

Słowo wstępne

WSTĘP: WŁÓCZENIE KODU CUDA C JEDNOCZESNIE NA WIELU GPU	177
11.1. Streszczenie rozdziału	177
11.2. Pamięć hosta niewymagająca kopiowania	178
11.2.1. Obliczanie iloczynu skalarnego za pomocą pamięci niekopiowanej	178
11.2.2. Wydajność pamięci niekopiowanej	183
11.3. Użycie kilku procesorów GPU jednocześnie	184
11.4. Przenośna pamięć zablokowana	188
11.5. Podsumowanie	192
EPILOG	193
12.1. Streszczenie rozdziału	194
12.2. Narzędzia programistyczne	194
12.2.1. CUDA Toolkit	194
12.2.2. Biblioteka CUFFT	194
12.2.3. Biblioteka CUBLAS	195
12.2.4. Pakiet GPU Computing SDK	195
12.2.5. Biblioteka NVIDIA Performance Primitives	196
12.2.6. Usuwanie błędów z kodu CUDA C	196
12.2.7. CUDA Visual Profiler	198
12.3. Literatura	199
12.3.1. Książka Programming Massively Parallel Processors: A Hands-on Approach	199
12.3.2. CUDA U	199
12.3.3. Fora NVIDII	200
12.4. Zasoby kodu źródłowego	201
12.4.1. Biblioteka CUDA Parallel Primitives Library	201
12.4.2. CULATools	201
12.4.3. Biblioteki osłonowe	202
12.5. Podsumowanie	202
OPERACJE ATOMOWE DLA ZAAWANSOWANYCH	203
A.1. Iloczyn skalarny po raz kolejny	203
A.1.1. Blokady atomowe	205
A.1.2. Iloczyn skalarny: blokady atomowe	207
A.2. Implementacja tablicy skrótów	210
A.2.1. Tablice skrótów — wprowadzenie	210
A.2.2. Tablica skrótów dla CPU	212
A.2.3. Wielowątkowa tablica skrótów	216
A.2.4. Tablica skrótów dla GPU	217
A.2.5. Wydajność tablicy skrótów	223
A.3. Podsumowanie	224
Skorowidz	225

Śledząc ostatnie poczynania największych producentów układów graficznych, takich jak NVIDIA, można wyciągnąć wniosek, że przyszłość mikroprocesorów i dużych systemów HPC będzie należeć do układów hybrydowych. Budowa tych systemów będzie polegała na integracji dwóch elementów w różnych proporcjach:

- **Technologia wielordzeniowych procesorów CPU:** liczba rdzeni będzie cały czas rosła, aby spełnić wymóg upakowania coraz to większej liczby elementów na jednym chipie i uniknięcia przy tym ograniczeń związanych z poborem mocy, możliwością równoległego wykonywania instrukcji oraz pamięcią.
- **Specjalistyczny sprzęt i akceleratory o dużych możliwościach przetwarzania równoległego:** na przykład procesory GPU NVIDIA niedawno prześcignęły standardowe procesory CPU w kategorii wykonywania obliczeń zmiennoprzecinkowych. Co więcej, programowanie GPU nie jest już trudniejsze niż wielordzeniowych CPU.

Nie da się na razie przewidzieć, jakie będą proporcje tych dwóch rodzajów komponentów w projektach, które powstaną w przyszłości, ale z pewnością będą się zmieniać. Z dużą dozą pewności można jednak przypuszczać, że budowa przyszłych generacji komputerów, od laptopów po superkomputery, będzie oparta na architekturze hybrydowej. To właśnie tego rodzaju systemowi udało się przekroczyć granicę jednego petaflop, czyli wykonać 10^{15} operacji zmiennoprzecinkowych w ciągu sekundy.

Lecz mimo wszystko problemy i wyzwania, jakie stoją przed programistami tych nowych hybrydowych układów, wydają się niesamowicie skomplikowane. Krytyczne elementy infrastruktury programowej już mają duże problemy w dotrzymaniu kroku tempu zmian. W niektórych przypadkach wydajność programów nie pozwala na pełne wykorzystanie wszystkich rdzeni, ponieważ znaczna część czasu zamiast na obliczenia arytmetyczne jest przeznaczona na przenoszenie danych. Często też oprogramowanie zoptymalizowane pod kątem wydajności pojawia się długo po sprzecie, co powoduje, że już na samym starcie jest ono przestarzałe. Ponadto w niektórych przypadkach (dotyczy to np. niektórych najnowszych GPU) oprogramowanie w ogóle nie działa, gdyż środowiska programistyczne za bardzo się zmieniły.

W książce *CUDA w przykładach* poruszamy kwestie dotyczące samego sedna wyzwań stojących przed programistą. Książka ta zawiera opis jednego z najbardziej innowacyjnych i najlepszych rozwiązań problemu, jakim jest programowanie najnowszych akceleratorów o dużych możliwościach przetwarzania równoległego.

iąduje się w niej także wprowadzenie do programowania w języku CUDA C. Wykład oparty na praktycznych przykładach oraz informacjach na temat procesu konstrukcji i efektywnego wykorzystywania procesorów GPU firmy NVIDIA. Na początku zamieszczony jest opis podstawowych pojęć programowania równoległego, wzbogacony zarówno o proste przykłady, jak i bardziej skomplikowane techniki diagnostyki programów (zarówno warstwy logicznej, jak i pod względem wydajności). Dalej poruszone są zaawansowane techniki i tematy związane z budową i użytkowaniem wielu programów. Wszystkie prezentowane koncepcje są poparte odpowiednimi przykładami kodu źródłowego.

książka ta jest obowiązkową pozycją dla wszystkich programistów pracujących z systemami sterującymi akceleratorami. Zawiera bogaty opis technik programowania równoległego oraz wiązań wielu często spotykanych problemów. Najbardziej na jej lekturze skorzystają programiści aplikacji, twórcy bibliotek numerycznych oraz studenci i nauczyciele równoległego przetwarzania danych.

książka ta bardzo mi się podobała i wiele się z niej nauczyłem. Jestem pewien, że Ty również będziesz żałować poświęconego na jej lekturę czasu.

Jack Dongarra

*Profesor z tytułem University Distinguished Professor, Distinguished Research Staff Member
uniwersytetu University of Tennessee, Oak Ridge National Laboratory*

Przedmowa

Z książki tej nauczysz się wykorzystywać moc procesora GPU komputera do tworzenia wydajnych programów o szerokim spektrum zastosowań. Zgodnie z pierwotnymi założeniami jednostki GPU miały służyć tylko do wyświetlania grafiki na ekranie monitora komputerowego. I mimo że do dziś są do tego używane, zakres ich użycia znacznie się jednak poszerzył, gdy weszły także do takich dziedzin programowania jak aplikacje naukowe, inżynierijne czy ekonomiczne. Programy, które wykorzystują GPU do innych celów niż przetwarzanie grafiki, określamy mianem **programów ogólnych**. Najlepsze jest to, że chociaż do zrozumienia treści tej książki przydatna jest znajomość języków C i C++, to nie trzeba w ogóle znać się na grafice komputerowej. Poznając techniki programowania GPU, po prostu rozszerzysz swój zakres umiejętności o jedno dodatkowe, ale niezwykle potężne narzędzie.

Aby programować procesory GPU NVIDIA do ogólnych celów, trzeba znać technologię CUDA, ponieważ budowa tych jednostek jest oparta na tzw. **architekturze CUDA** (ang. *CUDA architecture*). Można ją traktować jako specjalny plan budowy procesorów graficznych przeznaczonych do przetwarzania zarówno grafiki, jak i wykonywania zadań ogólnych. Procesory GPU CUDA programuje się przy użyciu języka programowania o nazwie **CUDA C**. Jak się niebawem przekonasz, jest to w istocie język C wzbogacony o garść rozszerzeń umożliwiających programowanie równoległe.

Treść książki skierowana jest przede wszystkim do programistów, którzy znają języki C i C++ oraz mają wystarczające doświadczenie w programowaniu w języku C, żeby swobodnie czytać napisany w nim kod źródłowy. Jeśli zatem biegły posługujesz się językiem C, to dzięki tej książce rozszerzysz swoje umiejętności o podstawy języka CUDA C. Nie oznacza to w żadnym wypadku, że książka nadaje się tylko dla osób, które w swojej karierze zbudowały jakiś wielki projekt, napisały kompilator albo jądro systemu operacyjnego, czy też znają na pamięć treść standardu ANSI C. Zakładamy jednak, że znasz składnię tego języka i wiesz, do czego służą takie podstawowe funkcje jak `malloc()` czy `memcpy()`.

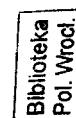
W kilku miejscach opisaliśmy techniki, które można określić jako ogólne zasady programowania równoległego. Nie należy jednak sądzić, że celem tej książki jest właśnie nauka tych technik. Ponadto mimo że znajduje się w niej opis wielu elementów API CUDA, nie należy wyciągać wniosku, iż książka ta jest jego dokumentacją ani też, że zawiera szczegółowy opis wszystkich narzędzi, których można używać do pisania programów w języku CUDA C. Dlatego zalecamy dodatkowe zaopatrzenie się w udostępnianą bezpłatnie przez firmę NVIDIA dokumentację,

łaszcza przewodnik dla programistów *NVIDIA CUDA Programming Guide* i przewodnik po lepszych praktykach *NVIDIA CUDA Best Practices Guide*. Zdobycie tych dokumentów nie jednak absolutnie konieczne, ponieważ w książce opisano wszystko, co trzeba wiedzieć.

rzebne oprogramowanie NVIDIA można pobrać pod adresem <http://developer.nvidia.com/doc/gpucomputing.html>. Dokładny opis narzędzi potrzebnych do rozpoczęcia pracy znajduje się rozdziale 2. Ponieważ w książce przyjęto filozofię nauki na praktycznych przykładach, znajduje się w niej wiele listingów kodu źródłowego. Można je pobrać pod adresem <ftp://ftp.helion.pl/yklady/cudawp.zip>.

przeciągając dłużej, czas wejść do świata programowania GPU NVIDIA i języka CUDA C!

Podziękowania



W powstanie każdej książki technicznej zaangażowana jest cała rzesza ludzi i tak też było w tym przypadku. Autorzy mają dług wdzięczności wobec wielu osób, którym pragną podziękować na tych kartach.

Dziękujemy Ianowi Buckowi, starszemu dyrektorowi programowania GPU w firmie NVIDIA, który nie tylko gorąco poparł pomysł napisania tej książki, lecz również wziął na siebie wiele związanych z tym obowiązków. Także nasz zawsze uśmiechnięty recenzent Tim Murray znaczco przyczynił się do tego, że książka ta da się czytać i trzyma jakieś standardy naukowej przyczyni. Dziękujemy również Darwinowi Tatowi, projektantowi znakomitej okładki i ilustracji, który znakomicie sobie poradził, mimo że pracował pod ogromną presją czasu. Ponadto jesteśmy zobowiązani Johnowi Parkowi za zajęcie się delikatną stroną prawną publikacji tego dzieła.

Bez pomocy pracowników wydawnictwa Addison-Wesley książka ta nadal pozostawałaby w sferze naszych marzeń. Prace nad nią udało się zakończyć bez większych problemów dzięki cierpliwości i profesjonalizmowi takich osób, jak Peter Gordon, Kim Boedigheimer oraz Julie Nahil. Ostateczny kształt produkt ten zawdzięcza pracy dwóch osób: Molly Sharp (produkcja) i Kim Wimpsett (adiustacja). Bez nich byłaby to nadal tylko sterta najeżonych błędami gryzmołów.

Gdyby nie pomóc pewnych osób, niektórych części tej książki w ogóle by nie było. Na wyróżnienie zasługuje Nadeem Mohammad, który skrupulatnie zbadał studia przypadku opisane w rozdziale 1. Natomiast bez pomocy Nathana Whiteheada nie byłoby prezentowanego w przykładach kodu źródłowego.

Nie możemy też zapomnieć o osobach, które przeczytały wstępne wersje tekstu i podzieliły się z nami swoimi spostrzeżeniami. Te osoby to Genevieve Breed i Kurt Wall. Duży wkład od strony technicznej w powstanie książki mają niektórzy programiści NVIDIA. Mark Hairgrove, przeglądając jej zawartość, odkrył całą masę wszelkiego rodzaju niedociągnięć: błędy techniczne, typograficzne i gramatyczne. Steve Hines, Nicholas Wilt i Stephen Jones udzielili konsultacji na tematy dotyczące wybranych fragmentów API CUDA i objaśnili nam pewne niuanse, które bez ich pomocy zostałyby pominięte. Podziękowania także dla Randimy Fernando za pomoc w uruchomieniu projektu oraz dla Michaela Schidlowsky'ego za wymienienie Jasona w swojej książce.

Co byłyby warte podziękowania bez wyrazów wdzięczności dla rodziców i rodzeństwa? Dziękujemy zatem naszym rodzinom, które towarzyszą nam od zawsze i dzięki którym to wszystko stało się możliwe. Specjalne podziękowania kierujemy do kochanych rodziców Edwarda i Kathleen Kandrot oraz Stephena i Helen Sanders. Dziękujemy też naszym braciom, Kennethowi Kan-

Rozdział 1

Dlaczego CUDA? Dlaczego teraz?

Jeszcze nie tak dawno programowanie równoległe uważano za zajęcie egzotyczne i najczęściej klasyfikowano je jako specjalizację szerszej dziedziny informatyki. Jednak w ciągu ostatnich kilku lat sposób postrzegania tej dziedziny radykalnie się zmienił. Obecnie prawie każdy aspirujący programista, jeśli chce być w swej pracy efektywny, **musi** znać techniki programowania równoległego. Trzymając w rękach tę książkę, nie jesteś jeszcze zapewne przekonany o tym, jak bardzo ważne jest programowanie równoległe, ani o roli, jaką będzie ono odgrywać w przyszłości. Dlatego w tym rozdziale zamieściliśmy najważniejsze informacje na temat rozwoju sprzętu, który wykonuje ciężką pracę, jaką my, programiści, mu zadajemy. Mamy nadzieję przekonać Cię, że rewolucja programistyczna **już** się dokonała i że nauka języka CUDA C pozwala pisać wydajne programy dla platform heterogenicznych zawierających zarówno procesory CPU, jak i GPU.

1.1. Streszczenie rozdziału

W tym rozdziale:

- Dowiesz się, jak ważne są techniki programowania równoległego.
- Poznasz historię procesorów GPU i technologii CUDA.
- Poznasz kilka programów, do których budowy użyto języka CUDA C.

1.2. Era przetwarzania równoległego

W ostatnich latach przemysł komputerowy ostro skręcił w kierunku technologii przetwarzania równoległego. W 2010 roku prawie wszystkie komputery zawierały już procesory wielordzeniowe. Od kiedy pojawiły się tanie netbooki z dwoma rdzeniami oraz potężne stacje robocze z procesorami o 8 i 16 rdzeniach, programowanie równoległe przestało należeć do egzotycznej sfery wielkich superkomputerów. Nawet w gadżetach elektronicznych, takich jak telefony i odtwarzacze muzyki, zaczęto stosować te techniki, aby umożliwić korzystanie z funkcji, o których kiedyś można było tylko marzyć.

czasem liczba rozmaitych platform przetwarzania równoległego będzie rosła, a przed programistami będą stawiane kolejne wyzwanie, tak aby zadowolić ciągle zmieniające się zapotrzebowanie klientów na coraz to nowsze i bardziej wymyślne produkty. Wiersz poleceń ododzi do lamusa. Nadeszła era wielowątkowych interfejsów graficznych. Telefony komórkowe mające tylko do dzwonienia też odeszły już do lamusa. Nadeszła era telefonów do jednocześnie odtwarzania muzyki, przeglądania internetu i korzystania z usług GPS.

2.1. PROCESORY CPU

ze 30 lat wydajność komputerów poprawiano poprzez zwiększanie szybkości działania zegara procesorów. Pierwsze komputery osobiste z początku lat 80. ubiegłego wieku miały procesory taktowane zegarem o częstotliwości 1 MHz. Obecnie większość procesorów będących w użytku mała z szybkością od 1 do 4 GHz, a więc przynajmniej tysiąc razy szybciej niż ich dawni zodkowie. Oczywiście zwiększanie szybkości zegara CPU to nie jedyny sposób na poprawienie wydajności procesora, ale metoda ta zawsze dawała dobre rezultaty.

jednak od pewnego czasu producenci są zmuszeni szukać innych rozwiązań, ponieważ zbliżyły się do granicy ulepszania technik produkcji układów scalonych. Nie da się już dalej zwiększać wydajności jednostek przetwarzających poprzez zwiększanie częstotliwości taktowania zegara, oraz większe zapotrzebowanie na moc, które pociąga za sobą zwiększoną ilość generowanego ciepła, a także szybkie zbliżanie się do fizycznej granicy miniaturyzacji tranzystorów nusły naukowców i producentów do szukania nowych możliwości.

świecie superkomputerów w podobny sposób zwiększa się wydajność już od 40 lat. Wydajność procesora używanego w superkomputerze urosła do astronomicznych wartości, podobnie jak się to stało z procesorami zwykłych komputerów. Jednak w przypadku superkomputerów producenci nie ograniczali się tylko do optymalizacji pojedynczych procesorów, lecz zwiększały również wydajność poprzez dodawanie kolejnych jednostek przetwarzających. Nieprzypadkowo najszybsze superkomputery zawierają dziesiątki, a nawet setki tysięcy współpracujących ze sobą rdzeni procesorów.

szukując możliwości zwiększenia mocy komputerów osobistych, zaczęto zauważać, że może pójść śladem konstruktorów superkomputerów i spróbować zamiast jednego rdzenia montować w nich po kilka rdzeni. W ten sposób można zapewnić ciągły przyrost wydajności komputerów bez konieczności przyspieszania taktowania zegara.

erwsze procesory dwurdzeniowe pojawiły się w sprzedaży w 2005 roku. Krok ten na producentów wymusiły bardzo zażarta konkurencja oraz brak innych możliwości. Później stopniowo zaczętoły produkować procesory z trzema, czterema, sześcioma i ośmioma rdzeniami. Ten trend, zwany **wolucją wielordzeniową**, stanowi wyznacznik dużego zwrotu w ewolucji komputerów osobistych.

zakup komputera z jednordzeniowym procesorem to nie lada wyzwanie. Nawet niskobudżetowe i energooszczędne jednostki przetwarzające mają przynajmniej dwa rdzenie. Wiodący

1.3. Era procesorów GPU

Jednostki przetwarzania grafiki (procesory GPU) stanowią odjście od tradycyjnego modelu potoku przetwarzania, jaki stosuje się w procesorach CPU. Sama dziedzina nauki zajmująca się budową GPU jest bardzo młoda w porównaniu z ogólną informatyką. A jednak pomysł wykorzystania procesorów graficznych do wykonywania obliczeń jest starszy, niż wielu się wydaje.

1.3.1. HISTORIA PROCESORÓW GPU

Wcześniej prześledziliśmy ewolucję jednostek centralnych zarówno pod względem szybkości zegara, jak i liczby rdzeni. Ale w tym samym czasie także procesory graficzne uległy radykalnym przemianom. Zapotrzebowanie na ten nowy typ procesora pojawiło się pod koniec lat 80. ubiegłego wieku, kiedy to do masowego użytku weszły systemy operacyjne z graficznym interfejsem użytkownika, takie jak Microsoft Windows. Na początku lat 90. w sprzedaży pojawiły się dwuwymiarowe akceleratory graficzne. Były to karty rozszerzeń wspomagające operacje na mapach bitowych i pomagające w wyświetlaniu graficznych elementów systemów operacyjnych oraz usprawniające ich obsługę.

Mniej więcej w tym samym czasie w branży komputerowej działała firma o nazwie Silicon Graphics, która w latach 80. popularyzowała grafikę trójwymiarową, kierując swoją ofertę do różnych odbiorców, m.in. programów tworzonych na potrzeby rządu i wojska, aplikacji naukowych, wizualizacji technicznych. Ponadto firma produkowała narzędzia umożliwiające uzyskanie niesamowitych filmowych efektów specjalnych. W 1992 roku dzięki opublikowaniu biblioteki OpenGL firma udostępniła swój interfejs programistyczny producentom sprzętu. Według założeń firmy OpenGL miała być standardową i niezależną od platformy technologią do tworzenia programów korzystających z grafiki trójwymiarowej. Podobnie jak w przypadku przetwarzania równoległego i procesorów CPU, trafienie tej technologii do masowego odbiorcy było tylko kwestią czasu.

W połowie lat 90. gwałtownie wzrosło zapotrzebowanie na aplikacje 3D. Doprowadziło to do dwóch doniosłych wydarzeń. Po pierwsze powstały wciągające gry, takie jak Doom, Duke Nukem 3D czy Quake, w których rozgrywka odbywa się z perspektywy pierwszej osoby. Pojawienie się tych gier było silnym bodźcem do opracowania jeszcze bardziej realistycznych środowisk trójwymiarowych na potrzeby nowych gier. Podczas gdy wcześniej czy później i tak w końcu prawie wszystkie gry tworzono by przy użyciu tych technik, popularność tzw. strzelanek znacznie przyspieszyła proces adaptacji technologii 3D w komputerach osobistych. W tym samym czasie takie firmy jak NVIDIA, ATI Technologies i 3dfx Interactive zaczęły produkować akceleratory grafiki po przystępnej cenie, co szybko przyciągnęło ogólną uwagę. Wydarzenia te umocniły rozwój technologii 3D, która nadal burzliwie się rozwija.

Kolejnym ważnym wydarzeniem w rozwoju sprzętu graficznego było pojawienie się układu

we możliwości pozwalające uzyskać jeszcze ciekawsze efekty wizualne. Ponieważ przekształcenia i oświetlenie były już wówczas składnikami potoku przetwarzania biblioteki OpenGL, układ GeForce 256 stał się naturalnym wyznacznikiem kierunku rozwoju zmierzającego ku zwracaniu na procesor graficzny coraz to większej części potoku graficznego.

W względem przetwarzania równoleglego przełomowym wydarzeniem w branży procesorów GPU było pojawienie się w 2001 roku serii GeForce 3. Były to pierwsze na świecie układy z implementacją nowego wówczas standardu DirectX 8.0, który wymagał od sprzętu możliwości programowania zarówno shaderów pikseli, jak i wierzchołków. Po raz pierwszy w historii programista miał wpływ na to, jakie dokładne obliczenia będą wykonywane przez GPU.

3.2. POCZĄTKI PROGRAMOWANIA GPU

Procesory GPU umożliwiające programowanie potoków przetwarzania przyciągnęły wielu badaczy, którzy chcieli je wykorzystać do wielu innych celów, a nie tylko do generowania grafiki. Użycie bibliotek OpenGL albo DirectX. Jednak początki nie były łatwe, ponieważ jedynym sposobem interakcji z procesorem GPU było wówczas użycie bibliotek graficznych. Zatem bez względu na to, jakiego rodzaju obliczenia wykonywano, trzeba było postępować zgodnie z zasadami programowania grafiki przy użyciu wymienionych API. Programiści radzili sobie z tym problemem, definiując problemy do rozwiązywania w taki sposób, żeby dla procesora graficznego gądały tak jak zwykłe renderowanie grafiki.

Początku tego wieku procesory GPU obliczały kolor każdego piksela na ekranie za pomocą programowalnych jednostek arytmetycznych, zwanych **shaderami pikseli**. Ogólnie rzecz biorąc, shader pikseli oblicza ostateczną wartość koloru na podstawie położenia (x, y) piksela na ekranie i kilku dodatkowych informacji. Tymi dodatkowymi informacjami mogły być podane koordynaty, współrzędne teksturowe i inne atrybuty przekazane podczas działania shadera. Ponieważ programista miał teraz pełną władzę nad działaniami arytmetycznymi, które GPU wykonywał na kolorach i teksturach, prędko zauważono, że w miejsce tych „kolorów” można w istocie prowadzić **dowolne dane**.

Sam razem gdyby na wejściu podano rzeczywiste dane liczbowe oznaczające coś innego niż kolory, to można by było zmusić shadery pikseli do wykonywania na nich dowolnych obliczeń. Wniki można by było przekazywać do GPU jako wartości kolorów, przy czym w istocie byłyby to kolory, lecz wyniki zaplanowanych przez programistę obliczeń. Następnie można by było pozbierać z GPU, który nie miałby pojęcia, co się dzieje. Mówiąc krótko, sztuczka polegałaby na „zukaniu” procesora GPU, że wykonuje zadania związane z renderowaniem grafiki, a w rzeczywistości zajmowałaby się całkiem innymi obliczeniami. Byłoby to bardzo sprytne, ale niestety nie gmatwane podejście.

W względzie na duże możliwości arytmetyczne procesorów GPU wstępne wyniki tych eksperymentów pozwalały przewidywać świetlną przyszłość dla tego typu technik. Jednak ograniczenie

Dostęp do zasobów był bardzo ograniczony, ponieważ dane do programów można było pobierać tylko z kilku kolorów wejściowych i jednostek teksturowych. Ponieważ poważnie ograniczone były też możliwości wyboru sposobu i miejsca zapisu wyników w pamięci, użycie algorytmów zapisujących dane w losowych miejscach było niemożliwe. Co więcej z niemożliwością graniczyła próba przewidzenia, jak konkretny GPU potraktuje dane zmiennoprzecinkowe, jeśli w ogóle je obsługiwał. To uniemożliwiało wykonywanie w GPU większości obliczeń naukowych. Ponadto jeśli program zwrócił niepoprawny wynik, nie dał się zamknąć albo po prostu spowodował zawieszenie komputera, nie było żadnego dobrego sposobu na sprawdzenie kodu, który był wykonywany w GPU.

Jakby tego było mało, jeśli już ktoś mimo wszystko zdecydował się na wykonywanie ogólnych obliczeń przy użyciu GPU, to musiał nauczyć się używania biblioteki OpenGL lub DirectX, ponieważ tylko za ich pośrednictwem można było się z nim komunikować. Zmuszało to programistę nie tylko do przechowywania wyników obliczeń w teksturach graficznych i wykonywania obliczeń za pomocą funkcji OpenGL lub DirectX, lecz również do pisania algorytmów przy użyciu specjalnych języków programowania do obróbki grafiki, zwanych **językami do cieniowania (ang. shading language)**. Wymaganie od naukowców, aby walczyli z poważnymi ograniczeniami zasobów i możliwości programistycznych oraz dodatkowo uczyli się języków programowania do cieniowania i przetwarzania grafiki to było już zbyt wiele, aby zyskać szeroką akceptację.

1.4. CUDA

Jednak czasy świetności procesorów GPU miały nadejść dopiero pięć lat po pojawieniu się układów GeForce 3. W listopadzie 2006 roku NVIDIA zaprezentowała pierwszy na świecie GPU z obsługą DirectX 10 — układ o nazwie GeForce 8800 GTX. Był to zarazem pierwszy procesor zbudowany w architekturze CUDA. Architektura ta została specjalnie tak zaprojektowana, aby nie było wielu ograniczeń, które by uniemożliwiały wykorzystanie wcześniejszych procesorów graficznych do ogólnych zastosowań.

1.4.1. CO TO JEST ARCHITEKTURA CUDA

W odróżnieniu od poprzednich generacji procesorów, w których jednostki wykonujące obliczenia były podzielone na shadery wierzchołków i pikseli, w architekturze CUDA zastosowano jeden połączony potok przetwarzania. Dzięki temu program wykonujący ogólne obliczenia miał w końcu do dyspozycji wszystkie jednostki arytmetyczno-logiczne (ALU) procesora. Ponieważ w NVIDIA chciano, aby te nowe procesory mogły być wykorzystywane do celów ogólnych, jednostki ALU zbudowano zgodnie z normą IEEE dotyczącą arytmetyki liczb zmiennoprzecinkowych pojedynczej precyzji oraz wbudowano im zestaw instrukcji, które zamiast do przetwarzania grafiki są przeznaczone do wykonywania obliczeń ogólnych. Ponadto jednostkom wykonawczym GPU zezwolono na swobodny dostęp do pamięci w celu odczytu i zapisu,

modyfikacje architektury CUDA zostały dodane po to, aby stworzyć procesor GPU, który nie tylko dobrze radzi sobie ze zwykłymi zadaniami graficznymi, ale również doskonale wykonywało zwykłe obliczenia.

4.2. UŻYWANIE ARCHITEKTURY CUDA

Wysiłki NVIDIA, aby wyprodukować procesor nadający się zarówno do przetwarzania grafiki, jak i wykonywania zwykłych obliczeń, nie mogły jednak zakończyć się na zaprojektowaniu jedynie układu na bazie architektury CUDA. Niezależnie od tego, ile rozmaitych nowych funkcji dodano do układów, nadal jedynym sposobem na uzyskanie do nich dostępu byłoby użycie OpenGL albo DirectX. Nie dość że programiści nadal musieliby przedstawiać procesorowi wszystkie obliczenia postacią problemów graficznych, to na dodatek należałoby jeszcze używać do tego celu specjalistycznych języków do przetwarzania grafiki, takich jak GLSL z OpenGL czy HLSL Microsoftu.

Wciąż dogodni jak największej liczbie programistów, zdecydowano się na rozszerzenie języka CUDA. Dodano do niego pewną liczbę słów kluczowych umożliwiających korzystanie ze specjalnych funkcji architektury CUDA. I tak po kilku miesiącach od debiutu układu GeForce 8800 GTX NVIDIA zaprezentowała kompilator dla tego nowego języka, który nazwano CUDA C. Ten sposób językowy stał się pierwszym językiem programowania opracowanym w firmie zajmującej się produkcją GPU oraz przeznaczonym do wykonywania ogólnych obliczeń.

Poza specjalnego języka programowania NVIDIA dostarcza także specjalny sterownik szeregowy, który pozwala wykorzystać całą potężną moc obliczeniową architektury CUDA. Nie trzeba już znać bibliotek OpenGL i DirectX ani też przedstawiać problemów obliczeniowych do zadań graficznych.

5. Zastosowania technologii CUDA

Mimo że debiut architektury CUDA nastąpił nie tak dawno, bo na początku 2007 roku, to kościoły z użycia języka CUDA C odniosły już wiele podmiotów. Wśród największych sukcesów należy wymienić zwiększenie wydajności programów nawet o kilka rzędów wielkości w porównaniu z wcześniejszymi najwyższą klasą implementacjami. Ponadto rozwiązania budowane na bazie nowych procesorów graficznych NVIDIA cieszą się lepszym stosunkiem wydajności do ceny z wydajnością do ilości pobieranej mocy w porównaniu z tradycyjnymi rozwiązaniami. Poniżej znajdują się kilka przykładów udanego zastosowania języka CUDA C i architektury CUDA.

5.1. OBRAZOWANIE MEDYCZNE

W ciągu ostatnich 20 lat znaczająco wzrosła liczba kobiet cierpiących na raka piersi. Na szczęście dzięki wytrwałym wysiłkom wielu osób udało się także zwiększyć świadomość społeczeństwa

Aby uniknąć wyniszczających skutków ubocznych chemioterapii i naświetlania oraz konieczności interwencji chirurgicznej, a nawet zgonu pacjentki, jeśli leczenie nie poskutkuje, każdy przypadek choroby musi być zdiagnozowany we wczesnym stadium. Dlatego naukowcy ciągle poszukują szybkich, dokładnych i jak najmniej inwazyjnych metod wykrywania wczesnych objawów tego rodzaju raka.

Niestety mammografia, jedna z aktualnie najlepszych technik wczesnego wykrywania raka piersi, ma kilka poważnych wad. Aby wykryć potencjalnie niebezpieczne zmiany, trzeba wykonać przynajmniej dwa prześwietlenia rentgenem, a następnie film musi zostać przekazany do wywołania i oceny przez wykwalifikowanego specjalistę. Każde badanie polegające na prześwietleniu klatki piersiowej pacjentki przy użyciu promienia rentgena jest jednak szkodliwe. Jeśli po bardzo dokładnym przeanalizowaniu wyników lekarz nie jest pewien diagnozy, zleca bardziej szczegółowe badania — w razie potrzeby także biopsję. Czasami się okazuje, że to fałszywy alarm i dodatkowe badania były tylko niepotrzebną stratą czasu i pieniędzy, nie mówiąc już o tym, co przeżyła sama pacjentka.

Bezpieczniejszą metodą jest badanie ultradźwiękowe, które lekarze często stosują w połączeniu z mammografią w celu diagnostowania i leczenia raka piersi. Jednak i ta metoda ma pewne wady, których rozwiązanie podjęła się nowo utworzona firma TechniScan Medical Systems. Jej pracownicy opracowali bardzo obiecującą trójwymiarową metodę obrazowania ultradźwiękowego, ale rozwiązania tego nie można było zastosować z bardzo prostego powodu: ograniczonej mocy obliczeniowej komputerów. Mówiąc krótko, przekształcenie zebranych metodą ultradźwiękową danych w trójwymiarowy obraz w rozsądny czasie wymagało nieosiągalnej wówczas mocy obliczeniowej i było po prostu zbyt drogie.

Dopiero pojawienie się procesorów GPU NVIDIA opartych na architekturze CUDA i języka CUDA C pozwoliło założycielom firmy TechniScan zamienić marzenia w rzeczywistość. Opracowany przez nich system obrazowania ultradźwiękowego o nazwie Svara wykonuje obraz klatki piersiowej pacjentki za pomocą fal ultradźwiękowych. System ten w ciągu piętnastominutowego badania generuje 35 GB danych, które są analizowane przez dwa procesory NVIDIA o nazwie Tesla C1060. Dzięki tym jednostkom lekarz już po 20 minutach ma do dyspozycji bardzo szczegółowy trójwymiarowy obraz klatki piersiowej swojej pacjentki.

1.5.2. SYMULACJA DYNAMIKI PŁYNÓW

Przez wiele lat sztukę projektowania wydajnych wirników i ich łopat uważano za czarną magię. Proste metody badawcze zawodziły, ponieważ ruch cząstek powietrza i płynów wokół tych urządzeń jest rzadzony niezwykle skomplikowanymi prawami. Niestety realistyczne modelowanie komputerowe również nie było możliwe ze względu na ograniczoną moc obliczeniową komputerów. Tylko największe na świecie superkomputery były w stanie podjąć zadaniem obliczeniowym, jakie stawiały przed nimi wyrafinowane modele numeryczne potrzebne do projektowania i weryfikacji projektów. Ponieważ jednak niewielu badaczy ma dostęp do takich maszyn, dziedzina ta była pograżona w stagnacji.

dnym z najprzéniej działających ośrodków badań nad zaawansowanymi technologiami zetwarzania równoległego, których prekursorem jest Charles Babbage, jest Uniwersytet Cambridge. Należący do grupy „wielordzeniowców” dr Graham Pullan i dr Tobias Brandvik awidłowo przewidzieli, że architektura CUDA pozwoli znacznie przyspieszyć komputerowe dania dynamiki płynów. Ich wstępne badania wskazywały, że już zwykłe osobiste stacje roczne wyposażone w GPU NVIDIA mogą mieć wystarczającą moc obliczeniową. Gdy później wykryto niewielkiego klastra procesorów GPU, prześcignął on w osiągnięciach ich znacznie drożny superkomputer. Stało się jasne, że procesory NVIDIA doskonale nadawały się do rozwiązywania tego rodzaju problemów, którymi się zajmowali.

a badaczy z Cambridge ogromny wzrost mocy obliczeniowej oferowany przez język CUDA to nie tylko zwykłe zwiększenie szybkości ich superkomputerów. Dostępność dużych ilości GPU pozwoliła na szybkie przeprowadzenie wielu eksperymentów. Dzięki otrzymywaniu wyników w ciągu kilku sekund naukowcy mogą łatwiej dokonywać przełomowych odkryć. astryle procesorów GPU całkowicie zmieniły sposób podejścia naukowców do badań naukowych. awie interaktywne symulacje pozwoliły wyzwolić nowe pokłady innowacyjności i kreatywności, które do tej pory były uśpione.

5.3. OCHRONA ŚRODOWISKA

aturalną konsekwencją postępującej industrializacji jest powstanie i ciągły wzrost zapotrzebowania na produkty ekologiczne. Rosnący niepokój związany ze zmianami klimatu, wzrostem cen paliwa oraz zwiększającą się ilością zanieczyszczeń w powietrzu i wodzie spowodowały, że często dostrzegać skutki uboczne burzliwego postępu cywilizacyjnego. Od dawna wiadomo, takie produkty jak detergenty i środki czystości są najbardziej potrzebnymi, ale jednocześnie niebezpieczniejszymi dla środowiska produktami codziennego użytku. Dlatego naukowcy czelni szukać metod redukcji szkodliwego wpływu detergentów na środowisko bez zmniejszania ich skuteczności. Lecz uzyskanie czegoś w zamian za nic nie jest takie łatwe.

ajważniejsze składniki detergentów to tzw. **środki powierzchniowo czynne**. To właśnie nich opiera się skuteczność i konsystencja detergentów i szamponów. Niestety uważa się je również za najbardziej niebezpieczne dla środowiska. Cząsteczki środków powierzchniowo czynnych łączą się z brudem, a następnie mieszają z wodą, którą się następnie usuwa wraz zanieczyszczeniami. Tradycyjne metody oceny skuteczności środka powierzchniowo czynnego wymagają przeprowadzenia wielu prób laboratoryjnych na różnych kombinacjach materiałów i różnych rodzajach brudu. Nietrudno się domyślić, że jest to proces długotrwały i kosztowny.

irma Procter & Gamble we współpracy z Temple University pracuje nad zastosowaniem symulacji do przewidywania sposobu zachowania cząsteczek środków powierzchniowo czynnych zetknięciu z brudem, wodą i innymi substancjami. Zastosowanie komputerów pozwoliło nie tylko przyspieszyć tradycyjne metody, lecz również rozszerzyć zestaw możliwych do wykonania testów o dodatkowe warunki otoczenia, co wcześniej było praktycznie niemożliwe. Naukowcy Temple University użyli oprogramowania powstałego w należącym do Departamentu Energii

Stanów Zjednoczonych Ames Laboratory do symulacji o nazwie Highly Optimized Object-oriented Many-particle Dynamics (HOOMD). Dzieląc zadanie symulacji na dwa procesory Tesla NVIDIA, osiągnęli wydajność obliczeniową porównywalną z wydajnością 128 rdzeni CPU komputera Cray XT3 i 1024 procesorów CPU komputera BlueGene/L firmy IBM. Po zwiększeniu liczby GPU NVIDIA naukowcy wykonują teraz symulacje zachowania cząsteczek środków powierzchniowo czynnych z prędkością 16 razy większą od wymienionych maszyn. Dzięki redukcji czasu potrzebnego na wykonanie tak skomplikowanych symulacji z tygodni do godzin należy w niedalekiej przyszłości spodziewać się lawinowego pojawiania się produktów o znacznie większej efektywności, a zarazem mniejszej szkodliwości dla środowiska niż ich starsze odpowiedniki.

1.6. Podsumowanie

Branża komputerowa znajduje się dziś na skraju rewolucji, którą wywoła masowe rozposzczelnienie technologii przetwarzania równoległego, a CUDA C NVIDIA jest jak na razie najlepszym językiem przeznaczonym do tego rodzaju programowania. Dzięki tej książce nauczysz się pisać programy za pomocą tego języka. Poznasz rozszerzenia dodane do języka C oraz interfejsy programistyczne utworzone przez NVIDIA w celu ułatwienia pracy programistom. Nie musisz znać OpenGL i DirectX ani znać się na grafice komputerowej.

Ponieważ w książce tej nie zostały opisane podstawy języka C, nie polecamy jej osobom początkującym w zawodzie programisty. Pomocna podczas lektury może być ogólna wiedza na temat programowania równoległego, ale **nie jest warunkiem** praktyczne doświadczenie w pisaniu tego typu programów. Wszystkie potrzebne pojęcia związane z programowaniem równoległym są objaśnione w tekście. Jeśli znasz ogólnie zasady programowania równoległego, to od czasu do czasu możesz znaleźć fakty dotyczące programowania GPU, które będą niezgodne z Twoimi wyobrażeniami. Mówiąc krótko, jedynym warunkiem, aby w pełni zrozumieć treść tej książki, jest znajomość języka C lub C++ na średnim poziomie.

W następnym rozdziale znajduje się opis tego, jak należy przygotować swój komputer do programowania GPU, zarówno od strony sprzętowej, jak i programowej. Od kolejnego rozdziału zaczyna się już praktyczna nauka. Osoby, które wiedzą, jak używać języka CUDA C, albo są pewne, że ich komputer jest odpowiednio skonfigurowany, mogą pominąć rozdział 2.

Rozdział 2

Konfiguracja komputera

Pierwszy rozdział miał na celu rozpalić w Czytelniku chęć do nauki języka CUDA C, a ponieważ zgodnie z naszym założeniem nauka ta ma się odbywać na bazie praktycznych przykładów, potrzebne jest odpowiednie środowisko programistyczne. Oczywiście można poprzedzać tylko na przeczytaniu tekstu, ale jeśli napiszesz i uruchomisz kilka programów własnoręcznie, to dłużej utrzymasz swój zapał i będziesz mieć lepszą zabawę. Dlatego w tym rozdziale znajduje się opis elementów programowych i sprzętowych, które będą Ci potrzebne, aby zacząć pracę. Najlepsze jest to, że wszystkie potrzebne programy są dostępne bezpłatnie, dzięki czemu zaoszczędzoną gotówkę możesz wydać na inne przyjemności.

2.1. Streszczenie rozdziału

W tym rozdziale:

- Dowiesz się, skąd pobrać potrzebne oprogramowanie.
- Nauczysz się konfigurować środowisko pracy, w którym będzie można kompilować programy w języku CUDA C.

2.2. Środowisko programistyczne

Aby rozpocząć naukę języka CUDA C, należy skonfigurować specjalne środowisko programistyczne. Do pisania programów w tym języku potrzebne są następujące elementy:

- Procesor graficzny z obsługą technologii CUDA
- Sterownik urządzeń NVIDIA
- Zestaw narzędzi programistycznych NVIDIA
- Standardowy kompilator języka C

Aby zaoszczędzić Ci kłopotów, poniżej szczegółowo opisaliśmy, skąd wziąć każdy z wymienionych składników środowiska.

2.2.1. PROCESOR GRAFICZNY Z OBSŁUGĄ TECHNOLOGII CUDA

Znalezienie procesora graficznego zbudowanego na bazie architektury CUDA nie jest trudne, ponieważ na architekturze tej oparte są wszystkie procesory NVIDIA, poczynając od wydanego w 2006 roku układu GeForce 8800 GTX. W tabeli 2.1 znajduje się lista procesorów GPU zbudowanych na bazie architektury CUDA, ale ponieważ NVIDIA cały czas wydaje coraz to nowsze układy, lista ta na pewno nie jest pełna. Niemniej jednak wszystkie wymienione tu procesory obsługują CUDA.

Tabela 2.1. Procesory GPU oparte na architekturze CUDA

GeForce GTX 480	GeForce 9800 GTX	GeForce 8300 mGPU
GeForce GTX 470	GeForce 9800 GT	GeForce 8200 mGPU
GeForce GTX 295	GeForce 9600 GSO	GeForce 8100 mGPU
GeForce GTX 285	GeForce 9600 GT	Tesla S2090
GeForce GTX 285 for Mac	GeForce 9500 GT	Tesla M2090
GeForce GTX 280	GeForce 9400GT	Tesla S2070
GeForce GTX 275	GeForce 8800 Ultra	Tesla M2070
GeForce GTX 260	GeForce 8800 GTX	Tesla C2070
GeForce GTS 250	GeForce 8800 GTS	Tesla S2050
GeForce GT 220	GeForce 8800 GT	Tesla M2050
GeForce G210	GeForce 8800 GS	Tesla C2050
GeForce GTS 150	GeForce 8600 GTS	Tesla S1070
GeForce GT 130	GeForce 8600 GT	Tesla C1060
GeForce GT 120	GeForce 8500 GT	Tesla S870
GeForce G100	GeForce 8400 GS	Tesla C870
GeForce 9800 GX2	GeForce 9400 mGPU	Tesla D870
GeForce 9800 GTX+	GeForce 9300 mGPU	
Procesory do urządzeń przenośnych z serii Quadro		
Quadro FX 3700M	Quadro NVS 130M	Quadro FX 470
Quadro FX 3600M	Quadro FX 5800	Quadro FX 380
Quadro FX 2700M	Quadro FX 5600	Quadro FX 370
Quadro FX 1700M	Quadro FX 4800	Quadro FX 370 Low Profile
Quadro FX 1600M	Quadro FX 4800 for Mac	Quadro CX
Quadro FX 770M	Quadro FX 4700 X2	Quadro NVS 450

Tabela 2.1. Procesory GPU oparte na architekturze CUDA — ciąg dalszy

Procesory do urządzeń przenośnych z serii Quadro		
Quadro FX 570M	Quadro FX 4600	Quadro NVS 420
Quadro FX 370M	Quadro FX 3800	Quadro NVS 295
Quadro FX 360M	Quadro FX 3700	Quadro NVS 290
Quadro NVS 320M	Quadro FX 1800	Quadro Plex 2100 D4
Quadro NVS 160M	Quadro FX 1700	Quadro Plex 2200 D2
Quadro NVS 150M	Quadro FX 580	Quadro Plex 2100 S4
Quadro NVS 140M	Quadro FX 570	Quadro Plex 1000 Model IV
Quadro NVS 135M		
Procesory do urządzeń przenośnych z serii GeForce		
GeForce GTX 280M	GeForce G102M	GeForce 9500M G
GeForce GTX 260M	GeForce 9800M GTX	GeForce 9300M GS
GeForce GTS 260M	GeForce 9800M GT	GeForce 9300M G
GeForce GTS 250M	GeForce 9800M GTS	GeForce 9200M GS
GeForce GTS 160M	GeForce 9800M GS	GeForce 9100M G
GeForce GTS 150M	GeForce 9700M GTS	GeForce 8800M GTS
GeForce GT 240M	GeForce 9700M GT	GeForce 8700M GT
GeForce GT 230M	GeForce 9650M GS	GeForce 8600M GT
GeForce GT 130M	GeForce 9600M GT	GeForce 8600M GS
GeForce G210M	GeForce 9600M GS	GeForce 8400M GT
GeForce G110M	GeForce 9500M GS	GeForce 8400M GS
GeForce G105M		

Pełna lista znajduje się na stronie www.nvidia.com/cuda, ale i bez tego można bezpiecznie przyjąć, że wszystkie niezbyt stare GPU (które pojawiły się po 2006 roku i które mają nie mniej niż 256 MB pamięci graficznej) powinny obsługiwać technologię CUDA i język CUDA C.

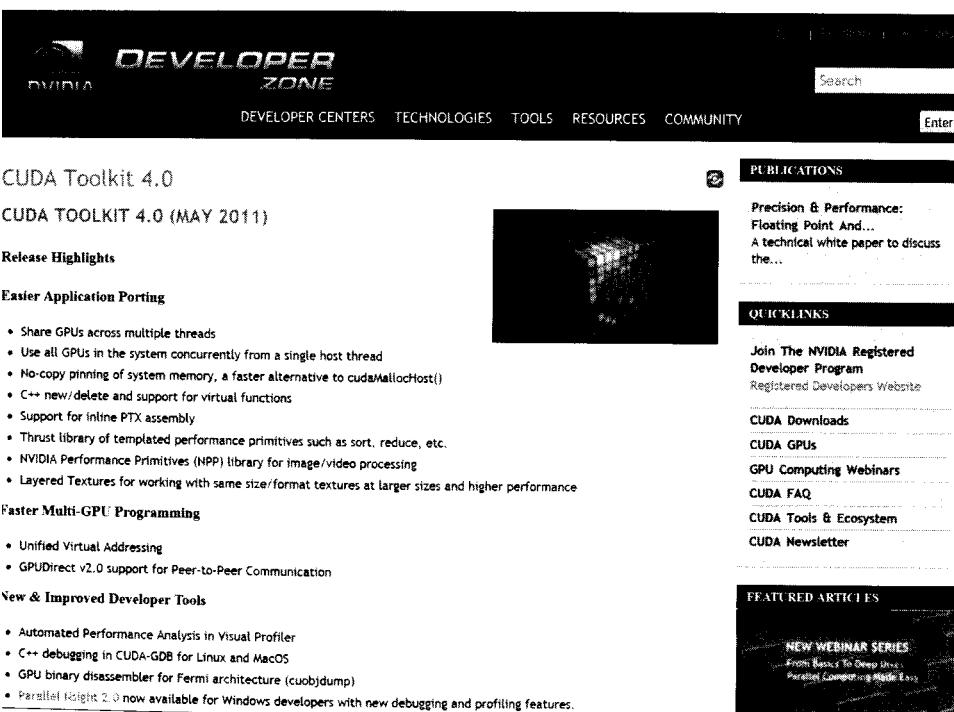
2.2.2. STEROWNIK URZĄDZEŃ NVIDIA

Firma NVIDIA udostępnia oprogramowanie pośredniczące w komunikacji między programami a sprzętem opartym na architekturze CUDA. Jeśli masz w swoim komputerze poprawnie zainstalowany układ GPU, to najprawdopodobniej masz też to oprogramowanie. Sprawdzenie, czy ma się zainstalowane najnowsze sterowniki, nigdy nie zaszkodzi, a więc zalecamy wejście na stronę www.nvidia.com/cuda i kliknięcie odnośnika *Download Drivers* (pobierz sterowniki). Następnie wybierz opcje odpowiednie dla swojej karty graficznej i systemu operacyjnego, którego masz zamiar używać. Zainstaluj oprogramowanie, wykonując wyświetlane instrukcje.

2.3. NARZĘDZIA PROGRAMISTYCZNE CUDA

Jeśli masz procesor GPU NVIDIA oparty na architekturze CUDA i sterownik urządzeń NVIDIA, możesz już uruchamiać na swoim komputerze programy napisane w języku CUDA C. To znaczy, że możesz pobrać na dysk dowolny program korzystający z technologii CUDA i będzie on Cię działał. Podejrzewamy jednak, że skoro trzymasz w ręku tę książkę, to zapewne chodzi o coś więcej niż tylko uruchamianie gotowych programów. Jeśli chcesz pisać programy dla procesorów GPU NVIDIA za pomocą języka CUDA C, potrzebujesz dodatkowego oprogramowania. Zgodnie z wcześniejszą obietnicą, nie musisz za nic płacić.

bez niepotrzebnego zgłębiania szczegółów, którymi i tak zajmiemy się później, musisz wiedzieć, że Twoje programy w CUDA C będą działały na dwóch różnych procesorach, a w związku z tym potrzebujesz do ich komplikacji dwóch kompilatorów. Jeden do komplikacji kodu dla GPU, a drugi dla CPU. Kompilator kodu dla GPU pobierzesz z serwisu internetowego NVIDIA pod adresem [tp://developer.nvidia.com/cuda-downloads](http://developer.nvidia.com/cuda-downloads). Kliknij odnośnik GET LATEST CUDA TOOLKIT PRODUCTION RELEASE, tak aby przejść do strony widocznej na rysunku 2.1.



Rysunek 2.1. Strona pobierania narzędzi programistycznych CUDA

zwykając stronę w dół, znajdziesz opcje wyboru wersji oprogramowania dla 32- i 64-bitowych wersji systemów Windows XP, Windows Vista, Windows 7, Linux oraz Mac OS. Z dostępnych opcji należy wybrać CUDA Toolkit, czyli zestaw narzędzi potrzebnych do komplikacji przykładowych programów przedstawionych w tej książce. Dodatkowo możesz, choć nie jest to konieczne,

pobrać pakiet GPU Computing SDK zawierający wiele przykładów kodu źródłowego. Nie opisujemy ich w tej książce, ale stanowią one znakomite uzupełnienie prezentowanego przez nas materiału, a poza tym, jak to zwykle bywa z nauką programowania, im więcej przykładów kodu, tym lepiej. Należy także podkreślić, że chociaż prawie wszystkie przedstawione w tej książce przykłady powinny działać w systemach Linux, Windows i Mac OS, naszym priorytetem było dostosowanie ich do dwóch pierwszych z nich. Jeśli używasz systemu Mac OS X, musisz się liczyć z tym, że mogą wystąpić jakieś niedogodności.

2.2.4. STANDARDOWY KOMPILATOR JĘZYKA C

Jak zaznaczylśmy, do komplikacji przykładów potrzebny będzie nie tylko kompilator kodu dla GPU, lecz również kompilator kodu dla CPU. Po instalacji pakietu CUDA Toolkit zgodnie z wcześniejszymi wskazówkami masz już kompilator dla GPU. Natomiast kompilator dla CPU musisz jeszcze zdobyć. Poniżej znajdziesz wskazówki, skąd można go wziąć.

WINDOWS

W systemach Windows (XP, Vista, Server 2008 i 7) polecamy używanie kompilatora Microsoft Visual Studio. Najnowsza wersja CUDA 4.0 jest już obsługiwana przez Microsoft Visual Studio 2010, natomiast Microsoft Visual Studio 2005 i 2008 są obsługiwane przez starsze wersje CUDA. Gdy tylko pojawią się nowa wersja tego środowiska, NVIDIA porzuca obsługę starych wersji i przechodzi na najnowszą. Wielu programistów ma już w swoim komputerze którąś z wersji Visual Studio. Osoby te mogą pominąć dalszą część tego podrozdziału.

Jeśli nie masz wymienionego oprogramowania i nie chcesz go kupować, na stronach Microsoftu możesz znaleźć bezpłatną wersję o nazwie Visual Studio 2010 Express (lub starszą). Środowisko to raczej nie nadaje się do tworzenia komercyjnych programów, ale w zupełności wystarczy do nauki programowania w języku CUDA C. Jeśli zatem potrzebujesz oprogramowania Visual Studio, odwiedź stronę <http://www.microsoft.com/visualstudio/>¹.

LINUX

Większość dystrybucji Linuksa ma standardowo zainstalowany kompilator GNU C (gcc). Poniższe dystrybucje Linuksa mają odpowiednie wersje kompilatora gcc już dla CUDA 3.0:

- Red Hat Enterprise Linux 4.8
- Red Hat Enterprise Linux 5.3
- OpenSUSE 11.1
- SUSE Linux Enterprise Desktop 11

¹ Na stronie <http://www.ademiller.com/blogs/tech/2011/05/visual-studio-2010-and-cuda-easier-with-rc2/> znajduje się szczegółowy opis sposobu tworzenia projektu CUDA C w środowisku Microsoft Visual Studio 2010 — przyp. tłum.

- Ubuntu 9.04
- Fedora 10

Zagorzali wielbicie Linuksa wiedzą, że pakiety programowe tego systemu często działają na znacznie większej liczbie platform niż tylko „oficjalnie obsługiwane”. Także pakiet CUDA Toolkit nie jest tu wyjątkiem, a więc nawet jeśli na powyższej liście nie ma Twojego ulubionego systemu, warto i tak spróbować. Za zgodność w głównej mierze odpowiedzialne są wersje jądra systemu, kompilatora gcc oraz biblioteki glibc.

MAC OS X

Jeśli chcesz pracować w systemie Mac OS X, musisz mieć wersję nie starszą od numeru 10.5.7, a więc np. wersję 10.6 czyli Mac OS X Snow Leopard. Dodatkowo musisz zainstalować środowisko programistyczne Apple o nazwie Xcode, które zawiera kompilator gcc. Dla użytkowników programu Apple Developer Connection (ADC) oprogramowanie to jest dostępne bezpłatnie i można je pobrać pod adresem <http://developer.apple.com/tools/Xcode>. Kod źródłowy programów przedstawionych w tej książce został napisany w systemach Linux i Windows, ale powinien też działać bez żadnych modyfikacji w systemie Mac OS X.

2.3. Podsumowanie

Po zastosowaniu się do wskazówek zamieszczonych w tym rozdziale można rozpocząć pisanie programów w języku CUDA C. Osobom, które zdążyły się już pobawić przykładowymi programami z pakietu GPU Computing SDK, gratulujemy chęci do pracy! Oczywiście osoby, które tego nie zrobiły, też nie mają sobie nic do zarzucenia. Wszystko, co będzie potrzebne w czasie studiowania tej książki, znajduje się w tekście. Skoro wszystko jest już gotowe, czas wziąć się do pracy.

Rozdział 3

Podstawy języka CUDA C

W pierwszym rozdziale staraliśmy się przekonać Cię, jak duży potencjał obliczeniowy drzemie w procesorach graficznych oraz że Ty również możesz go wykorzystać. W drugim rozdziale opisaliśmy, jak skonfigurować środowisko programistyczne odpowiednie do komplikacji i uruchamiania programów w języku CUDA C. Jeśli ta karta jest pierwszą stroną, którą czytasz w tej książce, to zapewne szukasz tylko przykładów kodu, przeglądasz książkę w księgarni albo po prostu nie możesz się już doczekać, żeby rozpocząć programowanie. W porządku, nie ma sprawy. Skoro jesteś gotowy na rozpoczęcie pracy, to zaczynamy.

3.1. Streszczenie rozdziału

W tym rozdziale:

- Napiszesz pierwszy program w języku CUDA C.
- Dowiesz się, jaka jest różnica między kodem przeznaczonym dla **hosta** a kodem przeznaczonym dla **urządzenia**.
- Nauczysz się uruchamiać z hosta programy dla urządzeń.
- Poznasz metody użycia pamięci urządzenia na urządzeniach obsługujących CUDA.
- Nauczysz się pobierać z systemu informacje na temat urządzeń obsługujących CUDA.

3.2. Pierwszy program

Ponieważ obiecaliśmy praktyczną naukę na przykładach kodu, poniżej przedstawiamy pierwszy program w języku CUDA C. Zgodnie z kunsztem pisania książek na temat programowania komputerowego na początek prezentujemy program typu „Witaj, świecie!”.

3.2.1. WITAJ, ŚWIECIE!

```
#include "../common/book.h"
int main( void ) {
    printf( "Witaj, świecie!\n" );
    return 0;
}
```

W tej chwili pewnie się zastanawiasz, czy ta książka to nie jest przypadkiem jakiś żart. Przecież ten kod jest w języku C. A skoro tak, to czy CUDA C w ogóle istnieje? Tak, ten program jest w języku C i tak, CUDA C istnieje. Wcale nie żartujemy. Celem tego prostego przykładu jest wykazanie, że języków CUDA C i standardowego C, który dobrze znasz, tak naprawdę nic nie różni.

Powyższy program jest tak banalnie prosty dlatego, że w całości działa na **hoście**. Dokonamy tu pewnego rozróżnienia, które musisz zapamiętać: procesor CPU i pamięć systemową nazywamy **hostem**, natomiast GPU i jego pamięć nazywamy **urządzeniem**. Program ten jest tak bardzo podobny do innych Twoich programów, ponieważ nie wykorzystuje żadnych jednostek liczących oprócz hosta.

Abyś nie czuł się oszukany, w dalszej części książki będziemy ten przykład sukcesywnie rozbudowywać, wzbogacając go o coraz to nowe elementy. Teraz zobaczymy, jak wygląda kod programu korzystającego z GPU (czyli **urządzenia**). Funkcja wykonywana na urządzeniu nazywana jest **jądrem** (ang. *kernel*).

3.2.2. WYWOŁYWANIE FUNKCJI JĄDRA

Poniżej znajduje się przykład kodu, który już mniej przypomina zwykły język C niż poprzedni program „Witaj, świecie”.

```
#include <stdio.h>
__global__ void kernel( void ) {
}
int main( void ) {
    kernel<<<1,1>>>();
    printf( "Witaj, świecie!\n" );
    return 0;
}
```

W programie tym znajdują się dwa ważne dodatki:

- Pusta funkcja o nazwie `kernel()` z kwalifikatorem `__global__`.
- Wywołanie pustej funkcji ozdobione kodem `<<<1,1>>>`.

Pamiętamy z poprzedniego podrozdziału, że kod ten można skompilować za pomocą standar-dowego kompilatora języka C. Na przykład w systemie Linux do komplikacji kodu hosta można użyć kompilatora GNU gcc, natomiast w systemach Windows może to być kompilator Microsoft Visual C. Narzędzia NVIDIA podają kod kompilatorowi hosta i wszystko działa tak, jakby technologia CUDA wcale nie była użyta.

Kwalifikator `__global__` to dodatek pochodzący z języka CUDA C. Informuje on kompilator o tym, że dana funkcja powinna zostać skompilowana dla urządzenia, a nie dla hosta. W tym prostym przykładzie nvcc przekazuje funkcję `kernel()` do kompilatora zajmującego się kodem przeznaczonym dla urządzenia, a funkcję `main()` do kompilatora hosta, tak jak to było w po-przednim przykładzie.

Do czego służy wywołanie funkcji `kernel()` i dlaczego musimy dokonywać gwałtu na naszym standardowym kodzie C, dodając do niego trójkątne nawiasy i krotkę z liczbami? Zapnij pasy, bo teraz zacznie się ostra jazda.

Wiemy już, że w języku CUDA C potrzebny był jakiś sposób na zaznaczenie, że dana funkcja powinna być wykonywana na urządzeniu. Nie ma w tym nic nadzwyczajnego. Jest to prosty zabieg pozwalający w łatwy sposób wysłać kod hosta do jednego kompilatora, a kod urządzenia do innego. Sztuka polega na wywołaniu kodu urządzenia w kodzie hosta. Jedną z zalet języka CUDA C jest właśnie ta integracja językowa, dzięki której wywołania funkcji urządzenia wy-gładają bardzo podobnie do wywołań funkcji hosta. Później opiszymy to bardziej szczegółowo, a na razie wystarczy, że zapamiętasz, iż za wywołanie kodu urządzenia na hoście odpowiadają kompilator CUDA i system wykonawczy.

A zatem to nietypowe z wyglądu wywołanie uruchamia kod przeznaczony dla urządzenia, ale po co w takim razie są te trójkątne nawiasy i liczby? W nawiasach znajdują się argumenty, które powinny zostać przekazane do systemu wykonawczego. Nie są one przeznaczone dla kodu urządzenia, lecz są parametrami określającymi, w jaki sposób system wykonawczy ma uru-chomić kod urządzenia. Ich bardziej szczegółowy opis znajduje się w następnym rozdziale. Argumenty do kodu urządzenia przekazuje się w nawiasach okrągłych, tak samo jak w przy-padku zwykłych funkcji.

3.2.3. PRZEKAZYWANIE PARAMETRÓW

Obiecaliśmy, że będzie można przekazywać parametry do funkcji jądra, i teraz przyszedł czas na spełnienie tych obietnic. Spójrz na poniższą wersję programu „Witaj, świecie!” w kolejnej rozszerzonej wersji:

```
#include <stdio.h>
#include "../common/book.h"
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}
```

```

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );
    add<<<1,1>>>( 2, 7, dev_c );
    HANDLE_ERROR( cudaMemcpy( &c,
                           dev_c,
                           sizeof(int),
                           cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );
    return 0;
}

```

Nowości jest kilka, ale wszystkie one dotyczą tylko dwóch koncepcji:

- Do funkcji jądra parametry można przekazywać w taki sam sposób jak do zwykłych funkcji języka C.
- Aby zrobić cokolwiek pozytecznego na urządzeniu, np. zwrócić wartość do hosta, trzeba dokonać alokacji pamięci.

Jeśli chodzi o przekazywanie argumentów do funkcji jądra, to nie ma w tym nic specjalnego. Pomijając trójkątne nawiasy, wywołanie tej funkcji wygląda i działa dokładnie tak samo jak wywołanie każdej innej standardowej funkcji języka C. Wszystkie skomplikowane czynności związane z przesłaniem tych parametrów z hosta do urządzenia są wykonywane przez system wykonawczy.

