub tablica o dużym rozmiarze.

Ponieważ pamięć lokalna, fizycznie jest identyczna z pamięcią globalną, rezydując poza procesorem GPU, dziedziczy po niej ten sam wysoki czas dostępu do danych oraz niską przepustowość. Jednakże, niektóre architektury sprzętowe (np. zgodne z *Compute Capability* 2.x) buforują pamięć lokalną w szybkiej pamięci podręcznej L1 lub L2.

3.1.5. Pamięć tekstur

Pamięć tekstur (`texture memory`) rezyduje obok pamięci globalnej poza procesorem urządzenia ale jest jednocześnie buforowana w pamięci podręcz-

nej zwanej *texture cache*. Oznacza to, że sięgnięcie do pamięci tekstury jest dodatkowym kosztem tylko w przypadku chybienia pamięci podręcznej, w przeciwnym wypadku koszt odczytu elementu tekstury z pamięci podręcznej jest o rząd wielkości mniejszy.

Pamięć tekstur jest zoptymalizowana pod kątem dostępu do niewielkiego dwuwymiarowego sąsiedztwa w obrębie danego bloku wątków. Jednakże, nic nie stoi na przeszkodzie do użycia pamięci tekstur jako pamięci globalnej w zagadnieniu nie związanym z grafiką. Co więcej, w niektórych przypadkach, pamięć ta może sprawdzić się lepiej od pamięci globalnej, ponieważ:

— jest buforowana w obrębie niewielkiego sąsiedztwa,
— obliczenia przesunięć adresów są realizowane przez dedykowane układy, poza funkcją kernela,
— może być rozgłaszana (ang. *broadcast*) do kilku zmiennych w pojedynczej operacji,
— 8- lub 16-bitowe liczby całkowite mogą być konwertowane do 32-bitowych liczb zmiennoprzecinkowych zawartych w przedziale $[0.0, 1.0]$ lub $[-1.0, 1.0]$,
— jest automatyczna obsługa przypadków sięgnięcia poza obszar tekstury,
— możliwa jest automatyczna liniowa interpolacja pomiędzy sąsiadującymi elementami tekstury.

Szersze omówienie wykorzystania pamięci tekstur wraz z przykładami jest zawarte w rozdziale 6.

3.2. Wykorzystanie pamięci współdzielonej do optymalizacji dostępu do pamięci urządzenia

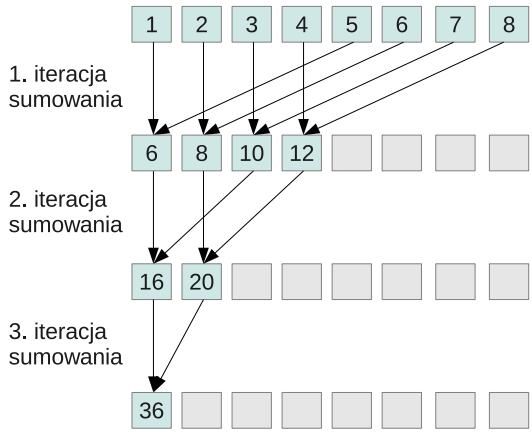
Przeanalizujmy prosty problem algorytmu redukcji, który używa binarnej operacji do obliczenia z całej sekwencji pojedynczej wartości. Niech to będzie przypadek obliczający sumę elementów wektora. W szeregowym klasycznym podejściu taki algorytm sprowadziłby się do iteracji po wszystkich elementach wektora i dodania każdego z nich do wspólnej sumy.

Listing 3.4. Klasyczny algorytm redukcji z sumą.

```
1 float reduceCPU(float* vec, int size)
2 {
3   double sum=0.0f;
4   while(size)
5     sum += vec[--size];
6   return sum;
7 }
```

Obliczenia będą przeprowadzane dla dużych wektorów (rzędu 100 milionów elementów), stąd potrzeba zastosowania liczby zmiennoprzecinkowej podwójnej precyzji `double` w linii 3 do przechowywania sumy cząstkowej, która jest w stanie, bez utraty dokładności dodawać małe wartości (`vec[i]`) do wartości dużych (`sum`).

W przypadku zrównoleglonym, algorytm redukcji należy rozbić na iteracyjny proces, który w każdej iteracji dodaje w danym wątku tylko dwie liczby zapisując wynik w pomocniczym wektorze. W pierwszej iteracji ilość działających wątków powinna być zatem równa połowie ilości elementów wektora. Jeżeli każdy wątek doda dwie liczby to całkowita ilość liczb do przesumowania zredukuje się dwukrotnie. W kolejnych iteracjach ilość wątków zmniejsza się zawsze dwukrotnie, aż do sytuacji gdy pozostają tylko dwie liczby do zsumowania. To podejście wymaga jedynie `O(log(N))` iteracji w porównaniu do `O(N)` iteracji klasycznego szeregowego algorytmu. Na rysunku 3.3 przedstawiony został uproszczony algorytm redukcji 8-elementowego wektora.



Rysunek 3.3. Zrównoleglony algorytm redukcji z sumą. Pierwotny 8-elementowy wektor w pierwszej iteracji został zredukowany za pomocą 4 wątków do 4-elementowego wektora. W drugiej iteracji dwa wątki zredukowały ilość elementów do dwóch, które po zsumowaniu w trzeciej iteracji dały całkowitą sumę wektora.

W przypadku kart graficznych nie ma do dyspozycji tak dużej ilości wątków działających równocześnie zatem problem musi łączyć oba przedstawione powyżej podejścia, tzn. każdy wątek najpierw obliczy sumę pewnej

liczby elementów, a następnie przeprowadzi redukcję w obrębie każdego bloku wątków.

CUDA

Na listingu 3.5 przedstawiony został kod kernela obliczającego sumy cząstkowe wektora. Dla celów testowych siatkę podzielono na 256 bloków po 256 wątków w każdym bloku, co daje łącznie 65536 wykonań kernela wewnątrz siatki.

Funkcja kernela o nazwie `reduce()` przyjmuje w parametrze tablicę wejściową `vec` oraz tablicę pomocniczą `vec_out`. Tablica `vec_out` jest globalną tablicą, do której zostaną zapisane wyniki sum cząstkowych obliczonych z poszczególnych bloków siatki a jej wielkość będzie równa dokładnie ilości tych bloków.

Listing 3.5. CUDA – Algorytm redukcji z sumą – funkcja kernela.

```
1  #include <cuda_runtime_api.h>
2
3  const int SIZE = 67108864;
4  const int N_THREADS = 256;
5  const int N_BLOCKS  = 256;
6
7  dim3 blocks(N_BLOCKS);
8  dim3 threads(N_THREADS);
9
10 __global__ void reduce(float* vec, float* vec_out, int size)
11 {
12   __shared__ float cache[N_THREADS];
13   float sum = 0.0f;
14
15   int idx = blockIdx.x*blockDim.x + threadIdx.x;
16   for(int i=idx; i<size; i+=blockDim.x*gridDim.x)
17     sum += vec[i];
18
19   cache[threadIdx.x] = sum;
20   __syncthreads();
21
22   for(int k=blockDim.x/2; k; k/=2)
23   {
24     if(threadIdx.x < k)
25     {
26       cache[threadIdx.x] += cache[threadIdx.x+k];
27     }
28     __syncthreads();
29   }
30
31   if(threadIdx.x == 0)
32     vec_out[blockIdx.x] = cache[0];
33 }
```

W przypadku, gdy sumowany wektor ma więcej niż 65536 elementów, to każdy wątek, w pętli, w liniach 16–17 oblicza sumę cząstkową dla kolejnych elementów o indeksach będących wielokrotnościami tej liczby. W ten sposób, liczba elementów zostanie zredukowana dokładnie do tej wartości.

W następnym kroku należy wykonać redukcję w obrębie każdego bloku.

- Do tego celu w linii 12 została utworzona pomocnicza tablica:

  ```
  __shared__ float cache[N_THREADS]
  ```

  Specyfikator `__shared__` oznacza, że dana zmienna zostanie umieszczona w pamięci współdzielonej. W tym przypadku tworzona jest 256-elementowa tablica liczb zmiennoprzecinkowych pojedynczej precyzji, zajmująca w sumie 1024 bajty pamięci dla każdego bloku działającego na danym Multiprocesorze. W przypadku karty grafiki zgodnej z *Compute Capability* 1.x całkowita ilość pamięci współdzielonej przypadającej na Multiprocesor (MP) wynosi $16KB$. Na MP, w danej chwili może być uruchomione maksymalnie 8 bloków. W analizowanym przypadku 8 bloków będzie potrzebowało $1024B * 8 = 8192B$ pamięci, zatem zarezerwowana ilość pamięci współdzielonej mieści się w wyznaczonym limicie, w żaden sposób nie ograniczając pełnej wydajności GPU.

- W linii 19 do każdego elementu tej współdzielonej tablicy jest przypisywana suma elementów obliczonych przez każdy wątek w obrębie bloku w pierwszym kroku.

- W linii 20 wywoływana jest wbudowana funkcja:

  ```
  __syncthreads()
  ```

  która stanowi punkt synchronizacji wewnątrz funkcji kernela, działając jak bariera, do której wszystkie wątki danego bloku muszą dojść zanim wykona się dalsza część kodu. Po wykonaniu tej instrukcji jest już pewne, że wszystkie elementy tablicy `cache` zostały poprawnie uzupełnione.

- W liniach 22–29 w obrębie danego bloku wykona się redukcja 256-elementowej tablicy `cache`. W każdej iteracji pętli, każdy wątek doda dwa elementy tablicy `cache` a następnie ilość wątków przeprowadzających obliczenia zostanie zmniejszona dwukrotnie, aż do momentu, w którym pozostanie już tylko pojedyncza wartość w elemencie `cache[0]`. Instrukcja warunkowa `if` w linii 24 powoduje, że tylko część wątków w danym *warpie* będzie wykonywała operacje sumowania. Pozostała część wątków tego *warpa* wykona puste instrukcje. Zajmie to w sumie dwa cykle, ponieważ w danym cyklu mogą być wykonywane tylko te wątki, które realizują identyczny zestaw instrukcji.

- W liniach 31–32 tylko pierwszy wątek danego bloku wykona instrukcję przypisania wartości sumy znajdującej się w zerowym elemencie `cache[0]` do globalnej tablicy `vec_out`.
  W ten sposób cały wejściowy wektor `vec` został zredukowany do 256 sum cząstkowych zapisanych w tablicy `vec_out` przez każdy z bloków.

Aby obliczyć całkowitą sumę należy jeszcze dodać wszystkie sumy cząstkowe. Ta część zadania zostanie wykonana już po stronie *hosta*.

Na listingu 3.6 przedstawiona została dalsza część programu począwszy od funkcji głównej `main()`.

Listing 3.6. CUDA – Algorytm redukcji z sumą.

```
34 int main(int argc, char* argv[])
35 {
36   float* vec = new float[SIZE];
37   float  vec_out[N_BLOCKS];
38   float* cu_vec, cu_vec_out;
39   float  sum=0;
40   double time, time2;
41
42   for(int i=0; i<SIZE; i++)
43     vec[i] = 1.0f - 2.0f*rand()/RAND_MAX;
44
45   time = timeStamp();
46     sum = reduceCPU(vec, SIZE);
47   time2 = timeStamp();
48   cout<<"CPU sum="<<sum<<", time="<<time2-time<<"[ms]"<<endl;
49
50   cudaMalloc((void**)&cu_vec, sizeof(float)*SIZE);
51   cudaMalloc((void**)&cu_vec_out, sizeof(float)*N_BLOCKS);
52   cudaMemcpy(cu_vec, vec, sizeof(float)*SIZE,
53              cudaMemcpyHostToDevice);
54
55   sum = 0.0f;
56   time = timeStamp();
57     reduce<<<blocks,threads>>>(cu_vec, cu_vec_out, SIZE);
58     cudaMemcpy(vec_out, cu_vec_out, sizeof(float)*N_BLOCKS,
59                cudaMemcpyDeviceToHost);
60     cudaThreadSynchronize();
61
62     for(int i=0;i<N_BLOCKS; i++)
63       sum += vec_out[i];
64   time2 = timeStamp();
65
66   cout<<"GPU sum="<<sum<<", time="<<time2-time<<"[ms]"<<endl;
67
68   cudaFree(cu_vec);
69   cudaFree(cu_vec_out);
```

```
70   delete[] vec;
71   return 0;
72 }
```

W testowanym przypadku, sumowany wektor będzie się składał z `SIZE=67108864` elementów typu **`float`** (256MB).

- W liniach 42–43 wektor `vec` został wypełniony wartościami pseudolosowymi z zakresu $[-1, 1]$.
- W liniach 45–48 przeprowadzony został test obliczeń wykonanych na CPU za pomocą funkcji `reduceCPU()` zdefiniowanej na listingu 3.4.
- W liniach 50–53 alokowane są niezbędne obszary pamięci globalnej GPU `vec` oraz `vec_out` oraz kopiowana jest zawartość testowego wektora.
- W liniach 56–64 przeprowadzony został właściwy test funkcji `reduce()` realizowanej na GPU. Funkcja ta oblicza jedynie cząstkowe sumy, które zostały następnie dodane klasycznie na CPU w liniach 62–63.
  Dla pewności poprawności danych zawartych w tablicy `vec_out` w linii 60 została wywołana funkcja:

  ```
  cudaDeviceSynchronize()
  ```

  która blokuje aktualny wątek *hosta*, dopóki nie wykonają się wszystkie zakolejkowane na urządzeniu zadania.

Poniżej przedstawiony został kod programu realizującego analogiczną funkcjonalność, tj. wyznaczającego sumę elementów wektora, w środowisku OpenCL. Kod zostanie przytoczony w całości, natomiast omówienie ograniczy się jedynie do wskazania różnic pomiędzy oboma środowiskami. Na listingu 3.7 przedstawiona została funkcja kernela realizująca w OpenCL analogiczne zadanie do funkcji CUDA z listingu 3.5.

OpenCL

Listing 3.7. OpenCL – Algorytm redukcji z sumą – funkcja kernela.

```
1  #define N_THREADS 256
2  __kernel void reduce(__global  float* vec,
3                       __global  float* vec_out,
4                       int size)
5  {
6    __local float cache[N_THREADS];
7    float sum = 0.0f;
8
9    int idx = get_global_id(0);
10   for(int i=idx; i<size; i+=get_global_size(0))
11     sum += vec[i];
```

```
12
13   cache[get_local_id(0)] = sum;
14   barrier(CLK_LOCAL_MEM_FENCE);
15
16   for(int k=get_local_size(0)/2; k; k/=2)
17   {
18     if(get_local_id(0) < k)
19       cache[get_local_id(0)] += cache[get_local_id(0)+k];
20     barrier(CLK_LOCAL_MEM_FENCE);
21   }
22
23   if(get_local_id(0)==0)
24     vec_out[get_group_id(0)] = cache[0];
25 }
```

W środowisku OpenCL pamięć współdzielona nazywana jest lokalną a deklaracja zmiennej umieszczonej w tej pamięci musi zawierać specyfikator `__local`.

Punktem synchronizacji *work-items* w obrębie *work-group* jest funkcja:

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

Parametr `flags` może dowolną kombinację dwóch flag: (1) `CLK_LOCAL_MEM_FENCE`, która zapewnia spójność zmiennych znajdujących się w pamięci lokalnej oraz (2) `CLK_GLOBAL_MEM_FENCE`, która zapewnia spójność pamięci globalnej.

Listing 3.8. OpenCL – Algorytm redukcji z sumą.

```
1  #include <CL/opencl.h>
2
3  cl_platform_id   platform;
4  cl_device_id     device;
5  cl_context       context;
6  cl_command_queue cmdQueue;
7  cl_program       hProgram;
8  cl_kernel        hKernel;
9
10 const int SIZE      = 67108864;
11 const int N_THREADS = 256;
12 const int N_BLOCKS  = 256;
13 size_t GLOBAL_WS[]  = {N_THREADS*N_BLOCKS};
14 size_t LOCAL_WS[]   = {N_THREADS};
15
16 int main(int argv, char *argc[])
17 {
18   float* vec = new float[SIZE];
19   float  vec_out[N_BLOCKS];
20   cl_mem cl_vec = 0;
```

```
21   cl_mem cl_vec_out = 0;
22   double time, time2;
23   float  sum=0;
24
25   for (int i=0; i<SIZE; i++)
26     vec[i] = 1.0f-2.0f*rand()/RAND_MAX;
27
28   clGetPlatformIDs(1, &platform, NULL)
29   cl_uint num_dev;
30   clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device,
31                  &num_dev);
32   context = clCreateContext(0, 1, &device, 0,0,0);
33   cmdQueue = clCreateCommandQueue(context, device, 0,0);
34
35   size_t kernelLength;
36   char* programSource = loadProgSource("reduce.cl", "",
37                                        &kernelLength);
38   cmdQueue = clCreateCommandQueue(context, devices, 0,0);
39   hProgram = clCreateProgramWithSource(context, 1,
40              (const char**)&programSource, &kernelLength, 0);
41
42   clBuildProgram(hProgram, 0, 0, 0, 0, 0);
43   hKernel = clCreateKernel(hProgram, "reduce", 0);
44
45   cl_vec = clCreateBuffer(context, CL_MEM_READ_ONLY |
46            CL_MEM_COPY_HOST_PTR, SIZE*sizeof(float), vec, 0);
47   cl_vec_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
48                               N_THREADS*sizeof(float), 0,0);
49
50   time = timeStamp();
51     clSetKernelArg(hKernel, 0, sizeof(cl_mem), &cl_vec);
52     clSetKernelArg(hKernel, 1, sizeof(cl_mem), &cl_vec_out);
53     clSetKernelArg(hKernel, 2, sizeof(int), &SIZE);
54     clEnqueueNDRangeKernel(cmdQueue, hKernel, 1, 0,
55                            GLOBAL_WS, LOCAL_WS, 0,0,0);
56     clEnqueueReadBuffer(cmdQueue, cl_vec_out, CL_TRUE, 0,
57                         N_BLOCKS*sizeof(float), vec_out, 0,0,0);
58     clFinish(cmdQueue);
59
60     for(int i=0;i<N_BLOCKS; i++)
61       sum += vec_out[i];
62   time2 = timeStamp();
63
64   cout<<"GPU sum="<<sum<<", time="<<time2-time<<"[ms]"<<endl;
65
66   clReleaseMemObject(cl_vec);
67   clReleaseMemObject(cl_vec_out);
68   delete[] vec;
69
70   return 0;
71 }
```

Po inicjalizacji wektorów oraz samego środowiska OpenCL, w pamięci globalnej zostały utworzone w liniach 45–48 GPU dwa wektory: wejściowy `cu_vec` oraz wyjściowy `cu_vec_out`. Pierwszy wektor został zadeklarowany jako tylko-do-odczytu `CL_MEM_READ_ONLY` a drugi jako tylko-do-zapisu `CL_MEM_WRITE_ONLY`.

