Rozdział 17. CUDA w .NET

Tomasz Dziubak

W latach dziewięćdziesiątych wyłącznym zadaniem kart graficznych było wyświetlanie grafiki na monitorach komputerów. Jednak z końcem ubiegłego tysiąclecia rozpoczął się przewrót – ich moc obliczeniową zaczęto wykorzystywać nie tylko do przetwarzania graficznego, ale także do obliczeń ogólnych problemów numerycznych. Powstała idea GPGPU (z ang. *General-Purpose computing on Graphics Processor Units*), czyli pomysł wykonywania obliczeń nie związanych z grafiką na układach GPU. W krótkim czasie powstały dwie technologie realizujące ideę GPGPU: CUDA (ang. *Compute Unified Device Architecture*) stworzona przez firmę NVidia oraz *ATI Stream*[1] firmy ATI. Większą popularność zdobyła jednak ta pierwsza. Stało się tak dzięki znacznie prostszemu interfejsowi programowania.

Od czasu pojawienia się technologii CUDA na rynku minęło już sporo czasu. CUDA dość mocno się zmieniła, lecz dzięki pełnej kompatybilności wstecz programy napisane w tamtych czasach, można uruchomić również na obecnych modelach kart graficznych. CUDA wykorzystuje język *C for CUDA* oparty na języku C, ale wzbogacony o nowe słowa kluczowe i konstrukcje umożliwiające tworzenie aplikacji wykonujacych obliczenia z użyciem GPU. Należy zaznaczyć, że technologia CUDA wspierana jest tylko przez karty graficzne firmy NVidia[2].

Powstało wiele „wrapperów” umożliwiających wykorzystanie technologii CUDA w programach pisanych w innych językach niż C. Dzięki nim możliwe jest wykorzystanie CUDA w Pythonie[3], w środowisku MATLAB[4] czy pisząc kod w języku Java[5]. Nas interesuje jednak przede wszystkim język C#. W tym przypadku możemy skorzystać z biblioteki CUDA.NET, którą można pobrać ze strony *http://www.cass-hpc.com/solutions/libraries/cuda-net*. Niestety CUDA.NET nie jest już rozwijany.

Rozwiązanie, które chciałbym opisać w tym rozdziale, to jednak coś więcej niż tylko „wrapper”. *CUDAfy.NET*, bo o nim tu mowa, jest zbiorem bibliotek oraz nieoficjalnym rozszerzeniem języka C# umożliwiającym pisanie funkcji wykonywanych przez karty graficzne, tzw. kerneli[6], bezpośrednio w języku C#. Kernele piszemy zatem w języku C#, bezpośrednio w kodzie projektu .NET. Cały projekt kompilujemy używając standardowego kompilatora języka C#. Tłumaczeniem kerneli z C# na C for CUDA i kompilowaniem ich za pomocą kompilatora NVCC zajmują się biblioteki CUDAfy.NET[7].

Pragnę zaznaczyć, że w tym rozdziale Czytelnik nie znajdzie informacji na temat podstaw technologii CUDA. Rozdział nie jest także opisem pisania i optymalizacji kerneli. Zakładam, że czytelnik posiada już tę wiedzę i chciałby nauczyć się ją wykorzystywać w kontekście platformy .NET i języka C#. Osoby, które chcą dopiero

---

[1] Obecnie ta technologia znana jest pod nazwą *AMD Accelerated Parallel Processing*

[2] Lista kart dostępna jest pod adresem *https://developer.nvidia.com/cuda-gpus*.

[3] Zob. *http://mathema.tician.de/software/pycuda*

[4] Zob. *http://sourceforge.net/projects/gpumat*

[5] Zob. *http://www.jcuda.org*

[6] W polskiej literaturze używane jest czasem określenie „jądro”.

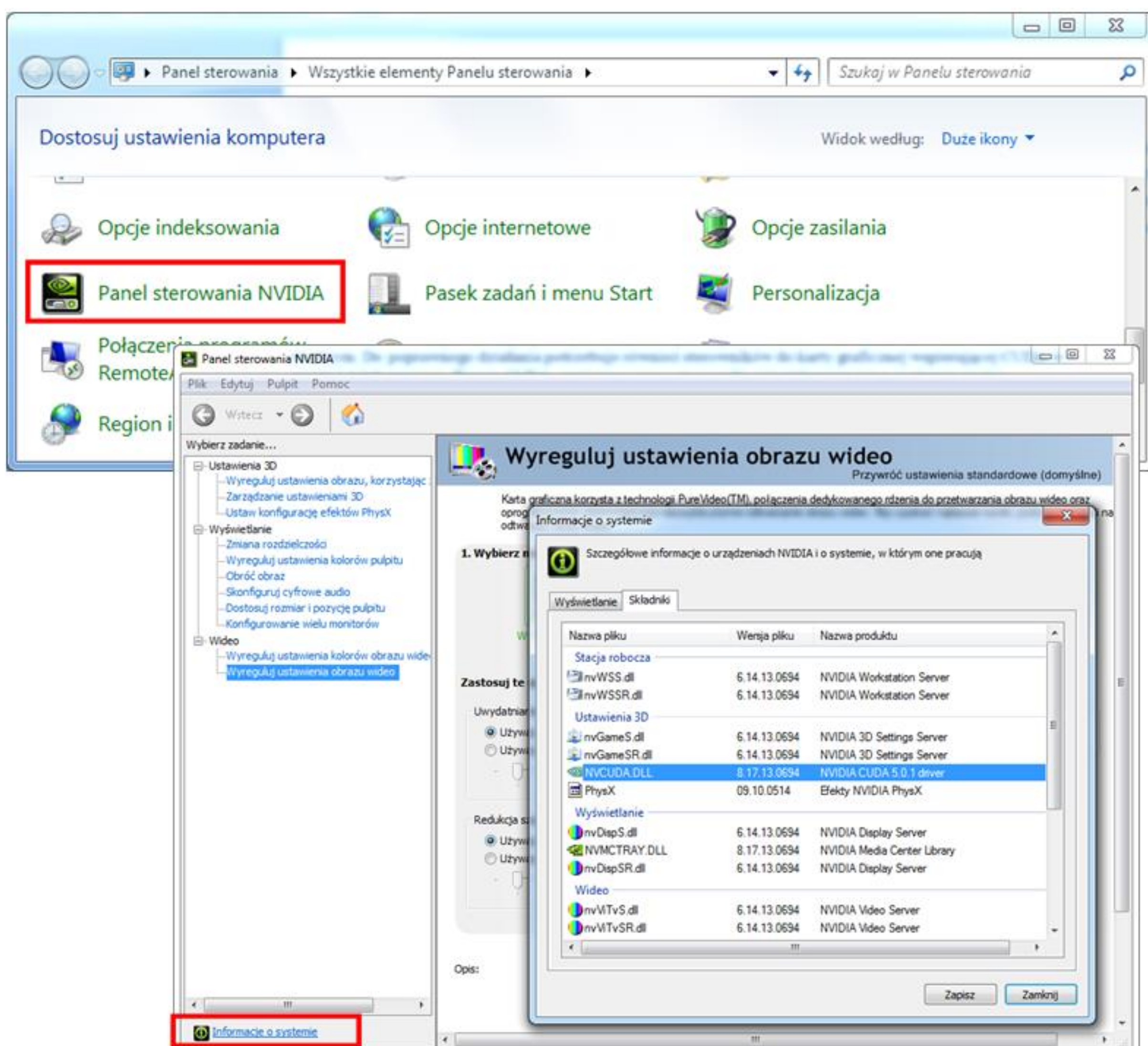
[7] Biblioteka CUDAfy.NET korzysta z translatora (piszę o tym niżej), który wpierw za pomocą dekompilatora ILSpy firmy SharpDevelop dekompiluje plik uruchomieniowy, a następnie tłumaczy go na C for CUDA.

zacząć naukę programowania z wykorzystaniem technologii CUDA odsyłam do książki pt. *CUDA w przykładach. Wprowadzenie do ogólnego programowania procesorów GPU* autorstwa J. Sandersa i E. Kandrot, która ukazała się w wydawnictwie Helion w 2012 roku.

Konfiguracja środowiska dla CUDAfy.NET

Zestaw bibliotek CUDAfy.NET został stworzony przez firmę Hybrid DSP. Można go pobrać ze strony internetowej *http://www.hybriddsp.com/Downloads.aspx*. CUDAfy.NET jest dostępny na licencji LGPL oraz w wersji, która może być wykorzystywana do celów komercyjnych. W momencie pisania tego rozdziału dostępna jest wersja stabilna 1.12.