O wiele ciekawszy jest dodatek funkcji alokacji pamięci o nazwie `cudaMalloc()`. Działa ona bardzo podobnie jak standardowa funkcja C o nazwie `malloc()`, lecz alokuje pamięć na urządzeniu za pośrednictwem systemu wykonawczego CUDA. Pierwszy argument to wskaźnik na wskaźnik, który ma wskazywać adres nowo alokowanej pamięci, a drugi określa rozmiar wykonywanej alokacji. Pomijając szczegółowo, że wartością zwrotną tej funkcji nie jest wskaźnik na nowo alokowany obszar pamięci, działanie tej funkcji aż do typu zwrotnego `void*` jest takie samo jak funkcji `malloc()`. Struktura `HANDLE_ERROR()`, w której znajdują się opisywane wywołania, to pomocnicze makro, które napisaliśmy specjalnie na potrzeby tej książki. Jeśli wykonanie wywołania spowoduje błąd, makro wykryje ten przypadek, wydrukuje odpowiedni komunikat o błędzie i zamknie program, zwracając kod `EXIT_FAILURE`. Możesz tego makra używać w swoich programach, ale mieć świadomość, że w kodzie przeznaczonym do użytku taka prosta procedura obsługi błędów może być niewystarczająca.

To porusza subtelną, ale ważną kwestię. Język CUDA C jest tak prosty i użyteczny głównie dlatego, że podczas jego używania zaciera się różnica między kodem przeznaczonym dla hosta i urządzenia. Jednak to programista musi uważać, aby nie wyłuskać wskaźnika zwróconego przez funkcję `cudaMalloc()` za pomocą kodu działającego na hoście. W kodzie hosta można go

przekazywać w różne miejsca, wykonywać na nim działania arytmetyczne, a nawet rzutować go na inny typ, ale jednego robić nie wolno — nie można go używać do zapisu ani odczytu pamięci.

Niestety kompilator nie chroni przed tego rodzaju błędami, ponieważ wskaźniki na pamięć urządzenia wyglądają tak samo, jak wskaźniki na pamięć hosta. A w związku z tym kompilator nie będzie robił trudności przy ich wyłuskiwaniu. Poniżej znajduje się zestawienie ograniczeń dotyczących używania wskaźników urządzenia:

Wskaźniki na pamięć alokowaną przez funkcję `cudaMalloc()` **można** przekazywać do funkcji działających na urządzeniu.

Wskaźników utworzonych przez funkcję `cudaMalloc()` **można** używać do odczytu i zapisu pamięci w kodzie działającym na urządzeniu.

Wskaźniki na pamięć alokowaną przez funkcję `cudaMalloc()` **można** przekazywać do funkcji działających na hoście.

Wskaźników utworzonych przez funkcję `cudaMalloc()` **nie można** używać do odczytu i zapisu pamięci w kodzie działającym na hoście.

Kto uważnie przeczytał powyższe akapity, może przewidzieć, o czym będzie mowa teraz. Do zwalniania pamięci alokowanej przez funkcję `cudaMalloc()` nie można używać funkcji `free()`. Do tego celu służy specjalna funkcja o nazwie `cudaFree()`, która działa dokładnie tak samo jak `free()`.

Pokazaliśmy, jak za pomocą hosta można alokować i zwalniać pamięć na urządzeniu, a przy okazji dobrze podkreśliliśmy, że pamięci, tej z poziomu hosta, modyfikować się nie da. Dwa ostatnie wiersze kodu źródłowego przedstawionego programu ilustrują dwie najczęściej używane metody dostępu do pamięci urządzenia: za pomocą wskaźników urządzenia w kodzie działającym na urządzeniu oraz przy użyciu funkcji `cudaMemcpy()`.

Jeśli chodzi o kod działający na urządzeniu, to wskaźników używa się w nim tak samo jak w standardowym kodzie C działającym na hoście. Instrukcja `*c = a + b` działa dokładnie tak, jak na to wygląda, tzn. sumuje wartości argumentów `a` i `b`, a następnie uzyskany wynik zapisuje w pamięci w miejscu wskazywanym przez wskaźnik `c`. To powinno być tak banalne, że aż nieciekawe.

Wiesz już, co możesz, a czego nie możesz robić ze wskaźnikami urządzenia w kodzie działającym na hoście i urządzeniu. Dokładnie w takich samych proporcjach dotyczy to wskaźników na pamięć hosta. Na urządzeniu można je swobodnie przekazywać itd., ale wszelkie próby użycia ich w celu dostępu do pamięci urządzenia zakończą się fiaskiem. Podsumowując, wskaźniki hosta służą do manipulowania pamięcią hosta, a wskaźniki urządzenia — do manipulowania pamięcią urządzenia.

Dodatkowo dostęp do pamięci urządzenia na hoście można uzyskać za pomocą funkcji `cudaMemcpy()`. Działa ona dokładnie tak samo jak standardowa funkcja C o nazwie `memcpy()` wzbogacona o dodatkowy parametr pozwalający określić, który ze wskaźników (źródłowy

czy docelowy) wskazuje na pamięć urządzenia. Zwróć uwagę, że ostatnim parametrem funkcji `cudaMemcpy()` jest `cudaMemcpyDeviceToHost`, co oznacza, że wskaźnik źródłowy wskazuje pamięć urządzenia, a docelowy — pamięć hosta.

Oczywiście istnieje też parametr `cudaMemcpyHostToDevice`, który oznacza, że dane źródłowe znajdują się na hoście, a miejsce ich przeznaczenia — pod odpowiednim adresem na urządzeniu. Można także podać informację, że **oba** wskaźniki wskazują pamięć urządzenia. Służy do tego parametr `cudaMemcpyDeviceToDevice`. Jeśli oba wskaźniki, źródłowy i docelowy, wskazują miejsca w pamięci hosta, to do kopiowania danych pomiędzy nimi można użyć standardowej funkcji `memcpy()`.

3.3. Sprawdzanie właściwości urządzeń

Ponieważ będziemy chcieli używać pamięci urządzeń i wykonywać na nich kod, dobrze by było, gdybyśmy mogli sprawdzić, ile pamięci mamy do dyspozycji i jakie są ich możliwości. Ponadto w komputerze może być więcej niż jedno urządzenie obsługujące technologię CUDA. W takich przypadkach zdecydowanie chcielibyśmy móc odróżnić od siebie poszczególne układy.

Na przykład na rynku dostępnych jest wiele płyt głównych, które posiadają zintegrowany układ graficzny NVIDIA. Jeśli producent albo użytkownik komputera doda jeszcze kartę grafiki w postaci karty rozszerzeń, to w komputerze tym będą się znajdować dwa procesory obsługujące technologię CUDA. Także niektóre produkty NVIDIA, np. karty GeForce GTX 295, mają po dwa procesory GPU, a więc zawierające je komputery dysponują dwoma procesorami opartymi na architekturze CUDA.

Przed rozpoczęciem pisania kodu dla urządzeń powinno się dokładnie sprawdzić, które urządzenia są dostępne i jakie są ich możliwości. Nie będzie z tym żadnego problemu. Najpierw trzeba sprawdzić, ile urządzeń opartych na architekturze CUDA znajduje się w systemie. Wszystkie one mogą wykonywać funkcję jądra napisaną w języku CUDA C. Liczbę urządzeń CUDA sprawdza się za pomocą funkcji `cudaGetDeviceCount()`. Nie musimy dodawać, że liczymy na nagrodę za najbardziej innowacyjną nazwę funkcji wszechczasów:

```
int count;
HANDLE_ERROR( cudaGetDeviceCount( &count ) );
```

Wynik zwrócony przez funkcję `cudaGetDeviceCount()` można przejrzeć za pomocą iteracji, tak aby uzyskać szczegółowe informacje na temat każdego z urządzeń. Dane na temat właściwości urządzeń system wykonawczy zwraca w postaci struktury typu `cudaDeviceProp`. Czego można się z niej dowiedzieć? W CUDA 3.0 struktura `cudaDeviceProp` zawiera następujące dane:

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
    int maxTexture1D;
    int maxTexture2D[2];
    int maxTexture3D[3];
    int maxTexture2DArray[3];
    int concurrentKernels;
}
```

Część z tych składowych nie wymaga objaśnienia, ale niektóre z pewnością tak (tabela 3.1).

Na razie nie ma sensu zagłębiać się zbytnio w szczegóły. Dlatego w powyższej tabeli brak wielu ważnych informacji o opisanych tam właściwościach. Można je znaleźć w podręczniku *NVIDIA CUDA Reference Manual*. Bardzo się przydadzą, gdy zaczniesz pisać własne aplikacje. Na razie jednak skupimy się na odpytywaniu urządzeń i sprawdzaniu ich właściwości. Teraz nasze zapytanie do urządzenia wygląda tak:

```
#include "../common/book.h"
int main( void ) {
    cudaDeviceProp prop;
    int count;
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        // Kod wykorzystujący zdobyte informacje o właściwościach
    }
}
```

Tabela 3.1. Właściwości urządzeń CUDA

Właściwość	Opis
char name[256];	Łańcuch ASCII stanowiący identyfikator urządzenia, np. GeForce GTX 280
size_t totalGlobalMem	Ilość (w bajtach) pamięci globalnej dostępnej na urządzeniu
size_t sharedMemPerBlock	Maksymalna ilość (w bajtach) pamięci wspólnej, jaką może być używana przez jeden blok
int regsPerBlock	Liczba 32-bitowych rejestrów na blok
int warpSize	Liczba wątków w osnowie
size_t memPitch	Maksymalna szerokość (w bajtach) kopii pamięci
int maxThreadsPerBlock	Maksymalna liczba wątków w bloku
int maxThreadsDim[3]	Maksymalna liczba wątków w każdym wymiarze bloku
int maxGridSize[3]	Liczba bloków dozwolona w każdym wymiarze siatki
size_t totalConstMem	Ilość dostępnej pamięci stałej
int major	Główny numer wersji potencjału obliczeniowego (ang. <i>compute capability</i>) urządzenia
int minor	Drugorzędny numer wersji potencjału obliczeniowego urządzenia
size_t textureAlignment	Wymagania urządzenia dotyczące wyrównania tekstur
int deviceOverlap	Wartość logiczna określająca, czy urządzenie może jednocześnie wykonywać funkcje cudaMemcpy() oraz jądra
int multiProcessorCount	Liczba wieloprocesorów w urządzeniu
int kernelExecTimeoutEnabled	Wartość logiczna określająca, czy na tym urządzeniu istnieje ograniczenie czasowe wykonywania funkcji jądra
int integrated	Wartość logiczna określająca, czy dany GPU jest układem zintegrowanym (tzn. jest częścią chipsetu, a nie osobnym procesorem)
int canMapHostMemory	Wartość logiczna określająca, czy urządzenie może rzutować pamięć hosta na przestrzeń adresową urządzenia CUDA
int computeMode	Wartość określająca tryb działania urządzenia: domyślny, wyłączny lub zakazany
int maxTexture1D	Maksymalny rozmiar tekstur jednowymiarowych
int maxTexture2D[2]	Maksymalny rozmiar tekstur dwuwymiarowych
int maxTexture3D[3]	Maksymalny rozmiar tekstur trójwymiarowych
int maxTexture2DArray[3]	Maksymalny rozmiar tablic tekstur dwuwymiarowych
int concurrentKernels	Wartość logiczna określająca, czy urządzenie pozwala na jednoczesne wykonywanie wielu funkcji jądra w jednym kontekście

Wiedząc, jakie pola są dostępne, można w miejsce dwuznacznego komentarza „Kod wykorzystujący...” wpisać coś bardziej pożytecznego:

```
#include "../common/book.h"
int main( void )
{
    cudaDeviceProp prop;
    int count;
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for ( int i=0; i< count; i++ ) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        printf( " --- Ogólne informacje o urządzeniu %d ---\n", i );
        printf( "Nazwa: %s\n", prop.name );
        printf( "Potencjał obliczeniowy: %.2f.%d\n", prop.major, prop.minor );
        printf( "Zegar: %d\n", prop.clockRate );
        printf( "Ustawienie deviceOverlap: " );
        if (prop.deviceOverlap)
            printf( " Włączone\n" );
        else
            printf( " Wyłączone\n" );
        printf( "Limit czasu działania jądra: " );
        if (prop.kernelExecTimeoutEnabled)
            printf( " Wyłączony\n" );
        else
            printf( " Włączony\n" );
        printf( " --- Informacje o pamięci urządzenia %d ---\n", i );
        printf( "Ilość pamięci globalnej: %ld\n", prop.totalGlobalMem );
        printf( "Ilość pamięci stałej: %ld\n", prop.totalConstMem );
        printf( "Maks. szerokość pamięci: %ld\n", prop.memPitch );
        printf( "Wyrównanie tekstur: %ld\n", prop.textureAlignment );
        printf( " --- Informacje na temat wieloprocesorów urządzenia %d ---\n",
                i );
        printf( "Liczba wieloprocesorów: %d\n",
                prop.multiProcessorCount );
        printf( "Pamięć wspólna na wieloprocesor: %ld\n", prop.sharedMemPerBlock );
        printf( "Rejestry na wieloprocesor: %d\n", prop.regsPerBlock );
        printf( "Liczba wątków w osnowie: %d\n", prop.warpSize );
        printf( "Maks. liczba wątków na blok: %d\n",
                prop.maxThreadsPerBlock );
        printf( "Maks. liczba wymiarów wątków: (%d, %d, %d)\n",
                prop.maxThreadsDim[0], prop.maxThreadsDim[1],
                prop.maxThreadsDim[2] );
        printf( "Maks. liczba wymiarów siatki: (%d, %d, %d)\n",
                prop.maxGridSize[0], prop.maxGridSize[1],
                prop.maxGridSize[2] );
        printf( "\n" );
    }
}
```

3.4. Korzystanie z wiedzy o właściwościach urządzeń

Do czego — oprócz drukowania wszystkich możliwych informacji o układzie graficznym — może przydać się możliwość sprawdzania właściwości GPU? Ponieważ jesteśmy programistami i chcemy, aby nasze programy były jak najszybsze, możemy zechcieć wybrać GPU z największą liczbą wieloprocesorów. A jeśli jądro wymaga jak najbliżej współpracy z procesorem CPU, to lepiej jest je uruchomić na zintegrowanym GPU, który korzysta z tej samej pamięci systemowej co CPU. Obie te właściwości można zbadać za pomocą funkcji `cudaGetDeviceProperties()`.

Przypuśćmy, że piszemy program, którego sprawne działanie zależy od możliwości wykonywania obliczeń na liczbach zmiennoprzecinkowych podwójnej precyzji. Z dodatku A przewodnika *NVIDIA CUDA Programming Guide* dowiadujemy się, że obsługa tego rodzaju działań matematycznych zaczyna się od układów wyposażonych w potencjał obliczeniowy o numerze 1.3. Aby więc dało się uruchomić tę aplikację, w komputerze musi być przynajmniej jedno urządzenie o potencjałie obliczeniowym nie niższym niż 1.3.

Mając do dyspozycji funkcje `cudaGetDeviceCount()` i `cudaGetDeviceProperties()`, możemy przerzeć właściwości wszystkich urządzeń i poszukać takiego, które ma główny numer wersji większy od 1 albo takiego, które ma główny numer równy 1, a drugorzędny nie mniejszy od 3. Ponieważ sprawdzenie to wykonuje się dość często i nie jest to zbyt przyjemne, postanowiono ten proces zautomatyzować. Najpierw szukane właściwości należy wstawić do struktury `cudaDeviceProp`.

```
cudaDeviceProp prop;
memset( &prop, 0, sizeof( cudaDeviceProp ) );
prop.major = 1;
prop.minor = 3;
```

Następnie strukturę tę przekazuje się do funkcji `cudaChooseDevice()`. Funkcja ta automatycznie znajdzie urządzenie spełniające postawione wymagania i zwróci jego identyfikator, który następnie można przekazać do funkcji `cudaSetDevice()`. Od tej pory wszystkie działania będą wykonywane na urządzeniu znalezionym przez funkcję `cudaChooseDevice()`.

```
#include "../common/book.h"
int main( void ) {
    cudaDeviceProp prop;
    int dev;
    HANDLE_ERROR( cudaGetDevice( &dev ) );
    printf( "Identyfikator bieżącego urządzenia CUDA: %d\n", dev );
    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 3;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
    printf( "Identyfikator urządzenia CUDA o właściwościach najbliższych
    do wersji 1.3: %d\n", dev );
    HANDLE_ERROR( cudaSetDevice( dev ) );
}
```

Coraz częściej spotyka się systemy z kilkoma GPU. Na przykład wiele płyt głównych opartych na układzie NVIDIA ma zintegrowany układ graficzny bazujący na architekturze CUDA. Gdy do takiego systemu komputerowego doda się jeszcze kartę graficzną w postaci karty rozszerzeń, powstaje system wieloprocesorowy. Ponadto technologia NVIDIA o nazwie SLI umożliwia jednoczesne korzystanie z dwóch kart graficznych w postaci kart rozszerzeń. W każdym z opisanych przypadków jeden z dostępnych GPU może być dla danego programu bardziej odpowiedni niż inny. Jeśli więc piszesz programy wymagające konkretnych funkcji GPU lub potrzebujące najszybszego dostępnego układu, to koniecznie zapoznaj się z tym API, gdyż nigdy nie ma gwarancji, że system wykonawczy CUDA sam automatycznie wybierze najlepszą jednostkę.

3.5. Podsumowanie

Pierwsze kroki za płyty. Okazało się, że napisanie programu w CUDA C jest łatwiejsze, niż można się było spodziewać. Język ten to tak naprawdę standardowy język C z pewnymi dodatkami pozwalającymi zdecydować, które części kodu mają być wykonywane na urządzeniu, a które na hoście. Do zaznaczania, że dana funkcja ma zostać wykonana przez GPU, służy słowo kluczowe `_global_`. Do używania pamięci GPU służy natomiast specjalne API CUDA, którego funkcje są podobne do standardowych funkcji języka C `malloc()`, `memcpy()` i `free()`. Ich nazwy to odpowiednio `cudaMalloc()`, `cudaMemcpy()` oraz `cudaFree()`. Można ich używać do alokowania pamięci na urządzeniu, kopowania danych między urządzeniem a hostem oraz zwalniania nieużywanej pamięci na urządzeniu.

W dalszych rozdziałach znajduje się więcej przykładów efektywnego wykorzystania urządzenia jako równoległego koprocesora. Celem tego rozdziału było tylko pokazanie, jak łatwo da się rozpocząć pracę w języku CUDA C, natomiast w następnym rozdziale pokażemy, jak łatwo można równolegle wykonać kod na GPU.

Rozdział 4

Programowanie równoległe w języku CUDA C

W poprzednim rozdziale wykazaliśmy, jak łatwo jest napisać program wykonywany przez GPU. Obliczyliśmy nawet sumę dwóch liczb, aczkolwiek niezbyt dużych, bo tylko 2 i 7. Przyznajemy, tamten przykład nie był zbyt porywający, ani też praktyczny. Mamy jednak cichą nadzieję, że dzięki niemu mogłeś się przekonać, iż pisanie programów w CUDA C to nic trudnego, i że obudziliśmy w Tobie ciekawość, aby dowiedzieć się więcej na ten temat. Jedną z największych zalet wykonywania obliczeń na procesorze GPU jest możliwość wykorzystania jego potencjału w zakresie przetwarzania równoległego. Dlatego w tym rozdziale znajduje się opis technik równoległego wykonywania kodu CUDA C na GPU.

4.1. Streszczenie rozdziału

W tym rozdziale:

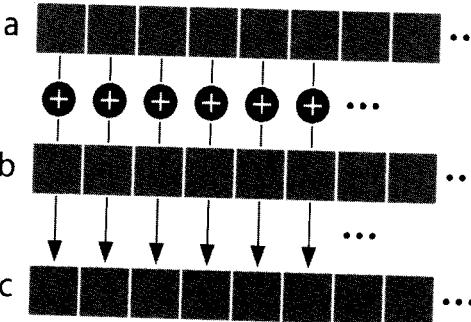
- Poznasz podstawową technikę programowania równoległego CUDA.
- Napiszesz pierwszy równoległy program w języku CUDA C.

4.2. Programowanie równoległe w technologii CUDA

W jednym z poprzednich rozdziałów pokazaliśmy, jak spowodować wykonanie standardowej funkcji języka C na urządzeniu. W tym celu należy do funkcji dodać słowo kluczowe `_global_`, a następnie wywołać ją za pomocą specjalnej składni z użyciem nawiasów trójkątnych. Nie dość, że jest to technika prymitywna, to na dodatek jeszcze i bardzo nieefektywna, gdyż spece z NVIDIA przecież tak zaprojektowali procesory graficzne, aby mogły wykonywać setki obliczeń równocześnie. Na razie nie skorzystaliśmy z tej możliwości, ponieważ dotychczasowe programy zawierały tylko jądro działające na GPU szeregowo. W tym rozdziale dowiesz się, jak napisać jądro wykonujące obliczenia równolegle.

4.2.1. SUMOWANIE WEKTORÓW

Poniżej przedstawiamy prosty program, na którego przykładzie wprowadzimy pojęcie wątków i pokażemy, jak ich używać. Przypuśćmy, że mamy dwie listy liczb i chcemy zsumować ich elementy znajdujące się na odpowiadających sobie pozycjach, a następnie wyniki zapisać w trzeciej liście. Ilustracja przebiegu tego procesu znajduje się na rysunku 4.1. Osoby znające algebrę liniową od razu rozpoznają, że jest to sumowanie dwóch wektorów.



Rysunek 4.1. Sumowanie dwóch wektorów

SUMOWANIE WEKTORÓW PRZY UŻYCIU PROCESORA CPU

Najpierw zobaczymy, jak taką operację można wykonać za pomocą zwykłego kodu w języku C:

```
#include "../common/book.h"
#define N 10
void add( int *a, int *b, int *c ) {
    int tid = 0; // To jest CPU nr zero, a więc zaczynamy od zera
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1; // Mamy tylko jeden CPU, a więc zwiększamy o jeden
    }
}
int main( void ) {
    int a[N], b[N], c[N];
    // Zapełnienie tablic a i b danymi za pomocą CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    add( a, b, c );
    // Wyświetlenie wyników
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    return 0;
}
```

Większa część kodu tego programu nie wymaga objaśnień. Napiszemy tylko kilka słów o funkcji `add()`, aby wy tłumaczyć się z tego, dlaczego ją niepotrzebnie skomplikowaliśmy.

```
void add( int *a, int *b, int *c ) {
    int tid = 0; // To jest CPU zero, a więc zaczynamy od zera
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1; // Mamy tylko jeden CPU, a więc zwiększamy o jeden
    }
}
```

Suma obliczana jest za pomocą pętli `while`, w której zmienna indeksowa o nazwie `tid` przyjmuje wartości od 0 do $N-1$. Sumowane są kolejno odpowiadające sobie elementy tablic `a[]` i `b[]`, a wyniki są zapisywane w odpowiednich elementach tablicy `c[]`. Działanie to można by było zapisać prościej:

```
void add( int *a, int *b, int *c ) {
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Skorzystaliśmy z nieco bardziej pokrętnej metody, aby uwidoczyć możliwość równoleglenia tego kodu, gdyby działał w systemie wieloprocesorowym lub z procesorem wielordzeniowym. Gdyby na przykład procesor był dwurdzeniowy, to można by było zmienić wartość inkrementacji na 2 i dla pierwszego rdzenia zainicjować pętlę z wartością `tid = 0`, a dla drugiego z wartością `tid = 1`. Wówczas pierwszy rdzeń sumowałby elementy znajdujące się pod indeksami parzystymi, a drugi — pod indeksami nieparzystymi. W związku z tym na poszczególnych rdzeniach procesora byłby wykonywany następujący kod:

RDZEN 1

```
void add( int *a, int *b, int *c ) {
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

RDZEN 2

```
void add( int *a, int *b, int *c ) {
    int tid = 1;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

Oczywiście, aby to zadziałało zgodnie z opisem, trzeba by było napisać sporo dodatkowego kodu. Należałyby utworzyć wątki robocze do wykonywania funkcji `add()` oraz przyjąć założenie, że wszystkie wątki będą działać równolegle, co niestety nie zawsze jest prawdą.

SUMOWANIE WEKTORÓW ZA POMOCĄ PROCESORA GPU

Działanie to można zrealizować w bardzo podobny sposób na procesorze GPU, pisząc funkcję `add()` dla urządzenia. Kod będzie podobny do tego, który został już pokazany. Najpierw jednak zapoznamy się z funkcją `main()`. Mimo że jej implementacja dla GPU jest nieco inna niż dla CPU, to nie ma w niej jednak nic nowego:

```
#include "../common/book.h"
#define N 10
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    // Alokacja pamięci na GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
    // Zapelnienie tablic a i b na CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    // Kopiowanie tablic a i b do GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                           cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                           cudaMemcpyHostToDevice ) );
    add<<<N,1>>>( dev_a, dev_b, dev_c );
    // Kopiowanie tablicy c z GPU do CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                           cudaMemcpyDeviceToHost ) );
    // Wyświetlenie wyniku
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    // Zwolnienie pamięci alokowanej na GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Można łatwo zauważać pewne powtarzające się wzorce:

- Alokacja trzech tablic na urządzeniu za pomocą funkcji `cudaMalloc()`: tablice `dev_a` i `dev_b` zawierają dane wejściowe, a `dev_c` — wyniki.
- Ponieważ leży nam na sercu czystość środowiska, sprzątamy po sobie za pomocą funkcji `cudaFree()`.

- Za pomocą funkcji `cudaMemcpy()` z parametrem `cudaMemcpyHostToDevice` kopujemy dane wejściowe na urządzenie, a następnie kopujemy wynik do hosta za pomocą tej samej funkcji z parametrem `cudaMemcpyDeviceToHost`.
- Uruchamiamy funkcję `add()` urządzenia w funkcji `main()` na hoście, używając składni z trzema nawiasami trójkątnymi.

Przy okazji warto wyjaśnić, dlaczego tablice są zapełniane danymi przez CPU. Nie ma żadnego konkretnego powodu, aby tak było. Gdyby w dodatku operację tę przeniesiono na GPU, to by została wykonana szybciej. Jednak celem tego przykładu było zaprezentowanie sposobu implementacji konkretnego algorytmu (w tym przypadku sumowania wektorów) do wykonania na procesorze GPU. Wyobraź sobie, że jest to tylko jeden z wielu etapów wykonywania jakiejś większej aplikacji, w której tablice `a[]` i `b[]` zostały utworzone przez jakiś inny algorytm albo wczytane z dysku twardego. Po prostu udawajmy, że dane pojawiły się nie wiadomo skąd i że trzeba coś z nimi zrobić.

Wracając do sedna, kod źródłowy tej funkcji `add()` jest podobny do poprzedniej implementacji dla CPU:

```
global void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x; // Działanie na danych znajdujących się pod tym indeksem
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

I znowu widać znany już wzorzec postępowania:

- Ta funkcja `add()` zostanie wykonana na urządzeniu. Spowodowaliśmy to poprzez dodanie do standardowego kodu tej funkcji w języku C słowa kluczowego `global`.

Jak na razie nie pokazaliśmy jeszcze nic nowego, pomijając fakt, że ten program już nie sumuje liczb 2 i 7. A jednak są dwie rzeczy **warte uwagi**. Nowe są parametry w nawiasach trójkątnych oraz kod źródłowy jądra.

Do tej pory funkcja jądra była zawsze wywoływana za pomocą następującej ogólnej składni:

`Jądro<<<1,1>>>(param1, param2, ...);`

Natomiast tym razem zmieniła się liczba w nawiasach:

`add<<<N,1>>>(dev_a, dev_b, dev_c);`

O co tu chodzi?

Przypomnijmy, że liczby w nawiasach trójkątnych pozostawiliśmy bez objaśnienia. Napisaliśmy jedynie, że stanowią one dla systemu wykonawczego informację o sposobie uruchomienia jądra. Pierwsza z nich określa liczbę równoległych bloków, w których urządzenie ma wykonywać jądro. W tym przypadku została podana wartość N .

Gdyby na przykład w programie użyto wywołania jądra `kernel<<<2,1>>()`, to system wykonawczy utworzyłby dwie jego kopie i wykonywałby je równolegle. Każde z takich równoległych wywołań nazywa się **blokiem**. Gdyby napisano wywołanie `kernel<<<256,1>>()`, to system utworzyłby **256 bloków** wykonywanych równolegle na GPU. Programowanie równolegle jeszcze nigdy nie było takie proste.

Teraz nasuwa się pytanie: skoro GPU wykonuje N kopii funkcji jądra, to jak poznać, który blok wykonuje daną kopię kodu? Aby odpowiedzieć na to pytanie, musimy poznać drugą z nowości wprowadzonych w tej aplikacji. Znajduje się ona w kodzie jądra, a konkretnie chodzi o zmienną `blockIdx.x`:

```
_global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x; // Działanie na danych znajdujących się pod tym indeksem
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Na pierwszy rzut oka wydaje się, że zmienna ta powinna podczas komplikacji spowodować błąd składni, ponieważ przypisujemy ją do zmiennej `tid`, mimo że nigdzie nie ma jej definicji. A jednak zmiennej `blockIdx` nie trzeba definiować, ponieważ jest to jedna ze standardowych zmiennych systemu wykonawczego CUDA. Jej przeznaczenia można domyślić się po nazwie, a najciekawsze jest to, że używamy jej nawet zgodnie z przeznaczeniem. Zawiera ona indeks bloku, który aktualnie wykonuje dany kod urządzenia.

Dlaczego w takim razie zmienna ta nie nazywa się po prostu `blockIdx`, tylko `blockIdx.x`? Ponieważ w języku CUDA C można definiować grupy bloków w dwóch wymiarach. Jest to przydatne w rozwiązywaniu dwuwymiarowych problemów, np. wykonywaniu działań na macierzach albo przy przetwarzaniu grafiki, gdyż pozwala uniknąć kłopotliwego zamianiania współrzędnych liniowych na prostokątne. Nie masz się co przejmować, jeśli nie wiesz, o co chodzi. Po prostu pamiętaj, że czasami indeksowanie dwuwymiarowe jest wygodniejsze od jednowymiarowego. Ale **nie musisz** z tego korzystać. Nie pogniewamy się.

Liczبę równoległych bloków w wywołaniu jądra ustawiliśmy na N . Zbiór równoległych bloków nazywa się **siatką**. Zatem nasze wywołanie informuje system wykonawczy, że chcemy utworzyć jednowymiarową siatkę zawierającą N bloków (wartości skalarne są interpretowane jako jednowymiarowe). Każdy z tych wątków będzie miał inną wartość zmiennej `blockIdx.x`, a więc pierwszy będzie miał 0, a ostatni $N-1$. Wyobraź sobie cztery bloki, wszystkie wykonujące ten sam kod urządzenia, ale każdy z inną wartością zmiennej `blockIdx.x`. Poniżej znajduje się kod, jaki zostałby wykonany przez każdy z tych czterech bloków po podstawieniu w miejsce zmiennej `blockIdx.x` odpowiedniej wartości:

BLOK 1

```
_global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOK 2

```
_global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOK 3

```
_global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOK 4

```
_global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Jeśli pamiętasz kod dla CPU pokazany na początku, to pamiętasz też, że w celu obliczenia sumy wektorów trzeba było przejść przez indeksy od 0 do $N-1$. Ponieważ system wykonawczy, wywołując blok, od razu wstawia w nim jeden z tych indeksów, wykonuje więc on za nas większość pracy. A ponieważ nie jesteśmy zbyt pracowici, bardzo nam się to podoba, ponieważ dzięki temu mamy więcej czasu na pisanie na blogu o tym, jak nam się nic nie chce.

A oto ostatnie pytanie, które do tej pory pozostawało bez odpowiedzi: dlaczego sprawdzamy, czy zmienna `tid` ma wartość mniejszą od N ? Okazuje się, że zmienna ta zawsze **powinna** być mniejsza od N , ponieważ tak uruchomiliśmy jądro, iż warunek ten musi być spełniony. Niestety nasze pragnienie leniuchowania doprowadza nas do paranoicznego strachu przed tym, że ktoś złamie nasze warunki. A złamanie przyjętych warunków nieuchronnie prowadzi do błędów. W wyniku tego zamiast pisać bloga, musimy siedzieć po nocach, analizować komunikaty o błędach, szukać przyczyn niewłaściwego działania programu i ogólnie robić wiele rzeczy, na które nie mamy ochoty. Gdybyśmy nie sprawdzali, czy zmienna `tid` jest mniejsza od N , i w pewnym momencie pobraли zawartość pamięci, która do nas nie należy, to byśmy wpadli w tarapaty. Mogliby to nawet spowodować zakończenie działania jądra, ponieważ GPU mają wbudowane wyrafinowane jednostki zarządzające pamięcią, które zamkują każdy proces, który by łamał zasady korzystania z pamięci.

Jeśli w programie wystąpi tego rodzaju błąd, jedno z makr `HANDLE_ERROR()`, którymi szczodrze **sypię** w całym kodzie, wykryje go i poinformuje Cię o tym. Należy pamiętać, że tak samo jak w standardowym języku C, funkcje zwracają kody błędów nie bez powodu. Wiemy, że łatwo **ulec** pokusie, aby zignorować pojawiający się kod błędu, ale chcielibyśmy zaoszczędzić Ci wielu przykrych godzin, których sami nie zdołaliśmy uniknąć, i dlatego nalegamy, aby **zawsze weryfikować wynik wszystkich działań, które mogą się nie udać**. Jak to zwykle bywa, żaden z tych błędów pewnie nie spowoduje natychmiastowego zamknięcia programu. Zamiast tego będą raczej **wywoływać** najrozmaitsze nietypowe i nieprzyjemne efekty uboczne w dalszej perspektywie.

W tym momencie wiesz już, jak na GPU wykonać kod równolegle. Możliwe, że mówiono Ci, iż jest to bardzo skomplikowane albo że trzeba znać się na programowaniu grafiki, aby tego dokonać. Dotychczasowe przykłady stanowią jednak dowód na to, że dzięki językowi CUDA C jest zupełnie inaczej. Ostatni program sumuje tylko dwa wektory zawierające po 10 elementów. Jeśli chcesz zobaczyć równolegle wykonywanie kodu w pełnej skali, zmień wierszu `#define N` 10 liczbę na 10000 albo 50000, tak aby utworzyć kilkadziesiąt tysięcy równoległych bloków wykonawczych. Pamiętaj tylko, że w każdym wymiarze maksymalna liczba bloków wynosi 65535. Jest to ograniczenie sprzętowe, którego przekroczenie wywoła wiele różnych błędów w programie. W następnym rozdziale nauczysz się pracować w tym wyznaczonym zakresie.

4.2.2. ZABAWNY PRZYKŁAD

Wcale nie twierdzimy, że dodawanie wektorów to nie jest świetna zabawa, ale teraz pokażemy program, który zaspokoi wielbicieli bardziej wyszukanych efektów specjalnych.

Program ten będzie wyświetlał fragmenty zbioru Julii. Dla niewtajemniczonych wyjaśniamy, że zbiór Julii to granica pewnej klasy funkcji w zbiorze liczb zespolonych. To chyba brzmi jeszcze gorzej niż dodawanie wektorów czy mnożenie macierzy. Lecz dla prawie wszystkich wartości parametrów tych funkcji granica ta tworzy fraktal, czyli jedną z najpiękniejszych i zarazem największych matematycznych osobliwości.

Obliczenia, jakie należy wykonać w celu wygenerowania takiego zbioru, są stosunkowo proste. Wszystko sprowadza się do iteracyjnego rozwiązywania równania, którego parametrami są punkty płaszczyzny zespolonej. Punkty, dla których ciąg rozwiązań równania dąży do nieskończoności, **nie należą** do zbioru. Natomiast punkty, dla których ciąg rozwiązań równania nie dąży do nieskończoności, **należą** do zbioru.

Równanie, o które chodzi, pokazano na listingu 4.1. Jak widać, jest ono bardzo proste do obliczenia:

Listing 4.1.

$$Z_{n+1} = Z_n^2 + C$$

Aby więc obliczyć jedną iterację powyższego równania, należałoby podnieść do kwadratu bieżącą wartość i dodać stałą C . W ten sposób obliczyłoby się kolejną wartość równania.

ZBIÓR JULII NA CPU

Poniżej przedstawiamy kod źródłowy programu obliczającego i wizualizującego zbiór Julii. Ponieważ jest on bardziej skomplikowany niż wszystkie poprzednie, podzieliliśmy go na części. Dalej pokazany jest też ten kod w całości.

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();
```

```
    kernel( ptr );
    bitmap.display_and_exit();
}
```

Funkcja główna jest bardzo prosta. Tworzy przy użyciu funkcji bibliotecznej mapę bitową o odpowiednim rozmiarze, a następnie do funkcji jądra przekazuje wskaźnik na tę mapę.

```
void kernel( unsigned char *ptr ){
    for ( int y=0; y<DIM; y++ ) {
        for ( int x=0; x<DIM; x++ ) {
            int offset = x + y * DIM;
            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
}
```

Funkcja jądra po prostu przegląda iteracyjnie wszystkie punkty, które wyrenderujemy, i dla każdego z nich wywołuje funkcję `julia()`, aby sprawdzić, czy należy on do zbioru, czy nie. Jeśli dany punkt należy do zbioru, funkcja zwraca 1, jeśli nie — zwraca 0. W pierwszym przypadku kolor punktu ustawiamy na czerwony, a w drugim na czarny. Wybór konkretnych kolorów nie ma znaczenia, więc możesz ustawić swoje ulubione.

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);
    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);
    int i = 0;
    for ( i=0; i<200; i++ ) {
        a = a * a + c;
        if ( a.magnitude2() > 1000 )
            return 0;
    }
    return 1;
}
```

Powyższa funkcja stanowi serce programu. Najpierw zamienia współrzędne piksela na współrzędne na płaszczyźnie zespolonej. W celu wypośrodkowania tej płaszczyzny na obrazie stosujemy przesunięcie o $DIM/2$. Następnie skalujemy każdą współrzędną o $DIM/2$, tak aby obraz zajmował zakres od -1.0 do 1.0. Zatem dla dowolnego punktu (x,y) na płaszczyźnie zespolonej otrzymujemy punkt $((DIM/2-x)/(DIM/2), (DIM/2-y)/(DIM/2))$.

Aby umożliwić powiększanie i pomniejszanie obrazu, wprowadziliśmy współczynnik `scale`. Aktualnie skala została ustawiona na sztywno na 1.5, ale można tę wartość dowolnie zmienić. Bardziej ambitne osoby mogą nawet zdefiniować to ustawienie jako parametr wiersza polecen.

Po obliczeniu współrzędnych punktu na płaszczyźnie zespolonej przechodzimy do sprawdzenia, czy należy on do zbioru Julii. Pamiętamy, że aby to zrobić, trzeba obliczyć wartości rekurencyjnego równania $Z_{n+1} = Z_n^2 + C$. Ponieważ C jest stałą liczbą zespoloną, której wartość można dowolnie wybrać, ustawimy ją na $-0.8 + 0.156i$, gdyż wartość ta pozwala uzyskać bardzo ciekawy efekt. Warto skorzystać z tej możliwości, aby zobaczyć różne inne wersje zbioru Julii.

W prezentowanym programie obliczamy 200 iteracji funkcji. Po każdym powtórzeniu sprawdzamy, czy wartość bezwzględna wyniku nie przekracza pewnej ustalonej wartości (tu progi ustawiliśmy na 1000). Jeśli tak, to przyjmujemy, że równanie dąży do nieskończoności, a więc zwracamy 0, aby zaznaczyć, że dany punkt **nie** należy do zbioru. W przeciwnym razie, tzn. jeśli po 200 iteracjach wartość nie przekracza 1000, przyjmujemy, że punkt należy do zbioru, i zwracamy 1 do wywołującego, czyli funkcji `kernel()`.

Ponieważ wszystkie obliczenia są wykonywane na liczbach zespolonych, zdefiniowaliśmy ogólną strukturę do ich przechowywania.

```
struct cuComplex {
    float r;
    float i;
    cuComplex( float a, float b ) : r(a), i(b) {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

Struktura ta zawiera dwie składowe reprezentujące liczbę zespoloną. Pierwsza z nich to liczba zmiennoprzecinkowa pojedynczej precyzji o nazwie `r` reprezentująca część rzeczywistą, a druga to liczba zmiennoprzecinkowa pojedynczej precyzji o nazwie `i`, która reprezentuje część urojoną. Dodatkowo w strukturze znajdują się definicje operatorów dodawania i mnożenia liczb zespolonych (jeśli nie masz pojęcia o liczbach zespolonych, podstawowe wiadomości możesz szybko znaleźć w internecie). Ponadto w strukturze znajduje się definicja metody zwracającej wartość bezwzględną liczby zespolonej.

ZBIÓR JULII NA GPU

Implementacja dla GPU tradycyjnie jest bardzo podobna do implementacji dla CPU.

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                             bitmap.image_size() ) );
    dim3 grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(),
                            dev_bitmap,
                            bitmap.image_size(),
                            cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();
    cudaFree( dev_bitmap );
}
```

Mimo że ta wersja funkcji `main()` wygląda na bardziej skomplikowaną od poprzedniej, działa dokładnie tak samo jak tamta. Najpierw przy użyciu standardowej funkcji bibliotecznej tworzymy mapę bitową o wymiarach `DIM x DIM`. Ponieważ obliczenia będą wykonywane na GPU, dodatkowo zadeklarowaliśmy wskaźnik o nazwie `dev_bitmap`, który będzie wskazywał kopię danych na urządzeniu. A do przechowywania tych danych potrzebna jest pamięć alokowana za pomocą funkcji `cudaMalloc()`.

Następnie (podobnie jak w wersji dla CPU) uruchamiamy funkcję `kernel()`, lecz tym razem dodajemy do niej kwalifikator `_global_`, aby zaznaczyć, że ma ona zostać wykonana na GPU. Tak jak poprzednio przekazujemy do niej utworzony wcześniej wskaźnik na miejsce w pamięci, w którym mają być przechowywane dane. Jedyna różnica polega na tym, że teraz dane zamiast w systemie hosta są przechowywane na GPU.

Największa różnica między tymi dwiema implementacjami polega na tym, że w wersji dla GPU określona jest liczba bloków wykonawczych funkcji `kernel()`. Ponieważ obliczenia dla każdego punktu można wykonywać niezależnie od pozostałych, utworzyliśmy po jednej kopii funkcji dla każdego interesującego nas punktu. Wcześniej wspomnieliśmy, że w niektórych przypadkach **wygodniej** jest używać indeksowania dwuwymiarowego. Jednym z nich jest właśnie obliczanie wartości funkcji w dwuwymiarowej dziedzinie, takiej jak płaszczyzna zespolona. W związku z tym poniższy wiersz zawiera definicję dwuwymiarowej siatki bloków:

```
dim3 grid(DIM,DIM);
```

Jeśli martwisz się, że zaczynasz zapominać podstawowe informacje, to pragniemy Cię uspokoić, gdyż `dim3` wcale nie jest standardowym typem języka C. W plikach nagłówkowych systemu wykonawczego CUDA znajdują się definicje kilku typów pomocniczych reprezentujących wielowymiarowe struktury. Typ `dim3` reprezentuje krótką trójwymiarową, jakiej użyjemy do określenia liczby uruchomionych bloków. Ale dlaczego używamy trójwymiarowej wartości, skoro wcześniej bardzo wyraźnie podkreślaliśmy, że **utworzymy siatkę dwuwymiarową**?

Zrobiliśmy to dlatego, że system wykonawczy CUDA oczekuje właśnie typu `dim3`. Mimo że aktualnie trójwymiarowe siatki nie są obsługiwane, system wykonawczy CUDA wymaga zmiennej typu `dim3`, w której ostatni element ma wartość 1. Jeśli do inicjacji tej zmiennej została podana tylko dwie wartości, tak jak w instrukcji `dim3 grid(DIM,DIM)`, system automatycznie wstawi w miejsce trzeciego wymiaru wartość 1, dzięki czemu program będzie działał poprawnie. Możliwe, że w przyszłości NVIDIA doda obsługę także trójwymiarowych siatek, ale na razie musimy grzecznie postępować z API wywoływanego jądra, ponieważ w sporach między API a programistą zawsze API jest góra.

Następnie zmienną `grid` typu `dim3` przekazujemy do systemu wykonawczego CUDA za pomocą poniższego wiersza kodu:

```
kernel<<<grid,1>>>( dev _ bitmap );
```

Ponieważ wyniki działania funkcji `kernel()` są zapisywane w pamięci urządzenia, trzeba je stamtąd skopiować do hosta. Jak już wiemy, służy do tego funkcja `cudaMemcpy()` z ostatnim argumentem wywołania `cudaMemcpyDeviceToHost`.

```
HANDLE _ERROR( cudaMemcpy( bitmap.get_ptr(),
                           dev_bitmap,
                           bitmap.image_size(),
                           cudaMemcpyDeviceToHost ) );
```

Kolejna różnica między dwiema prezentowanymi wersjami dotyczy implementacji funkcji `kernel()`:

```
__global__ void kernel( unsigned char *ptr ) {
    //Odwzorowanie z blockIdx na współrzędne piksela
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * blockDim.x;

    //Obliczenie wartości dla tego punktu
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

Po pierwsze, aby funkcja `kernel()` mogła być wywoływana z hosta, a wykonywana na urządzeniu, musi zostać zadeklarowana jako `__global__`. W odróżnieniu od wersji dla CPU nie potrzebujemy zagnieżdżonych pętli `for()` do generowania indeksów pikseli przekazywanych do funkcji `julia()`. Podobnie jak było w przypadku dodawania wektorów, system wykonawczy CUDA generuje je za nas w zmiennej `blockIdx`. Możemy skorzystać z tej możliwości dlatego, że wymiary siatki bloków ustawiliśmy tak samo jak wymiary obrazu, dzięki czemu dla każdej parы liczb całkowitych (x,y) z przedziału od $(0,0)$ do $(DIM-1, DIM-1)$ otrzymujemy jeden blok.

Kolejna informacja, jakiej potrzebujemy, to pozycja w liniowym buforze wyjściowym `ptr`. Obliczana jest ona przy użyciu innej standardowej zmiennej o nazwie `gridDim`. Jej wartość jest stała we wszystkich blokach i reprezentuje wymiary siatki. W tym przypadku będzie to zawsze wartość (DIM, DIM) . Zatem mnożąc indeks wiersza przez szerokość siatki i dodając indeks kolumny, otrzymamy indeks w `ptr`, należący do przedziału wartości od 0 do $(DIM \times DIM - 1)$.

```
int offset = x + y * gridDim.x;
```

Na koniec przeanalizujemy kod decydujący o tym, czy dany punkt należy do zbioru Julii. Jak zwykle wygląda on bardzo podobnie jak implementacja dla CPU.

```
device __ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);
    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);
    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }
    return 1;
}
```

W kodzie tym znajduje się definicja struktury `cuComplex`, która służy do reprezentacji liczb zespolonych w postaci dwóch liczb zmiennoprzecinkowych pojedynczej precyzji. Ponadto struktura ta zawiera definicje operatorów dodawania i mnożenia oraz funkcję zwracającą wartość bezwzględną liczb zespolonych.

```
struct cuComplex {
    float r;
    float i;
    cuComplex( float a, float b ) : r(a), i(b) {}

    device __ float magnitude2( void ) {
        return r * r + i * i;
    }

    device __ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }

    device __ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    };
};
```

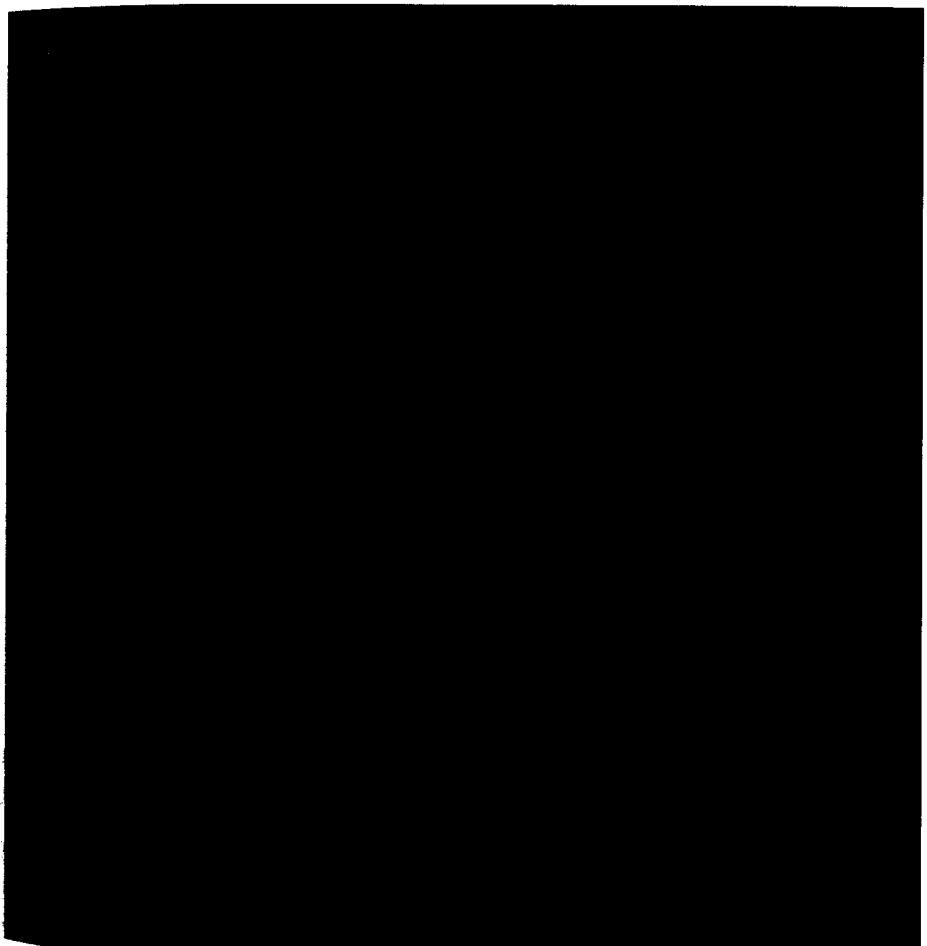
Zwróć uwagę, że w wersji CUDA C programu używane są takie same konstrukcje językowe jak w wersji dla CPU. Jedyną różnicą jest użycie kwalifikatora `_device_` oznaczającego, że dany fragment kodu ma zostać wykonany na GPU. Należy pamiętać, że funkcje zadeklarowane jako `_device_` można wywoływać tylko z innych funkcji tego samego typu lub typu `_global_`.

Poniżej znajduje się w całości kod źródłowy opisanego programu.

```
#include "../common/book.h"
#include "../common/cpu_bitmap.h"
#define DIM 1000
struct cuComplex {
    float r;
    float i;
    cuComplex( float a, float b ) : r(a), i(b) {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);
    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);
    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }
    return 1;
}
__global__ void kernel( unsigned char *ptr ) {
    // Odwzorowanie z blockIdx na położenie piksela
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;
    // Obliczenie wartości dla tego punktu
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                             bitmap.image_size() ) );
    dim3 grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                            bitmap.image_size(),
                            cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();
    HANDLE_ERROR( cudaFree( dev_bitmap ) );
}
```

Gdy uruchomisz ten program, zobaczysz wizualizację zbioru Julii. Jako dowód, że podrozdział ten słusznie ma w tytule słowo „zabawny”, na rysunku 4.2 pokazany jest zrzut ekranu z tej aplikacji.



Rysunek 4.2. Zrzut ekranu z wersji GPU programu

4.3. Podsumowanie

Gratulacje! Potrafisz już pisać, kompilować i uruchamiać programy równoległe na procesorze GPU. Koniecznie pochwal się znajomym. Jeśli nadal trwają oni w błędnych przekonaniach, że programowanie GPU to egzotyczna i trudna do opanowania sztuka, to na pewno zrobisz na nich piorunujące wrażenie. Jak udało Ci się tego dokonać, będzie naszym małym sekretem. A jeśli są to ludzie, którym można bezpiecznie powierzyć tajemnice, powiedz im, żeby też kupili sobie tę książkę.

W rozdziale tym pokazaliśmy, jak zmusić system wykonawczy CUDA do jednoczesnego wykonywania wielu kopii jednego programu w tzw. **blokach**. Zbiór bloków uruchamianych na GPU nazwaliśmy **siatką**. Zbiory bloków mogą być jedno- lub dwuwymiarowe. Korzystając ze zmiennej `blockIdx`, można sprawdzić w każdej kopii funkcji jądra, który blok ją wykonuje. Analogicznie dzięki wbudowanej zmiennej `gridDim` można sprawdzić rozmiar siatki. Obie te zmienne posłużyły nam w programie do obliczenia indeksu danych do przetworzenia dla każdego z bloków.

Rozdział 5

Wątki

Napisaliśmy już pierwszy program w języku CUDA C i wiemy, jak pisać programy równolegle wykonywane na GPU. To świetnie jak na początek! Jednak jednym z najważniejszych elementów programowania równoległego jest sposób współpracy poszczególnych jednostek wykonawczych nad rozwiązaniem zadanego problemu. Rzadko się zdarza, aby każdy procesor mógł wykonać swoje zadanie i zakończyć działanie, nie interesując się kompletnie tym, co robią pozostałe procesory. Nawet wykonywanie średnio skomplikowanych algorytmów wymaga współpracy i komunikacji między równoległymi partiami kodu. Jak do tej pory nie napisaliśmy jeszcze nic o tym, jak można to osiągnąć. Ale oczywiście da się to zrobić i o tym właśnie jest ten rozdział.

5.1. Streszczenie rozdziału

W tym rozdziale:

- Dowiesz się, czym są **wątki** w języku CUDA C.
- Poznasz sposoby komunikacji między wątkami.
- Nauczysz się synchronizować równoległe wykonywanie wielu wątków.

5.2. Dzielenie równoległych bloków

W poprzednim rozdziale pokazaliśmy, jak wykonywać na GPU programy równolegle. Technika ta opiera się na podaniu systemowi wykonawczemu CUDA liczby równoległych kopii funkcji jądra, jakie mają zostać uruchomione. Te równoległe kopie jądra nazywają się **blokami**.

Bloki można dzielić na **wątki**. Przypomnijmy, że wywołując równoległe bloki, zmieniliśmy wartość 1 pierwszego argumentu w nawiasach trójkątnych na liczbę bloków, jaką chcieliśmy utworzyć. Na przykład w programie dodającym wektory uruchamialiśmy osobny blok dla każdego elementu wektora o rozmiarze N za pomocą następującego kodu:

```
add<<<N,1>>>( dev_a, dev_b, dev_c );
```

Drugi parametr w tej konstrukcji reprezentuje liczbę wątków, jaką system wykonawczy CUDA ma utworzyć w każdym bloku. Do tej pory ograniczaliśmy się tylko do jednego wątku na blok. Na przykład w poprzednim przykładzie uruchomiliśmy następującą liczbę wątków:

```
N bloków x 1 wątek/blok = N równoległych wątków
```

Z takim samym skutkiem moglibyśmy zatem uruchomić $N/2$ bloków po dwa wątki, albo $N/4$ bloków po cztery wątki itd. Korzystając z tych nowych wiadomości, jeszcze raz napiszemy implementację programu sumującego wektory.

5.2.1. SUMOWANIE WEKTORÓW — NOWE SPOJRZENIE

Postaramy się wykonać takie samo zadanie jak poprzednio, tzn. pobrać zawartość dwóch wektorów i zapisać ich sumę w trzecim wektorze. Jednak tym razem zamiast bloków użyjemy wątków.

Zastanawiasz się zapewne, jakie zalety mają wątki w porównaniu z blokami. Na razie nie mają żadnych, o których warto by było wspominać. Ale równolegle wątki w bloku mają pewne możliwości, których równolegle bloki są pozbawione. Apelujemy więc o cierpliwość i uważne przestudiowanie wątkowej implementacji programu.

SUMOWANIE WEKTORÓW NA GPU ZA POMOCĄ WĄTKÓW

Na początek objaśnienie dwóch najważniejszych zmian, jakie pojawią się w wątkowej wersji programu. W funkcji jądra zamiast wywoływać N bloków po jednym wątku, jak np.:

```
add<<<N,1>>>( dev_a, dev_b, dev_c );
```

będziemy wywoływać jeden blok zawierający N wątków:

```
add<<<1,N>>>( dev_a, dev_b, dev_c );
```

Druga zmiana będzie dotyczyć sposobu indeksowania danych. Poprzednie dane wejściowe i wyjściowe były indeksowane według indeksów bloków.

```
int tid = blockIdx.x;
```

Pewnie się nie zdziwisz, że skoro teraz jest tylko jeden blok, to indeksowanie będzie się odbywało według indeksów wątków.

```
int tid = threadIdx.x;
```

Zamiana wersji blokowej na wątkową wymaga zastosowania tylko dwóch modyfikacji. Poniżej znajduje się cały kod programu z wyróżnieniem zmian za pomocą pogrubienia:

```
#include "../common/book.h"
#define N 10
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    // Alokacja pamięci na GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
    // Zapelnienie danymi tablic a i b na CPU
    for (int i=0; i<N; i++)
        a[i] = i;
        b[i] = i * i;
    }
    // Skopiowanie tablic a i b na GPU
    HANDLE_ERROR( cudaMemcpy( dev_a,
        a,
        N * sizeof(int),
        cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b,
        b,
        N * sizeof(int),
        cudaMemcpyHostToDevice ) );
    add<<<1,N>>>( dev_a, dev_b, dev_c );
    // Skopiowanie tablicy c z GPU do CPU
    HANDLE_ERROR( cudaMemcpy( c,
        dev_c,
        N * sizeof(int),
        cudaMemcpyDeviceToHost ) );
    // Wyświetlenie wyników
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    // Zwolnienie pamięci alokowanej na GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Proste, prawda? Wkrótce poznasz jedno z ograniczeń stosowania tej metody. Oczywiście później wyjaśnimy, dlaczego w ogóle warto silić się na dzielenie bloków na kolejne elementy równolegle.

SUMOWANIE DŁUGICH WEKTORÓW PRZY UŻYCIU GPU

W poprzednim rozdziale napisaliśmy, że maksymalna liczba bloków w jednym uruchomieniu wynosi 65535 i że jest to ograniczenie sprzętowe. Liczba wątków na blok także jest ograniczona, a jej wartość można znaleźć w polu `maxThreadsPerBlock` struktury właściwości urządzenia, której opis znajduje się w rozdziale 3. Wiele procesorów graficznych ma tę wartość ustawioną na 512. Jak w takim razie skorzystać z tej metody, jeśli chce się zsumować dwa wektory zawierające więcej niż 512 elementów? Trzeba użyć kombinacji wątków i bloków.

Tak jak poprzednio, potrzebne będą dwie zmiany: w sposobie obliczania indeksu w jądrze oraz w sposobie wywoływania samego jądra.

Przy wielu blokach i wątkach indeksowanie będzie przypominać standardową metodę konwersji z dwuwymiarowej przestrzeni indeksowej na liniową.

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

W powyższym przypisaniu użyta została nowa zmienna wbudowana o nazwie `blockDim`. Jej wartość jest stała w każdym bloku i reprezentuje liczbę wątków w każdym wymiarze bloku. Ponieważ nasz blok jest jednowymiarowy, używamy tylko odwołań `blockDim.x`. Może pamiętaś, że podobną wartość zawiera zmienna `gridDim`, która reprezentuje liczbę bloków w każdym wymiarze siatki. Ponadto zmienna `gridDim` jest dwuwymiarowa, a `blockDim` trójwymiarowa. To znaczy, że system wykonawczy CUDA pozwala na tworzenie dwuwymiarowych siatek bloków, w których każdy blok może być trójwymiarową tablicą wątków. To dużo, ale raczej nieczęsto będziesz korzystać ze wszystkich pięciu wymiarów. Ale w razie potrzeby są do dyspozycji.

Indeksowanie danych w tablicy liniowej (czyli jednowymiarowej) za pomocą poprzedniego przypisania jest tak naprawdę bardzo intuicyjne. Jeśli Ci się tak nie wydaje, to może pomoże Ci wyobrażenie sobie zbioru bloków w sposób przestrzenny, podobny do dwuwymiarowej tablicy pikseli, pokazanej na rysunku 5.1.

Jeśli wątki reprezentują kolumny, a bloki wiersze, to dowolny indeks można obliczyć, mnożąc indeks bloku przez liczbę wątków na blok i dodając do tej wartości numer indeksu wątku w tym bloku. Dokładnie taką samą metodę zastosowaliśmy do liniowego indeksowania dwuwymiarowego obrazu w przykładzie zbioru Julii.