W liniach 50–62 przeprowadzony został właściwy test redukujący wszystkie elementy wektora do 256 sum cząstkowych za pomocą GPU i następnie obliczający całkowitą sumę już na *hoście* w liniach 60–61.

Założona ilość wątków w pojedynczym bloku jest standardową, optymalną ilością dla współczesnych kart graficznych, natomiast ilość bloków jest w zasadzie dowolna a jej konkretna wartość jest podyktowana możliwościami danego urządzenia, na którym będzie się dany kernel wykonywał. Powinna to być wartość na tyle duża żeby obsadzić wszystkie jednostki obliczeniowe urządzenia. Na przykład, karta grafiki, na której testowano powyższe programy NVIDIA GeForce GTX560Ti ma 8 Multiprocesorów, co daje w sumie 8MP * 8 bloków na MP = 64 bloki wątków uruchomione jednocześnie. Oznacza to, że mniejsza ilość bloków może nie wykorzystać optymalnie wszystkich możliwych jednostek obliczeniowych.

Czas wykonania sumy wektora (o 64M elementach) obliczonej przez CPU (Intel Core 2 Quad) dla tego konkretnego przykładu wyniósł 280[ms] natomiast czas wykonania obliczeń za pomocą GPU wyniósł 2.9[ms]. Wzrost wydajności jest niemalże o dwa rzędy wielkości. Jednakże, wydajność obliczeń malała liniowo wraz ze zmniejszaniem ilości elementów wektora. Taką graniczną wartością, przy której procesor GPU był wydajniejszy od CPU było 32768 elementów. Przy mniejszych wektorach bardziej opłacalne jest przeprowadzanie obliczeń za pomocą głównego procesora.

3.3. Pamięć zabezpieczona przed stronicowaniem

Ogólny przebieg programu wykorzystującego obliczenia na GPU składa się z transferu danych z pamięci *hosta* do pamięci GPU, wykonania obliczeń oraz transferu danych z powrotem z GPU do *hosta*. Z przeanalizowanego w rozdziale 1.3 programu jasno wynika, że koszt samego transferu danych pomiędzy *hostem* a *urządzeniem* jest dość znaczny i często stanowi wąskie gardło całego procesu. W pewnych przypadkach można nieco zmniejszyć te straty czasowe korzystając z tzw. „**pamięci zablokowanej przed stronicowaniem**“ (ang. *page-locked memory*). Stosowana jest również nazwa „**pamięć przypięta**“ (ang. *pinned memory*).

Pamięć zablokowana przez stronicowaniem jest fragmentem wirtualnej pamięci *hosta*, dla której system zapewnia niezmienne położenie w fizycznej

pamięci, w szczególności zapewnia, że dany obszar nie będzie przeniesiony do części pamięci wirtualnej rezydującej na dysku. Taki warunek umożliwia bezpośrednie kopiowanie bloku danych z pamięci *hosta* na kartę grafiki przy użyci mechanizmu zwanego DMA (ang. *Direct Memory Access*) bez pośrednictwa CPU. Klasyczna, niezablokowana pamięć wymaga pośrednictwa procesora podczas aktualizacji tablic stronicowania pamięci i jej realokacji. W konsekwencji kopiowanie danych z *hosta* na kartę grafiki odbywa się wtedy w dwóch krokach, po pierwsze bufor jest kopiowany z pamięci stronicowanej do pomocniczego przypiętego bufora, a stamtąd w drugim kroku do pamięci GPU poprzez DMA.

Niestety używanie tego mechanizmu wiąże się z pewnymi kosztami. Przede wszystkim w systemie musi być wystarczająco dużo fizycznej pamięci dla każdego przypiętego bufora, ponieważ nie mogą one być realokowane ani zapisane na dyskowej części pamięci wirtualnej. Może to powodować dużo szybsze wyczerpywanie się pamięci komputera oraz wpływać ogólnie na wydajność systemu, zmuszając inne działające aplikacje do rezydowania na dysku.

Wykorzystanie mechanizmu pamięci zablokowanej wymaga alokacji bufora w pamięci *hosta* przy użyciu dedykowanych funkcji analogicznych do klasycznej funkcji `malloc()`. Poniżej przedstawiony zostanie sposób użycia tego mechanizmu oraz jego wpływu na wydajność transferu dla obu środowisk.

3.3.1. CUDA

Poniżej przedstawiony został program testujący wydajność prostego problemu obliczeniowego z uwzględnieniem transferu pamięci przy wykorzystaniu różnych technik kopiowania. Listing 3.9 przedstawia początek programu testującego wydajność obliczeń przeprowadzonych na CPU, nie wymagających transferów pamięci.

Listing 3.9. CUDA – Pamięć zablokowana przez stronicowaniem - część CPU.

```
1  #include <cuda_runtime_api.h>
2
3  #define N_ITER 100
4  #define SIZE 16777216
5
6  dim3 BLOCKS(256, 256);
7  dim3 THREADS(256);
8
9  __global__ void pow2_gpu(float* in, float* out)
10 {
```

```
11   int i = gridDim.x*blockDim.x*blockIdx.y+
12           blockIdx.x*blockDim.x + threadIdx.x;
13   out[i] = in[i]*in[i];
14 }
15
16 void pow2_cpu(float* in, float* out, int size)
17 {
18   for (int i=0; i<size; i++)
19     out[i] = in[i]*in[i]);
20 };
21
22 int main()
23 {
24   double time1, time2;
25
26   float* vec  = new float[SIZE];
27   float* vec2 = new float[SIZE];
28
29   for (int i=0; i<SIZE; i++)
30     vec[i] = rand()/(float)RAND_MAX;
31
32   time1 = timeStamp();
33   for(int i=0; i<N_ITER; i++)
34     pow2_cpu(vec, vec2, SIZE);
35   time2 = timeStamp();
36
37   cout<<"CPU time="<<(time2-time1)/N_ITER<<"[ms]"<<endl;
```

- W liniach 9–14 oraz 16–20 zostały zdefiniowane dwie funkcje `pow2_gpu()` oraz `pow2_cpu()`, odpowiednio dla GPU i CPU, obliczające kwadrat wartości przekazanej w parametrze tablicy.
- Na potrzeby testu w liniach 26–27 zaalokowane zostały dwie tablice `vec` oraz `vec2` typu **`float`** o wielkości `SIZE` elementów każda.
- Tablica `vec` w liniach 29–30 została wypełniona wartościami pseudolosowymi.
- W liniach 32–35 został zmierzony czas `N_ITER=100` iteracji obliczeń przeprowadzonych na procesorze CPU.

Przedstawiona na listingu 3.10 dalsza część programu zawiera fragment obliczeń przeniesionych na GPU ale przy użyciu klasycznego modelu alokacji i transferu pamięci.

Listing 3.10. CUDA – Pamięć zablokowana przez stronicowaniem - klasyczna alokacja GPU.

```
39  float* vec_res = new float[SIZE];
40  float* cu_vec;
41  cudaMalloc((void**)&cu_vec, sizeof(float)*SIZE);
42
43  time1 = timeStamp();
44  for(int i=0; i<N_ITER; i++)
45  {
46    cudaMemcpy(cu_vec, vec, sizeof(float)*SIZE,
47               cudaMemcpyHostToDevice);
48    pow2_gpu<<<BLOCKS, THREADS>>>(cu_vec, cu_vec);
49    cudaMemcpy(vec_res, cu_vec, sizeof(float)*SIZE,
50               cudaMemcpyDeviceToHost);
51    cudaThreadSynchronize();
52  }
53  time2 = timeStamp();
54
55  cout<<"GPU 1 time="<<(time2-time1)/N_ITER<<"[ms]"<<endl;
56
57  if(std::equal(vec2, vec2+SIZE, vec_res))
58    cout<<"CPU and GPU vec are identical"<<endl;
59  else
60    cout<<"CPU and GPU vec are DIFFERENT"<<endl;
61
62  cudaFree(cu_vec);
63  delete[] vec_res;
```

- Na potrzeby testu zadeklarowane zostały nowe dwa wskaźniki, `vec_res` wskazujący na tablicę przechowującą wyniki obliczeń oraz `cu_vec` wskazujący na obszar pamięci zaalokowany w linii 41 na karcie grafiki.
- Pętla testująca w liniach 44–52 składa się z transferu danych z CPU do GPU, wywołania funkcji rdzenia `pow2_gpu()`, transferu bufora z GPU do CPU oraz funkcji synchronizacji `cudaThreadSynchronize()` zapewniającej całkowite wykonanie kernela w każdej iteracji pętli.
- W liniach 57–60 przeprowadzony został test porównawczy wyniku obliczeń przeprowadzonych na GPU oraz CPU, sprawdzający czy wszystkie elementy tablicy wynikowej są identyczne dla obu obliczeń.

Dalsza część programu obliczenia wykonuje na karcie grafiki ale transfer pamięci został wykonany przy użyciu pamięci zablokowanej przez stronicowaniem.

Listing 3.11. CUDA – Pamięć zablokowana przez stronicowaniem - alokacja przypięta GPU.

```
65  cudaMalloc((void**)&cu_vec, sizeof(float)*SIZE);
66
67  float* vec_pinned;
68  float* vec_pinned_res;
69
70  cudaHostAlloc((void**)&vec_pinned, sizeof(float)*SIZE,
71                cudaHostAllocDefault);
72  cudaHostAlloc((void**)&vec_pinned_res, sizeof(float)*SIZE,
73                cudaHostAllocDefault);
74  cudaThreadSynchronize();
75
76  memcpy(vec_pinned, vec, sizeof(float)*SIZE);
77
78  time1 = timeStamp();
79  for(int i=0; i<N_ITER; ++i)
80  {
81    cudaMemcpy(cu_vec, vec_pinned, sizeof(float)*SIZE,
82               cudaMemcpyHostToDevice);
83    pow2_gpu<<<BLOCKS, THREADS>>> (cu_vec, cu_vec);
84    cudaMemcpy(vec_pinned_res, cu_vec, sizeof(float)*SIZE,
85               cudaMemcpyDeviceToHost);
86    cudaThreadSynchronize();
87  }
88  time2 = timeStamp();
89  cout<<"GPU 2 time="<<(time2-time1)/N_ITER<<"[ms]"<<endl;
90
91  if( equal(vec2, vec2+SIZE, vec_pinned_res) )
92    cout<<"CPU and GPU vec are identical"<<endl;
93  else
94    cout<<"CPU and GPU vec are DIFFERENT"<<endl;
95
96  cudaFreeHost(vec_pinned);
97  cudaFreeHost(vec_pinned_res);
98  cudaFree(cu_vec);
```

W linii 65 został zaalokowany obszar pamięci GPU `cu_vec`, na którym zostaną przeprowadzone testy obliczeniowe. Dodatkowo w liniach 70 i 72 za pomocą funkcji:

```
cudaError_t cudaHostAlloc(void **pHost, size_t size,
                          unsigned int flags)
```

zostały zaalokowane dwa obszary `size` bajtów pamięci *hosta* zablokowanej przed stronicowaniem. Jest to odpowiednik funkcji `malloc` ale zapewniający przypięcie tego obszaru pamięci oraz śledzenie stronicowania pamięci przez sterownik CUDA. Pamięć ta z poziomu *hosta* jest traktowana w klasyczny sposób co zostało wykorzystane w linii 76 do skopiowania zawartości orygi-

nalnych danych z tablicy `vec` do nowo zaalokowanej pamięci `vec_pinned` za pomocą standardowej funkcji `memcpy()`.

Dzięki takiej alokacji w liniach 81 oraz 84 testu, podczas kopiowania danych w funkcji `cudaMemcpy()`, używany jest mechanizm DMA do przyspieszenia transferu.

Ostatnia część programu pokazuje użycie pamięci zabezpieczonej przed stronicowaniem do bezpośredniego dostępu z poziomu GPU do pamięci CPU. Ten mechanizm nazywa się „zero-kopiowaną pamięcią" (ang. *zero-copy memory*) i w ogólności nie wymaga, ze strony API, żadnego kopiowania danych pomiędzy GPU a *hostem*.

Listing 3.12. CUDA – Pamięć zablokowana przez stronicowaniem - zero-kopiowana pamięć.

```
100   float* cu_vec_res;
101
102   cudaHostAlloc((void**)&vec_pinned, sizeof(float)*SIZE,
103         cudaHostAllocMapped|cudaHostAllocWriteCombined);
104   cudaHostAlloc((void**)&vec_pinned_res, sizeof(float)*SIZE,
105                 cudaHostAllocMapped);
106
107   memcpy(vec_pinned, vec, sizeof(float)*SIZE);
108
109   cudaHostGetDevicePointer((void**)&cu_vec, vec_pinned, 0);
110   cudaHostGetDevicePointer((void**)&cu_vec_res,
111                            vec_pinned_res, 0);
112   cudaThreadSynchronize();
113
114   time1 = timeStamp();
115   for(int i=0; i<N_ITER; ++i)
116   {
117     pow2_gpu<<< blocks, threads >>> (cu_vec, cu_vec_res);
118     cudaThreadSynchronize();
119   }
120   time2 = timeStamp();
121   cout<<"GPU 3 time="<<(time2-time1)/N_ITER<<"[ms]"<<endl;
122
123   if( equal(vec2, vec2+SIZE, vec_pinned_res) )
124     cout<<"CPU and GPU vec are identical"<<endl;
125   else
126     cout<<"CPU and GPU vec are DIFFERENT"<<endl;
127
128   cudaFreeHost(vec_pinned);
129   cudaFreeHost(vec_pinned_res);
130   delete[] vec;
131   delete[] vec2;
132
133   return 0;
134 }
```

W tym przypadku funkcja alokująca pamięć po stronie CPU `cudaHostAlloc()` wymaga podania dodatkowej flagi w ostatnim parametrze o wartości `cudaHostAllocMapped` odwzorowującej alokowaną pamięć na przestrzeń adresową GPU.

- W linii 102 w funkcji `cudaHostAlloc()` została użyta dodatkowa flaga o wartości `cudaHostAllocWriteCombined`. Flaga ta może poprawić szybkość transferu danych przez szynę PCI Express w niektórych systemach, ale za cenę utraty szybkości odczytu danych przez procesor CPU. Zatem, jest to dobra opcja, gdy dane są przez procesor centralny jedynie zapisywane do takiego bufora a odczytywane przez GPU.

- W linii 107 do zaalokowanego bufora `vec_pinned` kopiowana jest zawartość oryginalnej tablicy `vec`. Z poziomu GPU tak zaalokowana pamięć jest dostępna jedynie po odwzorowaniu przestrzeni pamięci CPU na przestrzeń adresową GPU za pomocą funkcji:

  ```
  cudaError_t cudaHostGetDevicePointer(void** pDevice,
                      void* pHost, unsigned int flags)
  ```

  ustawiającej wskaźnik `pDevice` z przestrzeni adresowej GPU na odpowiedni adres pamięci `pHost` w przestrzeni adresowej CPU. Funkcja ta została dwukrotnie wykorzystana w liniach 107 i 108 przemapowując wskaźniki `cu_vec` oraz `cu_vec_res` na odpowiednie adresy po stronie *hosta* `vec_pinned` oraz `vec_pinned_res`.

- Posługując się tak odwzorowanymi wskaźnikami w pętli testowej w linii 117 wywoływana jest jedynie funkcja kernela `pow2_gpu`, bez zbędnych wywołań funkcji kopiujących.

3.3.2. OpenCL

Listingi programu OpenCL zostaną ograniczone w stosunku do kodu CUDA do elementów charakterystycznych dla tego środowiska.

Kod funkcji rdzenia został przedstawiony na listingu 3.13

Listing 3.13. OpenCL – Pamięć zablokowana przez stronicowaniem - kod kernela.

```
1 __kernel void pow2_gpu(__global float* in, __global float* out)
2 {
3   int i = get_global_id(0) + get_global_id(1) *
4                              get_global_size(0);
5   out[i] = in[i]*in[i];
6 }
```

Część programu zawierająca obliczenia na CPU pozostaje identyczna z programem 3.9. Poniżej przedstawiona została część odpowiedzialna za inicjalizację środowiska oraz za przeprowadzenie testu dla pełnego transferu pamięci.