CUDAfy.NET w wersji 1.12 współpracuje z systemem operacyjnym Windows XP (z Service Pack 3) lub wyższym. Do poprawnego działania potrzebuje również sterowników do karty graficznej wspierającej CUDA w wersji 5.0 lub wyższej. Wersję CUDA, wspieraną przez sterowniki zainstalowane już w systemie, można sprawdzić wybierając z panelu sterowania ikonkę *Panel Sterownia NVIDIA* a następnie klikając ikonę *Informacje o systemie* w lewym dolnym rogu okna *Panelu sterowania NVIDIA*.

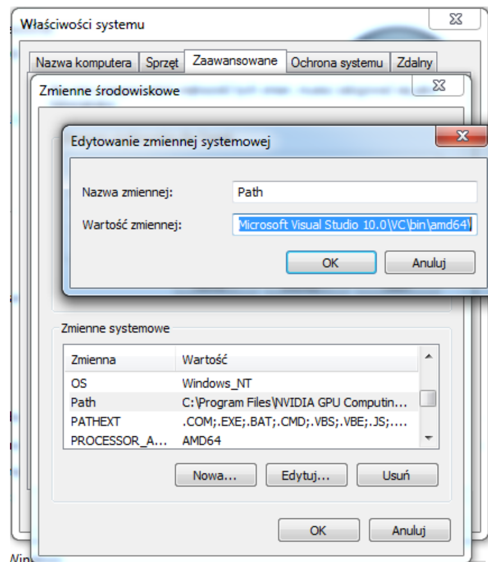


Rysunek 17.1. Informacje o wersji CUDA wspieranej przez sterowniki karty graficznej zainstalowane w systemie.

Jeśli obecne sterowniki nie wspierają wymaganej przez CUDAfy.NET wersji CUDA to wówczas należy je pobrać i zainstalować. Sterowniki dostępne są na stronach NVidii pod adresem *https://developer.nvidia.com/cuda-downloads*. Podczas instalacji możemy wybrać składniki jakie mają zostać

zainstalowane. Zalecam zainstalowanie całego pakietu zwłaszcza, że pozostałe składniki również są wymagane przez CUDAfy.NET.

Aby móc korzystać z CUDAfy.NET, w zmiennej środowiskowej PATH musi być obecna ścieżka do kompilatora *cl.exe* języka C/C++ z Visual Studio 2010. Jeśli takiej ścieżki nie ma, należy ją dodać. W przypadku mojego systemu jest to ścieżka *c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64*.



Rysunek 17.2. Ustawianie ścieżki dostępu do kompilatora cl.exe.

Pierwsze kroki

Po pobraniu i rozpakowaniu paczki z CUDAfy.NET widzimy, że składa się ona z kilku podkatalogów. Dwa z nich, a mianowicie *CudafyByExample* oraz *CudafyExamples*, zawierają przykłady użycia biblioteki. Pierwszy podkatalog zawiera skonwertowane do CUDAfy.NET przykłady ze wspomnianej już książki *CUDA w przykładach. Wprowadzenie do ogólnego programowania procesorów GPU*. Wszystkie projekty w tych katalogach są stworzone w Visual Studio 2010. W podkatalogu *CudafyExamples* znajdziemy natomiast przykłady użycia typów zespolonych czy wielowymiarowych tablic. W paczce najdziemy także katalog *bin*, który zawiera kilka plików. Są to miedzy innymi biblioteka *Cudafy.NET.dll* oraz translator *cudafycl.exe* języka C# do *C for CUDA*.

CUDAfy.NET korzysta również z „wrappera" *CUDA.NET* oraz dekompilatora *ILSpy*, dlatego też w pobranej paczce można znaleźć pliki potrzebne do działania tych dwóch bibliotek. O „wrapperze" *CUDA.NET* wspominałem już wcześnej (warto nadmienić, że pomimo tego, że ta bibliotek nie jest już rozwijana, to autorzy CUDAfy.NET modyfikują ją, lecz wyłącznie na potrzeby ich produktu) natomiast informacje na temat dekompilatora *ILSpy* można znaleźć na stronie *http://ilspy.net/*. Bardzo przydatnymi plikami z punktu widzenia programisty są także *CUDAfy API Documentation.url*, który jest skrótem do strony zawierającej opis API biblioteki *Cudafy.NET.dll* oraz podręcznik użytkownika *CUDAfy_User_Manual_1.12.pdf*.

Spróbujmy otworzyć w Visual Studio rozwiązanie dostarczone z *CUDAfy.NET* z podkatalogu *CudafyByExample*. W tym celu otwieramy plik *CudafyByExample.sln*, który znajduje się w tym podkatalogu, wciskamy klawisz *F6* i kompilujemy całe rozwiązanie. Jeśli wszystkie sterowniki mamy poprawnie zainstalowane, kompilacja powinna przebiec bez problemów. Spróbujmy więc uruchomić program wciskając klawisz *F5*. Niestety na moim systemie podczas próby uruchomienia programu zobaczyłem komunikat przedstawiony na rysunku 17.3. Problemem jest domyślny potencjał obliczeniowy (ang. *compute capability*) karty graficznej, który ustawiony jest w CUDAfy.NET na 1.3. Dla mojej karty graficznej powinien on być równy 1.1.

```
file:///c:/Users/Tomek/Desktop/CUDA w .NET -  książka/CudafyV1.12/CudafyByExample/bin/x64/R...
8

Chapter 3

hello_world
Hello, World!

simple kernel
Cudafy.Host.CudafyHostException: CUDA.NET exception: ErrorNoBinaryForGPU (Ensure
 that compiled architecture version is suitable for device).
   at Cudafy.Host.CudaGPU.HandleCUDAException(CUDAException ex)
   at Cudafy.Host.CudaGPU.LoadModule(CudafyModule module, Boolean unload)
   at CudafyByExample.simple_kernel.Execute() in c:\Users\Tomek\Desktop\CUDA w .
NET -  książka\CudafyV1.12\CudafyByExample\chapter03\simple_kernel.cs:line 24
   at CudafyByExample.Program.Main(String[] args) in c:\Users\Tomek\Desktop\CUDA
 w .NET -  książka\CudafyV1.12\CudafyByExample\Program.cs:line 33
```

Rysunek 17.3. Typowy problem, który może się pojawić podczas próby uruchomienia przykładowych programów spowodowany niewłaściwą wersją potencjału obliczeniowego karty graficznej, która w bibliotece CUDAfy.NET w wersji 1.12 jest domyślnie ustawiona na 1.3.

Problem można łatwo rozwiązać, ale wymaga to ingerencji w kody źródłowe przykładów. Spróbujmy więc zmodyfikować kod nie wnikając na razie głębiej w jego zawartość. Zlokalizujmy miejsce wystąpienia problemu. W tym celu otwieramy plik *Program.cs* i ustawiamy punkt przerwania programu (klawisz *F9*) tuż za nagłówkiem funkcji `Main`. Uruchamiamy program i wykonujemy go linia po linii wciskając klawisz *F10*. Linią stwarzającą problem okazuje się ta zawierająca wywołanie metody `simple_kernel.Execute`. Uruchamiamy więc program raz jeszcze i tym razem wchodzimy do tej metody wciskając *F11* (w momencie jej wykonywania). W ten oto sposób dotarliśmy do kodu źródłowego metody widocznej na listingu 17.1.. To ona powoduje pojawienie się komunikatu widocznego na rysunku 17.3.