```
int offset = x + y * DIM;
```

`DIM` reprezentuje wymiar bloku (wyrażony w wątkach), `y` to indeks bloku, a `x` to indeks wątku w tym bloku. Stąd następujący wzór na indeks: `tid = threadIdx.x + blockIdx.x * blockDim.x;`

Blok 0	Wątek 0	Wątek 1	Wątek 2	Wątek 3
Blok 1	Wątek 0	Wątek 1	Wątek 2	Wątek 3
Blok 2	Wątek 0	Wątek 1	Wątek 2	Wątek 3
Blok 3	Wątek 0	Wątek 1	Wątek 2	Wątek 3

Rysunek 5.1. Dwuwymiarowe rozmieszczenie zbioru bloków i wątków

Druga zmiana dotyczy sposobu wywoływania jądra. Nadal potrzebujemy N równoległych wątków, ale musimy je rozłożyć na kilka bloków, aby nie przekroczyć limitu 512 wątków na blok. Jedynym rozwiązaniem może być ustawienie rozmiaru bloku na jakąś dowolną liczbę wątków. Jeśli przyjmiemy wartość 128, to będziemy potrzebować $N/128$ bloków, aby uruchomić wszystkie N wątków.

Jest tylko jeden haczyk. Działanie $N/128$ to dzielenie całkowitoliczbowe, co oznacza, że jeśli N będzie mieć np. wartość 127, to wynikiem działania $N/128$ będzie zero, a więc nic nie obliczymy, bo system uruchomi zero wątków. Tak naprawdę zbyt małą liczbę wątków będziemy mieć zawsze wtedy, gdy wartość N nie będzie wielokrotnością liczby 128. Niedobrze. Musimy ten wynik dzielenia zaokrąglić.

Istnieje pewna często stosowana sztuczka, która pozwala wykonać takie dzielenie całkowitoliczbowe bez konieczności wywoływania funkcji `ceil()`. Wystarczy zamiast $N/128$ wykonać działanie $(N+127)/128$. Możesz uwierzyć nam na słowo, że wynikiem będzie największa wielokrotność liczby 128 większa od N lub mu równa, albo zastanowić się nad tym przez chwilę i samemu się o tym przekonać.

Ponieważ postanowiliśmy utworzyć 128 wątków na blok, stosujemy następujące wywołanie jądra:

```
add<<< (N+127)/128, 128 >>>( dev _ a, dev _ b, dev _ c );
```

Jeśli wartość N nie będzie wielokrotnością liczby 128, to z powodu sztuczki mającej zapobiec uruchomieniu zbyt małej liczby wątków w szczególnych przypadkach uruchomimy teraz zbyt dużo wątków. Jest jednak proste rozwiązanie tego problemu i już nawet je zastosowaliśmy. Przed użyciem wątku w celu dostępu do tablic należy sprawdzić, czy indeks miejsca, do którego się odnosi, mieści się w przedziale od 0 do N :

```
if (tid < N)
    c[tid] = a[tid] + b[tid];
```

Dzięki temu w momencie gdy indeks będzie sięgał poza tablicę (co ma miejsce zawsze wtedy, gdy wartość N nie jest wielokrotnością liczby 128), program nie wykona obliczeń. Co ważniejsze, nie nastąpi też ani odczyt, ani zapis pamięci poza tablicą.

SUMOWANIE WEKTORÓW O DOWOLNEJ DŁUGOŚCI ZA POMOCĄ GPU

Gdy za pierwszym razem opisywaliśmy uruchamianie równoległych bloków na GPU, nie powiedzieliśmy wszystkiego. Oprócz limitu liczby wątków istnieje jeszcze sprzętowy limit liczby bloków (aczkolwiek znacznie większy niż dla wątków). Jak pisaliśmy, maksymalny rozmiar każdego wymiaru bloku wynosi 65535.

To powoduje, że nasza aktualna implementacja algorytmu sumującego wektory jest obarczona poważną usterką. Jeśli w celu zsumowania wektorów będziemy uruchamiać $N/128$ wątków, to po przekroczeniu liczby $65535 * 128 = 8\,388\,480$ elementów zaczniemy otrzymywać błędy. Może się wydawać, że to bardzo duża liczba, ale jeśli weźmie się pod uwagę możliwości nowoczesnych kart graficznych, które są wyposażone w pamięć o pojemności od 1 do 4 GB, to nie trudno policzyć, iż najlepsze procesory graficzne mogą przechowywać wartości większe nawet o kilka rzędów wielkości.

Na szczęście problem ten można bardzo łatwo rozwiązać. Po pierwsze potrzebna jest zmiana w jądrze:

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

Ten kod wygląda bardzo podobnie do [pierwszej wersji](#) implementacji! Porównajmy go z poniższym algorytmem dla CPU z poprzedniego rozdziału:

```
void add( int *a, int *b, int *c ) {
    int tid = 0; // To jest CPU nr zero, a więc zaczynamy od zera.
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1; // Mamy tylko jeden CPU, a więc zwiększamy o jeden.
    }
}
```

Tu także do iteracji przez dane użyta jest pętla `while()`. Przypomnijmy, że wcześniej twierdziliśmy, iż w systemie wieloprocesorowym lub wielordzeniowym zamiast o 1, indeks tablicowy można zwiększać o liczbę procesorów, jaką chce się użyć. Tę samą metodę zastosujemy teraz w wersji dla GPU.

W implementacji dla GPU rolę procesorów będą pełnić wątki. Mimo że w rzeczywistości GPU może mieć mniej lub więcej jednostek przetwarzających, każdy wątek traktujemy jako logiczną jednostkę działającą równolegle, a rzeczywiste rozplanowanie wykonywania pozostawiamy w gestii sprzętu. Oddzielenie zrównoleglenia od rzeczywistej metody działania sprzętu to jeden z ciężarów, które CUDA C zdejmuję z barków programisty. To chyba dobrze, bo aktualnie pojedynczy układ NVIDIA może zawierać od 8 do 480 jednostek arytmetycznych!

Wiemy już, jakie są podstawy działania tej implementacji. Pozostają nam jeszcze tylko dwie niewiadome: jak określany jest początkowy indeks każdego wątku oraz jak określany jest krok inkrementacji. Ponieważ każdy wątek powinien rozpoczynać działanie od innego indeksu, musimy indeksy wątków i bloków przerobić na indeksy liniowe, podobnie jak w części „[Sumowanie dłuższego wektora przy użyciu GPU](#)”. Każdy wątek zostanie uruchomiony na danych znajdujących się pod indeksem obliczonym za pomocą poniższej instrukcji:

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

Po zakończeniu pracy wątku w bieżącym indeksie zwiększamy jego indeks o liczbę wszystkich wątków w siatce, czyli iloczyn liczby wątków na blok i liczby bloków w siatce, a więc $\text{blockDim.x} * \text{gridDim.x}$. W związku z tym krok inkrementacji jest zaimplementowany następująco:

```
tid += blockDim.x * gridDim.x;
```

Prawie skończono! Pozostało jeszcze tylko poprawić sam sposób uruchamiania. Przypomnijmy, że ta mała dygresja była spowodowana tym, iż wywołanie jądra `add<<<(N+127)/128,128>>>(dev_a, dev_b, dev_c)` było niemożliwe do wykonania, gdy wartość działania $(N + 127)/128$ była większa od 65535. Aby uniemożliwić uruchomienie zbyt dużej liczby bloków, wystarczy zmniejszyć ich liczbę do jakiejś rozsądnej wartości. Ponieważ bardzo lubimy kopowanie i wklejanie, użyjemy 128 bloków po 128 wątków.

```
add<<<128,128>>>( dev_a, dev_b, dev_c );
```

Wartości te można dowolnie zmienić, pod warunkiem że nie przekroczy się opisanych limitów. Później się dowiesz, jaki to ma wpływ na wydajność, ale na razie zaspokoimy się 128 blokami po 128 wątków. Teraz długość sumowanych wektorów jest ograniczona tylko ilością pamięci RAM układu graficznego. Poniżej znajduje się opisany kod w całości:

```
#include "../common/book.h"
#define N (33 * 1024)
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

int main( void ) {
```

```

int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
// Alokacja pamięci na GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
// Zapełnienie tablic a i b na CPU
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i * i;
}
// Skopiowanie tablic do GPU
HANDLE_ERROR( cudaMemcpy( dev_a,
    a,
    N * sizeof(int),
    cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b,
    b,
    N * sizeof(int),
    cudaMemcpyHostToDevice ) );
add<<<128,128>>>( dev_a, dev_b, dev_c );
// Skopiowanie tablicy c z GPU do CPU
HANDLE_ERROR( cudaMemcpy( c,
    dev_c,
    N * sizeof(int),
    cudaMemcpyDeviceToHost ) );
// Sprawdzenie, czy GPU poprawnie wykonał zadanie.
bool success = true;
for (int i=0; i<N; i++) {
    if ((a[i] + b[i]) != c[i]) {
        printf( "Błąd: %d + %d != %d\n", a[i], b[i], c[i] );
        success = false;
    }
}
if (success) printf( "Udało się!\n" );
// Zwolnienie pamięci alokowanej na GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}

```

5.2.2. GENEROWANIE ROZCHODZĄCYCH SIĘ FAL ZA POMOCĄ WĄTKÓW

Tak jak w poprzednim rozdziale, cierpliwość wykazaną podczas studiowania przykładu dodawania wektorów wynagrodzimy Ci ciekawszym programem, na podstawie którego przy okazji zaprezentujemy opisane wcześniej techniki. Ponownie wykorzystamy moc obliczeniową procesora GPU, tak aby wygenerować pewien obraz za pomocą odpowiednich procedur. Żeby było jeszcze ciekawszej, tym razem będzie to nawet animacja. Ale nie musisz się martwić, ponieważ

cały kod dotyczący warstwy graficznej i animacyjnej tego programu umieściliśmy w specjalnych funkcjach pomocniczych, dzięki czemu znajomość tych technik nie będzie Ci do niczego potrzebna.

```

struct DataBlock {
    unsigned char *dev_bitmap;
    CPUAnimBitmap *bitmap;
};

// Zwolnienie pamięci alokowanej na GPU
void cleanup( DataBlock *d ) {
    cudaFree( d->dev_bitmap );
}

int main( void ) {
    DataBlock data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_bitmap,
        bitmap.image_size() ) );
    bitmap.anim_and_exit( (void (*)(void*,int))generate_frame,
        (void (*)(void*))cleanup );
}

```

Większość logiki funkcji `main()` jest ukryta w strukturze pomocniczej `CPUAnimBitmap`. Tym razem znowu stosujemy znany nam już wzorzec: alokujemy pamięć za pomocą funkcji `cudaMalloc()`, wykonujemy na urządzeniu kod przetwarzający dane zapisane w alokowanej pamięci i po zakończeniu pracy zwalniamy tę pamięć za pomocą funkcji `cudaFree()`. Teraz to już dla nas bułka z masłem.

W tym programie nieco skomplikowaliśmy drugi etap, czyli wykonywanie kodu urządzenia, wykorzystującego alokowaną pamięć. Metodzie `anim_and_exit()` przekazujemy wskaźnik na funkcję `generate_frame()`. Funkcja ta będzie wywoływana przez strukturę w celu generowania kolejnych klatek animacji.

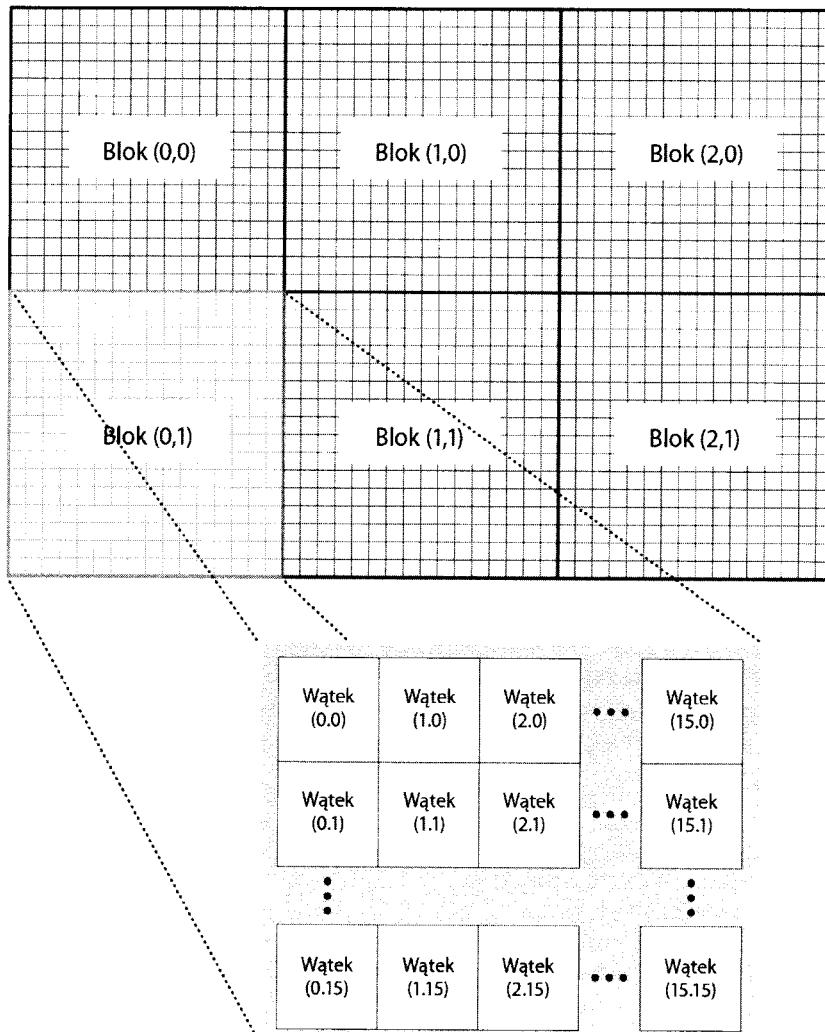
```

void generate_frame( DataBlock *d, int ticks ) {
    dim3 blocks(DIM/16,DIM/16);
    dim3 threads(16,16);
    kernel<<<blocks,threads>>>( d->dev_bitmap, ticks );
    HANDLE_ERROR( cudaMemcpy( d->bitmap->get_ptr(),
        d->dev_bitmap,
        d->bitmap->image_size(),
        cudaMemcpyDeviceToHost ) );
}

```

Mimo że funkcja zawiera tylko cztery wiersze kodu, każdy z nich zawiera ważne elementy dotyczące programowania w języku CUDA C. Najpierw definiujemy dwie dwuwymiarowe zmienne o nazwach `blocks` i `threads`. Pierwsza oczywiście reprezentuje liczbę bloków, jakie mają zostać uruchomione w siatce, a druga — liczbę wątków na blok. Ponieważ będziemy

generować obraz graficzny, korzystamy z indeksowania dwuwymiarowego, dzięki czemu każdy wątek będzie miał niepowtarzalny indeks (x,y), który można łatwo powiązać z pikselem w obrazie wyjściowym. Bloki będą zawierały matryce wątków o wymiarach 16 x 16. Aby przy rozmiarze obrazu $\text{DIM} \times \text{DIM}$ pikseli uzyskać po jednym wątku na piksel, trzeba uruchomić $\text{DIM}/16 \times \text{DIM}/16$ bloków. Na rysunku 5.2 pokazano, jak to powinno wyglądać w przypadku śmiesznie małego obrazu o szerokości 48 i wysokości 32 pikseli.



Osoby, które znają się na programowaniu wielowątkowym CPU, pewnie się zastanawiają, po co uruchamiać aż tyle wątków. Aby na przykład wygenerować animację w wysokiej rozdzielcości 1920 x 1080, trzeba by było uruchomić ich aż ponad dwa miliony. W programowaniu GPU jest to całkiem normalne, natomiast programistom CPU takie coś nawet się nie śniło. Powodem

tego jest fakt, że na CPU zarządzanie i planowanie działania wątków jest wykonywane programowo, co uniemożliwia osiągnięcie tak dużej skali jak w przypadku GPU. Dzięki możliwości utworzenia osobnego wątku dla każdego elementu danych, jakie chce się przetworzyć, programowanie równoległe GPU jest znacznie prostsze niż CPU.

Za deklaracjami zmiennych do przechowywania wymiarów uruchamiamy jądro, które będzie obliczało wartości pikseli.

```
kernel<<< blocks,threads>>>( d->dev _ bitmap, ticks );
```

Funkcji jądra są potrzebne dwie informacje, które przekazujemy jej w postaci parametrów. Po pierwsze potrzebuje wskaźnika na miejsce w pamięci urządzenia, w którym mają być przechowywane piksele wyjściowe. Miejscem tym jest zmienna globalna, której pamięć została alokowana w funkcji main(). Zmienna ta jednak jest „globalna” dla kodu hosta i dlatego musimy ją przekazać jako parametr, aby system wykonawczy CUDA udostępnił ją także dla kodu urządzenia.

Po drugie, aby jądro mogło generować odpowiednie klatki animacji, musi na bieżąco śledzić jej czas. Bieżący czas, ticks, jest przekazywany do funkcji generate_frame() ze struktury CPUAnimBitmap, a więc możemy po prostu przekazać tę wartość do jądra.

Poniżej znajduje się kod źródłowy opisywanej funkcji jądra:

```
_global_ void kernel( unsigned char *ptr, int ticks ) {
    // Odwzorowanie z threadIdx/BlockIdx na położenie pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    // Obliczenie wartości dla danego miejsca
    float fx = x - DIM/2;
    float fy = y - DIM/2;
    float d = sqrtf( fx * fx + fy * fy );
    unsigned char grey = (unsigned char)(128.0f + 127.0f *
        cos(d/10.0f - ticks/7.0f) /
        (d/10.0f + 1.0f));
    ptr[offset*4 + 0] = grey;
    ptr[offset*4 + 1] = grey;
    ptr[offset*4 + 2] = grey;
    ptr[offset*4 + 3] = 255;
}
```

Najważniejsze są trzy pierwsze wiersze kodu tej funkcji:

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;
```

W kodzie tym każdy wątek otrzymuje swój indeks w bloku oraz indeks swojego bloku w siatce i z danych tych generuje swój niepowtarzalny indeks (x,y) na powierzchni obrazu. Gdy na przykład w bloku o indeksie (12, 8) rozpoczyna działanie wątek o indeksie (3, 5), wie on, że po jego lewej stronie znajduje się 12 całych bloków, a nad nim — 8 całych bloków. Wewnątrz bloku wątek o indeksie (3, 5) ma trzy wątki po lewej stronie i pięć wątków nad sobą. Ponieważ w bloku jest 16 wątków, oznacza to, że omawiany wątek ma:

$$3 \text{ wątki} + 12 \text{ bloków} * 16 \text{ wątków/blok} = 195 \text{ wątków po swojej lewej stronie}$$

$$5 \text{ wątków} + 8 \text{ bloków} * 16 \text{ wątków/blok} = 128 \text{ wątków nad sobą}$$

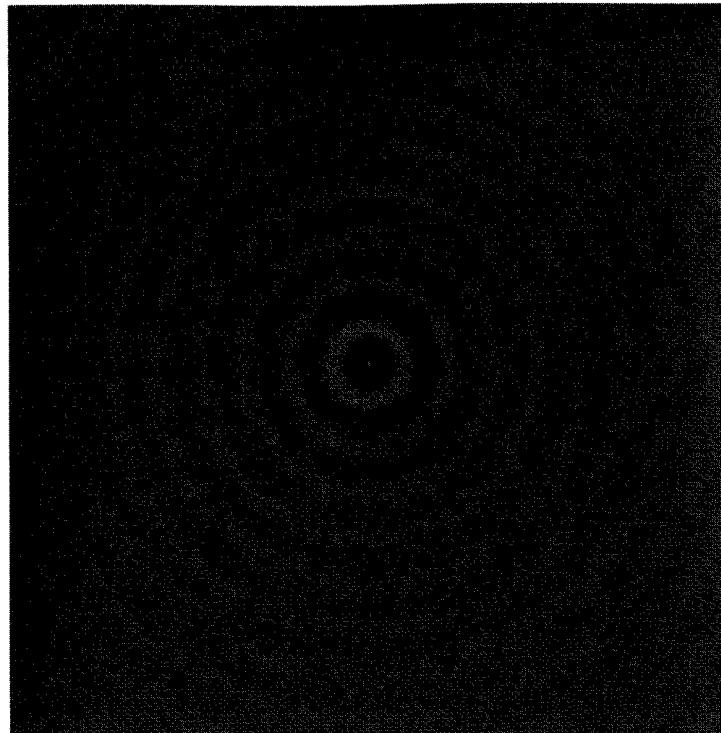
Dokładnie takie same obliczenia x i y są wykonywane w dwóch pierwszych wierszach powyższego kodu i w ten właśnie sposób dokonujemy translacji indeksów wątków i bloków na współrzędne obrazu. Następnie po prostu te wartości x i y zamieniamy na liniowe, tak abytrzymać pozycję w buforze wyjściowym. Dokładnie to samo zrobiliśmy w programach obliczających sumę wektorów w tym rozdziale.

```
int offset = x + y * blockDim.x * gridDim.x;
```

Ponieważ znamy współrzędne (x,y) piksela, który powinien zostać obliczony przez wątek, oraz wiemy, kiedy powinno nastąpić wykonanie tych obliczeń, możemy obliczyć dowolną funkcję $z(x, y, t)$ i zapisać otrzymaną wartość w buforze wyjściowym. W tym przypadku jest to funkcja generująca zmienną w czasie sinusoidalną „falę”.

```
float fx = x - DIM/2;
float fy = y - DIM/2;
float d = sqrtf( fx * fx + fy * fy );
unsigned char grey = (unsigned char)(128.0f + 127.0f *
    cos(d/10.0f - ticks/7.0f) /
    (d/10.0f + 1.0f));
```

Nie ma sensu poświęcać zbyt dużo czasu na zastanawianie się, jak działa funkcja `grey`. Jest to dwuwymiarowa funkcja czasu, która w animacji daje ciekawy efekt rozchodzących się fal. Na rysunku 5.3 pokazany jest zrzut ekranu jednej klatki.



Rysunek 5.3. Klatka z animacji generowanej na GPU

Cieszymy się, że pytasz. Zmienne zapisane w pamięci wspólnej są przez język CUDA C traktowane inaczej niż zwykłe zmienne. Dla każdego bloku tworzona jest osobna kopia takiej zmiennej. Wszystkie wątki w danym bloku mają dostęp do kopii przeznaczonej dla tego bloku, ale nie „widzą” i nie mogą modyfikować kopii innych bloków. Jest to zatem doskonały sposób pozwalający na komunikację i współpracę wątków w obrębie jednego bloku. Co więcej, buforey pamięci wspólnej są zapisane fizycznie na GPU, a nie w oddzielnej pamięci DRAM. Dzięki temu opóźnienia dostępu do nich są znacznie krótsze niż do typowych buforów, co sprawia, że pamięć ta znakomicie nadaje się do wykorzystania jako blokowa, programowalna pamięć podręczna.

Wiadomość, że można nawiązać kontakt między wątkami, powinna Cię bardzo uścisnąć. My na przykład bardzo się radujemy z tego powodu. Niestety w życiu nie ma nic za darmo, nawet komunikacji między wątkami. Jeśli chcemy przesyłać informacje między wątkami, to musimy także zatroszczyć się o ich synchronizację. Jeśli na przykład wątek A zapisuje w pamięci wspólnej jakąś wartość, której ma użyć do czegoś innego wątek B, to wątek B może rozpoczęć pracę dopiero wtedy, gdy się dowie, że wątek A zakończył już operację zapisu. Bez takiej synchronizacji mielibyśmy ciągły wyścig wątków, którego wynik zależałby od niedających się przewidzieć szczegółów budowy sprzętu.

Obaczmy przykładowy kod z użyciem opisywanych mechanizmów.

5.3. Pamięć wspólna i synchronizacja

Do tej pory jedynym powodem, dla którego dzieliliśmy bloki na wątki, była chęć ominięcia ograniczenia liczby równolegle uruchomionych bloków. Nie jest to jednak technika najwyższych lotów, ponieważ bez problemu to samo mógłby niezauważalnie robić system wykonawczy CUDA. Są jednak jeszcze inne powody, dla których warto czasami dzielić bloki na wątki.

CUDA C udostępnia obszar pamięci nazywany **pamięcią wspólną** (ang. *shared memory*). Z obszarem tym związane jest kolejne obok kwalifikatorów `_device_` i `_global_` rozszerzenie standardu C. Jest to kwalifikator `_shared_`, który powoduje zapisanie zmiennej w pamięci wspólnej. Ale po co?

5.3.1. ILOCZYN SKALARNY

Gratulacje! Przeszliśmy już przez etap sumowania wektorów i weszliśmy na wyższy poziom obliczania iloczynu skalarnego wektorów. Dla osób niezorientowanych w tej dziedzinie matematyki (to przecież już tyle lat) prezentujemy krótkie przypomnienie, czym jest iloczyn skalarny wektorów. Obliczenia są dwuetapowe. Najpierw mnoży się odpowiadające sobie elementy dwóch wektorów wejściowych, podobnie jak wcześniej się je dodawało. Następnie (w odróżnieniu od dodawania) zamiast zapisywać poszczególne wyniki w trzecim wektorze, sumuje się wszystkie iloczyny w celu otrzymania jednej wartości skalarnej.

Na listingu 5.1 przedstawiony jest ogólny przykład dodawania dwóch wektorów zawierających po cztery elementy.

Listing 5.1.

$$(x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

Pewnie zaczyna Ci już świtać w głowie, jakiego algorytmu mamy zamiar użyć. Pierwszą część obliczeń możemy wykonać dokładnie tak samo jak przy dodawaniu. Każdy wątek obliczy iloczyn określonej pary elementów i przejdzie do następnej. Ponieważ wynikiem musi być suma wszystkich takich iloczynów, każdy wątek musi przechowywać bieżącą sumę wszystkich obliczonych przez siebie iloczynów. Podobnie jak było w przypadku dodawania, krok inkrementacji w każdym wątku powinien być równy liczbie wszystkich wątków. Dzięki temu jest pewne, że żadna para nie zostanie pominięta ani że któraś z nich zostanie pomnożona dwa razy. Oto pierwsza część opisywanego algorytmu:

```
#include "../common/book.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    // Ustawianie wartości w pamięci podręcznej
    cache[cacheIndex] = temp;
}
```

Zdefiniowaliśmy bufor pamięci wspólnej o nazwie cache. Będą w nim zapisywane sumy bieżące wszystkich wątków. Wkrótce wyjaśnimy, dlaczego wybraliśmy takie rozwiązanie, a na razie po prostu prześledzimy sposób jego implementacji. Deklaracja zmiennej w pamięci wspólnej

jest banalnie prosta i nie różni się niczym od deklaracji zmiennej statycznej albo ulotnej w standardowym języku C:

```
_shared_ float cache[threadsPerBlock];
```

Ponieważ rozmiar tablicy ustawiłyśmy na threadsPerBlock (liczba wątków przypadających na blok), to zmieszcza się w niej wyniki pośrednie wszystkich wątków. Przypomnijmy, że kiedy alokowaliśmy pamięć globalnie, rezerwowałyśmy jej ilość wystarczającą dla każdego wątku wykonującego jądro, czyli threadsPerBlock razy liczba bloków. Ponieważ w przypadku zmiennych zapisanych w pamięci wspólnej kompilator tworzy ich kopię dla każdego bloku, wystarczy, że alokujemy tylko ilość pamięci wystarczającą dla każdego wątku w bloku.

Po alokacji pamięci wspólnej obliczamy indeksy danych w taki sam sposób jak poprzednio:

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIndex = threadIdx.x;
```

Algorytm obliczania wartości zmiennej tid powinien już wyglądać znajomo. Z połączenia indeksów blokowych i wątkowych tworzy on globalny indeks do tablic wejściowych. W pamięci wspólnej indeks ten jest po prostu indeksem wątku. Nie musimy do tych obliczeń angażować indeksów bloków, ponieważ każdy blok ma własną prywatną kopię tej pamięci.

Na zakończenie opróżniamy bufor, aby później móc zsumować całą tablicę bez obawy, że gdzieś mogłyby być zapisane jeszcze jakieś potrzebne dane.

```
// Ustawianie wartości pamięci podręcznej
cache[cacheIndex] = temp;
```

Istnieje ryzyko, że jeśli rozmiar wektorów wejściowych będzie różny od wielokrotności liczby wątków na blok, to nie każdy element zostanie użyty w obliczeniach. Wówczas ostatni blok będzie zawierał bezczynne wątki, które nie będą zapisywać żadnych wartości.

Każdy wątek oblicza sumę iloczynów par elementów tablic a i b. Po dojściu do końca tablicy zapisuje on swoją cząstkową sumę we wspólnym buforze:

```
float temp = 0;
while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
}
```

```
// Ustawianie wartości pamięci podręcznej
cache[cacheIndex] = temp;
```

Teraz nadszedł czas na zsumowanie wszystkich wartości cząstkowych, które są zapisane w pamięci podręcznej. Do tego potrzebne są wątki, które je stamtąd odczytają. Jest to jednak, jak już wiemy, operacja ryzykowna. Przed próbą odczytu danych z bufora musimy mieć absolutną

pewność, że zapis wszystkich wartości w reprezentującej go tablicy cache[] już się zakończył. Oczywiście da się to zrobić:

```
// Synchronizacja wątków w bloku
__syncthreads();
```

Najpierw muszą zakończyć się wszystkie operacje wątków znajdujące się w kodzie przed wywołaniem funkcji __syncthreads(), a dopiero potem mogą być wykonywane dalsze instrukcje wątkowe. Tego właśnie było nam potrzeba! Wiemy teraz, że kiedy pierwszy wątek wykona jakąś instrukcję znajdująca się za wywołaniem funkcji __syncthreads(), oznacza to, że pozostałe wątki również wykonały wszystkie instrukcje do tej funkcji.

Po zagwarantowaniu poprawnego zapełnienia pamięci podręcznej danymi można zsumować zapisane w niej wartości. Ogólnie proces polegający na pobraniu danych z tablicy i wykonaniu na nich obliczeń, w wyniku których powstaje mniejsza tablica, nazywamy **redukcją**. Tego typu działania są często wykonywane w programowaniu równoległym i stąd potrzeba nadania im jakieś nazwy.

Najprostszym sposobem na wykonanie tego zadania jest utworzenie jednego wątku, który przejrzalby iteracyjnie zawartość pamięci wspólnej i obliczył sumę elementów. Czasowo złożoność obliczeniowa tej operacji byłaby równa długości tablicy. Ponieważ jednak mamy do dyspozycji setki wątków, redukcję tę możemy wykonać równolegle i skrócić czas wykonywania do wartości proporcjonalnej do logarytmu z długości tablicy. Implementacja tego algorytmu będzie się początkowo wydawała zawiła, ale poniżej zamieściliśmy jej szczegółowy opis.

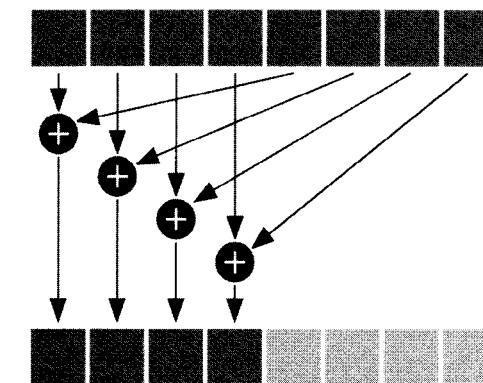
Ogólnie chodzi o to, aby każdy użyty wątek sumował dwie wartości tablicy cache[] i obliczony wynik zapisywał z powrotem w tej tablicy. Ponieważ każdy wątek z dwóch elementów robi jeden, po zakończeniu pierwszego etapu tablica będzie o połowę krótsza. W drugim kroku robimy to samo, aby zredukować tablicę o kolejną połowę. Powtarzamy te czynności $\log_2(\text{threadsPerBlock})$ razy, aż będą uzyskane sumy wszystkich elementów tablicy. Ponieważ w przykładzie używamy 256 wątków na blok, do zsumowania 256 elementów tablicy cache[] będzie potrzeba ośmiu powtórzeń.

Poniżej znajduje się odpowiedni kod źródłowy:

```
// Aby redukcja była możliwa, wartość threadsPerBlock musi być potęgą 2.
// Wymusza to poniższy kod.
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
```

W pierwszym kroku wartość zmiennej *i* wynosi połowę liczby wątków na blok (threadsPerBlock). Ponieważ pracować powinny tylko wątki o indeksach mniejszych od tej wartości, sumowanie elementów wykonujemy jedynie pod warunkiem, że indeks wątku jest mniejszy od *i*. O ochronę operacji dodawania dba blok instrukcji `if (cacheIndex < i)`. Każdy wątek pobierze wartość znajdująca się w tablicy cache[] pod odpowiadającym mu indeksem, doda ją do wartości znajdującej się o *i* dalej i zapisze otrzymaną sumę z powrotem w cache[].

Przypuśćmy na przykład, że w tablicy cache[] jest osiem elementów, a więc zmienna *i* ma wartość 4. Jeden z kroków redukcji tej tablicy wyglądałby tak, jak pokazano na rysunku 5.4.



Rysunek 5.4. Jeden z kroków operacji redukcji

Po zakończeniu każdego kroku mamy ten sam problem z synchronizacją co podczas obliczania iloczynów par elementów. Zanim spróbujemy odczytać wartości zapisane z powrotem w tablicy cache[], musimy mieć pewność, że wszystkie wątki zapisujące w niej dane zakończyły już działanie. Spełnienie tego warunku gwarantuje wywołanie funkcji __syncthreads() za przypisaniem.

Po wykonaniu tej pętli `while()` każdy blok zawiera tylko jedną liczbę. Liczba ta, reprezentująca sumę iloczynów par obliczonych przez wątki danego bloku, zapisana jest w pierwszym elemencie tablicy cache[]. Zapisujemy ją następnie w pamięci globalnej i kończymy funkcję jądra:

```
if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
```

Dlaczego tę operację zapisu w pamięci globalnej wykonujemy tylko dla wątku o indeksie `cacheIndex == 0`? Ponieważ jest tu tylko jedna liczba. Niewątpliwie działanie to mogłoby wykonać każdy wątek, ale jak na zapisanie jednej liczby spowodowałoby to niepotrzebne obciążenie pamięci. Żeby nie komplikować, wybraliśmy wątek o indeksie 0, chociaż można by było użyć dowolnego innego. I w końcu, ponieważ w globalnej tablicy `c[]` każdy blok zapisze dokładnie jedną wartość, możemy zastosować indeksowanie poprzez zmienną `blockIdx`.

Otrzymaliśmy tablicę `c[]`, której każdy element zawiera sumę elementów obliczoną przez jeden równoległy blok. Ostatnią czynnością, jaka nam została, aby obliczyć iloczyn skalarny,

jest zsumowanie elementów tej tablicy. Mimo że jeszcze tego nie skończyliśmy, to w tym miejscu kończymy funkcję jądra i wracamy do hosta. Dlaczego to robimy, skoro jeszcze nie zakończyliśmy obliczeń?

Wcześniej napisaliśmy, że działania typu obliczanie iloczynu skalarnego określa się mianem **redukcji**. Nazwa ta odzwierciedla naturę tych operacji, które ogólnie rzecz biorąc, zawsze zwracają mniejszą liczbę elementów, niż było ich na wejściu. Eksperymenty wykazały, że późne urządzenia przetwarzania równoległego, takie jak GPU, podczas wykonywania ostatniego etapu redukcji marnują zasoby, ponieważ zbiór danych jest po prostu zbyt mały. Trudno jest zająć 480 jednostek arytmetycznych w celu zsumowania 32 liczb!

Dlatego właśnie przekazujemy sterowanie z powrotem do hosta, tak aby ostatni etap obliczeń, czyli sumowanie elementów tablicy `c[]`, przeprowadzić już na CPU. W większym programie GPU można by było w tym czasie wykorzystać do wykonywania jakichś innych skomplikowanych obliczeń. Natomiast w tym przykładzie dajemy mu już wolne.

W prezentacji tego przykładu złamaliśmy tradycję, ponieważ zamiast pokazać funkcję główną, od razu przeszliśmy do jądra. Nie powinno być jednak żadnych problemów ze zrozumieniem, jak ta funkcja działa, ponieważ jest ona bardzo podobna do wcześniejszych przykładów.

```
const int blocksPerGrid =  
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );  
  
int main( void ) {  
    float *a, *b, c, *partial_c;  
    float *dev_a, *dev_b, *dev_partial_c;  
    // Alokacja pamięci na CPU  
    a = (float*)malloc( N*sizeof(float) );  
    b = (float*)malloc( N*sizeof(float) );  
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );  
    // Alokacja pamięci na GPU  
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,  
        N*sizeof(float) ) );  
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,  
        N*sizeof(float) ) );  
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,  
        blocksPerGrid*sizeof(float) ) );  
    // Zapelnienie danymi pamięci hosta  
    for (int i=0; i<N; i++) {  
        a[i] = i;  
        b[i] = i*2;  
    }  
    // Skopiowanie tablic a i b na GPU  
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),  
        cudaMemcpyHostToDevice ) );  
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),  
        cudaMemcpyHostToDevice ) );  
    dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a,  
        dev_b,  
        dev_partial_c );
```

Żeby nie zanudzić Czytelnika na śmierć, poniżej prezentujemy opis najważniejszych części tego kodu w punktach:

1. Alokacja pamięci na hoście i urządzeniu do zapisu tablic wejściowych i wyjściowych.
2. Zapełnienie tablic `a[]` i `b[]` danymi oraz skopiowanie ich na GPU za pomocą funkcji `cudaMemcpy()`.
3. Wywołanie funkcji jądra obliczającej iloczyn skalarny za pomocą z góry ustalonej liczby wątków na blok i bloków w siatce.

Większość tego kodu jest już nam znana, ale warto przyjrzeć się bliżej sposobowi obliczania liczby bloków do uruchomienia. Napisaliśmy, że obliczanie iloczynu skalarnego to działanie polegające na redukcji oraz że każdy uruchomiony blok obliczy jedną część ostatecznego wyniku. Długość listy sum częściowych powinna być na tyle mała, aby bez trudu poradził sobie z nią procesor CPU, a zarazem na tyle duża, aby trzeba było uruchomić tyle bloków, żeby nawet najszybszy GPU miał co robić. Zdecydowaliśmy się na uruchomienie 32 bloków, ale wybierając inne wartości, możesz zanotować zauważalny wzrost lub spadek wydajności. Wszystko zależy od szybkości działania CPU i GPU.

Co jednak zrobić, jeśli lista jest bardzo krótka i liczba 32 bloków po 256 wątków to stanowiąco za dużo? Jeśli mamy N elementów danych, to do obliczenia iloczynu skalarnego potrzebujemy N wątków. Zatem w tym przypadku potrzebujemy najmniejszej wielokrotności wartości `threadsPerBlock` większej od N lub mu równej. Mieliśmy już coś takiego przy obliczaniu sumy wektorów. W tym przykładzie najmniejszą wielokrotność wartości `threadsPerBlock`, większą od N lub równą N , obliczamy za pomocą instrukcji $(N+(threadsPerBlock-1)) / threadsPerBlock$. Przypuszczalnie wiesz, że jest to często używana sztuczka stosowana przy dzieleniu liczb całkowitych, a więc warto sobie ją przyswoić, nawet jeśli języka CUDA C będziesz używać tylko sporadycznie.

Zatem liczba bloków powinna być równa mniejszej z wartości i wynosić $32 \cdot (N+(threadsPerBlock-1)) / threadsPerBlock$.

```
const int blocksPerGrid =  
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
```

Teraz kod źródłowy funkcji `main()` powinien być już jasny. Gdy funkcja jądra zakończy działanie, pozostaje jeszcze zsumowanie elementów zwróconej tablicy. Ale podobnie jak mieliśmy skopiować dane do GPU w celu ich przetworzenia, teraz musimy je skopiować z powrotem do CPU w celu dokończenia pracy. Gdy zatem jądro zakończy działanie, tablicę sum częściowych kopujemy z powrotem do hosta i kończymy sumowanie za pomocą CPU.

```
// Skopiowanie tablicy c z GPU do CPU  
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,  
    blocksPerGrid*sizeof(float),  
    cudaMemcpyDeviceToHost ) );  
  
// Dokończenie pracy za pomocą CPU  
c = 0;
```

```

for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}

```

Na koniec sprawdzamy wynik i zwalniamy pamięć alokowaną na GPU i CPU. Sprawdzanie wyników mamy ułatwione, ponieważ dane wejściowe tablic były przewidywalne. Przypomnijmy, że do tablicy a[] wstawiliśmy liczby całkowite z przedziału od 0 do N-1, a do b[] — $2^*a[]$.

```

// Zapelnienie tablic w pamieci hosta danymi
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}

```

Otrzymany iloczyn skalarny powinien być liczbą dwukrotnie większą od sumy kwadratów liczb całkowitych ze zbioru od 0 do N-1. Dla osób, które kochają matematykę dyskretną (a jak można jej nie kochać?), świetną rozrywką może być wyprowadzenie jawnego wzoru na wykonanie tego zadania. Osobom mniej cierpliwym podajemy poniżej gotowe rozwiążanie oraz pozostałą część funkcji main():

```

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Czy wartość GPU %.6g = %.6g?\n", c,
        2 * sum_squares( (float)(N - 1) ) );
// Zwolnienie pamieci GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_partial_c );
// Zwolnienie pamieci CPU
free( a );
free( b );
free( partial_c );
}

```

Jeśli nasze objaśnienia kodu tylko Ci przeszkadzały, poniżej przedstawiamy opisywany kod źródłowy w całości (wielokrotnie bez objaśnień):

```

#include "../common/book.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
}

```

```

float temp = 0;
while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
}
// Ustawienie wartosci pamieci podrecznej
cache[cacheIndex] = temp;

// Synchronizacja wątków w tym bloku
syncthreads();
// Aby redukcja była możliwa, wartość threadsPerBlock musi być potęgą 2.
// Wymusza to poniższy kod.
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    syncthreads();
    i /= 2;
}
if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
}

int main( void ) {
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;
// Alokacja pamieci na CPU
a = (float*)malloc( N*sizeof(float) );
b = (float*)malloc( N*sizeof(float) );
partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
// Alokacja pamieci na GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                           N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                           N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                           blocksPerGrid*sizeof(float) ) );
// Zapelnienie tablic w pamieci hosta danymi
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}
// Skopiowanie tablic a i b do GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                           cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                           cudaMemcpyHostToDevice ) );
dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                             dev_partial_c );
// Skopiowanie tablicy z GPU do CPU

```

```

HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                        blocksPerGrid*sizeof(float),
                        cudaMemcpyDeviceToHost ) );
// Dokonczenie pracy za pomocą CPU
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Czy wartość GPU %.6g = %.6g?\n", c,
        2 * sum_squares( (float)(N - 1) ) );
// Zwolnienie pamięci GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_partial_c );
// Zwolnienie pamięci CPU
free( a );
free( b );
free( partial_c );
}

```

5.3.2. OPTYMALIZACJA (NIEPOPRAWNA) PROGRAMU OBliczAJĄCEGO ILOCZYN SKALARNY

W poprzednim przykładzie drugą funkcję `_syncthreads()` opisaliśmy tylko pobieżnie. Teraz przyjrzymy się jej dokładniej i spróbujemy ją zoptymalizować. Jak zapewne pamiętasz, to drugie wywołanie funkcji `_syncthreads()` było nam potrzebne, ponieważ aktualizujemy zapisaną w pamięci wspólnej zmienną `cache[]`, a chcemy, aby aktualizacje te były widoczne dla wszystkich wątków w następnej iteracji pętli.

```

int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}

```

Zwróć uwagę, że bufor `cache[]` jest aktualizowany wyłącznie wówczas, gdy zmienna `cacheIndex` ma wartość mniejszą od `i`. Ponieważ `cacheIndex` to w istocie `threadIdx.x`, wiadomo, że tylko **niektóre** wątki zmieniają zawartość wspólnej pamięci podręcznej. Ponieważ funkcji `_syncthreads` używamy po to, aby zapewnić wykonanie wszystkich tych modyfikacji przed przejściem dalej, można rozumować, że gdybyśmy czekali tylko na te wątki, które rzeczywiście dokonują zapisu w pamięci wspólnej, moglibyśmy nieco przyspieszyć działanie programu. Dokonujemy tego, przenosząc wywołanie synchronizacyjne do bloku `if()`:

```

int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}

```

Mimo że podjęliśmy śmiałą próbę optymalizacji, to nic nam to nie da, a nawet wręcz przeciwnie, stracimy na tym. Powyższa zmiana w jądrze spowoduje, że GPU przestanie odpowiadać, co zmusi nas do zamknięcia programu. Ale co takiego strasznego jest w tej z pozoru niewinnej zmianie, że wywołuje tak opłakany skutek?

Najłatwiej jest to zrozumieć, wyobrażając sobie, w jaki sposób każdy wątek bloku wykonuje kod źródłowy wiersz po wierszu. Wszystkie wątki wykonują te same instrukcje, ale każdy z nich ma do przetworzenia inne dane. Co zatem się dzieje, gdy instrukcja, którą ma wykonać każdy wątek, znajduje się wewnątrz takiego bloku warunkowego jak `if()`? Skoro jest to blok warunkowy, to oczywiście znaczy, że jego zawartość nie powinna zostać wykonana przez wszystkie wątki, prawda? Przypuśćmy na przykład, że w jądrze znajduje się poniższy fragment kodu, w którym wątki o nieparzystych numerach modyfikują wartość pewnej zmiennej:

```

int myVar = 0;
if( threadIdx.x % 2 )
    myVar = threadIdx.x;

```

W powyższym przykładzie pogrubiony wiersz kodu zostanie wykonany tylko przez wątki nieparzyste, ponieważ parzyste nie spełniają warunku `if(threadIdx.x % 2)`. Oznacza to, że wątki nieparzyste nic nie robią, a parzyste wykonują przedstawioną instrukcję. Sytuację, w której tylko niektóre wątki wykonują jakąś instrukcję, nazywa się **dywergencją wątków** (ang. *thread divergence*). Po prostu w normalnych warunkach niektóre wątki pozostają bezczynne, a inne wykonują instrukcje.

W przypadku funkcji `_syncthreads()` sprawia wygląda jednak o wiele gorzej. Architektura CUDA gwarantuje, że dopóki **wszystkie wątki** nie ukończą wykonywania funkcji `_syncthreads()`, nie może zostać wykonana żadna dalsza instrukcja. A ponieważ w omawianym przypadku **wywołanie** tej funkcji znajduje się również w nieaktywnej gałęzi, przez niektóre wątki nie zostanie ona nigdy wykonana i oczekiwanie, aż to nastąpi, przeciągnie się w nieskończoność.

Tak właśnie się dzieje w powyższym programie po przesunięciu wywołania funkcji `_syncthreads()` do bloku `if()`. Żaden wątek o indeksie `cacheIndex` większym od `i` lub mu równym **nigdy** nie **wykonuje** funkcji `_syncthreads()`. Powoduje to zawieszenie procesora, ponieważ GPU zostaje w takiej sytuacji zmuszony do oczekiwania na zdarzenie, które nigdy nie ma nastąpić:

```

if (cacheIndex < i) {
    cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
}

```

Morał z tego taki, że `__syncthreads()` to bardzo przydatna funkcja pozwalająca zmusić programy wielowątkowe do poprawnego wykonywania obliczeń. Ponieważ jednak łatwo przy tym popełnić błąd, należy jej używać z rozważą.

5.3.3. GENEROWANIE MAPY BITOWEJ ZA POMOCĄ PAMIĘCI WSPÓŁNEJ

Wcześniej pokazane zostały przykładowe programy przygotowujące dane do pracy za pomocą pamięci wspólnej i funkcji `__syncthreads()`. Czasami istnieje pokusa, aby w celu optymalizacji szybkości działania programu pominąć funkcję `__syncthreads()`. Dlatego w tej części rozdziału pokażemy graficzną ilustrację tego, co się dzieje, gdy zaniedba się poprawną synchronizację. Zobaczysz dwie ilustracje. Na jednej z nich będzie przedstawione to, co chcielibyśmytrzymać, a na drugiej to, co uzyskaliśmy, zaniedbując synchronizację przy użyciu funkcji `__syncthreads()` (nie będzie to piękne).

W porównaniu z programem obliczającym zbiór Julii na GPU treść funkcji `main()` pozostaje prawie bez zmian — zmieni się tylko liczba wątków na blok:

```

#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_bitmap.h"
#define DIM 1024
#define PI 3.1415926535897932f
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                            bitmap.image_size() ) );
    dim3 grids(DIM/16,DIM/16);
    dim3 threads(16,16);
    kernel<<<grids,threads>>>( dev_bitmap );
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                            bitmap.image_size(),
                            cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();
    cudaFree( dev_bitmap );
}

```

Tak jak w programie generującym zbiór Julii, każdy wątek oblicza wartość piksela dla jednego punktu wyjściowego. Każdy wątek zaczyna pracę od obliczenia swoich współrzędnych x i y na tworzonym obrazie. Obliczenia te są identyczne z obliczeniami `tid` w programie dodającym wektory, z tym że tym razem są one wykonywane w dwóch wymiarach:

```

_global_ void kernel( unsigned char *ptr ) {
    // Rzutowanie threadIdx/blockIdx na położenie piksela
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
}

```

Ponieważ wyniki obliczeń będą tymczasowo przechowywane w pamięci podręcznej, deklarujemy taki bufor, w którym każdy wątek z bloku 16×16 będzie miał jedno miejsce na swoje dane.

```

__shared__ float shared[16][16];

```

Następnie każdy wątek oblicza wartość, którą później zapisze w buforze:

```

// Obliczenie wartości dla każdego punktu na obrazie
const float period = 128.0f;
shared[threadIdx.x][threadIdx.y] =
    255 * (sinf(x*2.0f*PI/ period) + 1.0f) *
    (sinf(y*2.0f*PI/ period) + 1.0f) / 4.0f;

```

Na koniec zapisujemy te wartości w pikselach, odwracając kolejność x i y :

```

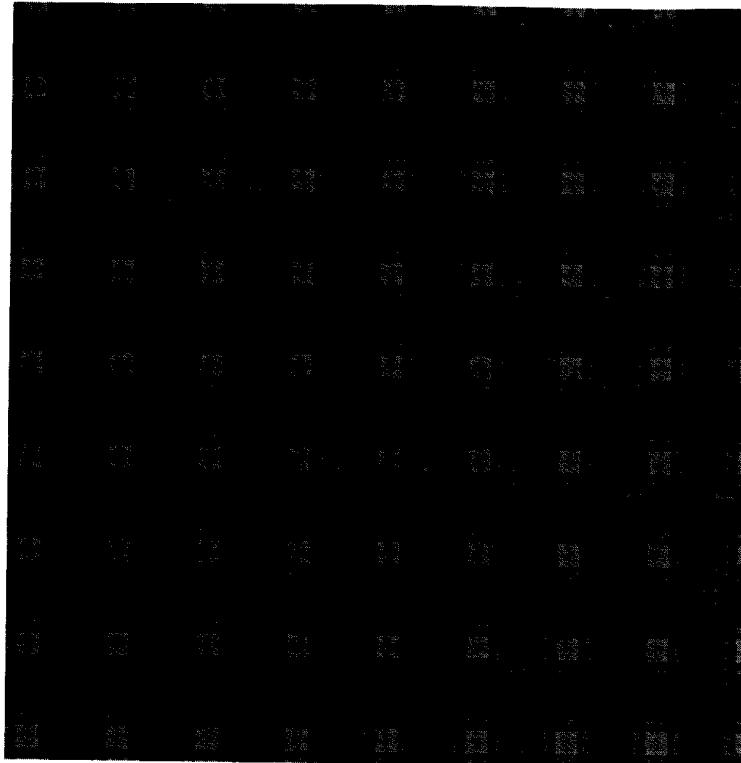
ptr[offset*4 + 0] = 0;
ptr[offset*4 + 1] = shared[15-threadIdx.x][15-threadIdx.y];
ptr[offset*4 + 2] = 0;
ptr[offset*4 + 3] = 255;
}

```

Przyznajemy, że te obliczenia są całkowicie arbitralne. Po prostu chcieliśmy napisać program rysujący siatkę z zielonych kulek. Po skompilowaniu i uruchomieniu tego jądra otrzymaliśmy wynik widoczny na rysunku 5.5.

Dlaczego tak się stało? Można się oczywiście domyślić, że w programie zabrakło synchronizacji. Gdy jakiś wątek obliczoną wartość z tablicy `shared[][]` zapisuje w pikselu, nie ma gwarancji, że wątek dokonujący zapisu tej wartości w `shared[][]` zakończył już działanie. Jedynym sposobem na uniknięcie takiej sytuacji jest użycie funkcji `__syncthreads()`. Wówczas kształty na rysunku przypominałyby zielone kulki.

Powyższa grafika to nic wielkiego, ale programów używa się też do poważniejszych celów i lepiej, żeby poprawnie wykonywały obliczenia.

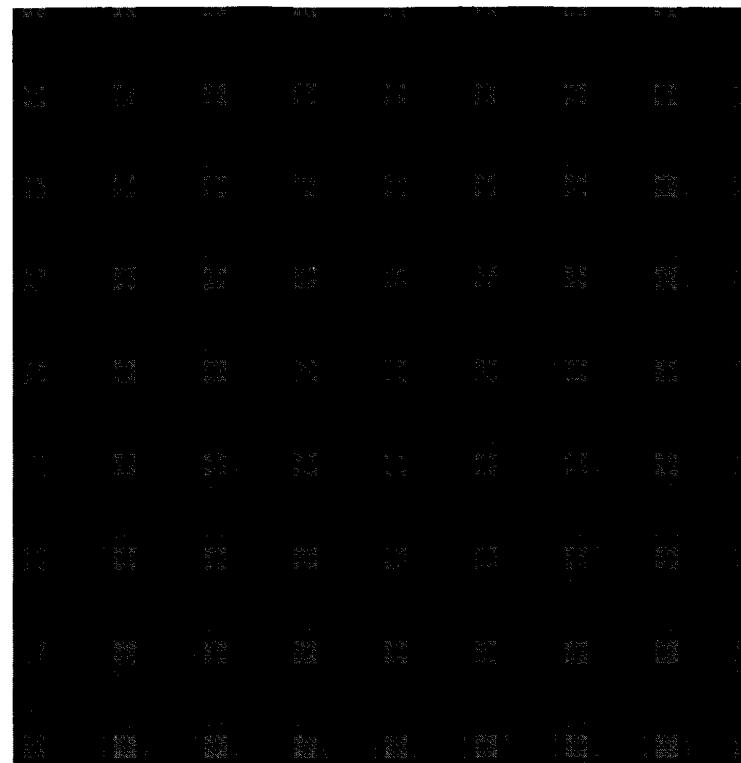


Rysunek 5.5. Wynik działania programu bez synchronizacji

Dlatego operacje zapisu i odczytu danych z pamięci wspólnej należy zsynchronizować, tak jak poniżej:

```
shared[threadIdx.x][threadIdx.y] =
    255 * (sinf(x*2.0f*PI/ period) + 1.0f) *
        (sinf(y*2.0f*PI/ period) + 1.0f) / 4.0f;
__syncthreads();
ptr[offset*4 + 0] = 0;
ptr[offset*4 + 1] = shared[15-threadIdx.x][15-threadIdx.y];
ptr[offset*4 + 2] = 0;
ptr[offset*4 + 3] = 255;
}
```

Dzięki dodaniu funkcji `__syncthreads()` wynik działania programu jest o wiele łatwiejszy do przewidzenia (i milszy dla oka), o czym świadczy zrzut ekranu przedstawiony na rysunku 5.6.



Rysunek 5.6. Wynik działania programu z synchronizacją

5.4. Podsumowanie

Wiemy już, jak dzielić bloki na mniejsze równoległe jednostki wykonawcze, zwane **wątkami**. Poprawiliśmy program sumujący wektory z poprzedniego rozdziału, aby dodawał struktury dowolnej długości. Ponadto obejrzaliśmy przykład **redukci** i pokazaliśmy, jak ją osiągnąć, używając pamięci wspólnej i synchronizacji. Na tym przykładzie pokazaliśmy, jak procesory GPU i CPU mogą współpracować w celu wspólnego wykonania zadania. Na koniec pokazaliśmy, jak niebezpieczne może być zaniedbanie synchronizacji.

Znasz już większość podstawowych technik języka CUDA C oraz wiesz, jakie są najważniejsze różnice i podobieństwa między tym językiem a językiem C. To doskonały moment na zastanowienie się, które z dotychczas napotkanych problemów można by było rozwiązać za pomocą równoległych technik CUDA C. W dalszej części książki poznasz jeszcze inne sposoby programowania GPU i nauczysz się używać zaawansowanych funkcji API CUDA.

Rozdział 6

Pamięć stała i zdarzenia

Mamy nadzieję, że dzięki lekturze poprzednich rozdziałów masz już solidne podstawy programowania procesorów graficznych. Zapewne potrafisz tworzyć równoległe bloki do wykonywania funkcji jądra oraz w razie potrzeby dzielić te bloki dalej na równoległe wątki. Wiesz także, jak zapewnić komunikację i synchronizację między wątkami. Ponieważ jednak to nie jest jeszcze koniec książki, to na pewno domyślasz się, że język CUDA C ma do zaofrowania o wiele więcej.

W tym rozdziale poznasz kilka zaawansowanych technik programowania procesorów graficznych, a konkretnie nauczysz się korzystać z tzw. **pamięci stałej**, czyli jednego ze specjalnych obszarów pamięci GPU, które można wykorzystywać do przyspieszenia działania programów. Ponieważ będzie to Twoje pierwsze zetknięcie z technikami optymalizacji w CUDA C, przy okazji nauczysz się używać tzw. **zdarzeń CUDA**, czyli metody mierzenia wydajności programów. Za jej pomocą można sprawdzić, o ile dany program działa szybciej (lub wolniej) po zastosowaniu określonych modyfikacji.

6.1. Streszczenie rozdziału

W tym rozdziale:

- Nauczysz się korzystać z pamięci stałej za pomocą języka CUDA C.
- Poznasz charakterystykę wydajnościową pamięci stałej.
- Nauczysz się mierzyć wydajność programów za pomocą zdarzeń CUDA.

2. Pamięć stała

W wcześniejszych rozdziałach zachwalaliśmy wielkie możliwości procesorów GPU w zakresie wykonywania działań arytmetycznych. To właśnie ta ich właściwość sprawiła, że zaczęto próbować nimi procesory CPU w zastosowaniach ogólnych. Gdy liczba jednostek arytmetycznych

wynosi kilka setek, największym problemem nie jest szybkość wykonywania działań, lecz przepustowość pamięci układu. Procesory graficzne mają tak dużo jednostek arytmetyczno-logicznych, że najczęstszym problemem jest dostarczanie im danych do przetwarzania z odpowiednią prędkością. Dlatego też warto poznać sposoby pozwalające zredukować wykorzystanie pamięci do niezbędnego minimum, potrzebnego do wykonania danego zadania.

Wszystkie dotychczas prezentowane programy korzystały albo z pamięci globalnej albo wspólnej, lecz język CUDA C umożliwia także użycie tzw. **pamięci stałej**. Zgodnie z nazwą w pamięci tej można przechowywać dane, które pozostają niezmienne przez cały czas wykonywania funkcji jądra. Ta 64-kilobajtowa pamięć jest traktowana nieco inaczej niż standardowa pamięć globalna, a jej użycie w niektórych przypadkach pozwala zmniejszyć apetyt programów na przepustowość tego zasobu.

6.2.1. PODSTAWY TECHNIKI ŚLEDZENIA PROMIENI

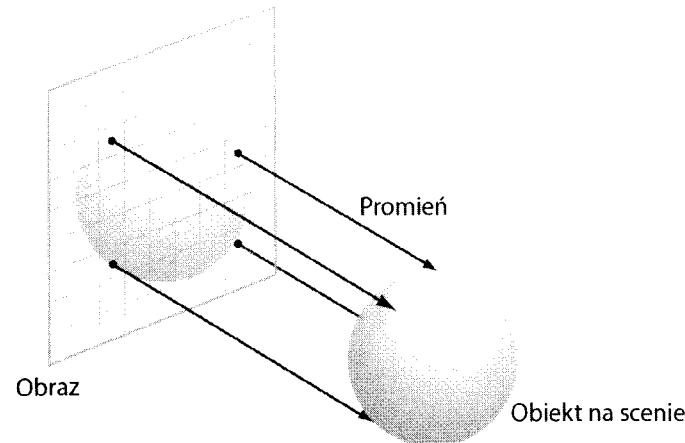
Metody wykorzystania pamięci stałej przedstawimy na prostym przykładzie programu śledzącego promienie (ang. *ray tracing*). Jeśli wiesz, na czym to polega, możesz od razu przejść do podrozdziału „Śledzenie promieni na GPU”, ponieważ tutaj wyjaśniamy podstawy tej techniki.

Najkrócej ujmując, śledzenie promieni to technika umożliwiająca przenoszenie trójwymiarowych scen na obrazy dwuwymiarowe. Ale czy to nie do tego właśnie stworzono procesory GPU? Czym to się różni od tego, co robią biblioteki OpenGL albo DirectX, gdy gram w swoju ulubioną grę komputerową? Rzeczywiście procesory GPU rozwiązuje dokładnie ten sam problem, ale wykorzystują do tego celu **rasteryzację**. Opis tej techniki można znaleźć w wielu świetnych książkach, a więc nie będziemy się tu nad nią rozwodzić. Wystarczy powiedzieć, że są to dwie całkiem różne metody rozwiązywania tego samego problemu.

Jak w takim razie śledzenie promieni pozwala uzyskać trójwymiarowy obraz sceny? Idea jest prosta: wybieramy na scenie punkt, w którym umieszczać hipotetyczną kamerę. Urządzenie to jest wyposażone w czujnik natężenia światła, aby więc wytworzyć obraz, musimy określić natężenie docierającego doń światła. Każdy piksel powstałego obrazu powinien mieć taki sam kolor i takie samo natężenie światła jak promień docierający do odpowiadającego mu punktu na czujniku.

Ponieważ światło może padać na czujnik z każdego miejsca na scenie, okazuje się, że łatwiej jest odtworzyć drogę promienia od końca, tzn. od piksela do sceny, i wówczas każdy piksel jest jak oko „patrzące” na tę scenę. Rysunek 6.1 przedstawia promienie padające z pikseli na scenę.

O tym, jaki kolor „widzi” każdy piksel, dowiadujemy się, śledząc drogę promienia biegnącego od tego piksela do jednego z obiektów na scenie. Następnie możemy powiedzieć, że piksel „zobaczył” ten obiekt, i przypiszemy mu kolor tego obiektu. Większość obliczeń wykonywanych w technice śledzenia promieni dotyczy właśnie tych punktów przecięcia promieni z obiektem na scenie.



Rysunek 6.1. Śledzenie promieni w uproszczeniu

Ponadto w bardziej wyrafinowanych modelach obecne na scenie lśniące obiekty mogą odbijać promienie światła, a przezroczyste mogą je załamywać. W ten sposób powstają promienie wtórne, trzeciorzędne itd. To właśnie stanowi o atrakcyjności techniki śledzenia promieni. Prosty model można utworzyć w bardzo łatwy sposób. W razie potrzeby można zbudować także o wiele bardziej zaawansowane modele pozwalające bardziej realistycznie odwzorowywać różne zjawiska.

6.2.2. ŚLEDZENIE PROMIENI NA GPU

Ponieważ ani OpenGL, ani DirectX nie umożliwiają renderowania grafiki techniką śledzenia promieni, do implementacji algorytmu śledzącego promienie użyjemy języka CUDA C. Abyśmy mogli skupić się przede wszystkim na sposobach wykorzystania pamięci stałej, a nie na aspektach graficznych, sam algorytm będzie bardzo prosty — osoby liczące na w pełni funkcjonalny mechanizm renderujący poczujałyby się bardzo rozczarowane. Program będzie działał tylko na scenach zawierających obiekty sferyczne, a kamera będzie skierowana tylko według osi x , w kierunku początku układu. Aby uniknąć niepotrzebnych komplikacji związanych z postawaniem promieni wtórnego, pominiemy też kwestię oświetlenia sceny. Zamiast obliczać efekty oświetleniowe, każdej kuli przypiszemy po prostu jakiś kolor, a następnie będziemy je cieniować za pomocą gotowej funkcji w taki sposób, jakby były one widoczne.

W takim razie, co będzie robił sam algorytm śledzenia promieni? Z każdego piksela wysłać promień, a następnie sprawdzić, do której kuli on dotarł. Ponadto będzie śledził, na jaką głębokość sięga każdy promień. Jeśli któryś przejdzie przez kilka obiektów, widoczny będzie tylko ten obiekt, który jest najbliższy kamery. Mówiąc krótko, algorytm będzie zasadniczo ukrywał te powierzchnie, których nie widać z kamery.

Do modelowania kul użyjemy struktury danych, w której zapisane będą współrzędne środka (x , y , z), długość promienia oraz kolor (r , g , b).

