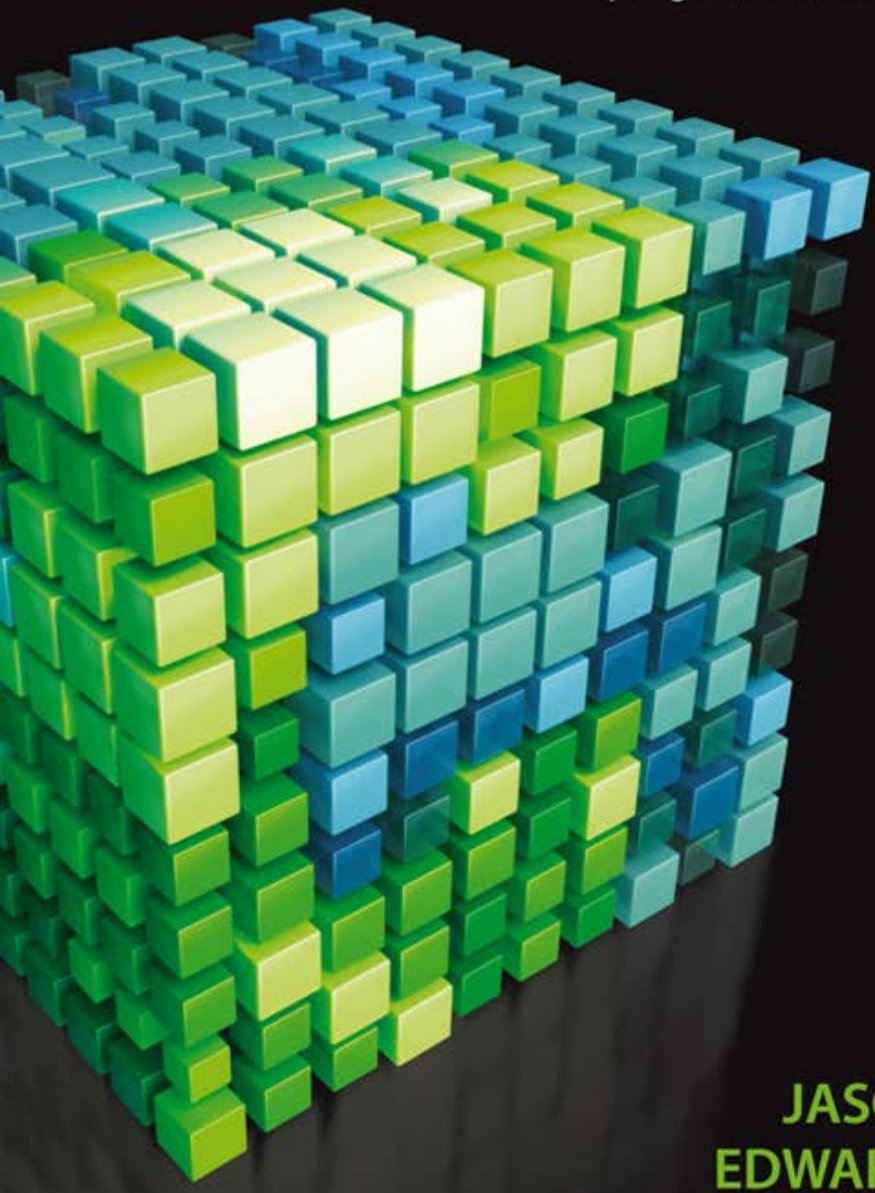
Helion

NVIDIA

CUDA
W PRZYKŁADACH

Wprowadzenie do ogólnego programowania procesorów GPU



JASON SANDERS
EDWARD KANDROT

Tytuł oryginału: CUDA by Example: An Introduction to General-Purpose GPU Programming

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-3817-8

Authorized translation from the English language edition, entitled: CUDA by Example: An Introduction to General-Purpose GPU Programming; ISBN 0131387685, by Jason Sanders and Edward Kandrot; published by Pearson Education, Inc, publishing as
Addison-Wesley Professional; Copyright © 2011 by NVIDIA Corporation.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education Inc.

Polish language edition published by Helion S.A.
Copyright © 2012.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniej¬szej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficz¬ną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cudawp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Słowo wstępne ... 9
Przedmowa ... 11
Podziękowania ... 13
O autorach ... 15

1 DLACZEGO CUDA? DLACZEGO TERAZ? 17

1.1. Streszczenie rozdziału ... 17
1.2. Era przetwarzania równoległego ... 17
1.2.1. Procesory CPU ... 18
1.3. Era procesorów GPU ... 19
1.3.1. Historia procesorów GPU ... 19
1.3.2. Początki programowania GPU ... 20
1.4. CUDA ... 21
1.4.1. Co to jest architektura CUDA ... 21
1.4.2. Używanie architektury CUDA ... 22
1.5. Zastosowania technologii CUDA ... 22
1.5.1. Obrazowanie medyczne ... 22
1.5.2. Symulacja dynamiki płynów ... 23
1.5.3. Ochrona środowiska ... 24
1.6. Podsumowanie ... 25