Listing 17.1. Kod źródłowy metody simple_kernel.Execute, która powoduje problemy podczas uruchomienia programu jeśli karta graficzna ma potencjał obliczeniowy niższy niż 1.3

```
public static void Execute()
{
    CudafyModule km = CudafyTranslator.Cudafy();

    GPGPU gpu = CudafyHost.GetDevice(CudafyModes.Target);
    gpu.LoadModule(km);
    gpu.Launch().kernel();
    Console.WriteLine("Hello, World!");
}
```

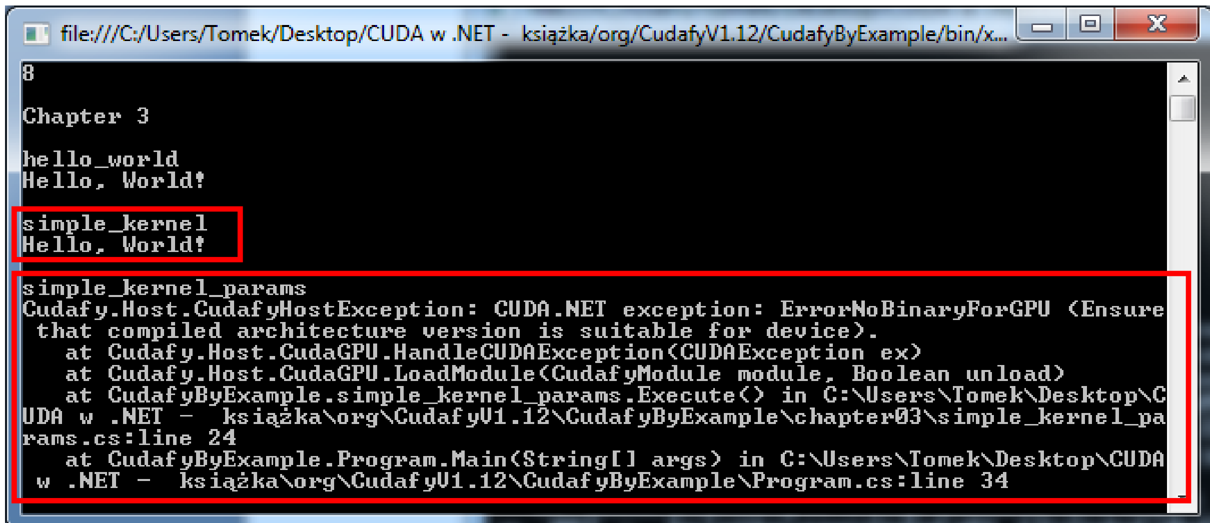
Przyjrzyjmy się pierwszej linii kodu umieszczonej wewnątrz tej metody. Tworzy ona obiekt `km` klasy `CudafyModule` za pomocą statycznej metody `Cudafy` klasy `CudafyTranslator`. Ta klasa jest wrapperem programu *ILSpy*. Umożliwia ona tłumaczenie kodu napisanego w .NET na *C for CUDA* oraz jego opakowanie w informacje wykorzystywane przez mechanizm refleksji. Wszystkie te dane przechowuje klasa `CudafyModule`.

Metoda `Cudafy` klasy `CudafyTranslator` jest metodą przeciążoną. Jedna z jej wersji przyjmuje tylko jeden argument typu wyliczeniowego `eArchitecture`. Umożliwia on ustawienie odpowiedniej wersji potencjału obliczeniowego karty graficznej. Najniższą wersją wspieraną przez CUDAfy.NET jest 1.1. Ustawmy więc taką wartość i spróbujmy skompilować rozwiązanie oraz uruchomić program.

Listing 17.2. Ustawienie wersji potencjału obliczeniowego karty graficznej na 1.1 w przykładowym programie

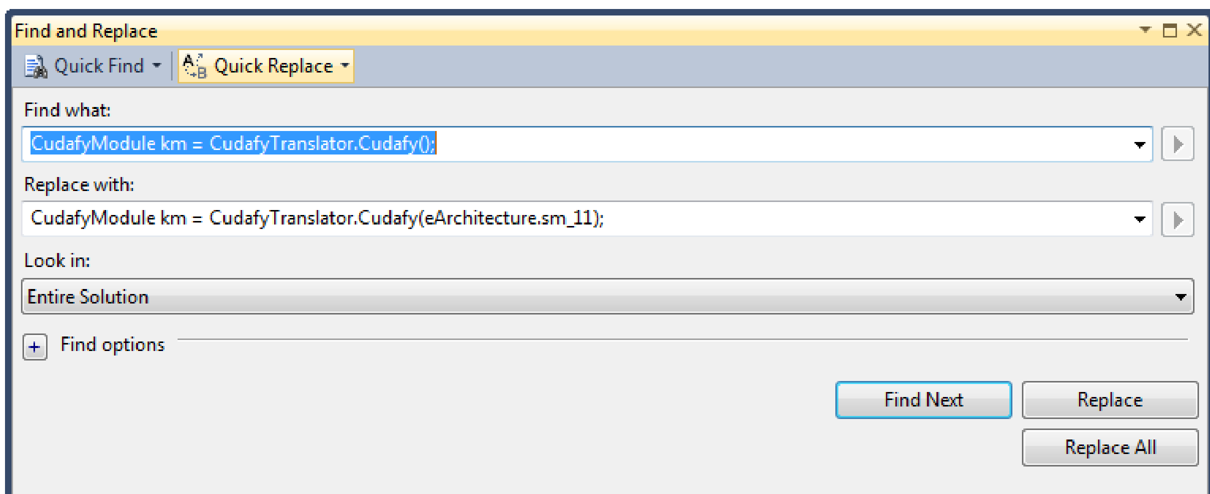
```
public static void Execute()
{
```

```
            CudafyModule km = CudafyTranslator.Cudafy(eArchitecture.sm_11);

            GPGPU gpu = CudafyHost.GetDevice(CudafyModes.Target);
            gpu.LoadModule(km);
            gpu.Launch().kernel(); // or gpu.Launch(1, 1, "kernel");
            Console.WriteLine("Hello, World!");
        }
```



Rysunek 17.4. Prawidłowe wykonanie pierwszego programu przykładowego używającego karty graficznej. Pojawiły się kolejne błędy przy próbie uruchomienia pozostałych programów przykładowych.

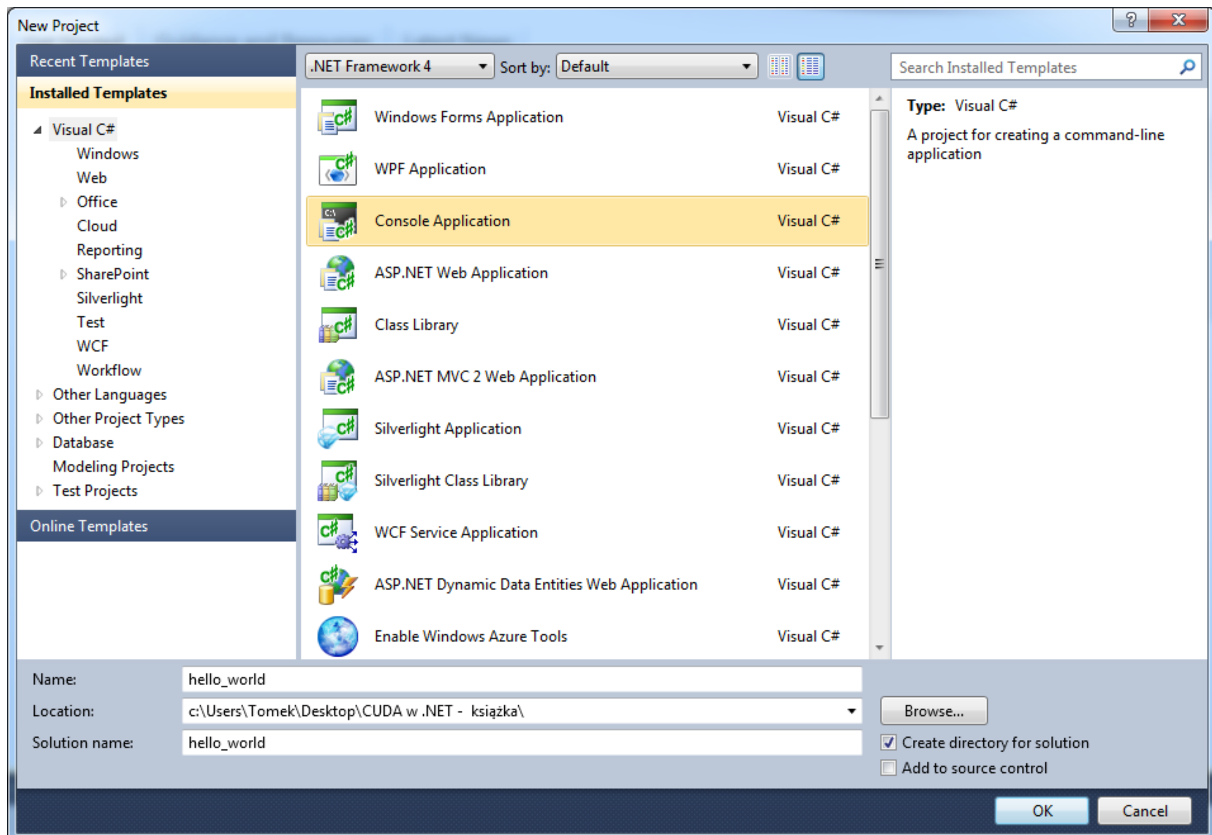
Tym razem pierwszy program przykładowy wykorzystujący kartę graficzną wykonał się prawidłowo. Pojawiły się natomiast kolejne błędy przy próbie uruchomienia kolejnego programu przykładowego. Aby wyeliminować wszystkie problemy tego typu, należy w każdym dostarczonym przykładzie ustawić odpowiednią wersję potencjału obliczeniowego karty graficznej. Można to wykonać bardzo szybko używając narzędzia wielokrotnej zamiany, tak jak to pokazano na rysunku 17.5. Po wprowadzeniu zmian można już bez problemów uruchomić pozostałe przykłady.