Listing 3.14. OpenCL – Pamięć zablokowana przez stronicowaniem – klasyczna alokacja GPU.

```
39  size_t GLOBAL_WS[] = {256,256};
40  size_t LOCAL_WS[] = {256,1};
41
42  cl_platform_id platform;
43  cl_device_id devices;
44  cl_context context;
45  cl_command_queue cmdQueue;
46  cl_program hProgram;
47  cl_kernel hKernel;
48  cl_mem cl_vec, cl_vec2;
49  float *p_vec, *p_vec2;
50
51  clGetPlatformIDs(1, &platform, NULL);
52  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
53                 &devices, 0);
54  context = clCreateContext(0, 1, &devices, 0,0,0);
55
56  size_t kernelLength;
57  char* programSource = loadProgSource("pow2.cl", "",
58                                       &kernelLength);
59  cmdQueue = clCreateCommandQueue(context, devices, 0,0);
60  hProgram = clCreateProgramWithSource(context, 1,
61             (const char**)&programSource, &kernelLength, 0);
62  clBuildProgram(hProgram, 0, 0, 0, 0, 0);
63  hKernel = clCreateKernel(hProgram,"pow2_gpu",0);
64
65  cl_vec = clCreateBuffer(context, CL_MEM_READ_WRITE,
66                          SIZE*sizeof(float),  0,0);
67  cl_vec2 = clCreateBuffer(context, CL_MEM_READ_WRITE,
68                           SIZE*sizeof(float),  0,0);
69
70  time1 = timeStamp();
71  for(int i=0; i<N_ITER; ++i)
72  {
73    clEnqueueWriteBuffer(cmdQueue, cl_vec,  CL_FALSE, 0,
74                 SIZE*sizeof(float), vec, 0, NULL, NULL);
75    clSetKernelArg(hKernel, 0, sizeof(cl_mem), &cl_vec);
76    clSetKernelArg(hKernel, 1, sizeof(cl_mem), &cl_vec2);
77    clEnqueueNDRangeKernel(cmdQueue, hKernel, 2, 0,
78                           GLOBAL_WS, LOCAL_WS, 0, 0, 0);
79    clEnqueueReadBuffer(cmdQueue, cl_vec2, CL_FALSE, 0,
80           SIZE*sizeof(float), vec_res, 0, NULL, NULL);
```

```
81      clFinish(cmdQueue);
82    }
83    time2 = timeStamp();
84    cout<<"GPU 1 time:"<<(time2-time1)/N_ITER<<"[ms]"<<endl;
```

W klasycznym przypadku wymagane jest stworzenie bufora pamięci dla źródłowych `cl_vec` oraz wynikowych `cl_vec2` danych w liniach 65 i 67. Sam test jest przeprowadzany w liniach 71–82 w `N_ITER` iteracjach.

Wykorzystanie pamięci zabezpieczonej przed stronicowaniem wymaga, analogicznie jak dla środowiska CUDA, alokacji pamięci przypiętej za pomocą dedykowanej funkcji OpenCL.

Listing 3.15. OpenCL – Pamięć zablokowana przez stronicowaniem – alokacja przypięta GPU.

```
86    cl_mem pinned_vec, pinned_vec2;
87
88    pinned_vec = clCreateBuffer(context, CL_MEM_READ_WRITE |
89                 CL_MEM_ALLOC_HOST_PTR, SIZE*sizeof(float), 0,0);
90    pinned_vec2 =clCreateBuffer(context, CL_MEM_READ_WRITE |
91                 CL_MEM_ALLOC_HOST_PTR, SIZE*sizeof(float), 0,0);
92
93    p_vec = (float*)clEnqueueMapBuffer(cmdQueue, pinned_vec,
94                    CL_TRUE, CL_MAP_WRITE|CL_MAP_READ, 0,
95                    SIZE*sizeof(float), 0, 0, 0, 0);
96    p_vec2 = (float*)clEnqueueMapBuffer(cmdQueue, pinned_vec2,
97                     CL_TRUE, CL_MAP_WRITE|CL_MAP_READ, 0,
98                     SIZE*sizeof(float), 0, 0, 0, 0 );
99    clFinish(cmdQueue);
100
101   memcpy(p_vec, vec, SIZE*sizeof(float));
102
103   time1 = timeStamp();
104   for(int i=0; i<N_ITER; ++i)
105   {
106     clEnqueueWriteBuffer(cmdQueue, cl_vec,  CL_FALSE, 0,
107                          SIZE*sizeof(float), p_vec, 0, 0, 0);
108     clSetKernelArg(hKernel, 0, sizeof(cl_mem), &cl_vec);
109     clSetKernelArg(hKernel, 1, sizeof(cl_mem), &cl_vec2);
110     clEnqueueNDRangeKernel(cmdQueue, hKernel, 2, 0,
111                            GLOBAL_WS, LOCAL_WS, 0, 0, 0);
112     clEnqueueReadBuffer(cmdQueue, cl_vec2, CL_FALSE, 0,
113                         SIZE*sizeof(float), p_vec2, 0, 0, 0);
114     clFinish(cmdQueue);
115   }
116   time2 = timeStamp();
117   cout<<"GPU 2 time:"<<(time2-time1)/N_ITER<<"[ms]"<<endl;
118
```

```
119   clEnqueueUnmapMemObject(cmdQueue, pinned_vec, p_vec, 0,0,0);
120   clEnqueueUnmapMemObject(cmdQueue, pinned_vec2,p_vec2,0,0,0);
121
122   clReleaseMemObject(cl_vec);
123   clReleaseMemObject(cl_vec2);
124   clReleaseMemObject(pinned_vec);
125   clReleaseMemObject(pinned_vec2);
```

- Aby zaalokować pamięć przypiętą funkcja `clCreateBuffer()`, w drugim parametrze przyjmuje dodatkową flagę `CL_MEM_ALLOC_HOST_PTR`. W liniach 88–91 zostały w ten sposób zaalokowane w pamięci *hosta* dwa pomocnicze bufory `pinned_vec` oraz `pinned_vec2`.
- Ich użycie w przestrzeni adresowej *hosta* będzie możliwe dopiero po odwzorowaniu ich z przestrzeni adresowej karty grafiki. W liniach 93–98 wskaźnikom `p_vec` oraz `p_vec2` za pomocą funkcji:

  ```
  void* clEnqueueMapBuffer(cl_command_queue command_queue,
                       cl_mem buffer, cl_bool blocking_map,
                    cl_map_flags map_flags, size_t offset,
                size_t cb, cl_uint num_events_in_wait_list,
                           const cl_event *event_wait_list,
                     cl_event *event, cl_int *errcode_ret)
  ```

  przypisane zostały odpowiednie adresy.
- W linii 101 do obszaru pamięci przypiętej `p_vec` zostały skopiowane dane z tablicy `vec` za pomocą standardowej funkcji `memcpy()`.
- Podczas testu w liniach 106 i 112 kopiowanie danych pomiędzy pamięcią CPU i GPU zachodzi już z wykorzystaniem mechanizmu dostępu bezpośredniego do pamięci (DMA).
- W liniach 119 i 120 pozostaje jeszcze odmapowanie wskaźników `p_vec` i `p_vec2` oraz usunięcie odpowiednich buforów w liniach 122–125.

Wykorzystanie mechanizmu pamięci zero-kopiowanej w środowisku OpenCL umożliwia korzystanie przez GPU z pamięci *hosta* zarówno dla standardowo alokowanej pamięci jak i pamięci zabezpieczonej przed stronicowaniem. Następny przykład pokazuje użycie pamięci zero-kopiowanej razem z pamięcią przypiętą.

Listing 3.16. OpenCL – Pamięć zablokowana przez stronicowaniem – zero-kopiowana pamięć.

```
127   pinned_vec = clCreateBuffer(context, CL_MEM_READ_ONLY |
128                CL_MEM_ALLOC_HOST_PTR, SIZE*sizeof(float), 0,0);
129   pinned_vec2 =clCreateBuffer(context, CL_MEM_WRITE_ONLY |
130                CL_MEM_ALLOC_HOST_PTR, size*sizeof(float), 0,0);
131
132   p_vec = (float*)clEnqueueMapBuffer(cmdQueue, pinned_vec,
133           CL_TRUE, CL_MAP_READ, 0, SIZE*sizeof(float),0,0,0,0);
134   p_vec2 =(float*)clEnqueueMapBuffer(cmdQueue, pinned_vec2,
135           CL_TRUE, CL_MAP_WRITE, 0,SIZE*sizeof(float),0,0,0,0);
136
137   memcpy(p_vec, vec, SIZE*sizeof(float));
138
139   cl_vec = clCreateBuffer(context, CL_MEM_READ_ONLY |
140            CL_MEM_USE_HOST_PTR, SIZE*sizeof(float), p_vec, 0);
141   cl_vec2 =clCreateBuffer(context, CL_MEM_WRITE_ONLY |
142            CL_MEM_USE_HOST_PTR, SIZE*sizeof(float), p_vec2,0);
143
144   time1 = timeStamp();
145   for(int i=0; i<N_ITER; ++i)
146   {
147     clSetKernelArg(hKernel, 0, sizeof(cl_mem), &cl_vec);
148     clSetKernelArg(hKernel, 1, sizeof(cl_mem), &cl_vec2);
149     clEnqueueNDRangeKernel(cmdQueue, hKernel, 2, 0,
150                            GLOBAL_WS, LOCAL_WS, 0,0,0);
151     clFinish(cmdQueue);
152   }
153   time2 = timeStamp();
154   cout<<"GPU 2 time:"<<(time2-time1)/N_ITER<<"[ms]"<<endl;
155
156   clEnqueueUnmapMemObject(cmdQueue, pinned_vec, p_vec, 0,0,0);
157   clEnqueueUnmapMemObject(cmdQueue, pinned_vec2,p_vec2,0,0,0);
158
159   clReleaseMemObject(cl_vec);
160   clReleaseMemObject(cl_vec2);
161   clReleaseMemObject(pinned_vec);
162   clReleaseMemObject(pinned_vec2);
163
164   return 0;
165 }
```

- W liniach 127–130 alokowana jest pamięć przypięta dzięki użyciu flagi `CL_MEM_ALLOC_HOST_PTR`, a w liniach 132–135 uzyskiwane są wskaźniki w przestrzeni adresowej CPU na odpowiednie porcje pamięci.
- W linii 137 kopiowana jest zawartość bufora `vec` do pamięci przypiętej `p_vec`.

- Zasadniczy element pamięci zero-kopiowanej znajduje się w liniach 139–142. Tworzone są tu dwa bufory, jeden do odczytu `cl_vec` i jeden do zapisu `cl_vec2` przy użyciu dodatkowej flagi `CL_MEM_USE_HOST_PTR`. Flaga ta informuje system, że *urządzenie obliczeniowe* ma, dla tego bufora, korzystać z pamięci *hosta*.
- W liniach 145–152 przeprowadzony został test bez jawnego kopiowania danych pomiędzy *hostem* i *urządzeniem*.
- Na koniec pozostaje jeszcze odmapowanie wskaźników w liniach 156 i 157 oraz usunięcie wszystkich stworzonych buforów w liniach 159–162.

3.3.3. Podsumowanie

Program został skompilowany i uruchomiony na komputerze wyposażonym w procesor klasy Intel Core2 Quad oraz w kartę grafiki opartą na procesorze NVIDIA GeForce GTX560. Dla obu środowisk czas obliczeń był niemal identyczny a różnice były rzędu błędu statystycznego.

Uśredniony czas obliczeń (i transferu dla GPU) dla pojedynczej iteracji przedstawia się następująco:
— 95 [ms] – dla obliczeń CPU funkcji `pow2_cpu()`,
— 84 [ms] – dla obliczeń przeniesionych na GPU z klasycznym transferem pamięci,
— 25 [ms] – dla obliczeń przeniesionych na GPU z pamięcią zablokowaną przed stronicowaniem,
— 24 [ms] – dla obliczeń przeniesionych na GPU z zero-kopiowaną pamięcią.

Czas wykonania funkcji rdzenia, oscylujący w okolicy 1,35[ms] jest w zasadzie pomijalny i z dobrym przybliżeniem można przyjąć powyższe czasy za czasy transferu danych pomiędzy *hostem* a *urządzeniem*.

Zysk z zastosowania pamięci przypiętej jest ponad trzykrotny w stosunku do klasycznego transferu i prawie czterokrotny w stosunku do obliczeń na CPU. Czas transferu danych pamięci przypiętej i zero-kopiowanej jest w ogólności taki sam. Nie powinno to budzić zdziwienia, ponieważ w przypadku pamięci zero-kopiowanej dane i tak muszą zostać niejawnie przesłane do pamięci globalnej znajdującej się na karcie grafiki, a czas tego transferu będzie identyczny jak w przypadku jawnego kopiowania. Zysk z użycia pamięci zero-kopiowanej będzie odczuwalny w sytuacji gdy karta graficzna współdzieli swoją pamięć z pamięcią *hosta*, co ma miejsce często w systemach ze zintegrowaną kartą grafiki oraz w systemach wbudowanych. Wtedy dzięki wykorzystaniu tego samego obszaru pamięci nie ma potrzeby przeprowadzania jakiegokolwiek transferu.

W przypadku środowiska OpenCL test pamięci zero-kopiowanej dał średni rezultat równy 4 [ms] (przy 100 iteracjach testowych). Jest to wartość sześciokrotnie mniejsza od testu przeprowadzonego dla pamięci przypiętej. Różnica ta wynika ze sposobu działania sterownika OpenCL, który przechowuje w pamięci podręcznej (ang. *cache*) dane, w przypadku użycia pamięci zero-kopiowanej. Test był przeprowadzony w pętli wykonującej się 100 razy (`N_ITER`) a niejawny transfer odbył się tylko dla pierwszej iteracji przy wczytaniu danych i dla ostatniej przy ich zapisie do pamięci *hosta*. Przy jednokrotnej iteracji testu wynik był identyczny jak dla pamięci przypiętej i oscylował w okolicy 25 [ms].

Rozdział 4

Język CUDA C

4.1. Wstęp

Język CUDA C jest oparty na standardzie języka C rozszerzając jednocześnie ten standard o dodatkowe typy danych, wbudowane zmienne oraz funkcje, które mogą być wykonywane równolegle na procesorze zgodnym z architekturą CUDA. Wprowadza również nową składnię definiowania meta-parametrów fukncji rdzenia. Niniejszy rozdział przedstawia jedynie te dodatkowe elementy rozszerzające własności standardowego języka C.

4.2. Typy kwalifikatorów

Kwalifikatory typu funkcji specyfikują czy dana funkcja może być wykonana na *hoście*, karcie grafiki czy na obu urządzeniach.
Kwalifikator funkcji:

```
__host__
```

określa funkcję, która może być wywołana jedynie na *hoście* i jedynie z poziomu innej funkcji *hosta*. Kwalifikator ten może być pominięty.
Kwalifikator funkcji:

```
__device__
```

określa funkcję, która może być wywołana jedynie na urządzeniu i jedynie z poziomu innej funkcji urządzenia.
Kwalifikator:

```
__global__
```

specyfikuje funkcję będącą kernelem. Taka funkcja może wykonywać się jedynie na urządzeniu, natomiast wywoływana jest jedynie z poziomu innej funkcji *hosta*. Funkcja tego typu musi zwracać typ `void`. Jej wywołanie, za pomocą dedykowanej składni, następuje asynchronicznie w stosunku do *hosta*, tzn. funkcja kernela powraca do wątku *hosta*, z którego została wywołana, natychmiast, nie czekając na jej ukończenie na urządzeniu.

Każda funkcja, która nie ma jawnie podanego kwalifikatora, niejawnie jest funkcją typu `__host__`. Kwalifikatory `__host__` oraz `__device__` można łączyć ze sobą. W takim przypadku kompilator wygeneruje dwie wersje funkcji, jedną, którą można wykonać na *hoście* i drugą możliwą do użycia na *urządzeniu*.

Kwalifikatory:

```
__noinline__
__forceinline__
```

służą do wyraźnego specyfikowania czy dana funkcja typu `__device__` nie ma być czy musi być funkcją wstawioną.

Kwalifikatory zmiennych specyfikują położenie danej zmiennej w konkretnym typie pamięci urządzenia. Tabela 4.1 zawiera listę oraz opis wszystkich kwalifikatorów zmiennych zdefiniowanych w języku CUDA C.

Tabela 4.1: Kwalifikatory zmiennych z języku CUDA C.

Kwalifikator	Opis
`__shared__`	Deklaruje zmienną rezydująca w pamięci współdzielonej urządzania, która ma długość życia bloku i jest dostępna dla wszystkich wątków w obrębie danego bloku. W przypadku zmiennych tablicowych typu `__shared__` wielkość tablicy musi być znana w momencie uruchomienia aplikacji.
`__constant__`	Deklaruje zmienną rezydująca w pamięci stałej urządzania, która ma długość życia aplikacji i jest dostępna dla wszystkich wątków w siatce oraz z poziomu *hosta* za pomocą funkcji `cudaGetSymbolAddress()`, `cudaGetSymbolSize()`, `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`.
`__device__`	Deklaruje zmienne rezydujące na urządzeniu. Jeżeli jest użyty razem z powyższymi kwalifikatorami posiada wszystkie ich własności, w innym przypadku, zmienne tego typu mają długość życia aplikacji, są dostępne dla wszystkich wątków w siatce oraz z poziomu *hosta* za pomocą funkcji `cudaGetSymbolAddress()`, `cudaGetSymbolSize()`, `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`.
`__restrict__`	Działanie identyczne jak dla wskaźników typu `restricted` w standardzie języka C99.

4.3. Podstawowe typy danych

Język CUDA C przejmuje wszystkie typy proste zdefiniowane w standardzie języka C i rozszerza ten zbiór o typy wektorowe. Typy wektorowe zostały wywiedzione z typów prostych całkowitoliczbowych oraz zmiennoprzecinkowych. Nazwa takiego typu składa się z nazwy typu podstawowego

oraz liczby określającej ilość komponentów danego wektora. W większości przypadków ilość możliwych komponentów to 1, 2, 3 lub 4. Dla typów wywiedzionych z `longlong`, `ulonglong` oraz `double` możliwe są jedynie 1 lub 2 komponenty.