2 KONFIGURACJA KOMPUTERA 27

2.1. Streszczenie rozdziału ... 27
2.2. Środowisko programistyczne ... 27
2.2.1. Procesor graficzny z obsługą technologii CUDA ... 28
2.2.2. Sterownik urządzeń NVIDII ... 29
2.2.3. Narzędzia programistyczne CUDA ... 30
2.2.4. Standardowy kompilator języka C ... 31
2.3. Podsumowanie ... 32

Kup książkę Poleć książkę

3 PODSTAWY JĘZYKA CUDA C 33

3.1. Streszczenie rozdziału 33
3.2. Pierwszy program 33
3.2.1. Witaj, świecie! 34
3.2.2. Wywoływanie funkcji jądra 34
3.2.3. Przekazywanie parametrów 35
3.3. Sprawdzanie właściwości urządzeń 38
3.4. Korzystanie z wiedzy o właściwościach urządzeń 42
3.5. Podsumowanie 43

4 PROGRAMOWANIE RÓWNOLEGŁE W JĘZYKU CUDA C 45

4.1. Streszczenie rozdziału 45
4.2. Programowanie równoległe w technologii CUDA 45
4.2.1. Sumowanie wektorów 46
4.2.2. Zabawny przykład 52
4.3. Podsumowanie 60

5 WĄTKI 61

5.1. Streszczenie rozdziału 61
5.2. Dzielenie równoległych bloków 61
5.2.1. Sumowanie wektorów — nowe spojrzenie 62
5.2.2. Generowanie rozchodzących się fal za pomocą wątków 68
5.3. Pamięć wspólna i synchronizacja 72
5.3.1. Iloczyn skalarny 74
5.3.2. Optymalizacja (niepoprawna) programu obliczającego iloczyn skalarny 82
5.3.3. Generowanie mapy bitowej za pomocą pamięci wspólnej 84
5.4. Podsumowanie 87

6 PAMIĘĆ STAŁA I ZDARZENIA 89

6.1. Streszczenie rozdziału 89
6.2. Pamięć stała 89
6.2.1. Podstawy techniki śledzenia promieni 90
6.2.2. Śledzenie promieni na GPU 91
6.2.3. Śledzenie promieni za pomocą pamięci stałej 96
6.2.4. Wydajność programu a pamięć stała 97
6.3. Mierzenie wydajności programów za pomocą zdarzeń 99
6.3.1. Pomiar wydajności algorytmu śledzenia promieni 100
6.4. Podsumowanie 103

Kup książkę Poleć książkę

7 PAMIĘĆ TEKSTUR 105

7.1. Streszczenie rozdziału ... 105
7.2. Pamięć tekstur w zarysie ... 105
7.3. Symulacja procesu rozchodzenia się ciepła ... 106
7.3.1. Prosty model ogrzewania ... 106
7.3.2. Obliczanie zmian temperatury ... 108
7.3.3. Animacja symulacji ... 110
7.3.4. Użycie pamięci tekstur ... 114
7.3.5. Użycie dwuwymiarowej pamięci tekstur ... 117
7.4. Podsumowanie ... 121

8 WSPÓŁPRACA Z BIBLIOTEKAMI GRAFICZNYMI 123

8.1. Streszczenie rozdziału ... 124
8.2. Współpraca z bibliotekami graficznymi ... 124
8.3. Generowanie rozchodzących się fal za pomocą GPU i biblioteki graficznej ... 130
8.3.1. Struktura GPUAnimBitmap ... 130
8.3.2. Algorytm generujący fale na GPU ... 133
8.4. Symulacja rozchodzenia się ciepła za pomocą biblioteki graficznej ... 135
8.5. Współpraca z DirectX ... 139
8.6. Podsumowanie ... 139

9 OPERACJE ATOMOWE 141

9.1. Streszczenie rozdziału ... 141
9.2. Potencjał obliczeniowy ... 141
9.2.1. Potencjał obliczeniowy procesorów GPU NVIDII ... 142
9.2.2. Kompilacja dla minimalnego potencjału obliczeniowego ... 144
9.3. Operacje atomowe w zarysie ... 144
9.4. Obliczanie histogramów ... 146
9.4.1. Obliczanie histogramu za pomocą CPU ... 146
9.4.2. Obliczanie histogramu przy użyciu GPU ... 148
9.5. Podsumowanie ... 156

10 STRUMIENIE 157

10.1. Streszczenie rozdziału ... 157
10.2. Pamięć hosta z zablokowanym stronicowaniem ... 158
10.3. Strumienie CUDA ... 162
10.4. Używanie jednego strumienia CUDA ... 162
10.5. Użycie wielu strumieni CUDA ... 166
10.6. Planowanie pracy GPU ... 171
10.7. Efektywne wykorzystanie wielu strumieni CUDA jednocześnie ... 173
10.8. Podsumowanie ... 175

Kup książkę Poleć książkę

11 WYKONYWANIE KODU CUDA C JEDNOCZEŚNIE 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

12 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

A 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

Kup książkę Poleć książkę

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.