Rysunek 17.5. Metoda na szybką zmianę domyślnej wersji potencjału obliczeniowego w przykładowych programach. Narzędzie można wywołać wciskając kombinację klawiszy Ctrl+H. W pole „Find what" wpisujemy CudafyModule km = CudafyTranslator.Cudafy(); a w pole „Replace with" CudafyModule km = CudafyTranslator.Cudafy();. Po wypełnieniu odpowiednich pól klikamy na „Replace All".

Hello World, czyli pierwszy program CUDAfy.NET

W większości kursów programowania pierwszym programem jest odmiana „Hello World” – program drukujący na ekranie (konsoli) napis „Hello World!”. My również spróbujemy napisać taki program. Zacznijmy od uruchomienia Visual Studio i stworzenia projektu aplikacji konsolowej o nazwie *hello_world*.



Rysunek 17.6. Tworzenie pierwszego projektu w języku C# o nazwie hello_world wykorzystującego moc obliczeniową karty graficznej.

Tworzony program będzie wykorzystywał funkcję umożliwiającą bezpośrednie wyświetlanie napisów na monitorze komputera przez karty graficzne (bez konieczności jawnego pobierania danych z karty graficznej w celu ich wyświetlenia przez hosta). Brzmi to może dość dziwnie – przecież karty graficzne właśnie od tego są, aby wyświetlać grafikę. Mi chodzi jednak o funkcję podobną do `printf` z języka C. Taką możliwość dostarczają niestety dopiero karty graficzne wspierające wersję potencjału obliczeniowego 2.0 i wyższą. Użytkownicy posiadający starsze modele kart nie będą mogli uruchomić programu. Dla nich zmodyfikujemy nieco program w taki sposób, aby i oni mogli zobaczyć swoje pierwsze „Hello World”.

Po utworzeniu projektu pierwszą czynnością, jaką musimy wykonać jest dodanie do niego referencji do biblioteki *Cudafy.NET.dll*. Jak już wcześniej pisałem, bibliotekę tą znajdziemy w pobranej paczce, w katalogu *CudafyV1.12\bin*. Następnym krokiem jest dodanie do sekcji instrukcji *using* trzech przestrzeni nazw, których dostarcza ta biblioteka. W tym momencie kod źródłowy powinien wyglądać tak, jak na listingu 17.3.

Listing 17.3. Przestrzenie nazw dostarczone przez bibliotekę Cudafy.NET.dll.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Cudafy;
using Cudafy.Host;
using Cudafy.Translator;
```

```
namespace hello_world
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

W metodzie Main stwórzmy instancję klasy CudafyModule o nazwie modułCuda za pomocą statycznej metody Cudafy (por. listing 17.2). Ustawiamy wersję potencjału obliczeniowego karty graficznej na 2.0. Ta klasa oprócz własności, które już wcześniej opisałem, posiada także szereg innych udogodnień. Posiada metodę Compile, która kompiluje kod źródłowy przeznaczony do wykonania na GPU za pomocą kompilatora NVCC. Moduł skompilowany do formatu PTX[8] również jest przechowywany przez instancję klasy CudafyModule. Klasa zawiera także mechanizmy, które dbają o ta aby kod źródłowy wykonywany na GPU był kompilowany tylko wtedy, kiedy jest to wymagane, czyli tylko jeżeli został zmieniony. W przeciwnym przypadku używany jest wcześniej skompilowany moduł PTX. Takie podejście znacznie przyśpiesza uruchamianie programu. Wszystkie te dane są przechowywane w pliku XML o domyślnym rozszerzeniu *cdfy* tworzonym podczas wywołania metody CudafyTranslator.Cudafy.

Kolejnym krokiem jest pozyskanie uchwytu reprezentującego kartę graficzną, na której chcemy wykonywać obliczenia. Odbywa się to za pomocą statycznej metody GetDevice pochodzącej z klasy CudafyHost. Uchwyt jest reprezentowany przez instancje klasy GPGPU. Następnie musimy załadować modułCuda.. Służy do tego metoda LoadModule klasy GPGPU. Jej argumentem jest instancja klasy CudafyModule. Listing 17.4 pokazuje jak wygląda funkcja Main na tym etapie tworzenia aplikacji.

Listing 17.4. Utworzenie instancji klasy CudafyModule, pobranie uchwytu reprezentującego kartę graficzną oraz załadowanie modułu modułCuda.

```
static void Main(string[] args)
{
    CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_20);
    GPGPU uchwytGPU = CudafyHost.GetDevice();

    uchwytGPU.LoadModule(modułCuda);
}
```

W tym momencie powinniśmy przejść do implementacji funkcji wykonywanej przez kartę graficzną. Funkcję taką nazywa się kernelem. Implementowany przez nas kernel wyświetli na monitorze komputera tekst *Witaj swiecie!*. Kod kernela przedstawia listing 17.5.

Listing 17.5. Kernel wyświetlający na monitorze komputera tekst „Witaj świecie!” .

```
[Cudafy]
public static void witajSwiecie()
{
    Console.WriteLine("Witaj swiecie!");
}
```

Jak widzimy funkcja, która ma być wykonywana przez GPU posiada atrybut Cudafy. W programie może być oczywiście zdefiniowanych więcej kerneli. Wówczas wszystkie muszą posiadać takich atrybut. Dzięki niemu klasa CudafyTranslator wie jakie funkcje należy skonwertować do języka *C for CUDA*. Ponadto kernele muszą być metodami statycznymi. Jeżeli funkcja nie zwraca żadnej wartości, tak jak jest to w naszym przypadku, wówczas jest ona traktowana jako funkcja wywoływana przez hosta i wykonującą się na GPU (funkcja z kwalifikatorem __global__). Jeśli funkcja zwracałaby wartość, wówczas traktowana byłaby jako

[8] Zob. *http://docs.nvidia.com/cuda/parallel-thread-execution/index.html*

funkcja wywoływana przez GPU i wykonującą się na GPU (kwalifikator `__device__`). W kernelu użyłem dobrze znanej programistom C# metody `Console.WriteLine`. Wyświetla ona ciąg znaków na monitorze komputera. W przypadku kernela wykonuje ona taką samą czynność. Różnica polega tylko na tym, że tekst jest wyświetlany przez wątek GPU, który „wykonuje” tę metodę. A dokładnie nie tą metodę, a efekt jej tłumaczenia na język *C for CUDA*. Podczas tłumaczenia (konwersji) instrukcja `Console.WriteLine` jest zamieniana na funkcję `printf`. Wszystkie skonwertowane kernele są zapisywane do pliku o rozszerzeniu *cu*, który znajduje się w katalogu uruchomieniowym. Listing 17.6 przedstawia zawartość tego pliku w przypadku metody z listingu 17.5.

Listing 17.6. Kernel z listing 17.5 skonwertowany do języka „C for CUDA”. Poniższy kod znajduje się w pliku CUDAFYSOURCETEMP.cu.

```
#include <stdio.h>


// hello_world.Program
extern "C" __global__ void witajSwiecie();


// hello_world.Program
extern "C" __global__ void witajSwiecie()
{
    printf("Witaj swiecie!\n");
}
```

Prosty kernel jest już gotowy. Wywołajmy go zatem z metody `Main` w taki sposób, aby został wykonany przez kartę graficzną. Niezbędne modyfikacje przedstawia listing 17.7. Jak widać dodane zostały trzy linie kodu z czego ważne są dwie pierwsze. Pierwsza z nich tworzy dynamiczny obiekt `startKernel`. Obiekt ten zawiera metody, które są efektem wykorzystania w CUDAfy.NET mechanizmu DLR (od ang. *Dynamic Language Runtime*[9]) dodanego do platformy.NET w wersji 4.0. W drugiej linii wywołujemy wobec tego przygotowany wcześniej kernel `witajSwiecie`.