Tabela 4.2 zawiera zestawienie wszystkich typów wektorowych zdefiniowanych w języku CUDA C wraz z ich opisem.

Tabela 4.2: Typy wektorowe w języku CUDA C. Literał n może przyjmować wartość `1`, `2`, `3` lub `4`, literał m może przyjmować wartości `1` lub `2`.

Typ	Opis
`char`n	8-bitowy n-elementowy wektor liczb całkowitych ze znakiem
`uchar`n	8-bitowy n-elementowy wektor liczb całkowitych bez znaku
`short`n	16-bitowy n-elementowy wektor liczb całkowitych ze znakiem
`ushort`n	16-bitowy n-elementowy wektor liczb całkowitych bez znaku
`int`n	32-bitowy n-elementowy wektor liczb całkowitych ze znakiem
`uint`n	32-bitowy n-elementowy wektor liczb całkowitych bez znaku
`long`n	32-bitowy (jeżeli `sizeof(long)== sizeof(int)`) lub 64-bitowy n-elementowy wektor liczb całkowitych ze znakiem
`ulong`n	32-bitowy (jeżeli `sizeof(ulong)== sizeof(uint)`) lub 64-bitowy n-elementowy wektor liczb całkowitych bez znaku
`longlong`m	64-bitowy n-elementowy wektor liczb całkowitych ze znakiem
`ulonglong`m	64-bitowy n-elementowy wektor liczb całkowitych bez znaku
`float`n	32-bitowy n-elementowy wektor liczb zmiennoprzecinkowych pojedynczej precyzji
`double`m	64-bitowy n-elementowy wektor liczb zmiennoprzecinkowych podwójnej precyzji.

Komponenty wektora są dostępne przez pola struktury wektora dla komponentu: 1 – `.x`, 2 – `.y`, 3 – `.z`, 4 – `.w`. Wszystkie typy wektorowe mają konstruktor o następującej składni:

```
type make_type(type p1, type p2, ...)
```

gdzie literał `type` może przyjmować jedną z wartości z tabeli 4.2. Przykłady użycia typów wektorowych w CUDA C:

```
float4 vf = make_float4(1.0, 2.0, 3.0, 4.0);
vf.y = 5.0;
int2 vi = make_int2(0, 0);
vi.x = (int)vf.x;     // jawne rzutowanie typu float na typ int
vf.y = vi.x;          // niejawne rzutowanie typu int na typ float
```

4.4. Zmienne wbudowane

Dla opisu rozmiaru bloku i siatki język CUDA C wprowadza nowy typ

```
dim3
```

oparty na typie wektorowym liczbowym całkowitym `uint3`. W odróżnieniu od typu `uint3`, podczas definicji zmiennej typu `dim3`, wszystkie jawnie niewyspecyfikowane komponenty są inicjalizowane wartością 1.

Zmienne opisujące rozmiar oraz aktualny indeks bloku oraz siatki są dostępne jedynie z poziomu funkcji wykonywanej na urządzeniu. Tabela 4.3 zawiera listę oraz opis wszystkich wbudowanych zmiennych CUDA C.

Tabela 4.3: Zmienne wbudowane języka CUDA C

Zmienna	Opis
`gridDim`	Zmienna typu `dim3` zawierająca rozmiar siatki bloków.
`blockIdx`	Zmienna typu `uint3` zawierająca indeks aktualnego bloku w siatce.
`blockDim`	Zmienna typu `dim3` zawierająca rozmiar bloku.
`threadIdx`	Zmienna typu `uint3` zawierająca indeks aktualnego wątku w bloku.
`warpSize`	Zmienna typu `int` zawierająca rozmiar *warpa*.

4.5. Funkcje wbudowane

Wbudowana funkcja:

```
void __syncthreads()
```

możliwa do wywołania jedynie w funkcji rdzenia, blokuje wykonywanie wątków od tego punktu w bloku, dopóki wszystkie wątki tego bloku nie dotrą do punktu synchronizacji oraz dopóki wszystkie wątki tego bloku nie zakończą dostępu do pamięci globalnej i współdzielonej.

Funkcja `__syncthreads()` jest używana do koordynacji komunikacji pomiędzy wątkami tego samego bloku. W sytuacji, gdy kilka wątków próbuje zapisu/odczytu z tej samej komórki pamięci globalnej lub współdzielonej, istnieje ryzyko wystąpienia problemu odczyt-po-zapisie, zapis-po-odczycie lub zapis-po-zapisie. Tego ryzyka można uniknąć synchronizując wątki w razie potrzeby dostępu do pamięci.

Urządzenie zgodne z *Compute Capability* 2.x definiują dodatkowe funkcje synchronizacji

```
int __syncthreads_count(int predicate)
int __syncthreads_and(int predicate)
int __syncthreads_or(int predicate)
```

które podobną funkcjonalność do funkcji `__syncthreads()` ale dodatkowo, każdy wątek oblicza wartość argumentu `preticate` i zwraca, odpowiednio dla funkcji: (1) ilość wątków, dla których `predicate` miał wartość niezerową, (2) wartość niezerową jeżeli `predicat` miał wartość niezerową dla wszystkich wątków, (3) wartość niezerową jeżeli `predicat` miał wartość niezerową dla któregokolwiek z wątków.

Wbudowana funkcja:

```
void __threadfence_block()
```

możliwa do wywołania jedynie w funkcji rdzenia, blokuje wykonywanie wątków bloku od tego punktu, dopóki nie zakończą się wszystkie odwołania do pamięci globalnej i współdzielonej w obrębie bloku wątków.
Funkcja:

```
void __threadfence()
```

blokuje wykonywanie wątków bloku od tego punktu, dopóki nie zakończą się, dla funkcji wywołującej, wszystkie odwołania do pamięci współdzielonej

w obrębie danego bloku wątków oraz pamięci globalnej w obrębie wszystkich wątków w siatce na danym urządzeniu.

Dla urządzeń o *Compute Capability* 2.x została zdefiniowana dodatkowa funkcja:

```
void __threadfence_system()
```

blokująca wykonywanie wątków bloku od tego punktu, dopóki nie zakończą się, dla funkcji wywołującej, wszystkie odwołania do pamięci współdzielonej w obrębie danego bloku wątków, pamięci globalnej w obrębie wszystkich wątków w siatce na danym urządzeniu oraz odwołania do pamięci pamięci zabezpieczonej przed stronicowaniem dla wątków *hosta*.

4.6. Funkcje matematyczne

Język CUDA C definiuje szereg standardowych funkcji matematycznych, które mogą być wywoływane na *hoście* i *urządzeniu* lub tylko na *urządzeniu*. Obsługa liczb zmiennoprzecinkowych podwójnej precyzji `double` została wprowadzona w urządzeniach zgodnych z *Compute Cabability* 1.3 i wyższych.

Tabela 4.4 zawiera listę wszystkich funkcji matematycznych dla liczb zmiennoprzecinkowych pojedynczej precyzji `float`, które mogą być uruchamiane z poziomu *hosta* i *urządzania*.

Tabela 4.4: Standardowe funkcje matematyczne dla liczb zmiennoprzecinkowych pojedynczej precyzji `float` (tylko lista).

x+y	x*y	x/y	1/x
rsqrtf(x)	1/sqrtf(x)	sqrtf(x)	cbrtf(x)
rcbrtf(x)	hypotf(x,y)	expf(x)	exp2f(x)
exp10f(x)	expm1f(x)	logf(x)	log2f(x)
log10f(x)	log1pf(x)	sinf(x)	cosf(x)
tanf(x)	sincosf(x,sptr,cptr)	sinpif(x)	cospif(x)
asinf(x)	acosf(x)	atanf(x)	atan2f(y,x)
sinhf(x)	coshf(x)	tanhf(x)	asinhf(x)
acoshf(x)	atanhf(x)	powf(x,y)	erff(x)
erfcf(x)	erfinvf(x)	erfcinvf(x)	erfcxf(x)
lgammaf(x)	tgammaf(x)	fmaf(x,y,z)	frexpf(x,exp)
ldexpf(x,exp)	scalbnf(x,n)	scalblnf(x,l)	logbf(x)
ilogbf(x)	j0f(x)	j1f(x)	jnf(x)

y0f(x)	y1f(x)	ynf(x)	fmodf(x,y)
remainderf(x,y)	remquof(x,y,iptr)	modff(x,iptr)	fdimf(x,y)
truncf(x)	roundf(x)	rintf(x)	nearbyintf(x)
ceilf(x)	floorf(x)	lrintf(x)	lroundf(x)
llrintf(x)	llroundf(x)		

Tabela 4.5 zawiera listę wszystkich funkcji matematycznych dla liczb zmiennoprzecinkowych podwójnej precyzji `double`, które mogą być uruchamiane z poziomu *hosta* i *urządzania*.

Tabela 4.5: Standardowe funkcje matematyczne dla liczb zmiennoprzecinkowych podwójnej precyzji `double` (tylko lista).

x+y	x*y	x/y	1/x
sqrt(x)	rsqrt(x)	cbrt(x)	rcbrt(x)
hypot(x,y)	exp(x)	exp2(x)	exp10(x)
expm1(x)	log(x)	log2(x)	log10(x)
log1p(x)	sin(x)	cos(x)	tan(x)
sincos(x,sptr,cptr)	sinpi(x)	cospi(x)	asin(x)
acos(x)	atan(x)	atan2(y,x)	sinh(x)
cosh(x)	tanh(x)	asinh(x)	acosh(x)
atanh(x)	pow(x,y)	erf(x)	erfc(x)
erfinv(x)	erfcinv(x)	erfcx(x)	lgamma(x)
tgamma(x)	fma(x,y,z)	frexp(x,exp)	ldexp(x,exp)
scalbn(x,n)	scalbln(x,l)	logb(x)	ilogb(x)
j0(x)	j1(x)	jn(x)	y0(x)
y1(x)	yn(x)	fmod(x,y)	remainder(x,y)
remquo(x,y,iptr)	modf(x,iptr)	fdim(x,y)	trunc(x)
round(x)	rint(x)	nearbyint(x)	ceil(x)
floor(x)	lrint(x)	lround(x)	llrint(x)
llround(x)			

Funkcje dedykowane do wykonania tylko na urządzeniu są wysoce zoptymalizowanymi odpowiednikami standardowych funkcji matematycznych. Ceną za szybkość wykonania jest jednakże dokładność obliczeń. Specyfikacja języka CUDA C [6] sugeruje użycie tych funkcji, tylko w przypadku gdy

niezbędna jest szybkość wykonania a zredukowana dokładność może być tolerowana.

W przypadku niektórych funkcji matematycznych wykonywanych tylko na urządzaniu istnieje możliwość wyboru trybu zaokrąglenia. Dodanie do nazwy funkcji przyrostka `_rn` powoduje, że funkcja stosuje zaokrąglenia do najbliższej liczby parzystej. Funkcje z przyrostkiem `_rz` działają z zaokrągleniem w kierunku zera. Funkcje z przyrostkiem `_ru` działają z zaokrągleniem w kierunku plus nieskończoności. Funkcje z przyrostkiem `_rd` działają z zaokrągleniem w kierunku nieskończoności.

Tabela 4.6 zawiera listę optymalizowanych funkcji matematycznych dla liczb zmiennoprzecinkowych pojedynczej precyzji `float`, które mogą być uruchamiane tylko z poziomu funkcji *urządzania*.

Tabela 4.6: Funkcje matematyczne dla liczb zmiennoprzecinkowych pojedynczej precyzji `float` dedykowane tylko dla *urządzenia* (tylko lista).

`__fadd_[rn,rz,ru,rd](x,y)`	`__fmul_[rn,rz,ru,rd](x,y)`
`__fmaf_[rn,rz,ru,rd](x,y,z)`	`__frcp_[rn,rz,ru,rd](x)`
`__fsqrt_[rn,rz,ru,rd](x)`	`__fdiv_[rn,rz,ru,rd](x,y)`
`__fdividef(x,y)`	`__expf(x)`
`__exp10f(x)`	`__logf(x)`
`__log2f(x)`	`__log10f(x)`
`__sinf(x)`	`__cosf(x)`
`__sincosf(x,sptr,cptr)`	`__tanf(x)`
`__powf(x, y)`	

Tabela 4.7 zawiera listę optymalizowanych funkcji matematycznych dla liczb zmiennoprzecinkowych podwójnej precyzji `double`, które mogą być uruchamiane tylko z poziomu *urządzania*.

Tabela 4.7: Funkcje matematyczne dla liczb zmiennoprzecinkowych podwójnej precyzji `double` dedykowane tylko dla *urządzenia* (tylko lista).

`__dadd_[rn,rz,ru,rd](x,y)`	`__dmul_[rn,rz,ru,rd](x,y)`
`__fma_[rn,rz,ru,rd](x,y,z)`	`__ddiv_[rn,rz,ru,rd](x,y)(x,y)`
`__drcp_[rn,rz,ru,rd](x)`	`__dsqrt_[rn,rz,ru,rd](x)`

Rozdział 5

Język OpenCL C

5.1. Wstęp

Język OpenCL C służy do tworzenia programów wykonujących wysoce zrównoleglone funkcje rdzeni. Funkcje te mogą zostać wykonane na wielu heterogenicznych urządzeniach, takich jak CPU, GPU czy innych dedykowanych akceleratorach. Sam program OpenCL można opisać przez podobieństwo do biblioteki łączonej dynamicznie, a funkcje kerneli do funkcji eksportowanych przez taką bibliotekę. Jednakże, w przypadku biblioteki łączonej dynamicznie, funkcje przez nią eksportowane mogą być wywoływane przez *hosta* bezpośrednio, natomiast w przypadku OpenCL funkcje kernela są kolejkowane w kolejce poleceń na urządzeniu obliczeniowym i wykonywane asynchronicznie razem z kodem *hosta*.

Język OpenCL C jest oparty na standardzie ISO/IEC 9899:1999 języka C z kilkoma wyjątkami oraz rozszerzeniami języka związanymi z przystosowaniem go do wielowątkowości. Dokładnie, w stosunku do języka C (C99) w języku OpenCL C pojawiły się następujące rozszerzenia:

— Wektorowe typy danych operujące na kilku (2, 3, 4, 8 lub 16) liczbach całkowitych lub zmiennoprzecinkowych. Zmienne typu wektorowego mogą być używane w analogiczny sposób jak zmienne typów prostych.
— Kwalifikatory przestrzeni adresowej – używane do identyfikacji konkretnego typu pamięci (zostały opisane w rozdziale 3).
— Typy i funkcje wspierające równoległe wykonywanie kodu, tj. obsługa *work-items*, *work-groups*, synchronizacji.
— Typy obrazowe reprezentujące obraz oraz typ umożliwiający próbkowanie obrazu (zostały opisane w rozdziale 6).
— Bogaty zbiór wbudowanych funkcji matematycznych, geometrycznych czy relacyjnych operujących na typach prostych, wektorowych oraz obrazach.

5.2. Słowa kluczowe języka OpenCL C

W języku OpenCL zostały zdefiniowane następujące słowa kluczowe, które nie mogą zostać użyte w nazwach własnych typów, zmiennych i funkcji:

— nazwy zastrzeżone jako słowa kluczowe w standardzie języka C99;
— kwalifikatory przestrzeni adresowej: `__global`, `global`, `__local`, `local`, `__constant`, `constant`, `__private`, `private`;
— kwalifikatory dostępu: `__read_only`, `read_only`, `__write_only`, `write_only`, `__read_write`, `read_write`;

Dodatkowo zdefiniowano dwa kwalifikatory:

```
__kernel
kernel
```

specyfikujące funkcję będącą kernelem OpenCL. Jedynym sposobem wywołania takiej funkcji jest zakolejkowanie jej za pomocą funkcji OpenCL API `clEnqueueNDRangeKernel()`.

Oprócz wymienionych powyżej słów kluczowych, zarezerwowane są wszystkie nazwy wymienione w niniejszym rozdziale w tabelach 5.1, 5.2, 5.3, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11.

5.3. Podstawowe typy danych

5.3.1. Typy skalarne

Język OpenCL obsługuje standardowe typy skalarne języka C, tj. liczby całkowite jedno-, dwu-, cztero-, oraz ośmio-bajtowe, ze znakiem i bez, oraz zmiennoprzecinkowe połówkowej, pojedynczej oraz podwójnej precyzji. Dla obsługi liczb zmiennoprzecinkowych o połówkowej precyzji (`half`), dane urządzenie obliczeniowe musi obsługiwać rozszerzenie `cl_khr_fp16`, a dla obsługi liczb zmiennoprzecinkowych o podwójnej precyzji (`double`), dane urządzenie obliczeniowe musi obsługiwać rozszerzenie `cl_khr_fp64`. Tabela 5.1 zawiera typy proste języka OpenCL C wraz z ich opisem i odpowiednimi typami dostępnymi z poziomu aplikacji w API OpenCL.

Tabela 5.1: Typy skalarne w języku OpenCL C wraz z odpowiednikami w API OpenCL.