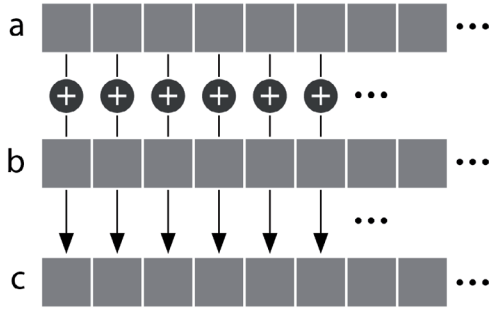
4.2. Programowanie równoległe w technologii CUDA

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 NVIDII 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.

Kup książkę Poleć książkę

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;
}
```


Kup książkę Poleć książkę

Większa część kodu tego programu nie wymaga objaśnień. Napiszemy tylko kilka słów o funkcji `add()`, aby wytłumaczyć 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 uwidocznić możliwość zrównoleglenia 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:

RDZEŃ 1

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

RDZEŃ 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. Należałoby utworzyć wątki robocze do wykonywania funkcji `add()` oraz przyjąć założenie, że wszystkie wątki będą działać równolegle, co niestety nie zawsze jest prawdą.

Kup książkę Poleć książkę

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) ) );
    // Zapełnienie 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 zauważyć 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()`.

Kup książkę Poleć książkę

- Za pomocą funkcji `cudaMemcpy()` z parametrem `cudaMamcpyHostToDevice` kopiujemy dane wejściowe na urządzenie, a następnie kopiujemy 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?

Kup książkę Poleć książkę

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 równoległe 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 kompilacji 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 zamieniania 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.

Liczbę 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:

Kup książkę Poleć książkę

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 pobrali zawartość pamięci, która do nas nie należy, to byśmy wpadli w tarapaty. Mogłoby to nawet spowodować zakończenie działania jądra, ponieważ GPU mają wbudowane wyrafinowane jednostki zarządzające pamięcią, które zamykają każdy proces, który by łamał zasady korzystania z pamięci.

Jeśli w programie wystąpi tego rodzaju błąd, jedno z makr `HANDLE_ERROR()`, którymi szczodrze sypiemy 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.

Kup książkę Poleć książkę

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 równoległe wykonywanie kodu w pełnej skali, zmień w 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 najciekawszych 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();
```


Kup książkę Poleć książkę

```
    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))`.

Kup książkę Poleć książkę

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 poleceń.

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 próg 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.

Kup książkę Poleć książkę

```
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 krotkę trójwymiarową, jakiej użyjemy do określenia liczby uruchomionych bloków. Ale dlaczego używamy trójwymiarowej wartości, skoro wcześniej bardzo wyraźnie podkreślaliśmy, że **utworzymy siatkę dwuwymiarową**?

Kup książkę Poleć książkę

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 zostaną podane 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 wywoływania jądra, ponieważ w sporach między API a programistą zawsze API jest górą.

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 * 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;
}
```

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 pary liczb całkowitych `(x,y)` z przedziału od `(0,0)` do `(DIM–1, DIM–1)` otrzymujemy jeden blok.

Kup książkę Poleć książkę

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*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 liczby zespolonej.

```
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);
    }
};
```


Kup książkę Poleć książkę

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__`.

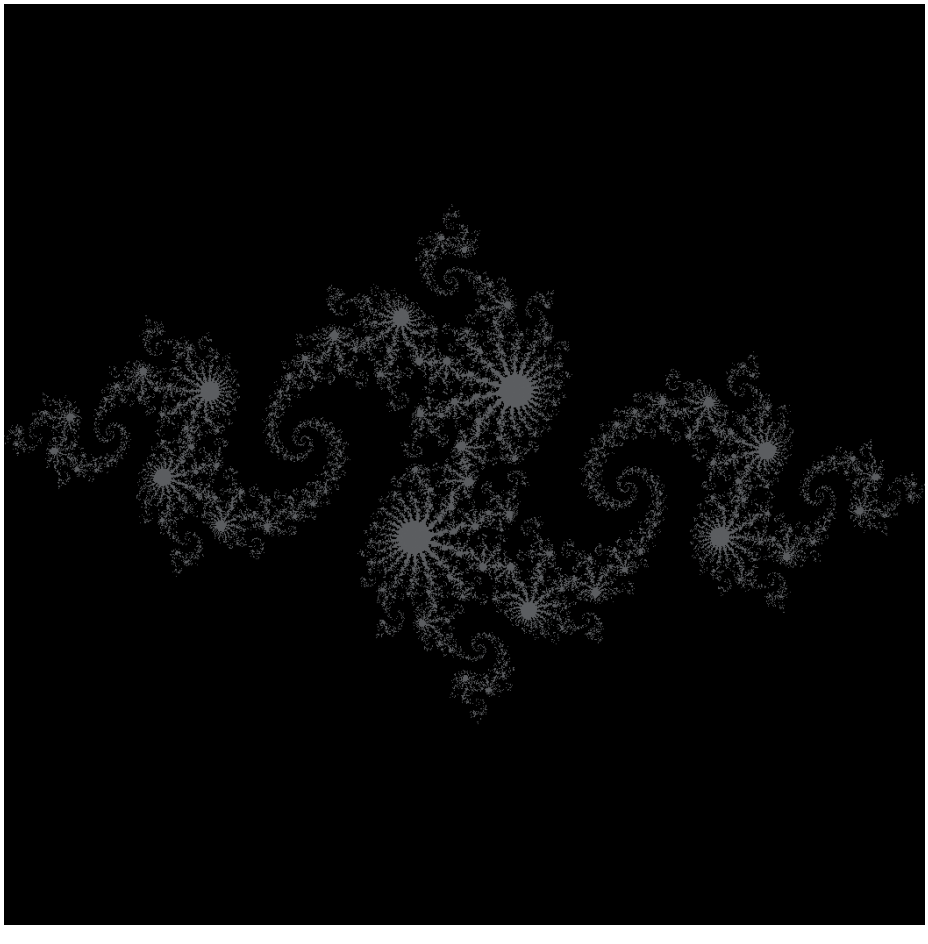
Poniżej znajduje się w całości kod źródłowy opisanego programu.