```

// Kopiowanie mapy bitowej z GPU w celu wyświetlenia
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                         bitmap.image_size(),
                         cudaMemcpyDeviceToHost ) );
bitmap.display_and_exit();
// Zwolnienie pamięci
cudaFree( dev_bitmap );
cudaFree( s );
}

```

Nie ma tu już dla nas nic nowego. Jak w takim razie odbywa się rzeczywiste śledzenie promieni? Dzięki temu, że prezentowany przykładowy model jest bardzo prosty, bez trudu zrozumiesz, jak działa funkcja jądra. Ponieważ każdy wątek generuje jeden piksel obrazu wynikowego, zaczynamy tradycyjnie od obliczenia współrzędnych x i y każdego wątku oraz obliczenia liniowej pozycji wątków w buforze wyjściowym. Ponadto przesuniemy współrzędne (x, y) naszego obrazu o $DIM/2$, aby oś z przebiegała dokładnie przez jego środek:

```

__global__ void kernel( unsigned char *ptr ) {
    // Rzutowanie threadIdx/BlockIdx na pozycje piksela
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float ox = (x - DIM/2);
    float oy = (y - DIM/2);
}

```

Ponieważ każdy promień musimy sprawdzić pod kątem, czy nie przecina jednej z kul, przejrzymy teraz iteracyjnie naszą tablicę tych obiektów i sprawdzimy po kolej każdy z nich:

```

float r=0, g=0, b=0;
float maxz = -INF;
for(int i=0; i<SPHERES; i++) {
    float n, t = s[i].hit( ox, oy, &n );
    if (t > maxz) {
        float fscale = n;
        r = s[i].r * fscale;
        g = s[i].g * fscale;
        b = s[i].b * fscale;
        maxz = t;
    }
}

```

Większość interesującego nas kodu znajduje się w pętli `for()`. Za pomocą iteracji przeglądamy wszystkie kule wejściowe, wywołując metodę `hit()` każdej z nich, w celu sprawdzenia, czy promień wysłany z naszego piksela ją „widzi”. Jeśli tak, sprawdzamy, czy punkt styku jest bliżej kamery niż poprzedni, i w przypadku potwierdzenia zapisujemy informację o głębi jako dotyczącej nowej najbliższej kuli. Dodatkowo zapisujemy kolor tej figury, dzięki czemu po zakończeniu iteracji wątek będzie znał kolor kuli, która znajduje się najbliżej kamery. Ponieważ

jest to kolor „widziany” przez promień wysłany z piksela, przyjmujemy go jako kolor właśnie tego piksela i zapisujemy jego wartość w buforze obrazu wyjściowego.

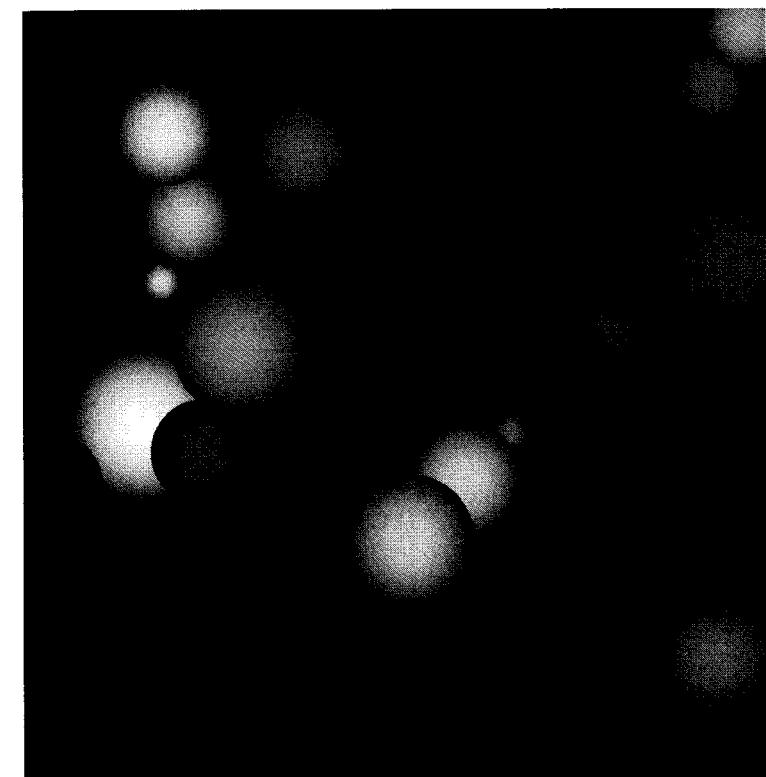
Po sprawdzeniu wszystkich kul możemy bieżący kolor zapisać w obrazie wyjściowym:

```

ptr[offset*4 + 0] = (int)(r * 255);
ptr[offset*4 + 1] = (int)(g * 255);
ptr[offset*4 + 2] = (int)(b * 255);
ptr[offset*4 + 3] = 255;
}

```

Zwróć uwagę, że jeśli promień nie natrafi na żadną kulę, to na wyjściu zostanie zapisany kolor odpowiadający początkowym wartościom zmiennych r, g i b . A ponieważ w tym przypadku wszystkie one zostały ustawione na 0, tło obrazu będzie czarne. Jeśli chcesz, aby tło miało inny kolor, musisz zmienić te wartości początkowe. Na rysunku 6.2 widać efekt działania programu ustawionego na wyrenderowanie 20 kul i czarnego tła.



Rysunek 6.2. Zrzut ekranu z programu śledzącego promienie

Jeśli otrzymasz u siebie inny obraz, nie panikuj — przypominamy, że współrzędne, kolory i rozmiary kul są generowane losowo.

```

// Kopiowanie mapy bitowej z GPU w celu wyświetlenia
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                         bitmap.image_size(),
                         cudaMemcpyDeviceToHost ) );
bitmap.display_and_exit();
// Zwolnienie pamięci
cudaFree( dev_bitmap );
cudaFree( s );
}

```

Nie ma tu już dla nas nic nowego. Jak w takim razie odbywa się rzeczywiste śledzenie promieni? Dzięki temu, że prezentowany przykładowy model jest bardzo prosty, bez trudu zrozumiesz, jak działa funkcja jądra. Ponieważ każdy wątek generuje jeden piksel obrazu wynikowego, zaczynamy tradycyjnie od obliczenia współrzędnych x i y każdego wątku oraz obliczenia liniowej pozycji wątków w buforze wyjściowym. Ponadto przesuniemy współrzędne (x, y) naszego obrazu o $DIM/2$, aby oś z przebiegała dokładnie przez jego środek:

```

__global__ void kernel( unsigned char *ptr ) {
    // Rzutowanie threadIdx/BlockIdx na pozycje piksela
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float ox = (x - DIM/2);
    float oy = (y - DIM/2);
}

```

Ponieważ każdy promień musimy sprawdzić pod kątem, czy nie przecina jednej z kul, przejrzymy teraz iteracyjnie naszą tablicę tych obiektów i sprawdzimy po kolej każdy z nich:

```

float r=0, g=0, b=0;
float maxz = -INF;
for(int i=0; i<SPHERES; i++) {
    float n, t = s[i].hit( ox, oy, &n );
    if (t > maxz) {
        float fscale = n;
        r = s[i].r * fscale;
        g = s[i].g * fscale;
        b = s[i].b * fscale;
        maxz = t;
    }
}

```

Większość interesującego nas kodu znajduje się w pętli `for()`. Za pomocą iteracji przeglądamy wszystkie kule wejściowe, wywołując metodę `hit()` każdej z nich, w celu sprawdzenia, czy promień wysłany z naszego piksela ją „widzi”. Jeśli tak, sprawdzamy, czy punkt styku jest bliżej kamery niż poprzedni, i w przypadku potwierdzenia zapisujemy informację o głębi jako dotyczącej nowej najbliższej kuli. Dodatkowo zapisujemy kolor tej figury, dzięki czemu po zakończeniu iteracji wątek będzie znał kolor kuli, która znajduje się najbliżej kamery. Ponieważ

jest to kolor „widziany” przez promień wysłany z piksela, przyjmujemy go jako kolor właśnie tego piksela i zapisujemy jego wartość w buforze obrazu wyjściowego.

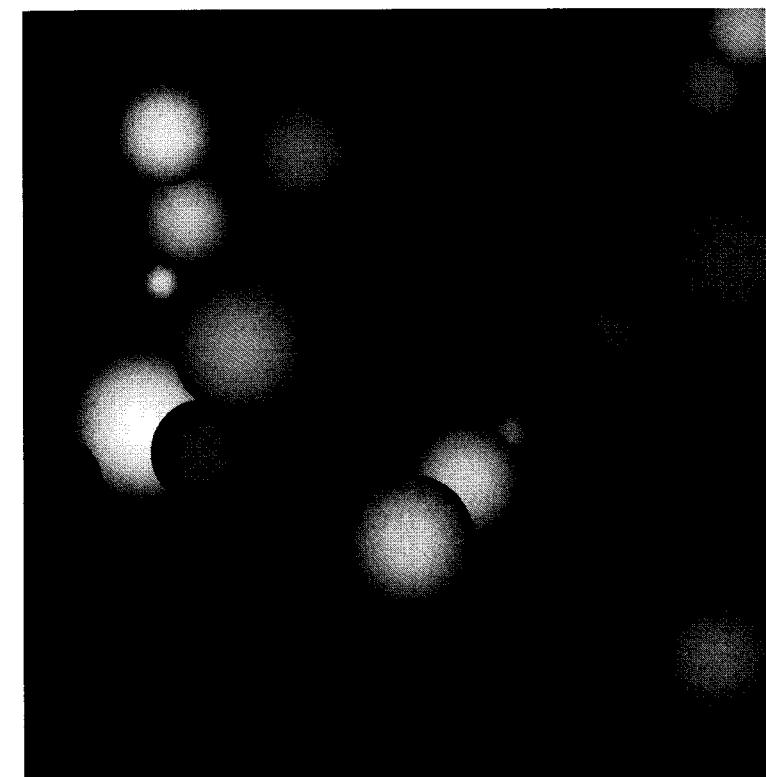
Po sprawdzeniu wszystkich kul możemy bieżący kolor zapisać w obrazie wyjściowym:

```

ptr[offset*4 + 0] = (int)(r * 255);
ptr[offset*4 + 1] = (int)(g * 255);
ptr[offset*4 + 2] = (int)(b * 255);
ptr[offset*4 + 3] = 255;
}

```

Zwróć uwagę, że jeśli promień nie natrafi na żadną kulę, to na wyjściu zostanie zapisany kolor odpowiadający początkowym wartościom zmiennych r, g i b . A ponieważ w tym przypadku wszystkie one zostały ustawione na 0, tło obrazu będzie czarne. Jeśli chcesz, aby tło miało inny kolor, musisz zmienić te wartości początkowe. Na rysunku 6.2 widać efekt działania programu ustawionego na wyrenderowanie 20 kul i czarnego tła.



Rysunek 6.2. Zrzut ekranu z programu śledzącego promienie

Jeśli otrzymasz u siebie inny obraz, nie panikuj — przypominamy, że współrzędne, kolory i rozmiary kul są generowane losowo.

6.2.3. ŚLEDZENIE PROMIENI ZA POMOCĄ PAMIĘCI STAŁEJ

Zapewne zauważłeś brak jakichkolwiek informacji na temat użycia pamięci stałej w poprzednim przykładzie. Dopiero teraz ulepszymy nasz program, korzystając z tego zasobu. Oczywiście zawartości tej pamięci nie można zmieniać w czasie działania jądra, a więc na pewno nie zapiszemy w niej danych wyjściowych. A ponieważ nasz program ma tylko jedno źródło danych wejściowych, czyli tablicę kul, nietrudno zgadnąć, co w niej zapiszemy.

Sposób deklaracji pamięci jako stałej jest dokładnie taki sam jak deklarowanie bufora jako pamięci wspólnej. Poniższą deklarację tablicy:

```
Sphere *s;
```

zastąpimy konstrukcją z modyfikatorem `_constant_`:

```
_constant_ Sphere s[SPHERES];
```

W pierwszej wersji programu zadeklarowaliśmy wskaźnik i alokowaliśmy dla niego pamięć na GPU za pomocą funkcji `cudaMalloc()`. Gdy jednak postanowiliśmy użyć pamięci stałej, napisaliśmy deklarację statycznie alokującą miejsce w pamięci stałej. W związku z tym, żeby alokować lub zwolnić pamięć przeznaczoną dla tablicy kul, nie potrzebujemy już ani wywołań funkcji `cudaMalloc()`, ani `cudaFree()`, natomiast musimy ustalić rozmiar przeznaczonej dla niej pamięci już w czasie komplikacji. Kompromis ten w wielu przypadkach się opłaca, ponieważ pozwala skorzystać z przyspieszenia, jakie daje użycie pamięci stałej. Więcej na temat wynikających z tego korzyści dowiesz się nieco dalej, a na razie zobaczymy, co zmieniło się w funkcji `main()`:

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;

    // Alokacja pamięci w GPU na wyjściową mapę bitową
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                             bitmap.image_size() ) );
    // Tymczasowa alokacja pamięci, a następnie jej inicjacja, skopiowanie do pamięci
    // stałej w GPU i zwolnienie
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                                    sizeof(Sphere) * SPHERES ) );
}
```

```
free( temp_s );
// Generowanie mapy bitowej z danych kul
dim3 grids(DIM/16,DIM/16);
dim3 threads(16,16);
kernel<<<grids,threads>>>( dev_bitmap );
// Skopiowanie mapy bitowej z GPU w celu wyświetlenia
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                         bitmap.image_size(),
                         cudaMemcpyDeviceToHost ) );
bitmap.display_and_exit();
// Zwolnienie pamięci
cudaFree( dev_bitmap );
}
```

Znajduje się tu niewiele różnic w stosunku do poprzedniej wersji. Tak jak wspominaliśmy, nie potrzebujemy już funkcji `cudaMalloc()`, aby alokować pamięć do przechowywania tablicy kul. Druga zmiana została wyróżniona na listingu:

```
HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                                    sizeof(Sphere) * SPHERES ) );
```

W kodzie tym użyta jest specjalna wersja funkcji `cudaMemcpy()` służąca do kopiowania danych z pamięci hosta do pamięci stałej GPU. Jedyna różnica między funkcjami `cudaMemcpy()` i `cudaMemcpyToSymbol()` jest taka, że `cudaMemcpyToSymbol()` kopiuje dane do pamięci stałej, a za pomocą `cudaMemcpy()` można wskaźnik skopiować tylko na miejsce w pamięci globalnej.

Poza konstrukcją z modyfikatorem `_constant_` i tymi dwiema zmianami w funkcji `main()` pozostała część kodu źródłowego obu programów jest identyczna.

6.2.4. WYDAJNOŚĆ PROGRAMU A PAMIĘĆ STAŁA

Deklarując pamięć jako stałą za pomocą modyfikatora `_constant_`, sprawiamy, że staje się ona pamięcią tylko do odczytu. Decydując się na takie wyrzeczenie, oczekujemy czegoś w zamian. Jak napisaliśmy wcześniej, dzięki użyciu 64-kilobajtowej pamięci stałej można zmniejszyć ruch w pamięci w porównaniu z tym, jaki by był, gdyby użyto zwykłej pamięci globalnej. Oto, jak to jest możliwe:

Pojedynczą operację odczytu z pamięci stałej można rozpropagować na inne „poblijskie” wątki, co pozwala zaoszczędzić do 15 operacji odczytu.

Zawartość pamięci stałej jest zapisywana w buforze podręcznym, a więc kolejne odczyty danych spod tego samego adresu nie powodują żadnego dodatkowego ruchu w pamięci.

Co konkretnie oznacza użyte w pierwszym punkcie słowo **pobłiski**? Aby odpowiedzieć na to pytanie, musimy najpierw objąć, czym jest **osnowa**. Niezorientowanym w sztuce tkackiej spieszmy z wyjaśnieniem, że osnowa to zbiór **wątków**, z których utkany jest materiał. Natomiast w architekturze CUDA osnowa jest grupą 32 wątków, które są „spłcone razem” i wykonywane w zsynchronizowany sposób, tzn. w każdym wierszu programu każdy wątek osnowy wykonuje tę samą instrukcję, ale na innych danych.

Wracając do pamięci stałej, urządzenia NVIDIA potrafią rozpropagować jedną operację odczytu na połowę osnowy. Połowa osnowy to oczywiście 16 wątków. Jeśli każdy wątek znajdujący się w połówce osnowy żąda danych znajdujących się pod tym samym adresem w pamięci stałej, to GPU wykona tylko jedną operację odczytu, a następnie rozszerzy dane do wszystkich wątków. Jeśli program często pobiera dane z pamięci stałej, to wygeneruje tylko 1/16 (około 6 procent) ruchu w pamięci w porównaniu z tym, co by było, gdyby użyto zwykłej pamięci globalnej.

Ale to nie wszystko! I tak już świetny wynik na poziomie 94% może zostać jeszcze poprawiony, ponieważ dzięki temu, że pamięć stała jest przeznaczona tylko do odczytu, karta graficzna może jej zawartość buforować w pamięci podręcznej GPU. Jeśli więc którakolwiek połówka osnowy zażąda danych spod adresu, który był już wcześniej użyty, nie spowoduje to żadnego dodatkowego ruchu w pamięci.

W naszym programie śledzącym promień każdy wątek wczytuje dane dotyczące pierwszej kuli, aby sprawdzić, czy reprezentowany przez ten promień gdzieś ją przecina. Gdy dane kul są przechowywane w pamięci stałej, urządzenie musi wykonać tylko jedno żądanie, aby je otrzymać. Później zostają one zapisane w buforze podręcznym i kolejne (chcąc z nich skorzystać) wątki już nie generują dodatkowego ruchu w pamięci, ponieważ:

- otrzymają je w ramach propagacji do połówek osnow albo
- same pobiorą je z pamięci stałej.

Niestety z używaniem pamięci stałej wiąże się też potencjalne niebezpieczeństwo zmniejszenia wydajności programu. Propagacja danych na całe połówki osnow to tak naprawdę broń obojętną. Jeśli wszystkie z 16 wątków używają tych samych danych, wówczas osiąga się radykalny wzrost wydajności. Jeżeli jednak każdy z nich pobiera dane spod innego adresu, wydajność równie radykalnie spada.

Ceną za propagację danych z jednego odczytu na 16 wątków jest to, że te same 16 wątków mogą wykonać tylko jedną operację odczytu jednocześnie. Gdyby każdy z nich potrzebował innej porcji danych, to te 16 operacji odczytu z pamięci stałej zostały wykonane jedna po drugiej, czego efektem byłoby 16-krotne wydłużenie niezbędnego na to czasu. Taki sam efekt można by uzyskać za pomocą standardowej pamięci globalnej. W tym przypadku odczyt z pamięci stałej byłby prawdopodobnie nawet trochę wolniejszy niż z pamięci globalnej.

6.3. Mierzenie wydajności programów za pomocą zdarzeń

Mając świadomość potencjalnych negatywnych konsekwencji, decydujesz się jednak na użycie w swoim programie pamięci stałej. Skoro tak, to dobrze by było w jakiś sposób zmierzyć wydajność algorytmu. Jedną z najprostszych metod jest sprawdzenie, która z wersji programu działa krócej. Do tego celu można użyć czasomierza procesora CPU albo systemu operacyjnego, ale metoda ta nie jest w pełni wiarygodna, ponieważ wynik pomiaru w niej zależy od wielu czynników, takich jak opóźnienia i różne wartości rozmaitych parametrów (harmonogramowanie wątków przez system operacyjny, dostępność precyzyjnych czasomierzy w CPU itd.). Ponadto podczas działania jądra GPU możemy asynchronicznie wykonywać jakieś obliczenia na CPU. Jedynym sposobem na zmierzenie czasu wykonywania tych operacji wykonywanych na hoście jest użycie mechanizmu do odmierzania czasu, w który wyposażony jest procesor lub system operacyjny. Aby zatem zmierzyć czas wykonywania zadania przez GPU, należy użyć zdarzeniowego API technologii CUDA.

Zdarzenie w technologii CUDA to zasadniczo znacznik czasowy, który jest rejestrowany w momencie określonym przez użytkownika. Dzięki temu, że znacznik ten jest rejestrowany przez sam GPU, pozbywamy się wielu problemów, które trzeba by było rozwiązać, gdybyśmy próbowali mierzyć czas wykonywania zadań na GPU za pomocą czasomierzy CPU. Interfejs, o którym tu mowa, jest bardzo łatwy w użyciu. Skorzystanie ze znacznika czasowego wymaga tylko dwóch czynności: utworzenia zdarzenia, a następnie jego rejestracji. Na przykład na początku wybranego fragmentu kodu nakazujemy systemowi wykonawczemu CUDA, aby zarejestrował bieżący czas. W tym celu najpierw tworzymy, a następnie rejestrujemy zdarzenie:

```
cudaEvent_t start;  
cudaEventCreate(&start);  
cudaEventRecord( start, 0 );
```

Zwróci uwagę, że w instrukcji nakazującej rejestrację zdarzenia start systemowi wykonawczemu przekazany został dodatkowy argument o wartości 0. Na razie nie ma on dla nas znaczenia, a więc pozostawimy go bez wyjaśnienia, żeby nie mieszać Ci zbytnio w głowie. Jeśli jednak nie możesz wytrzymać, to wiedz, że jego opis znajduje się w części poświęconej **strumieniom**.

Aby zmierzyć czas wykonywania bloku kodu, musimy utworzyć zdarzenie zarówno początku, jak końca. Najpierw nakażemy systemowi zarejestrować moment rozpoczęcia operacji, następnie damy mu jakieś zadanie do wykonania przez GPU, a na koniec zażądamy zakończenia rejestracji:

```
cudaEvent_t start, stop;  
cudaEventCreate(&start);  
cudaEventCreate(&stop);  
cudaEventRecord( start, 0 );  
  
// Jakiś instrukcje do wykonania przez GPU  
  
cudaEventRecord( stop, 0 );
```

Niestety powyższy kod nie jest idealny i trzeba w nim coś poprawić. I chociaż poprawka będzie niewielka, bo składająca się tylko z jednego wiersza kodu, to jej wprowadzenie wymaga jednak pewnych objaśnień. Użytkownikom zdarzeń w CUDA największe trudności sprawia fakt, że niektóre wywołania w języku CUDA C są wykonywane **asynchronicznie**. Weźmy na przykład nasz algorytm śledzenia promieni. Gdy uruchamialiśmy funkcję jądra, pracę rozpoczęłał procesor GPU, ale CPU również się nie wyłączał, tylko wykonywał dalszy kod, nie czekając, aż GPU skończy. Jest to bardzo korzystne z punktu widzenia wydajności, ponieważ w ten sposób można równocześnie wykonywać kod zarówno na CPU, jak i GPU. Jeśli jednak chodzi o mierzenie czasu wykonywania zadań, powoduje to pewne trudności.

Wywołania funkcji `cudaEventRecord()` należy traktować jako instrukcje nakazujące rejestrację bieżącego czasu, które zostają dopisane do kolejki zadań oczekujących na wykonanie przez GPU. W związku z tym nasze zdarzenie zostanie zarejestrowane dopiero po wykonaniu przez GPU wszystkich instrukcji znajdujących się przed wywołaniem funkcji `cudaEventRecord()`. Dzięki temu, rejestrując zdarzenie `stop`, możemy precyzyjnie określić czas wykonywania wybranych instrukcji programu. Nie możemy jednak **bezpiecznie odczytać** wartości zdarzenia `stop`, dopóki GPU nie zakończy wcześniejszych zadań i go nie zarejestruje. Na szczęście można nakazać procesorowi CPU dokonanie synchronizacji na wybranym zdarzeniu. Służy do tego funkcja API zdarzeniowego o nazwie `cudaEventSynchronize()`:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

// Jakiś instrukcje do wykonania przez GPU

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
```

W ten sposób zabroniliśmy systemowi wykonywanie dalszych instrukcji, dopóki GPU nie dotrze do zdarzenia `stop`. Gdy następuje zwrot funkcji `cudaEventSynchronize()`, wiadomo, że GPU zakończył wykonywanie wszystkich instrukcji znajdujących się przed zdarzeniem `stop` i że można odczytać zapisany w nim znacznik czasu. Warto przy tej okazji napomknąć, że zdarzenia CUDA są zaimplementowane bezpośrednio na GPU, a więc nie nadają się do badania mieszanek kodu przeznaczonego dla hosta i urządzenia. To znaczy zdarzenia CUDA pozwalają wiarygodnie zmierzyć tylko czas wykonywania funkcji jądra i operacji kopiowania danych z pamięci za pomocą urządzeń CUDA.

6.3.1. POMIAR WYDAJNOŚCI ALGORYTMU ŚLEDZENIA PROMIENI

Aby zmierzyć czas działania naszego algorytmu śledzenia promieni, musimy, tak jak przy poznawaniu API zdarzeń, utworzyć zdarzenie początku i końca. Poniżej znajduje się program śledzący promienie z włączonym pomiarem czasu wykonywania, w którym **nie jest** używana pamięć stała.

```
int main( void ) {
    // Zarejestrowanie czasu początkowego
    cudaEvent_t      start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;

    // Alokacja pamięci na GPU dla wyjściowej mapy bitowej
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                            bitmap.image_size() ) );
    // Alokacja pamięci dla tablicy kul
    HANDLE_ERROR( cudaMalloc( (void**)&s,
                            sizeof(Sphere) * SPHERES ) );
    // Tymczasowa alokacja pamięci, a następnie jej inicjacja, skopiowanie
    // do pamięci GPU i zwolnienie
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
    HANDLE_ERROR( cudaMemcpy( s, temp_s,
                           sizeof(Sphere) * SPHERES,
                           cudaMemcpyHostToDevice ) );
    free( temp_s );

    // Generowanie mapy bitowej z danych kul
    dim3   grids(DIM/16,DIM/16);
    dim3   threads(16,16);
    kernel<<<grids,threads>>>( s, dev_bitmap );

    // Skopiowanie mapy bitowej z GPU w celu wyświetlenia
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                            bitmap.image_size(),
                            cudaMemcpyDeviceToHost ) );

    // Rejestracja czasu końcowego i wyświetlenie wyniku pomiaru
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    float elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                       start, stop ) );
    printf( "Czas, który upłynął: %3.1f ms\n", elapsedTime );

    HANDLE_ERROR( cudaEventDestroy( start ) );
```

```

HANDLE_ERROR( cudaEventDestroy( stop ) );

// Wyświetlenie
bitmap.display_and_exit();

// Zwolnienie pamięci
cudaFree( dev_bitmap );
cudaFree( s );
}

```

W powyższym kodzie pojawiły się dwie nowe funkcje: cudaEventElapsedTime() i cudaEventDestroy(). Funkcja cudaEventElapsedTime() zwraca liczbę milisekund, jaka upłynęła między dwoma wcześniejszymi zarejestrowanymi zdarzeniami. Wartość ta jest zwracana w pierwszym argumentem, którym jest adres zmiennej zmiennoprzecinkowej.

Funkcję cudaEventDestroy() wywołuje się po zakończeniu korzystania ze zdarzenia utworzonego za pomocą funkcji cudaEventCreate(), podobnie jak wywoływana jest funkcja free() w celu zwolnienia pamięci alokowanej za pomocą funkcji malloc(). Nie musimy więc chyba podkreślić, jak ważne jest, aby każdemu wywołaniu funkcji cudaEventCreate() towarzyszyło wywołanie funkcji cudaEventDestroy().

W taki sam sposób jak wcześniej możemy zbadać program wykorzystujący pamięć stałą:

```

int main( void ) {
    // Zarejestrowanie czasu początkowego
    cudaEvent_t start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;

    // Alokacja pamięci na GPU dla wyjściowej mapy bitowej
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                           bitmap.image_size() ) );

    // Tymczasowa alokacja pamięci, a następnie jej inicjacja, skopiowanie do pamięci
    // stałej w GPU i zwolnienie
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
}

```

```

HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                                 sizeof(Sphere) * SPHERES ) );
free( temp_s );

// Generowanie mapy bitowej z danych kul
dim3 grids(DIM/16,DIM/16);
dim3 threads(16,16);
kernel<<<grids,threads>>>( dev_bitmap );

// Skopiowanie mapy bitowej z GPU w celu wyświetlenia
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                        bitmap.image_size(),
                        cudaMemcpyDeviceToHost ) );

// Rejestracja czasu końcowego i wyświetlenie wyniku pomiaru
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Czas, który upłynął: %3.1f ms\n", elapsedTime );

HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );

// Wyświetlenie
bitmap.display_and_exit();

// Zwolnienie pamięci
cudaFree( dev_bitmap );
}

```

Teraz możemy uruchomić obie wersje programu i porównać ich czasy, aby się dowiedzieć, czy użycie pamięci stałej zwiększyło wydajność, czy wręcz przeciwnie — zmniejszyło ją. W tym przypadku wzrost wydajności jest bardzo wyraźny. Z naszych badań przeprowadzonych na karcie GeForce GTX 280 wynika, że wersja z użyciem pamięci stałej działa o około 50% szybciej od wersji, w której używana jest pamięć globalna. Na innych GPU liczby mogą być trochę inne, ale mimo to wersja z pamięcią stałą nigdy nie powinna być wolniejsza od standardowej.

6.4. Podsumowanie

W urządzeniach NVIDIA, oprócz opisanych w poprzednich rozdziałach pamięciach globalnej i wspólnej, dostępny jest jeszcze inny rodzaj pamięci. Tak zwana pamięć stała ma pewne ścisłe ograniczenia, na które jednak czasami warto się zgodzić, aby zyskać na wydajności programu. Największe korzyści z jej używania można mieć w przypadkach, gdy wiele wątków w osnowie zetwarza tę samą porcję danych tylko do odczytu. Ta metoda korzystania z pamięci ma dwie

zalety pozwalające uzyskać wzrost wydajności. Po pierwsze dzięki propagacji danych na połówki osów zostaje zmniejszony ruch w pamięci, a po drugie zawartość pamięci stałej jest zapisywana w buforze podręcznym układu. Jak wiadomo, przepustowość pamięci stanowi wąskie gardło dla wielu algorytmów, a więc możliwość wyeliminowania tej niedogodności jest czymś niezwykłe przydatnym.

Ponadto nauczyliśmy się mierzyć czas wykonywania zadań przez GPU za pomocą zdarzeń CUDA. Potrafimy już zsynchronizować działanie CPU i GPU na wybranym zdarzeniu oraz sprawdzić, ile czasu minęło między dwoma wybranymi zdarzeniami. Dysponując tymi nowymi wiadomościami, zmierzyliśmy czasy wykonywania dwóch wersji programu śledzącego promienie i w wyniku ich porównania stwierdziliśmy, że wersja z użyciem pamięci stałej jest mniej więcej o połowę szybsza.

Rozdział 7

Pamięć tekstur

Poznając pamięć stałą, dowiedzieliśmy się, że dzięki wykorzystaniu specjalnego rodzaju pamięci można znacznie przyspieszyć działanie niektórych programów. Dowiedzieliśmy się też przy okazji, jak zmierzyć wyniki uzyskiwane przez program, aby mieć pewność, że dokonane zmiany rzeczywiście były korzystne. W tym rozdziale nauczysz się korzystać z kolejnego rodzaju pamięci tylko do odczytu, tzw. **pamięci tekstur** (ang. *texture memory*). Jej użycie, podobnie jak omawianej poprzednio pamięci stałej, w niektórych ściśle określonych przypadkach może zwiększyć wydajność aplikacji poprzez zmniejszenie ogólnego ruchu. Jak sama nazwa wskazuje, pamięć tekstur została oczywiście zaprojektowana do przechowywania danych graficznych, ale można ją z powodzeniem wykorzystywać także do zastosowań ogólnych.

7.1. Streszczenie rozdziału

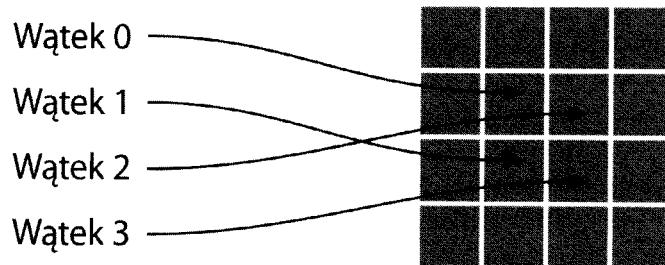
W tym rozdziale:

- Poznasz charakterystykę wydajnościową pamięci tekstur.
- Poznasz sposoby użycia pamięci tekstur jednowymiarowych w języku CUDA C.
- Nauczysz się używać pamięci tekstur dwuwymiarowych w języku CUDA C.

7.2. Pamięć tekstur w zarysie

Na samym początku rozdziału zdradziliśmy naszą małą tajemnicę — istnieje jeszcze jeden rodzaj pamięci, której można używać w programach w języku CUDA C. Dla czytelników obeznych ze sprzętem graficznym wiadomość, że wyspecjalizowaną **pamięć tekstur** procesorów NVIDIA można wykorzystywać także do zastosowań ogólnych, nie będzie zapewne wielkim założeniem. Mimo że jednostki teksturowe NVIDIA zostały zaprojektowane specjalnie dla powtarzających bibliotek OpenGL i DirectX, pewne cechy pamięci tekstur sprawiają, że ta sama technologia może się ona doskonale do wspomagania zwykłych obliczeń.

Podobnie jak pamięć stała, pamięć tekstur jest buforowana w samym układzie GPU, dzięki czemu w niektórych przypadkach pozwala zmniejszyć ruch między układem graficznym a znajdującą się poza nim pamięcią DRAM. Podręczne bufory teksturowe są przeznaczone do zastosowań w sytuacjach, gdy porcje danych graficznych są **rozproszone na niewielkiej powierzchni**. W programowaniu równoległym jest to sytuacja, w której istnieje duże prawdopodobieństwo, że dany wątek dokona odczytu z miejsca znajdującego się „w pobliżu” miejsc odczytanych przez inne wątki, jak pokazano na rysunku 7.1.



Rysunek 7.1. Wątki pobierające dane z dwuwymiarowego obszaru pamięci

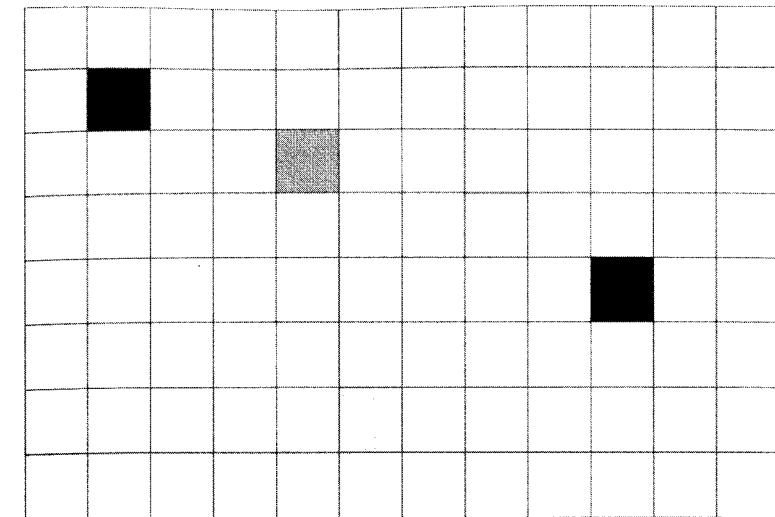
W typowej sytuacji te cztery adresy nie zostałyby przez CPU zapisane razem w buforze podręcznym, ponieważ ich adresy nie mają kolejnych numerów. Jednak podręczne bufory tekstur procesorów GPU to całkiem co innego. Są one specjalnie zaprojektowane w taki sposób, aby jak najefektywniej obsługiwać tego rodzaju sytuacje. Dlatego jeśli użyjesz się ich we właściwy sposób, następuje znaczna poprawa wydajności. Czy jednak warto zaprzątać sobie głowę jakimś rzadkim przypadkiem? Za chwilę się przekonasz, że zdarza się to częściej, niż myślisz.

7.3. Symulacja procesu rozchodzenia się ciepła

Jeśli chodzi o poziom złożoności obliczeniowej, to w ścisłej czołówce najbardziej skomplikowanych problemów znajdują się wszelkie symulacje fizyczne. Z tego względu przeprowadzanie ich często wiąże się z koniecznością zgody na kompromis między dokładnością wyników a złożonością obliczeniową algorytmu. Jednak dzięki ogromnemu postępowi, jakiego dokonano w dziedzinie przetwarzania równoległego, symulacje te zyskują na znaczeniu, ponieważ można je przeprowadzać z coraz większą precyzją. A ponieważ algorytmy tego typu często bardzo łatwo poddają się zrównolegleniu, w ramach przykładu przeanalizujemy właśnie prosty model symulacji fizycznej.

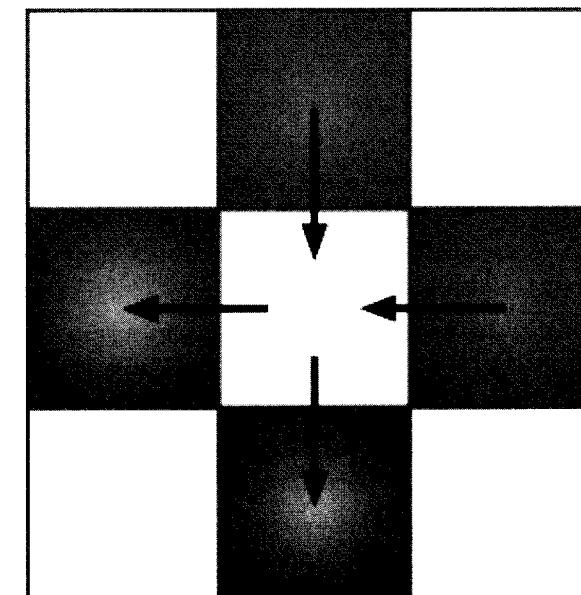
7.3.1. PROSTY MODEL OGRZEWANIA

Metody wykorzystania pamięci tekstur przedstawimy na prostym przykładzie dwuwymiarowej symulacji rozchodzenia się ciepła. Przyjmijmy, że mamy prostokątny pokój podzielony na małe kwadraty, w których losowo rozmieszczono grzejniki o różnych temperaturach. Jedna z możliwych konfiguracji pokazana jest na rysunku 7.2.



Rysunek 7.2. Pokój z „grzejnikami” o różnej temperaturze

Mając podaną prostokątną siatkę i znając rozmieszczenie grzejników, chcemy zbadać sposób rozchodzenia się ciepła w każdej z komórek w czasie. Dla uproszczenia przyjmiemy, że komórki zawierające grzejniki mają zawsze taką samą temperaturę. Co pewien z góry określony czas ciepło „przepływa” z komórki zawierającej grzejnik do sąsiednich komórek. Jeśli dana komórka jest cieplejsza od sąsiadnej, to ją ogrzewa, i odwrotnie, jeśli dana komórka jest chłodniejsza od sąsiadnej, to ją nieco ochłodzi. Proces ten przedstawiony jest na rysunku 7.3.



Rysunek 7.3. Ciepło rozchodzące się z cieplejszych komórek na chłodniejsze

W modelu, który zaprogramujemy, temperatura każdej komórki będzie wyznaczana jako suma różnic między jej temperaturą a temperaturami komórek sąsiednich. Przedstawia to poniższy wzór 7.1:

Równanie 7.1.

$$T_{\text{nowa}} = T_{\text{stara}} + \sum_{\text{sąsiedzi}} k \times (T_{\text{sąsiada}} - T_{\text{stara}})$$

W równaniu tym stała k reprezentuje tempo przepływu ciepła w czasie symulacji. Duża wartość sprawi, że system szybko ustabilizuje się na stałym poziomie, natomiast mniejsze wartości pozwolą na utrzymanie się przez dłuższy czas dużych różnic między komórkami. Ponieważ pod uwagę bierzymy tylko cztery komórki sąsiednie (górną, dolną, lewą i prawą), a k i T_{stara} to wartości stałe, powyższe równanie można przekształcić w następujący sposób (równanie 7.2):

Równanie 7.2.

$$T_{\text{nowa}} = T_{\text{stara}} + k \times (T_{\text{górna}} + T_{\text{dolna}} + T_{\text{lewa}} + T_{\text{prawa}} - 4 \times T_{\text{stara}})$$

Podobnie jak przykład algorytmu śledzenia promieni z poprzedniego rozdziału, model ten nie nadaje się do rzeczywistych zastosowań (nawet w przybliżeniu nie odzwierciedla żadnych realnych zjawisk fizycznych). Maksymalnie go uprościliśmy, aby móc skoncentrować się na opisywanych technikach. Wracając do sedna sprawy, zobaczymy, jak można obliczyć wynik równania 7.2 za pomocą GPU.

7.3.2. OBLCZANIE ZMIAN TEMPERATURY

Szczegółami zajmiemy się za moment. Teraz natomiast zobaczymy proces aktualizacji temperatur w ogólnym zarysie:

1. Mając daną wejściową siatkę temperatur, kopujemy do niej temperatury komórek zawierających grzejniki. Spowoduje to zamianę zapisanych w nich wcześniej wartości i spełnienie warunku, że „komórki z grzejnikami” mają stałą temperaturę. Kopiowanie to będzie wykonywane przez funkcję `copy_const_kernel()`.
2. Mając daną wejściową siatkę temperatur, obliczamy temperatury wyjściowe za pomocą równania 7.2. Do tego celu wykorzystujemy funkcję `blend_kernel()`.
3. Przygotowujemy się do następnego kroku poprzez dokonanie zamiany buforów wejściowego i wyjściowego. W następnym kroku symulacji siatka temperatur obliczona w punkcie 2. zostanie użыта jako wejściowa w punkcie 1.
4. Przed rozpoczęciem symulacji przyjmujemy, że wygenerowaliśmy siatkę stałych wartości. Większość z nich to zera, ale niektóre wartości są wyższe, ponieważ reprezentują grzejniki o niezmiennej temperaturze. Ten bufor stałych wartości będzie pozostawał niezmienny przez cały czas trwania symulacji i będzie odczytywany w każdym jej kroku.

W związku z przyjętą metodą przeprowadzania symulacji zaczynamy od siatki wyjściowej poprzedniego kroku. A zatem zgodnie z punktem 1. kopujemy do niej wartości temperatury komórek z grzejnikami, zastępując nimi wcześniej obliczone wartości. Robimy to, ponieważ przyjęliśmy założenie, że temperatury w komórkach zawierających grzejniki przez cały czas pozostają bez zmian. Operację nakładania tej stałej siatki na siatkę wejściową wykonujemy za pomocą poniższej funkcji jądra:

```
_global_ void copy_const_kernel( float *iptr,
                                 const float *cptr ) {
    // Rzutowanie threadIdx/BlockIdx na pozycje pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    if (cptr[offset] != 0) iptr[offset] = cptr[offset];
}
```

Trzy pierwsze wiersze kodu są nam już znane. Pierwsze dwa zamieniają indeksy `threadIdx` i `blockIdx` wątków na współrzędne x i y . Natomiast trzeci oblicza liniową pozycję w buforach danych stałych i wejściowych. Wyróżniony wiersz kopiuje temperatury grzejników zapisane w tablicy `cptr[]` do siatki wejściowej zapisanej w tablicy `iptr[]`. Zwróć uwagę, że operacja kopowania jest wykonywana tylko dla komórek siatki danych stałych, które zawierają wartości niezerowe. Robimy tak, aby zachować obliczone w poprzednim kroku wartości komórek, które nie zawierają grzejników. Ponieważ jedynymi komórkami w tablicy `cptr[]` zawierającymi wartości różne od zera są te, które zawierają grzejniki, ich wartości zostaną zachowane między kolejnymi krokami symulacji.

Największą złożoność obliczeniową ma drugi krok algorytmu. Aktualizacje temperatur możemy wykonać w ten sposób, że każdy wątek będzie odpowiadał za jedną komórkę. Odczyta jej temperaturę oraz temperatury sąsiednich komórek, wykona odpowiednie obliczenia aktualizacyjne, a następnie dokona aktualizacji temperatury nowo obliczoną wartością. Kod tej funkcji jądra w dużym stopniu pokrywa się z tym, co widzieliśmy już wcześniej.

```
_global_ void blend_kernel( float *outSrc,
                            const float *inSrc ) {
    // Rzutowanie threadIdx/BlockIdx na pozycje pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    int left = offset - 1;
    int right = offset + 1;
    if (x == 0) left++;
    if (x == DIM-1) right--;
    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0) top += DIM;
    if (y == DIM-1) bottom -= DIM;
```

```

    outSrc[offset] = inSrc[offset] + SPEED * ( inSrc[top] +
        inSrc[bottom] + inSrc[left] + inSrc[right] -
        inSrc[offset]*4);
}

```

Zwróć uwagę, że początek kodu jest dokładnie taki sam jak w programach rysujących grafikę. Lecz w tym przypadku wątki zamiast kolorów pikseli obliczają wartości temperatury w poszczególnych komórkach siatki. Ponadto znów pojawiają się wiersze kodu zamieniające identyfikatory `threadIdx` i `blockIdx` na współrzędne x i y oraz pozycje. Gdyby Cię obudzić o północy, to pewnie byś wyrecytował ten kod bez zajęcia (chociaż dla własnego zdrowia psychicznego staraj się śnić o czymś innym).

Następnie określamy położenie komórek sąsiednich (górną, dolną, lewą i prawa), tak aby móc odczytać ich temperatury. Wartości te będą nam potrzebne do obliczenia nowej temperatury w bieżącej komórce. Jedyna występująca tu komplikacja to konieczność wyregulowania indeksów komórek znajdujących się na obrzeżach siatki, tak aby nie doszło do zawinięcia. Na koniec w wyroźnym fragmencie kodu wykonujemy działanie z równania 7.2, które polega na dodaniu starej temperatury do przeskalowanych różnic tej temperatury oraz temperatur w komórkach sąsiednich.

7.3.3. ANIMACJA SYMULACJI

Pozostała część kodu programu odpowiada za konstrukcję siatki i wyświetlanie animacji reprezentującej efekt obliczeń. Oto ten kod:

```

#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_anim.h"
#define DIM 1024
#define PI 3.1415926535897932f
#define MAX_TEMP 1.0f
#define MIN_TEMP 0.0001f
#define SPEED 0.25f
//Dane globalne używane przez funkcję aktualizującą
struct DataBlock {
    unsigned char *output_bitmap;
    float *dev_inSrc;
    float *dev_outSrc;
    float *dev_constSrc;
    CPUAnimBitmap *bitmap;
    cudaEvent_t start, stop;
    float totalTime;
    float frames;
};
void anim_gpu( DataBlock *d, int ticks ) {
    HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
    dim3 blocks(DIM/16,DIM/16);

```

```

    dim3 threads(16,16);
    CPUAnimBitmap *bitmap = d->bitmap;
    for (int i=0; i<90; i++) {
        copy_const_kernel<<<blocks,threads>>>( d->dev_inSrc,
            d->dev_constSrc );
        blend_kernel<<<blocks,threads>>>( d->dev_outSrc,
            d->dev_inSrc );
        swap( d->dev_inSrc, d->dev_outSrc );
    }
    float_to_color<<<blocks,threads>>>( d->output_bitmap,
        d->dev_inSrc );
    HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(),
        d->output_bitmap,
        bitmap->image_size(),
        cudaMemcpyDeviceToHost ) );
    HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
    float elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
        d->start, d->stop ) );
    d->totalTime += elapsedTime;
    ++d->frames;
    printf( "Średni czas na ramkę: %3.1f ms\n",
        d->totalTime/d->frames );
}

void anim_exit( DataBlock *d ) {
    cudaFree( d->dev_inSrc );
    cudaFree( d->dev_outSrc );
    cudaFree( d->dev_constSrc );
    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}

```

W kodzie tym, podobnie jak w programie śledzącym promień, użyliśmy zdarzeń do zmierzenia czasu działania algorytmu. Jest nam to potrzebne, ponieważ później będziemy ten kod optymalizować i musimy wiedzieć, czy optymalizacja odniosła pożądany skutek.

Funkcja `anim_gpu()` jest wywoływaną przez algorytm dla każdej klatki animacji. Jej argumentami są wskaźnik na strukturę `DataBlock` oraz liczba kroków animacji, które już zostały wykonane. Do obsługi animacji używamy 256 wątków zorganizowanych w dwuwymiarową siatkę o wymiarach 16×16 . Zgodnie z trzykrokom algorytmem, zdefiniowanym na początku części 7.3.2, w każdej iteracji pętli `for()` znajdującej się w opisywanej funkcji obliczany jest pojedynczy krok czasowy symulacji. Ponieważ struktura `DataBlock` zawiera stały bufor grzejników oraz wynik ostatniego kroku czasowego, w rzeczywistości obejmuje ona cały stan animacji i w konsekwencji funkcja `anim_gpu()` zasadniczo nigdzie nie potrzebuje wartości parametru `ticks`.

W każdej klatce animacji wykonywanych będzie 90 kroków czasowych. Liczba ta nie wzięła się znikąd. Do wartości tej doszliśmy metodą prób i błędów, szukając kompromisu między koniecznością pobierania mapy bitowej dla każdego kroku czasowego a koniecznością obliczania odpowiedniej liczby kroków czasowych na klatkę, tak aby animacja była płynna. Gdyby wynik każdego kroku symulacji bardziej Cię interesował niż płynna animacja, możesz zmienić kod w taki sposób, aby obliczał tylko jeden krok czasowy na klatkę.

Po obliczeniu 90 kroków czasowych funkcja `anim_gpu()` jest gotowa do skopiowania mapy bitowej klatki animacji do CPU. Ponieważ pętla `for()` zamienia wejście z wyjściem, przekazujemy bufor wejściowy do następnej funkcji jądra, która zawiera wynik 90. kroku czasowego. Za pomocą funkcji jądra o nazwie `float_to_color()` zamieniamy temperatury na kolory, natomiast otrzymany obraz kopujemy do CPU za pomocą funkcji `cudaMemcpy()` z parametrem `cudaMemcpyDeviceToHost`. Na zakończenie w ramach przygotowania do kolejnej serii obliczeń kroków czasowych rejestrujemy zdarzenie zakończenia, synchronizujemy procesory CPU i GPU oraz wyświetlamy informację o tym, ile czasu zajęło obliczenie poprzednich 90 kroków czasowych.

```

int main( void ) {
    DataBlock    data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR( cudaEventCreate( &data.start ) );
    HANDLE_ERROR( cudaEventCreate( &data.stop ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.output_bitmap,
                           bitmap.image_size() ) );
    // Przyjęto założenie, że float == 4 znaki (tzn. rgba).
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                           bitmap.image_size() ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                           bitmap.image_size() ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                           bitmap.image_size() ) );
    float *temp = (float*)malloc( bitmap.image_size() );
    for (int i=0; i<DIM*DIM; i++) {
        temp[i] = 0;
        int x = i % DIM;
        int y = i / DIM;
        if ((x>300) && (x<600) && (y>310) && (y<601))
            temp[i] = MAX_TEMP;
    }
    temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
    temp[DIM*700+100] = MIN_TEMP;
    temp[DIM*300+300] = MIN_TEMP;
    temp[DIM*200+700] = MIN_TEMP;
    for (int y=800; y<900; y++) {
        for (int x=400; x<500; x++) {

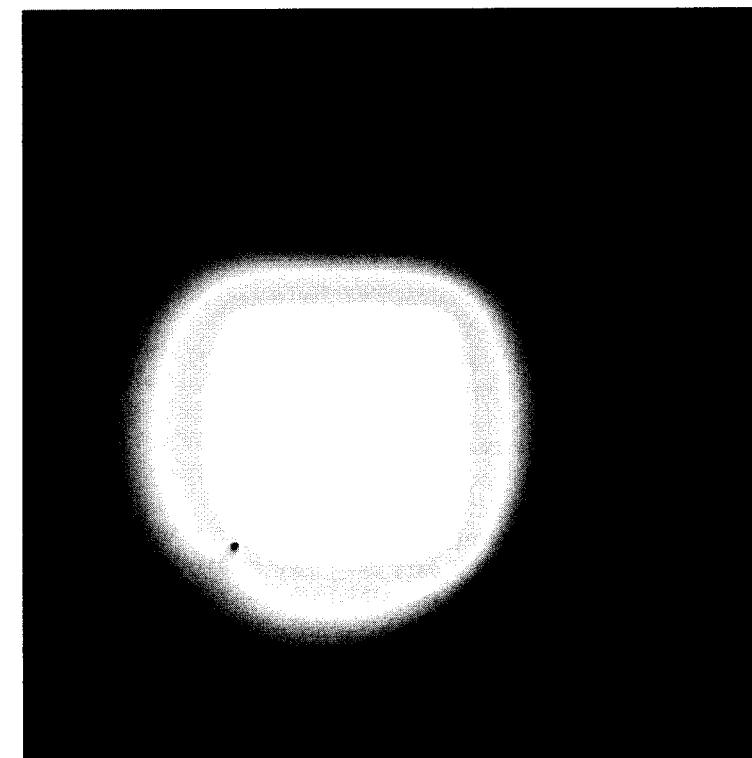
```

```

            temp[x+y*DIM] = MIN_TEMP;
        }
    }
    HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp,
                           bitmap.image_size(),
                           cudaMemcpyHostToDevice ) );
    for (int y=800; y<DIM; y++) {
        for (int x=0; x<200; x++) {
            temp[x+y*DIM] = MAX_TEMP;
        }
    }
    HANDLE_ERROR( cudaMemcpy( data.dev_inSrc, temp,
                           bitmap.image_size(),
                           cudaMemcpyHostToDevice ) );
    free( temp );
    bitmap.anim_and_exit( (void (*)(void*,int))anim_gpu,
                          (void (*)(void*))anim_exit );
}

```

Na rysunku 7.4 pokazany jest przykładowy wynik działania tego programu. Widać na nim grzejniki, które mają postać jednopikselowych wysepek na ciągłym obrazie rozkładu temperatur.



Rysunek 7.4. Zrzut ekranu przedstawiający symulację wymiany ciepła

7.3.4. UŻYCIE PAMIĘCI TEKSTUR

Dostęp do pamięci w operacjach aktualizacji temperatur charakteryzuje się wysokim stopniem **lokalności przestrzennej**. A jak wiemy, jest to doskonała okazja do wykorzystania potencjału pamięci teksturowej. Jeśli jednak chcemy użyć tego rodzaju pamięci, to oczywiście musimy się nauczyć, jak to zrobić.

Najpierw należy zadeklarować dane wejściowe jako odwołania teksturowe. W tym programie użyjemy odwołań do tekstur zmiennoprzecinkowych, ponieważ dane dotyczące temperatur są właśnie tego typu liczbami.

// Te dane istnieją tylko po stronie GPU.

```
texture<float> texConstSrc;
texture<float> texIn;
texture<float> texOut;
```

Kolejna ważna nowość, która pojawia się po alokowaniu pamięci GPU dla trzech powyższych buforów, to **powiązanie** zadeklarowanych odwołań z buforem pamięci za pomocą funkcji `cudaBindTexture()`. Wywołanie tej funkcji informuje system wykonawczy CUDA, że:

- planujemy użyć danego bufora jako tekstury,
- planujemy użyć odwołania do danej tekstury jako jej „nazwy”.

Po dokonaniu alokacji trzech buforów wiążemy je z wcześniej zadeklarowanymi odwołaniami do tekstur (`texConstSrc`, `texIn` oraz `texOut`):

```
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                           imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texConstSrc,
                               data.dev_constSrc,
                               imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texIn,
                               data.dev_inSrc,
                               imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texOut,
                               data.dev_outSrc,
                               imageSize ) );
```

Tekstury są już gotowe, a więc można zabrać się za funkcję jądra. Należy tylko pamiętać, że gdy odczytujemy dane z tekstur, musimy użyć specjalnej funkcji nakazującej GPU przesłanie naszych żądań poprzez jednostkę teksturową, a nie przez standardową pamięć globalną.

W rezultacie do odczytu danych z buforów nie możemy użyć po prostu nawiasów kwadratowych. Konieczna jest modyfikacja funkcji `blend_kernel()` w taki sposób, aby do odczytu danych z pamięci używała funkcji `tex1Dfetch()`.

Jest jeszcze inna różnica między pamięcią globalną a teksturową, która wymusza wprowadzenie dodatkowej modyfikacji w programie. Mimo że `tex1Dfetch()` wygląda jak zwykła funkcja, jest to w istocie tzw. funkcja wewnętrzna kompilatora (ang. *compiler intrinsic*). A ponieważ odwołania do teksturowej należy deklarować globalnie w zakresie całego pliku, nie możemy przekazywać buforów wejściowego i wyjściowego jako parametrów do funkcji `blend_kernel()`, ponieważ kompilator musi już na etapie komplikacji wiedzieć, które tekstury `tex1Dfetch()` ma poddać próbki. Dlatego tym razem zamiast wskaźników do buforów wejściowego i wyjściowego funkcji `blend_kernel()` przekażemy znacznik logiczny `dstOut` informujący o tym, którego bufora należy używać jako źródła danych, a którego jako ich wyjścia. Opisane zmiany zostały wyróżnione na poniższym listingu:

```
_global_ void blend_kernel( float *dst,
                            bool dstOut ) {
    // Rzutowanie z threadIdx/BlockIdx na pozycje pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    int left = offset - 1;
    int right = offset + 1;
    if (x == 0) left++;
    if (x == DIM-1) right--;
    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0) top += DIM;
    if (y == DIM-1) bottom -= DIM;
    float t, l, c, r, b;
    if (dstOut) {
        t = tex1Dfetch(texIn,top);
        l = tex1Dfetch(texIn,left);
        c = tex1Dfetch(texIn,offset);
        r = tex1Dfetch(texIn,right);
        b = tex1Dfetch(texIn,bottom);
    } else {
        t = tex1Dfetch(texOut,top);
        l = tex1Dfetch(texOut,left);
        c = tex1Dfetch(texOut,offset);
        r = tex1Dfetch(texOut,right);
        b = tex1Dfetch(texOut,bottom);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
```

Ponieważ funkcja jądra `copy_const_kernel()` wczytuje dane z bufora zawierającego informacje o położeniu i temperaturach grzejników, musimy także w niej dokonać analogicznych modyfikacji, tak aby odczyt był przeprowadzany poprzez pamięć teksturową:

```
__global__ void copy_const_kernel( float *iptr ) {
    // Rzutowanie z threadIdx/BlockIdx na pozycje pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float c = tex1Dfetch(texConstSrc, offset);
    if (c != 0)
        iptr[offset] = c;
}
```

Zmiana w sygnaturze funkcji `blend_kernel()`, która ma na celu uwzględnienie znacznika do przełączania buforów wejściowego i wyjściowego, zmusza nas do dokonania analogicznej modyfikacji w funkcji `anim_gpu()`. Zamiast zamieniać bufore, będziemy po każdej serii wywołań wykonywać instrukcję `dstOut = !dstOut` przełączającą znacznik:

```
void anim_gpu( DataBlock *d, int ticks ) {
    HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
    dim3 blocks(DIM/16,DIM/16);
    dim3 threads(16,16);
    CPUAnimBitmap *bitmap = d->bitmap;
    // Ponieważ tekstura jest globalna i związana, musimy użyć specjalnego znacznika,
    // aby określić, co w każdej iteracji jest wejściem, a co wyjściem.
    volatile bool dstOut = true;
    for (int i=0; i<90; i++) {
        float *in, *out;
        if (dstOut) {
            in = d->dev_inSrc;
            out = d->dev_outSrc;
        } else {
            out = d->dev_inSrc;
            in = d->dev_outSrc;
        }
        copy_const_kernel<<<blocks,threads>>>( in );
        blend_kernel<<<blocks,threads>>>( out, dstOut );
        dstOut = !dstOut;
    }
    float_to_color<<<blocks,threads>>>( d->output_bitmap,
                                              d->dev_inSrc );
    HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(),
                           d->output_bitmap,
                           bitmap->image_size(),
                           cudaMemcpyDeviceToHost ) );
    HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
    float elapsedTime;
```

```
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    d->start, d->stop ) );
    d->totalTime += elapsedTime;
    ++d->frames;
    printf( "Średni czas na ramkę: %3.1f ms\n",
            d->totalTime/d->frames );
}
```

Ostatnia modyfikacja, jaką musimy wprowadzić, dotyczy zwolnienia zasobów po zakończeniu działania programu. Musimy nie tylko zwolnić globalne bufore, lecz także usunąć wiązania teksturowe:

```
// Zwolnienie pamięci alokowanej na GPU
void anim_exit( DataBlock *d ) {
    cudaUnbindTexture( texIn );
    cudaUnbindTexture( texOut );
    cudaUnbindTexture( texConstSrc );
    cudaFree( d->dev_inSrc );
    cudaFree( d->dev_outSrc );
    cudaFree( d->dev_constSrc );
    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}
```

3.5. UŻYCIE DWUWYMIAROWEJ PAMIĘCI TEKSTUR

Dotychczas wspominaliśmy, że niektóre dwuwymiarowe problemy łatwiej jest rozwiązywać za pomocą dwuwymiarowych bloków i siatek. Dotyczy to także pamięci teksturowe. Dwuwymiarowe szare pamięci mogą być przydatne w wielu sytuacjach, co z pewnością jest oczywiste dla każdego, kto kiedykolwiek używał tablic wielowymiarowych w standardowym języku C. Obaczmy, co możemy zrobić, aby nasz program podczas działania korzystał z teksturow dwuwymiarowych.

Pierwszy ogień pójdu deklaracje odwołań do teksturow. Jeśli programista nie napisze inaczej, one domyślnie traktowane jako jednowymiarowe. Aby zatem zdefiniować tekstyury dwuwymiarowe, musimy dodać nowy parametr o wartości 2:

```
texture<float,2> texConstSrc;
texture<float,2> texIn;
texture<float,2> texOut;
```

Przeczenia, które powinny być możliwe dzięki użyciu teksturow dwuwymiarowych, widoczne w kodzie funkcji `blend_kernel()`. Musimy też zamienić funkcję `tex1Dfetch()` na funkcję `tex2D()`, ale za to możemy zrezygnować ze zmiennej `offset`, której używaliśmy do obliczania położenia sąsiednich komórek. Dzięki użyciu teksturow dwuwymiarowej możemy się do niej woływać bezpośrednio za pomocą wartości `x` i `y`.

Ponadto nie grozi nam już wyjście poza granice siatki, ponieważ jeśli wartość x lub y będzie mniejsza od zera, to funkcja tex2D() zwróci wartość zerową. Analogicznie, jeśli ktoś z nich przekroczy szerokość, funkcja tex2D() zwróci wartość równą szerokości. Warto zauważyć, że w przypadku naszego programu jest to idealne działanie, ale w innych aplikacjach może być szkodliwe.

Dzięki tym uproszczeniom funkcja jądra staje się o wiele bardziej przejrzysta:

```
_global_ void blend_kernel( float *dst,
                            bool dstOut ) {
    // Rzutowanie threadIdx/BlockIdx na pozycje pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float t, l, c, r, b;
    if (dstOut) {
        t = tex2D(texIn,x,y-1);
        l = tex2D(texIn,x-1,y);
        c = tex2D(texIn,x,y);
        r = tex2D(texIn,x+1,y);
        b = tex2D(texIn,x,y+1);
    } else {
        t = tex2D(texOut,x,y-1);
        l = tex2D(texOut,x-1,y);
        c = tex2D(texOut,x,y);
        r = tex2D(texOut,x+1,y);
        b = tex2D(texOut,x,y+1);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}
```

Ponieważ wszystkie wcześniejsze wywołania funkcji tex1Dfetch() muszą być zamienione na wywołania funkcji tex2D(), dokonujemy analogicznych zmian w funkcji copy_const_kernel(). Podobnie jak w przypadku jądra blend_kernel(), nie potrzebujemy już zmiennej offset, aby odwoływać się do tekstury. Wystarczą nam do tego po prostu współrzędne x i y:

```
_global_ void copy_const_kernel( float *iptr ) {
    // Rzutowanie threadIdx/BlockIdx na pozycje pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float c = tex2D(texConstSrc,x,y);
    if (c != 0)
        iptr[offset] = c;
}
```

Potrzebna jest jeszcze tylko jedna zmiana, tym razem w funkcji main(). Musimy zmodyfikować wiązania tekstur, tak aby system wykonawczy wiedział, że bufor, którego mamy zamiar używać, będzie przez nas traktowany jako tekstura dwu-, a nie jednowymiarowa:

```
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                           imageSize ) );
cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
HANDLE_ERROR( cudaBindTexture2D( NULL, texConstSrc,
                                 data.dev_constSrc,
                                 desc, DIM, DIM,
                                 sizeof(float) * DIM ) );
HANDLE_ERROR( cudaBindTexture2D( NULL, texIn,
                                 data.dev_inSrc,
                                 desc, DIM, DIM,
                                 sizeof(float) * DIM ) );
HANDLE_ERROR( cudaBindTexture2D( NULL, texOut,
                                 data.dev_outSrc,
                                 desc, DIM, DIM,
                                 sizeof(float) * DIM ) );
HANDLE_ERROR( cudaBindTexture2D( NULL, texOut,
                                 data.dev_outSrc,
                                 desc, DIM, DIM,
                                 sizeof(float) * DIM ) );
```

Podobnie jak w poprzednich dwóch wersjach programu, zaczęliśmy od alokacji pamięci dla tablic wejściowych. Nie możemy pozostać przy kodzie z wersji jednowymiarowej, ponieważ system wykonawczy CUDA wymaga, aby podczas wiązania tekstur dwuwymiarowych podawać deskryptor formatu kanału cudaChannelFormatDesc. Powyższy kod źródłowy zawiera deklarację takiego deskryptora. W tym przypadku możemy zaakceptować domyślne parametry i wystarczy, że po prostu oznajmimy, iż potrzebujemy deskryptora zmiennoprzecinkowego. Następnie dokonujemy wiązania naszych trzech buforów wejściowych jako tekstur dwuwymiarowych za pomocą funkcji cudaBindTexture2D(), wymiarów tekstuры (DIM x DIM) oraz deskryptora formatu kanału (desc). Reszta kodu funkcji main() pozostaje bez zmian:

```
int main( void ) {
    DataBlock data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR( cudaEventCreate( &data.start ) );
    HANDLE_ERROR( cudaEventCreate( &data.stop ) );
    int imageSize = bitmap.image_size();
    HANDLE_ERROR( cudaMalloc( (void**)&data.output_bitmap,
                           imageSize ) );
    // Przyjęto założenie, że float == 4 znaki (tzn. rgba).
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                           imageSize ) );
```

```

HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                        imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                        imageSize ) );
cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
HANDLE_ERROR( cudaBindTexture2D( NULL, texConstSrc,
                                data.dev_constSrc,
                                desc, DIM, DIM,
                                sizeof(float) * DIM ) );
HANDLE_ERROR( cudaBindTexture2D( NULL, texIn,
                                data.dev_inSrc,
                                desc, DIM, DIM,
                                sizeof(float) * DIM ) );
HANDLE_ERROR( cudaBindTexture2D( NULL, texOut,
                                data.dev_outSrc,
                                desc, DIM, DIM,
                                sizeof(float) * DIM ) );

// Inicjacja danych statycznych
float *temp = (float*)malloc( imageSize );
for (int i=0; i<DIM*DIM; i++) {
    temp[i] = 0;
    int x = i % DIM;
    int y = i / DIM;
    if ((x>300) && (x<600) && (y>310) && (y<601))
        temp[i] = MAX_TEMP;
}
temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
temp[DIM*700+100] = MIN_TEMP;
temp[DIM*300+300] = MIN_TEMP;
temp[DIM*200+700] = MIN_TEMP;
for (int y=800; y<900; y++) {
    for (int x=400; x<500; x++) {
        temp[x+y*DIM] = MIN_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp,
                        imageSize,
                        cudaMemcpyHostToDevice ) );

// Inicjacja danych wejściowych
for (int y=800; y<DIM; y++) {
    for (int x=0; x<200; x++) {
        temp[x+y*DIM] = MAX_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_inSrc, temp,
                        imageSize,
                        cudaMemcpyHostToDevice ) );
free( temp );
bitmap.anim_and_exit( (void (*)(void*,int))anim_gpu,
                      (void (*)(void*))anim_exit );
}

```

Może do wiązania tekstur jedno- i dwuwymiarowych potrzebne są dwie różne funkcje, to usuwania tych wiązań używa się już jednej funkcji — `cudaUnbindTexture()`. Dzięki temu funkcja porządkująca może pozostać bez zmian.