Typ	Opis	Odp. API
`bool`	Typ boolowski o wartościach `true` lub `false`, rzutowalny na typ całkowity o wartości 1 dla `true` oraz 0 dla `false`	nd.
`char`	8-bitowy typ całkowity ze znakiem o wartościach $[-128, 127]$	`cl_char`
`unsigned char`, `uchar`	8-bitowy typ całkowity bez znaku o wartościach $[0, 255]$	`cl_uchar`
`short`	16-bitowy typ całkowity ze znakiem o wartościach $[-32768, 32767]$	`cl_short`
`unsigned short`, `ushort`	8-bitowy typ całkowity bez znakum o wartościach $[0, 65535]$	`cl_ushort`
`int`	32-bitowy typ całkowity ze znakiem o wartościach $[-2147483648, 2147483647]$	`cl_int`
`unsigned int`, `uint`	32-bitowy typ całkowity bez znaku o wartościach $[0, 4294967295]$	`cl_uint`

`long`	64-bitowy typ całkowity ze znakiem w uzupełnieniu do dwóch	`cl_long`
`unsigned long`, `ulong`	64-bitowy typ całkowity bez znaku	`cl_ulong`
`float`	32-bitowy typ zmiennoprzecinkowy zgodny z *IEEE 754 single-precision storage format*	`cl_float`
`double`	64-bitowy typ zmiennoprzecinkowy zgodny z *IEEE 754 double-precision storage format*. Typ dostępny jedynie dla urządzeń obsługujących rozszerzenie `cl_khr_fp64`	`cl_double`
`half`	16-bitowy typ zmiennoprzecinkowy zgodny z *IEEE 754-2008 half-precision storage format*. Typ dostępny jedynie dla urządzeń obsługujących rozszerzenie `cl_khr_fp16`	`cl_half`
`size_t`	32-bitowy typ całkowity bez znaku dla urządzeń o 32-bitowej przestrzeni adresowej lub 64-bitowy typ całkowity bez znaku dla urządzeń o 64-bitowej przestrzeni adresowej	nd.
`ptrdiff_t`	typ całkowity ze znakiem opisujący różnicę dwóch wskaźników o wielkości typu `sizeof`	nd.
`void`	Typ reprezentujący pusty zbiór wartości	`void`

Typ zmiennoprzecinkowy o połówkowej precyzji `half` może być użyty jedynie przy deklaracji wskaźnika na bufor zawierający wartości typu `half`, np:

```
void fun(global half* p) {...}

global half ptr*;
fun(ptr);
```

Nie jest dozwolone tworzenie obiektów automatycznych tego typu, tzn. nie są dozwolone deklaracje:

```
half v;
half tab[100];
```

Do wstawiania i odczytu zmiennych typu `half` do/z bufora służą funkcje `vloadhalf()` i `vloahalfn()` oraz `vstorehalf()` i `vstorehalfn()` dokonujące automatycznej konwersji typu `hafl` do `float`.

5.3.2. Typy wektorowe

Typy wektorowe zostały zdefiniowane jedynie dla typów liczbowych. Nazwa typu wektorowego składa się z nazwy typu prostego oraz liczby definiującej ilość komponentów danego wektora, tj. `type`n. Dopuszczalna ilość komponentów n to 2, 3, 4, 8 lub 16.

Tabela 5.2 zawiera wektorowe typy języka OpenCL wraz z ich opisem i odpowiednimi typami dostępnymi z poziomu aplikacji w API OpenCL.

Tabela 5.2: Typy wektorowe w języku OpenCL C wraz z odpowiednikami w API OpenCL.

Typ	Opis	Odp. API
`char`n	8-bitowy n-elementowy wektor liczb całkowitych ze znakiem	`cl_char`n
`uchar`n	8-bitowy n-elementowy wektor liczb całkowitych bez znaku	`cl_uchar`n
`short`n	16-bitowy n-elementowy wektor liczb całkowitych ze znakiem	`cl_short`n
`ushort`n	16-bitowy n-elementowy wektor liczb całkowitych bez znaku	`cl_ushort`n
`int`n	32-bitowy n-elementowy wektor liczb całkowitych ze znakiem	`cl_int`n
`uint`n	32-bitowy n-elementowy wektor liczb całkowitych bez znaku	`cl_uint`n
`long`n	64-bitowy n-elementowy wektor liczb całkowitych ze znakiem	`cl_long`n
`ulong`n	64-bitowy n-elementowy wektor liczb całkowitych bez znaku	`cl_ulong`n
`float`n	32-bitowy n-elementowy wektor liczb zmiennoprzecinkowych pojedynczej precyzji	`cl_float`n
`double`n	64-bitowy n-elementowy wektor liczb zmiennoprzecinkowych podwójnej precyzji. Typ dostępny jedynie dla urządzeń obsługujących rozszerzenie `cl_khr_fp64`	`cl_double`n

`half`n	16-bitowy n-elementowy wektor liczb zmiennoprzecinkowych połówkowej precyzji. Typ dostępny jedynie dla urządzeń obsługujących rozszerzenie `cl_khr_fp16`	`cl_half`n

Nazwa typu wektorowego może posłużyć do utworzenia obiektu danego typu z innych skalarów, wektorów lub kombinacji skalarów i wektorów za pomocą odpowiedniego konstruktora, np:

```
(float4)(float, float, float, float)
(int4)(int2, int2)
(uchar4)(uchar2, uchar, uchar)
(long4)(long3, long)
```

Przy konstrukcji wektora nazwa typu wektorowego musi być otoczona nawiasami okrągłymi, np:

```
float2 f2 = (float2)(1.0, 2.0);
float3 f3 = (float3)(f2, 3.0);
float4 f4 = (float4)(f2, f2);
float4 f5 = (float4)(1.0, (float2)(1.0, 2.0), 4.0);
```

Odpowiednie komponenty typów wektorowych są dostępne analogicznie jak w typach złożonych za pomocą operatora . (kropka). W przypadku typów wektorowych złożonych z 2, 3 lub 4 komponentów dostęp do kolejnych elementów wektora jest realizowany za pomocą zmiennych `x` dla pierwszego elementu, `y` dla drugiego elementu, `z` dla trzeciego elementu i `w` dla czwartego elementu. Niedopuszczalne jest odwołanie do nieistniejącego elementu, np. do składowej `z` lub `w` wektora dwuelementowego. Poniżej przedstawione zostało kilka możliwych odwołań do elementów wektora:

```
float4 vec;
vec.x = 1.0f;
vec.xy = (float2)(1.0, 2.0);
vec.zw = vec.xy;
vec.xyzw = vec.wzyx;

float2 vec2;
vec2 = vec.xy;
vec2.yx = vec.zw;

vec2.z = 1.0f;     // instrukcja niedopuszczalna
vec2.xy = vec;     // instrukcja niedopuszczalna
```

Możliwe jest również odwołanie się do konkretnego komponentu wektora poprzez podanie numeru indeksu tego komponentu. Jest to równocześnie

jedyny sposób dla wektorów złożonych z większej niż 4 ilości komponentów. Dopuszczalne numery indeksów to `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, `F`. Wielkość liter nie jest uwzględniana. Sam numer indeksu musi być poprzedzony literą `s` lub `S`. Przykładowo:

```
float16 vec;
vec.s5 = 1.0f;  // 6 element wektora
vec.s7aF = (float3)(1.0, 2.0, 3.0);  // 8, 11 i 16 element wektora
```

Ten sposób odwoływania się do elementów nie może być mieszany z odwołaniem przez notację `.xyzw`.

```
vec.xs5 = vec.s01;    // instrukcja niedopuszczalna, mieszanie notacji
```

Za pomocą notacji `.lo` oraz `.hi` możliwy jest dostęp do odpowiednio dolnej lub górnej połowy danego wektora. Za pomocą notacji `.odd` oraz `.even` możliwy jest dostęp do odpowiednio nieparzystych i parzystych elementów wektora.

```
float8 vec;
float4 vec2 = vec.lo;      // to samo co vec.s0123
float2 vec3 = vec2.odd;    // to samo co vec2.xz
```

W przypadku odwołań `.hi` oraz `.lo` dla wektora 3-elementowego, traktowany jest on jak wektor 4-elementowy z niezdefiniowanym czwartym elementem.

5.3.3. Inne typy

Inne typy wbudowane języka OpenCL zostały przedstawione w tabli 5.3

Tabela 5.3: Inne typy wbudowane w języku OpenCL C

Typ	Opis
`image_2d`	Typ reprezentujący obraz dwuwymiarowy
`image_3d`	Typ reprezentujący obraz trójwymiarowy
`sampler_t`	Typ reprezentujący próbnik obrazów
`event_t`	Typ zdarzeniowy. Może być użyty do identyfikacji asynchronicznego kopiowania pamięci z globalnej do lokalnej i odwrotnie.

Typy reprezentujące obrazy w języku OpenCL (`image_2d` i `image_3d`) oraz typ próbkujący obrazy `sampler_t` zostały omówione w rozdziale 6 *Współpraca z OpenGL*. Typ zdarzeniowy (`event_t`) został opisany w rozdziale 2.5 *Architektura środowisk CUDA i OpenCL*.

5.3.4. Konwersje typów

Dla typów liczbowych skalarnych zachodzi niejawna konwersja realizowana przez kompilator. Możliwa jest również jawna konwersja typów skalarnych liczbowych przeprowadzona za pomocą operatora rzutowania. Sama konwersja jest przeprowadzana zgodnie z regułami standardu języka C99. Przykłady:

```
float f = 5;        // niejawna konwersja int do float
int i = (int)f;     // jawna konwersja przez rzutowanie
```

Nie są dopuszczalne niejawne konwersje typów wektorowych. Do jawnej konwersji typów wektorowych (oraz skalarnych) służą wbudowane funkcje o następującej składni:

```
destType convert_destType[_sat][roundingMode](srcType val)
destTypen convert_destTypen[_sat][roundingMode](srcTypen val)
```

gdzie `destType` jest typem zwracanym, `roundingMode` opcjonalnym trybem zaokrąglenia (tabela 5.4 zawiera możliwe typy zaokrągleń), `srcType` typem konwertowanym. W przypadku konwersji do typu całkowitego możliwe jest wymuszenie wysycenia przez dodanie literału `_sat` po nazwie typu. Argument oraz typ zwracany musi mieć taką samą ilość komponentów.

Tabela 5.4: Typy zaokrągleń podczas konwersji jawnej

Modyfikator	Opis
`_rte`	Zaokrąglenie do najbliższej parzystej
`_rtz`	Zaokrąglenie w kierunku zera
`_rtp`	Zaokrąglenie w kierunku plus nieskończoności
`_rtn`	Zaokrąglenie w kierunku minus nieskończoności
brak modyfikatora	Dla typów całkowitych będzie użyty modyfikator `_rtz`, dla typów zmiennoprzecinkowych będzie użyty modyfikator `_rte`

Przykładowe konwersje:

```
int4 i;
float4 f = convert_float4( i );
float4 f = convert_float4_rtp( i );
uchar4 c = convert_uchar4_sat_rtz(f);
```

Możliwe jest również przeprowadzenie konwersji typów przez reinterpretację bitów danej zmiennej za pomocą wbudowanej funkcji:

```
type as_type(srcType val)
typen as_typen(srcTypen val)
```

Przykładowo:

```
float f = 1.0f;
uint u = as_uint(f);  //  u zawiera wartość 0x3f800000

float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
int4 i = as_int4(f);
 //  i zawiera wartości (0x3f800000, 0x40000000, 0x40400000, 0x40800000)

float4 f;
float3 g = as_float3(f);     //  g zawiera elementy f.xyz
```

5.4. Funkcje wbudowane

5.4.1. Funkcje operujące na *work-items*

Tabela 5.5 zawiera zestawienie wszystkich wbudowanych funkcji służących do pobierania informacji o ilości wymiarów, ilości *work-items* oraz *work-groups*.

Tabela 5.5: Funkcje operujące na *work-items*

Funkcja	Opis
`uint get_work_dim()`	Zwraca ilość wykorzystywanych wymiarów
`size_t get_global_size( uint dimindx )`	Zwraca całkowitą ilość *work-items* względem wymiaru `dimindx`. Dopuszczalne wartości `dimindx` to [0, `get_work_dim()-1`]
`size_t get_global_id( uint dimindx )`	Zwraca aktualny globalny indeks *work-item* dla danego wymiaru. Dopuszczalne wartości `dimindx` to [0, `get_work_dim()-1`]
`size_t get_local_size( uint dimindx )`	Zwraca lokalną ilość *work-items* względem wymiaru `dimindx`. Dopuszczalne wartości `dimindx` to [0, `get_work_dim()-1`]
`size_t get_local_id( uint dimindx )`	Zwraca aktualny lokalny indeks *work-item* dla danego wymiaru. Dopuszczalne wartości `dimindx` to [0, `get_work_dim()-1`]
`size_t get_num_groups( uint dimindx )`	Zwraca ilość *work-groups* dla danego wymiaru. Dopuszczalne wartości `dimindx` to [0, `get_work_dim()-1`]

`size_t get_group_id( uint dimindx )`	Zwraca aktualny indeks *work-group* dla danego wymiaru. Dopuszczalne wartości `dimindx` to [0, `get_work_dim()-1`]
`size_t get_global_offset( uint dimindx )`	Zwraca przesunięcie podane podczas wywołania `clEnueueNDRangeKernel()` dla danego wymiaru. Dopuszczalne wartości `dimindx` to [0, `get_work_dim()-1`]

5.4.2. Funkcje matematyczne

Poniższy podrozdział zawiera wybrane funkcje matematyczne operujące na skalarach i wektorach, pogrupowane na standardowe i ogólne funkcje matematyczne, funkcje geometryczne oraz funkcje matematyczne operujące tylko na liczbach całkowitych.

Tabela 5.6 zawiera listę i opis wbudowanych funkcji geometrycznych. W przypadku funkcji działających na typach wektorowych, wszystkie operacje przeprowadzane są dla każdego komponentu wektora osobno. Typ `floatn` może być każdym z typów: `float`, `float2`, `float3`, `float4`.

Tabela 5.6: Funkcje geometryczne

Funkcja	Opis
`float{3|4} cross(float{3|4}, float{3|4})`	Iloczyn wektorowy dwóch wektorów.
`float dot(floatn, floatn)`	Iloczyn skalarny dwóch wektorów.
`float distance(floatn, floatn)`	Euklidesowy dystans pomiędzy wektorami.
`float length(floatn)`	Euklidesowa długość wektora.
`floatn normalize(floatn)`	Znormalizowany wektor.
`float fast_distance(floatn, floatn)`	Odpowiednik funkcji `distance()` wykorzystujący algorytm przybliżony.
`float fast_length(floatn)`	Odpowiednik funkcji `length()` wykorzystujący algorytm przybliżony.
`floatn fast_normalize(floatn)`	Odpowiednik funkcji`normalize()` wykorzystujący algorytm przybliżony.

Tabela 5.7 zawiera listę wszystkich standardowych wbudowanych funkcji matematycznych, które mogą przyjmować w parametrze skalar lub wektor. Typ `type` może być każdym z typów: `float`, `float2`, `float3`, `float4`, `float8`, `float16`. Poza specyficznymi wyjątkami, w danej funkcji typ `type` musi być identyczny we wszystkich parametrach funkcji oraz typie zwracanym. W przypadku funkcji działających na typach wektorowych, wszystkie operacje przeprowadzane są dla każdego komponentu wektora osobno.

Tabela 5.7: Standardowe funkcje matematyczne (tylko lista).

`type acos(type)`	`type acosh(type)`	`type acospi(type)`
`type asin(type)`	`type asinh(type)`	`type asinpi(type)`
`type atan(type y_over_x)`	`type atan2(type y, type x)`	`type atanh(type)`
`type atanpi(type x)`	`type atan2pi(type y, type x)`	`type cbrt(type)`
`type ceil(type)`	`type copysign(type, type)`	`type cos(type)`
`type cosh(type)`	`type cospi(type x)`	`type erfc(type)`
`type erf(type)`	`type exp(type x)`	`type exp2(type)`
`type exp10 (type)`	`type expm1 (type x)`	`type fabs (type)`
`type fdim(type, type)`	`type floor (type)`	`type fma (type, type, type)`
`type fmax(type, type)`	`type fmax(type, float)`	`type fmin(type, type)`
`type fmin(type, float)`	`type fmod(type, type)`	`type fract(type, __global type*)`
`type fract(type, __local type*)`	`type fract(type, __private type*)`	`floatn frexp(floatn, __global intn*)`
`floatn frexp(floatn, __local intn*)`	`floatn frexp(floatn, __private intn*)`	`float frexp(float, __global int*)`
`float frexp(float, __local int*)`	`float frexp(float, __private int*)`	`type nextafter(type, type)`
`type hypot(type, type)`	`intn ilogb(floatn)`	`int ilogb(float)`
`floatn ldexp(floatn, intn)`	`floatn ldexp(floatn, int)`	`float ldexp(float, int)`
`type lgamma(type)`	`floatn lgamma_r(floatn, __global intn*)`	`floatn lgamma_r(floatn, __local intn*)`
`floatn lgamma_r(floatn, __private intn*)`	`float lgamma_r(float, __global int*)`	`float lgamma_r(float, __local int*)`
`float lgamma_r(float, __private int*)`	`type log(type)`	`type log2(type)`

type log10(type)	type log1p(type)	type logb(type)
type mad(type, type, type)	type maxmag(type, type)	type minmag(type, type)
type modf(type x, __global type*)	type modf(type, __local type*)	type modf(type, __private type*)
floatn nan(uintn)	float nan(uint)	type nextafter(type, type)
type pow(type, type)	floatn pown(floatn, intn)	float pown(float, int)
type powr(type, type)	type remainder(type, type)	floatn remquo(floatn, floatn, __global intn*)
floatn remquo(floatn, floatn,__local intn*)	floatn remquo(floatn, floatn,__private intn*)	float remquo(float, float, __global int*)
float remquo(float, float,__local int*)	float remquo(float, float,__private int*)	type rint(type)
floatn rootn(floatn, intn)	float rootn(float, int)	type round(type)
type rsqrt(type)	type sin(type)	type sincos(type, __global type*cosval)
type sincos(type, __local type*cosval)	type sincos(type, __private type*cosval)	type sinh(type)
type sinpi(type)	type sqrt(type)	type tan(type)
type tanh(type)	type tanpi(type)	type tgamma(type)
type trunc(type)		

Tabela 5.8 zawiera listę funkcji matematycznych ogólnego przeznaczenia. W przypadku funkcji działających na typach wektorowych, wszystkie operacje przeprowadzane są dla każdego komponentu wektora osobno. Typ `type` oznacza każdy z typów: `float`, `float2`, `float3`, `float4`, `float8`, `float16`.