Listing 17.7. Wywołanie kernela witajSwiecie w głównej funkcji programu z wykorzystaniem mechanizmu DLR. Metoda Console.ReadKey zatrzymuje program umożliwiając obejrzenie wyników działania kernela.

```
static void Main(string[] args)
{
    CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_20);
    GPGPU uchwytGPU = CudafyHost.GetDevice();

    uchwytGPU.LoadModule(modułCuda);

    dynamic startKernel = uchwytGPU.Launch();
    startKernel.witajSwiecie();

    Console.ReadKey();
}
```

Ponieważ `startKernel` jest typu `dynamic`, kompilator nie sprawdza czy kernel `witajSwiecie` jest zdefiniowany w tworzonym programie. W architekturze DLR znaczenie wywoływanej metody (w naszym przypadku kernela) jest analizowane dopiero podczas wykonania programu, a sam kod skompiluje się dla dowolnej nazwy kernela. Jeśli ktoś nie lubi korzystania z architektury DLR, może użyć klasycznej metody tak, jak to pokazano na listingu 17.8.

---

[9] O typach dynamicznych i programowaniu z wykorzystaniem możliwości dynamicznych języków programowania można przeczytać miedzy innymi na stronach *http://msdn.microsoft.com/pl-pl/library/csharp-4-0--typy-dynamiczne.aspx* oraz *http://msdn.microsoft.com/pl-pl/library/programowanie-z-wykorzystaniem-mozliwosci-dynamicznych-jezykow-programowania.aspx*

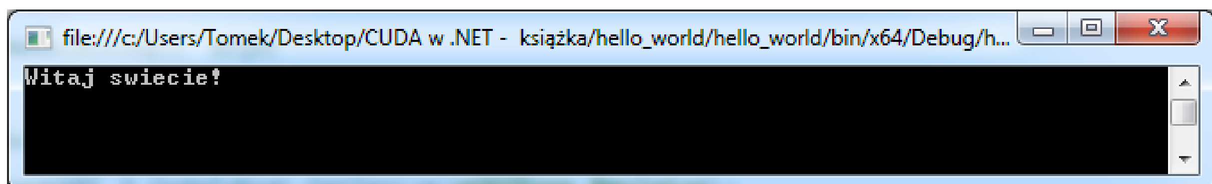
Listing 17.8. Alternatywny sposób wywołania kernela witajSwiecie w głównej funkcji programu bez wykorzystywania mechanizmu DLR.

```
static void Main(string[] args)
{
    CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_20);
    GPGPU uchwytGPU = CudafyHost.GetDevice();

    uchwytGPU.LoadModule(modułCuda);

    uchwytGPU.Launch(1, 1, "witajSwiecie");

    Console.ReadKey();
}
```

Oba sposoby wywołania kernela `witajSwiecie` zaprezentowane na listingach 17.7 i 17.8 są równoważne. Obydwie metody uruchamiają kernel na sieci o rozmiarze 1 i ilości wątków w bloku również równym 1. Rozmiar sieci oraz bloku (ilość wątków w bloku) określa się podczas wywoływania metody `Launch`.

Dostępny jest także trzeci sposób uruchamiania kerneli. Oprócz dwóch poznanych (standardowej oraz z wykorzystaniem DLR), możliwe jest uruchamianie z mocnym typowaniem.

Na tym skończyliśmy implementację naszego pierwszego programu wykonywanego na GPU. Rezultat jego działania jest zaprezentowany na rysunku 17.7. Efekty nie są być może zbyt spektakularne, nie mniej jednak jest to w pełni działający program wykorzystujący moc obliczeniową karty graficznej. Niestety efektami jego działania mogą się cieszyć tylko posiadacze karty graficznej, która wspiera potencjał obliczeniowy 2.0 i wyższy. W kolejnym podrozdziale zaprezentuję narzędzie, które pozwoli na podziwianie efektów swojej pracy również posiadaczom starszych kart graficznych. Zachęcam jednak wszystkich czytelników do zapoznania się z informacjami zawartymi w tym podrozdziale ponieważ będą one pomocne każdemu programiście, który zamierza wykorzystywać potencjał kart graficznych w swojej pracy.



Rysunek 17.7. Program hello_world w działaniu.

Emulator GPU

Co zrobić jeśli nie posiadamy karty graficznej wspierającej technologię CUDA z potencjałem obliczeniowym większym lub równym 2.0? W takim przypadku do testów programu możemy użyć emulatora GPU dostarczonego przez firmę NVidia i wspieranego przez CUDAfy.NET. W tym celu musimy zmodyfikować nieco program. Wróćmy do metody `CudafyHost.GetDevice`. Ta metoda może przyjąć dwa opcjonalne parametry. Pierwszy z tych parametrów jest typem wyliczeniowym `eGPUType`, który domyślnie przyjmuje wartość `eGPUType.Cuda`. Wartość ta oznacza, że nasz program ma się wykonywać na katcie graficznej. Wartość tego parametru możemy jednak zmienić na `eGPUType.Emulator` (listing 17.9) Wówczas do uruchomienia programu wykorzystywany jest emulator GPU.

Listing 17.9. Modyfikacja umożliwiająca uruchomienie program na kartach graficznych posiadających potencjał obliczeniowy niższy niż 2.0.

```
static void Main(string[] args)
{
```

```
        CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_20);
        GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Emulator);

        uchwytGPU.LoadModule(modułCuda);

        dynamic startKernel = uchwytGPU.Launch(1, 1);
        startKernel.witajSwiecie();

        Console.ReadKey();
    }
```

Spróbujmy teraz uruchomić program wciskając klawisz *F5*. Program powinien skompilować się i po chwili uruchomić. Tym razem w wykonanie programu nie jest zaangażowany procesor GPU, lecz jedynie CPU emulując GPU. Dzięki temu możemy program uruchomić również na komputerach z kartami graficznymi nie wspierającymi technologii CUDA lub o zbyt niskim potencjale obliczeniowym. Efekt jego działania jest taki sam jak na rysunku 17.7.

Warto przyjrzeć się również drugiemu parametrowi metody `CudafyHost.GetDevice`. Nie ma on nic wspólnego z emulacją karty graficznej. Jest natomiast przydatny, gdy komputer wyposażony jest w kilka kart graficznych. Umożliwia on bowiem określenie na której karcie mają wykonywać się nasze obliczenia. Domyślna wartość tego parametru to 0, co oznacza kartę graficzną o numerze identyfikacyjnym równym 0.

Własności GPU

Chcąc pisać programy wykorzystujące moc obliczeniową procesora graficznego należy poznać podstawowe parametry karty graficznej. Biblioteka CUDAfy.Net umożliwia pobranie w bardzo prosty sposób wszystkich niezbędnych parametrów. Klasa przechowująca te informacje to `GPGPUProperties`. Do pobierania tych informacji dla wszystkich kart graficznych znajdujących się w systemie przeznaczona jest statyczna metoda `GetDeviceProperties` z klasy `CudafyHost`. Ta metoda zwraca wartość typu `IEnumerable<GPGPUProperties>` co oznacza, że możemy użyć pętli `foreach` w celu wylistowania informacji dla wszystkich kart graficznych. Metoda `GetDeviceProperties` posiada dwa argumenty wywołania. Pierwszy z nich jest typem wyliczeniowym `eGPUType`, który poznaliśmy już wcześniej. Jego wartość ustawiamy na `eGPUType.Cuda` jeśli chcemy poznać własności karty graficznej. Nic jednak nie stoi na przeszkodzie aby jako pierwszy parametr wywołania ustawić `eGPUType.Emulator` co umożliwi poznanie podstawowych parametrów emulatora GPU. Drugi z argumentów przyjmuje wartość typu `bool`. Umożliwia on pobranie dodatkowych informacji o GPU za pomocą biblioteki *cudart.dll*. Domyślna wartość tego parametru jest ustawiona na `true`.

Zmodyfikujmy wcześniej opisany pierwszy program w taki sposób, aby oprócz zwykłego *Witaj świecie!* wyświetlał także informacje o kartach graficznych. Listing 17.20 przedstawia wymagane do tego modyfikacje metody `Main`. Obecnie coraz częściej pojawiają się systemy z wieloma kartami graficznymi. Również karty graficzne z wieloma procesorami nie są niczym nowym. Znajomość podstawowych parametrów tych układów pozwala na uruchomienie programu na układzie, który najlepiej spełnia jego wymogi. Dzięki temu wydajność programu może znacznie wzrosnąć. Wśród udostępnianych parametrów można znaleźć informacje o wspieranym potencjale obliczeniowym, pamięci karty graficznej czy dostępnej liczbie wątków. Wynik działania programu przedstawiony jest na rysunku 17.8.