```
#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;
}
```


Kup książkę Poleć książkę

```
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

Kup książkę Poleć książkę

4.3. Podsumowanie

Gratulacje! Potrafisz już pisać, kompilować i uruchamiać programy równoległe na procesorze GPU. Koniecznie pochwal się znajomym. Jeśli nadal trwają oni w błędnym przekonaniu, 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.

Kup książkę Poleć książkę

Skorowidz

A

algorytm
  do obliczania iloczynu skalarnego, 74
  dodawania par klucz-wartość, 214
  generujący fale na gpu, 133
algorytmy dekompozycji macierzy LU i QR, 201
alokacja pamięci, 36, 75, 192
alokacja pamięci na GPU, 114, 158
  funkcja cudaMalloc(), 158
alokacja pamięci na hoście
  funkcja cudaHostAlloc(), 158
  funkcja malloc(), 158
alokacja pamięci zablokowanej
  jako przenośnej, 188
alokacja puli wolnych elementów, 218
alokacja tablic, 48
alokacja tablic kubełków, 218
animacja symulacji, 110
aplikacje 3D, 19
architektura CUDA, 21
architektura sprzętowa, 172
asynchroniczne kopiowanie, 165

B

Babbage Charles, 24
badanie dynamiki płynów, 24
badanie ultradźwiękowe, 23
bezpośredni dostęp do pamięci, DMA, 158
biblioteka
  BLAS, Basic Linear Algebra Subprograms, 195, 201
  CUBLAS, 195, 202
  CUDA Parallel Primitives Library, 201
  CUDPP, 201
  CUFFT, CUDA Fast Fourier Transform, 194
  CULATools, 201
  DirectX, 123
  GLUT, 123
  LAPACK, 201
  Linear Algebra Package, 201
  NPP, 196
  NVIDIA Performance Primitives, NPP, 196
  OpenGL, 19, 20, 123
biblioteki graficzne, 124
biblioteki osłonowe, 202
bieżący czas, ticks, 71
blok, 50
blokada pamięci, 158
blokady atomowe, 205, 207
blokady wzajemnego wykluczania, 205
budowa tablicy skrótów, 222
bufor danych bufferObj, 131
bufor danych resource, 131
bufor pikseli, 126
bufor podręczny, 97, 104
bufor teksturowy, 106
bufor z wyłączonym stronicowaniem, 158

C

CUDA Toolkit, 30
CUDA U, 199
czas działania programu, 171
czas działania programu poprawionego, 175
czas wykonywania dwóch wersji programu, 104
czasomierz procesora CPU, 99
czujnik natężenia światła, 90

Kup książkę Poleć książkę

D

debugowanie funkcji jądra, 197
deklaracja deskryptora, 119
deklarowanie bufora jako pamięci wspólnej, 96
dekrementacja, 150
deskryptor, 119
device overlap, 163
DirectX, 20
długość promienia, 91
DMA, direct memory access, 158
dostęp do pamięci z funkcji jądra, 178
Dr. Dobb's, 200
droga promienia od piksela do sceny, 90
dwa przeplatające się wątki, 145
dwa strumienie, 167
dyrektywa #define, 93
dystrybucje Linuksa, 31
dywergencja wątków, thread divergence, 83
działanie procesorów GPU NVIDII, 199
dzielenie bloków na wątki, 72
dzielenie równoległych bloków, 61
dzielenie wspólnej pamięci zablokowanej, 192