Tabela 5.8: Funkcje matematyczne ogólne (tylko lista).

type clamp(type, type minval, type maxval)	type clamp(type, float minval, float maxval)	type degrees(type rad)
type max(type, type)	type max(type, float)	type min(type, type)
type min(type, float)	type mix(type, type, type a)	type mix(type, type, float a)
type radians(type deg)	type step(type edge, type)	type step(float edge, type)
type smoothstep(type edge0,type edge1,type)	type smoothstep(float edge0,float edge1,type)	type sign(type)

Tabela 5.9 zawiera listę wbudowanych funkcji matematycznych działających tylko na liczbach całkowitych, w postaci skalara lub wektora. Typ `type` oznacza każdy z typów: `char`, `char{2|3|4|8|16}`, `uchar`, `uchar{2|3|4|8|16}`, `short`, `short{2|3|4|8|16}`, `ushort`, `ushort{2|3|4|8|16}`, `int`, `int{2|3|4|8|16}`, `uint`, `uint{2|3|4|8|16}`, `long`, `long{2|3|4|8|16}`, `ulong`, `ulong{2|3|4|8|16}`. Typ `utype` oznacza wersję bez znaku każdego z typów `type`. Typ `stype` oznacza wersję ze znakiem każdego z typów `type`. Poza specyficznymi wyjątkami, w danej funkcji typ `type` musi być identyczny we wszystkich parametrach funkcji oraz typie zwracanym. W przypadku funkcji działających na typach wektorowych, wszystkie operacje przeprowadzane są dla każdego komponentu wektora osobno.

Tabela 5.9: Funkcje matematyczne działające tylko na liczbach całkowitych (tylko lista).

`utype abs(type)`	`utype abs_diff(type, type)`	`type add_sat(type, type)`
`type hadd(type, type)`	`type rhadd(type, type)`	`type clamp(type, type minval, type maxval)`
`type clamp(type, stype minval, stype maxval)`	`type clz(type)`	`type mad_hi(type a,type b, type c)`
`type mad_sat(type a, type b,type c)`	`type max(type, type)`	`type max(type, stype)`
`type min(type, type)`	`type min(type, stype)`	`type mul_hi(type, type)`
`type rotate(type, type)`	`type sub_sat(type,type)`	`short upsample(char hi, uchar lo)`
`ushort upsample(uchar hi, uchar lo)`	`shortn upsample(charn hi, ucharn lo)`	`ushortn upsample(ucharn hi, ucharn lo)`
`int upsample(short hi, ushort lo)`	`uint upsample(ushort hi, ushort lo)`	`intn upsample(shortn hi, ushortn lo)`
`uintn upsample(ushortn hi, ushortn lo)`	`long upsample(int hi, uint lo)`	`ulong upsample(uint hi, uint lo)`
`longn upsample(intn hi, uintn lo)`	`ulongn upsample(uintn hi, uintn lo)`	`type mad24(type, type, type)`
`type mul24(type, type)`		

5.4.3. Inne funkcje

Tabela 5.10 zawiera listę i opis funkcji synchronizacji.

Tabela 5.10: Funkcje synchronizacji.

Funkcja	Opis
`void barrier( cl_mem_fence_flags flags)`	Wszystkie *work-items* z danej grupy muszą wykonać tę funkcję zanim zostaną dopuszczone do wykonywania następnych instrukcji. Jeżeli bariera jest wewnątrz instrukcji warunkowej, wtedy wszystkie *work-items* muszą wejść do sekcji warunkowej o ile przynajmniej jeden *work-item* doszedł do bariery. Jeżeli bariera jest wewnątrz pętli, wszystkie *work-items* muszą wykonać barierę w każdej iteracji pętli zanim zostaną dopuszczone do wykonywania następnych instrukcji. Flaga `flag` może być jedną z lub kombinacją następujących wartości: (1) `CLK_LOCAL_MEM_FENCE` – funkcja bariery zadba o prawidłową obsługę dostępu do pamięci lokalnej, (2) `CLK_GLOBAL_MEM_FENCE` – funkcja bariery zadba o prawidłową obsługę dostępu do pamięci globalnej.

Tabela 5.11 zawiera listę oraz opis funkcji wykonujących operacje atomowe. Typ `itype` oznacza każdy z typów: `int`, `uint`. W zależności od literału `__access` operacje będą przeprowadzane na pamięci lokalnej `__local` lub globalnej `__global`.

Tabela 5.11: Funkcje atomowe.

Funkcja	Opis
`itype atomic_add(volatile __access itype*p, itype val)`	Zamienia wartość pod adresem `p` na wartość `(*p+val)`. Funkcja zwraca wartość `p`.
`itype atomic_sub(volatile __access itype*p, itype val)`	Zamienia wartość pod adresem `p` na wartość `(*p-val)`. Funkcja zwraca wartość `p`.

`itype atomic_xchg(volatile __access itype*p, itype val)` `float atomic_xchg(volatile __access float*p, float val)`	Zamienia wartość pod adresem `p` na wartość `val`. Funkcja zwraca wartość `p`.
`itype atomic_inc(volatile __access itype* p)`	Zamienia wartość pod adresem `p` na wartość `(*p+1)`. Funkcja zwraca wartość `p`.
`itype atomic_dec (volatile __access itype *p)`	Zamienia wartość pod adresem `p` na wartość `(*p-1)`. Funkcja zwraca wartość `p`.
`int atomic_cmpxchg (volatile __access int *p, int cmp, int val)`	Zamienia wartość pod adresem `p` na wartość `((*p==cmp)?val:*p)`. Funkcja zwraca wartość `p`.
`int atomic_min (volatile __access int *p, int val)`	Zamienia wartość pod adresem `p` na wartość `min(*p,val)`. Funkcja zwraca wartość `p`.
`int atomic_max (volatile __access int *p, int val)`	Zamienia wartość pod adresem `p` na wartość `max(*p,val)`. Funkcja zwraca wartość `p`.
`int atomic_and (volatile __access int *p, int val)`	Zamienia wartość pod adresem `p` na wartość `(*p & val)`. Funkcja zwraca wartość `p`.
`int atomic_or (volatile __access int *p, int val)`	Zamienia wartość pod adresem `p` na wartość `(*p | val)`. Funkcja zwraca wartość `p`.
`int atomic_xor (volatile __access int *p, int val)`	Zamienia wartość pod adresem `p` na wartość `(*p ^ val)`. Funkcja zwraca wartość `p`.

Rozdział 6

Współpraca z OpenGL

6.1. Wstęp

Niniejszy rozdział prezentuje możliwości kooperacji struktur OpenGL ze strukturami CUDA/OpenCL.

Współczesne karty graficzne zbudowane są w ten sposób, że pamięć GPU jest pewnym wspólnym obszarem, który będzie współdzielony pomiędzy procesami GPGPU oraz procesami przetwarzania grafiki. Oba te zadania mogą istnieć obok siebie lub wzajemnie się zazębiać. To znaczy, jako danych wejściowych procedur GPGPU można użyć struktur graficznych, takich jak, obiekty buforowe i tekstury, i odwrotnie potok graficzny może operować na danych pochodzących z procedur CUDA/OpenCL. Taka funkcjonalność jest realizowana przez odwzorowywanie zasobów OpenGL/Direct3D na przestrzeń adresową CUDA/OpenCL umożliwiając tym samym zapis i odczyt do tych zasobów.

W poniższym rozdziale zostanie omówiona jedynie zdolność współdziałania biblioteki OpenGL z CUDA/OpenCL. Jednakże, sposób kooperacji z Direct3D jest niemalże identyczny i nie odbiega zasadniczo od standardowych wywołań współpracy z OpenGL.

Do pełnego zrozumienia zawartości tego rozdziału niezbędna jest przynajmniej podstawowa znajomość biblioteki OpenGL i jej funkcjonalności. W bardzo wielu przypadkach, przedstawione przykładowe rozwiązania nie będą zawierały pełnego kodu i nie będą wyjaśnione we wszystkich szczegółach.

6.2. Ogólna struktura programu

Aby zademonstrować mechanizmy współpracy Cuda/OpenCL z OpenGL najpierw zostanie zdefiniowany ogólny szkielet programu zarządzającego OpenGL. Przydatna będzie również pomocnicza struktura przechowująca dane obrazu cyfrowego po stronie *hosta*.

Listing 6.1. OpenGL – Pomocnicza struktura przechowująca dane obrazu po stronie *hosta* – plik `image.h`.

```
1  struct Image
2  {
3    int width, height, pitch;
4    unsigned char* data;
5
6    Image() { data = NULL; }
7    Image(int w, int h) { create(w,h); }
8    ~Image() { delete[] data; }
9
10   void create(int w, int h)
11   {
12     width = w; height = h;
```

```
13      pitch = width*4;   // długość wiersza w bajtach
14      data = new unsigned char[size];
15    }
16
17    int size() { return height*pitch; }
18    void load(const char* fname);
19  };
```

Struktura `Image` będzie reprezentować obraz w po stronie *hosta*. Domyślnie stuktura ta będzie operować na obrazach 32-bitowych w formacie ARGB.

W 18 linii zadeklarowana została metoda `load()`, która wczytuje z podanego pliku obraz, alokuje odpowiednią ilość pamięci i ustawia odpowiednie pola tej struktury zgodnie z wartościami wczytywanego obrazu. Niestety z powodu olbrzymiej ilości typów i formatów plików graficznych implementacja tej metody pozostaje w rękach czytelnika. Istnieje wiele niezależnych i darmowych bibliotek obsługi popularnych formatów plików graficznych takich jak: `libpng` do obsługi plików PNG, `libjpeg` do obsługi plików JPG czy większe narzędzia typu `ImageMagick` obsługujące większość popularnych typów plików.

Do obsługi okna i kontekstu OpenGL zostanie użyta międzyplatformowa biblioteka GLUT.

Listing 6.2. OpenGL – Ogólna struktura programu.

```
1  #define GL_GLEXT_PROTOTYPES
2  #include <GL/gl.h>
3  #include <GL/glu.h>
4  #include <GL/glut.h>
5  #include "image.h"
6
7  Image image;
8  GLuint gl_buff;
9  GLuint gl_tex;
10
11 void initGL();
12 void render();
13 void copyBufferToTexture();
14 void key_event(int key, int x, int y);
15
16 int main(int argc, char *argv[])
17 {
18   glutInit(&argc, argv);
19   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
20   glutInitWindowSize(512, 512);
21   glutCreateWindow("OpenGL Interoperability");
```

```
22   glutDisplayFunc(render);
23   glutSpecialFunc(key_event);
24
25   initGL();
26   copyBufferToTexture();
27
28   glutMainLoop();
29   return 0;
30 }
```

Początkowa funkcjonalność programu ograniczy się do inicjalizacji biblioteki GLUT w liniach 18–23, inicjalizacji OpenGL w funkcji `initGL()`, oraz przekopiowania zawartości bufora `gl_buff` do tekstury `gl_tex` w funkcji `copyBufferToTexture()`. W funkcji `render()` zostanie narysowany prostokąt teksturowany obrazem zawartym w teksturze `gl_tex`. Funkcja `keyEvent()` będzie zawierała obsługę zdarzenia naciśnięcia przycisku na klawiaturze. Wywołana w linii 28 funkcja `glutMainLoop()` rozpoczyna pętlę reakcji na zdarzenia.

Funkcja inicjalizująca OpenGL przedstawiona jest na listingu 6.3.

Listing 6.3. OpenGL – Funkcja inicjalizacji.

```
32 void initGL();
33 {
34   glClearColor(0.0, 0.0, 0.0, 0.0);
35   glDisable(GL_DEPTH_TEST);
36   glDisable(GL_LIGHTING);
37   glEnable(GL_TEXTURE_2D);
38   glOrtho(0.0, 1.0, 0.0, 1.0, 0.0, 1.0);
39
40   image.load("texture");
41
42   glGenBuffers(1, &gl_buff);
43   glBindBuffer(GL_PIXEL_UNPACK_BUFFER, gl_buff);
44     glBufferData(GL_PIXEL_UNPACK_BUFFER, image.size,
45                  image.data, GL_DYNAMIC_DRAW);
46   glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
47
48   glGenTextures(1, &gl_tex);
49   glBindTexture(GL_TEXTURE_2D, gl_tex);
50     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
51                     GL_NEAREST );
52     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
53                     GL_NEAREST );
54     glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
55     glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
56     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image.width,
57                  image.height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
58                  NULL);
```

```
59   glBindTexture(GL_TEXTURE_2D, 0);
60 }
```

Po konfiguracji stanów OpenGL, w linii 40 wczytany został obraz z pliku `"texture"` do globalnego obiektu `image`.

W liniach 42–46 tworzony jest bufor `gl_buff` reprezentujący obraz po stronie serwera OpenGL. Do tego bufora wczytana została zawartość obrazu `image`.

W liniach 48–59 tworzony jest obiekt tekstury 2D `gl_tex` typu `RGBA` o wielkości równej wielkości obrazu `image`. Ta tekstura zostanie użyta do wyświetlenia przetworzonego obrazu w oknie aplikacji.

Listing 6.4. OpenGL – Funkcje `render()` i `copyBufferToTexture()`.

```
62 void render()
63 {
64   glClear(GL_COLOR_BUFFER_BIT);
65   glBindTexture(GL_TEXTURE_2D, gl_tex);
66   glBegin(GL_QUADS);
67     glTexCoord2f(0.0f, 1.0f); glVertex3f(0, 0, 0);
68     glTexCoord2f(1.0f, 1.0f); glVertex3f(1, 0, 0);
69     glTexCoord2f(1.0f, 0.0f); glVertex3f(1, 1, 0);
70     glTexCoord2f(0.0f, 0.0f); glVertex3f(0, 1, 0);
71   glEnd();
72   glBindTexture(GL_TEXTURE_2D, 0);
73   glutSwapBuffers();
74 }
75
76 void copyBufferToTexture()
77 {
78   glBindTexture(GL_TEXTURE_2D, gl_tex);
79   glBindBuffer(GL_PIXEL_UNPACK_BUFFER, gl_buff);
80     glTexSubImage2D(GL_TEXTURE_2D, 0,0,0, image.width,
81         image.height, GL_BGRA, GL_UNSIGNED_BYTE, NULL);
82   glBindTexture(GL_TEXTURE_2D, 0);
83   glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
84 }
```

Funkcja `render()` związuje aktualną teksturę z obiektem `gl_tex` i rysuje prostokąt o wielkości całego widoku z ustawioną teksturą.

Funkcja `copyBufferToTexture()` kopiuje zawartość obiektu buforowego `gl_buff` do tekstury `gl_tex`.

6.3. Realizacja LookUp Table w CUDA

Pierwszy program ilustrujący współpracę CUDA i OpenGL będzie lekką modyfikacją programu szkieletowego. W tym przypadku CUDA zostanie wykorzystana do modyfikacji wartości obrazu cyfrowego za pomocą tablicy LUT (ang. *LookUp Table*). Funkcjonalność programu będzie się sprowadzała do zmiany jasności i kontrastu wyświetlanego obrazu za pomocą klawiszy strzałek. Na listingach 6.5–6.8 znajdują się niezbędne zmienne oraz funkcje rozszerzające program szkieletowy.

Listing 6.5. CUDA – Inicjalizacja CUDA w kontekście OpenGL.

```
1  texture<uchar4, cudaTextureType2D> tex_arr;
2  cudaArray* image_arr;
3  cudaChannelFormatDesc format;
4  cudaGraphicsResource* cu_buff;
5  int* cu_lut;
6
7  int lut[256];
8  int brightness = 0;
9  int contrast = 0;
10
11 void initCUDA()
12 {
13   cudaGLSetGLDevice(0);
14
15   format = cudaCreateChannelDesc(8,8,8,8,
16            cudaChannelFormatKindUnsigned);
17   cudaMallocArray(&image_arr, &format, image.width,
18                   image.height);
19   cudaMemcpy2DToArray(image_arr, 0,0, image.data, image.pitch,
20            image.pitch, image.height, cudaMemcpyHostToDevice);
21
22   cudaGraphicsGLRegisterBuffer(&cu_buff, gl_buff,
23           cudaGraphicsRegisterFlagsWriteDiscard);
24 }
```

W linii 1 utworzony został globalny obiekt typu:

```
texture<DataType, Type, ReadMode> texRef
```

będący referencją na teksturę dostępną z poziomu funkcji kernela CUDA. Referencja tekstury musi być statycznym globalnym obiektem. Typ `DataType` określa typ opisujący punkty obrazu i jest ograniczony do podstawowych typów całkowitych lub zmiennoprzecinkowych, pojedynczej precyzji oraz ich typów wektorowych o 1-, 2- lub 4 elementach. Typ `Type` określa typ tekstury i może przyjmować jedną z kilku wartości: `cudaTextureType1D`, `cudaTextureType2D` lub `cudaTextureType3D` opisujących kolejno 1-, 2- lub 3-wymiarową teksturę.

Typ `ReadMode` określa opcjonalną konwersję typu całkowitego do typu zmiennoprzecinkowego.

Funkcja `initCUDA()` inicjalizuje kontekst CUDA oraz związuje go z kontekstem OpenGL. W linii 13 została w tym celu wywołana funkcja `cudaGLSetGLDevice()`, której w parametrze podano wartość 0, oznaczającą pierwsze urządzenie obliczeniowe. Przyjęto w tym przypadku ciche założenie, że urządzenie o numerze 0 istnieje w systemie i jest zgodne z CUDA.