```

// Zwolnienie pamięci alokowanej na GPU
void anim_exit( DataBlock *d ) {
    cudaUnbindTexture( texIn );
    cudaUnbindTexture( texOut );
    cudaUnbindTexture( texConstSrc );
    cudaFree( d->dev_inSrc );
    cudaFree( d->dev_outSrc );
    cudaFree( d->dev_constSrc );
    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}

```

Wersja programu korzystająca z tekstur dwuwymiarowych ma praktycznie identyczną wydajność jak wersja z teksturami jednowymiarowymi. W związku z tym dla szybkości działania kobiety jest, która z tych technik zostanie wybrana. Jednak w omawianym tu przypadku użycie tekstur dwuwymiarowych pozwoliło na uproszczenie kodu źródłowego, ponieważ rozwiązywany problem był dwuwymiarowy. Ale ogólnie rzecz biorąc, przecież nie zawsze tak jest. Dlatego wybór między teksturami jedno- i dwuwymiarowymi należy dokonywać za każdym razem oddzielnie.

4. Podsumowanie

W pamiętamy z poprzedniego rozdziału, w którym opisane zostały techniki użycia pamięci podręcznej, niektóre korzyści z wykorzystania pamięci tekstur są związane z zapisywaniem danych buforze podręcznym zlokalizowanym fizycznie w układzie scalonym. Szczególnie wyraźny skutek wydajności widać w takich programach jak nasza symulacja wymiany ciepła. Charakterystyczną cechą są one wysokim poziomem lokalności przestrzennej, jeśli chodzi o dostęp do pamięci. Mówiliśmy, jak można użyć tekstur jedno- i dwuwymiarowych bez widocznego wpływu na wydajność programu w każdym z tych dwóch przypadków. Dlatego wybór techniki to przedstawiamy kwestią wygody. W opisywanej tu symulacji, ze względu na uproszczenie kodu źródłowego i uzyskanie automatycznej obsługi przypadków brzegowych, ostatecznie raczej byśmy zdecydowali na użycie tekstur dwuwymiarowych. Ale jeszcze raz podkreślamy, że obie metody są równie dobre.

Użycie tekstur pozwala nawet jeszcze bardziej przyspieszyć działanie programów, jeśli skorzystamy z możliwości automatycznych konwersji wykonywanych przez teksturowe algorytmy pakujące, takich jak wypakowywanie spakowanych danych do osobnych zmiennych lub zapisywanie 8- i 16-bitowych wartości całkowitoliczbowych na znormalizowane liczby zmiennoprzecinkowe. Żadnej z tych technik nie opisaliśmy w tym rozdziale, ale warto wiedzieć o ich istnieniu, choćż któraś z nich Ci się przyda!

Rozdział 8

Współpraca z bibliotekami graficznymi

Od momentu pojawienia się CUDA C niektóre tematy związane z wykonywaniem obliczeń ogólnych na procesorach GPU były nieco marginalne. W tym rozdziale jednak skupimy się na nich, aby przedstawić, jak można wykorzystać możliwości CUDA C do tworzenia aplikacji, w których wykonywanie obliczeń ogólnych jest głównym zadaniem. W tym celu skorzystamy z bibliotek OpenGL i DirectX.

Wykorzystanie CUDA C do tworzenia aplikacji graficznych nie jest nowością. Wczesnymi przykładami takich aplikacji były takie gry jak *Grand Theft Auto IV* i *Call of Duty: Modern Warfare 2*. W tym rozdziale jednak skupimy się na bardziej prostych aplikacjach, takich jak renderowanie obrazów, generowanie animacji i przetwarzanie danych.

W tym rozdziale skoncentrujemy się na tworzeniu aplikacji, które wykorzystują możliwości CUDA C do wykonywania obliczeń ogólnych. Przykładem takiej aplikacji może być program, który wykorzystuje CUDA C do generowania obrazów, np. do renderowania sceny 3D. Aplikacja ta może wykorzystać CUDA C do wykonywania obliczeń ogólnych, aby wykonać m.in. komiksowe efekty, takie jak rozmycie, skrawanie i przekształcanie obrazów. Aplikacja ta może również wykorzystać CUDA C do wykonywania obliczeń ogólnych, aby wykonać m.in. komiksowe efekty, takie jak rozmycie, skrawanie i przekształcanie obrazów.

8.1. Streszczenie rozdziału

W tym rozdziale:

- Dowiesz się, na czym polega **współpraca z bibliotekami graficznymi** i jakie wynikają z niej korzyści.
- Dowiesz się, jak przygotować kartę graficzną do współpracy ze sterownikami graficznymi.
- Nauczysz się przesyłać dane między funkcjami jądra w języku CUDA C a kodem OpenGL.

8.2. Współpraca z bibliotekami graficznymi

Możliwości współpracy bibliotek graficznych z językiem CUDA C zaprezentujemy na przykładzie programu podzielonego na dwie części. Najpierw za pomocą funkcji jądra w języku CUDA C będziemy generować dane graficzne, a następnie prześlemy je do sterownika OpenGL w celu wyrenderowania. Znaczna część kodu źródłowego tego programu będzie napisana w języku CUDA C za pomocą znanych nam już technik, ale pojawią się też elementy bibliotek OpenGL i GLUT.

Najpierw dołączymy do projektu nagłówki GLUT i CUDA, tak aby umożliwić sobie korzystanie z odpowiednich funkcji i wyliczeń. Zdefiniujemy także okno, w którym będzie renderowany obraz. Ponieważ rysunki będą raczej małe, wystarczy nam obiekt o rozmiarach 512×512 pikseli.

```
#define GL_GLEXT_PROTOTYPES
#include "GL/glut.h"
#include "cuda.h"
#include "cuda_gl_interop.h"
#include "../common/book.h"
#include "../common/cpu_bitmap.h"
#define DIM 512
```

Ponadto zadeklarujemy dwie zmienne globalne do przechowywania uchwytów do danych, których będziemy używać w obu częściach programu. Oba te uchwyty będą się odnosić do **tego samego** bufora, a potrzebne są dwa, ponieważ bufor ten w OpenGL będzie się inaczej nazywać niż w CUDA C. W OpenGL będzie to bufferObj, a w CUDA C — resource.

```
GLuint bufferObj;
cudaGraphicsResource *resource;
```

Przyjrzymy się teraz prawdziwemu programowi. Na początku trzeba wybrać urządzenie CUDA, na którym ma być on uruchamiany. Wiele systemów jest wyposażonych tylko w jeden procesor graficzny, a więc wybór urządzenia z obsługą technologii CUDA nie powoduje w nich żadnego problemu. Ponieważ jednak coraz częściej spotyka się systemy zawierające więcej GPU obsługujących tę technologię, musimy opracować metodę wyboru jednego z nich. Na szczęście nie będzie to trudne, ponieważ system wykonawczy CUDA zawiera wszystkie potrzebne do tego narzędzia.

```
int main( int argc, char **argv ) {
    cudaDeviceProp prop;
    int dev;
    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 0;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
```

Funkcja cudaChooseDevice() była już mowa w rozdziale 3., ale wówczas opisaliśmy ją tylko skrótnie. Teraz trochę poszerzymy naszą wiedzę na jej temat. Ogólnie rzecz biorąc, powyższy fragment kodu nakazuje systemowi wybór procesora GPU o **potencjale obliczeniowym** nie niższym niż 1.0. W tym celu najpierw zdefiniowaliśmy strukturę cudaDeviceProp, a następnie wypełniliśmy jej składowe major i minor odpowiednio na 1 i 0. Później strukturę tę przekazaliśmy do funkcji cudaChooseDevice(). Nakazuje ona systemowi wykonawczemu wybór procesora GPU, który będzie odpowiadał wymaganiom zdefiniowanym w tej strukturze. Szczegółowy potencjał obliczeniowy procesora GPU znajduje się w następnym rozdziale, a na razie proszczem o powiadomienie, że jest to wskaźnik możliwości danego urządzenia. Ponieważ wszystkie procesory obsługujące CUDA mają potencjał obliczeniowy o numerze nie niższym niż 1.0, wyższy kod w istocie nakazuje systemowi wybór dowolnego urządzenia obsługującego tę technologię i zwraca jego identyfikator w zmiennej dev. Nie wiadomo, czy wybrany zostanie wyższy z dostępnych procesorów, nie ma też gwarancji, że różne wersje systemu wykonawczego zawsze wybiorą to samo urządzenie.

Teraz po potrzeba aż tyle zachodu, aby wybrać konkretne urządzenie, to po co w ogóle tracimy czas na upełnianie składowych struktury cudaDeviceProp i wywoływanie funkcji cudaChooseDevice()? Ciekież wcześniej tego nie robiliśmy i też było dobrze. Tak, ale teraz mamy inną sytuację. Musimy znać identyfikator urządzenia CUDA, aby móc poinformować system wykonawczy, że będziemy używać tego urządzenia do wykonywania kodu zarówno OpenGL, jak i CUDA C. Dla tego służy funkcja cudaGLSetGLDevice(), której przekażemy identyfikator dev uzyskany od funkcji cudaChooseDevice():

```
HANDLE_ERROR( cudaGLSetGLDevice( dev ) );
```

W inicjalizowaniu systemu wykonawczego CUDA możemy zainicjować sterownik OpenGL, korzystając z funkcji konfiguracyjnych biblioteki GLUT (ang. *GL Utility Toolkit*). Dla każdego, kto kiedyś używał tego narzędzia, poniższy kod będzie wyglądał znajomo:

```
// Poniższe wywołania funkcji biblioteki GLUT muszą zostać wykonane przed innymi
// wywołaniami biblioteki graficznej.
glutInit( &argc, argv );
glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB );
glutInitWindowSize( DIM, DIM );
glutCreateWindow( "bitmap" );
```

System wykonawczy CUDA jest już skonfigurowany do współpracy z biblioteką OpenGL za pomocą funkcji `cudaGLSetGLDevice()`. W powyższym kodzie natomiast zainicjowaliśmy bibliotekę GLUT i utworzyliśmy okno o nazwie *bitmap*, w którym będzie rysowany efekt pracy programu. Możemy zatem przejść do sedna sprawy!

Kluczem do współpracy między OpenGL a CUDA C są wspólne bufore. Tworzy się bufor, z którego mogą korzystać obie biblioteki, i zapisuje się w nim dane, które mają być między nimi przesypane. Najpierw utworzymy obiekt bufora pikseli w OpenGL i zapiszemy uchwyt do niego w zmiennej globalnej typu `GLuint` o nazwie `bufferObj`:

```
glGenBuffers( 1, &bufferObj );
 glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj );
 glBufferData( GL_PIXEL_UNPACK_BUFFER_ARB, DIM * DIM * 4,
 NULL, GL_DYNAMIC_DRAW_ARB );
```

Objaśnienie dla osób, które nigdy nie tworzyły obiektów bufora pikseli (PBO) w OpenGL: aby utworzyć taki bufor, najpierw należy wygenerować specjalny uchwyt za pomocą funkcji `glGenBuffers()`, następnie związać go z buforem pikseli za pomocą funkcji `glBindBuffer()`, a na koniec do sterownika OpenGL wysłać żądanie alokacji takiego bufora w formie wywołania funkcji `glBufferData()`. W tym przykładzie tworzymy bufor do przechowywania $DIM \times DIM$ 32-bitowych wartości i za pomocą parametru `GL_DYNAMIC_DRAW_ARB` zastrzegamy, że będziemy go często modyfikować. Ponieważ na razie nie mamy czym wypełnić bufora, jako przedostatni argument do funkcji `glBufferData()` przekazujemy słowo kluczowe `NULL`.

Pozostało nam jeszcze tylko powiadomić system CUDA, że zamierzamy w nim używać bufora OpenGL o nazwie `bufferObj`. W tym celu rejestrujemy rzeczywisty bufor w systemie wykonawczym CUDA jako zasób graficzny:

```
HANDLE_ERROR(
 cudaGraphicsGLRegisterBuffer( &resource,
 bufferObj,
 cudaGraphicsMapFlagsNone )
);
```

Powyższy kod za pomocą funkcji `cudaGraphicsGLRegisterBuffer()` informuje system wykonawczy CUDA, że planowane jest używanie bufora PBO o nazwie `bufferObj` zarówno w CUDA, jak i w OpenGL. W odpowiedzi system CUDA zwraca uchwyt do tego bufora w zmiennej o nazwie `resource`. Będziemy go używać w kodzie CUDA C do odwoływania się do bufora `bufferObj`.

Znacznik `cudaGraphicsMapFlagsNone` świadczy o tym, że nie określamy w żaden szczególny sposób, jak będziemy tego bufora używać. Gdybyśmy jednak zmienili zdanie, to moglibyśmy użyć znacznika `cudaGraphicsMapFlagsReadOnly`, aby zezwolić tylko na odczyt danych z bufora, albo znacznika `cudaGraphicsMapFlagsWriteDiscard`, tak aby poprzednie dane były usuwane, co w istocie by sprawiło, że bufor byłby tylko do zapisu. Znaczniki te umożliwiają sterownikom CUDA i OpenGL optymalizację ustawień sprzętowych dla buforów z ograniczeniami dostępu, ale nie są obowiązkowe.

Wywołanie funkcji `glBufferData()` nakazuje sterownikowi OpenGL, aby alokował bufor o rozmiarze wystarczającym do przechowywania $DIM \times DIM$ 32-bitowych wartości. W dalszych wywołaniach funkcji z biblioteki OpenGL będziemy odwoływać się do tego bufora za pomocą uchwytu `bufferObj`, natomiast w CUDA będzie nam do tego samego celu służyć uchwyt `resource`. Ponieważ jednak z poziomu funkcji jądra CUDA C chcielibyśmy mieć możliwość zarówno zapisu do tego bufora, jak i odczytu z niego, sam uchwyt do obiektu nam nie wystarczy. Potrzebny nam jest jego adres zawarty w pamięci urządzenia, który przekażemy jądru. W tym celu nakażemy systemowi wykonawczemu CUDA wykonanie mapowania tego wspólnego zasobu, a następnie utworzenie wskaźnika na ten zmapowany zasób:

```
uchar4* devPtr;
size_t size;
HANDLE_ERROR( cudaGraphicsMapResources( 1, &resource, NULL ) );
HANDLE_ERROR(
 cudaGraphicsResourceGetMappedPointer( (void**)&devPtr,
 &size,
 resource )
);
```

Wskaźnika `devPtr` możemy używać tak jak każdego innego wskaźnika z tym wyjątkiem, że z tych tych można również używać z poziomu OpenGL jako źródła danych pikselowych. W tych wszystkich akrobacjach konfiguracyjnych reszta kodu funkcji `main()` jest już prosta. Pierwsze wywołujemy funkcję jądra, przekazując jej wskaźnik do naszego wspólnego bufora. Funkcja ta, której kodu jeszcze nie znamy, generuje dane obrazu do wyświetlenia. Następnie zaczynamy mapowanie tego zasobu. Ważne jest, aby dokonać tego przed przeprowadzeniem operacji renderowania, ponieważ stanowi to mechanizm synchronizacji kodu CUDA z kodem OpenGL. Mówiąc konkretnie, chodzi o to, że zanim rozpoczęcie się wykonywanie kodu OpenGL, najpierw muszą zakończyć działanie wszystkie funkcje CUDA wywołane przed funkcją `cudaGraphicsUnmapResources()`.

Po zakończeniu rejestrujemy funkcje zwrotne klawiatury i ekranu w GLUT (`key_func` i `draw_func`). Następnie oddajemy sterowanie do pętli renderującej GLUT, wywołując funkcję `glutMainLoop()`:

```
dim3 grids(DIM/16,DIM/16);
dim3 threads(16,16);
kernel<<<grids,threads>>>( devPtr );
HANDLE_ERROR( cudaGraphicsUnmapResources( 1, &resource, NULL ) );
// Konfiguracja biblioteki GLUT i uruchomienie pętli głównej
glutKeyboardFunc( key_func );
glutDisplayFunc( draw_func );
glutMainLoop();
```

W kodzie programu to implementacja trzech funkcji: `kernel()`, `key_func()` oraz `draw_func()`. Oto, jak wygląda ich implementacja.

Funkcja kernel() pobiera wskaźnik urządzenia i generuje dane obrazu. Poniżej znajduje się kod funkcji jądra, który napisaliśmy, wzorując się na programie generującym rozchodzące się fale z rozdziału 5.:

```
// Funkcja napisana na bazie kodu generującego rozchodzące się fale, lecz
// z użyciem typu uchar4, który może być używany w aplikacjach interoperacyjnych
__global__ void kernel( uchar4 *ptr ) {
    // Rzutowanie threadIdx/BlockIdx na pozycje pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    // Obliczenie wartości dla danego miejsca
    float fx = x/(float)DIM - 0.5f;
    float fy = y/(float)DIM - 0.5f;
    unsigned char green = 128 + 127 *
        sin( abs(fx*100) - abs(fy*100) );
    // Dostęp do typu uchar4 w porównaniu z dostępem do unsigned char*
    ptr[offset].x = 0;
    ptr[offset].y = green;
    ptr[offset].z = 0;
    ptr[offset].w = 255;
}
```

W kodzie tym wykorzystano wiele z opisywanych już technik. Kod zamieniający indeksy wątków i bloków na współrzędne x i y oraz obliczający liniowe położenie był już analizowany kilka razy. Po wykonaniu tych podstawowych czynności dokonaliśmy dość arbitralnych obliczeń kolorów pikseli w każdym miejscu (x,y), a następnie zapisaliśmy otrzymane wartości w pamięci. Do wygenerowania obrazu na GPU po raz kolejny użyliśmy proceduralnego kodu CUDA C. Najważniejsze jednak jest to, że tak uzyskane dane zostaną przekazane **bezpośrednio** do OpenGL w celu wyrenderowania, z całkowitym pominięciem procesora CPU na wszystkich etapach. Natomiast w programie generującym fale, który zaprezentowaliśmy w rozdziale 5., dane obrazu były generowane na GPU, podobnie jak tutaj, lecz później zostały skopiowane do CPU w celu wyświetlenia.

Jak w takim razie narysujemy zawartość bufora wygenerowanego przez CUDA za pomocą OpenGL? Jeśli jeszcze pamiętasz, co znajduje się w funkcji main(), może rozpoznasz poniższy wiersz kodu:

```
glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj );
```

Wywołanie to wiąże wspólny bufor jako źródło pikseli ze sterownikiem OpenGL, tak aby można było używać w wywołaniach funkcji glDrawPixels(). Oznacza to, że do wyrenderowania obrazu potrzebna nam jest tylko funkcja glDrawPixels(). W związku z tym kod funkcji draw_func() jest następujący:

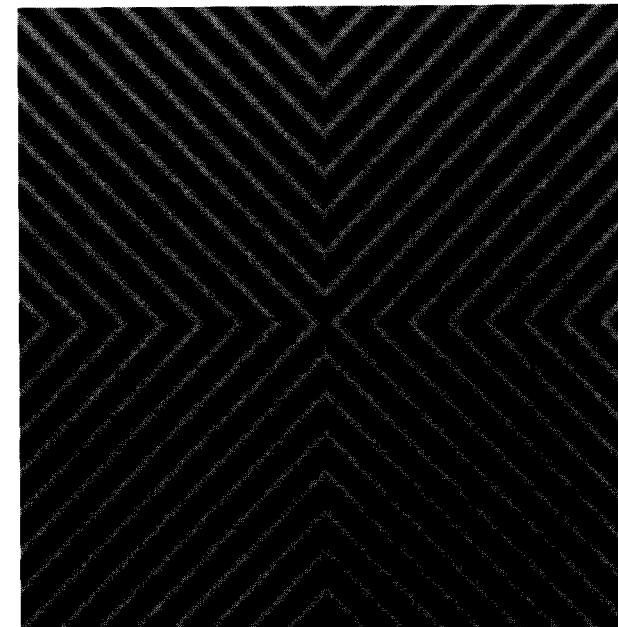
```
static void draw_func( void ) {
    glDrawPixels( DIM, DIM, GL_RGBA, GL_UNSIGNED_BYTE, 0 );
    glutSwapBuffers();
}
```

Później też spotkać wywołania funkcji glDrawPixels() ze wskaźnikiem na bufor jako ostatnim argumentem. Sterownik OpenGL pobiera dane z tego bufora wówczas, gdy wcześniej żaden bufor nie został związanego jako źródło GL_PIXEL_UNPACK_BUFFER_ARB. Ponieważ w naszym przypadku dane znajdują się już w GPU i bufor został związanego jako źródło GL_PIXEL_UNPACK_BUFFER_ARB, ostatni parametr tej funkcji określa miejsce w tym buforze. A ponieważ chcemy wyrenderować zawartość bufora, wpisaliśmy tam wartość 0.

Ostatni składnik omawianego programu wydaje się nieco odbiegać od meritum, ale zdecydowanie możemy powiedzieć, że warto też pokazać, jak można zakończyć działanie programu. Funkcja key_func() reaguje tylko na naciśnięcie klawisza Esc, które jest dla niej sygnałem do uruchomienia algorytmów porządkowych i zamknięcia aplikacji:

```
static void key_func( unsigned char key, int x, int y ) {
    switch (key) {
        case 27:
            // Porządkowanie po OpenGL i CUDA
            HANDLE_ERROR( cudaGraphicsUnregisterResource( resource ) );
            glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, 0 );
            glDeleteBuffers( 1, &bufferObj );
            exit(0);
    }
}
```

Po uruchomieniu programu w oknie pojawi się oślepiający zielono-czarny wzór, taki jak na rysunku 8.1. Możesz za jego pomocą zahipnotyzować kogoś ze znajomych (albo jakiegoś wroga).



Rysunek 8.1. Zrzut ekranu z programu do hipnotyzowania

8.3. Generowanie rozchodzących się fal za pomocą GPU i biblioteki graficznej

W poprzednim podrozdziale kilka razy odwoływaliśmy się do programu generującego fale z rozdziału 5. Przypomnijmy, że w programie tym została utworzona struktura o nazwie CPUAnimBitmap, której za każdym razem, gdy trzeba było wygenerować klatkę animacji, przekazywano odpowiednią funkcję:

```
int main( void ) {
    DataBlock data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_bitmap,
                             bitmap.image_size() ) );
    bitmap.anim_and_exit( (void (*)(void*,int))generate_frame,
                          (void (*)(void*))cleanup );
}
```

Teraz, korzystając z technik poznanych w poprzednim podrozdziale, napiszemy strukturę GPUAnimBitmap. Będzie ona pełnić taką samą rolę jak struktura CPUAnimBitmap, lecz w tym przypadku przy rozwiązywaniu zadania będą współpracować kod CUDA i OpenGL, z całkowitym pominięciem procesora CPU. Gdy skończymy pisać kod, program będzie korzystał ze struktury GPUAnimBitmap, a jego funkcja main() przyjmie następujący wygląd:

```
int main( void ) {
    GPUAnimBitmap bitmap( DIM, DIM, NULL );
    bitmap.anim_and_exit(
        (void (*)(uchar4*,void*,int))generate_frame, NULL );
}
```

W strukturze GPUAnimBitmap będziemy używać tych samych funkcji, o których była mowa w poprzednim podrozdziale. Tym razem będą one jednak zamknięte właśnie w tej strukturze, dzięki czemu kod korzystających z niej programów będzie prostszy.

8.3.1. STRUKTURA GPUANIMBITMAP

Niektóre składowe struktury GPUAnimBitmap rozpoznasz z podrozdziału 8.2.:

```
struct GPUAnimBitmap {
    GLuint bufferObj;
    cudaGraphicsResource *resource;
    int width, height;
    void *dataBlock;
    void (*fAnim)(uchar4*,void*,int);
    void (*animExit)(void*);
    void (*clickDrag)(void*,int,int,int,int);
    int dragStartX, dragStartY;
```

Wiemy, że biblioteki OpenGL i CUDA będą korzystać z różnych nazw bufora danych oraz że w zależności od potrzeby będziemy musieli korzystać z obu tych nazw. Dlatego w strukturze zapisaliśmy zarówno nazwę bufferObj dla OpenGL, jak i resource dla CUDA. A ponieważ wiadomo też, że celem programu jest wygenerowanie obrazu graficznego, który zostanie wyświetlony na ekranie, trzeba określić jego szerokość i wysokość.

Ponadto, aby umożliwić użytkownikom naszej struktury rejestrowanie funkcji zwrotnych do obsługi pewnych zdarzeń, w składowej dataBlock zapiszemy wskaźnik typu void*, który będzie mógł być używany do wskazywania na dowolne dane. Struktura sama nigdy nie będzie wykorzystywać tych danych, lecz będzie je odsyłać do zarejestrowanych funkcji zwrotnych. Funkcje zwrotne, które będzie mógł zarejestrować użytkownik, są przechowywane w składowych fAnim, animExit i clickDrag. Funkcja fAnim() jest wywoływana w każdym wywołaniu funkcji glutIdleFunc() odpowiadającej za tworzenie danych graficznych, które zostaną użyte do renderowania obrazu. Funkcja animExit() zostanie wywołana tylko raz — przy zamknięciu animacji. W niej powinien znajdować się kod porządkowy wykonywany na zakończenie działania programu. Funkcja clickDrag(), której implementacja nie jest obowiązkowa, reaguje na zdarzenia kliknięcia i przeciągania myszy. Jeśli użytkownik zdecyduje się ją zarejestrować, będzie ona wywoływana po każdej sekwencji naciśnięcia przycisku, przeciągnięcia i zwolnienia przycisku myszy. Miejsce kliknięcia inicjującego ten proces jest przechowywane w zmiennych dragStartX i dragStartY, dzięki czemu po zwolnieniu przycisku będzie można do użytkownika wysłać dane dotyczące miejsca rozpoczęcia i zakończenia operacji. Jeśli chcesz, możesz to wykorzystać do zaprogramowania interaktywnej animacji, tak aby zrobić wrażenie na swoich znajomych.

Do inicjacji struktury GPUAnimBitmap potrzebny jest podobny kod, jaki był użyty w poprzednim przykładzie. Po zapisaniu wartości w składowych nakazujemy systemowi wykonawczemu CUDA znalezienie dla nas odpowiedniego GPU:

```
GPUAnimBitmap( int w, int h, void *d ) {
    width = w;
    height = h;
    dataBlock = d;
    clickDrag = NULL;
    // Znalezienie odpowiedniego GPU i przygotowanie go do naszych celów
    cudaDeviceProp prop;
    int dev;
    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 0;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
```

Po znalezieniu właściwego urządzenia obsługującego CUDA wywołujemy funkcję cudaGLSetDevice(), aby system wykonawczy CUDA poinformować o tym, że planujemy używać urządzenia dev do współpracy z OpenGL:

```
cudaGLSetGLDevice( dev );
```

Musimy też zainicjować bibliotekę GLUT, której używamy w tym programie do utworzenia okna. Niestety nie będzie to najelegantsze rozwiązanie, ponieważ funkcja `glutInit()` przekazuje argumenty wiersza polecen do systemu okienkowego. Ponieważ my nie mamy takich argumentów do przekazania, nie powinniśmy ich podawać. Problem w tym, że niektóre wersje tej biblioteki zawierają błąd, który powoduje awarie programów, jeśli tych argumentów się nie poda. Dlatego zastosujemy poniższą sztuczkę, aby się wydawało, że przekazaliśmy argumenty:

```
int      c=1;
char    *foo = "name";
glutInit( &c, &foo );
```

Dalej postępujemy z inicjacją GLUT dokładnie w taki sam sposób jak we wcześniejszym przykładzie. Tworzymy okno, w którym będzie renderowany obraz, i ustawiamy jego tytuł na `bitmap`. Jeśli wolisz nazwać je jakoś inaczej, to nie mamy nic przeciwko.

```
glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
glutInitWindowSize( width, height );
glutCreateWindow( "bitmap" );
```

Następnie sterownikowi OpenGL nakazujemy alokację uchwytu do bufora, który natychmiast po tym wiążemy z punktem docelowym `GL_PIXEL_UNPACK_BUFFER_ARB`, dzięki czemu przyszłe wywołania funkcji `glDrawPixels()` będą rysować w naszym wspólnym buforze:

```
glGenBuffers( 1, &bufferObj );
 glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj );
```

Na koniec za pomocą sterownika OpenGL alokujemy obszar w pamięci GPU. Po zrobieniu tego powiadamy system wykonawczy CUDA o istnieniu bufora i tworzymy dla niego nazwę do użytku w CUDA C, rejestrując `bufferObj` za pomocą funkcji `cudaGraphicsGLRegisterBuffer()`.

```
glBufferData( GL_PIXEL_UNPACK_BUFFER_ARB, width * height * 4,
               NULL, GL_DYNAMIC_DRAW_ARB );
HANDLE_ERROR(
    cudaGraphicsGLRegisterBuffer( &resource,
                                 bufferObj,
                                 cudaGraphicsMapFlagsNone ) );
}
```

Jeśli chodzi o strukturę `GPUAnimBitmap`, to praca została zakończona. Teraz musimy tylko wyrenderować obraz. Sercem algorytmu rysującego jest funkcja `glutIdleFunc()`, w której wykonywane będą trzy czynności. Po pierwsze dokona ona mapowania naszego wspólnego bufora i utworzy wskaźnik do niego w pamięci GPU:

```
// Metoda statycznych używa się do implementacji funkcji zwrotnych GLUT.
static void idle_func( void ) {
    static int ticks = 1;
    GPUAnimBitmap* bitmap = *(get_bitmap_ptr());
    uchar4* devPtr;
    size_t size;
    HANDLE_ERROR(
        cudaGraphicsMapResources( 1, &(bitmap->resource), NULL )
    );
    HANDLE_ERROR(
        cudaGraphicsResourceGetMappedPointer( (void**)&devPtr,
                                              &size,
                                              bitmap->resource )
    );
}
```

Po drugie funkcja ta wywoła funkcję użytkownika `fAnim()`, która przypuszczalnie będzie uruchamiać jądro CUDA C wypełniające bufor wskazywany przez `devPtr` danymi graficznymi:

```
bitmap->fAnim( devPtr, bitmap->dataBlock, ticks++ );
```

Po trzecie omawiana funkcja wyłączy mapowanie wskaźnika na pamięć GPU, umożliwiając tym samym wykorzystanie go przez sterownik OpenGL do renderowania. Operację renderingu będzie uruchamiać wywołanie funkcji `glutPostRedisplay()`:

```
HANDLE_ERROR(
    cudaGraphicsUnmapResources( 1,
                               &(bitmap->resource),
                               NULL ) );
glutPostRedisplay();
}
```

Pozostała część struktury `GPUAnimBitmap` zawiera ważny, ale drugoplanowy kod infrastrukturalny. Jeśli Cię to interesuje, przeanalizuj go dokładnie. Ale jeśli nie masz na to czasu lub ochoty, to bez jego znajomości też nic złego Ci się nie stanie.

8.3.2. ALGORYTM GENERUJĄCY FALE NA GPU

Mając wersję struktury `GPUAnimBitmap` działającą w pełni na GPU, możemy zmodyfikować program generujący rozchodzące się fale, tak aby również on był w całości wykonywany przez procesor graficzny. Najpierw dołączymy do projektu plik nagłówkowy `gpu_anim.h`, który zawiera implementację wspomnianej struktury. Użyjemy też prawie takiego samego jądra jak w rozdziale 5:

```

#include "../common/book.h"
#include "../common/gpu_anim.h"
#define DIM 1024

__global__ void kernel( uchar4 *ptr, int ticks ) {
    //Rzutowanie threadIdx/BlockIdx na pozycje pikseli
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    //Obliczenie wartości dla każdej z pozycji
    float fx = x - DIM/2;
    float fy = y - DIM/2;
    float d = sqrtf( fx * fx + fy * fy );
    unsigned char grey = (unsigned char)(128.0f + 127.0f *
                                         cos(d/10.0f -
                                              ticks/7.0f) /
                                         (d/10.0f + 1.0f))

    ptr[offset].x = grey;
    ptr[offset].y = grey;
    ptr[offset].z = grey;
    ptr[offset].w = 255;
}

```

Jedyną zmianę wprowadzoną w powyższym kodzie oznaczyliśmy wyróżnieniem. Była ona konieczna ze względu na bibliotekę OpenGL, która wymaga, aby wspólne obszary pamięci były „przyjazne dla grafiki”. Ponieważ do renderowania na bieżąco najczęściej używa się tablic elementów czteroskładnikowych (czerwony, zielony, niebieski i alfa), nasz bufor docelowy nie może już być zwykłą tablicą, lecz musi być tablicą typu uchar4. W rzeczywistości już w rozdziale 5. bufor ten był przez nas traktowany jako czteroskładnikowy, z którego to powodu do jego indeksowania używaliśmy instrukcji `ptr[offset*4+k]`, gdzie k oznacza numer składnika od 0 do 3. Teraz jednak dzięki zastosowaniu typu uchar4 tę właściwość bufora wysunęliśmy na pierwszy plan.

Ponieważ kernel() to funkcja języka CUDA C, która generuje dane graficzne, pozostało nam już tylko napisać funkcję hosta, która będzie używana jako funkcja zwrotna w składowej idle_func() struktury GPUAnimBitmap. W tym programie jej jedynym zadaniem będzie uruchamianie funkcji jądra:

```
void generate_frame( uchar4 *pixels, void*, int ticks )
{
    dim3 grids(DIM/16,DIM/16);
    dim3 threads(16,16);
    kernel<<<grids,threads>>>( pixels, ticks );
}
```

Ponieważ większość pracy wykonuje struktura GPUAnimBitmap, w zasadzie jest to już wszystko czego nam brakowało. Aby rozpocząć zabawę, tworzymy tylko egzemplarz tej struktury i rejestrujemy funkcję zwrotną animacji generate frame();

```
int main( void ) {
    GPUAnimBitmap bitmap( DIM, DIM, NULL );
    bitmap.anim_and_exit(
        (void (*) (uchar4*, void*, int)) generate_frame, NULL );
}
```

8.4. Symulacja rozchodzenia się ciepła za pomocą biblioteki graficznej

Ale w sumie po co ta cała szopka z bibliotekami graficznymi? Jeśli dobrze przyjrzeć się kodowi struktury `GPUAnimBitmap`, można zauważyć, że prawie niczym nie różni się on od kodu renderującego z podrozdziału 8.2.

No właśnie. Prawie

Kluczowa różnica między strukturą GPUAnimBitmap a poprzednim przykładem kryje się w wywołaniu funkcji `glDrawPixels()`:

```
glDrawPixels( bitmap->x,  
              bitmap->y,  
              GL_RGBA,  
              GL_UNSIGNED_BYTE,  
              bitmap->pixels );
```

Podczas omawiania pierwszego przykładu w tym rozdziale wspomnialiśmy, że czasami można spotkać wywołania funkcji `glDrawPixels()`, w których ostatnim argumentem jest wskaźnik na bufor. Jeśli nie miałeś okazji oglądać czegoś takiego wcześniej, to masz ją teraz. Powyższe wywołanie, które znajduje się w procedurze `Draw()` struktury `CPUAnimBitmap`, przekazuje kopię bufora `bitmap->pixels` z CPU do GPU w celu wyrenderowania jego zawartości. W tym celu CPU musi przerwać swoje aktualne działanie i zainicjować kopię danych na GPU dla każdej klatki animacji. Wymaga to synchronizacji CPU i GPU oraz powoduje dodatkowy narzut związany z inicjacją transferu i przesaniem danych poprzez szynę PCI Express. Ponieważ ostatnim argumentem funkcji `glDrawPixels()` powinien być wskaźnik na pamięć hosta, oznacza to również, że program rozchodzących się fal z rozdziału 5. po wygenerowaniu klatki obrazu przez jądro CUDA C musiał klatkę tę skopiować z GPU do CPU za pomocą funkcji `cudaMemcpy()`.

ożna teraz powiedzieć, że nasza pierwotna implementacja programu generującego fale była, jak patrząc mówiąc, niezbyt mądra. Program ten za pomocą CUDA C obliczał dane graficzne, które następnie kopiował do CPU, aby później z powrotem skopiować je do GPU w celu wyświetlenia. Ruch między hostem a kartą graficzną był generowany niepotrzebnie, co niekiedy odbijało się na wydajności algorytmu. Zobaczmy, jak można to naprawić, korzystając z możliwości współpracy z biblioteką graficzną.

Pamiętajmy, że w poprzednim rozdziale został opisany program symulujący proces rozchodzącego się ciepła, w którym użyta została struktura o nazwie `CPUAnimBitmap` służąca do graficznej reprezentacji wyniku obliczeń. Teraz wstawimy do niego naszą nową wersję tej struktury o nazwie `AnimBitmap` i zobaczymy, jaki będzie to miało wpływ na jego wydajność. Podobnie jak było w przypadku programu generującego fale, zamiana struktury `CPUAnimBitmap` na `GPUAnimBitmap` nie jest bardziej prosta, a jedyną zmianą, jakiej musimy dokonać, jest zastąpienie typu `unsigned char` typem `uchar4`. W związku z tym zmieni się też sygnatura naszej procedury animacyjnej:

```
void anim_gpu( uchar4* outputBitmap, DataBlock *d, int ticks ) {
    HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
    dim3 blocks(DIM/16,DIM/16);
    dim3 threads(16,16);
    // Ponieważ tekstura jest globalna i związana, musimy użyć znacznika, aby
    // wybrać, co w każdej iteracji jest wejściem, a co wyjściem.
    volatile bool dstOut = true;
    for (int i=0; i<90; i++) {
        float *in, *out;
        if (dstOut) {
            in = d->dev_inSrc;
            out = d->dev_outSrc;
        } else {
            out = d->dev_inSrc;
            in = d->dev_outSrc;
        }
        copy_const_kernel<<<blocks,threads>>>( in );
        blend_kernel<<<blocks,threads>>>( out, dstOut );
        dstOut = !dstOut;
    }
    float_to_color<<<blocks,threads>>>( outputBitmap,
                                              d->dev_inSrc );
    HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
    float elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                         d->start, d->stop ) );
    d->totalTime += elapsedTime;
    ++d->frames;
    printf( "Średni czas na klatkę: %3.1f ms\n",
           d->totalTime/d->frames );
}
```

Ponieważ zmienna `outputBitmap` tak naprawdę jest używana tylko przez jądro `float_to_color()`, przejście na typ `uchar4` wymaga modyfikacji jeszcze tylko tej funkcji. W poprzednim rozdziale pełniła ona wyłącznie rolę pomocniczą i tak też pozostało tutaj. Lecz w pliku `book.h` zastosowaliśmy przeciążanie i zamieściliśmy dwie wersje tej funkcji — z typem `unsigned char` i `uchar4`. Można zauważyć, że różnice wydajności między tymi dwiema funkcjami są takie same jak między funkcjami `kernel()` w wersjach dla CPU i GPU programu animującego rozchodzące się fale. Dla uproszczenia prezentujemy poniżej tylko część kodu funkcji `float_to_color()`, ale jeśli koniecznie musisz zobaczyć całość, zajrzyj do pliku `book.h`.

```
_global_ void float_to_color( unsigned char *optr,
                             const float *outSrc ) {
    // Konwersja wartości zmiennoprzecinkowych na czteroskładnikowe kolory
    optr[offset*4 + 0] = value( m1, m2, h+120 );
    optr[offset*4 + 1] = value( m1, m2, h );
    optr[offset*4 + 2] = value( m1, m2, h -120 );
    optr[offset*4 + 3] = 255;
}
_global_ void float_to_color( uchar4 *optr,
                            const float *outSrc ) {

    // Konwersja wartości zmiennoprzecinkowych na czteroskładnikowe kolory
    optr[offset].x = value( m1, m2, h+120 );
    optr[offset].y = value( m1, m2, h );
    optr[offset].z = value( m1, m2, h -120 );
    optr[offset].w = 255;
}
```

Oprócz tego jedyne znaczące modyfikacje dotyczą zamiany struktury `CPUAnimBitmap` na `GPUAnimBitmap`.

```
int main( void ) {
    DataBlock data;
    GPUAnimBitmap bitmap( DIM, DIM, &data );
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR( cudaEventCreate( &data.start ) );
    HANDLE_ERROR( cudaEventCreate( &data.stop ) );
    int imageSize = bitmap.image_size();
    // Przyjęto założenie, że float == 4 znaki w size (tzn. rgba).
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                           imageSize ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                           imageSize ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                           imageSize ) );
    HANDLE_ERROR( cudaBindTexture( NULL, texConstSrc,
                                 data.dev_constSrc,
                                 imageSize ) );
    HANDLE_ERROR( cudaBindTexture( NULL, texIn,
```

```

        data.dev_inSrc,
        imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texOut,
                               data.dev_outSrc,
                               imageSize ) );

// Inicjacja stałych danych
float *temp = (float*)malloc( imageSize );
for (int i=0; i<DIM*DIM; i++) {
    temp[i] = 0;
    int x = i % DIM;
    int y = i / DIM;
    if ((x>300) && (x<600) && (y>310) && (y<601))
        temp[i] = MAX_TEMP;
}
temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
temp[DIM*700+100] = MIN_TEMP;
temp[DIM*300+300] = MIN_TEMP;
temp[DIM*200+700] = MIN_TEMP;
for (int y=800; y<900; y++) {
    for (int x=400; x<500; x++) {
        temp[x+y*DIM] = MIN_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp,
                        imageSize,
                        cudaMemcpyHostToDevice ) );

// Inicjacja danych wejściowych
for (int y=800; y<DIM; y++) {
    for (int x=0; x<200; x++) {
        temp[x+y*DIM] = MAX_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_inSrc, temp,
                        imageSize,
                        cudaMemcpyHostToDevice ) );
free( temp );
bitmap.anim_and_exit( (void (*)(uchar4*,void*,int))anim_gpu,
                      (void (*)(void*))anim_exit );
}

```

Może i warto było obejrzeć pozostałą część kodu tego programu, ale różnice w stosunku do wersji z poprzedniego rozdziału są naprawdę niewielkie. Lepiej sprawdzić, jak zmieniła się wydajność. Powinna być znacznie większa, ponieważ w wersji w całości wykonywanej na GPU odchodzi narzut spowodowany kopiowaniem każdej klatki do hosta.

O jakich konkretnie wartościach jest tu mowa? Program przedstawiony w poprzednim rozdziale na karcie GeForce GTX 285 generował klatki ze średnią prędkością 25,3 ms. Zastosowanie technik współpracy z grafiką pozwoliło uzyskać 15-procentowy wzrost wydajności, a więc średni czas generowania klatki wyniósł 21,6 ms. Innymi słowy pętla renderująca działa o 15 procent szybciej i nie angażuje hosta do wyświetlania każdej klatki. Całkiem niezły wynik jak na jeden dzień pracy!

3.5. Współpraca z DirectX

Chociaż w rozdziale tym skoncentrowaliśmy się głównie na współpracy języka CUDA C z biblioteką OpenGL, to w prawie identyczny sposób można korzystać też z DirectX. W tym przypadku również używa się typu `cudaGraphicsResource` do odwoływanego się do wspólnych buforów oraz występuje się funkcje `cudaGraphicsMapResources()` i `cudaGraphicsResourceGetMappedPointer()` do tworzenia użytecznego w CUDA wskaźnika na wspólne zasoby.

Większość różnic między OpenGL a DirectX w kodzie źródłowym to rzeczy banalne. Na przykład zamiast `cudaGLSetGLDevice()` wywołuje się funkcję `cudaD3D9SetDirect3DDevice()`, która przygotowuje kartę graficzną do współpracy z Direct3D 9.0. Analogicznie `cudaD3D10SetDirect3DDevice()` przygotowuje urządzenie do współpracy z Direct3D 10, a funkcja `cudaD3D11SetDirect3DDevice()` – z Direct3D 11.

Przeciążony dotyczące wykorzystania biblioteki DirectX nie wydadzą Ci się niczym zaskakującym, jeśli dokładnie przeczytałeś cały ten rozdział i przeanalizowałeś przykłady z użyciem OpenGL. Jeśli chcesz nauczyć się korzystać z produktu Microsoftu, to możesz na przykład wziąć kod źródłowy z tego rozdziału i odpowiednio go zmodyfikować. Początkującym polecamy też skorzystanie z podręcznika NVIDIA CUDA Programming Guide, w którym znajduje się opis API, oraz przejrzenie przykładów kodu dotyczących współpracy z DirectX zamieszczonych w pakiecie GPU Computing SDK.

3.6. Podsumowanie

Mimo że tematem głównym tej książki jest wykorzystanie procesorów GPU do równoległego wykonywania obliczeń ogólnych, nie należy zapominać, iż urządzenia te są doskonale przystosowane do renderowania grafiki. Istnieje przecież wiele programów, które mogłyby skorzystać na użyciu tych technik. Procesor GPU to mistrz renderingu i jedynym, co nam przeszkało w wykorzystaniu tych jego możliwości, był brak wiedzy o tym, jak zmusić CUDA do współpracy ze sterownikami graficznymi. Teraz dzięki temu, że już wiemy, jak to się robi, nie musimy przerywać hostowi jego pracy tylko po to, aby móc wyświetlić graficzny efekt obliczeń. W ten sposób optymalizujemy pętlę renderującą aplikacji i odciążamy procesor CPU, który może być zajęty czymś innym. Nawet jeśli nie ma on w danym czasie nic konkretnego do wykonania, to przynajmniej system działa płynniej, ponieważ sprawniej może reagować na zdarzenia.

Mogliwe są także inne sposoby współpracy ze sterownikami graficznymi, o których nawet nie wspomnieliśmy. Tutaj skoncentrowaliśmy się na technice zapisania przez jądro CUDA C danych w buforze pikseli w celu ich wyrenderowania. Dane te można także wykorzystać jako tekstury i nałożyć na dołączaną powierzchnię na ekranie. Oprócz obiektów buforów pikseli sterownik graficzny może współpracować z CUDA także obiekty buforów wierzchołków. To pozwala między innymi na pisanie jąder CUDA wykrywających kolizje między obiektami na scenie czy obliczających mapy przemieszczeń wierzchołków, których można używać do renderowania obiektów i powierzchni wchodzących w interakcję z użytkownikiem i otoczeniem. Jeśli interesuje Cię programowanie grafiki, to API języka CUDA do współpracy ze sterownikami graficznymi daje Ci szeroki wachlarz całkiem nowych możliwości!

Operacje atomowe

W pierwszej części książki przedstawiliśmy wiele przykładów zadań, których wykonanie za pomocą aplikacji jednowątkowej jest bardzo skomplikowane, a przy wykorzystaniu języka CUDA C staje się niezwykle proste. Na przykład dzięki działającemu zakulisowo systemowi wykonawczemu CUDA nie musieliszmy pisać pętli `for()`, aby wykonywać działania na pikselach animacji i symulacji. Analogicznie wystarczy z poziomu hosta wywołać funkcję oznaczoną kwalifikatorem `_global_`, aby utworzyć tysiące automatycznie indeksowanych równoległych loków i wątków wykonawczych.

jest jednak i druga strona tego medalu. Niektóre zadania, których implementacja w programie jednowątkowym jest niesłychanie prosta, w systemach równoległych mogą sprawiać poważne trudności. W tym rozdziale zajmiemy się właśnie takimi sytuacjami, w których trzeba użyć specjalnych konstrukcji, aby bezpiecznie wykonać jakieś zadanie w sposób równoległy, a których wykonanie w środowisku jednowątkowym byłoby banalnie proste.

1. Streszczenie rozdziału

W tym rozdziale:

- Dowiesz się, jaki **potencjał obliczeniowy** mają różne GPU NVIDIA.
- Poznasz operacje atomowe i dowiesz się, kiedy mogą być przydatne.
- Nauczysz się wykonywać działania arytmetyczne za pomocą operacji atomowych w funkcjach jądra języka CUDA C.

2. Potencjał obliczeniowy

Zystkie rozważane do tej pory przypadki wymagały takich funkcji, jakie ma każdy procesor obsługujący technologię CUDA. Na przykład każdy taki GPU może uruchamiać funkcje, używać pamięci globalnej oraz odczytywać dane z pamięci stałej i teksturowej. Podobnie

jednak jak procesory CPU mają różne możliwości i zbiory rozkazów (np. MMX, SSE i SSE2), tak i procesory GPU również się między sobą różnią. Taki zbiór możliwości procesora GPU firma NVIDIA nazywa jego **potencjałem obliczeniowym** (ang. *compute capability*).

9.2.1. POTENCJAŁ OBliczeniowy PROCESORÓW GPU NVIDIA

W chwili druku tej książki w sprzedaży dostępne były procesory GPU o potencjałach obliczeniowych 1.0, 1.1, 1.2, 1.3 oraz 2.0. Każdy kolejny numer reprezentuje nadzbiór poprzedniej wersji, tzn. zawiera implementację wszystkiego tego, co zawierał poprzedni standard, oraz nowe dodatki. Na przykład GPU z potencjałem obliczeniowym 1.2 ma wszystko to, co mają procesory z potencjałami 1.0 i 1.1. Aktualną listę wszystkich procesorów GPU zbudowanych na bazie architektury CUDA wraz z wyszczególnieniem ich potencjałów obliczeniowych można znaleźć w podręczniku *NVIDIA CUDA Programming Guide*. W tabeli 9.1 znajduje się natomiast wykaz tych procesorów, które były dostępne w czasie oddawania oryginału tej książki do druku. Obok każdego procesora podany jest numer jego potencjału obliczeniowego.

Tabela 9.1. Wybrane procesory GPU zbudowane na bazie architektury CUDA oraz ich potencjały obliczeniowe

GPU	Potencjał obliczeniowy
GeForce GTX 480, GTX 470	2.0
GeForce GTX 295	1.3
GeForce GTX 285, GTX 280	1.3
GeForce GTX 260	1.3
GeForce 9800 GX2	1.1
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	1.1
GeForce 8800 Ultra, 8800 GTX	1.0
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	1.1
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTX 260M, 9800M GT	1.1
GeForce 8800 GTS	1.0
GeForce 9600 GT, 8800M GTS, 9800M GTS	1.1
GeForce 9700M GT	1.1
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS	1.1
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU	1.1
GeForce 9300M GS, 9200M GS, 9100M G, 8400M G	1.1
Tesla S2070, S2050, C2070, C2050	2.0
Tesla S1070, C1060	1.3

Tabela 9.1. Wybrane procesory GPU zbudowane na bazie architektury CUDA oraz ich potencjały obliczeniowe
ciąg dalszy

GPU	Potencjał obliczeniowy
Tesla S870, D870, C870	1.0
Quadro Plex 2200 D2	1.3
Quadro Plex 2100 D4	1.1
Quadro Plex 2100 Model S4	1.0
Quadro Plex 1000 Model IV	1.0
Quadro FX 5800	1.3
Quadro FX 4800	1.3
Quadro FX 4700 X2	1.1
Quadro FX 3700M	1.1
Quadro FX 5600	1.0
Quadro FX 3700	1.1
Quadro FX 3600M	1.1
Quadro FX 4600	1.0
Quadro FX 2700M	1.1
Quadro FX 1700, FX 570, NVS 320M, FX 1700M, FX 1600M, FX 770M, FX 570M	1.1
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	1.1
Quadro FX 370M, NVS 130M	1.1

Oficjalnie cały czas pojawiają się nowe procesory, więc w chwili gdy czytasz tę książkę, powyższa lista jest już na pewno nieaktualna. Na szczęście firma NVIDIA prowadzi serwis internetowy, w którym znajduje się strefa o nazwie CUDA Zone, gdzie obok wielu innych rzeczy można znaleźć także aktualną listę urządzeń opartych na architekturze CUDA. Zdecydowanie polecamy najpierw ją przejrzeć, zanim podejmiesz jakieś radykalne kroki tylko dlatego, że nie dało Ci się znaleźć swojej karty graficznej w tabeli 9.1. Możesz też uruchomić program z rozdziału 3., który drukuje numer potencjału obliczeniowego każdego urządzenia CUDA znajdującego się w systemie.

Przez tematem tego rozdziału są operacje atomowe, szczególnie znaczenie ma dla nas to, w sprawę, którym dysponujemy, potrafi wykonywać tego rodzaju działania na pamięci. Zanim zajedziemy do szczegółowego objaśniania, czym w ogóle są operacje atomowe, pragniemy zacząć, że ich obsługa dla pamięci globalnej pojawiła się w GPU o potencjałach obliczeniowych 1.1. Natomiast dla pamięci **wspólnej** operacje te wprowadzono w potencjałach obliczeniowych 1.2. Nieważkaż kolejny numer potencjału zawiera wszystkie właściwości poprzednika, wiadomo, że procesory GPU z potencjałem 1.2 obsługują operacje atomowe zarówno na pamięci globalnej, jak i wspólnej. Analogicznie procesory z potencjałem 1.3 również je obsługują.

Jeśli procesor graficzny, którym dysponujesz, ma potencjał obliczeniowy 1.0, a więc w ogóle nie obsługuje operacji atomowych, to może jest to doskonały powód, aby trochę odnowić swój sprzęt. Jeżeli akurat nie masz możliwości, aby wysupić z trzosa środki na nową kartę grafiki, możesz przeczytać ten rozdział, aby przynajmniej teoretycznie się dowiedzieć, czym są operacje atomowe i jak się ich używa. Gdyby jednak miało Ci krewić serce z tego powodu, że nie możesz wypróbować przykładów, przejdź od razu do następnego rozdziału.

9.2.2. KOMPILACJA DLA MINIMALNEGO POTENCJAŁU OBLCZENIOWEGO

Przypuśćmy, że napisaliśmy kod, który do działania wymaga określonego minimalnego potencjału obliczeniowego, na przykład program wykorzystujący operacje atomowe na pamięci globalnej. Dzięki lekturze tego rozdziału wiesz, że do wykonywania tego typu działań potrzebny jest potencjał minimum 1.1. Aby skompilować aplikację, musisz poinformować kompilator, że funkcja jądra nie może działać na urządzeniach o potencjale niższym od 1.1. Dzięki temu kompilator może zastosować dodatkowe techniki optymalizacji dostępne tylko dla GPU o potencjale obliczeniowym 1.1 i wyższym. Samo przekazanie informacji jest bardzo proste i wymaga jedynie dodania parametru wiersza poleceń do wywołania kompilatora nvcc:

```
nvcc -arch=sm_11
```

Aby natomiast skompilować jądro, w którym wykorzystywane są operacje atomowe na pamięci wspólnej, należy poinformować kompilator, że wymagany jest potencjał obliczeniowy nie niższy niż 1.2:

```
nvcc -arch=sm_12
```

9.3. Operacje atomowe w zarysie

W zwykłych programach jednowątkowych operacje atomowe wykonuje się tylko sporadycznie. Dlatego jeśli nie miałeś z nimi do czynienia do tej pory, to się nie przejmuj, wszysko objaśnimy oraz pokażemy, do czego mogą się one przydać w programach wielowątkowych. Zastosowanie tych działań opiszymy na podstawie jednej z pierwszych konstrukcji, jakich uczy się każdy poznający języki C i C++ — operatora inkrementacji:

```
x++;
```

Jest to poprawne wyrażenie w języku C, po wykonaniu którego wartość zmiennej x będzie o jeden większa niż przed jego wykonaniem. Jakie działania są wykonywane, aby tak się stało? Aby móc zwiększyć x o jeden, przede wszystkim musimy wiedzieć, jaka wartość jest aktualnie w tej zmiennej przechowywana. Dopiero gdy ją odczytamy, możemy ją zmodyfikować. Na koniec nową wartość musimy z powrotem zapisać w x .

W związku z tym działanie powyższego operatora inkrementacji dzieli się na trzy kroki:

1. Odczytanie wartości zmiennej x .
2. Dodanie 1 do wartości odczytanej w punkcie 1.
3. Zapisanie wyniku z powrotem w zmiennej x .

Zasami proces ten nazywa się operacją **odczyt-modyfikacja-zapis**, ponieważ drugi krok może być dowolnym działaniem zmieniającym w jakiś sposób wartość odczytaną z x .

Teraz wyobraź sobie sytuację, w której przedstawioną operację inkrementacji muszą wykonać dwa wątki — dla uproszczenia nazwiemy je A i B. Aby dokonać inkrementacji, każdy z nich musi wykonać trzy kroki opisane powyżej w punktach. Przyjmując założenie, że początkowo zmieniona x ma wartość 7, idealnie by było, gdyby wątki A i B wykonały swoje zadania zgodnie z opisem zawartym w tabeli 9.2.

Tabela 9.2. Dwa wątki zwiększające wartość zmiennej x

Krok	Przykład
Wątek A odczytuje wartość zmiennej x .	$A \text{ wczytuje } 7 \text{ z } x$
Wątek A dodaje 1 do odczytanej wartości.	$A \text{ otrzymuje wartość } 8$
Wątek A zapisuje wynik działania z powrotem w zmiennej x .	$x \leftarrow 8$
Wątek B odczytuje wartość zmiennej x .	$B \text{ wczytuje } 8 \text{ z } x$
Wątek B dodaje 1 do odczytanej wartości.	$B \text{ otrzymuje wartość } 9$
Wątek B zapisuje wynik działania z powrotem w zmiennej x .	$x \leftarrow 9$

Ponieważ wartość początkowa zmiennej x wynosi 7, a następnie jest zwiększana o jeden przez dwa wątki, spodziewamy się, że po zakończeniu operacji będzie miała wartość 9. Gdyby sekwencja działań była taka, jak opisana w powyższej tabeli, to rzeczywiście otrzymalibyśmy taki wynik. Jednak na nasze nieszczęście działania te mogą zostać wykonane w innej kolejności, czego efektem może być całkiem inny wynik ostateczny. Spójrz na opis kolejności wykonywania działań przedstawiony w tabeli 9.3, w którym poszczególne operacje wątków A i B przeplatają się ze sobą.

Tabela 9.3. Dwa przeplatające się wątki zwiększające wartość zmiennej x

Krok	Przykład
Wątek A odczytuje wartość zmiennej x .	$A \text{ wczytuje } 7 \text{ z } x$
Wątek B odczytuje wartość zmiennej x .	$B \text{ wczytuje } 7 \text{ z } x$
Wątek A dodaje 1 do odczytanej wartości.	$A \text{ otrzymuje wartość } 8$
Wątek B dodaje 1 do odczytanej wartości.	$B \text{ otrzymuje wartość } 8$
Wątek A zapisuje wynik działania z powrotem w zmiennej x .	$x \leftarrow 8$
Wątek B zapisuje wynik działania z powrotem w zmiennej x .	$x \leftarrow 8$

Jeśli zatem wątki zostaną wykonane w nieodpowiedniej kolejności, otrzymamy błędny wynik. Oczywiście te sześć działań można wykonać jeszcze na wiele innych sposobów i czasami otrzyma się wynik poprawny, a czasami nie. Innymi słowy, w programach wielowątkowych, gdy kilka wątków korzysta ze wspólnego zasobu, mogą wystąpić trudności z przewidywalnością wyników.

Rozwiązaniem tego problemu jest wykonanie sekwencji działań **odczyt-modyfikacja-zapis** przez każdy wątek bez interwencji innych wątków. Albo, mówiąc inaczej, dopóki jeden wątek nie skończy działania, żaden inny nie może odczytać ani zapisać wartości zmiennej x . Takie operacje, których nie można podzielić na części przez różne wątki, nazywamy **atomowymi**. W języku CUDA C mamy ich do dyspozycji kilka. Umożliwiają one bezpieczne wykonywanie działań na pamięci nawet wówczas, gdy o dostęp konkuruje kilka tysięcy wątków. Obejrzymy przykład zastosowania tych operacji w praktyce.

9.4. Obliczanie histogramów

Często programy wykorzystuje się do obliczania danych, które następnie są stosowane do prezentacji w postaci **histogramów**. Jeśli nie wiesz, czym są histogramy, to bez obaw, zaraz się doświesz. Mówiąc krótko, dla danego zbioru elementów histogram przedstawia częstość występowania każdego z tych elementów. Na przykład dla wyrażenia **Programming with CUDA C** otrzymalibyśmy histogram pokazany na rysunku 9.1.

2	2	1	2	1	2	2	1	1	1	2	1	1	1	1
A	C	D	G	H	I	M	N	O	P	R	T	U	W	

Rysunek 9.1. Histogram częstości występowania liter w łańcuchu Programming with CUDA C

9.4.1. OBLCZANIE HISTOGRAMU ZA POMOCĄ CPU

Ponieważ nie każdy wie, jak się oblicza dane do histogramów, najpierw przedstawimy przykładowy program wykonywany przez procesor CPU. A przy okazji wykażemy, że tego typu aplikacjom jednowątkowym zadania również nie sprawiają wielkich problemów. Programowi, który napiszemy, podamy na wejściu duży (100 MB) strumień danych. Normalnie mogłyby to być dane o kolorach pikseli albo próbkach audio, ale tutaj będą to po prostu losowo wygenerowane bajty. Do ich tworzenia użyjemy funkcji pomocniczej, którą sami napisaliśmy, o nazwie `big_random_block()`:

```
#include "../common/book.h"
#define SIZE (100*1024*1024)
int main( void ) {
    unsigned char *buffer = (unsigned char*)big_random_block( SIZE );
```

Wiadomo, że jeśli bajt składa się z ośmiu bitów, to istnieje 256 różnych kombinacji wartości (od 0x00 do 0xFF). W związku z tym histogram musi zawierać 256 **pojemników** do zapisywania liczebności każdego z elementów rozpatrywanego zbioru. Dlatego utworzymy 256-pojemnikową tablicę i wszystkie jej pojemniki zainicjujemy zerem:

```
unsigned int histo[256];
for (int i=0; i<256; i++)
    histo[i] = 0;
```

Mamy już utworzony histogram z pojemnikami zainicjowanymi zerami. Teraz musimy ująć w formie tabeli częstotliwości występowania każdej z wartości zapisanych w tablicy `buffer[]`. Idea jest taka, że gdy napotkamy wartość z w tablicy `buffer[]`, wówczas wartość w pojemniku z histogramu zwiększymy o jeden. W ten sposób rejestrujemy, ile razy napotkaliśmy wartość z w analizowanym zbiorze danych.

Jeśli aktualnie rozpatrujemy wartość `buffer[i]`, to powinniśmy zwiększyć liczbę zapisaną w pojemniku właśnie o numerze `buffer[i]`. Ponieważ pojemnik `buffer[i]` znajduje się w elemencie `histo[buffer[i]]`, do zwiększenia wartości odpowiedniego licznika wystarczy nam jeden wiersz kodu:

```
histo[buffer[i]]++;
```

Czynność tę wykonujemy dla każdego elementu tablicy `buffer[]` za pomocą prostej pętli `for()`:

```
for (int i=0; i<SIZE; i++)
    histo[buffer[i]]++;
```

W ten sposób ukończymy nasz histogram danych wejściowych. W realnym programie mógłby on stanowić dane wejściowe dla jakiegoś kolejnego etapu obliczeń. Natomiast w naszym uproszczonym przykładzie poprzestaniemy na tym i jeszcze tylko sprawdzimy, czy suma wartości wszystkich pojemników zgadza się z przewidywaniami:

```
long histoCount = 0;
for (int i=0; i<256; i++) {
    histoCount += histo[i];
}
printf( "Suma wartości histogramu: %ld\n", histoCount );
```

Jeśli uważnie śledziłeś dotychczasowy tekst, z pewnością zauważyleś, że bez względu na wartość losowej tablicy wejściowej suma ta zawsze będzie taka sama. Ponieważ w każdym pojemniku znajduje się liczba reprezentująca częstość występowania jednego z elementów danych w zbiorze, to suma wartości tych pojemników powinna być równa liczbie wszystkich elementów poddanych badaniu. W naszym przypadku jej wartość wynosi tyle samo co `SIZE`.

I na zakończenie nie musimy chyba przypominać (ale to zrobimy), że trzeba po sobie zostawić porządek:

```
    free( buffer );
    return 0;
}
```

W naszym komputerze testowym, wyposażonym w procesor Core 2 Duo, utworzenie histogramu dla 100 MB tablicy danych zajęło 0,416 sekundy. Później porównamy tę wartość z wersją dla GPU.

9.4.2. OBLICZANIE HISTOGRAMU PRZY UŻYCIU GPU

Teraz zaadaptujemy napisany wcześniej algorytm dla procesora GPU. Przy dużym rozmiarze tablicy wejściowej możliwość wykorzystania wielu wątków do analizy różnych części bufora wejściowego może pomóc znacznie skrócić czas wykonywania obliczeń. Ponadto opracowanie wielowątkowego algorytmu przetwarzającego te dane nie powinno być trudne. Już nieraz wykonywaliśmy podobne zadania. Jednak w tym przypadku występuje ten problem, że kilka różnych wątków może próbować zwiększać wartość tego samego pojemnika histogramu jednocześnie. Aby go rozwiązać, musimy użyć operacji atomowych, których teoretyczny opis znajduje się w podrozdziale 9.3.

Kod funkcji `main()` będzie w dużej części pokrywał się z wersją dla CPU, z tą różnicą, że znajdzie się w niej dodatek instrukcji w języku CUDA C przenoszących na GPU dane wejściowe i odbierających z GPU wynik obliczeń. Początek będzie jednak identyczny jak poprzednio:

```
int main( void ) {
    unsigned char *buffer = (unsigned char*)big_random_block( SIZE );
```

Ponieważ chcemy porównać wydajność naszych algorytmów, zainicjujemy zdarzenia służące do zmierzenia czasu działania programu:

```
cudaEvent_t start, stop;
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

Po utworzeniu zbioru danych i zdarzeń przechodzimy do pamięci GPU. Musimy alokować miejsce na losowe dane wejściowe i histogram, którego obliczenie jest naszym celem. W związku z tym alokujemy na GPU bufor wejściowy i kopujemy do niego tablicę wygenerowaną za pomocą funkcji `big_random_block()`. Następnie, podobnie jak w wersji dla CPU, alokujemy histogram i inicjujemy go zerami:

```
// Alokacja pamięci na GPU
unsigned char *dev_buffer;
unsigned int *dev_hist;
HANDLE_ERROR( cudaMalloc( (void**)&dev_buffer, SIZE ) );
HANDLE_ERROR( cudaMemcpy( dev_buffer, buffer, SIZE,
                        cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_hist,
                        256 * sizeof( int ) ) );
HANDLE_ERROR( cudaMemset( dev_hist, 0,
                        256 * sizeof( int ) ) );
```

powyższym kodzie przemyciliśmy nową funkcję języka CUDA C o nazwie `cudaMemset()`. Jej natura jest podobna do standardowej funkcji `memset()` języka C i rzeczywiście obie te funkcje działają prawie identycznie. Różnica między nimi polega na tym, że `cudaMemset()` w przypadku wystąpienia błędu zwraca jego kod, a `memset()` tego nie robi. Zwrócony kod pozwala programiście zorientować się, że podczas alokacji pamięci na GPU wystąpił jakiś problem. Różnica między tymi funkcjami polega oczywiście na tym, że `cudaMemset()` działa na pamięci GPU, a `memset()` na pamięci hosta.