Listing 17.20. Modyfikacja programu hello_world, która wyświetla podstawowe informacje o kartach graficznych zainstalowanych w komputerze.

```
    static void Main(string[] args)
    {
        CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_20);
        GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Emulator);

        uchwytGPU.LoadModule(modułCuda);
```

```
    dynamic startKernel = uchwytGPU.Launch(1, 1);
    startKernel.witajSwiecie();

    //wykonanie kernela standardową metodą
    //uchwytGPU.Launch(1, 1, "witajSwiecie");

    Console.WriteLine();
    Console.WriteLine("Informacje o kartach graficznych zainstalowanych w systemie");
    Console.WriteLine();
    foreach (GPGPUProperties parametryGPU in
CudafyHost.GetDeviceProperties(eGPUType.Cuda))
    {
        Console.WriteLine("Nazwa urządzenia: " + parametryGPU.Name);
        Console.WriteLine("Numer identyfikacyjny urządzenia: " + parametryGPU.DeviceId);
        Console.WriteLine("Potencjał obliczeniowy: " + parametryGPU.Capability);
        Console.WriteLine("Częstotliwość taktowania zegara: " + parametryGPU.ClockRate);
        Console.WriteLine("Status ECC: " + (parametryGPU.ECCEnabled ? "włączone" :
"wyłączone"));
        Console.WriteLine("Limit czasu działania kernela: " +
(parametryGPU.KernelExecTimeoutEnabled ? "włączone" : "wyłączone"));
        Console.WriteLine("Liczba multiprocesorów: " +
parametryGPU.MultiProcessorCount);
        Console.WriteLine("Ilość pamięci globalnej: " + parametryGPU.TotalGlobalMem +
"B");
        Console.WriteLine("Ilość pamięci stałej: " + parametryGPU.TotalConstantMemory +
"B");
        Console.WriteLine("Ilość pamięci współdzielonej przypadającej na jeden blok: " +
parametryGPU.SharedMemoryPerBlock + "B");
        Console.WriteLine("Maksymalna liczba wątków: " + parametryGPU.MaxThreadsSize.x +
"x" + parametryGPU.MaxThreadsSize.y + "x" + parametryGPU.MaxThreadsSize.z);
        Console.WriteLine("Maksymalna liczba wątków na blok: " +
parametryGPU.MaxThreadsPerBlock);
        Console.WriteLine("Maksymalna liczba wątków na multiprocesor: " +
parametryGPU.MaxThreadsPerMultiProcessor);
        Console.WriteLine("Liczba wątków w osnowie: " + parametryGPU.WarpSize);
        Console.WriteLine("Maksymalny wymiar sieci: " + parametryGPU.MaxGridSize.x + "x"
+ parametryGPU.MaxGridSize.y + "x" + parametryGPU.MaxGridSize.z);
        Console.WriteLine("Maksymalna liczba kerneli wykonywanych równocześnie: " +
parametryGPU.ConcurrentKernels);
        Console.WriteLine("Zintegrowana karta graficzna: " +
(parametryGPU.Integrated?"TAK":"NIE"));
        Console.WriteLine("Liczba rejestrów na blok:" + parametryGPU.RegistersPerBlock);

        Console.WriteLine();
    }

    Console.ReadKey();
}
```

```
C:\Users\Tomek\Desktop\CUDA w .NET - książka\hello_world\hello_world\bin\x64\Debug\hello_w...
Witaj swiecie!

Informacje o kartach graficznych zainstalowanych w systemie

Nazwa urządzenia: GeForce GT 130M
Numer identyfikacyjny urządzenia: 0
Potencjał obliczeniowy: 1.1
Częstotliwość taktowania zegara: 1500000
Status ECC: wyłączone
Limit czasu działania kernela: włączone
Liczba multiprocesorów: 4
Ilość pamięci globalnej: 536870912B
Ilość pamięci stałej: 65536B
Ilośc pamieci współdzielonej przypadającej na jeden blok: 16384B
Maksymalna liczba wątków: 512x512x1
Maksymalna liczba wątków na blok: 512
Maksymalna liczba wątków na multiprocesor: 768
Liczba wątków w osnowie: 32
Maksymalny wymiar sieci: 65535x65535x1
Maksymalna liczba kerneli wykonywanych równocześnie: 0
Zintegrowana karta graficzna: NIE
Liczba rejestrów na blok:8192
```

Rysunek 17.8. Wynik działania programu z listingu 17.20, który wyświetla podstawowe informacje o karcie graficznej.

Przekazywanie parametrów do kerneli

Stwórzmy nowy projekt typu *Console Application*, o nazwie *Iloczyn_Schura*. Pamiętajmy o dodaniu odpowiednich referencji do biblioteki CUDAfy.NET oraz przestrzeni nazw w kodzie źródłowym tak jak to robiliśmy w programie *hello_world*.

Przed przystąpieniem do implementacji przydałoby się wyjaśnić pojęcie iloczynu Schura (inaczej nazywany także iloczynem Hadamarda lub iloczynem po współrzędnych). Iloczyn Schura (oznaczany przez •) jest operacją matematyczną, która może być wykonywana na macierzach o dowolnych rozmiarach n×m i która dana jest wzorem $(\mathbf{A}\bullet\mathbf{B})_{ij}=a_{ij}b_{ij}$. Zgodnie z tym wzorem, ta operacja polega na pomnożeniu każdego elementu macierzy **A** o indeksie *ij* przez element macierzy **B** o tym samym indeksie i zapisywaniu w macierzy wynikowej, także w elemencie o indeksie *ij*. Dla przykładu

$$\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}\cdot\begin{pmatrix}5 & 6\\ 7 & 8\end{pmatrix}=\begin{pmatrix}1\cdot 5 & 2\cdot 6\\ 3\cdot 7 & 4\cdot 8\end{pmatrix}=\begin{pmatrix}5 & 12\\ 21 & 32\end{pmatrix}$$

Naszym zadaniem będzie przygotowanie kernela umożliwiającego wykonanie takiej operacji. Kernel będzie się nazywał `iloczynSchura`, a jego argumentami będą dwie macierze (elementy macierzy **A** i **B** ułożone będą wierszami) oraz trzecia, do której zapiszemy wynik iloczynu Schura. Jednak z punktu widzenia uczenia się CUDAfy.NET najważniejszym parametrem kernela będzie obiekt `thread` klasy `GThread`. Ta klasa reprezentuje wątki CUDA. Umożliwia ona dostęp do identyfikatora wątku za pomocą pola `threadIdx` typu `dim3` (która reprezentuje natywny typ `dim3` obecny w języku *C for CUDA*). Dzięki niej możemy także odczytać identyfikator bloku (pole `blockIdx`) oraz rozmiar bloku (pole `blockDim`). Wszystkie te pola użyjemy przy implementacji kernela `iloczynSchura`. Zacznijmy więc od zdefiniowania parametrów jakie będzie przyjmować nasz kernel. Będzie nam potrzebny obiekt klasy `GThread` do reprezentowania sieci na jakiej wykonywane będą nasze obliczenia oraz trzy macierze: dwie na dane wejściowe oraz jedna umożliwiająca zwrócenie wyniku obliczeń. Operacje będziemy wykonywać na macierzach o elementach rzeczywistych typu `float`. W związku z tym wszystkie trzy macierze będą typu `float[,]`. Wobec tego implementowany przez nas kernel ma sygnaturę jak na listingu 17.21.

Listing 17.21. Nagłówek kernela wykonującego operację iloczynu Schura.

```
[Cudafy]
```

```
public static void iloczynSchura(GThread thread, float[,] macierzA, float[,] macierzB,
float[,] wynik)
{
}
```

W kolejnym etapie musimy przygotować ciało kernela. Najpierw musimy wyznaczyć indeksy elementów macierzy, które będziemy mnożyć. Muszą one uwzględniać rozmiar bloku oraz identyfikator bloku i wątku, który w danym momencie wykonuje obliczenia. Listing 17.22 pokazuje służące do tego instrukcje. Zmienna `xIndex` reprezentuje wiersze macierzy a zmienna `yIndex` jej kolumny. Kernela `iloczynSchura` możemy użyć do wyznaczenia iloczynu Schura dowolnie dużych macierzy dzięki uwzględnieniu numeru identyfikacyjnego bloku oraz jego rozmiaru przy obliczaniu indeksów wyznaczanego element.