F

fala, 68
firma
  3dfx Interactive, 19
  ATI Technologies, 19
  NVIDIA, 19, 195
  Procter & Gamble, 24
  TechniScan Medical Systems, 23
fora NVIDII, 200
fraktal, 52
funkcja
  __synchthreads(), 77, 83
  __syncthreads(), 76
  add(), 47
  add_to_table(), 220
  anim_gpu(), 111, 116
  animExit(), 131
  asynchroniczna, 165
  atomicAdd(), 204, 205
  atomicCAS(), 206
  big_random_block(), 148
  blend_kernel(), 115
  ceil(), 65
  clickDrag(), 131
  copy_const_kernel(), 118
  cuda_malloc_test(), 160
  cudaBindTexture(), 114
  cudaBindTexture2D(), 119
  cudaChooseDevice(), 42, 125
  cudaEventCreate(), 102
  cudaEventDestroy(), 102
  cudaEventElapsedTime(), 102
  cudaEventSynchronize(), 100
  cudaFree(), 37, 43, 48, 69
  cudaFreeHost(), 181, 190
  cudaGetDeviceCount(), 38, 185
  cudaGetDeviceProperties(), 42, 182
  cudaGLSetDevice(), 131
  cudaGLSetGLDevice(), 125, 126
  cudaGraphicsGLRegisterBuffer(), 126, 132
  cudaGraphicsMapResources(), 139
  cudaGraphicsResourceGetMappedPointer(), 139
  cudaGraphicsUnmapResources(), 127
  cudaHostAlloc(), 158, 160, 180, 192
  cudaHostGetDevicePointer(), 181
  cudaMalloc(), 36, 43, 69
  cudaMemcpy(), 37, 43, 49, 112, 135, 219
  cudaMemcpyAsync(), 165
  cudaMemset(), 149, 218
  cudaSetDevice(), 42
  cudaSetDeviceFlags(), 182, 189
  cudaStreamSynchronize(), 166
  draw_func(), 128
  fAnim(), 131
  float_to_color(), 112, 137
  free(), 37, 43
  generate_frame(), 134
  glBufferData(), 126
  glDrawPixels(), 128
  glGenBuffers(), 126
  glutIdleFunc(), 132
  glutInit(), 132
  glutPostRedisplay(), 133
  grey, 72
  jądra, 49, 118
  jądra obliczająca histogram, 152, 154
  julia(), 53
  kernel(), 35
  lock(), 207
  main(), 35, 130, 151
  malloc(), 36, 43
  memcpy(), 37, 43
  memset(), 149

Kup książkę Poleć książkę

rand(), 164
routine(), 186
skrótu, hash function, 211
start_thread(), 186
synchronicznej, 165
tex1Dfetch(), 115
tex2D(), 117
transpose(), 198
verify_table(), 218
debugera CUDA-GDB, 197
glBindBuffer(), 126

G

GeForce 8800 GTX, 21
GeForce GTX 280, 103, 184
GeForce GTX 285, 138
GeForce GTX 295, 38, 177
generowanie fal, 68
generowanie fal za pomocą GPU i biblioteki graficznej, 130
generowanie mapy bitowej, 84
GLUT, GL Utility Toolkit, 125
gra
Doom, 19
Duke, 19
Nukem 3D, 19
Quake, 19

H

histogram, 146
czas tworzenia, 153
funkcja jądra, 155
obliczanie przy użyciu gpu, 148
obliczanie za pomocą cpu, 146
operacje atomowe dla pamięci globalnej, 152
operacje atomowe dla pamięci globalnej i wspólnej, 154
weryfikacja, 150
histogram cząstkowy, 155
histogram finalny, 155
HOOMD, 25
host, 34

I

identyfikator urządzenia CUDA, 125
iloczyn skalarny, 178, 203
blokady atomowe, 207
iloczyn skalarny, *Patrz także* obliczanie iloczynu skalarnego
implementacja
draw_func(), 127
funkcji kernel(), 56
kernel(), 127
key_func(), 127
tablicy skrótów, 210
indeks blockIdx, 109
indeks bloku, 50
indeks threadIdx, 109
indeks wątku, 67
inicjacja struktury GPUAnimBitmap, 131
inkrementacja, 145
inkrementacja dwóch wątków, 145
integracja językowa, 35
interfejs PCIE, 158

J

jądro działające na GPU szeregowo, 45
jądro wykonujące obliczenia równolegle, 45
jądro, kernel, 34
jednostki ALU, 21
jednostki arytmetyczno-logiczne, 21
język C, 25
język C++, 25
język CUDA, 156
język CUDA C, 17, 22
język do cieniowania, shading language, 21
język FORTRAN, 195
język GLSL, 22
język HLSL, 22

K

Kirk David, 199
klucz, 210
kod EXIT_FAILURE, 36
kod obliczający histogram na GPU, 152
kod źródłowy funkcji main(), 151
kolejka zadań, 100
kolejkowanie operacji wszerz, 173
kolejkowanie zadań dla GPU, 166
kolejność dodawania operacji do strumieni, 172
kolizje, 211
kolor figury, 94
kombinacja wątków i bloków, 64

Kup książkę Poleć książkę

kompilator, 30
  gcc, 31
  GNU C, 31
  kodu dla CPU, 31
  kodu dla GPU, 31
  Microsoft Visual Studio, 31
  nvcc, 144
komputer BlueGene/L, 25
konfiguracja środowiska programistycznego
  kompilator kodu dla CPU, 31
  kompilator kodu dla GPU, 31
  narzędzia programistyczne NVIDII, 30
  procesory GPU oparte na architekturze CUDA, 28
  sterownik urządzeń NVIDII, 29
krok redukcji, 77
kubełki, 211
kursy uniwersyteckie, 200
kursy z CUDA, 199
kwalifikator __device__, 58, 217
kwalifikator __global__, 35, 55
kwalifikator __shared__, 72