Do obsługi obrazów w CUDA zdefiniowany jest specjalny typ `cudaArray` umożliwiający obsługę obrazów 1-, 2- oraz 3-wymiarowych. W powyższym przykładzie zostanie utworzony obraz 2D `cudaArray* image_arr`, w którym każdy piksel jest opisany czterema 8-bitowymi liczbami całkowitymi bez znaku. Format ten jest zdefiniowany w linii 15 za pomocą zmiennej `cudaChannelFormatDesc format`.

W linii 17 alokowana jest pamięć na obraz `image_arr` za pomocą funkcji `cudaMallocArray()` poprzez podanie w parametrach wywołania formatu obrazu oraz jego szerokości i wysokości.

W linii 19 obraz `image` z pamięci *hosta* jest kopiowany do obrazu `image_arr` za pomocą funkcji `cudaMemcpy2DToArray()`.

W linii 22 za pomocą funkcji:

```
cudaError_t cudaGraphicsGLRegisterBuffer(
            struct cudaGraphicsResource** resource,
            GLuint buffer,
            unsigned int flags)
```

bufor OpenGL o nazwie `buffer` jest rejestrowany do użycia przez CUDA. Z poziomu GPU będzie on dostępny przez zmienną zwróconą w parametrze `cudaGraphicsResource** resource`. Użyta w przykładzie flaga `cudaGraphicsRegisterFlagsWriteDiscard` określa, że bufor nigdy nie będzie czytany z poziomu GPU, a zapis nastąpi na całej zawartości bufora wymazując poprzednie dane w nim zawarte.

Główna część algorytmu odwzorowującego kolory względem tablicy LUT jest przedstawiona na listingu 6.6.

Listing 6.6. CUDA – Funkcja `lutmap()`.

```
27 void lutmap()
28 {
29   for(int i=0; i<256; ++i)
30   {
31     int v = i + brightness;
32     lut[i] = v + contrast - v*contrast/127;
33   }
34   cudaMalloc(&cu_lut, 256*sizeof(int));
35   cudaMemcpy(cu_lut, lut, 256*sizeof(int),
```

```
36              cudaMemcpyHostToDevice);
37
38   glFinish();
39   uchar4* buff_ptr;
40   cudaGraphicsMapResources(1, &cu_buff);
41   cudaGraphicsResourceGetMappedPointer((void**)&buff_ptr,
42                                        NULL, cu_buff);
43   cudaBindTextureToArray(tex_arr, image_arr, format);
44
45   dim3 threads(16,16);
46   dim3 blocks(image.width/16, image.height/16);
47   lut_kernel<<<blocks, threads>>>(buff_ptr, image.width,
48                                   cu_lut);
49
50   cudaFree(cu_lut);
51   cudaGraphicsUnmapResources(1, &cu_buff);
52   cudaUnbindTexture(tex_arr);
53
54   glBindTexture(GL_TEXTURE_2D, gl_tex);
55   glBindBuffer(GL_PIXEL_UNPACK_BUFFER, gl_buff);
56   glTexSubImage2D(GL_TEXTURE_2D, 0, 0,0, image.width,
57       image.height, GL_BGRA, GL_UNSIGNED_BYTE, NULL);
58   glBindTexture(GL_TEXTURE_2D, 0);
59   glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
60 }
```

W liniach 29–33 tworzone jest odwzorowanie LUT uwzględniające zmianę wartości jasności (globalna zmienna `brightness`) oraz kontrastu (globalna zmienna `contrast`). W obrazach, w których na opis każdego kanału przypada jedna 8-bitowa liczba wystarczy tablica LUT o 256 elementach. Tablica ta jest następnie kopiowana do pamięci globalnej GPU wskazywanej przez zmienną `cu_lut` zaalokowanej w linii 34.

Aby, z poziomu CUDA, użyć zarejestrowanego bufora `cu_buff` należy odwzorować jego adres na odpowiedni wskaźnik w przestrzeni adresowej CUDA za pomocą funkcji `cudaGraphicsMapResources()` oraz `cudaGraphicsResourceGetMappedPointer()`. W ten sposób w liniach 39–42 wskaźnik `buff_ptr` ustawiony został na obiekt `cu_buff`.

W linii 43 za pomocą funkcji `cudaBindTextureToArray()` następuje związanie obrazu `image_arr` z globalną referencją na teksturę `tex_arr`. Dopiero po takim związaniu, obiekt `cudaArray` jest dostępny, z poziomu funkcji kernela, do odczytu za pomocą specjalnych funkcji pobierających wartości punktów.

W dalszej części funkcji uruchamiana jest funkcja kernela `lut_kernel<<<...>>>()` przetwarzająca dane znajdujące się w obiekcie związanym z teksturą `tex_arr`. Dane te zostaną zapisane do bufora `cu_buff` związanego z buforem OpenGL `gl_buff`.

W liniach 50–52 zwalniane są odpowiednie zasoby i wiązania.

W liniach 54–59 bufor `gl_buff` kopiowany jest do tekstury `gl_tex`, która zostanie narysowana w funkcji `render()`.

Potrzebna jeszcze jest definicja funkcji `key_event()`, obsługującej zdarzenia naciśnięcia klawiszy strzałek oraz funkcja `main()`.

Listing 6.7. CUDA – Funkcja obsługi zdarzeń GLUT.

```
62 void key_event(int key, int x, int y)
63 {
64   switch(key)
65   {
66     case GLUT_KEY_UP:   brightness+=5; break;
67     case GLUT_KEY_DOWN: brightness-=5; break;
68     case GLUT_KEY_RIGHT:contrast-=5;   break;
69     case GLUT_KEY_LEFT: contrast+=5;   break;
70   }
71   lutmap();
72   render();
73 }
74
75 int main(int argc, char *argv[])
76 {
77   glutInit(&argc, argv);
78   glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
79   glutInitWindowSize(512, 512);
80   glutCreateWindow("Test Window");
81   glutDisplayFunc(render);
82   glutSpecialFunc(key_event);
83
84   initGL();
85   initCUDA();
86   lutmap();
87   glutMainLoop();
88   return 0;
89 }
```

W funkcji `main()` zostały dodane jedynie dwie linie, 85 – funkcja `initCUDA()` inicjalizująca CUDA oraz 86 – pierwsze wywołanie funkcji `lutmap()`.

Na koniec funkcje kernela przetwarzające punkty obrazu.

Listing 6.8. CUDA – Funkcja rdzenia `lut_kernel()`.

```
1 inline __device__ __host__ int clamp(int f, int a, int b)
2 {
3     return max(a, min(f, b));
4 }
```

```
5
6  __global__ void lut_kernel(uchar4* buff, int w, const int* lut)
7  {
8    int2 pos = make_int2(threadIdx.x + blockIdx.x*blockDim.x,
9                         threadIdx.y + blockIdx.y*blockDim.y);
10
11   uchar4 color = tex2D(tex_arr, pos.x, pos.y);
12   uchar4 pixel;
13   pixel.x = clamp(lut[color.x], 0, 255);
14   pixel.y = clamp(lut[color.y], 0, 255);
15   pixel.z = clamp(lut[color.z], 0, 255);
16   pixel.w = color.w;
17   buff[pos.x + pos.y*w] = pixel;
18 }
```

W liniach 1–4 zdefiniowana została pomocnicza funkcja `clamp()` ograniczająca wartość zmiennej `f` do przedziału domkniętego $[a, b]$. Dzięki użyciu specyfikatorów `__device__ __host__` funkcja ta może być wykorzystana zarówno w kodzie wykonywanym przez CPU jak i przez funkcje kernela.

Warto zauważyć, że funkcja rdzenia `lut_kernel()` nie ma parametru zawierającego obraz wejściowy, a jedynie wynikowy bufor, szerokość linii obrazu oraz tablicę `lut`.

Dostęp do punktów wejściowego obrazu realizowany jest przez globalną referencję na teksturę `tex_arr` zdefiniowaną na listingu 6.5 w linii 1 a związaną z konkretnym obiektem obrazu `image_arr` w linii 43 na listingu 6.6. Funkcją pobierającą wartości punktów z obrazu tekstury jest:

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2D(texture<DataType,cudaTextureType2D,readMode> texRef,
           float x, float y);
```

która wymaga podania referencji tekstury oraz współrzędnych odczytywanego punktu. Sposób interpretacji obrazu tekstury przez tą funkcję jest w pełni uzależniony od konfiguracji referencji tekstury zdefiniowanej w kodzie *hosta*. Na listingu 6.5 w 1 linii została ta referencja zdefiniowana zdefiniowana przez:

```
texture<uchar4, cudaTextureType2D> tex_arr;
```

co oznacza teksturę 2-wymiarową 4-kanałową, w której na kanał przypada 8-bitowa liczba całkowita bez znaku. Z poziomu *hosta* dostępnych jest jeszcze kilka możliwych opcji konfiguracyjnych dostępu do tekstury. Na listingu 6.9 podano przykład konfiguracji referencji tekstury z filtracją liniową pomiędzy tekselami, znormalizowanymi współrzędnymi tekstury oraz odbiciem lustrzanym tekstury przy odczycie punktu spoza tekstury.

Listing 6.9. CUDA – Przykład konfiguracji referencji tekstury.

```
1  int main()
2  {
3    ...
4    tex_arr.filterMode = cudaFilterModeLinear;
5    tex_arr.normalized = true;
6    tex_arr.addressMode[0] = cudaAddressModeMirror;
7    tex_arr.addressMode[1] = cudaAddressModeMirror;
8    ...
9  }
10
11 __global__ void kernel(...)
12 {
13   ...
14   float2 pos = make_float2(-0.1f, 1.3f);
15   float4 color = tex2D(tex_arr, pos.x, pos.y);
16   ...
17 }
```

Przy znormalizowanych współrzędnych tekstury (`tex_arr.normalized = true`) poprawne współrzędne tekstury zawierają się w domkniętym przedziale $[0.0, 1.0]$. Tymczasem w funkcji kernela podano ujemną współrzędną $x = -0.1$ oraz współrzędną $y = 1.3$ wykraczającą poza teksturę. Dzięki odbiciu lustrzanemu współrzędnych (`cudaAddressModeMirror`) te wartości zostaną zamienione na wartości $x = 0.1$ i $y = 0.7$.

W kodzie programu pominięto część odpowiedzialną za zwalnianie zaalokowanych zasobów oraz zakończenie samego programu.

6.4. Filtracja uśredniająca w OpenCL

Filtracja uśredniająca obrazu dwuwymiarowego jest realizowana za pomocą operacji splotu obrazu z funkcją maski filtru. Przez maskę będziemy rozumieć dwuwymiarową kwadratową tablicę o wielkości $d = 2r + 1$ gdzie r jest promieniem maski. Wszystkie wartości maski są ustawione na wartość równą 1. W efekcie takiej filtracji każdy punkt obrazu zostanie zastąpiony średnią arytmetyczną punktów znajdujących się w jego sąsiedztwie o promieniu r, powodując w ten sposób wrażenie rozmycia obrazu.

Z własności operacji splotu wielowymiarowych funkcji wynika, że tego typu filtracja może być zrealizowana przez separację wymiarów, tzn. najpierw mogą być przefiltrowane wszystkie wiersze obrazu maską jednowymiarową

poziomą a następnie kolumny obrazu również maską jednowymiarową ale pionową. Zabieg taki zmniejsza złożoność obliczeniową z $O(n^2)$ do $O(2n)$.

Najpierw niezbędne jest zdefiniowanie kilku dodatkowych funkcji po stronie *hosta* inicjalizujących OpenCL oraz wywołujących funkcje kernela.

Listing 6.10. OpenCL – Inicjalizacja OpenCL w kontekście OpenGL.

```
1  cl_platform_id platform;
2  cl_device_id devices;
3  cl_context context;
4  cl_command_queue cmdQueue;
5  cl_program hProgram;
6  cl_kernel hBoxFilter;
7  cl_kernel vBoxFilter;
8
9  cl_mem cl_tex;
10 cl_mem cl_image;
11 cl_mem cl_image_temp;
12
13 int radius = 1;
14
15 void initCL()
16 {
17   clGetPlatformIDs(1, &platform, NULL);
18   cl_uint num_dev;
19   clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &devices,
20                  &num_dev);
21
22 #if defined unix  // UNIX
23   cl_context_properties props[] =
24   {
25     CL_GL_CONTEXT_KHR,   (cl_context_properties)
26                          glXGetCurrentContext(),
27     CL_GLX_DISPLAY_KHR,  (cl_context_properties)
28                          glXGetCurrentDisplay(),
29     CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
30     0
31   };
32 #else    // Windows
33   cl_context_properties props[] =
34   {
35     CL_GL_CONTEXT_KHR,   (cl_context_properties)
36                          wglGetCurrentContext(),
37     CL_WGL_HDC_KHR,      (cl_context_properties)
38                          wglGetCurrentDC(),
39     CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
40     0
41   };
42 #endif
43
```

```
44   context = clCreateContext(props, 1, &devices, 0,0,0);
45   size_t kernelLength;
46   char* programSource = loadProgSource("kernels.cl", "",
47                                        &kernelLength);
48   cmdQueue = clCreateCommandQueue(context, devices, 0,0);
49   hProgram = clCreateProgramWithSource(context, 1,
50              (const char**)&programSource, &kernelLength, 0);
51
52   clBuildProgram(hProgram, 0, 0, 0, 0, 0);
53
54   hBoxFilter = clCreateKernel(hProgram, "hbox", 0);
55   vBoxFilter = clCreateKernel(hProgram, "vbox", 0);
56
57   cl_image_format im_format;
58   im_format.image_channel_data_type = CL_UNSIGNED_INT8;
59   im_format.image_channel_order = CL_BGRA;
60
61   cl_image = clCreateImage2D(context, CL_MEM_READ_ONLY,
62              &im_format, image.width, image.height,
63              image.pitch, image.data, 0);
64   cl_image_temp = clCreateImage2D(context, CL_MEM_READ_WRITE,
65                   &im_format, image.height, image.width,0,0,0);
66
67   glBindTexture(GL_TEXTURE_2D, gl_tex);
68   cl_tex = clCreateFromGLTexture2D(context, CL_MEM_WRITE_ONLY,
69                                    GL_TEXTURE_2D, 0, gl_tex, 0);
70   glBindTexture(GL_TEXTURE_2D, 0);
71 }
```

Sama inicjalizacja środowiska przebiega dosyć standardowo, tzn. tworzony jest obiekt platformy, uzyskiwane są kompatybilne urządzenia oraz tworzony jest kontekst OpenCL. Jednakże, w tym przypadku, kontekst OpenCL musi być powiązany z kontekstem OpenGL. W tym celu tworzona jest pomocnicza tablica:

```
cl_context_properties props[]
```

zawierająca listę odpowiednich dla systemu operacyjnego wartości opisujących kontekst OpenGL. W liniach 23–31 tablica ta jest uzupełniana odpowiednimi wartościami w przypadku systemu Linux a w liniach 33–41 dla systemu Windows. Lista wartości może być ustawiona na `NULL` lub składać się z listy par: nazwy własności i wartości tej własności. W takim przypadku ostatnią wartością tablicy musi być znak `0`.

W linii 44 tworzony jest obiekt kontekstu przez podanie wyspecyfikowanej powyżej tablicy własności. Taki kontekst może być prawidłowo utworzony tylko wtedy gdy istnieje już w danym wątku prawidłowy kontekst OpenGL i jest on kontekstem aktywnym.

W liniach 54–55 tworzone są dwa obiekty funkcji rdzenia, pierwszy dla przejścia pionowego obrazu a drugi dla przejścia poziomego.

W liniach 57–65 tworzone są dwa obiekty obrazu OpenCL. Najpierw wyspecyfikowany został format obrazu w zmiennej `cl_image_format im_format` przez podanie typu danych i ilości/kolejności kanałów obrazu. Następnie, za pomocą funkcji `clCreateImage2D()` tworzone są dwa obrazy. Pierwszy obraz `cl_image` będzie zawierał obraz oryginalny, tzn. będzie kopią obrazu `image` po stronie GPU. Dane tego obrazu są od razu kopiowane do pamięci tekstury w wywołaniu konstruktora obrazu. Drugi obraz `cl_image_temp` będzie obrazem pomocniczym służącym do przechowywania wartości pośrednich filtracji. W tym przypadku obraz będzie przetransponowany w stosunku do obrazu oryginalnego, tzn. zamienione będą wiersze z kolumnami obrazu.

W liniach 67–70 za pomocą funkcji `createFromGLTexture2D()` tworzony jest obiekt `cl_tex` reprezentujący teksturę po stronie OpenCL. Obiekt ten jest jedynie uchwytem na teksturę OpenGL `gl_tex` i posłuży jako finalny bufor do zapisu przetworzonego obrazu w funkcji kernela po stronie OpenCL.

Czas teraz na definicję funkcji filtrującej po stronie *hosta*.

Listing 6.11. OpenCL – Funkcja filtrująca po stronie *hosta*.