Listing 17.22. Wyznaczenie indeksów elementu macierzy

```
[Cudafy]
public static void iloczynSchura(GThread thread, float[,] macierzA, float[,] macierzB,
float[,] wynik)
{
   int xIndex = thread.blockIdx.x * thread.blockDim.x + thread.threadIdx.x;
   int yIndex = thread.blockIdx.y * thread.blockDim.y + thread.threadIdx.y;
}
```

Obliczenie wartości elementu macierzy `wynik` jest już bardzo prostą czynnością. Używając wyznaczonych wartości indeksów mnożymy element macierzy **A** przez odpowiedni element macierzy **B**, a wyniki zapisujemy do zmiennej `wynik` (listing 17.23).

Listing 17.23. Wyznaczenie iloczynu Schura dla elementu macierzy o indeksach xIndex i yIndex.

```
[Cudafy]
public static void iloczynSchura(GThread thread, float[,] macierzA, float[,] macierzB,
float[,] wynik)
{
   int xIndex = thread.blockIdx.x * thread.blockDim.x + thread.threadIdx.x;
   int yIndex = thread.blockIdx.y * thread.blockDim.y + thread.threadIdx.y;

   wynik[xIndex, yIndex] = macierzA[xIndex, yIndex] * macierzB[xIndex, yIndex];
}
```

W efekcie mamy już zaimplementowany, najprościej jak tylko to możliwe, kernel obliczający iloczyn Schura dwóch macierzy o elementach typu `float`. Musimy go teraz wywołać w funkcji `Main`. W tym celu najpierw należy stworzyć odpowiednie obiekty reprezentujące moduł CUDA oraz uchwyt do karty graficznej, tak jak to zrobiliśmy w przypadku programu *hello_world*. Na listingu 17.24 przedstawiam realizującą to zadanie funkcję `Main`.

Listing 17.24. Funkcja Main programu Iloczyn_Schura.

```
static void Main(string[] args)
{
   CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_11);
   GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Cuda);

   uchwytGPU.LoadModule(modułCuda);

   //w tym miejscu dodamy kod wykonujący kernela iloczynSchura na przykładowych danych

   Console.ReadKey();
}
```

Operacje na pamięci globalnej karty graficznej

Zanim jednak pierwszy raz wykonamy nasz kernel, musimy przygotować przykładowe dane, a następnie skopiować je do pamięci karty graficznej. W przypadku kart graficznych mamy do czynienia z kilkoma rodzajami pamięci. Można tu wymienić miedzy innymi pamięć globalną, pamięć stałą czy współdzieloną. O typach pamięci dostępnej na kartach graficznych więcej możemy przeczytać w dokumencie dostarczonym przez firmę NVidia w formie pliku PDF o nazwie *CUDA_C_Programming_Guide.pdf*. W naszym programie na początek do przechowywania macierzy, na których będziemy wykonywać operacje wykorzystamy pamięć globalną, potem skorzystamy z pamięci współdzielonej w celu optymalizacji dostępu do danych

Przed przesłaniem danych do karty graficznej musimy je stworzyć w pamięci operacyjnej komputera. Zainicjujmy więc macierze **A** i **B**, których iloczyn będziemy obliczać oraz stwórzmy macierz `Wynik`, do której skopiujemy wynik operacji znajdujący się w pamięci karty graficznej. Zmodyfikujmy kod programu przedstawionym na listing 17.24 według wzoru z listingu 17.25. Aby móc łatwo ocenić czy nasz kernel poprawnie wykonuje obliczenia stwórzmy macierze o rozmiarze 2 na 2 i o wartościach elementów takich samych jak w przykładzie z podrozdziału *Przekazywanie parametrów do kerneli*.

Listing 17.24. Inicjacja macierzy w pamięci operacyjnej komputera

```
static void Main(string[] args)
{
    CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_11);
    GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Cuda);

    uchwytGPU.LoadModule(modułCuda);

    //w tym miejscu dodamy kod wykonujący kernela iloczynSchura na przykładowych danych
    float[,] A = new float[,] { { 1, 2 }, { 3, 4 } };
    float[,] B = new float[,] { { 5, 6 }, { 7, 8 } };
    float[,] Wynik = new float[2, 2];

    Console.ReadKey();
}
```

Mając już przygotowane dane musimy przydzielić pamięć globalną karty graficznej, w której te dane będziemy przechowywać. W tym celu użyjemy metody `Allocate` – składowej obiektu `uchwytGPU`. Ta metoda dla typu `float[,]` pobiera jako argument macierz, którą chcemy skopiować do pamięci globalnej układu graficznego. Metoda nie kopiuje jednak danych do pamięci karty. Na jej podstawie ustala ona tylko rozmiar pamięci jaki należy dla niej zaalokować. Metoda zwraca tablicę jednoelementową, która przechowywana jest w pamięci globalnej karty. Nie należy jednak używać jej w kodzie tak jak zwykłej tablicy znanej z języka C#. Należy ją traktować raczej jak wskaźnik do pamięci globalnej karty.

Następnym krokiem jest skopiowanie danych z tablic znajdujących się w pamięci operacyjnej komputera do pamięci globalnej karty. W tym celu używamy metody `CopyToDevice`, która także jest składową klasy dostępną przez obiekt `uchwytGPU`. W najprostszej postaci przyjmuje ona jako argumenty wywołania dwie macierze typu `float[,]`: źródłową oraz docelową. Po skopiowaniu wszystkich danych możemy uruchomić kernel i wykonać obliczenia. Będziemy je wykonywać na jednym bloku o rozmiarze 2 na 2 wątki. Ogólna ilość wątków biorących udział w obliczeniach jest dokładanie taka sama jak ilość elementów w naszych macierzach. Oznacza to, że obliczenia wykonamy dla wszystkich elementów w tym samym czasie. Zwróćmy uwagę, że teraz, uruchamiając kernel, musimy podać argumenty jego wywołania. Wszystkie powyższe modyfikacje kodu programu przedstawione są na listingu 17.25.

Listing 17.25. Inicjacja tablic w pamięci globalnej układu graficznego oraz wykonanie kernela iloczynSchura

```
static void Main(string[] args)
{
    CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_11);
    GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Cuda);
```

```
        uchwytGPU.LoadModule(modułCuda);

        //w tym miejscu dodamy kod wykonujący kernela iloczynSchura na przykładowych danych
        float[,] A = new float[,] { { 1, 2 }, { 3, 4 } };
        float[,] B = new float[,] { { 5, 6 }, { 7, 8 } };
        float[,] Wynik = new float[2, 2];

        float[,] gpu_A = uchwytGPU.Allocate<float>(A);
        float[,] gpu_B = uchwytGPU.Allocate<float>(B);
        float[,] gpu_Wynik = uchwytGPU.Allocate<float>(Wynik);

        uchwytGPU.CopyToDevice<float>(A, gpu_A);
        uchwytGPU.CopyToDevice<float>(B, gpu_B);

        dynamic startKernel = uchwytGPU.Launch(1, new dim3(2, 2));
        startKernel.iloczynSchura(gpu_A, gpu_B, gpu_Wynik);

        Console.ReadKey();
    }
```

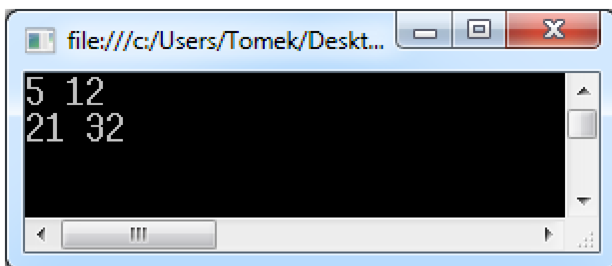
Kod programu przedstawionego na listingu 17.25 powinien skompilować się i wykonać bez problemu. Niestety nie umożliwia on obejrzenia wyników działania kernela. Aby to było możliwe musimy przede wszystkim skopiować wynik operacji do pamięci operacyjnej komputera. Służy do tego metoda `uchwytGPU.CopyFromDevice`, która podobnie jak `CopyToDevice` jako argumenty wywołania przyjmuje dwie tablice: źródłową w pamięci karty graficznej i docelową w pamięci gospodarza. Po skopiowaniu danych możemy już wyświetlić je na monitorze komputera używając metody `Console.WriteLine`. Na zakończenie programu dobrym zwyczajem jest zwolnienie przydzielonej pamięci karty graficznej. Służy do tego metoda `uchwytGPU.Free`, która zwalnia przydzielone zasoby. Jej argumentem jest tablica („wskaźnik”) znajdująca się w pamięci karty.