L

liczba bloków 65535, 52
liczba bloków funkcji kernel(), 55
liczba uruchamianych bloków, 151
liczba wątków, 64, 70
liczba zmiennoprzecinkowa pojedynczej precyzji, 21
liczby w nawiasach trójkątnych, 50
lokalność przestrzenna, 114

M

magistrala FSB, 158
magistrala PCI Express, 158
makro HANDLE_ERROR(), 36
maksymalna liczba bloków, 52
mammografia, 23
mapa bitowa, 84
matematyka dyskretna, 80
metoda anim_and_exit(), 69
metoda hit(), 92
model numeryczny, 23
model ogrzewania, 106
modelowanie komputerowe, 23
modelowanie kul, 91
modyfikator __constant__, 97
muteks, 205
muteksy CPU, 205

N

nagłówki bibliotek, 124
najmniejsza wielokrotność wartości, 79
narzędzia programistyczne, 194
  CUDA Toolkit, 194
  CUDA-GDB, 196
  GPU Computing SDK, 195
  NVIDIA Parallel Nsight, 197
  Visual Profiler, 198
narzędzia programistyczne NVIDII, 30
narzędzie profilujące, 198
nawiasy trójkątne, 45
norma IEEE, 21
NVIDIA CUDA Programming Guide, 42, 142
NVIDIA CUDA Reference Manual, 39

O

obiekty sferyczne, 91
obliczanie adresu wskaźnika, 220
obliczanie histogramów, 146
obliczanie iloczynu skalarnego
  algorytm, 203
  alokacja pamięci, 79
  doskonalenie algorytmu, 204
  funkcja cudaMemcpy(), 79
  generowanie danych wejściowych, 190
  kod źródłowy, 80
  nowa struktura danych, 184
  optymalizacja programu, 82
  użycie przenośnej pamięci zablokowanej, 189
  wykorzystanie kilku GPU, 185
  wywołanie funkcji jądra, 79
  za pomocą pamięci niekopiowanej, 178
obliczanie indeksu w jądrze, 64
obliczanie zmian temperatury, 108
obraz klatki piersiowej, 23
obsługa liczb zmiennoprzecinkowych, 205
obsługa strumieni przez sterownik CUDA, 171
odczyt-modyfikacja-zapis, 146
odczytywanie danych z tekstur, 114
odnośnik Download Drivers, 29
odnośnik get latest cuda toolkit production release, 30

Kup książkę Poleć książkę

ograniczenia wskaźników urządzenia, 37
określanie kroku inkrementacji, 67
opcja -O3, 223
operacja odczytu z pamięci stałej, 97
operacje atomowe, 144, 146, 156, 203
operacje atomowe na pamięci globalnej, 144, 154
operacje zapisu i odczytu danych, 86
osnowa, 98
ostatni etap redukcji, 78
oś czasu, 167, 173, 174
oświetlenie sceny, 91

P

pakiet GPU Computing SDK, 195, 201
pamięć
  DRAM, 106, 183
  globalna, 97, 103
  hosta, 164
  hosta niewymagająca kopiowania, 178
  niekopiowana, zero-copy memory, 178
  stała, 89, 97–98, 100
  stronicowana, 158
  pamięć tekstur dwuwymiarowa, 117
  pamięć tekstur, texture memory, 105, 114
  tylko do odczytu, 97
  wirtualna, 158
  wspólna, shared memory, 21, 72, 143
  z wyłączonym stronicowaniem, 178
  zablokowana, 178, 188
para klucz-wartość, 210
parametr
  cudaHostAllocPortable, 189
  cudaMemcpyDeviceToDevice, 38
  cudaMemcpyDeviceToHost, 112
  ticks, 111
platforma ION NVIDII, 183
plik book.h, 137
plik lock.h, 217
plik nagłówkowy gpu_anim.h, 133
pojedynczy układ NVIDII, 67
pole maxThreadsPerBlock, 64
pomiar czasu wykonywania operacji, 159
pomiar wydajności algorytmu, 100
porównywanie algorytmów transpozycji macierzy, 198
potencjał minimum 1.1, 144
potencjał obliczeniowy, 141
  minimalny, 144
  procesorów gpu nvidii, 142
potencjał obliczeniowy, compute capability, 142
powiązanie odwołań z buforem pamięci, 114
pozycja w buforze liniowym, 57
procesor
  CPU, 18
  GPU, 19, 20, 22, 142, 183
procesory GPU oparte na architekturze CUDA, 28
procesory graficzne *Patrz* procesor GPU
program
  Apple Developer Connection, ADC, 32
  bez synchronizacji, 86
  CUDA-GDB, 196
  do hipnotyzowania, 129
  NVIDIA Parallel Nsight, 197
  śledzący promienie, 92, 95
    pomiar wydajności, 100
  Visual Profiler, 198
  z synchronizacją, 87
programowanie GPU, 25
programowanie równoległe, 17, 25
projekt CUDA.NET, 202
projekt PyCUDA, 202
projektowanie wirników, 23
promienie wtórne, 91
propagacja danych, 104
propagacja danych na połówki osnów, 98
próba optymalizacji, 83
przekazanie sterowania do hosta, 78
przekazywanie parametrów, 35
przenośna pamięć zablokowana, 189
przeplatanie, 166
przeplatanie operacji, 167
przepustowość pamięci, 104
przesunięcie wywołania funkcji do bloku if(), 83
przewidywalność wyników, 146
pula, pool, 213