Najac zainicjowane bufore wejściowy i wyjściowy, możemy rozpocząć obliczanie histogramu. Służące do tego funkcji jądra pokażemy za chwilę, a na razie przyjmijmy, że już zakończyły te obliczenia. Skoro tak, to musimy skopiować otrzymany histogram z powrotem do CPU. W tym celu alokujemy 256-elementową tablicę i wykonujemy operacje kopiowania urządzenia do hosta:

```
unsigned int histo[256];
HANDLE_ERROR( cudaMemcpy( histo, dev_hist,
                        256 * sizeof( int ),
                        cudaMemcpyDeviceToHost ) );
```

W tym momencie obliczanie histogramu jest już zakończone, więc można zatrzymać odmianę czasu i wyświetlić wynik. Poniższy kod jest taki sam jak we wcześniejszych programach, których mierzyliśmy czas wykonywania algorytmów.

```
// Sprawdzenie czasu zakończenia oraz wyświetlenie wyniku pomiaru
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Czas generowania: %3.1f ms\n", elapsedTime );
```

Taz moglibyśmy nasz histogram przekazać na wejście do innego algorytmu, ale ponieważ nie mamy takich planów, to tylko sprawdzimy, czy wynik otrzymany na GPU odpowiada temu, co wyskaliśmy na CPU. Najpierw weryfikujemy sumę elementów histogramu. Kod, którego do tego użyjemy, jest identyczny z poprzednią wersją programu:

```

long histoCount = 0;
for (int i=0; i<256; i++) {
    histoCount += histo[i];
}
printf( " Suma wartości histogramu: %ld\n", histoCount );

```

Aby jednak w pełni zweryfikować histogram obliczony przez GPU, te same obliczenia wykonamy za pomocą CPU. Oczywiście najprostszym sposobem na zrobienie tego jest alokacja nowej tablicy na histogram, obliczenie histogramu z danych wejściowych, które już mamy, za pomocą kodu z części 9.4.1 oraz porównanie zawartości pojemników z wersji zwróconej przez CPU i GPU w celu dowiedzenia się, czy są identyczne. My jednak, zamiast alokować nową tablicę na histogram, zaczniemy od histogramu obliczonego na GPU i obliczymy histogram CPU „w odwrotnej kolejności”.

Słowa „w odwrotnej kolejności” oznaczają, że zamiast zaczynać od zera i zwiększać wartości w pojemnikach histogramu po napotkaniu odpowiednich elementów danych, pracę rozpoczęliśmy od gotowego histogramu obliczonego przez GPU i będziemy **zmnieszać** wartości pojemników, wtedy gdy CPU napotka odpowiednie elementy danych. Jeśli po zakończeniu pracy programu histogram będzie zawierał same zera, będzie to dowodem na to, że GPU i CPU obliczyły dokładnie te same wartości. Można powiedzieć, że w pewnym sensie obliczymy różnicę tych dwóch histogramów. Kod źródłowy, którego użyjemy, jest do złudzenia podobny do wersji dla CPU, z tą różnicą, że zamiast operatora inkrementacji będzie zawierał operator dekrementacji:

```

// Weryfikacja histogramu przez CPU
for (int i=0; i<SIZE; i++)
    histo[buffer[i]]--;
for (int i=0; i<256; i++) {
    if (histo[i] != 0)
        printf( "Błąd w %d!\n", i );
}

```

Na zakończenie zwalniamy jak zwykle alokowane zdarzenia CUDA oraz pamięć GPU i hosta:

```

HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
cudaFree( dev_histo );
cudaFree( dev_buffer );
free( buffer );
return 0;
}

```

Wcześniej przyjęliśmy, że już uruchomiliśmy funkcję jądra, która obliczyła histogram, i kontynuowaliśmy rozważania dalej, tak jakby to rzeczywiście się zdarzyło. Zrobiliśmy to, gdyż tym razem ze względu na pewne aspekty związane z wydajnością funkcja ta będzie nieco bardziej skomplikowana niż zwykle. Ponieważ histogram zawiera 256 pojemników, to użycie 256 wątków na blok jest rozwiązaniem bardzo wygodnym i wydajnym. W rzeczywistości do wyboru

mamy sporo możliwości. Na przykład w 100 MB danych mamy 104 857 600 bajtów. W związku z tym moglibyśmy uruchomić jeden blok, którego każdy wątek przetwarzałby 409 600 elementów danych, albo utworzyć 409 600 bloków i mieć dla każdego elementu po jednym wątku.

Może się zapewne domyślasz, idealnym rozwiązaniem będzie coś pośredniego między tymi dwoma ekstremami. Przeprowadziliśmy kilka eksperymentów i doszliśmy do wniosku, że najlepszą wydajność uzyskuje się, gdy liczba uruchamianych bloków jest dwukrotnie większa od liczby wieloprocesorów w GPU. Na przykład procesor GeForce GTX 280 ma 30 wieloprocesorów, a więc najlepszą wydajność uzyskamy, wykonując naszą funkcję jądra w 60 równoległych blokach.

W rozdziale 3. pokazaliśmy metody sprawdzania różnych właściwości sprzętu. Możemy teraz umiejętnie wykorzystać, aby napisać program, który liczbę uruchamianych bloków będzie automatycznie dostosowywał do możliwości sprzętowych komputera. Użyjemy kodu, który jest pokazany poniżej. Chociaż jeszcze nie przedstawiliśmy implementacji jądra, nie powinno być problemu ze zrozumieniem dotychczasowych fragmentów.

```

cudaDeviceProp prop;
HANDLE_ERROR( cudaGetDeviceProperties( &prop, 0 ) );
int blocks = prop.multiProcessorCount;
histo_kernel<<<blocks*2,256>>>( dev_buffer, SIZE, dev_histo );

```

By pozbierać wszystko razem do kupy, przedstawiamy poniżej kod źródłowy funkcji main() całości:

```

int main( void ) {
    unsigned char *buffer =
        (unsigned char*)big_random_block( SIZE );
    cudaEvent_t start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );
    // Alokacja pamięci na GPU
    unsigned char *dev_buffer;
    unsigned int *dev_histo;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_buffer, SIZE ) );
    HANDLE_ERROR( cudaMemcpy( dev_buffer, buffer, SIZE,
                           cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_histo,
                           256 * sizeof( int ) ) );
    HANDLE_ERROR( cudaMemset( dev_histo, 0,
                           256 * sizeof( int ) ) );
    cudaDeviceProp prop;
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, 0 ) );
    int blocks = prop.multiProcessorCount;
    histo_kernel<<<blocks*2,256>>>( dev_buffer, SIZE, dev_histo );
    unsigned int histo[256];
}

```

```

HANDLE_ERROR( cudaMemcpy( histo, dev_histo,
                        256 * sizeof( int ),
                        cudaMemcpyDeviceToHost ) );
// Sprawdzenie czasu zakończenia oraz wyświetlenie wyniku pomiaru
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Czas wygenerowania: %3.1f ms\n", elapsedTime );
long histoCount = 0;
for (int i=0; i<256; i++) {
    histoCount += histo[i];
}
printf( "Suma histogramu: %ld\n", histoCount );
// Weryfikacja histogramu przez CPU
for (int i=0; i<SIZE; i++)
    histo[buffer[i]]--;
for (int i=0; i<256; i++) {
    if (histo[i] != 0)
        printf( "Błąd w %d!\n", i );
}
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
cudaFree( dev_histo );
cudaFree( dev_buffer );
free( buffer );
return 0;
}

```

FUNKCJA JĄDRA OBLCZAJĄCA HISTOGRAM ZA POMOCĄ OPERACJI ATOMOWYCH DLA PAMIĘCI GLOBALNEJ

Teraz będzie najlepsze — napiszemy kod obliczający histogram na GPU! Funkcji jądra, która oblicza histogram, należy przekazać wskaźnik na tablicę danych wejściowych, długość tej tablicy oraz wskaźnik na histogram wyjściowy. Pierwszą rzeczą, jaką funkcja ta musi obliczyć, jest liniowa pozycja w wejściowej tablicy danych. Wątki będą wykonywały pracę na pozycjach ponumerowanych od zera do liczby wątków pomniejszonej o jeden. Następnie po zakończeniu działania w tym miejscu każdy z nich przeskoczy tyle miejsc, ile jest uruchomionych wątków. W taki sam sposób sumowaliśmy wektory o dowolnej długości w jednym z wcześniejszych rozdziałów.

```

#include "../common/book.h"
#define SIZE    (100*1024*1024)
__global__ void histo_kernel( unsigned char *buffer,
                             long size,
                             unsigned int *histo ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1 );
        i += stride;
    }
}

```

Gdy każdy wątek wie, od którego miejsca i ma zacząć i o ile miejsc ma później przeskoczyć, program przegląda tablicę wejściową i zwiększa wartości w odpowiednich pojemnikach histogramu.

```

while (i < size) {
    atomicAdd( &(histo[buffer[i]]), 1 );
    i += stride;
}
}

```

Wyróżniony wiersz na powyższym listingu stanowi przykład użycia operacji atomowych. Wywołanie funkcji `atomicAdd(addr, y)`; powoduje wygenerowanie niepodzielnej sekwencji działań, które polegają na odczycie wartości zapisanej pod adresem `addr`, dodaniu do niej wartości `y` oraz zapisaniu wyniku z powrotem pod adresem `addr`. Mamy sprzętową gwarancję, że wykonywania tych czynności przez jeden wątek nie zakłóci żaden inny wątek, dzięki czemu wynik operacji jest przewidywalny. W przypadku tego programu adres, o którym mowa, wskazuje lokalizację pojemnika histogramu odpowiadającego aktualnie przetwarzanemu bajtowi. Jeśli bieżący bajt jest elementem `buffer[i]`, to odpowiadający mu pojemnik w histogramie jest elementem `histo[buffer[i]]`, podobnie jak w wersji dla CPU. Do wykonania operacji atomowej potrzebny jest adres tego pojemnika i dlatego pierwszym argumentem omawianej funkcji jest `&(histo[buffer[i]])`. A ponieważ naszym celem jest zwiększenie wartości w danym pojemniku o jeden, jako drugi argument przekazaliśmy liczbę 1.

Pomijając te wszystkie fikołki, algorytm dla GPU jest w istocie bardzo podobny do wersji dla CPU:

```

#include "../common/book.h"
#define SIZE    (100*1024*1024)
__global__ void histo_kernel( unsigned char *buffer,
                             long size,
                             unsigned int *histo ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1 );
        i += stride;
    }
}

```

Na razie jednak musimy powstrzymać się przed wiwatowaniem, ponieważ uruchamiając program na GPU GeForce GTX 285, stwierdziliśmy, że czas tworzenia histogramu dla 100 MB danych wyniósł aż 1,752 sekundy. Porównując ten wynik z rezultatem uzyskanym przez wersję dla CPU, musimy stwierdzić, że jest katastrofalny. Wersja dla GPU działa ponad cztery razy wolniej! Ale przecież właśnie po to mierzmy wydajność naszych algorytmów, aby takie sytuacje wykrywać. Trudno pogodzić się z tak słabym wynikiem tylko dlatego, że program działa na GPU.

Ponieważ w funkcji jądra znajduje się niewiele kodu, należy przypuszczać, że przyczyną niskiej wydajności są operacje atomowe na pamięci globalnej. Rzeczywiście, gdy kilka tysięcy wątków próbuje uzyskać dostęp do kilku miejsc w pamięci, występuje silne współzawodnictwo o dostęp do 256 pojemników. Aby operacje zwiększenia były niepodzielne, działania na tych samych adresach pamięci GPU musi wykonywać szeregowo. To może spowodować powstawanie długich kolejek zadań oczekujących i zniweczyć wszystkie osiągnięte zyski wydajności. Musimy poprawić ten algorytm.

FUNKCJA JĄDRA OBLICZAJĄCA HISTOGRAM ZA POMOCĄ OPERACJI ATOMOWYCH DLA PAMIĘCI GLOBALNEJ I WSPÓLNEJ

Co ciekawe, mimo że spowolnienie działania programu jest spowodowane zastosowaniem operacji atomowych, to aby go przyspieszyć, trzeba zastosować jeszcze więcej operacji tego typu. Problem po prostu dotyczył nie tyle samych operacji atomowych, ale tego, że bardzo duża ich liczba była wykonywana na względnie niewielkiej puli adresów pamięci. W ramach rozwiązania podzielimy algorytm na dwie części.

W pierwszej fazie każdy z bloków obliczy osobny histogram danych, który będzie przetwarzany przez jego wątki. Ponieważ każdy blok działa niezależnie od pozostałych, obliczenia te możemy wykonać w pamięci wspólnej, oszczędzając tym samym na czasie wysyłania danych do pamięci DRAM. Oczywiście nie zwalnia nas to od konieczności użycia operacji atomowych, ponieważ także w obrębie jednego bloku różne wątki mogą uzyskiwać dostęp do elementów o tych samych wartościach. Jednak w tym przypadku tylko 256 wątków będzie współzawodniczyć o dostęp do 256 adresów, a więc zator będzie znacznie mniejszy niż w poprzedniej wersji algorytmu.

W związku z tym pierwsza faza algorytmu będzie polegała na alokowaniu i wyzerowaniu w pamięci wspólnej bufora do przechowywania histogramu każdego z bloków. Przypomnijmy z rozdziału 5., że skoro następną czynnością będzie odczyt i modyfikacja tego bufora, musimy użyć funkcji `_syncthreads()`, aby zagwarantować zakończenie działania przez wszystkie wątki przed przejściem do dalszych działań:

```
__global__ void histo_kernel( unsigned char *buffer,
                             long size,
                             unsigned int *histo ) {
    __shared__ unsigned int temp[256];
    temp[threadIdx.x] = 0;
    __syncthreads();
```

Kolejny krok po wyzerowaniu histogramu będzie bardzo podobny do pierwotnej wersji. Dwie główne różnice między tamtym kodem a obecnym będą polegały na tym, że teraz zamiast bufora alokowanego w pamięci globalnej `histo[]` użyjemy bufora alokowanego w pamięci wspólnej o nazwie `temp[]` oraz że użyjemy wspomnianej przed chwilą funkcji `_syncthreads()`.

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
int offset = blockDim.x * gridDim.x;
while (i < size) {
    atomicAdd( &temp[buffer[i]], 1);
    i += offset;
}
__syncthreads();
```

Na koniec musimy połączyć wszystkie histogramy częściowe w jednym globalnym buforze `histo[]`. Przypuśćmy, że podzieliśmy dane wejściowe na dwie części, tak że mamy dwa wątki, których każdy przetwarza inną połowę i oblicza inny histogram. Jeśli w danych wejściowych wątek A widzi bajt 0xFC 20 razy, a wątek B widzi go 5 razy, to wiadomo, że bajt 0xFC wystąpił w zbiorze danych wejściowych w sumie 25 razy. Analogicznie wartość każdego pojemnika w histogramie finalnym stanowi sumę wartości odpowiednich pojemników w histogramach wątków A i B. Ma to zastosowanie bez względu na liczbę wątków, a więc żeby obliczyć ostateczny wynik, należy wartości z wszystkich histogramów częściowych dodać do odpowiednich wartości jednego wspólnego histogramu finalnego. Wiemy już, że operacja ta musi zostać wykonana atomowo.

```
atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
}
```

Ponieważ utworzyliśmy 256 wątków i histogram składa się z 256 pojemników, każdy wątek samodzielnie dodaje jeden pojemnik do histogramu finalnego. Gdyby te liczby się ze sobą nie zgadzały, to obliczenia tego etapu algorytmu byłyby o wiele bardziej skomplikowane. Zwróć uwagę, że nie wiadomo, w jakiej kolejności bloki będą dodawać swoje wartości do histogramu finalnego. Nie jest to jednak problemem, gdyż dodawanie jest działaniem przemiennym, więc wszystko będzie w porządku, dopóki operacje te będą wykonywane atomowo.

Teraz kończy naszą funkcję jądra. Poniżej przedstawiamy jej kod źródłowy w całości:

```
__global__ void histo_kernel( unsigned char *buffer,
                             long size,
                             unsigned int *histo ) {
    __shared__ unsigned int temp[256];
    temp[threadIdx.x] = 0;
    __syncthreads();
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int offset = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &temp[buffer[i]], 1);
        i += offset;
    }
    __syncthreads();
    atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
```

Wydajność tej wersji algorytmu jest znacznie lepsza niż poprzedniej. Na GPU GeForce GTX 285 czas wykonywania programu skrócił się do 0,057 sekundy. Wynik ten bije implementację dla CPU na głowę, bo aż o rząd wielkości (0,057 w porównaniu z 0,416). Jest to ponad siedmiokrotne przyspieszenie, a więc należy wysnuć wniosek, że mimo przejściowych problemów użycie operacji atomowych na pamięci wspólnej i globalnej zakończyło się pełnym sukcesem.

9.5. Podsumowanie

Przez cały czas podkreślaliśmy, jak bardzo język CUDA upraszcza programowanie równoległe, przemilczeliśmy jednak fakt, że w niektórych przypadkach może się to obrócić przeciwko nam. Jedną z najczęściej występujących sytuacji, w których możliwości równoległego wykonywania wielu fragmentów kodu przez GPU komplikują pracę programisty, jest modyfikacja tych samych miejsc w pamięci przez wiele wątków. Na szczęście są operacje atomowe, które pomagają rozwiązać te problemy.

Niestety jednak, jak przekonaliśmy się w tym rozdziale, czasami użycie operacji atomowych powoduje duże spadki wydajności, których można się pozbyć jedynie poprzez rekonstrukcję całego algorytmu. W programie obliczającym histogram zmieniliśmy metodę wykonywania obliczeń na dwuetapową, co pozwoliło nam zmniejszyć zatory w dostępie do pamięci globalnej. Metoda ta, polegająca na zmniejszeniu kolejek do pamięci, często przynosi dobre efekty i dlatego warto o niej pamiętać, gdy ma się zamiar stosować operacje atomowe.

Rozdział 10

Strumienie

Do tej pory wielokrotnie już wykazywaliśmy, że procesory GPU w porównaniu z analogicznym modelem dla procesorów CPU pozwalają radykalnie przyspieszyć działanie algorytmów. Do tej pory jednak nie napisaliśmy jeszcze ani słowa o innej technice zrównoleglania dostępnej w procesorach GPU NVIDIA. Jest to metoda podobna do **zrównoleglania na poziomie zadań** charakterystyczna dla programowania procesorów CPU. W odróżnieniu od zrównoleglania na poziomie danych, w którym chodzi o wykonywanie tej samej funkcji na dużej porcji danych, zrównoleglanie na poziomie zadań polega na jednoczesnym wykonywaniu wielu całkowicie różnych działań.

W kontekście zrównoleglania **zadanie** może mieć różne definicje. Na przykład można napisać program wykonujący dwa zadania: rysowanie graficznych elementów interfejsu użytkownika za pomocą jednego wątku i pobieranie aktualizacji z internetu za pomocą drugiego wątku. Zadania te są wykonywane równolegle, mimo że nie mają ze sobą nic wspólnego. Techniki zrównoleglania na poziomie zadań w GPU może i nie są tak doskonałe jak w CPU, ale w niektórych sytuacjach ich zastosowanie pozwala wycisnąć z procesora jeszcze więcej. W tym rozdziale opisujemy strumienie CUDA i pokazujemy, jak za ich pomocą można wykonywać pewne operacje równolegle na GPU.

0.1. Streszczenie rozdziału

W tym rozdziale:

• **Nauczysz się alokować pamięć hosta z zablokowanym stronicowaniem.**

• **Dowiesz się, czym są strumienie w CUDA.**

• **Nauczysz się używać strumieni CUDA w celu przyspieszenia działania swoich programów.**

10.2. Pamięć hosta z zablokowanym stronicowaniem

We wszystkich poprzednich rozdziałach do alokacji pamięci na GPU używaliśmy funkcji `cudaMalloc()`. Natomiast na hoście do tego samego celu używaliśmy standardowej funkcji języka C o nazwie `malloc()`. Nie wspomnieliśmy jednak, że w CUDA dostępny jest także mechanizm pozwalający alokować pamięć na hoście, który ma postać funkcji `cudaHostAlloc()`. Po co mielibyśmy używać tej funkcji, skoro do tej pory funkcja `malloc()` służyła nam bardzo dobrze?

Między pamięcią alokowaną za pomocą funkcji `malloc()` a pamięcią alokowaną przez funkcję `cudaHostAlloc()` jest zasadnicza różnica. Funkcja języka C alokuje standardową stronicowaną pamięć hosta, natomiast funkcja CUDA alokuje na hoście bufor z **wyłączonym stronicowaniem**. Pamięć taka ma jedną ważną cechę: system operacyjny nie może przenieść jej zawartości na dysk, co oznacza, że będzie ona zawsze fizycznie w tym samym miejscu. Dzięki temu system operacyjny może bezpiecznie zezwalać na dostęp do fizycznego adresu tej pamięci, ponieważ nie ma obawy, że bufor ten zostanie nagle gdzieś przeniesiony.

Ponieważ procesor GPU zna fizyczny adres bufora, może kopiować dane z niego i do niego, korzystając z techniki bezpośredniego dostępu do pamięci (ang. *direct memory access* — DMA). Ponieważ operacje kopiowania tą techniką są wykonywane bez udziału CPU, procesor ten teoretycznie może w tym samym czasie realokować kopiowane bufore albo przenieść je na dysk. Dlatego tak ważne jest, aby w tych przypadkach używać pamięci z wyłączonym stronicowaniem. W rzeczywistości sterownik CUDA do kopiowania danych do GPU stosuje technikę DMA nawet wówczas, gdy do wykonania tej operacji używa się pamięci stronicowanej. W efekcie operacja kopiowania jest wykonywana dwukrotnie. Najpierw ze stronicowanego bufora systemowego do bufora z zablokowanym stronicowaniem, a następnie dopiero z tego zablokowanego bufora do GPU.

Oznacza to, że szybkość operacji kopiowania danych z pamięci stronicowanej jest ograniczona szybkością interfejsu PCIE lub magistrali FSB, w zależności od tego, która z nich jest mniejsza. Duże różnice przepustowości tych dwóch magistral w niektórych systemach sprawiają, że operacje kopiowania danych między hostem a GPU za pomocą pamięci z zablokowanym stronicowaniem są około dwa razy szybsze niż przy użyciu pamięci stronicowanej. Ale gdyby nawet magistrale FSB i PCI Express miały identyczne szybkości, użycie buforów stronicowanych i tak powodowałoby pewien narzut związany z niepotrzebnym kopiowaniem danych za pomocą CPU.

Nie oznacza to jednak, że powinieneś teraz zamienić wszystkie wywołania funkcji `malloc()` na `cudaHostAlloc()`, ponieważ blokada pamięci oznacza jednocześnie rezygnację ze wszystkich chomiony jest program korzystający z zablokowanej pamięci, musi zawierać wystarczającą ilość pamięci fizycznej, aby pomieścić w niej wszystkie zablokowane bufore. Z tego powodu programy takie znacznie szybciej zużywają pamięć niż te, w których używana jest funkcja `malloc()`. Oznacza to, że mogą one nie tylko zająć całą pamięć, jeśli jest jej niezbyt dużo, ale również spowolnić działanie innych programów.

Nie piszemy tego po to, aby odwieść Cię od używania funkcji `cudaHostAlloc()`, lecz po to, by Cię uświadomić, jakie ryzyko niesie ze sobą jej używanie. Naszym zdaniem najlepiej jest alokować w ten sposób pamięć przeznaczoną jako źródło lub cel wywołań funkcji `cudaMemcpy()` i zwalniać ją natychmiast, gdy stanie się niepotrzebna, zamiast czekać na zakończenie programu. Korzystanie z funkcji `cudaHostAlloc()` samo w sobie nie jest trudniejsze niż używanie innych funkcji, które już znasz. Poniżej przedstawiamy program, na przykładzie którego zademonstrujemy techniki użycia pamięci z zablokowanym stronicowaniem i pokażemy, ile dzięki nim można zyskać.

Jest to bardzo prosty program, którego głównym zadaniem będzie porównanie wydajności funkcji `cudaMemcpy()` działającej na pamięci stronicowanej i na pamięci z wyłączonym stronicowaniem. Alokujemy po jednym buforze o takim samym rozmiarze na hoście i GPU, a następnie przeprowadzimy między nimi kilka operacji kopiowania. Pozwolimy użytkownikowi wybrać kierunek kopiowania — z hosta do GPU (up) lub z GPU do hosta (w przeciwnym razie). Pomiaru czasu wykonywania operacji dokonamy za pomocą zdarzeń CUDA. Powinieneś pamiętać z poprzednich rozdziałów, jak to się robi, ale jeśli zapomniałeś, poniżej znajduje się krótkie przypomnienie:

```
float cuda_malloc_test( int size, bool up ) {
    cudaEvent_t      start, stop;
    int              *a, *dev_a;
    float            elapsedTime;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    a = (int*)malloc( size * sizeof( *a ) );
    HANDLE_NULL( a );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                           size * sizeof( *dev_a ) ) );
```

Zarówno na hoście, jak i na GPU, niezależnie od kierunku kopiowania, alokujemy po buforze o rozmiarze `size` liczb całkowitych. Następnie wykonujemy 100 kopii w kierunku określonym przez argument `up` i po zakończeniu kopiowania zatrzymujemy odmierzanie czasu:

```
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
for (int i=0; i<100; i++) {
    if (up)
        HANDLE_ERROR( cudaMemcpy( dev_a, a,
                               size * sizeof( *dev_a ), cudaMemcpyHostToDevice ) );
    else
        HANDLE_ERROR( cudaMemcpy( a, dev_a,
                               size * sizeof( *dev_a ), cudaMemcpyDeviceToHost ) );
}
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
```

Po zakończeniu kopowania zwalniamy oba bufora i usuwamy zdarzenia użyte do pomiaru czasu:

```
free( a );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
return elapsedTime;
}
```

Funkcja `cuda_malloc_test()` alokuje stronicowaną pamięć hosta za pomocą standardowej funkcji `malloc()` języka C. Natomiast wersja korzystająca z pamięci zablokowanej alokuje bufor przy użyciu funkcji `cudaHostAlloc()`.

```
float cuda_host_alloc_test( int size, bool up ) {
    cudaEvent_t      start, stop;
    int              *a, *dev_a;
    float            elapsedTime;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&a,
                                size * sizeof( *a ),
                                cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                            size * sizeof( *dev_a ) ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );
    for (int i=0; i<100; i++) {
        if (up)
            HANDLE_ERROR( cudaMemcpy( dev_a, a,
                                    size * sizeof( *a ),
                                    cudaMemcpyHostToDevice ) );
        else
            HANDLE_ERROR( cudaMemcpy( a, dev_a,
                                    size * sizeof( *a ),
                                    cudaMemcpyDeviceToHost ) );
    }
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                       start, stop ) );
    HANDLE_ERROR( cudaFreeHost( a ) );
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );
    return elapsedTime;
}
```

Jak widać, sposób użycia bufora alokowanego za pomocą funkcji `cudaHostAlloc()` jest taki sam jak bufora alokowanego przez funkcję `malloc()`. Jednak w wywołaniu funkcji `cudaHostAlloc()` jako ostatni argument wywołania została wpisana wartość `cudaHostAllocDefault`. Służy on do

definiowania specjalnych znaczników, za pomocą których można zmienić działanie tej funkcji, tak aby alokowała różne rodzaje pamięci zablokowanej. Pozostałe wartości tego argumentu poznasz w następnym rozdziale, a teraz zadowolimy się ustaleniem domyślnym, czyli wartością `cudaHostAllocDefault`. Do zwalniania pamięci alokowanej za pomocą funkcji `cudaHostAlloc()` służy funkcja `cudaFreeHost()`. Innymi słowy, każdemu wywołaniu funkcji `malloc()` musi towarzyszyć wywołanie funkcji `free()`, a każdemu wywołaniu funkcji `cudaHostAlloc()` musi towarzyszyć wywołanie funkcji `cudaFreeHost()`.

W treści funkcji `main()` nie ma nic zaskakującego:

```
#include "../common/book.h"
#define SIZE      (10*1024*1024)
int main( void ) {
    float      elapsedTime;
    float      MB = (float)100*SIZE*sizeof(int)/1024/1024;
    elapsedTime = cuda_malloc_test( SIZE, true );
    printf( "Czas działania przy użyciu funkcji cudaMalloc: %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s podczas kopiowania z hosta: %3.1f\n",
            MB/(elapsedTime/1000) );
```

Ponieważ w miejsce parametru `up` funkcji `cuda_malloc_test()` wpisaliśmy wartość `true`, powyższe wywołanie pozwoli nam sprawdzić czas wykonywania kopii z hosta do GPU. Aby wykonać te same czynności w odwrotnym kierunku, wpisujemy te same wywołania, ale jako drugi argument wywołania zapiszemy `false`:

```
elapsedTime = cuda_malloc_test( SIZE, false );
printf( "Czas działania przy użyciu funkcji cudaMalloc: %3.1f ms\n",
        elapsedTime );
printf( "\tMB/s podczas kopiowania do hosta: %3.1f\n",
        MB/(elapsedTime/1000) );
```

W podobny sposób możemy zmierzyć wydajność wersji z funkcją `cudaHostAlloc()`. Wywołujemy funkcję `cuda_host_alloc_test()` dwukrotnie: z parametrem `up` ustawionym na `true` i potem na `false`.

```
elapsedTime = cuda_host_alloc_test( SIZE, true );
printf( "Czas działania przy użyciu funkcji cudaHostAlloc: %3.1f ms\n",
        elapsedTime );
printf( "\tMB/s podczas kopiowania z hosta: %3.1f\n",
        MB/(elapsedTime/1000) );
elapsedTime = cuda_host_alloc_test( SIZE, false );
printf( "Czas działania przy użyciu funkcji cudaHostAlloc: %3.1f ms\n",
        elapsedTime );
printf( "\tMB/s podczas kopiowania z hosta: %3.1f\n",
        MB/(elapsedTime/1000) );
```

Na GPU GeForce GTX 285 stwierdziliśmy, że dzięki użyciu pamięci z zablokowanym stronicowaniem szybkość kopiowania danych z hosta do GPU wzrosła z 2,77 GB/s do 5,11 GB/s. Przy kopiowaniu w drugą stronę, z GPU do hosta, liczby te wyniosły odpowiednio 2,43 GB/s i 5,46 GB/s. Można zatem wyciągać wniosek, że użycie pamięci zablokowanej pozwala uzyskać znaczny wzrost wydajności. Ale pamięć z zablokowanym stronicowaniem służy nie tylko do poprawiania wydajności. W niektórych sytuacjach jej użycie jest wręcz obowiązkowe. Szczególny w następnych podrozdziałach.

10.3. Strumienienie CUDA

W rozdziale 6., zawierającym opis zdarzeń CUDA, poznaliśmy funkcję `cudaEventRecord()`. Szczegółowy opis jej drugiego argumentu funkcji służącej do określania **strumienia**, do którego ma być wstawione dane zdarzenie, odłożyliśmy na później.

```
cudaEvent_t start;
cudaEventCreate(&start);
cudaEventRecord( start, 0 );
```

Strumienie CUDA mogą odgrywać bardzo ważną rolę dla wydajności aplikacji. **Strumień CUDA** to kolejka operacji GPU, które mają zostać wykonane w określonej kolejności. Można do niego wstawać wywołania jądra, operacje kopiowania pamięci, a także zdarzenia początkowe i końcowe. Kolejność ich wstawiania decyduje o kolejności wykonywania. Strumieniem można traktować jako swego rodzaju zadania dla GPU, które można wykonywać równolegle. Najpierw pokazemy Ci, jak używać strumieni, a potem, w jaki sposób można za ich pomocą przyspieszyć działanie programów.

Tak naprawdę korzyści ze strumieni można odnieść wyłącznie wtedy, gdy używa się więcej niż jednego z nich. Jednak podstawowe koncepcje z nimi związane objaśnimy na podstawie programu wykorzystującego tylko jeden strumień. Przypuśćmy, że mamy funkcję jądra w języku CUDA C, która przyjmuje na wejściu dwa bufore danych, a i b, a następnie wykonuje jakieś działania na ich wartościach i tworzy bufor wyjściowy o nazwie c. Podobnie działał program obliczający sumę wektorów, ale tym razem obliczamy tylko średnią trzech wartości z bufora a i trzech wartości z bufora b:

```
if ( idx < N ) {
    int idx1 = (idx + 1) % 256;
    int idx2 = (idx + 2) % 256;
    float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
    float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
    c[idx] = (as + bs) / 2;
}
```

Sam sposób działania tej funkcji jest teraz dla nas mało ważny. Jeśli więc nie całkiem chwytasz, o co w niej chodzi, to nie marnuj czasu na jałowe głowienie się. Służy nam ona tylko jako wypełniacz miejsca, a to, co nas interesuje w tym przypadku, czyli strumienie, znajduje się w funkcji `main()`:

```
int main( void ) {
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
    if ( !prop.deviceOverlap ) {
        printf( "Urządzenie nie obsługuje przepłatania, a więc "
               "użycie strumieni nie spowoduje przyspieszenia \n" );
    }
    return 0;
}
```

Na początku wybieramy urządzenie i sprawdzamy, czy jest wyposażone w funkcję, zwaną **device overlap** (przepłatanie urządzeń). GPU, które ją mają, pozwalają na jednocześnie wykonywanie funkcji jądra CUDA C i kopowanie danych między urządzeniem a hostem. Zgodnie z obietnicą pokażemy wkrótce, jak można symulować taki przepłot za pomocą kilku strumieni, ale najpierw nauczymy Cię tworzyć pojedynczy strumień. Jak zawsze, gdy chcemy zmierzyć wydajność programu, pisanie kodu rozpoczynamy od utworzenia zdarzeń początku i zakończenia pomiaru:

```
cudaEvent_t start, stop;
float elapsedTime;
// Uruchomienie czasomierza
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

Po uruchomieniu stopera tworzymy strumień:

```
// Inicjacja strumienia
cudaStream_t stream;
HANDLE_ERROR( cudaStreamCreate( &stream ) );
__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

Żeby utworzyć strumień, wystarczy taki prosty kod. Nie ma nawet sensu rozwodzić się nad jego znaczeniem, więc od razu przechodzimy do alokacji danych:

```
int *host_a, *host_b, *host_c;
int *dev_a, *dev_b, *dev_c;
// Alokacja pamięci na GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
                           N * sizeof(int) ) );
// Alokacja pamięci z zablokowanym stronicowaniem do użycia przez strumień
HANDLE_ERROR( cudaHostAlloc( (void**)&host_a,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_b,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_c,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );
for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}
```

lokowaliśmy bufore wyjściowy i wyjściowy zarówno na GPU, jak i na hostie. Zwróci uwagę, że tym drugim przypadku użyliśmy funkcji `cudaHostAlloc()`, co oznacza, że będziemy korzystać z pamięci zablokowanej. Zrobiliśmy to nie tylko dlatego, aby szybciej wykonywać kopowanie innych, lecz również dlatego, że za chwilę użyjemy nowego rodzaju funkcji `cudaMemcpy()`, która wymaga właśnie, aby pamięć hosta miała wyłączone stronicowanie. Po dokonaniu alokacji buforów wyjściowych wypełniamy pamięć alokowaną na hostie losowymi liczbami całkowitymi pomocy funkcji `rand()` z biblioteki języka C.

amy już utworzone zdarzenia do mierzenia czasu, strumień oraz bufore na hostie i w urządzeniu, a więc możemy rozpoczęć wykonywanie obliczeń! Zazwyczaj w tym miejscu kopujemy bufor wyjściowy na GPU, uruchamiamy funkcję jądra, a po zakończeniu jej działania wprowadzamy bufor wyjściowy z powrotem do hosta. Również i tym razem będziemy działać pozbawieni, lecz z drobnymi modyfikacjami.

pierwsze nie będziemy kopować buforów wejściowych do GPU w całości, lecz podzielimy na mniejsze części i opisany trzyetapowy proces wykonamy na każdej z nich z osobna. Innymi słowy, weźmiemy niewielką część buforów wejściowych, skopiujemy ją do GPU, wykonamy na niej funkcję jądra, a następnie skopiujemy odpowiednią część bufora wyjściowego z powrotem do hosta. Wyobraź sobie, że nie możemy skopiować całości naraż, ponieważ GPU ma znacznie więcej pamięci niż host i po prostu cała zawartość buforów może się w niej nie zmieścić. Oto d

```
// Przeglądanie całego zbioru danych po kawałku
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    // Asynchroniczne skopiowanie zablokowanej pamięci do GPU
    HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i,
                                  N * sizeof(int),
                                  cudaMemcpyHostToDevice,
                                  stream ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b, host_b+i,
                                  N * sizeof(int),
                                  cudaMemcpyHostToDevice,
                                  stream ) );
    kernel1<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );
    // Skopiowanie danych z urządzenia do zablokowanej pamięci
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c,
                                  N * sizeof(int),
                                  cudaMemcpyDeviceToHost,
                                  stream ) );
}
```

W kodzie tym znajdują się jeszcze dwa nietypowe fragmenty. Po pierwsze, do kopowania danych między GPU i hostem została użyta nowa funkcja o nazwie `cudaMemcpyAsync()` (zamiast dobrze nam znanej funkcji `cudaMemcpy()`). Różnica między tymi dwiema konstrukcjami jest wprawdzie subtelną, ale bardzo znaczącą. Pierwsza działa podobnie jak standardowa funkcja języka C o nazwie `memcpy()`, czyli jest wykonywana synchronicznie. Oznacza to, że po zakończeniu jej działania kopowanie jest zakończone i bufor wyjściowy zawiera całą treść, która miasta się w nim znaleźć.

Przeciwieństwem funkcji synchronicznej jest funkcja `asynchronous`, np. `cudaMemcpyAsync()`, która stanowi żądanie skopiowania zawartości pamięci do strumienia podanego w argumencie `stream`. Gdy ta funkcja zakończy działanie, wcale nie jest pewne, że kopowanie zostało zakończone, a wręcz nie wiadomo nawet, czy w ogóle już się rozpoczęło. Wiadomo jedynie, że kopowanie zostanie na pewno przeprowadzone przed wykonaniem następnej operacji wstawionej do tego samego strumienia. Bardzo ważne jest, aby wszelkie wskaźniki na pamięć hosta przekazywane do funkcji `cudaMemcpyAsync()` wskazywały pamięć alokowaną za pomocą funkcji `cudaHostAlloc()`. Innymi słowy, asynchroniczne kopowanie można planować tylko w przypadku kopowania do i z pamięcią z zablokowanym stronicowaniem.

Zwrócić uwagę, że opcjonalny argument strumieniowy może pojawić się także w wywołaniu funkcji jądra w nawiąsach trójkątnych. Wywołanie jądra w tym kodzie jest asynchroniczne, podobnie jak wcześniejsze dwie operacje kopowania do GPU i dalsze kopowanie z GPU. W zasadzie pętla ta może zakończyć iterację, nawet nie rozpoczynając operacji kopowania ani wykonywania jądra. Jak już pisaliśmy, jedynie, co wiemy na pewno, jest to, że pierwsza operacja kopowania wstawiona do strumienia zostanie wykonana przed drugą. Ponadto drugie kopowanie zostanie zakończone przed uruchomieniem funkcji jądra, a jądro zakończy działanie, zanim rozpocznie się trzecia operacja kopowania. Krótko mówiąc, strumień jest jak uporządkowana kolejka zadań do wykonania dla GPU.

zakończeniu iteracji pętli `for()` GPU nadal może mieć jeszcze dużo pracy do wykonania. Chcesz zagwarantować zakończenie przez GPU obliczeń i operacji kopiowania, musisz go synchronizować z hostem. Krótko mówiąc, nakazujesz procesorowi CPU, aby siedział cicho, aż GPU skończy działanie. Do tego służy funkcja o nazwie `cudaStreamSynchronize()`jmująca jako argument strumień, na który chcemy poczekać:

```
// Skopiowanie wyniku z pamięci zablokowanej do pełnego bufora
HANDLE_ERROR( cudaStreamSynchronize( stream ) );
```

Po zsynchronizowaniu strumienia `stream` z hostem wiemy, kiedy obliczenia i operacje kopiowania zostaną zakończone, możemy zatrzymać stoper, pobrać dane dotyczące wydajności i liczyć wszystkie bufore.

```
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Czas działania: %3.1f ms\n", elapsedTime );
// Zwolnienie strumieni i pamięci
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );
```

Zakończenie przed zamknięciem programu usuwamy strumień, którego używaliśmy do realizacji zadań dla GPU:

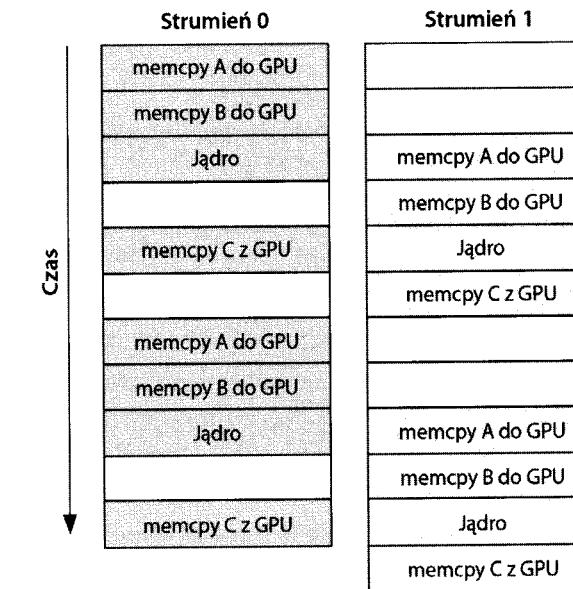
```
HANDLE_ERROR( cudaStreamDestroy( stream ) );
return 0;
```

W związku z tym, ten przykład nie pozwala dostrzec w pełni możliwości, jakie daje użycie wielu strumieni. Oczywiście nawet jeden strumień może przyczynić się do przyspieszenia aplikacji, jeśli będzie wykonywał jakieś operacje na hoście, podczas gdy GPU poci się nad przetwarzaniem strumienia zestawem zadań. Za pomocą strumieni można jednak przyspieszyć nawet tych programów, w których host ma niewiele do roboty. Tematowi temu poświęcamy kolejny podrozdział.

Użycie wielu strumieni CUDA

Zmienimy program z poprzedniego podrozdziału i zmodyfikujemy go tak, aby działał dwóch strumieni. Na początku tego programu sprawdziliśmy, czy urządzenie, którego korzystać, obsługuje tzw. **przeplatanie**, i podzieliliśmy obliczenia na części.

Udoskonalenia, jakie wprowadzimy w nowej wersji programu, są proste i bazują na dwóch elementach: „częściowych” obliczeniach i przeplataniu kopiowania danych z pamięci z wykonywaniem funkcji jądra. Będziemy chcieli, aby strumień nr 1 kopiował swoje bufore wejściowe do GPU, a strumień 0 wykonywał w tym czasie funkcję jądra. Następnie strumień 1 będzie wykonywał funkcję jądra, a strumień 0 w tym czasie będzie kopiował wyniki do hosta. Później strumień 1 skopiuje wyniki do hosta, a strumień 0 rozpocznie wykonywanie funkcji jądra na następnej części danych. Przy założeniu, że operacje kopiowania i wykonywania jądra zajmują mniej więcej po tyle samo czasu, o czasu wykonywania naszego programu może przedstawiać się tak jak na rysunku 10.1. Na ilustracji przyjęto założenie, że GPU może wykonywać operację kopiowania i wykonywania jądra jednocześnie, a wiec puste pola oznaczają momenty, gdy jeden strumień czeka na wykonanie operacji, której nie może przeplaść z operacją drugiego strumienia. I jeszcze drobna uwaga: na tym oraz dalszych rysunkach nazwę funkcji `cudaMemcpyAsync()` skróciliśmy dla wygody do formy `memcpy`.



Rysunek 10.1. Przewidywana oś czasu wykonywania programu za pomocą dwóch strumieni

Tak naprawdę to oś czasu może być nawet jeszcze bardziej korzystna, ponieważ niektóre GPU NVIDIA obsługują równoczesne wykonywanie funkcji jądra i przeprowadzanie dwóch operacji kopiowania danych, po jednej do i z urządzenia. Jednak wystarczy zwykłe przeplatanie operacji wykonywania jądra i operacji kopiowania danych, aby zauważycy znaczny wzrost wydajności programu, gdy używa się strumieni.

Mimo naszych wielkich planów kod jądra pozostaje bez zmian:

```
#include "../common/book.h"
#define N    (1024*1024)
#define FULL_DATA_SIZE (N*20)
```

```

__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2
    }
}

```

Podobnie jak w wersji z jednym strumieniem, najpierw sprawdzamy, czy urządzenie obsługuje przeplatanie obliczeń kopiowaniem. Jeśli tak, kontynuujemy tak jak poprzednio, czyli tworzymy zdarzenia służące do pomiaru czasu działania programu:

```

int main( void ) {
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
    if (!prop.deviceOverlap) {
        printf( "Urządzenie nie obsługuje przeplatania, a więc "
               "użycie strumieni nie spowoduje przyspieszenia \n" );
        return 0;
    }
    cudaEvent_t start, stop;
    float elapsedTime;
    // Uruchomienie czasomierzy
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );
}

```

Następnie tworzymy dwa strumienie, dokładnie w taki sam sposób jak wcześniej tworzyliśmy len:

```

// Inicjacja strumieni
cudaStream_t stream0, stream1;
HANDLE_ERROR( cudaStreamCreate( &stream0 ) );
HANDLE_ERROR( cudaStreamCreate( &stream1 ) );

```

dalej podtrzymujemy założenie, że mamy dwa bufora wejściowe i jeden bufor wyjściowy na cie. Bufory wejściowe napełniamy losowymi liczbami całkowitymi, dokładnie tak jak poprzednio. Ponieważ jednak tym razem dane będą przetwarzane przez dwa strumienie, na GPU tworzymy dwa identyczne zestawy buforów, po jednym dla każdego strumienia.

```

int *host_a, *host_b, *host_c;
int *dev_a0, *dev_b0, *dev_c0; // Bufory na GPU dla strumienia 0
int *dev_a1, *dev_b1, *dev_c1; // Bufory na GPU dla strumienia 1

```

```

// Alokacja pamięci na GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a0,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b0,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c0,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_a1,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b1,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c1,
                           N * sizeof(int) ) );
// Alokacja pamięci z zablokowanym stronicowaniem dla strumieni
HANDLE_ERROR( cudaHostAlloc( (void**)&host_a,
                            FULL_DATA_SIZE * sizeof(int),
                            cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_b,
                            FULL_DATA_SIZE * sizeof(int),
                            cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_c,
                            FULL_DATA_SIZE * sizeof(int),
                            cudaHostAllocDefault ) );
for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}

```

Następnie za pomocą pętli przeglądamy kolejne części danych wejściowych, dokładnie tak jak poprzednio. Lecz tym razem dzięki użyciu dwóch strumieni w każdej iteracji przetwarzamy dwa razy więcej danych. W strumieniu 0 kolejkujemy zadania asynchronicznego kopiowania buforów a i b do GPU, wykonanie funkcji jądra oraz skopiowanie wyniku z powrotem do bufora c:

```

// Iteracja po kawałku przez pełne dane
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // Asynchroniczne skopiowanie zablokowanej pamięci do urządzenia
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,
                                  N * sizeof(int),
                                  cudaMemcpyHostToDevice,
                                  stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,
                                  N * sizeof(int),
                                  cudaMemcpyHostToDevice,
                                  stream0 ) );
    kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
    // Skopiowanie danych z urządzenia do pamięci zablokowanej
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,
                                  N * sizeof(int),
                                  cudaMemcpyDeviceToHost,
                                  stream0 ) );
}

```

Po skonstruowaniu kolejki dla strumienia 0 przechodzimy do zrobienia dokładnie tego samego dla kolejnej porcji danych w strumieniu 1:

```
// Asynchroniczne skopiowanie zablokowanej pamięci do urządzenia
HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N,
                               N * sizeof(int),
                               cudaMemcpyHostToDevice,
                               stream1 ) );
HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N,
                               N * sizeof(int),
                               cudaMemcpyHostToDevice,
                               stream1 ) );
kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
// Skopiowanie danych z urządzenia do pamięci zablokowanej
HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1,
                               N * sizeof(int),
                               cudaMemcpyDeviceToHost,
                               stream1 ) );
}
```

Pętla `for()` w kolejnych iteracjach kolejkuje poszczególne porcje danych raz dla jednego, raz dla drugiego strumienia i robi to aż do wyczerpania zapasów. Po zakończeniu jej działania synchronizujemy GPU z CPU, a następnie zatrzymujemy stoper. Ponieważ używamy dwóch strumieni, musimy zsynchronizować oba:

```
HANDLE_ERROR( cudaStreamSynchronize( stream0 ) );
HANDLE_ERROR( cudaStreamSynchronize( stream1 ) );
```

Na koniec dodajemy funkcję `main()` w takiej samej postaci jak w programie z jednym strumieniem. Zatrzymujemy stopery, wyświetlamy wynik pomiaru czasu, a następnie porządkujemy po sobie. Oczywiście pamiętamy, że tym razem musimy usunąć dwa strumienie i zwolnić dwa razy więcej buforów GPU, ale oprócz tych drobnych różnic reszta kodu jest identyczna jak w poprzedniej implementacji:

```
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Czas działania: %3.1f ms\n", elapsedTime );
// Zwolnienie strumieni i pamięci
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a0 ) );
HANDLE_ERROR( cudaFree( dev_b0 ) );
HANDLE_ERROR( cudaFree( dev_c0 ) );
HANDLE_ERROR( cudaFree( dev_a1 ) );
HANDLE_ERROR( cudaFree( dev_b1 ) );
HANDLE_ERROR( cudaFree( dev_c1 ) );
```

```
HANDLE_ERROR( cudaStreamDestroy( stream0 ) );
HANDLE_ERROR( cudaStreamDestroy( stream1 ) );
return 0;
}
```

Porównaliśmy wydajność obu wersji programu na GPU GeForce GTX 285. Czas działania wersji z jednym strumieniem wyniósł 62 ms, natomiast wersji dwustrumieniowej — 61 ms.

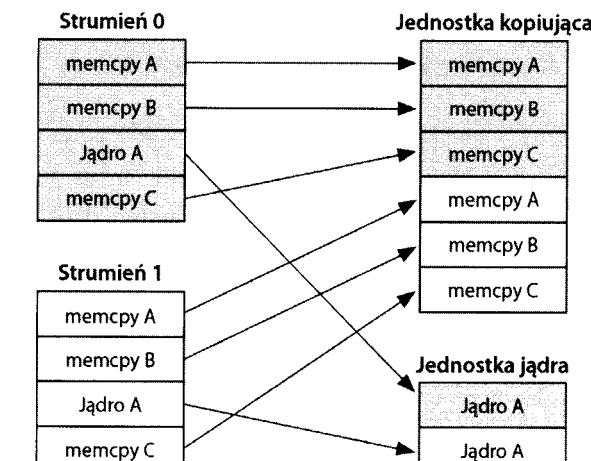
Coś chyba jest nie tak!

Na pocieszenie powiemy, że właśnie dlatego mierzymy szybkość działania naszych programów. Czasami mimo naszych najszczerzych chęci otrzymujemy tylko bardziej skomplikowany kod, zamiast uzyskać wzrost wydajności.

Ale dlaczego ta optymalizacja się nie udała? Przecież program **powinien** działać o wiele szybciej. Nie wszystko jeszcze stracone. To się uda, tylko musisz jeszcze dowiedzieć się trochę więcej na temat obsługi strumieni przez sterownik CUDA. Aby poznać dogłębnie zasady działania strumieni, musisz dokładniej przyjrzeć się sterownikowi i architekturze sprzętowej CUDA.

10.6. Planowanie pracy GPU

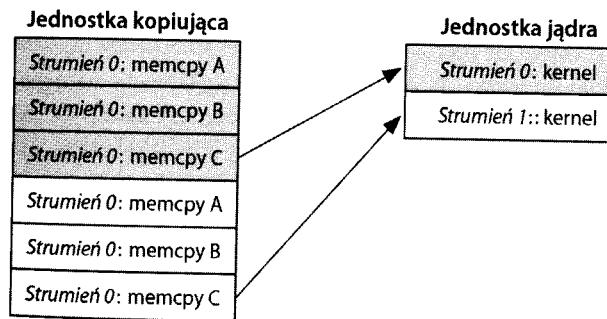
Mimo że strumienie to logicznie niezależne od siebie kolejki operacji do wykonania przez GPU, abstrakcja ta nie całkiem odpowiada mechanizmowi kolejowania samego procesora graficznego. Programiści myślą o swoich strumieniach jako o uporządkowanych sekwencjach operacji kopiowania danych i wywołań funkcji jądra. Lecz sprzęt sam w sobie nie wie, czym są strumienie. Zawiera po prostu pewną liczbę jednostek służących do przeprowadzania operacji kopiowania danych oraz jednostkę do wykonywania funkcji jądra. Jednostki te kolejują zadania niezależnie od siebie nawzajem, w wyniku czego wykonywanie programów może przebiegać tak, jak zilustrowano na rysunku 10.2.



Rysunek 10.2. rzut strumieni CUDA na jednostki wykonawcze GPU

W związku z tym programista ma nieco inne spojrzenie na kolejkowanie zadań dla GPU niż architektura sprzętowa, a obowiązek dogodzenia obu tym stronom spoczywa na sterowniku CUDA. Przede wszystkim istnieją ważne zależności związane z kolejnością dodawania operacji do strumieni. Na przykład na rysunku 10.2 widać, że operacja kopiowania porcji danych A przez strumień 0 musi zostać zakończona przed skopiowaniem porcji B, która to operacja z kolei musi się zakończyć przed wywołaniem funkcji jądra A. Gdy jednak operacje te zostaną wstawione do kolejek odpowiednich jednostek wykonawczych, opisane zależności zostają utrącone i żeby obie strony były zadowolone, do akcji w tym momencie musi wkroczyć sterownik CUDA.

Co to oznacza dla nas? Zobaczmy, co dokładnie dzieje się w naszym programie z podrozdziału 10.5. Jeśli przejrzymy kod, to zauważymy, że wszystko sprowadza się do skopiowania a za pomocą funkcji `cudaMemcpyAsync()`, skopiowania b za pomocą funkcji `cudaMemcpyAsync()`, wywołania jądra, a następnie skopiowania c z powrotem do hosta za pomocą funkcji `cudaMemcpyAsync()`. Aplikacja najpierw kolejkuje wszystkie operacje ze strumienia 0, a następnie ze strumienia 1. Sterownik CUDA rozplanowuje w sprzecie wykonywanie tych operacji w takiej kolejności, w jakiej zostały zdefiniowane przez programistę, zachowując zależności między jednostkami wykonawczymi. Zależności te przedstawiono na rysunku 10.3. Strzałki prowadzące od operacji kopiowania do jądra oznaczają, że kopowanie może się rozpocząć dopiero po zakończeniu działania jądra.

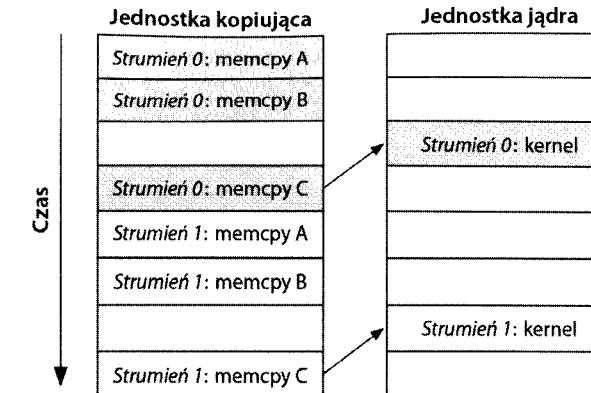


Rysunek 10.3. Strzałki wskazują zależności wywołań funkcji `cudaMemcpyAsync()` od wywołań jądra w programie z podrozdziału 10.5

Wyposażeni w nową wiedzę o zasadach harmonogramowania pracy GPU możemy spojrzeć na os czasu przedstawioną na rysunku 10.4.

Ponieważ wykonanie operacji kopiowania c do hosta w strumieniu 0 zależy od zakończenia jego funkcji jądra, kompletnie niezależne operacje kopiowania a i b na GPU strumienia 1 zostają zablokowane, ponieważ jednostki wykonawcze GPU wykonują zadania w takiej kolejności, w jakiej zostały one zdefiniowane. To wyjaśnia, dlaczego w wersji naszego programu z dwoma strumieniami nie uzyskaliśmy żadnego przyspieszenia. Powodem porażki było przyjęcie przez nas założenia, że sprzęt działa tak, jak na to wskazuje model programistyczny strumieni CUDA.

WIELU STRUMIENI CUDA JEDNOCZESNIE



Rysunek 10.4. Oś czasu wykonywania programu z podrozdziału 10.5

Morał z tego taki, że aby niezależne strumienie były wykonywane równolegle, programista musi im w tym trochę pomóc. Pamiętając, że sprzęt jest wyposażony w osobne jednostki wykonawcze do kopiowania danych i wykonywania funkcji jądra, musimy również mieć świadomość, że kolejność, w jakiej zdefiniujemy te operacje w naszych strumieniach, będzie miała wpływ na ich rozplanowanie do wykonania przez sterownik CUDA. W następnym podrozdziale pokazujemy, jak sprawić, żeby operacje kopiowania danych i wywołania funkcji jądra były przeplatane.

10.7. Efektywne wykorzystanie wielu strumieni CUDA jednocześnie

W poprzednim podrozdziale dowiedzieliśmy się, że jeśli zaplanujemy wykonanie wszystkich operacji jednego strumienia naraz, to możemy niechcący zablokować operacje kopiowania lub wywołania jądra innego strumienia. Problem ten można zminimalizować, kolejując operacje wszerz zamiast w głąb. To znaczy zamiast dodawać operacje kopiowania a, kopiowania b, wykonywania jądra i kopiowania c do strumienia 0 przed rozpoczęciem planowania strumienia 1, powinniśmy przypisywać zadania strumieniom na przemian. Dodajemy kopiowanie a do strumienia 0, a następnie dodajemy kopiowanie a do strumienia 1. Później dodajemy kopiowanie b do strumienia 0, a następnie kopiowanie b do strumienia 1. Kolejkujemy wykonanie jądra w strumieniu 0, a następnie robimy to samo w strumieniu 1. Na zakończenie kopiujemy c z powrotem do hosta w strumieniu 0 i powtarzamy to samo w strumieniu 1.

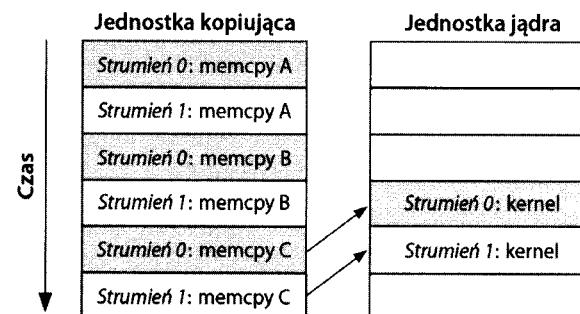
Najlepiej objąśnić to na konkretnym przykładzie. Zmieniamy tylko kolejność przypisywania operacji do strumieni, a więc cała optymalizacja będzie polegać wyłącznie na wycinaniu fragmentów kodu i wklejaniu ich w innych miejscach. Nic poza treścią pętli `for()` się nie zmieni. Oto nowa wersja kodu:

```

for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    //Kolejkowanie operacji kopiowania w strumieniach
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,
        N * sizeof(int),
        cudaMemcpyHostToDevice,
        stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N,
        N * sizeof(int),
        cudaMemcpyHostToDevice,
        stream1 ) );
    //Zaplanowanie kopiowania b w strumieniach 0 i 1
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,
        N * sizeof(int),
        cudaMemcpyHostToDevice,
        stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N,
        N * sizeof(int),
        cudaMemcpyHostToDevice,
        stream1 ) );
    //Zaplanowanie uruchomienia funkcji jądra w strumieniach 0 i 1
    kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
    kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
    //Zaplanowanie kopiowania c z GPU do pamięci zablokowanej
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,
        N * sizeof(int),
        cudaMemcpyDeviceToHost,
        stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1,
        N * sizeof(int),
        cudaMemcpyDeviceToHost,
        stream1 ) );
}

```

Jeśli przyjmiemy, że czas wykonywania operacji kopiowania jest mniej więcej taki sam jak czas działania funkcji jądra, oś czasu tego programu będzie wyglądała tak jak na rysunku 10.5. Strzałkami oznaczono zależności między jednostkami wykonawczymi, aby pokazać, że zmiany były korzystne.



Rysunek 10.5. Oś czasu wykonywania poprawionej wersji programu ze strzałkami wskazującymi zależności między jednostkami wykonawczymi

Dzięki zmianie sposobu kolejkowania operacji w strumieniach teraz operacja kopiowania z strumienia 0 nie blokuje początkowych operacji kopiowania a i b przez strumień 1. Umożliwia to procesorowi GPU równolegle wykonywanie operacji kopiowania i funkcji jądra, co oczywiście korzystnie wpływa na wydajność programu. Czas działania nowego algorytmu wyniósł 48 ms, a więc uzyskaliśmy 21-procentowy wzrost wydajności w porównaniu z pierwszą (naiwną) implementacją. W programach, które mogą w ten sposób przeplatać wszystkie obliczenia i operacje kopiowania, możliwe jest prawie podwojenie wydajności, ponieważ jednostki kopiujące i wykonawcze jąder działają w nich bez przerwy.

10.8. Podsumowanie

W tym rozdziale opisaliśmy technikę zrównoleglania na poziomie zadań w CUDA C. Za pomocą strumieni można zmusić GPU do jednoczesnego wykonywania funkcji jądra i kopowania danych między hostem a urządzeniem. Stosując tę technikę, należy jednak pamiętać o dwóch ważnych rzeczach. Po pierwsze pamięć, która będzie używana, należy alokować za pomocą funkcji cudaHostAlloc(), ponieważ operacje kopiowania kolejkuje się za pomocą asynchronicznej funkcji cudaMemcpyAsync() działającej tylko na pamięci z zablokowanym stronowaniem. Po drugie kolejność dodawania operacji do strumieni ma wpływ na to, czy operacje kopiowania i wykonania jądra będą przeplatane, czy nie. Ogólna zasada jest taka, aby strumieniom przypisywać zadania na przemian. Aby zrozumieć, dlaczego jest to konieczne, trzeba wiedzieć, jak działa mechanizm kolejkowania w sprzęcie. Dlatego też warto o tym pamiętać.

Rozdział 11

Wykonywanie kodu CUDA C jednocześnie na wielu GPU

Jest takie stare przysłowie, które brzmi mniej więcej tak: „Od jednego GPU lepsze mogą być tylko dwa GPU”. Ostatnimi czasy coraz częściej spotyka się komputery zawierające więcej niż jeden procesor graficzny. Oczywiście sytuacja ta jest podobna do sytuacji z procesorami CPU, tzn. systemy dwuprocesorowe wciąż nie są czymś powszechnym, ale zdarzają się coraz częściej. Na przykład takie produkty jak GeForce GTX 295 zawierają dwa GPU na jednej karcie, a Tesla S1070 ma aż cztery procesory graficzne zbudowane na bazie architektury CUDA. Komputery składane za pomocą najnowszych chipsetów NVIDII mają układ graficzny z obsługą CUDA zintegrowany z płytą główną. Wystarczy jeszcze tylko podłączyć kartę grafiki NVIDII do gniazda PCI Express, a już ma się system wieloprocesorowy. Ponieważ przypadki, o których mowa, nie są wcale niczym egzotycznym, warto nauczyć się wykorzystywać możliwości tak wyposażonych komputerów.

11.1. Streszczenie rozdziału

W tym rozdziale:

- Dowiesz się, na czym polega transfer danych bez kopiowania i nauczysz się stosować tę technikę.
- Poznasz sposoby wykorzystania wielu procesorów GPU w jednym programie.
- Nauczysz się alokować przenośną pamięć zablokowaną i używać jej.

11.2. Pamięć hosta niewymagająca kopiowania

W rozdziale 10. poznaliśmy nowy rodzaj pamięci hosta, która po alokacji nigdy nie zmienia swojego fizycznego położenia, czyli tzw. pamięć zablokowaną albo inaczej pamięć z wyłączonym stronicowaniem. Przypomnijmy, że do alokacji jej domyślnego typu służy funkcja `cudaHostAlloc()` z parametrem `cudaHostAllocDefault`. Obiecaliśmy też, że w rozdziale 11. pokażemy inne, jeszcze ciekawsze sposoby alokacji pamięci zablokowanej. Jeśli doczytałeś do tego miejsca tylko po to, aby się dowiedzieć, co to za techniki, to teraz będziesz na pewno usatysfakcjonowany. Zamiast aby się dowiedzieć, co to za techniki, to teraz będziesz na pewno usatysfakcjonowany. Zamiast aby się dowiedzieć, co to za techniki, to teraz będziesz na pewno usatysfakcjonowany. Zamiast aby się dowiedzieć, co to za techniki, to teraz będziesz na pewno usatysfakcjonowany. Zamiast aby się dowiedzieć, co to za techniki, to teraz będziesz na pewno usatysfakcjonowany. Podobnie jak w po- parametrze `cudaHostAllocDefault` można przekazać `cudaHostAllocMapped`. Podobnie jak w po- przednim przypadku, pamięć alokowana za jego pomocą jest **zablokowana**, a więc nie może być stronicowana ani fizycznie przenoszona. Różnica polega na tym, że ten rodzaj pamięci po- zwała na złamanie jednej z pierwszych zasad dotyczących pamięci hosta opisanych w rozdziale 3.: dostęp do niej można uzyskać bezpośrednio z funkcji jądra w CUDA C. W związku z tym nie trzeba jej kopiować i stąd jej nazwa — pamięć **niekopiowana** (ang. *zero-copy memory*).

11.2.1. OBLCZANIE ILOCZYNU SKALARNEGO ZA POMOCĄ PAMIĘCI NIEKOPIOWANEJ

Zazwyczaj procesor GPU ma dostęp tylko do pamięci GPU, a CPU — do pamięci hosta. Czasami jednak dobrze jest tę zasadę złamać. Jako przykład takiej sytuacji przedstawimy nasz ulubiony program redukujący, który oblicza iloczyn skalarny wektorów. Jeśli czytasz tę książkę od początku, to zapewne pamiętasz naszą pierwszą próbę wykonania tych obliczeń. Najpierw skopiowaliśmy oba wektory do GPU, wykonaliśmy odpowiednie obliczenia, a następnie wyniki częstkowe skopiowaliśmy do hosta, na którym dokonczyliśmy działania za pomocą CPU.

Teraz pominiemy etap kopiowania danych i uzyskamy do nich dostęp w GPU bezpośrednio z pamięci niekopiowanej. Program ten skonstruujemy dokładnie tak samo jak program, którego użyliśmy do testowania pamięci zablokowanej. To znaczy napiszemy dwie funkcje. Jedna wy- kona test za pomocą standardowej pamięci zablokowanej, a druga wykona redukcję na GPU przy użyciu pamięci niekopiowanej. Najpierw przeanalizujemy wersję ze standardową pamięcią. Rozpoczynamy tradycyjnie, czyli od utworzenia zdarzeń do odmierzania czasu, alokacji buforów wejściowego i wyjściowego oraz napełnienia ich danymi.

```
float malloc_test( int size ) {
    cudaEvent_t start, stop;
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    // Alokacja pamięci na hoście
    a = (float*)malloc( size*sizeof(float) );
    b = (float*)malloc( size*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
    // Alokacja pamięci na GPU
```

```
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                           size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                           size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                           blocksPerGrid*sizeof(float) ) );
// Napełnienie pamięci hosta danymi
for ( int i=0; i<size; i++ ) {
    a[i] = i;
    b[i] = i*2;
}
```

Po dokonaniu alokacji i utworzeniu zbioru danych można rozpoczęć wykonywanie obliczeń. Uruchamiamy stoper, kopujemy dane wejściowe do GPU, uruchamiamy funkcję jądra obliczającą iloczyn skalarny, a następnie kopujemy wyniki częstkowe z powrotem do hosta.

```
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
// Skopiowanie tablic a i b do GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, size*sizeof(float),
                         cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, size*sizeof(float),
                         cudaMemcpyHostToDevice ) );
dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
                                             dev_partial_c );
// Skopiowanie tablicy c z powrotem do CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                         blocksPerGrid*sizeof(float),
                         cudaMemcpyDeviceToHost ) );
```

Teraz musimy dokonczyć obliczenia na CPU, podobnie jak robiliśmy to w rozdziale 5. Najpierw jednak wyłączymy stoper, ponieważ służy on tylko do mierzenia czasu działania GPU.

```
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
```

Na zakończenie sumujemy wyniki częstkowe i zwalniamy bufore:

```
// Dokończenie działań na CPU
c = 0;
for ( int i=0; i<blocksPerGrid; i++ ) {
    c += partial_c[i];
}
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_partial_c ) );
// Zwolnienie pamięci na hoście
free( a );
```

```

        free( b );
        free( partial_c );
        //Zwolnienie zdarzeń
        HANDLE_ERROR( cudaEventDestroy( start ) );
        HANDLE_ERROR( cudaEventDestroy( stop ) );
        printf( "Obliczona wartość: %f\n", c );
        return elapsedTime;
    }
}

```

Wersja programu z użyciem pamięci niekopiowanej jest bardzo podobna, a jedyna różnica dotyczy sposobu alokacji pamięci. Zatem rozpoczynamy tak jak poprzednio, tzn. od alokacji buforów wejściowego i wyjściowego oraz napełnienia pierwszego z nich danymi:

```

float cuda_host_alloc_test( int size ) {
    cudaEvent_t      start, stop;
    float            *a, *b, c, *partial_c;
    float            *dev_a, *dev_b, *dev_partial_c;
    float            elapsedTime;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    //Alokacja pamięci na CPU
    HANDLE_ERROR( cudaHostAlloc( (void**)&a,
                                size*sizeof(float),
                                cudaHostAllocWriteCombined |
                                cudaHostAllocMapped ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&b,
                                size*sizeof(float),
                                cudaHostAllocWriteCombined |
                                cudaHostAllocMapped ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&partial_c,
                                blocksPerGrid*sizeof(float),
                                cudaHostAllocMapped ) );
    //Zapełnienie bufora na hoście danymi
    for (int i=0; i<size; i++) {
        a[i] = i;
        b[i] = i*2;
    }
}

```

Znowu, podobnie jak w rozdziale 10, widzimy w akcji funkcję `cudaHostAlloc()`, lecz tym razem z argumentem `flags` ustawnionym na `cudaHostAllocMapped`. Informuje on system wykonawczy, że zamierzamy odwoływać się do bufora z poziomu GPU, a więc, innymi słowy, zamienia ten bufor w pamięć niekopiowaną. Dla obu buforów wejściowych użyliśmy parametrów `flags` o wartości `cudaHostAllocWriteCombined`. Oznacza to, że system wykonawczy powinien je alokować jako bufore z możliwością grupowania operacji zapisu w odniesieniu do pamięci podręcznej CPU. Znacznik ten nie zmienia funkcjonalności aplikacji, lecz umożliwia zoptymalizowanie szybkości działania buforów, z których dane będą odczytywane tylko przez GPU. Jednak tego typu obszary pamięci mogą okazać się bardzo mało wydajne w przypadkach, gdy CPU również musi dokonywać odczytu z bufora. Dlatego przed podjęciem decyzji, czy zastosować tę technikę, zawsze należy przeanalizować potencjalne zyski i straty.

Ponieważ do alokacji pamięci na hoście został użyty argument `cudaHostAllocMapped`, do buforów tych mamy dostęp bezpośrednio z GPU. Wirtualna przestrzeń pamięci procesora graficznego jest jednak inna niż CPU, co oznacza, że bufore te w każdym z tych przypadków mają inne adresy. Funkcja `cudaHostAlloc()` zwraca wskaźnik do pamięci dla hosta, a żeby otrzymać prawidłowy wskaźnik dla GPU, należy użyć funkcji `cudaHostGetDevicePointer()`. Wskaźniki te zostaną przekazane do funkcji jądra i użyte przez GPU w celu odczytu i zapisu danych w pamięci alokowanej na hoście:

```

HANDLE_ERROR( cudaHostGetDevicePointer( &dev_a, a, 0 ) );
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_b, b, 0 ) );
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_partial_c,
                                        partial_c, 0 ) );

```

Mimo że wskaźniki `dev_a`, `dev_b` i `dev_partial_c` są przechowywane na hoście, dzięki wywołaniom funkcji `cudaHostGetDevicePointer()` dla jądra będą one wyglądały jak pamięć GPU. Ponieważ wyniki cząstkowe są już na hoście, nie musimy używać funkcji `cudaMemcpy()`, aby je tam skopiować. Zwróć jednak uwagę na fakt synchronizacji CPU z GPU za pomocą funkcji `cudaThreadSynchronize()`. Zawartość pamięci niekopiowanej podczas wykonywania kodu jądra, który może ją potencjalnie modyfikować, jest niezdefiniowana. Dzięki synchronizacji zyskujemy pewność, że jądro zakończyło działanie i że bufor w pamięci niekopiowanej zawiera wyniki, a więc można zatrzymać stoper i dokończyć obliczenia na CPU, tak jak poprzednio:

```

HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
//Dokończenie obliczeń na CPU
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}

```

Pozostało jeszcze tylko zwolnić zajęte zasoby:

```

HANDLE_ERROR( cudaFreeHost( a ) );
HANDLE_ERROR( cudaFreeHost( b ) );
HANDLE_ERROR( cudaFreeHost( partial_c ) );
//Zwolnienie zdarzeń
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
printf( "Obliczona wartość: %f\n", c );
return elapsedTime;
}

```

Jak widać, niezależnie od tego, jakich znaczników użyje się w wywołaniu funkcji `cudaHostAlloc()`, alokowaną pamięć zawsze zwalnia się w taki sam sposób, tzn. za pomocą funkcji `cudaFreeHost()`.

I to już prawie koniec! Pozostało jeszcze tylko połączyć to wszystko w funkcji main(). Przede wszystkim powinniśmy zacząć od sprawdzenia, czy urządzenie, którym dysponujemy, obsługuje mapowanie pamięci hosta. W tym celu do sprawdzania obsługi funkcji przeplotu zastosujemy taką samą technikę jak w poprzednim rozdziale, tzn. wywołamy funkcję cudaGetDeviceProperties():

```
int main( void ) {
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
    if (prop.canMapHostMemory != 1) {
        printf( "Urządzenie %d nie obsługuje funkcji mapowania pamięci.\n" );
        return 0;
    }
}
```

Zakładając, że urządzenie obsługuje pamięć niekopiowaną, przełączamy system wykonawczy na tryb, w którym będzie mógł alokować bufore tej pamięci. Do tego celu posłuży nam funkcja cudaSetDeviceFlags() wywołana ze znacznikiem cudaDeviceMapHost oznaczającym, że chcemy, aby urządzenie mogło mapować pamięć hosta:

```
HANDLE_ERROR( cudaSetDeviceFlags( cudaDeviceMapHost ) );
```

To w zasadzie wszystko, jeśli chodzi o funkcję main(). Uruchamiamy nasze dwa testy, wyświetlamy wynik pomiaru czasu wykonywania i zamykamy program:

```
float elapsedTime = malloc_test( N );
printf( "Czas przy użyciu funkcji cudaMalloc: %3.1f ms\n",
    elapsedTime );
elapsedTime = cuda_host_alloc_test( N );
printf( "Czas przy użyciu funkcji cudaHostAlloc: %3.1f ms\n",
    elapsedTime );
}
```

Funkcja jądra pozostaje taka sama jak w rozdziale 5, ale dla wygody prezentujemy ją poniżej w całości:

```
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
__global__ void dot( int size, float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while (tid < size) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    // Ustawienie wartości pamięci podręcznej
    cache[cacheIndex] = temp;
}
// Synchronizacja wątków w bloku
__syncthreads();
// W przypadku redukcji threadsPerBlock musi być potęgą 2
// ze względu na poniższy kod.
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
}
```

```
tid += blockDim.x * gridDim.x;
}

// Ustawienie wartości pamięci podręcznej
cache[cacheIndex] = temp;

// Synchronizacja wątków w bloku
__syncthreads();
// W przypadku redukcji threadsPerBlock musi być potęgą 2
// ze względu na poniższy kod.
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
}
```

11.2.2. WYDAJNOŚĆ PAMIĘCI NIEKOPIOWANEJ

Jakiego efektu należy się spodziewać po użyciu pamięci niekopiowanej? Oczywiście wynik zależy od konkretnego modelu GPU i od tego, czy jest to układ samodzielny, czy zintegrowany. Układy **samodzielne** mają własną pamięć DRAM i są montowane niezależnie od CPU na osobnych płytach drukowanych. Takie są na przykład wszystkie karty graficzne, które wkłada się do komputera. Natomiast **układy zintegrowane** to wbudowane w chipset systemowy procesory graficzne, które korzystają z tej samej pamięci co CPU. Wiele komputerów składanych na bazie chipsetu nForce i procesorów komunikacyjnych (MCP) NVIDIA zawiera procesory graficzne z obsługą technologii CUDA. Ponadto wszystkie notebooki, netbooki i komputery PC oparte na platformie ION NVIDIA również zawierają zintegrowane jednostki graficzne obsługujące CUDA. W przypadku układów zintegrowanych użycie pamięci niekopiowanej jest **zawsze korzystne**, ponieważ procesory te i tak fizycznie współdzielą pamięć z hostem. Zadeklarowanie bufora jako pamięci niekopiowanej powoduje jedynie wyeliminowanie niepotrzebnych operacji kopiowania. Należy jednak pamiętać, że nie ma nic za darmo. Bufory niekopiowane mają takie same wady jak wszystkie inne rodzaje pamięci zablokowanej: ograniczają ilość fizycznej pamięci, co czasami może doprowadzić do spadku wydajności.

W przypadkach gdy dane wejściowe i wyjściowe są używane dokładnie po jednym razie, wzrost wydajności można zaobserwować także na samodzielnych GPU. Ponieważ procesory graficzne są mistrzami w ukrywaniu opóźnień związanych z dostępem do pamięci, czas potrzebny na odczyt i zapis danych poprzez magistralę PCI Express można do pewnego stopnia zminimalizować, korzystając z opisywanego mechanizmu. Ponieważ jednak pamięć niekopiowana nie jest zapisywana w buforze podręcznym GPU, w przypadkach wielokrotnego odczytu pamięci wystąpi duże opóźnienie, którego można by było uniknąć, kopując po prostu dane do GPU.

czy GPU jest układem samodzielnym, czy zintegrowanym? Można po prostu komputer i zobaczyć, ale raczej nie uda się tego dokonać za pomocą programu. Dlatego jest poszukać odpowiedniej właściwości w strukturze `cudaGetDeviceProperties()`. Właściwość `integrated` ma wartość `true`, oznacza to, że urządzenie jest układem zintegrowanym, natomiast `false` znaczy, że mamy do czynienia z samodzielną kartą graficzną.

Program spełnia warunek „pojedynczego odczytu i zapisu pamięci”, a więc istnieje duże podobieństwo, że jeśli użyjemy w nim pamięci niekopiowanej, zaobserwujemy w jego wyniku wzrost wydajności. I rzeczywiście, na procesorze GeForce GTX 285 czas wykonywania zadania skrócony z 98,1 ms do 52,1 ms, czyli aż o 45%. Podobny wzrost wydajności zauważony na procesorze GeForce GTX 280 — ze 143,9 ms do 94,7 ms, czyli o 34%. Oczywiście na procesorze uzyska się inny wynik, ponieważ każdy ma inny współczynnik obliczeń i wartości oraz ze względu na różnice w szybkości transmisji danych poprzez magistralę PCI Express między chipsetami.

Użycie kilku procesorów GPU jednocześnie

W jednym podrozdziale napisaliśmy, że procesor GPU może być albo zintegrowany w komputerze, albo występować jako osobny układ na oddzielnej karcie rozszerzeń podłączanej do portu PCI Express. Coraz więcej komputerów zawiera układy **zarówno** jednego, jak i dwóch, czyli coraz częściej można spotkać systemy wieloprocesorowe. Także sama firma NVIDIA oferuje w sprzedaży produkty zawierające więcej niż jeden procesor GPU, np. GeForce GTX 470. Mimo że układ ten zajmuje tylko jedno złącze kart rozszerzeń, aplikacjom w języku C++ przedstawia się on jako dwa osobne GPU. Dodatkowo w jednym komputerze można instalować kilka kart graficznych, a następnie połączyć je za pomocą technologii SLI (ang. *Scalable Link Interface*). Z wymienionych powodów aplikacje CUDA już stosunkowo często są uruchamiane w systemach wieloprocesorowych. A ponieważ programy te zazwyczaj doskonale wykorzystują możliwości obu procesorów GPU. Pozwoliłyby to nam wykorzystać pełną moc obu kart.

W tym rozdziale Ci konieczności zapoznawania się z nowym programem, zmodyfikujemy poprzedni program, aby wykorzystać możliwości obu procesorów GPU. Dla uproszczenia wszystkie potrzebne w tych obliczeniach dane umieścimy w jednej strukturze danych. Wkrótce dowiesz się, na czym to polega.

```
DataStruct {
    deviceID;
    size;
    float *a;
    float *b;
    float returnValue;
```

Powyższa struktura zawiera dane identyfikacyjne urządzenia, na którym będzie obliczany iloczyn skalarny. Zawiera ona dane o rozmiarze buforów wejściowych, wskaźniki na bufore, których nazwy brzmiają `a` i `b`, oraz element do przechowywania iloczynu skalarnego `returnValue`.

Aby użyć `N` procesorów GPU, musimy się dowiedzieć, ile dokładnie wynosi `N`. Dlatego też program rozpoczynamy od wywołania funkcji `cudaGetDeviceCount()` w celu sprawdzenia, ile procesorów obsługujących CUDA znajduje się w systemie.

```
int main( void ) {
    int deviceCount;
    HANDLE_ERROR( cudaGetDeviceCount( &deviceCount ) );
    if (deviceCount < 2) {
        printf( "Potrzebne są przynajmniej dwa urządzenia z potencjałem "
                "obliczeniowym 1.0, lecz znaleziono tylko %d\n", deviceCount );
        return 0;
    }
```

Ponieważ niniejszy program służy do demonstracji technik wykorzystania kilku procesorów GPU obsługujących CUDA jednocześnie, to jeśli w komputerze znajduje się tylko jeden taki procesor, po prostu kończymy jego działanie. Oczywiście należy podkreślić, że nie jest to zalecane. Zeby niepotrzebnie nie komplikować kodu, bufore zapiszemy w standardowej pamięci hosta, a następnie napełnimy je danymi dokładnie w taki sam sposób, jak to robiliśmy wcześniej.

```
float *a = (float*)malloc( sizeof(float) * N );
HANDLE_NULL( a );
float *b = (float*)malloc( sizeof(float) * N );
HANDLE_NULL( b );
// Zapełnienie pamięci hosta danymi
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}
```

Teraz możemy przejść do właściwej części programu, czyli technik wykorzystania kilku GPU jednocześnie. Cała sztuka używania wielu GPU z poziomu API systemu wykonawczego CUDA polega na kontrolowaniu każdego z procesorów graficznych w osobnym wątku procesora CPU. Wczesniej nas to nie interesowało, bo używaliśmy tylko jednego GPU. Dużą część kodu związanej z wielowątkowością zapisaliśmy w pliku pomocniczym `book.h`. Teraz musimy tylko zapełnić strukturę danymi niezbędnymi do wykonania obliczeń. Mimo że w komputerze może być każda liczba procesorów GPU większa od jednego, my dla uproszczenia przyjmiemy, że mamy dwa takie procesory:

```
DataStruct data[2];
data[0].deviceID = 0;
data[0].size = N/2;
data[0].a = a;
data[0].b = b;
```

```
data[1].deviceID = 1;
data[1].size = N/2;
data[1].a = a + N/2;
data[1].b = b + N/2;
```

Jedną ze zmiennych DataStruct przekazujemy do funkcji użytkowej, której nadaliśmy nazwę `start_thread()`. Ponadto funkcji tej przekazujemy też wskaźnik na funkcję, która ma zostać wywołana przez nowo utworzony wątek. W tym przypadku jest to funkcja o nazwie `routine()`. Funkcja `start_thread()` utworzy nowy wątek, który następnie wywoła podaną funkcję, przekazując jej strukturę `DataStruct`. Drugie wywołanie funkcji `routine()` zostanie wykonane w domyślnym wątku aplikacji, co oznacza, że musielibyśmy utworzyć tylko jeden **dodatkowy wątek**:

```
CUTThread thread = start_thread( routine, &(data[0]) );
routine( &(data[1]) );
```

Zanim przejdziemy dalej, za pomocą funkcji `end_thread()` zmusimy główny wątek programu do poczekania, aż drugi wątek zakończy działanie:

```
end_thread( thread );
```

Gdy wiadomo, że oba wątki zakończyły już działanie, można zwolnić zasoby i wyświetlić wynik:

```
free( a );
free( b );
printf( "Obliczona wartość: %f\n",
       data[0].returnValue + data[1].returnValue );
return 0;
}
```

Zwróć uwagę, że sumujemy wyniki zwrócone przez każdy z wątków. Operacja ta stanowi ostatni etap naszej redukcji. W przypadku innych algorytmów na otrzymanych wynikach cząstkowych mogłyby być wykonywane jakieś inne działania. W rzeczywistości każdy z procesorów GPU może wykonywać kompletnie inne działania na różnych zbiorach danych. My celowo jednak wybraliśmy prosty przykład, aby niczego nie komplikować.

Procedura obliczająca iloczyn skalarny jest taka sama jak w poprzednich wersjach, a więc nie będziemy jej tu kolejny raz prezentować. Natomiast kod funkcji `routine()` powinien być już o wiele bardziej interesujący. Funkcja ta pobiera i zwraca typ `void*`, dzięki czemu funkcji `start_thread()` można użyć w dowolnej implementacji funkcji wątkowej. Chociaż chętnie przypisalibyśmy sobie autorstwo tego znakomitego pomysłu, to jednak w rzeczywistości jest to powszechnie stosowana technika definiowania funkcji zwrotnych w języku C:

```
void* routine( void *pvoidData ) {
    DataStruct *data = (DataStruct*)pvoidData;
    HANDLE_ERROR( cudaSetDevice( data->deviceID ) );
```

Każdy wątek wywołuje funkcję `cudaSetDevice()`, przekazując jej identyfikator innego urządzenia. W ten sposób sprawiamy, że każdy wątek steruje działaniem innego procesora. Używane GPU mogą być identyczne, np. dwa GeForce GTX 295, ale mogą też być całkiem różne, jak np. w systemach, które zawierają zarówno kartę grafiki, jak i zintegrowany układ graficzny. Te szczegóły nie są ważne dla samego programu, ale mogą Cię zainteresować. W szczególności wiedza ta jest przydatna, gdy do uruchomienia funkcji jądra potrzebne są układy o określonym minimalnym potencjale obliczeniowym albo gdy chce się zrównoważyć obciążenie obu jednostek. W przypadku dwóch różnych procesorów konieczne jest podzielenie zadania na takie części, aby każdy z nich był zajęty przez mniej więcej tyle samo czasu. Nie będziemy się tu jednak rozwodzić nad tym drobnym szczegółem.

Pomijając wywołanie funkcji `cudaSetDevice()` wybierającej procesor GPU, który ma być używany, kod funkcji `routine()` jest bardzo podobny do funkcji `malloc_test()` z podrozdziału 11.2.1. Alokujemy bufore dla kopii GPU danych wejściowych oraz bufor na wyniki cząstkowe, a następnie wywołujemy funkcję `cudaMemcpy()` w celu skopiowania każdej tablicy wejściowej do GPU:

```
int      size = data->size;
float   *a, *b, c, *partial_c;
float   *dev_a, *dev_b, *dev_partial_c;
// Alokacja pamięci na hoście
a = data->a;
b = data->b;
partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
// Alokacja pamięci na GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                           size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                           size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                           blocksPerGrid*sizeof(float) ) );
// Skopiowanie tablic a i b do GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, size*sizeof(float),
                           cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, size*sizeof(float),
                           cudaMemcpyHostToDevice ) );
```

Następnie uruchamiamy funkcję jądra, kopujemy wyniki z powrotem do hosta i kończymy obliczenia za pomocą CPU:

```
dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
                                              dev_partial_c );
// Skopiowanie tablicy c z powrotem do CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                           blocksPerGrid*sizeof(float),
                           cudaMemcpyDeviceToHost ) );
// Dokonanie obliczeń za pomocą CPU
```

```

c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}

```

Jak zwykle zwalniamy pamięć zajmowaną przez bufore i zwracamy iloczyn skalarny w polu `returnValue` struktury `DataStruct`:

```

HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_partial_c ) );
// Zwolnienie pamięci na hoście
free( partial_c );
data->returnValue = c;
return 0;
}

```

Gdy się tak dobrze zastanowić, to ogólnie pisanie kodu dla wielu procesorów GPU (poza kwestią dotyczącą rozplanowania wątków) nie jest dużo trudniejsze niż dla jednego. Użycie kodu pomocniczego do tworzenia wątków i uruchamiania na nich funkcji sprawia, że zadanie jest o wiele łatwiejsze. Jeśli masz własne biblioteki wątków, to nie wahaj się ich użyć w swoich programach. Musisz tylko pamiętać, że każdemu GPU przypisywany jest osobny wątek, a reszta to bułka z masłem.

11.4. Przenośna pamięć zablokowana

Ostatnia ważna kwestia dotycząca wykorzystania kilku procesorów GPU jednocześnie dotyczy użycia pamięci zablokowanej. Wiemy już z rozdziału 10, że pamięć zablokowana to obszar pamięci hosta z wyłączonym stronicowaniem, dzięki czemu nigdy nie zmienia się jego położenie. Jednakże strony pamięci mogą być zablokowane tylko dla jednego wątku CPU. Oznacza to, że jeśli **któryś** z wątków zablokuje obszar pamięci, będzie ona **wyglądała** na zablokowaną tylko dla niego. Jeśli wskaźnik na miejsce w tym obszarze zostanie przekazany do innych wątków, pamięć będzie w nich traktowana jak zwykły standardowy bufor.

W efekcie, jeśli jakiś wątek, który nie alokował zablokowanego bufora, spróbuje wykonać na nim funkcję `cudaMemcpy()`, to kopiowanie zostanie wykonane z taką szybkością, jakby to była zwykła pamięć niezablokowana. A jak pamiętamy z poprzedniego rozdziału, szybkość ta wynosi około połowy maksymalnej możliwej prędkości. Co gorsza, jeśli w jakimś strumieniu CUDA wątek ten spróbuje kolejkować wywołanie funkcji `cudaMemcpyAsync()`, operacja ta się nie powiedzie, ponieważ do jej wykonania potrzebna jest pamięć zablokowana, która wątkowi nie-dokonującemu alokacji jawi się jako stronicowana.

Na szczęście można temu zaradzić. Pamięć zablokowaną można alokować jako **przenośną**, co pozwoli nam przekazywać ją między wątkami hosta bez utraty jej podstawowej właściwości.

Pośłuży nam do tego nasza stara dobra funkcja `cudaHostAlloc()`, lecz z nowym parametrem `cudaHostAllocPortable`. Można go używać w połączeniu z innymi poznanymi dotychczas parametrami, np. `cudaHostAllocWriteCombined` i `cudaHostAllocMapped`. Oznacza to, że bufore na hoście można alokować na wiele różnych sposobów.

Sposób użycia przenośnej pamięci zablokowanej zademonstrujemy na przykładzie programu obliczającego iloczyn skalarny. Wykorzystamy jego wersję z pamięcią niekopiowaną, a więc to co napiszemy, będzie w istocie mieszanką wersji z pamięcią niekopiowaną oraz wersji wieloprocesorowej. Jak zwykle na początku programu będziemy sprawdzać, czy w systemie znajduje się wystarczająca liczba procesorów GPU opartych na architekturze CUDA:

```

int main( void ) {
    int deviceCount;
    HANDLE_ERROR( cudaGetDeviceCount( &deviceCount ) );
    if (deviceCount < 2) {
        printf( "Potrzebne są przynajmniej dwa urządzenia z potencjałem "
                "obliczeniowym 1.0, lecz znaleziono tylko %d\n", deviceCount );
        return 0;
    }
    cudaDeviceProp prop;
    for (int i=0; i<2; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        if (prop.canMapHostMemory != 1) {
            printf( "Urządzenie %d nie obsługuje funkcji mapowania pamięci.\n", i );
            return 0;
        }
    }
}

```

W poprzednich przypadkach moglibyśmy już alokować na hoście pamięć dla wektorów. Lecz aby użyć zablokowanej pamięci przenośnej, najpierw trzeba zainicjować urządzenie CUDA, na którym planowane jest wykonywanie programu. A ponieważ dodatkowo chcemy używać pamięci niekopiowanej, to po funkcji `cudaSetDevice()` wywołamy jeszcze (w podobny sposób jak w podrozdziale 11.2.1) funkcję `cudaSetDeviceFlags()`.

```

float *a, *b;
HANDLE_ERROR( cudaSetDevice( 0 ) );
HANDLE_ERROR( cudaSetDeviceFlags( cudaDeviceMapHost ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&a, N*sizeof(float),
                           cudaHostAllocWriteCombined |
                           cudaHostAllocPortable |
                           cudaHostAllocMapped ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&b, N*sizeof(float),
                           cudaHostAllocWriteCombined |
                           cudaHostAllocPortable |
                           cudaHostAllocMapped ) );

```

Wcześniej też używaliśmy funkcji `cudaSetDevice()`, ale wywoływałyśmy ją dopiero po alokowaniu pamięci i utworzeniu wątków. Jednakże jednym z warunków alokacji zablokowanej pamięci przenośnej za pomocą funkcji `cudaHostAlloc()` jest uprzednia inicjacja urządzenia za pomocą funkcji `cudaSetDevice()`. Zwróć też uwagę, że w obu alokacjach przekazujemy nowy znacznik `cudaHostAllocPortable`. Ponieważ bufore zostały alokowane po wywołaniu funkcji `cudaSetDevice(0)`, to gdybyśmy nie zdefiniowali ich jako przenośnych, byłyby widoczne jako pamięć zablokowana wyłącznie dla urządzenia zero.

Generujemy teraz dane wejściowe i przygotowujemy struktury `DataStruct` w podobny sposób jak w podrozdziale 11.3:

```
// Napełnienie pamięci hosta danymi
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}

// Przygotowanie do wielowątkowości
DataStruct data[2];
data[0].deviceID = 0;
data[0].offset = 0;
data[0].size = N/2;
data[0].a = a;
data[0].b = b;
data[1].deviceID = 1;
data[1].offset = N/2;
data[1].size = N/2;
data[1].a = a;
data[1].b = b;
```

Następnie możemy utworzyć drugi wątek i wywołać funkcję `routine()` w celu wykonania obliczeń na każdym z urządzeń:

```
CUTThread thread = start_thread( routine, &(data[1]) );
routine( &(data[0]) );
end_thread( thread );
```

Ponieważ alokacji pamięci hosta dokonał system wykonawczy CUDA, do jej zwolnienia użyjemy funkcji `cudaFreeHost()`. W funkcji `main()` oprócz zamiany funkcji `free()` nie ma więcej

```
// Zwolnienie pamięci hosta
HANDLE_ERROR( cudaFreeHost( a ) );
HANDLE_ERROR( cudaFreeHost( b ) );
printf( "Obliczona wartość: %f\n",
       data[0].returnValue + data[1].returnValue );
return 0;
```

Aby móc wykorzystać zablokowaną pamięć przenośną oraz niekopiowaną, musimy dokonać dwóch ważnych zmian w funkcji `routine()`. Pierwsza z nich jest dość subtelna i potrzeba jej wprowadzenia zapewne nie wyda Ci się taka oczywista.

```
void* routine( void *pvoidData ) {
    DataStruct *data = (DataStruct*)pvoidData;
    if (data->deviceID != 0) {
        HANDLE_ERROR( cudaSetDevice( data->deviceID ) );
        HANDLE_ERROR( cudaSetDeviceFlags( cudaDeviceMapHost ) );
    }
}
```

Przypomnijmy, że aby każdemu wątkowi przyporządkować inny GPU, w wieloprocesorowej wersji tego kodu konieczne jest wywołanie funkcji `cudaSetDevice()` w funkcji `routine()`. Tym razem jednak funkcję tę wywołyliśmy już w wątku głównym, aby móc alokować pamięć zablokowaną w funkcji `main()`. W związku z tym funkcje `cudaSetDevice()` i `cudaSetDeviceFlags()` powinniśmy wywoływać tylko dla tych urządzeń, dla których wywołanie to jeszcze nie zostało wykonane. Innymi słowy, wymienione funkcje wywołujemy tylko wówczas, gdy identyfikator urządzenia (`deviceID`) jest różny od zera. Może się wydawać, że kod byłby bardziej przejrzysty, gdyby te same wywołania zostały powtórzone także dla urządzenia zero, lecz to by spowodowało błąd. Po ustaleniu urządzenia dla danego wątku nie można ponownie wywołać funkcji `cudaSetDevice()`, nawet gdyby przekazało się do niej ten sam identyfikator. I właśnie ta wyróżniona instrukcja warunkowa usuwa tę drobną niedogodność. Teraz zajmiemy się drugą ze wspomnianych zmian.

Pamięć, której używamy po stronie hosta, nie tylko będzie zablokowana i przenośna, lecz również niekopiowana, co zapewni nam możliwość bezpośredniego dostępu do niej z poziomu GPU. W związku z tym musielibyśmy zrezygnować z używania funkcji `cudaMemcpy()`. Zamiast niej w celu pobrania wskaźników na pamięć hosta do użytku w urządzeniu wywołamy funkcję `cudaHostGetDevicePointer()` (tak samo uczyniliśmy w programie demonstrującym użycie pamięci niekopiowanej). Zauważ jednak, że do przechowywania wyników cząstkowych używamy standardowej pamięci GPU, którą jak zwykle alokujemy za pomocą funkcji `cudaMalloc()`:

```
int size = data->size;
float *a, *b, c, *partial_c;
float *dev_a, *dev_b, *dev_partial_c;
// Alokacja pamięci na hoście
a = data->a;
b = data->b;
partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_a, a, 0 ) );
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_b, b, 0 ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                         blocksPerGrid*sizeof(float) ) );
// Przejście w buforach a i b do miejsca, w którym ten GPU pobiera swoje dane.
dev_a += data->offset;
dev_b += data->offset;
```

zможемy uruchomić funkcję jądra i skopiować wyniki z GPU:

```
dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
                                              dev_partial_c );
// Skopiowanie tablicy c z GPU do CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                         blocksPerGrid*sizeof(float),
                         cudaMemcpyDeviceToHost ) );
```

akończenie sumujemy jak zwykle wyniki cząstkowe na CPU, zwalniamy zasoby i zwracamy ość do funkcji main():

```
// Dokończenie działania na CPU
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
HANDLE_ERROR( cudaFree( dev_partial_c ) );
// Zwolnienie pamięci na CPU
free( partial_c );
data->returnValue = c;
return 0;
```

5. Podsumowanie

zdziele tym poznaliśmy kilka nowych sposobów alokacji pamięci. Wszystkich ich daje się za pomocą funkcji `cudaHostAlloc()`. Za jej pomocą i przy użyciu kilku specjalnychników można alokować pamięć jako niekopionaną, przenośną i z grupowaniem operacji w dowolnych kombinacjach. Buforów niekopionowych używamy po to, aby nie musieć wać danych z i do GPU. Technika ta pozwala znacznie przyspieszyć działanie niektórychacji. Korzystając z biblioteki pomocniczej, sterowaliśmy dwoma procesorami GPU w jednym programie, co umożliwiło nam obliczenie iloczynu skalarnego za pomocą dwóch tychże jednoceśnie. W ostatniej części rozdziału nauczyliśmy się dzielić wspólną pamięćowaną między kilka GPU. W tym celu alokowaliśmy ją dodatkowo jako **przenośną**. W ostatnim przedstawionym przykładzie programu użyliśmy zablokowanej pamięci przenośnej, dwóch bufora niekopionowego, aby pokazać, jak można maksymalnie wykorzystać wszystkie możliwości sprzętu. Biorąc pod uwagę fakt, że liczba komputerów zawierających więcej niż jeden tale rośnie, warto znać te techniki, bo mogą się często przydawać.

Rozdział 12

Epilog

Gratulacje! Mamy nadzieję, że poznawanie języka CUDA C i eksperymentowanie z programowaniem procesorów graficznych sprawiło Ci dużą przyjemność. Materiału było dużo, więc teraz przejrzymy wszystko, o czym się dowiedzieliśmy, i zrobimy małe podsumowanie. Zaczęliśmy od znajomości języków C i C++. Na bazie tej wiedzy nauczyliśmy się uruchamiać wybraną liczbę kopii funkcji jądra na dowolnej liczbie wieloprocesorów za pomocą składni z nawiasami ostrymi systemu wykonawczego CUDA. Następnie poszerzyliśmy naszą wiedzę o **bloki i wątki** oraz dowiedzieliśmy się, jak przetwarzać zbiory danych o dowolnym rozmiarze. W tych bardziej skomplikowanych programach wykorzystywaliśmy techniki komunikacji międzywątkowej, która jest możliwa dzięki użyciu specjalnej pamięci wspólnej zlokalizowanej w układach GPU. Ponadto używaliśmy specjalnych konstrukcji synchronizacyjnych pozwalających działać w środowiskach obsługujących tysiące równolegle wykonywanych wątków.

Uzbrojeni w podstawową wiedzę na temat programowania równoległego w języku CUDA C na architekturze CUDA NVIDIA przeszliśmy do poznawania bardziej zaawansowanych koncepcji i interfejsów programistycznych. Ponieważ procesory GPU są przeznaczone do obróbki grafiki, nauczyliśmy się wykorzystywać pamięć tekstur w celu przyspieszenia działania programów korzystających z pamięci w specyficzny sposób. Biorąc pod uwagę fakt, że wielu użytkowników dodaje kod wykorzystujący GPU do graficznych aplikacji interaktywnych, opisaliśmy też zasady współpracy funkcji jądra w CUDA C z bibliotekami graficznymi, takimi jak OpenGL i DirectX. Z kolei operacje atomowe wykonywane zarówno na pamięci wspólnej, jak i globalnej umożliwiły nam bezpieczny dostęp do miejsc w pamięci wykorzystywanych przez wiele wątków. W dalszych rozdziałach zgłębialiśmy coraz bardziej zaawansowane techniki. Dzięki użyciu strumieni maksymalnie wykorzystywaliśmy możliwości całego systemu komputerowego, umożliwiając wykonywanie funkcji jądra równocześnie z przeprowadzaniem operacji kopiowania danych między hostem a GPU. Na koniec nauczyliśmy się alokować pamięć niekopionaną, której użycie pozwala przyspieszyć działanie programów uruchamianych na procesorach zintegrowanych. Ponadto dowiedzieliśmy się, jak inicjować kilka urządzeń jednocześnie i alokować przenośną pamięć zablokowaną w celu wykorzystania coraz częściej spotykanych systemów wieloprocesorowych.

12.1. Streszczenie rozdziału

W tym rozdziale:

- Poznasz niektóre narzędzia ułatwiające pisanie programów w języku CUDA C.
- Dowiesz się, gdzie można znaleźć przykłady kodu oraz literaturę poświęconą programowaniu w języku CUDA C, tak aby móc dalej rozwijać swoje umiejętności.

12.2. Narzędzia programistyczne

We wszystkich dotychczasowych rozdziałach korzystaliśmy z kilku składników platformy programistycznej języka CUDA C. Każdy napisany przez nas program skompilowaliśmy kompiutatorem CUDA C, aby przekształcić zawarte w nim funkcje jądra w kod wykonywalny dla procesorów GPU NVIDIA. Ponadto korzystaliśmy z pomocy systemu wykonawczego CUDA, który wykonywał za nas wiele czynności konfiguracyjnych potrzebnych do uruchomienia funkcji jądra i komunikacji z GPU. System ten z kolei komunikował się ze sterownikiem CUDA, który bezpośrednio kieruje działaniem procesora graficznego. Lecz oprócz tych wszystkich narzędzi firma NVIDIA udostępnia jeszcze wiele innych programów ułatwiających pisanie aplikacji w języku CUDA C. Rozdział ten nie stanowi dokumentacji tych narzędzi, lecz służy jedynie poinformowaniu Cię o ich istnieniu.

12.2.1. CUDA TOOLKIT

Zapewne masz już w swoim komputerze zestaw narzędzi o nazwie CUDA Toolkit. Był Ci on już potrzebny, ponieważ jest niezbędny do skompilowania programów CUDA C. Dlatego jeśli nie masz narzędzi CUDA Toolkit, to znaczy, że nie wypróbowałeś ani jednego programu z tej książki. Mamy Cię! Nieposiadanie wspomnianego oprogramowania to niby żadne przesępstwo, ale po co w takim razie czytałeś całą tę książkę? Z drugiej strony, jeśli przestudiowałeś wszystkie przykładowe programy, to powinienes mieć także wszystkie te biblioteki, których opis znajduje się ponizej.

12.2.2. BIBLIOTEKA CUFFT

Jeśli planujesz używać procesora GPU w swoich programach, to warto wiedzieć, że pakiet CUDA Toolkit zawiera dwie bardzo ważne biblioteki użytkowe. Pierwsza z nich to tzw. biblioteka CUFFT (ang. *CUDA Fast Fourier Transform*). W wersji 3.0 znalazło się w niej wiele ciekawych rzeczy:

- Jeden-, dwo- i trójwymiarowe przekształcenia zarówno liczb rzeczywistych, jak i zespolonych
- Serijne wykonywanie wielu równoległych przekształceń jednowymiarowych

12.2.3. BIBLIOTEKA CUBLAS

Druga bardzo przydatna biblioteka funkcji algebry liniowej to pakiet o nazwie CUBLAS, który jest implementacją szeroko znanej biblioteki o nazwie BLAS (ang. *Basic Linear Algebra Subprograms*). Jest ona bezpłatna i zawiera wersje wielu procedur zdefiniowanych w bibliotece BLAS, tzn. wersje wszystkich funkcji przyjmujących na wejściu liczby typu double pojedynczej i podwójnej precyzji oraz liczby rzeczywiste i zespolone. Ponieważ pierwotna implementacja biblioteki BLAS została napisana w języku FORTRAN, firma NVIDIA stara się zachować jak największą zgodność z wymogami tamtej implementacji. Na przykład w bibliotece CUBLAS stosowany jest kolumnowy porządek tablic zamiast standardowo stosowanego w językach C i C++ poządu kolumnowego porządku wierszowego. W praktyce nie ma to wielkiego znaczenia, ale umożliwia dotychczasowym użytkownikom biblioteki BLAS adaptację swoich programów do wymogów wykonywanej na GPU biblioteki CUBLAS bez większego wysiłku. Dodatkowo NVIDIA udostępnia wiązania do biblioteki CUBLAS w języku FORTRAN, aby pokazać, jak połączyć istniejące aplikacje napisane w tym języku z bibliotekami CUDA.

12.2.4. PAKIET GPU COMPUTING SDK

Dostępny niezależnie od sterowników NVIDIA i pakietu CUDA Toolkit zestaw GPU Computing SDK zawiera kilkadziesiąt przykładów kodu. Wspominaliśmy o nim już wcześniej, ponieważ jego zawartość stanowi znakomite uzupełnienie treści jedenastu poprzednich rozdziałów tej książki. Znajdujące się w nim programy są posortowane wg poziomów zaawansowania i tematyki. Oto jak z grubsza wygląda ten podział na kategorie:

CUDA Basic Topics (podstawy)

CUDA Advanced Topics (zaawansowane)

CUDA Systems Integration (integracja systemowa)

Data Parallel Algorithms (algorytm równoległy na poziomie danych)

Graphics Interoperability (współpraca z grafiką)

Texture (tekstury)

- Przekształcenia 2D i 3D o rozmiarach od 2 do 16384 elementów w każdym wymiarze
- Przekształcenia 1D danych wejściowych o rozmiarze do 8 milionów elementów
- Przekształcenia liczb rzeczywistych i zespolonych zapisem wyniku zarówno w miejscu danych wejściowych, jak i w innym miejscu
- Firma NVIDIA udostępnia bezpłatnie bibliotekę CUFFT i udziela na nią licencji zezwalającej na użytku zarówno osobistym, jak i komercyjnym, czy też w celach edukacyjnych.

Linear Algebra (algebra liniowa)

Image/Video Processing (przetwarzanie obrazów i filmów)

Computational Finance (finanse)

Data Compression (kompresja danych)

Physically-Based Simulation (symulacje fizyczne)

Programy te działają wszędzie tam, gdzie działa język CUDA C, i można je wykorzystać jako zaczątek własnych aplikacji. Jednak czytelników posiadających głęboką wiedzę z którejś z podanych dziedzin ostrzegamy, że nie mają co liczyć na wyrafinowane implementacje ulubionych algorytmów. Nie są to fragmenty kodu nadającego się do komercyjnego użytku, lecz programy edukacyjne, których celem jest zademonstrowanie sposobu działania języka CUDA C, podobnie jak wszystkich programów opisanych w tej książce.

12.2.5. BIBLIOTEKA NVIDIA PERFORMANCE PRIMITIVES

NVIDIA, oprócz procedur dostępnych w bibliotekach CUFFT i CUBLAS, rozwija także bibliotekę funkcji służących do przetwarzania danych o nazwie NVIDIA Performance Primitives (NPP). Aktualnie jej funkcjonalność koncentruje się głównie na przetwarzaniu obrazów i filmów, a więc najbardziej przydaje się specjalistom zajmującym się tymi dziedzinami. W planach firmy jest jednak rozszerzenie możliwości biblioteki NPP także na inne domeny. Jeśli zajmujesz się przetwarzaniem obrazu lub filmów i szukasz wydajnych rozwiązań, wypróbuj koniecznie bibliotekę NPP, którą można bezpłatnie pobrać pod adresem www.nvidia.com/object/npp.html (albo znaleźć za pomocą wyszukiwarki internetowej).

12.2.6. USUWANIE BŁĘDÓW Z KODU CUDA C

Doszły nas słuchy, że zdarza się czasami, iż przy pierwszym uruchomieniu niektóre programy nie działają tak, jak powinny. Niektóre zwracają niepoprawne wyniki, inne nie chcą się zamknąć, a jeszcze inne wprawiają cały komputer w taki stan, że reaguje on tylko na przycisk zasilania na obudowie. Autorzy tej książki sami **nigdy** nie splamili się takim brakiem profesjonalizmu, ale mają oni świadomość, że inni programiści mogą potrzebować pomocy w zakresie znajdowania usterek w programach i ich usuwania. Firma NVIDIA na szczęście udostępnia kilka narzędzi, które mogą ułatwić pracę w takich sytuacjach.

CUDA-GDB

Dla programistów pracujących w systemie Linux jednym z najbardziej przydatnych narzędzi jest program o nazwie CUDA-GDB. Firma NVIDIA do debugera GNU open-source o nazwie gdb dodała możliwość diagnozowania na bieżąco kodu urządzeń, a przy tym zachowała jego interfejs dobrze znany wszystkim użytkownikom. Wcześniej kod wykonywany przez urządzenie

można było diagnozować tylko poprzez tworzenie symulacji na CPU. Metoda ta była nie tylko niezwykle czasochłonna, lecz również często nie pozwalała na dokładne odwzorowanie działania GPU, gdyż dawała słabe rezultaty. Dzięki oprogramowaniu CUDA-GDB NVIDIA programiści mogą debugować funkcje jądra bezpośrednio na GPU, mając nad całym procesem tą samą kontrolę co w przypadku kodu działającego na CPU. Oto niektóre z najważniejszych funkcji debugera CUDA-GDB:

- Wyświetlanie stanu CUDA, np. informacji o zainstalowanych procesorach GPU i ich możliwościach
- Ustawianie punktów wstrzymania w kodzie źródłowym CUDA C
- Inspekcja wszystkich rodzajów pamięci globalnej i wspólnej GPU
- Inspekcja bloków i wątków, które aktualnie rezydują w GPU
- Wykonywanie krok po kroku osnów wątków
- Wchodzenie do działających programów, także zawieszonych i zakleszczonych

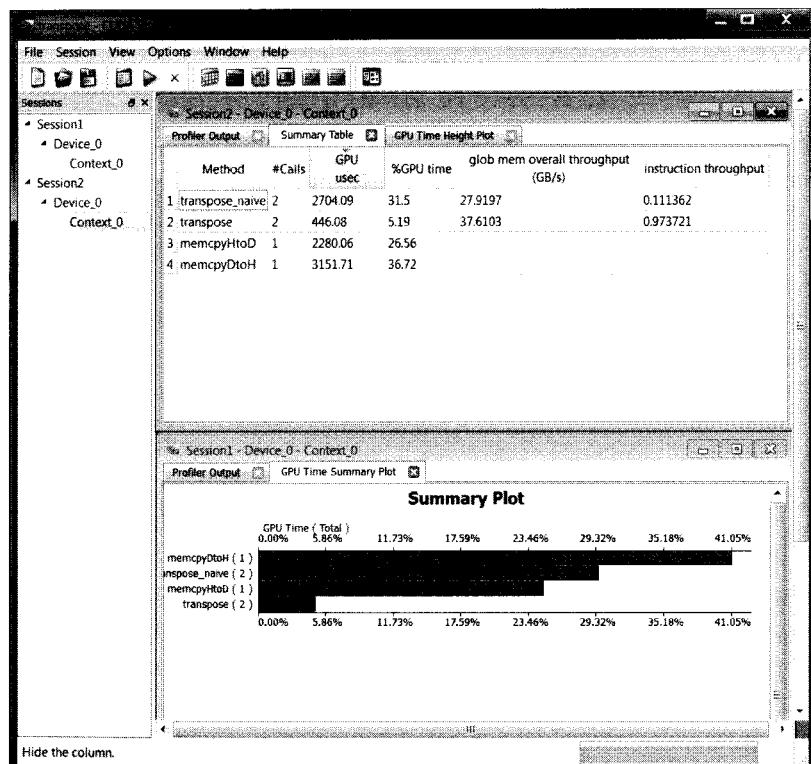
Wraz z debugerem dostarczane jest narzędzie CUDA Memory Checker, z którego można korzystać z poziomu debugera CUDA-GDB lub jako samodzielnego programu cuda-memcheck. Ponieważ architektura CUDA zawiera zaawansowaną jednostkę zarządzania pamięcią, która jest wbudowana bezpośrednio w sprzęt, wszelkie nieuprawnione próby dostępu do pamięci są wychwytywane i udaremniane przez samo urządzenie. Zgodnie z oczekiwaniemi w wyniku pogwałcenia zasad dostępu do pamięci program przestaje działać i wówczas dobrze jest mieć możliwość wglądu w to, dlaczego tak się dzieje. Program CUDA Memory Checker wykrywa przypadki nieautoryzowanego i nieprawidłowego dostępu do pamięci globalnej przez funkcje jądra i dostarcza bardzo szczegółowych informacji na ten temat.

NVIDIA PARALLEL NSIGHT

Mimo że CUDA-GDB to znakomite i dopracowane narzędzie do diagnozowania kodu, w firmie NVIDIA dostrzeżono, iż nie każdy programista jest zwariowany na punkcie Linuksa. Dlatego aby nie zmuszać użytkowników Windowsa do zajęcia się czymś innym niż programowanie, firma NVIDIA pod koniec 2009 roku opublikowała oprogramowanie NVIDIA Parallel Nsight (wcześniej znane pod nazwą kodową Nexus), czyli pierwszy debugger dla GPU i CPU zintegrowany ze środowiskiem Microsoft Visual Studio. Parallel Nsight, podobnie jak CUDA-GDB, umożliwia diagnozowanie aplikacji CUDA z uruchomionymi tysiącami wątków. Użytkownik może ustawiać punkty wstrzymania w dowolnych miejscach kodu źródłowego, a nawet tak je konfigurować, aby się uaktywniały w momencie zapisu danych w wybranych miejscach w pamięci. Można przeprowadzać inspekcję pamięci GPU bezpośrednio w oknie Visual Studio Memory oraz sprawdzać, czy nie ma operacji dostępu do pamięci poza jej granicami.

12.2.7. CUDA VISUAL PROFILER

Wielokrotnie wychwalaliśmy architekturę CUDA jako doskonałą platformę do budowy bardzo wydajnych programów. Niestety po usunięciu wszystkich błędów z kodu często ze zdumieniem odkrywamy, że zamiast „bardzo wydajnego programu” mamy zaledwie „program, który po prostu działa”. Wówczas zachodzimy w głowę, „dlaczego do diaska jest taki powolny”. Najlepszym lekarstwem na te bolączki jest uruchomienie aplikacji pod uważnym okiem narzędzia profilującego, które można pobrać w serwisie CUDA Zone. Na rysunku 12.1 przedstawiony jest program Visual Profiler w trakcie porównywania dwóch implementacji algorytmu transpozycji macierzy. Nie trzeba zaglądać do kodu źródłowego, żeby już na pierwszy rzut oka dostrzec, iż funkcja transpose() bije na głowę pod każdym względem funkcję transpose_naive() (ale czego można się spodziewać po funkcji mającej w nazwie słowo „naiwny”...).



Rysunek 12.1. CUDA Visual Profiler podczas porównywania algorytmów transpozycji macierzy

CUDA Visual Profiler uruchamia program i sprawdza odczyty specjalnych liczników wydajności wbudowanych w GPU. Po zakończeniu działania przetwarza zebrane dane i prezentuje je w postaci przejrzystego raportu. Potrafi określić, ile czasu zajęło wykonanie każdego jądra, ile bloków zostało uruchomionych, czy operacje dostępu do pamięci przez jądra są połączone, liczbę rozgałęzień wykonanych przez osnowy itd. Zalecamy używanie tego narzędzia wszystkim tym, którzy mają problemy z wydajnością programów i nie wiedzą, co je powoduje.

12.3. Literatura

Jeśli nie masz jeszcze dosyć czytania tej książki, to możliwe, że zechcesz sięgnąć także po inne. Zdajemy sobie sprawę, że wielu z Was woli zabawę z kodem źródłowym, ale dla tych, którzy lubią też coś poczytać, prezentujemy listę publikacji, zawierających dodatkowe informacje na temat programowania w języku CUDA C.

12.3.1. KSIĄŻKA PROGRAMMING MASSIVELY PARALLEL PROCESSORS: A HANDS-ON APPROACH

W pierwszym rozdziale napisaliśmy, że **nie** jest to książka o architekturach równoległych. Oczywiście wielokrotnie posługiwaliśmy się takimi terminami, jak **wieloprocesor** czy **osnowa**, ale naszym celem było przede wszystkim nauczenie Cię programowania w języku CUDA C i używania związkanych z nim interfejsów programistycznych. Kurs oparliśmy w dużym stopniu na modelu podręcznika *CUDA Programming Guide*, pomijając w większości przypadków aspekty związane ze sprzętową realizacją omawianych zadań.

Aby jednak stać się prawdziwym profesjonalnym programistą CUDA C, musisz lepiej poznać architekturę CUDA oraz różne niuanse działania procesorów GPU NVIDIA. Potrzebne Ci do tego informacje znajdziesz w książce *Programming Massively Parallel Processors: A Hands-on Approach* autorstwa Davida Kirka (byłego szefa działu badawczego NVIDIA) we współpracy z Wen-mei W. Hwu z University of Illinois. Znajdziesz w niej opis wielu znanych Ci już pojęć, ale poznasz także szczegóły budowy architektury CUDA, m.in. dowiesz się, jak planowane jest wykonywanie wątków, czym jest tolerancja opóźnienia, jaka jest przepustowość i efektywność pamięci, jak dokładnie traktowane są liczby zmennoprzecinkowe itd. Ponadto w książce programowanie równoległe opisane jest w nieco szerszym zarysie niż tutaj, dzięki czemu zyskasz wiedzę, jak projektować równoległe rozwiązania dużych i skomplikowanych problemów.

12.3.2. CUDA U

Niektoří z nas studia ukończyli w czasach, gdy jeszcze nikt nie słyszał o programowaniu procesorów graficznych. Jeśli akurat studujesz lub zamierzasz podjąć studia w niedalekiej przyszłości, to może zainteresuje Cię informacja, że aktualnie na świecie jest ponad 300 uniwersytów, na których prowadzone są wykłady zawierające także wiedzę o CUDA. Zanim jednak przejdziesz na dietę, żeby zmieścić się w swoje stare uniwersyteckie ciuchy, zajrzyj do serwisu CUDA Zone. Znajduje się tam odnośnik *CUDA U* prowadzący do internetowego uniwersytetu prowadzącego kursy z CUDA. Możesz też wpisać bezpośredni adres www.nvidia.com/object/cuda_education. Uczestnicząc w oferowanych kursach, możesz się sporo nauczyć o programowaniu GPU, lecz jak na razie niestety nic nam nie wiadomo o tym, żeby powstały jakieś bractwa do imprezowania po zajęciach.

Wśród niezliczonych materiałów edukacyjnych na temat technologii CUDA na wyróżnienie zasługuje pełny kurs programowania w języku CUDA C oferowany przez University of Illinois i firmę NVIDIA. Jest on dostępny bezpłatnie w formacie M4V, a więc można go odtwarzać na iPodzie, iPhone i na innych odtwarzaczach obsługujących ten format wideo. Już widzimy oczami wyobraźni, jak snujesz plany, że w końcu będziesz mógł nadrobić zaległości, stojąc w kolejce do myjni. Zapewne zastanawiasz się też, dlaczego czekaliśmy aż do samego końca książki z informacją, że istnieje coś, co można by uznać za jej filmową wersję. To chyba nie problem, bo przecież i tak książka zawsze jest lepsza od filmu, prawda? Oprócz materiałów University of Illinois i University of California Davis znajdziesz także materiały z CUDA Training Podcasts oraz łącza do innych kursów i usług w zakresie wsparcia.

DR. DOBB'S

Dr. Dobb's od ponad 30 lat zajmuje się wszystkim, co ma związek z programowaniem komputerów, a więc także technologią CUDA NVIDIA. W serwisie tym opublikowana została seria obszernych artykułów poświęconych temu wynalazkowi. Seria ta nosi tytuł *CUDA, Supercomputing for the Masses*. Rozpoczyna się ona od wprowadzenia do programowania GPU, a następnie przechodzi do objaśnienia sposobu pisania pierwszej funkcji jądra i bardziej zaawansowanych technik programowania CUDA. Dowiesz się z tych artykułów, jak obsługiwać błędy, jaka jest wydajność pamięci globalnej, jak korzystać z pamięci wspólnej, jak używać narzędzi CUDA Visual Profiler, pamięci tekstur, programu CUDA-GDB, biblioteki CUDPP oraz wielu innych rzeczy. Ponadto są one doskonałym źródłem dodatkowych informacji na tematy, które poruszaliśmy także w niniejszej książce. Znajdziesz w nich także bardziej szczegółowy opis programów, o których tylko tu wspomnieliśmy, czyli narzędzi do profilowania i debugowania. Odnośnik do serii można znaleźć w serwisie CUDA Zone albo poprzez wpisanie w wyszukiwarkę frazy **Dr Dobbs CUDA**.

12.3.3. FORA NVIDIA

Nawet po dokładnym przestudiowaniu całej dokumentacji NVIDIA niektóre kwestie mogą być dla Ciebie niejasne. Jeśli przydarzy Ci się coś dziwnego i zechcesz się dowiedzieć, czy ktoś inny również miał podobny problem, albo planujesz świętowanie technologii CUDA i szukasz chętnych do udziału w obchodach, możesz poszukać pomocy na stronach NVIDIA. Doskonałym miejscem do zadawania pytań na temat CUDA innym użytkownikom tej technologii jest forum znajdujące się pod adresem <http://forums.nvidia.com>. W istocie po lekturze tej książki możesz nawet pomagać innym! Fora są regularnie przeglądane także przez pracowników firmy NVIDIA, a więc masz szansę otrzymać pewne informacje wprost ze źródła. Lubimy także, gdy użytkownicy piszą nam, co im się w naszych produktach podoba, a co nie.

12.4. Zasoby kodu źródłowego

Mimo że pakiet GPU Computing SDK to znakomite źródło przykładowych programów edukacyjnych, zawarte w nim materiały nie nadają się do użytku w rzeczywistych projektach. Dlatego jeśli szukasz programów i bibliotek CUDA przeznaczonych do wykorzystania w realnych zastosowaniach, musisz poszperać gdzie indziej. Społeczność programistów CUDA działa bardzo wcześniej i jej członkowie napisali już mnóstwo doskonałych programów. Poniżej przedstawiamy tylko kilka z nich, ale jeśli potrzebujesz czegoś innego, to śmiało szukaj tego w internecie. Może któregoś pięknego dnia sam opublikujesz jakiś wartościowy kawałek kodu w CUDA C!

12.4.1. BIBLIOTEKA CUDA PARALLEL PRIMITIVES LIBRARY

Firma NVIDIA we współpracy z University of California Davis opublikowała bibliotekę o nazwie CUDA Parallel Primitives Library (CUDPP), która, jak sama nazwa wskazuje, zawiera podstawowe operacje do równoległego wykonywania na danych. Wśród nich można znaleźć na przykład równoległą sumę prefiksową (scan), sortowanie równolegle i równoległą redukcję. Operacje tego typu stanowią podstawę wielu równoległych algorytmów operujących na danych, takich jak sortowanie, kompaktowanie strumieni, budowanie struktur danych i wiele innych. Jeśli planujesz napisanie choćby średnio skomplikowanego algorytmu, istnieje duża szansa, że jeśli w bibliotece CUDPP nie znajdziesz dokładnie tego, czego potrzebujesz, to przynajmniej znajdziesz coś, co znacznie przybliży Cię do celu. Bibliotekę możesz pobrać pod adresem <http://code.google.com/p/cudpp>.

12.4.2. CULATOOLS

W podrozdziale 12.2.3 napisaliśmy, że firma NVIDIA do pakietu CUDA Toolkit dołącza własną implementację biblioteki BLAS. Czytelnicy potrzebujący bardziej kompleksowego rozwiązania z zakresu algebra liniowej powinni jednak zaopatrzyć się w opracowaną przez firmę EM Photonics implementację w CUDA powszechnie znanej biblioteki Linear Algebra Package (LAPACK), znaną pod nazwą CULATools. Zawiera ona znacznie bardziej zaawansowane procedury z dziedziny algebra liniowej niż biblioteka CUBLAS NVIDIA. Dostępny bezpłatnie pakiet podstawowy zawiera algorytmy dekompozycji macierzy LU i QR, które rozwiążają układy równań liniowych, dokonują rozkładów SVD, a także realizują metodę najmniejszych kwadratów i ograniczoną metodę najmniejszych kwadratów. Tę podstawową wersję można pobrać pod adresem www.culatools.com/versions/basic. Firma oferuje także wersje Premium i Commercial zawierające implementacje znacznie większej części biblioteki LAPACK, których licencje pozwalają na komercyjne wykorzystanie swoich produktów napisanych za pomocą CULATools.

2.4.3. BIBLIOTEKI OSŁONOWE

W tej książce skoncentrowaliśmy się wyłącznie na językach C i C++, ale istnieje wiele projektów, które nie mają z nimi nic wspólnego. Na szczęście dostęp do technologii CUDA można zyskać także w językach, których NVIDIA oficjalnie nie obsługuje, a to dzięki różnym pośredniczym dostępnym bibliotekom osłonowym. Sama firma NVIDIA udostępnia wiązania dla języka FORTRAN do biblioteki CUBLAS, ale pod adresem www.jcuda.com można też znaleźć wiązania dla Javy do kilku bibliotek CUDA. Istnieją także biblioteki pozwalające używać jąder języka CUDA C w aplikacjach napisanych w Pythonie. Są one częścią projektu PyCUDA, którego strona znajduje się pod adresem <http://mathematician.de/software/pycuda>. Natomiast w ramach projektu CUDA.NET opracowano wiązania do CUDA dla środowiska Microsoft .NET www.hoopoe-cloud.com/Solutions/CUDA.NET.

Mimo że żaden z tych projektów nie jest oficjalnie wspierany przez NVIDIA, wszystkie istniejące od kilku wersji technologii CUDA, są darmowe i każdy z nich ma na swoim koncie sporo sukcesów. Morał z tego jest taki, że jeśli nie używasz w pracy języka C lub C++, to nie musisz d'rzuż rezygnować z programowania GPU. Sprawdź najpierw w internecie, czy ktoś nie udostępni odpowiednich wiązań dla języka, którego używasz.

2.5. Podsumowanie

Co masz za swoje. Po przeczytaniu 11 rozdziałów na temat programowania w języku CUDA dowiadujesz się, że jest jeszcze tyle do przeczytania, pobrania, obejrzenia i skompilowania. Wszakże jest doskonały czas na naukę programowania GPU, ponieważ era platform heterogenicznych wchodzi w wiek dojrzały. Mamy nadzieję, że nauka o jednym z najpopularniejszych środowisk programowania równoległego sprawiła Ci dużo przyjemności. Chcielibyśmy także, aby po przeczytaniu tej książki zaczął odkrywać nowe fascynujące możliwości programowania komputerów i przetwarzania coraz to większych ilości danych, których potrzebują najnowsze aplikacje. To właśnie dzięki Tobie i Twojej innowacyjności techniki programowania procesorów GPU mogą się dalej rozwijać.

Dodatek A

Operacje atomowe dla zaawansowanych

W rozdziale 9. poznaliśmy kilka technik wykorzystania operacji atomowych do bezpiecznego wykonywania setek równoległych operacji na wspólnych danych. W tym dodatku natomiast nauczysz się za ich pomocą implementować blokujące struktury danych. Pozornie temat ten nie jest bardziej skomplikowany niż wcześniej poruszane zagadnienia. W poprzednich rozdziałach opisaliśmy przecież wiele dość złożonych technik, więc z tym materiałem też nie powinno być większych problemów. A jednak z jakiegoś powodu odłożyliśmy go aż do dodatku. Dlaczego? Nie będziemy od razu zdradzać wszystkich sekretów, żeby nie zepsuć Ci zabawy. Jeśli jesteś ciekaw, to czytaj dalej, a wszystko stanie się jasne.

A.1. Iloczyn skalarny po raz kolejny

W rozdziale 5. opisaliśmy implementację w języku CUDA C algorytmu obliczającego iloczyn skalarny dwóch wektorów. Jest to przedstawiciel większej rodziny algorytmów, zwanych **redukcjami**. Przypomnijmy, jak on działa:

1. Każdy wątek w każdym bloku mnoży dwa odpowiadające sobie elementy z wektorów wejściowych, a następnie zapisuje iloczyn w pamięci wspólnej.
2. Mimo że każdy blok ma więcej niż jeden iloczyn, wątek sumuje po dwa z nich i wynik zapisuje z powrotem w pamięci wspólnej. Po zakończeniu każdego kroku liczba elementów jest o połowę mniejsza od tej, jaka była przed jego rozpoczęciem (stąd nazwa **redukcja**).
3. Po obliczeniu ostatecznej wartości każdy z bloków zapisuje swój wynik w pamięci globalnej i kończy działanie.
4. Jeśli jądro uruchomiło N równoległych bloków, procesor CPU ma do zsumowania N wartości, aby otrzymać ostateczny wynik.

Te cztery punkty mają Ci tylko nieco odświeżyć pamięć. Jeśli mimo to nic już nie pamiętasz, bo dawno czytałeś rozdział 5. albo jesteś po paru głębszych, to może warto poświęcić chwilę i przeczytać go jeszcze raz. Jeśli natomiast nie masz z tym żadnego problemu, to skoncentruj się głównie na punkcie czwartym powyższej listy. Mimo że ilość danych, które muszą zostać skopiowane do hosta, nie jest duża, a i późniejsze obliczenia na nich też nie są specjalnie skomplikowane, to jednak rozwiązanie to jest, delikatnie mówiąc, mało eleganckie.

A jednak niezgrabność czy brak elegancji w powyższym algorytmie to nasz najmniejszy problem. Wyobraź sobie taką sytuację, w której obliczanie iloczynu skalarnego stanowi tylko jeden z wielu elementów jakiejś dłuższej sekwencji zadań. Gdybyś chciał **wszystkie** te działania wykonać na GPU, ponieważ CPU byłby w tym czasie zajęty czymś innym, to miałbyś pecha. Musiałbyś zatrzymać obliczenia na GPU, skopiować wyniki pośrednie do hosta, dokończyć obliczenia za pomocą CPU, a na koniec wysłać wynik z powrotem do GPU i wznowić obliczenia procesora graficznego przy użyciu następnej funkcji jądra.

Biorąc pod uwagę, że tematem tego dodatku są operacje atomowe oraz to, jak wiele miejsca poświęciliśmy na miażdżący krytykę naszej pierwotnej implementacji algorytmu obliczającego iloczyn skalarny, możesz zacząć się domyślać, do czego zmierzamy. Mamy zamiar udoskonalić ten algorytm za pomocą operacji atomowych, tak aby wszystkie obliczenia były wykonywane na GPU, a CPU mógł w tym czasie swobodnie zająć się innymi zadaniami. Najlepiej by było, gdybyśmy — zamiast zamkać jądro w kroku trzecim i wracać do CPU w kroku czwartym — dodawali wynik każdego bloku do ostatecznego wyniku zapisanego w pamięci globalnej. Gdyby każda z tych operacji była wykonywana atomowo, nie byłoby problemu z kolizjami i nieprzewidywalnością tego, co się stanie. Ponieważ znamy już funkcję `atomicAdd()`, której używaliśmy przy obliczaniu histogramów, możemy wnioskować, że i tym razem sprawdzi się ona znakomicie.

Niestety funkcja `atomicAdd()` może być używana do pracy na liczbach innych niż całkowite wyłącznie na GPU z potencjałem obliczeniowym nie niższym niż 2.0. Jeśli planujesz obliczać iloczyny skalarnie wektorów zawierających tylko liczby całkowite, to oczywiście nie ma problemu, ale niestety znacznie częściej trzeba operować na liczbach zmiennoprzecinkowych. Jednak większość urządzeń NVIDIA nie wykonuje atomowych działań arytmetycznych na liczbach zmiennoprzecinkowych! Zanim jednak wyrzucisz swoją kartę grafiki do kosza, przeczytaj dalsze wyjaśnienia.

Podczas wykonywania operacji atomowej na wartości zapisanej w pamięci mamy jedynie gwarancję, że sekwencja operacji odczytu, modyfikacji i zapisu wykonującego to działanie wątku nie zostanie zakłócona przez żaden inny wątek. Nie ma natomiast żadnych zapewnień co do kolejności, w jakiej wątki będą wykonywać te działania. Dlatego przy trzech wątkach wykonujących atomowe dodawanie GPU czasami wykona operacje $(A+B)+C$, a czasami $A+(B+C)$. W przypadku liczb całkowitych nie ma to znaczenia, ponieważ ich dodawanie jest przemienne, tzn. $(A+B)+C = A+(B+C)$. Natomiast arytmetyka liczb zmiennoprzecinkowych **nie** jest przemienna ze względu na zaokrąglanie wyników pośrednich, a więc $(A+B)+C$ nie zawsze równa się $A+(B+C)$. W związku z tym jakość wyników działań wykonywanych na liczbach zmiennoprzecinkowych jest bardzo wątpliwa, ponieważ w środowiskach wielowątkowych, takich jak GPU, są one po-

prostu niedeterministyczne. W wielu programach sytuacja, w której to same działanie raz daje jeden wynik, a innym razem inny, jest niedopuszczalna. Dlatego też początkowo obsługa liczb zmiennoprzecinkowych nie była priorytetem konstruktorów GPU.

Jeśli mimo to jesteśmy skłonni zaakceptować pewien stopień nieokreśloności wyniku, to całość obliczeń możemy wykonać przy użyciu GPU. Wcześniej jednak musimy znaleźć sposób na ominięcie problemu z brakiem obsługi operacji atomowych na liczbach zmiennoprzecinkowych. Rozwiązanie, które zastosujemy, również będzie polegało na użyciu tych operacji, ale nie do wykonywania działań arytmetycznych.

A.1.1. BLOKADY ATOMOWE

Funkcja `atomicAdd()`, której użyliśmy do obliczania histogramów na GPU, wykonuje operację odczyt-modyfikacja-zapis jako jedno niepodzielne zadanie. Analizując jej działanie na najniższym poziomie, można sobie to wyobrazić tak, że podczas wykonywania tych operacji następuje blokada określonego fragmentu pamięci, dzięki czemu żaden inny wątek nie może zakłócić procesu. Gdyby udało się naśladować taką blokadę w funkcjach jądra CUDA C, to można by było na wybranych fragmentach pamięci lub strukturach danych wykonywać dowolne operacje. Mechanizm blokujący, który mamy na myśli, będzie działał tak samo jak mutexy CPU. Jeśli nie wiesz, czym są blokady wzajemnego wykluczania, czyli właśnie **mutexy**, to nie masz powodu do obaw. To nic strasznego.

Wszystko sprowadza się do tego, że alokujemy niewielką ilość pamięci, której będziemy używać jako mutexu. Mutex ten, zarządzając dostępem do wybranego zasobu, będzie pełnił rolę podobną do sygnalizatora świetlnego. Zasobem tym może być struktura danych, bufor albo po prostu miejsce w pamięci, którego zawartość ma być modyfikowana atomowo. Gdy wątek odczyta z mutexu wartość 0, zinterpretuje ją jako „zielone światło” oznaczające, że żaden inny wątek w danym momencie nie korzysta z wyznaczonego obszaru pamięci. Wówczas może go zablokować i zrobić z nim, co chce, bez obaw, że jakiś inny wątek mu w tym przeszkodzi. W celu zablokowania tej pamięci wątek zapisuje w mutexie wartość 1, która dla innych wątków będzie oznaczała „czerwone światło”. Te inne wątki będą ze swoimi operacjami musiały poczekać, aż wątek będący aktualnie w posiadaniu zasobu zwolni go poprzez zapisanie w mutexie wartości 0.

Prosty kod realizujący to zadanie mógłby wyglądać następująco:

```
void lock( void ) {
    if( *mutex == 0 ) {
        *mutex = 1; // Zapisanie wartości 1 w celu zablokowania dostępu do zasobu
    }
}
```

Niestety ten kod ma jedną poważną wadę, ale na szczęście jest to dobrze znany problem. Wyobraź sobie, co się stanie, gdy po odczytaniu przez jeden wątek wartości 0 inny wątek zapisze w mutexie 1. To znaczy, dwa wątki sprawdzą wartość zmiennej mutex i dowiadują się, że wynosi

ona 0. Skoro tak, to oba zapisują w niej 1, aby powiadomić pozostałe wątki, że struktura jest zablokowana i nie można jej modyfikować. Po dokonaniu tego oba wątki „myślą”, że mają związek z blokadą zasobów na wyłączność i zaczynają dokonywać modyfikacji, które nie są bezpieczne. To oczywiście powoduje katastrofę!

Operacja, którą chcemy wykonać, jest tak naprawdę bardzo prosta: porównujemy wartość zmiennej `mutex` z zerem i zapisujemy w niej 1 wtedy i tylko wtedy, gdy wynik porównania był pozytywny. Aby to się udało, operacja musi zostać wykonana atomowo, tak aby żaden inny wątek nic nie zmienił, podczas gdy nasz wątek sprawdza i zmienia wartość zmiennej `mutex`. W języku CUDA C można do tego celu wykorzystać funkcję `atomicCAS()`, czyli atomową operację porównaj-i-zamień. Funkcja ta przyjmuje jako argumenty wskaźnik na miejsce w pamięci, wartość, z którą ma zostać porównana zawartość tego miejsca, oraz wartość, jaka ma w nim zostać zapisana, jeśli wynik porównania będzie pozytywny. Przy użyciu tej konstrukcji funkcję blokady dla GPU możemy zaimplementować następująco:

```
_device_ void lock( void ) {
    while( atomicCAS( mutex, 0, 1 ) != 0 );
}
```

Powyższe wywołanie funkcji `atomicCAS()` zwróci wartość znalezioną pod adresem `mutex`. Pętla `while()` będzie kontynuowała działanie do chwili, aż funkcja `atomicCAS()` znajdzie w zmiennej `mutex` wartość 0. Gdy to nastąpi, wynik porównania stanie się pozytywny i wątek zapisze w zmiennej `mutex` wartość 1. Krótko mówiąc, wątek ten będzie wykorzystywał pętlę `while()` do poczekania na moment, w którym uda mu się zablokować strukturę danych dla siebie. Przedstawionego mechanizmu użyjemy do implementacji tablicy skrótów na GPU. Najpierw jednak przeniesiemy ten kod do struktury, aby łatwiej się nam z niego korzystało przy obliczaniu iloczynu skalarnego:

```
struct Lock {
    int *mutex;
    Lock( void ) {
        int state = 0;
        HANDLE_ERROR( cudaMalloc( (void**)& mutex,
                                sizeof(int) ) );
        HANDLE_ERROR( cudaMemcpy( mutex, &state, sizeof(int),
                                cudaMemcpyHostToDevice ) );
    }
    ~Lock( void ) {
        cudaFree( mutex );
    }
    _device_ void lock( void ) {
        while( atomicCAS( mutex, 0, 1 ) != 0 );
    }
    _device_ void unlock( void ) {
        atomicExch( mutex, 0 );
    }
};
```

Zwróć uwagę, że wartość zmiennej `mutex` przywracamy za pomocą wywołania funkcji `atomicExch(mutex, 0)`. Funkcja ta wczytuje wartość zapisaną pod adresem `mutex`, zamienia ją na swój drugi argument (w tym przypadku 0) i zwraca pierwotną wartość, którą odczytała. Dlaczego użyliśmy tej atomowej funkcji zamiast poniższego, wydawałoby się oczywistego, wyrażenia?

```
*mutex = 0;
```

Jeśli oczekujesz, że zaraz się dowiesz o jakimś trudno uchwytnym powodzie, dla którego powyższe wyrażenie miałyby powodować problemy, to musimy Cię rozczarować. Ono również jest poprawne. Dlaczego zatem go nie użyliśmy? Transakcje atomowe i zwykłe operacje na pamięci globalnej w procesorach GPU to dwa różne światy. Gdybyśmy na pamięci globalnej użyli zarówno operacji atomowych, jak i zwykłych operacji, mogliby się wydawać, że funkcja `unlock()` nie ma nic wspólnego z funkcją `lock()`. Nie spowodowałoby to błędów w funkcjonowaniu programu, ale stosując tę samą metodę wykonywania działań we wszystkich miejscach, zapewniłyby większą spójność programowi. Krótko mówiąc, skoro użyliśmy operacji atomowej do zablokowania zasobu, to do jego odblokowania też użyjemy operacji tego samego typu.

A.1.2. ILOCZYN SKALARNY: BLOKADY ATOMOWE

Większość pierwotnej implementacji programu pozostawiamy bez zmian, a zmodyfikujemy tylko jego końcową część, w której przenosiliśmy się do CPU. W poprzednim podrozdziale opisaliśmy sposób implementacji mutexu na GPU. Strukturę `Lock` zawierającą tę implementację zapisaliśmy w pliku o nazwie `lock.h`, który dołączamy na początek nowej wersji programu:

```
#include "../common/book.h"
#include "lock.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
```

Początek kodu źródłowego funkcji jądra jest prawie taki sam jak w rozdziale 5., jedynie z dwoma wyjątkami w sygnaturze:

```
_global_ void dot( Lock lock, float *a, float *b, float *c )
```

W tej wersji programu funkcji jądra oprócz buforów wejściowych i wyjściowego przekazujemy jeszcze strukturę `Lock`. Będzie ona zarządzać dostępem do bufora wyjściowego podczas ostatniego etapu, w którym obliczany jest ostateczny wynik. Druga zmiana, mimo że jej **nie widać** w sygnaturze, też jej dotyczy. Poprzednio argument `float *c` był buforem `N` liczb zmiennoprzecinkowych, w którym każdy z `N` bloków mógł zapisać swój wynik cząstkowy. Bufor ten był kopiowany do CPU w celu dokonania obliczeń. Teraz argument ten nie wskazuje już tymczasowego

bufora, lecz pojedynczą wartość zmiennoprzecinkową, która będzie reprezentowała iloczyn skalarny buforów a i b. Ale nawet z tymi zmianami początek funkcji jądra pozostaje taki sam jak w rozdziale 5.:

```
_global_ void dot( Lock lock, float *a,
                    float *b, float *c ) {
    _shared_ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    // Ustawienie wartości pamięci podręcznej
    cache[cacheIndex] = temp;

    // Synchronizacja wątków w tym bloku
    __syncthreads();
    // W przypadku redukcji threadsPerBlock musi być potęgą 2
    // ze względu na poniższy kod.
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
}
```

W tym punkcie wykonywania programu wszystkie 256 wątków każdego bloku zsumowało już 256 swoich iloczynów i obliczyło pojedynczą wartość, która jest zapisana w elemencie cache[0]. Teraz każdy z tych bloków musi dodać swoją wartość do ostatecznego wyniku w zmiennej c. Aby sumowanie to odbyło się w sposób bezpieczny, dostępem do tego miejsca w pamięci będziemy sterować za pomocą blokady, tak że każdy wątek przed zmodyfikowaniem wartości *c będzie musiał najpierw uzyskać do niej dostęp na wyłączność. Po dodaniu swojej cząstkowej wartości do c blok zdejmuję blokadę z mutexu, aby pozostałe wątki również mogły dodać swoje wartości. Następnie blok nie ma już nic do zrobienia, więc kończy działanie.

```
if (cacheIndex == 0) {
    lock.lock();
    *c += cache[0];
    lock.unlock();
}
}
```

Także procedura main() jest bardzo podobna do pierwotnego, aczkolwiek występują między nimi pewne różnice. Po pierwsze nie ma już potrzeby alokować bufora na wyniki cząstkowe. Teraz potrzebujemy tylko miejsca na jedną wartość zmiennoprzecinkową:

```
int main( void ) {
    float *a, *b, c = 0;
    float *dev_a, *dev_b, *dev_c;
    // Alokacja pamięci po stronie CPU
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    // Alokacja pamięci na GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                             N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                             N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
                             sizeof(float) ) );
```

Podobnie jak w rozdziale 5., inicjujemy nasze tablice wejściowe i kopujemy je do GPU. Lecz tym razem wykonana została jeszcze jedna operacja kopiowania: wartości 0 do dev_c, czyli miejsca w pamięci, w którym będziemy sumować ostateczny wynik obliczeń. Zmienna ta musi mieć początkową wartość 0, ponieważ każdy blok musi ją odczytać, dodać do niej swoją sumę cząstkową i z powrotem zapisać w niej wynik:

```
// Napełnienie pamięci hosta danymi
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}
// Skopiowanie tablic a i b do GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                         cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                         cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_c, &c, sizeof(float),
                         cudaMemcpyHostToDevice ) );
```

Pozostało jeszcze tylko zadeklarować strukturę Lock, wywołać jądro i skopiować wynik do CPU:

```
Lock lock;
dot<<<blocksPerGrid,threadsPerBlock>>>( lock, dev_a,
                                              dev_b, dev_c );

// Skopiowanie c z GPU do CPU
HANDLE_ERROR( cudaMemcpy( &c, dev_c,
                         sizeof(float),
                         cudaMemcpyDeviceToHost ) );
```

W rozdziale 5. w tym miejscu nastąpiłoby wykonanie ostatniej pętli for() sumującej wyniki cząstkowe. Ponieważ czynność ta została już wykonana na GPU, możemy ominąć ten krok i przejść od razu do sprawdzania rezultatu i porządkowania:

```

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Czy wartość obliczona na GPU wynosi %.6g = %.6g?\n", c,
       2 * sum_squares( (float)(N - 1) ) );
// Zwolnienie pamięci na GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
// Zwolnienie pamięci po stronie CPU
free( a );
free( b );
}

```

Ponieważ nie da się dokładnie określić kolejności, w jakiej poszczególne bloki będą dodawać swoje wartości do ostatecznego wyniku, jest praktycznie pewne, że na GPU operacje dodawania zostaną wykonane w innej kolejności niż na CPU. W związku z tym istnieje bardzo duże prawdopodobieństwo, że ze względu na brak łączności operacji dodawania liczb zmiennoprzecinkowych wynik uzyskany na GPU będzie się nieco różnił od wyniku uzyskanego na CPU. Problem ten można rozwiązać tylko poprzez dodanie skomplikowanego kodu zapewniającego dostęp bloków do blokady w określonej kolejności, takiej samej jak kolejność sumowania wyników cząstkowych przez CPU. Jeśli nie boisz się wyzwań, możesz spróbować coś takiego napisać. Teraz jednak przejdziemy do sposobów wykorzystania blokad w celu zaimplementowania wielowątkowej struktury danych.

A.2. Implementacja tablicy skrótów

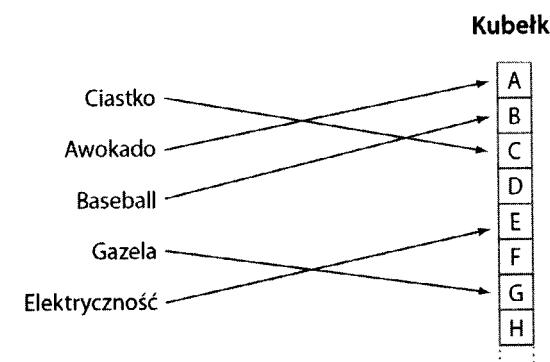
Tablica skrótów to jedna z najważniejszych i najczęściej używanych struktur danych w informatyce, odgrywająca dużą rolę w wielu dziedzinach nauki. Dla czytelników nieobeznanych z tematem przedstawiamy poniżej krótki kurs wstępny. Oczywiście musimy znacznie spłycić problem, ale wiedza, którą tu zaprezentujemy, wystarczy do tego, aby zrozumieć dalsze rozważania. Czytelnicy znający zasady działania tablic skrótów mogą przejść od razu do podrozdziału A.2.2.

A.2.1. TABLICE SKRÓTÓW — WPROWADZENIE

Tablica skrótów to w zasadzie struktura danych służąca do przechowywania par **klucz-wartość**. Na przykład słownik można traktować jako realizację tej struktury. Każde hasło w słowniku jest **kluczem**, z którym jest powiązana jakaś definicja, czyli **wartość**, i w ten sposób hasła i definicje tworzą pary klucz-wartość. Aby taka struktura danych była użyteczna, należy zminimalizować czas dostępu do wartości wg kluczy. Ogólnie rzecz biorąc, czas ten powinien być stały, tzn. dostęp do wartości po kluczu powinien zajmować tyle samo czasu bez względu na to, ile par klucz-wartość znajduje się w danym momencie w tablicy.

Operując abstrakcyjnymi pojęciami mówimy, że wartości zapisywane są w „kubelkach” (ang. *bucket*) wg ich kluczy. Metoda wybierania kubelka, do którego ma zostać wstawiona dana wartość, nazywana jest **funkcją skrótu** (ang. *hash function*). Dobrze skonstruowana funkcja skrótu rozmieszcza zbiór kluczy w kubelkach równomiernie, co umożliwia spełnienie warunku niezmienności czasu dostępu do wartości bez względem na liczbę elementów tablicy.

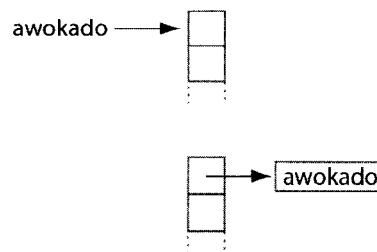
Weźmy na przykład tablicę skrótów dla słownika liter alfabetu łacińskiego. Jednym z automatycznie nasuwających się rozwiązań jest użycie 26 kubelków, po jednym dla każdej litery. Taka prosta funkcja skrótu mogłaby po prostu sprawdzać pierwszą literę klucza i umieszczać odpowiadającą mu wartość w jednym z 26 kubelków. Na rysunku A.1 przedstawiono jedną z możliwości przyporządkowania kilku wyrazów.



Rysunek A.1. Rozmieszczenie słów w kubelkach

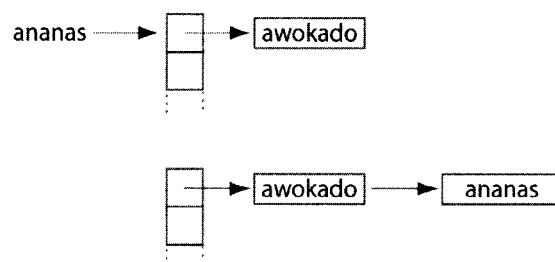
Biorąc jednak pod uwagę liczbę słów zaczynających się na różne litery, od razu wiadomo, że ta funkcja nie rozmieści wyrazów równomiernie między te 26 kubelkami. Niektóre z nich będą zawierały bardzo mało par klucz-wartość, a inne znacznie więcej. W związku z tym wyszukanie definicji odpowiadającej słowu zaczynającemu się na jakąś często występującą na początku literę, np. S, zajmie znacznie więcej czasu niż wyszukanie definicji słowa zaczynającego się np. na literę Y. A ponieważ nam potrzebna jest funkcja pozwalająca uzyskać stały czas dostępu do elementów, takie działanie jak opisane powyżej jest niedopuszczalne. Nad funkcjami skrótu przeprowadzono bardzo wiele badań, dlatego nawet powierzchowne nakreślenie tej problematyki wykracza poza ramy tematyczne niniejszej książki.

Ostatni składnik naszej struktury danych jest związany z kubelkami. Gdybyśmy mieli doskonałą funkcję skrótu, to każdy klucz powinien odnosić się do innego kubelka. Wówczas moglibyśmy po prostu pary klucz-wartość zapisać w tablicy, w której każdy element byłby tym, co do tej pory nazywaliśmy **kubikiem**. A jednak nawet najlepsza funkcja nie uchroni nas przed **kolizjami**. Kolizja to sytuacja, w której do jednego kubelka odnosi się więcej niż jeden klucz, np. gdybyśmy do naszego słownika dodali słowa **awokado** i **ananas**. Najprostszym sposobem przechowywania wszystkich wartości należących do danego kubelka jest zapisanie ich w liście w tym kubelku. Gdy wystąpi kolizja, jak w przypadku dodania słowa ananas do słownika zawierającego słowo awokado, wartość związaną z ananasem powinniśmy umieścić na końcu listy znajdującej się w kubelku A. Spójrz na rysunek A.2.



Rysunek A.2. Wstawianie słowa awokado do tablicy skrótów

Po dodaniu słowa **awokado** do słownika pierwszy kubek zawiera w swojej liście tylko jedną parę klucz-wartość. Następnie dodajemy do słownika słowo **ananas**, które koliduje ze słowem awokado, ponieważ również zaczyna się na literę A. Na rysunku A.3 widać, że w takim przypadku słowo to zostaje po prostu wstawione na końcu listy pierwszego kubelka:



Rysunek A.3. Rozwiązywanie konfliktu spowodowanego dodaniem do słownika słowa ananas

Tak z grubsza wyglądają podstawy działania funkcji skrótu i mechanizmu rozwiązywania kolizji. Uzbrojeni w tę wiedzę możemy rozpocząć implementację własnej struktury danych.

A.2.2. TABLICA SKRÓTÓW DLA CPU

Zgodnie z tym, co napisaliśmy w poprzednim podrozdziale, dwiema głównymi częściami naszej tablicy skrótów będą funkcja skrótu i struktura danych złożona z kubelków. Kubełki zaimplementujemy dokładnie tak samo jak poprzednio — alokujemy tablicę o długości N i w każdym elemencie tej tablicy zapiszemy listę par klucz-wartość. Najpierw zajmiemy się strukturą danych, a do funkcji przejdziemy później:

```
#include "../common/book.h"
struct Entry {
    unsigned int    key;
    void*          value;
    Entry         *next;
};
struct Table {
    size_t        count;
    Entry        **entries;
```

```
Entry    *pool;  
Entry    *firstFree;
```

Tak jak napisaliśmy wcześniej, struktura `Entry` zawiera zarówno klucze, jak i wartości. W naszym programie do przechowywania par klucz-wartość używamy kluczy typu całkowitoliczbowego bez znaku. Wartość związana z kluczem może być dowolnego typu i dlatego użyliśmy dla niej typu `void*`. Ponieważ nasza aplikacja ma za zadanie tylko utworzenie samej struktury danych, w polu `value` tak naprawdę nie będziemy niczego zapisywać. Dodaliśmy je tylko po to, żeby niczego nie brakowało, i na wypadek gdybyś chciał użyć tego kodu w swoich programach. Na końcu omawianej struktury znajduje się wskaźnik na następną strukturę `Entry`. Gdy wystąpią kolizje, w każdym kubelku będzie mogło się znajdować po kilka pozycji, a wcześniej zdecydowaliśmy, że będziemy je przechowywać w formie listy. Zatem każdy element będzie wskazywał na kolejny element w kubelku, w wyniku czego powstanie listy elementów przyporządkowanych do tego samego kubelka w tablicy. Ostatni element będzie zawierał wskaźnik pusty.

W uproszczeniu można powiedzieć, że struktura `Table` jest w istocie tablicą „kubelków”. Jej rozmiar wynosi `count`, a każdy kubek w `entries` jest tylko wskaźnikiem na strukturę `Entry`. Aby uniknąć komplikacji i strat wydajności związanych z alokacją pamięci za każdym razem, gdy dodawany jest nowy element, będziemy utrzymywać dużą tablicę dostępnych elementów w puli (`pool`). Pole `firstFree` wskazuje na następny element dostępny do użytku, a zatem gdy trzeba dodać nowy element do tablicy, możemy po prostu użyć elementu wskazywanego przez to pole, a następnie zwiększyć ten wskaźnik. Zwróć uwagę, że umożliwia to również uproszczenie kodu, ponieważ wszystkie elementy będzie można zwolnić za pomocą jednego wywołania funkcji `free()`. Gdybyśmy każdy element alokowali na bieżąco, musielibyśmy je potem pojedynczo zwalniać, przeglądając całą tablicę element po elemencie.

Wiedząc, jak wygląda z grubsza budowa struktur danych, możemy przejść do analizy innych części kodu źródłowego:

```
void initialize_table( Table &table, int entries,
                      int elements ) {
    table.count = entries;
    table.entries = (Entry**)calloc( entries, sizeof(Entry*) );
    table.pool = (Entry*)malloc( elements * sizeof( Entry ) );
    table.firstFree = table.pool;
}
```

Inicjacja tablicy skrótów polega głównie na alokowaniu pamięci i oczyszczeniu pamięci dla tablicy kubełków `entries`. Ponadto alokujemy miejsce dla puli elementów oraz inicjujemy wskaźnik `firstFree`, aby wskazywał pierwszy element w tej puli.

Na końcu programu musimy zwolnić alokowaną pamięć, a więc piszemy procedurę porządkującą, która będzie zwalniała tablice kubiełków i pudełek wolnych elementów:

```

void free_table( Table &table ) {
    free( table.entries );
    free( table.pool );
}

```

po rozmieszczeniu poro miejsca poświęciliśmy funkcji skrótu, a w szczególności rozwdzieliśmy się nad tym, jak d jej jakości zależy jakość całej implementacji tablicy skrótów. W tym przypadku jako klucz żywym liczb całkowitych bez znaku, które musimy rzutować na indeksy naszej tablicy kubelków. Najprostszym sposobem na zrobienie tego byłoby wybranie kubelka o indeksie równym kluczowi, tzn. element e moglibyśmy zapisać w elemencie table.entries[e.key]. Jednakże nie możemy zakładać, że wszystkie klucze będą miały wartości niższe niż długość tablicy kubelków. Na szczęście rozwiązanie tego problemu jest względnie proste:

```

size_t hash( unsigned int key, size_t count ) {
    return key % count;
}

```

Korzystając z tej funkcji skrótu, to jak to możliwe, że wystarczy nam taka prosta procedura? Wiemy już, że najlepiej jest wtedy, gdy klucze zostaną równomiernie rozproszone w kubelkach, a powyższa funkcja po prostu wykonuje dzielenie modulo klucza przez długość tablicy. W rzeczywistości funkcje skrótu nie są tak banalnie proste, ale ponieważ to jest tylko wersja demonstracyjna, klucze będziemy generować losowo. Jeśli przyjmiemy, że generator liczb losowych losuje, z grubsza rzecz biorąc, wartości o dość równomiernym rozproszeniu, powyższa funkcja powinna równomiernie rozmieszczać klucze w kubelkach. Gdy będziesz implementować własną tablicę skrótów, zapewne użyjesz bardziej skomplikowanej funkcji.

Teraz mamy już struktury danych i funkcję skrótu, a więc możemy przejść do algorytmu dodawania klucz-wartość do tablicy. Będzie to proces trzystopniowy:

1. Wywołanie funkcji skrótu na kluczu w celu określenia kubelka dla nowego elementu.
2. Pobranie alokowanej wcześniej struktury Entry z puli i zainicjowanie jej pól key i value.
3. Wstawienie elementu na początek listy odpowiedniego kubelka.

O przetłumaczeniu tego na kod źródłowy otrzymujemy następujący wynik:

```

void add_to_table( Table &table, unsigned int key, void* value )
{
    // Krok 1
    size_t hashValue = hash( key, table.count );
    // Krok 2
    Entry *location = table.firstFree++;
    location->key = key;
    location->value = value;
    // Krok 3
    location->next = table.entries[hashValue];
    table.entries[hashValue] = location;
}

```

Dla osób, które nigdy nie miały do czynienia z listami powiązanymi, krok trzeci może być niejasny. Pierwszy węzeł istniejącej listy zostaje zapisany w elemencie table.entries[hashValue]. Wiedząc o tym, kolejny węzeł na początku listy możemy dodać w dwóch krokach: ustawiając wskaźnik next nowego elementu na pierwszy węzeł istniejącej listy, a następnie zapisując nowy element w tablicy kubelków, tak aby stał się pierwszym węzłem nowej listy.

Ponieważ dobrze jest wiedzieć, czy nowo napisany kod działa, napisaliśmy też specjalną procedurę wykonującą podstawowy test naszej tablicy skrótów. Jej działanie polega na tym, że przegląda tablicę i sprawdza po kolei każdy węzeł. Wywołujemy funkcję skrótu na kluczu każdego węzła, aby zweryfikować, czy jest on zapisany we właściwym miejscu. Po wykonaniu tego testu sprawdzamy jeszcze, czy liczba węzłów **rzeczywiście** zapisanych w tablicy zgadza się z liczbą elementów, jaką **chcieliśmy** wstawić. Jeśli wystąpią rozbieżności, będzie to oznaczało, że albo jakiś węzeł został przypadkowo dodany do kilku kubelków, albo został wstawiony nieprawidłowo.

```

#define SIZE      (100*1024*1024)
#define ELEMENTS   (SIZE / sizeof(unsigned int))
void verify_table( const Table &table ) {
    int count = 0;
    for (size_t i=0; i<table.count; i++) {
        Entry  *current = table.entries[i];
        while (current != NULL) {
            ++count;
            if (hash( current->value, table.count ) != i)
                printf( "%d przyporządkowany do %ld, ale znajdował się "
                        "w %ld\n", current->value,
                        hash( current->value, table.count ), i );
            current = current->next;
        }
    }
    if (count != ELEMENTS)
        printf( "W tablicy skrótów znaleziono %d elementów. Powinno być %ld\n",
                count, ELEMENTS );
    else
        printf( "W tablicy skrótów znaleziono wszystkie %d elementów.\n",
                count );
}

```

Teraz możemy przejść do funkcji main(). Podobnie jak w przypadku wielu innych programów prezentowanych w tej książce, większość pracy wykonują funkcje pomocnicze, dzięki czemu kod funkcji głównej nie jest zbyt skomplikowany:

```

#define HASH_ENTRIES 1024
int main( void ) {
    unsigned int *buffer =
        (unsigned int*)big_random_block( SIZE );
    clock_t start, stop;
    start = clock();
    Table table;
}

```

```

initialize_table( table, HASH_ENTRIES, ELEMENTS );
for (int i=0; i<ELEMENTS; i++) {
    add_to_table( table, buffer[i], (void*)NULL );
}
stop = clock();
float elapsedTime = (float)(stop - start) /
    (float)CLOCKS_PER_SEC * 1000.0f;
printf( "Czas wstawiania elementów: %.3f ms\n", elapsedTime );
verify_table( table );
free_table( table );
free( buffer );
return 0;
}

```

Funkcja rozpoczyna działanie od alokowania dużej ilości losowych liczb całkowitych bez znaku. Zostaną one wykorzystane jako klucze w naszej tablicy skrótów. Po wygenerowaniu liczb sprawdzamy czas systemowy, aby zmierzyć wydajność naszej implementacji. Inicjujemy tablicę skrótów, a następnie za pomocą pętli for() wstawiamy do niej klucze. Później ponownie sprawdzamy czas systemowy, aby móc obliczyć, ile czasu zajęło programowi zainicjowanie i wstawienie kluczów. Na zakończenie przeprowadzamy podstawowy test naszej tablicy i zwalniamy alokowane bufore.

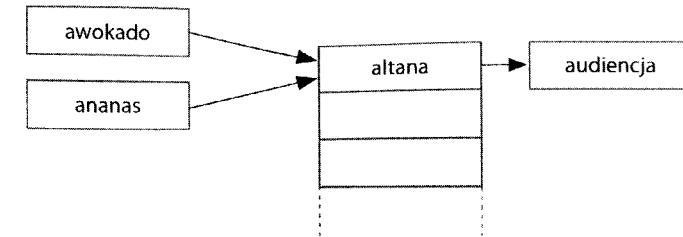
Zapewne zauważyłeś, że w każdej parze klucz-wartość jako wartości używamy słowa kluczowego NULL. Oczywiście w realnym programie tablicy skrótów do przechowywania używa się czegoś bardziej przydatnego, ale ponieważ nas interesuje sama implementacja struktury danych, a nie jej napełnianie, zapisujemy w niej tylko to.

A.2.3. WIELOWĄTKOWA TABLICA SKRÓTÓW

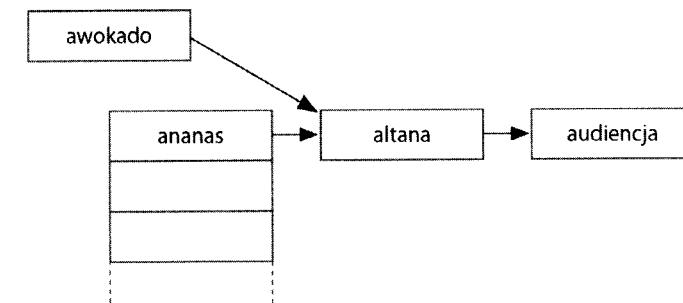
W implementacji naszej tablicy skrótów dla CPU przyjęliśmy kilka założeń, które po przeprowadzce na GPU stracą ważność. Po pierwsze przyjęliśmy, że w danym czasie do tablicy może być dodawany tylko jeden węzeł, co pozwoliło nam uprościć nieco kod odpowiadający za dodawanie węzłów. Gdyby kilka wątków jednocześnie spróbował dodać węzeł do tablicy, moglibyśmy mieć podobny problem jak z dodawaniem opisanym w rozdziale 9.

Wróćmy jeszcze na chwilę do przykładu z owocami. Wyobraź sobie, że wątki A i B próbują dodać do tablicy słowa awokado i ananas. Wątek A wywołuje funkcję skrótu na **awokado**, a wątek B — na słowie **ananas**. W obu przypadkach zapada decyzja, że klucz należy do pierwszego kubelka. Aby dodać nowy element do listy, każdy z wątków najpierw ustawia wskaźnik next tego swojego elementu na pierwszy węzeł istniejącej listy — rysunek A.4.

Następnie oba wątki próbują zamienić element znajdujący się w kubelku swoim własnym elementem. Lecz zapisany zostanie tylko element tego wątku, który zakończy działanie jako ostatni, oni eważ nadpisze on to, co zrobił poprzedni wątek. Wyobraźmy sobie zatem sytuację, w której wątek A zamienia element **altana** na element **awokado**. Kiedy tylko skończy, wątek B zamienia go, co w jego mniemaniu jest elementem **altana**, na swój element **ananas**. Niestety zamiast altany odbędzie się awokado, czego efektem będzie sytuacja przedstawiona na rysunku A.5.



Rysunek A.4. Dwa wątki próbujące dodać węzeł do tego samego kubelka



Rysunek A.5. Tablica skrótów po nieudanej modyfikacji przez dwa współbieżne wątki

Element wątku A zostanie „wypchnięty” poza tablicę. Całe szczęście, że nasz podstawowy test poprawności wykryłby taką sytuację, ponieważ by nas poinformował, że w tablicy jest mniej elementów, niż powinno być. No dobrze, ale jak w takim razie zaimplementować tablicę skrótów dla GPU? Z powyższych rozważań należy przede wszystkim wyciągnąć wniosek, że dany kubelk może być modyfikowany jednocześnie tylko przez jeden wątek. Jest to sytuacja podobna do tej z obliczaniem iloczynu skalarnego, gdy tylko jeden wątek mógł bezpiecznie dodawać swój wynik częściowy do wyniku ostatecznego. Aby zabezpieczyć strukturę danych przed nieprawidłowym modyfikowaniem kubelków, możemy z każdym z nich związać blokadę atomową.

A.2.4. TABLICA SKRÓTÓW DLA GPU

Skoro wiemy już, jak zaprojektować bezpieczną wątkowo implementację tablicy skrótów dla GPU, możemy zmodyfikować program z podrozdziału A.2.2, który był przeznaczony do działania na CPU. Do nowej wersji aplikacji musimy dołączyć plik *lock.h* zawierający strukturę Lock napisaną w podrozdziale A.1.1. Ponadto funkcję skrótu oznaczmy kwalifikatorem *_device_*. Oprócz tych dwóch zmian reszta kodu będzie dokładnie taka sama jak poprzednio:

```

#include "../common/book.h"
#include "lock.h"
struct Entry {
    unsigned int key;
    void* value;
}

```

```

    Entry          *next;
};

struct Table {
    size_t count;
    Entry **entries;
    Entry *pool;
};

__device__ __host__ size_t hash( unsigned int value
                               size_t count ) {

    return value % count;
}

```

Inicjacja i zwalnianie tablicy skrótów będą się odbywać tak samo jak na CPU, tylko oczywiście przy użyciu funkcji systemu wykonawczego CUDA. Do alokowania tablic kubełków i puli wolnych elementów użyjemy funkcji `cudaMalloc()`, natomiast do ustawienia elementów na zero skorzystamy z funkcji `cudaMemset()`. Po zakończeniu działania programu pamięć zwolnimy za pomocą funkcji `cudaFree()`:

```

void initialize_table( Table &table, int entries,
                      int elements ) {
    table.count = entries;
    HANDLE_ERROR( cudaMalloc( (void**)&table.entries,
                             entries * sizeof(Entry*) ) );
    HANDLE_ERROR( cudaMemcpy( table.entries, 0,
                            entries * sizeof(Entry*) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&table.pool,
                             elements * sizeof(Entry)) );
}

void free_table( Table &table ) {
    cudaFree( table.pool );
    cudaFree( table.entries );
}

```

W implementacji dla CPU mieliśmy specjalną procedurę weryfikującą poprawność naszej struktury danych. Podobną musimy napisać też dla GPU i mamy w tym zakresie dwie możliwości do wyboru: napisać nową wersję funkcji `verify_table()` dla GPU lub użyć poprzedniej oraz dodać funkcję kopiującą tablice skrótów z GPU do CPU. Oba wyjścia są możliwe do realizacji, lecz drugie z nich wydaje się lepsze z dwóch powodów. Po pierwsze pozwala nam ponownie wykorzystać już napisaną funkcję, dzięki czemu oszczędzimy na czasie, a dodatkowo gdy wprowadzimy jakieś zmiany, to wystarczy, że dokonamy tego tylko w jednym miejscu, bo będziemy mieli tylko jedną funkcję weryfikującą. Po drugie podczas implementowania funkcji kopiującej odkryjemy ciekawy problem, którego rozwiążanie może przydać Ci się w przyszłości.

Zgodnie z zapowiedzią funkcja `verify_table()` pozostaje bez zmian. Poniżej dla wygody prezentujemy jej kod jeszcze raz:

```

#define SIZE      (100*1024*1024)
#define ELEMENTS    (SIZE / sizeof(unsigned int))
#define HASH_ENTRIES   1024
void verify_table( const Table &dev_table ) {
    Table table;
    copy_table_to_host( dev_table, table );
    int count = 0;
    for (size_t i=0; i<table.count; i++) {
        Entry *current = table.entries[i];
        while (current != NULL) {
            ++count;
            if (hash( current->value, table.count ) != i)
                printf( "%d przyporządkowany do %ld, ale znajdował się "
                        "w %ld\n",
                        current->value,
                        hash(current->value, table.count), i );
            current = current->next;
        }
    }
    if (count != ELEMENTS)
        printf("W tablicy skrótów znalezione %d elementów. Powinno być %ld\n",
               count, ELEMENTS );
    else
        printf( "W tablicy skrótów znalezione wszystkie %d elementów.\n",
               count );
    free( table.pool );
    free( table.entries );
}

```

Ponieważ zdecydowaliśmy się na wykorzystanie implementacji dla CPU funkcji weryfującej poprawność struktury danych, musimy napisać funkcję kopującą tablicę z GPU do pamięci hosta. Jej działanie będzie składać się z trzech kroków, przy czym dwa pierwsze będą dość oczywiste, natomiast trzeci trochę bardziej skomplikowany. W związku z powyższym najpierw alokujemy na hoście pamięć, w której zapiszemy skopiowaną tablicę skrótów, a następnie skopiujemy do niej dane z GPU za pomocą funkcji `cudaMemcpy()`. Robiliśmy to już tyle razy, że chyba nic nas tu nie może zaskoczyć:

udność, o której wspominaliśmy, polega na tym, że część skopiowanych danych to wskaźniki. jak wiemy, wskaźników nie można tak po prostu skopiować, ponieważ zawierają one adresy pamięci GPU i na hoście staną się bezużyteczne. Natomiast wzajemne relacje między nimi są zostoną utracone nawet po skopiowaniu. Każdy wskaźnik na strukturę Entry wskazuje na miejsce w tablicy `table.pool[]`, ale żeby nasza tablica skrótów była także użyteczna na siebie, muszą one wskazywać te same struktury Entry w tablicy `hostTable.pool[]`.

li więc mamy dany wskaźnik X i chcemy przekształcić go w prawidłowy wskaźnik na hoście, musimy dodać jego pozycję z `table.pool` do `hostTable.pool`. Innymi słowy, nowy adres wskaźnika powinniśmy obliczyć w następujący sposób:

```
(X - table.pool) + hostTable.pool
```

terację tę wykonujemy na każdym wskaźniku na Entry skopiowanym z GPU, czyli na wskaźnikach na struktury Entry, które znajdują się w `hostTable.entries` i wskaźniku next każdej struktury Entry w puli elementów:

```
for (int i=0; i<table.count; i++) {
    if (hostTable.entries[i] != NULL)
        hostTable.entries[i] =
            (Entry*)((size_t)hostTable.entries[i] -
                     (size_t)table.pool + (size_t)hostTable.pool);
}
for (int i=0; i<ELEMENTS; i++) {
    if (hostTable.pool[i].next != NULL)
        hostTable.pool[i].next =
            (Entry*)((size_t)hostTable.pool[i].next -
                     (size_t)table.pool + (size_t)hostTable.pool);
}
```

my już konstrukcję samej struktury danych i funkcji skrótu, kod inicjujący i porządkujący funkcję weryfikującą poprawność struktury danych. Brakuje nam jeszcze tylko kodu, którym będziemy wykorzystywać operacje atomowe CUDA C. Funkcja `add_to_table()` jako argumenty będzie przyjmować tablicę kluczy i wartości, które mają zostać dodane do tablicy skrótów, samą tablicę skrótów oraz tablicę blokad do zablokowania każdego z kubelków struktury Entry. Ponieważ na wejściu mamy dwie tablice, które muszą być indeksowane przez wątki, musimy potrzebować też naszego starego dobrego kodu do obliczania indeksów liniowych:

```
_global_ void add_to_table( unsigned int *keys, void **values,
                           Table table, Lock *lock ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
```

Wątki będą przeglądać tablice wejściowe dokładnie w taki sam sposób jak w programie obliczającym iloczyn skalarny. Dla każdego klucza w tablicy `keys[]` wątek wywoła funkcję skrótu, aby się dowiedzieć, do którego kubelka przyporządkować daną parę klucz-wartość. Następnie ten sam wątek zablokuje wybrany kubelek, doda do niego parę klucz-wartość, a następnie odblokuje kubelek:

```
while (tid < ELEMENTS) {
    unsigned int key = keys[tid];
    size_t hashValue = hash( key, table.count );
    for (int i=0; i<32; i++) {
        if ((tid % 32) == i) {
            Entry *location = &(table.pool[tid]);
            location->key = key;
            location->value = values[tid];
            lock[hashValue].lock();
            location->next = table.entries[hashValue];
            table.entries[hashValue] = location;
            lock[hashValue].unlock();
        }
    }
    tid += stride;
}
```

W kodzie tym znajduje się bardzo szczególny fragment. Może się wydawać, że pętla `for()` i następująca po niej instrukcja `if()` są tu w ogóle niepotrzebne. Przypomnijmy jednak, że w rozdziale 6. poznaliśmy pojęcie **osnowy**, czyli zbioru 32 wątków wykonywanych synchronicznie. Pomijając szczegóły związane z implementacją tego bezpośrednio w konstrukcji procesora GPU, należy wiedzieć, że tylko jeden wątek z osnowy może założyć blokadę w danym czasie, dlatego jeśli wszystkim 32 wątkom pozwolimy równocześnie o nią rywalizować, to będziemy mieli spore kłopoty. W tej sytuacji stwierdziliśmy, że problem ten możemy rozwiązać programowo, czyli po prostu każdemu wątkowi z osnowy dać możliwość założenia blokady na strukturę danych, wykonać swoje zadanie i zwolnić blokadę.

Funkcja `main()` będzie działać tak samo jak w implementacji dla CPU. Najpierw alokuje dużą ilość losowych danych do użycia jako klucze. Następnie utworzy zdarzenia CUDA początku i końca do pomiaru wydajności programu i zarejestruje pierwsze z nich. Później nastąpi alokacja na GPU pamięci dla tablicy kluczy, skopiowanie tej tablicy na urządzenie oraz inicjacja tablicy skrótów:

```
int main( void ) {
    unsigned int *buffer =
        (unsigned int*)big_random_block( SIZE );
    cudaEvent_t start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

```

unsigned int *dev_keys;
void **dev_values;
HANDLE_ERROR( cudaMalloc( (void**)&dev_keys, SIZE ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_values, SIZE ) );
HANDLE_ERROR( cudaMemcpy( dev_keys, buffer, SIZE,
                        cudaMemcpyHostToDevice ) );
// Tu powinien być kod kopiujący wartości do dev_values.
// Należy go wstawić samodzielnie.
Table table;
initialize_table( table, HASH_ENTRIES, ELEMENTS );

```

Ostatni krok przygotowań do budowy tablicy skrótów to przygotowanie blokad dla kubeków. Alokujemy po jednej blokadzie dla każdego z nich. Gdybyśmy użyli tylko jednej blokady dla wszystkich, to byśmy oczywiście zaoszczędziли sporo pamięci, ale za to stracili na wydajności, ponieważ gdyby kilka wątków jednocześnie chciało dodać elementy w różnych miejscach tablicy, musiałyby one o tę blokadę rywalizować między sobą. Dlatego właśnie użyjemy całej tablicy blokad, po jednej dla każdego kubelka. Następnie alokujemy na GPU tablicę dla tych blokad i skopiuje je na urządzenie:

```

Lock lock[HASH_ENTRIES];
Lock *dev_lock;
HANDLE_ERROR( cudaMalloc( (void**)&dev_lock,
                        HASH_ENTRIES * sizeof( Lock ) ) );
HANDLE_ERROR( cudaMemcpy( dev_lock, lock,
                        HASH_ENTRIES * sizeof( Lock ),
                        cudaMemcpyHostToDevice ) );

```

Pozostały kod funkcji `main()` jest taki sam jak w wersji dla CPU: dodajemy wszystkie klucze do tablicy skrótów, zatrzymujemy stoper, weryfikujemy poprawność struktury i porządkujemy po sobie:

```

add_to_table<<<60,256>>>( dev_keys, dev_values,
                                table, dev_lock );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Czas wstawiania elementów: %3.1f ms\n", elapsedTime );
verify_table( table );
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
free_table( table );
cudaFree( dev_lock );
cudaFree( dev_keys );
cudaFree( dev_values );
free( buffer );
return 0;
}

```

A.2.5. WYDAJNOŚĆ TABLICY SKRÓTÓW

Budowa tablicy skrótów o rozmiarze 100 MB przy użyciu programu z podrozdziału A.2.2 zajęła na procesorze CPU Intel Core 2 Duo 360 ms. W czasie komplikacji włączona była opcja `-O3`, aby otrzymać maksymalnie zoptymalizowany kod maszynowy. Natomiast czas działania wielowątkowej wersji dla GPU z podrozdziału A.2.4 wyniósł 375 ms. Różnica wyniosła zatem około 5 procent, czyli wyniki można uznać za praktycznie jednakowe. Oczywiście nasuwa się pytanie: jak to możliwe, że posiadający tak duże możliwości w zakresie równoległego przetwarzania danych procesor GPU został pobity przez procesor CPU, który używa tylko jednego wątku? Przede wszystkim dlatego, że procesory graficzne nie zostały zaprojektowane do szybkiego przetwarzania złożonych struktur danych, jakimi są m.in. tablice skrótów. Dlatego też rzadko kiedy się zdarza, żeby można było odnieść jakieś korzyści z implementacji takich struktur danych dla GPU. Jeśli więc piszesz program, którego **głavnym** zadaniem jest budowa tablicy skrótów lub innej podobnej struktury danych, zapewne lepiej na tym wyjdiesz, jeśli użyjesz do tego celu procesora CPU.

Z drugiej strony może się zdarzyć, że będziesz pisać program, w którym operacje nieodpowiadające procesorowi GPU będą stanowić tylko jeden czy dwa etapy z o wiele dłuższego ciągu zadań. W takiej sytuacji masz trzy raczej oczywiste wyjścia:

- Wykonać wszystkie obliczenia na GPU
- Wykonać wszystkie obliczenia na CPU
- Wykonać niektóre obliczenia na CPU, a pozostałe na GPU

Najlepsza wydaje się ostatnia opcja, należy jednak pamiętać, że wówczas trzeba zadbać o synchronizację CPU z GPU w każdym miejscu w programie, w którym następuje przeniesienie obliczeń z jednego procesora na drugi. Koszty tej synchronizacji i opóźnienia spowodowane przenoszeniem danych mogą niestety zniwelować wszelkie korzyści uzyskane dzięki zastosowaniu hybrydowego rozwiązania.

W takim przypadku możliwe, że korzystne będzie wykonanie wszystkiego na GPU, nawet mimo to, że procesor ten nie najlepiej nadaje się do wykonywania niektórych zadań. W przypadku tablicy skrótów uniknąłbyś wówczas konieczności synchronizacji CPU z GPU oraz niepotrzebnego kopiowania danych między GPU a hostem, a także miałbyś procesor CPU wolny i gotowy do wykonywania innych działań. Możliwe że wówczas wersja programu działająca w całości na GPU byłaby wydajniejsza od wersji hybrydowej, mimo że niektóre kroki algorytmu byłyby wykonywane wolniej, niż to możliwe (czasami nawet mimo to, że CPU w niektórych zadaniach bije GPU na głowę).