```
74 void filter()
75 {
76   glFinish();
77   double time, time2;
78
79   time = timeStamp();
80   size_t GLOBAL_WS[2] = {image.width, 8};
81   size_t LOCAL_WS[2]  = {32,8};
82
83   clEnqueueAcquireGLObjects(cmdQueue, 1, &cl_tex, 0,0,0);
84
85   clSetKernelArg(vBoxFilter,0, sizeof(cl_mem), &cl_image);
86   clSetKernelArg(vBoxFilter,1, sizeof(cl_mem), &cl_image_temp);
87   clSetKernelArg(vBoxFilter,2, sizeof(int),    &radius);
88   clEnqueueNDRangeKernel(cmdQueue, vBoxFilter, 2, 0,
89                          GLOBAL_WS, LOCAL_WS, 0,0,0);
90
91   GLOBAL_WS[0] = image.height;
92   clSetKernelArg(hBoxFilter,0, sizeof(cl_mem), &cl_image_temp);
93   clSetKernelArg(hBoxFilter,1, sizeof(cl_mem), &cl_tex);
94   clSetKernelArg(hBoxFilter,2, sizeof(int),    &radius);
95   clEnqueueNDRangeKernel(cmdQueue, hBoxFilter, 2, 0,
96                          GLOBAL_WS, LOCAL_WS, 0,0,0);
97   clFinish(cmdQueue);
98
99   clEnqueueReleaseGLObjects(cmdQueue, 1, &cl_tex, 0,0,0);
100  clFinish(cmdQueue);
101
```

```
102   time2 = timeStamp();
103   cout<<"r="<<radius<<", time="<<time2-time<<"[ms]"<<endl;
104 }
```

W linii 76 następuje wywołanie funkcji `glFinish()` blokującej wątek *hosta* do czasu zakończenia wszystkich poleceń OpenGL. Takie zapewnienie jest konieczne w momencie gdy w kontekście OpenCL chcemy używać obiektów OpenGL. W tym celu należy jeszcze przejąć obsługę tych obiektów za pomocą funkcji:

```
cl_int clEnqueueAcquireGLObjects(cl_command_queue cmd_queue,
       cl_uint num_objects, const cl_mem *mem_objects,
       cl_uint newl, const cl_event *ewl, cl_event *event)
```

Po wywłaszczeniu danego obiektu przez OpenCL, próba odwołania się do niego z poziomu kontekstu OpenGL da rezultat niezdefiniowany. Taki obiekt, po wykonaniu na nim operacji w kontekście OpenCL, należy uwolnić za pomocą funkcji:

```
cl_int clEnqueueReleaseGLObjects(cl_command_queue cmd_queue,
       cl_uint num_objects, const cl_mem *mem_objects,
       cl_uint newl, const cl_event *ewl, cl_event *event)
```

W liniach 85–89 wywoływana jest pierwsza funkcja kernela, filtrująca obraz w pionie i zapisująca wynik do obrazu `cl_image_temp` z transponowanymi wierszami i kolumnami. Lokalny rozmiar *work-group* został ustalony na $[32, 8]$ natomiast globalna ilość *work-items* na szerokość obrazu w poziomie i 8 w pionie. Zatem, logicznie obraz został podzielony na pionowe pasy o szerokości 32 punktów. Każdy taki pas jest przetwarzany przez pojedynczy blok wątków.

W liniach 91–96 następuje wywołanie drugiego kernela filtrującego obraz w poziomie. Jednakże, i w tym przypadku zostanie zastosowany ten sam algorytm filtracji pionowymi pasami, ponieważ wynik poprzedniego etapu filtracji został przetransponowany, zatem rzeczywista filtracja pionowa będzie oznaczała logiczną filtrację poziomą. W takim razie po filtracji pionowej należy jeszcze raz przetransponować obraz aby uzyskać jego pierwotny kształt.

Zabieg z transponowaniem obrazu ma w zasadzie tylko wydajnościowe znaczenie i mógłby być pominięty, a druga funkcja filtrująca mogłaby przetwarzać obraz poziomymi pasami. Wyjaśnienie wzrostu wydajności jest dosyć proste. Mianowicie, odwołania do pamięci globalnej w danym *warpie* są realizowane równolegle, jeżeli kolejne wątki danego *warpa* pobierają wartości z kolejnych adresów pamięci globalnej. Takie założenie jest spełnione

gdy kolejne wątki pobierają punkty obrazu leżące w tym samym wierszu. W przypadku filtracji pasami poziomymi kolejne wątki *warpa* musiałyby pobierać punkty obrazu leżące w kolejnych liniach obrazu, co może powodować serializację wywołań tych wątków.

Funkcja obsługi zdarzeń GLUT `key_event()` rozpoznaje naciśnięcie przycisku strzałki do góry i do dołu. W pierwszym przypadku zostanie zwiększony o jeden promień maski filtru, a w drugim promień zostanie zmniejszony o 1.

Listing 6.12. OpenCL – Funkcja obsługi zdarzeń.

```
106 void key_event(int key, int x, int y)
107 {
108   switch(key)
109   {
110     case GLUT_KEY_UP:   radius = min(radius+1, 255); break;
111     case GLUT_KEY_DOWN: radius = max(radius-1, 0);   break;
112   }
113   filter();
114   render();
115 }
116
117 int main(int argc, char *argv[])
118 {
119   glutInit(&argc, argv);
120   glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
121   glutInitWindowSize(512, 512);
122   glutCreateWindow("Test Window");
123   glutDisplayFunc(render);
124   glutSpecialFunc(key_event);
125
126   initGL();
127   initCL();
128   filter();
129   glutMainLoop();
130   return 0;
131 }
```

W funkcji głównej, w stosunku do podstawowego szkieletu programu z listingu 6.2, doszły wywołania funkcji `initCL()` oraz `filter()` w liniach odpowiednio 127 i 128.

Do przeprowadzenia filtracji potrzeba jeszcze definicji dwóch funkcji kerneli `vbox()` oraz `hbox()` przeprowadzanących właściwą filtrację pionową i poziomą obrazu.

Listing 6.13. OpenCL – Funkcje rdzeni filtru uśredniającego.

```
const sampler_t samp = CLK_NORMALIZED_COORDS_FALSE |
                       CLK_ADDRESS_CLAMP_TO_EDGE |
                       CLK_FILTER_NEAREST;

__kernel void vbox(__read_only  image2d_t simg,
                   __write_only image2d_t dimg,
                   const int r)
{
  int h = get_image_height(simg)/get_local_size(1);
  int2 pos = (int2)(get_global_id(0), get_global_id(1)*h);
  int mask_dim = (2*r+1);
  uint4 pixel = (uint4)(0,0,0,0);

  for(int y=-r; y<=r; y++)
    pixel += read_imageui(simg, samp, (int2)(pos.x, pos.y+y));
  write_imageui(dimg, pos.yx, pixel/mask_dim);

  for (int y=pos.y+1; y<pos.y+h; y++)
  {
    pixel -= read_imageui(simg, samp, (int2)(pos.x, y-r-1));
    pixel += read_imageui(simg, samp, (int2)(pos.x, y+r));
    write_imageui(dimg, (int2)(y, pos.x), pixel/mask_dim);
  }
}

__kernel void hbox(__read_only image2d_t simg,
                   __write_only image2d_t dimg,
                   const int r)
{
  int h = get_image_height(simg)/get_local_size(1);
  int2 pos = (int2)(get_global_id(0), get_global_id(1)*h);
  float mask_rev = 1.0/(2*r+1)/255.0f;
  uint4 pixel = (uint4)(0,0,0,0);

  for(int y=-r; y<=r; y++)
    pixel += read_imageui(simg, samp, (int2)(pos.x, pos.y+y));
  write_imagef(dimg, pos.yx, convert_float4(pixel)*mask_rev);

  for (int y=pos.y+1; y<pos.y+h; y++)
  {
    pixel -= read_imageui(simg, samp, (int2)(pos.x, y-r-1));
    pixel += read_imageui(simg, samp, (int2)(pos.x, y+r));
    write_imagef(dimg, (int2)(y, pos.x),
                 convert_float4(pixel)*mask_rev);
  }
}
```

Przed użyciem funkcji pobierających i zapisujących punkty z/do tekstury należy zdefiniować obiekt próbnika czyli ***sampler***. Obiekt taki może być

utworzony w kodzie *hosta* za pomocą funkcji `clCreateSampler()` i następnie przekazany do funkcji kernela w postaci parametru lub utworzony statycznie w kodzie urządzenia. W powyższym przykładzie został użyty drugi sposób w linii 1:

```
const sampler_t samp = CLK_NORMALIZED_COORDS_FALSE |
                       CLK_ADDRESS_CLAMP_TO_EDGE |
                       CLK_FILTER_NEAREST;
```

Samper `samp` został skonfigurowany tak, aby używał niezmornmalizowanych współrzędnych `(CLK_NORMALIZED_COORDS_FALSE)`, przy odczycie wartości spoza tekstury współrzędne zostaną przybliżone wartościami brzegowymi `(CLK_ADDRESS_CLAMP_TO_EDGE)`, zostanie zastosowana interpolacja najbliższym sąsiadem `(CLK_FILTER_NEAREST)`.

W funkcji `vbox()` reazliowana jest filtracja pionowa obrazu. Sama realizacja splotu jednowymiarowego jest rozbita na dwie części. Najpierw, w liniach 14–16, obliczona jest nowa wartość punktu brzegowego przez zsumowanie wszystkich pionowych sąsiadów tego punktu. Odczyt wartości z tekstury jest realizowany za pomocą funkcji `read_imageui()` przyjmującej w parametrach obiekt obrazu `simg`, obiekt *samplera* `samp` oraz współrzędne pobieranego punktu. Następnie, w liniach 18–23, obliczenia prowadzone są dla punktów w kolejnych liniach przez odjęcie wartości punktu o współrzędnej $y = r - 1$ i dodanie punktu o współrzędnej $y = r$. Docelowa wartość jest zapisywana w obrazie wyjściowym `dimg` za pomocą funkcji `write_imageui()` przyjmującej w parametrze kolejno obraz docelowy `dimg`, współrzędne zapisywanego punktu oraz jego wartość. W linii 16 zastosowano notację wektorową zmiennej `pos.yx` zamieniając miejscami współrzędne $x \leftrightarrow y$.

Funkcja `hbox()` realizuje logicznie funkcję filtru poziomego. Jednakże, po przeanalizowaniu kodu, widać, że jest ona niemal identyczna z funkcją `vbox()`. Jedyna różnica jest w liniach zapisujących wartości obliczonych kolorów do obrazu docelowego. W tym przypadku została użyta funkcja `write_imagef()` (linie 37 i 43), która w pierwszym parametrze przyjmuje obraz typu `float`. Wartość zapisywanego punktu również musi być typu zmiennoprzecinkowego. Wynika to z faktu, że tekstura OpenGL zdefiniowana w linii 56 na listingu 6.3 jest interpretowana w potoku OpenGL jako struktura, która pojedynczy punkt opisuje czterema liczbami zmiennoprzecinkowymi pojedynczej precyzji.

Dodatek A

Funkcje pomocnicze

Listing A.1. Promiar czasu w systemie Linux.

```
#include <sys/time.h>

double timeStamp()
{
  struct timeval t;
  gettimeofday(&t, 0);
  return (1000.0*(t.tv_sec) + t.tv_usec/1000.0)/1000.0;
}
```

Listing A.2. Promiar czasu w systemie Windows.

```
#include <windows.h>

double timeStamp()
{
  SYSTEMTIME st;
  GetSystemTime(&st);
  return ((st.wMinute*60 st.wSecond*1000.0) +
           st.wMilisecond)/1000.0;
}
```

Listing A.3. OpenCL – Funkcja wczytująca program rdzenia.

```
#include <cstdio>
#include <cstdlib>
#include <cstring>

char* loadProgSource(const char* filename,
                     const char* preamble,
                     size_t* finalLength)
{
    FILE* file = fopen(filename, "rb");
    if(file == 0) return NULL;
    size_t preambleLength = strlen(preamble);

    fseek(file, 0, SEEK_END);
    size_t sourceLength = ftell(file);
    fseek(file, 0, SEEK_SET);

    char* sourceString = (char *)malloc(sourceLength +
                                        preambleLength + 1);
    memcpy(sourceString, preamble, preambleLength);
    if (fread((sourceString) + preambleLength, sourceLength,
              1, file) != 1)
    {
        fclose(file);
        free(sourceString);
        return 0;
```

```
    }
    fclose(file);

    *finalLength = sourceLength + preambleLength;
    sourceString[sourceLength + preambleLength] = '\0';
    return sourceString;
}
```

Listing A.4. OpenCL – Funkcja zwracająca kod błędu w postaci stringu.

```
const char* clErrorString(cl_int error)
{
    static const char* errorString[] = {
        "CL_SUCCESS",
        "CL_DEVICE_NOT_FOUND",
        "CL_DEVICE_NOT_AVAILABLE",
        "CL_COMPILER_NOT_AVAILABLE",
        "CL_MEM_OBJECT_ALLOCATION_FAILURE",
        "CL_OUT_OF_RESOURCES",
        "CL_OUT_OF_HOST_MEMORY",
        "CL_PROFILING_INFO_NOT_AVAILABLE",
        "CL_MEM_COPY_OVERLAP",
        "CL_IMAGE_FORMAT_MISMATCH",
        "CL_IMAGE_FORMAT_NOT_SUPPORTED",
        "CL_BUILD_PROGRAM_FAILURE",
        "CL_MAP_FAILURE",
        "", "", "", "", "", "", "", "", "", "",
        "", "", "", "", "", "", "", "", "",
        "CL_INVALID_VALUE",
        "CL_INVALID_DEVICE_TYPE",
        "CL_INVALID_PLATFORM",
        "CL_INVALID_DEVICE",
        "CL_INVALID_CONTEXT",
        "CL_INVALID_QUEUE_PROPERTIES",
        "CL_INVALID_COMMAND_QUEUE",
        "CL_INVALID_HOST_PTR",
        "CL_INVALID_MEM_OBJECT",
        "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR",
        "CL_INVALID_IMAGE_SIZE",
        "CL_INVALID_SAMPLER",
        "CL_INVALID_BINARY",
        "CL_INVALID_BUILD_OPTIONS",
        "CL_INVALID_PROGRAM",
        "CL_INVALID_PROGRAM_EXECUTABLE",
        "CL_INVALID_KERNEL_NAME",
        "CL_INVALID_KERNEL_DEFINITION",
        "CL_INVALID_KERNEL",
        "CL_INVALID_ARG_INDEX",
        "CL_INVALID_ARG_VALUE",
        "CL_INVALID_ARG_SIZE",
```

```
            "CL_INVALID_KERNEL_ARGS",
            "CL_INVALID_WORK_DIMENSION",
            "CL_INVALID_WORK_GROUP_SIZE",
            "CL_INVALID_WORK_ITEM_SIZE",
            "CL_INVALID_GLOBAL_OFFSET",
            "CL_INVALID_EVENT_WAIT_LIST",
            "CL_INVALID_EVENT",
            "CL_INVALID_OPERATION",
            "CL_INVALID_GL_OBJECT",
            "CL_INVALID_BUFFER_SIZE",
            "CL_INVALID_MIP_LEVEL",
            "CL_INVALID_GLOBAL_WORK_SIZE",
        };

        const int errorCount = sizeof(errorString) /
                               sizeof(errorString[0]);
        const int index = -error;
        return (index >= 0 && index < errorCount) ?
               errorString[index] : "Unspecified Error";
    }
```

Dodatek B

NVIDIA *Compute capabilities*

Tabela B.1: Tabela specyfikacji w zależności od *Compute Capabilities* dla urządzeń NVIDII

Specyfikacja	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Obsługa liczb zmiennoprzecinkowych podwójnej precyzji `double`	Nie			Tak	
Funkcje atomowe na liczbach całkowitych 32-bitowych w pamięci globalnej	Nie	Tak			
Funkcje atomowe na liczbach całkowitych 32-bitowych w pamięci współdzielonej	Nie		Tak		
Funkcje atomowe na liczbach całkowitych 64-bitowych w pamięci globalnej	Nie		Tak		
Funkcje atomowe na liczbach całkowitych 64-bitowych w pamięci współdzielonej	Nie				Tak
Funkcje atomowe na liczbach zmiennoprzecinkowych 32-bitowych w pamięci globalnej i współdzielonej	Nie				Tak
Maksymalna ilość wymiarów siatki bloków	2				3
Maksymalny rozmiar siatki bloków w każdym kierunku	65535				
Maksymalna ilość wymiarów bloku	3				
Maksymalny rozmiar bloku $[x : y]$	512				1024
Maksymalny rozmiar bloku z	64				
Maksymalna ilość wątków w bloku	512				1024
Rozmiar *warpa*	32				
Maksymalna ilość bloków rezydujących na MP	8				
Maksymalna ilość rezydujących *warpów* na MP	24		32		48
Maksymalna ilość rezydujących wątkół na MP	768		1024		1536
Ilość 32-bitowych rejestrów na MP	8K		16K		32K

Tabela B.1: Tabela specyfikacji w zależności od *Compute Capabilities* dla urządzeń NVIDII

Specyfikacja	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Maksymalna ilość pamięci współdzielonej `shared` na Multiprocesor	16KB				48KB
Ilość banków pamięci współdzielonej	16				32
Ilość pamięci lokalnej na wątek	16KB				512KB
Rozmiar pamięci stałej `constant`	64KB				
Rozmiar pamięci podręcznej dla pamięci stałej na MP	8KB				
Rozmiar pamięci podręcznej dla pamięci tekstur na MP	6KB–8KB zależnie od urządzenia				
Maksymalna ilość instrukcji na kernel	2 miliony				512 milionów

Bibliografia

[1] AMD Inc., *AMD Accelerated Parallel Processing OpenCL Programming Guide*, 2011.

[2] Kirk, D.B., Hwu, W.W., *Programming Massively Parallel Processors*, Elsevier Inc., 2010.

[3] Munshi, A., *OpenCL Programming Guide*, Addison-Wesley Professional, 2011.

[4] Munshi, A., *The OPENCL Specification*, Version 1.1, Khronos OpenCL Working Group, 2011.

[5] NVIDIA, *CUDA API Reference Manual*, Version 4.1, 2011.

[6] NVIDIA, *NVIDIA CUDA C Programming Guide*, Version 4.1, 2011.

[7] NVIDIA, *OpenCL Programming Guide for the CUDA Architecture*, Version 4.1, 2011.

[8] Sandlers, J., Kandrot, E., *CUDA by example. An Introduction to General-Purpose GPU Programming*, Addison-Wesley, 2010.

[9] Tsuchiyama, R., *OpenCL Programming Book*, Fixstars Corporation, 2010.