R

rasteryzacja, 90
redukcja, 76, 203
rejestracja zdarzenia, 99
relacje między wskaźnikami, 220
renderowanie obrazu, 131
resource, 131
rewolucja wielordzeniowa, 18
rozchodzenie się ciepła
  animacja symulacji, 110
  kod funkcji jądra, 109
  model symulacji, 106
  symulacja za pomocą biblioteki graficznej, 135

Kup książkę Poleć książkę

rozchodzenie się ciepła
tempo przepływu, 108
wykorzystanie pamięci tekstur, 114
wykorzystanie tekstur dwuwymiarowych, 117
zmiana temperatury, 108
rozmiar tablicy, 75
rozpropagowanie operacji odczytu na osnowę, 98
rozszerzenie języka C, 22

S

shader pikseli, 20
siatka, 50
słowo kluczowe __global__, 43
słowo kluczowe NULL, 216
sposoby wykorzystania blokad, 210
stały czas dostępu do elementów, 211
sterownik CUDA, 171, 172
sterownik urządzeń NVIDII, 29
struktura
bloków i wątków, 70
CPUAnimBitmap, 69, 130
cuComplex, 57
cudaDeviceProp, 38, 42, 125
DataBlock, 111
DataStruct, 186, 190
dla liczb zespolonych, 54
GPUAnimBitmap, 130
Lock, 207
Table, 213
strumienie, 99
strumienie CUDA, 157, 162, 166
jednoczesne wykorzystanie, 173
sumowanie wektorów, 46
na gpu za pomocą wątków, 62
o dowolnej długości, 66
przy użyciu procesora cpu, 46
za pomocą procesora gpu, 48
superkomputer, 18
Svara, 23
symulacja rozchodzenia się ciepła, 106, 135
symulacja wymiany ciepła, 113, 121
symulacje fizyczne, 106
synchronizacja, 72, 85, 86
synchronizacja CPU z GPU, 181
system obrazowania ultradźwiękowego, 23
system wieloprocesorowy, 43, 177, 184
szybkość kopiowania danych, 162

Ś

śledzenie promieni na gpu, 91
śledzenie promieni za pomocą pamięci stałej, 96
śledzenie promieni, ray tracing, 90
środki powierzchniowo czynne, 24
środowisko pracy, 27
środowisko programistyczne, 27

T

tablica
blokad, 222
buffer[], 147
c[], 77
cache[], 76
cptr[], 109
shared[][], 85
skrótów, 210, 212
skrótów dla CPU, 212
skrótów dla GPU, 217
skrótów wielowątkowa, 216
technika definiowania funkcji zwrotnych, 186
technologia CUDA, 22
technologia przetwarzania równoległego, 17
technologia SLI, scalable link interface, 184
Tesla C1060, 23
Tesla S1070, 177
test poprawności, 217
ticks, 71
trójwymiarowa siatka, 56
tworzenie zdarzeń do pomiaru czasu, 164, 168, 178
typ dim3, 55

U

uchwyt, 126
układy samodzielne, 183
układy zintegrowane, 183
urządzenia NVIDII, 103
urządzenie, 34
ustalanie rozmiaru pamięci, 96
usuwanie błędów, 196
użycie kilku procesorów GPU jednocześnie, 184

V

Visual Studio Memory, 197

Kup książkę Poleć książkę

W

wartość, 210
wartość threadsPerBlock, 79
wąskie gardło, 104
wątek, 46, 61
  generowanie fal, 68
  w osnowie, 103
wieloprocesor, 199
właściwości urządzeń, 38, 42
właściwości urządzeń CUDA, 40
właściwość integrated, 184
wskaźnik
  dla GPU, 181
  firstFree, 213
  na bufor, 135
  na funkcję generate_frame(), 69
  na histogram wyjściowy, 152
  na miejsce w pamięci, 206
  na miejsce w pamięci urządzenia, 71
  na pamięć hosta, 135, 165, 181
  na strukturę Entry, 213
  na tablicę danych wejściowych, 152
  na wskaźnik, 36
  typu void*, 131
wspólne bufory, 126
współbieżne wątki, 217
współczynnik scale, 54
współpraca z bibliotekami graficznymi, 124
  wspólne bufory, 126
współpraca z DirectX, 139
współrzędne na płaszczyźnie zespolonej, 53
współrzędne piksela, 53
wydajność, 98, 162
  pamięci niekopiowanej, 183
  programów, 97, 99, 198
  tablicy skrótów, 223
wykonywanie kodu CUDA C
  jednocześnie na wielu GPU, 177
wykonywanie kodu urządzenia, 69
wykonywanie programów, 171
wykorzystanie strumieni CUDA, 173
wyniki cząstkowe, 181
wywołania asynchroniczne, 100
wywołanie funkcji cudaSetDevice(), 191
wywołanie funkcji glDrawPixels(), 135
wywołanie funkcji kernel(), 34, 35
wywołanie jądra, 65, 67

Z

zablokowana pamięć przenośna, 192
zablokowane stronicowanie, 159
zależności wywołań funkcji od wywołań jądra, 172
zamiana wersji blokowej na wątkową, 63
zaokrąglanie wyników pośrednich, 204
zasoby kodu źródłowego, 201
zastosowania języka CUDA C, 22
  obrazowanie medyczne, 23
  ochrona środowiska, 24
  symulacja dynamiki płynów, 24
zawartość pamięci stałej, 104
zawieszenie procesora, 83
zbiór bloków i wątków, 65
zbiór Julii na cpu, 52
zbiór Julii na gpu, 54
zbiór wątków, 98
zdarzenia, 99
zdarzenia CUDA, 100
zdefiniowany bufor pamięci, 74
zegar CPU, 18
zerowanie bufora w pamięci wspólnej, 154
zintegrowane GPU, 42
zintegrowany układ graficzny, 43
zmienna
  blockDim, 64
  blockIdx.x, 50
  dragStartX, 131
  dragStartY, 131
  gridDim, 57, 60, 64
  mutex, 206
  offset, 118
  tid, 51
zmienne zapisane w pamięci wspólnej, 73
zmniejszanie kolejek do pamięci, 156
znacznik cudaGraphicsMapFlagsNone, 126
znacznik cudaGraphicsMapFlagsWriteDiscard, 126
zrównoleglanie na poziomie danych, 157
zrównoleglanie na poziomie zadań, 157
zwalnianie buforów, 166
zwalnianie pamięci, 218

Ż

żądanie alokacji, 126

Kup książkę Poleć książkę

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

CUDA
W PRZYKŁADACH

Wprowadzenie do ogólnego programowania procesorów GPU

Książka ta jest obowiązkową pozycją dla wszystkich programistów pracujących z systemami zawierającymi akceleratory.

Ze wstępu autorstwa
Jacka Dongarry'ego,
Uniwersytet Tennessee i Oak Ridge National Laboratory

Od astrofizyki i chemii, przez biologię obliczeniową, aż po analizę sejsmiczną i rekonstrukcję obrazu w tomografii komputerowej — architektura CUDA została entuzjastycznie przyjęta przez środowiska naukowe i akademickie. Znalazła też zastosowanie w wielu strategicznych gałęziach gospodarki i stała się niezwykłym ułatwieniem dla twórców programów równoległych, którym pozwoliła na wykorzystanie olbrzymiej mocy procesorów GPU do budowy ekstremalnie wydajnych aplikacji.

Oto podręcznik napisany przez członków zespołu tworzących architekturę CUDA. Stanowi on wyczerpujące wprowadzenie w świat programowania najnowszych akceleratorów o dużych możliwościach przetwarzania równoległego. Oparty na licznych przykładach, zilustrowany fragmentami przydatnego kodu przewodnik zawiera pełny opis tej platformy, wprowadzenie do języka CUDA C oraz szczegółowy opis wszystkich kluczowych technik pracy z tą niezwykłą architekturą.

OPANUJ JĘZYK CUDA C I PISZ PROGRAMY WYRÓŻNIAJĄCE SIĘ NIEZWYKŁĄ WYDAJNOŚCIĄ!

- Programowanie równoległe
- Współpraca wątków
- Pamięć stała i zdarzenia
- Pamięć teksturowa
- Interoperacyjność grafiki
- Operacje atomowe
- Strumienie
- CUDA C na wielu procesorach GPU
- Operacje atomowe dla zaawansowanych
- Dodatkowe zasoby CUDA

JASON SANDERS jest starszym programistą w zespole ds. platformy CUDA w firmie NVIDIA. Brał udział w pracach nad pierwszymi wersjami oprogramowania systemowego CUDA. Ma także ma swój wkład w specyfikację OpenCL 1.0. Zanim rozpoczął pracę w NVIDIA, pracował dla firm ATI Technologies, Apple oraz Novell.

EDWARD KANDROT jest starszym programistą w zespole ds. algorytmów CUDA w firmie NVIDIA. Przedtem pracował nad wydajnością programów takich firm, jak Adobe, Microsoft, Google czy Autodesk.

Addison-Wesley
Pearson Education

NVIDIA

helion.pl
księgarnia internetowa

Nr katalogowy: 7813

Księgarnia internetowa
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-3817-8

Cena: 59,00 zł.

sięgnij po WIĘCEJ
KOD KORZYŚCI

Informatyka w najlepszym wydaniu