Listing 17.26. Kopiowanie danych z pamięci karty graficznej i wyświetlenie ich na monitorze komputera

```
    static void Main(string[] args)
    {
        CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_11);
        GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Cuda);

        uchwytGPU.LoadModule(modułCuda);

        //w tym miejscu dodamy kod wykonujący kernela iloczynSchura na przykładowych danych
        float[,] A = new float[,] { { 1, 2 }, { 3, 4 } };
        float[,] B = new float[,] { { 5, 6 }, { 7, 8 } };
        float[,] Wynik = new float[2, 2];

        float[,] gpu_A = uchwytGPU.Allocate<float>(A);
        float[,] gpu_B = uchwytGPU.Allocate<float>(B);
        float[,] gpu_Wynik = uchwytGPU.Allocate<float>(Wynik);

        uchwytGPU.CopyToDevice<float>(A, gpu_A);
        uchwytGPU.CopyToDevice<float>(B, gpu_B);
```

```
        dynamic startKernel = uchwytGPU.Launch(1, new dim3(2, 2));
        startKernel.iloczynSchura(gpu_A, gpu_B, gpu_Wynik);

        uchwytGPU.CopyFromDevice(gpu_Wynik, Wynik);
        Console.WriteLine(Wynik[0, 0] + " " + Wynik[0, 1]);
        Console.WriteLine(Wynik[1, 0] + " " + Wynik[1, 1]);

        uchwytGPU.Free(gpu_A);
        uchwytGPU.Free(gpu_B);
        uchwytGPU.Free(gpu_Wynik);

        Console.ReadKey();
    }
```

Możemy teraz ponownie uruchomić nasz program i sprawdzić czy otrzymane wyniki (por. rysunek 17.9) zgadzają się z tymi z podrozdziału *Przekazywanie parametrów do kerneli*. Wyniki powinny być dokładnie takie same, co oznacza że prawidłowo zaimplementowaliśmy i użyliśmy kernela `iloczynSchura`.



Rysunek 17.9. Program Iloczyn_Schura w działaniu.

Pomiar czasu wykonania

Zostawmy na razie implementację kerneli. Zajmijmy się natomiast sprawdzeniem ile czasu potrzebuje karta graficzna na wykonanie naszych obliczeń. W tym celu zmodyfikujmy program z listingu 17.26 tak, aby obliczenia wykonywały się na znacznie większych macierzach przez co czas wykonywania programu będzie dłuższy. Szczegółowe zmiany w kodzie źródłowym pokazane są na listingu 17.27. Najpierw zdefiniowałem zmienne określające rozmiar macierzy (dla uproszczenia zakładam, że macierze są kwadratowe) oraz liczbę wątków w jednym wymiarze. Następnie zmodyfikowałem instrukcje tworzące macierze `A`, `B` i `Wynik`. W kolejnym kroku wywołałem funkcję `InicjacjaTablic` inicjującą elementy macierzy `A` i `B`. Nie będę omawiał szczegółowo tej funkcji – jej kod jest bardzo prosty. Poza tym sposób inicjacji tablic nie jest ważny z punktu widzenia korzystania z biblioteki CUDAfy.NET. Ponieważ w obecnej implementacji rozmiary macierzy są znacznie większe, to modyfikacji musiało ulec także wywołanie metody `uchwytGPU.Launch`. W tej wersji programu obliczenia są wykonywane na więcej niż jednym bloku. Liczba bloków w wymiarze X i w wymiarze Y jest taka sama i równa jest `liczbaDanychWjednymWymiarze/liczbaWątkówWjednymwymiarze`. Zmodyfikowałem także sposób wyświetlania danych wejściowych i wyniku. Ponieważ macierze są duże, wyświetlam tylko podmacierz o rozmiarach 3 na 3 elementy.

Listing 17.27. Modyfikacje mające na celu wykonywanie obliczeń na macierzach o dużych rozmiarach

```
    static void Main(string[] args)
    {
        CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_11);
        GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Cuda);

        uchwytGPU.LoadModule(modułCuda);
```

```
    int liczbaDanychWjednymWymiarze = 512;
    int liczbaWątkówWjednymWymiarze = 16;

    //w tym miejscu dodamy kod wykonujący kernela iloczynSchura na przykładowych danych
    float[,] A = new float[liczbaDanychWjednymWymiarze, liczbaDanychWjednymWymiarze];
    float[,] B = new float[liczbaDanychWjednymWymiarze, liczbaDanychWjednymWymiarze];
    float[,] Wynik = new float[liczbaDanychWjednymWymiarze,
liczbaDanychWjednymWymiarze];

    InicjacjaTablic(A, B);

    float[,] gpu_A = uchwytGPU.Allocate<float>(A);
    float[,] gpu_B = uchwytGPU.Allocate<float>(B);
    float[,] gpu_Wynik = uchwytGPU.Allocate<float>(Wynik);

    uchwytGPU.CopyToDevice<float>(A, gpu_A);
    uchwytGPU.CopyToDevice<float>(B, gpu_B);

    dynamic startKernel = uchwytGPU.Launch(new dim3(liczbaDanychWjednymWymiarze /
liczbaWątkówWjednymwymiarze, liczbaDanychWjednymWymiarze / liczbaWątkówWjednymWymiarze),
            new dim3(liczbaWątkówWjednymwymiarze, liczbaWątkówWjednymWymiarze));

    startKernel.iloczynSchura(gpu_A, gpu_B, gpu_Wynik);

    uchwytGPU.CopyFromDevice(gpu_Wynik, Wynik);
    Console.WriteLine("Macierz A:");
    WyświetlElementy(A);
    Console.WriteLine("Macierz B:");
    WyświetlElementy(B);
    Console.WriteLine("Wynik obliczeń:");
    WyświetlElementy(Wynik);

    uchwytGPU.Free(gpu_A);
    uchwytGPU.Free(gpu_B);
    uchwytGPU.Free(gpu_Wynik);

    Console.ReadKey();
}
private static void WyświetlElementy(float[,] macierz)
{
    Console.WriteLine(macierz[0, 0] + " " + macierz[0, 1] + " " + macierz[0, 2]);
    Console.WriteLine(macierz[1, 0] + " " + macierz[1, 1] + " " + macierz[1, 2]);
    Console.WriteLine(macierz[2, 0] + " " + macierz[2, 1] + " " + macierz[2, 2]);
}
static public void InicjacjaTablic(float[,] A, float[,] B)
{
    for (int i=0; i<A.GetLongLength(0); i++)
```

```
            for (int j = 0; j < A.GetLongLength(1); j++)
            {
                A[i, j] = B[i, j] = i + j;
            }
    }
```

Następnie dodajemy do funkcji `Main` instrukcje mające na celu wyznaczenie czasu wykonywania kernela `iloczynSchura`. Biblioteka CUDAfy.NET umożliwia pomiar tego czasu w bardzo łatwy sposób. Dostarcza ona metod `StartTimer` i `StopTimer` składowych klasy `GPGPU`. Pierwsza z nich uruchamia pomiar czasu, a druga zwraca czas jaki upłynął pomiędzy wywołaniami tych dwóch metod. Jednak aby czas wykonania został poprawnie zmierzony należy zsynchronizować wszystkie wątki, których używamy do obliczeń tuż przed zatrzymaniem pomiaru czasu. Służy do tego metoda `Synchronize`, również składowa klasy `GPGPU`. Instrukcje służące do pomiaru czasu wykonania kernela zostały wyróżnione na listingu 17.28.

W celu porównania o ile obliczenia wykonywane na GPU są szybsze od obliczeń na CPU dodałem funkcje obliczającą iloczyn Schura dwóch macierz na CPU. Do wyznaczenia czasu wykonania tej funkcji użyłem instancji klasy `Stopwatch` (listing 17.28).

Listing 17.28. Modyfikacje programy z listingu 17.27 umożliwiające pomiar czasu wykonania kernela oraz jego porównanie do czasu wykonywania takich samych obliczeń ale na CPU.

```
    static void Main(string[] args)
    {
        CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_11);
        GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Cuda);

        uchwytGPU.LoadModule(modułCuda);

        int liczbaDanychWjednymWymiarze = 512;
        int liczbaWątkówWjednymWymiarze = 16;

        //w tym miejscu dodamy kod wykonujący kernela iloczynSchura na przykładowych danych
        float[,] A = new float[liczbaDanychWjednymWymiarze, liczbaDanychWjednymWymiarze];
        float[,] B = new float[liczbaDanychWjednymWymiarze, liczbaDanychWjednymWymiarze];
        float[,] Wynik = new float[liczbaDanychWjednymWymiarze,
    liczbaDanychWjednymWymiarze];

        InicjacjaTablic(A, B);

        float[,] gpu_A = uchwytGPU.Allocate<float>(A);
        float[,] gpu_B = uchwytGPU.Allocate<float>(B);
        float[,] gpu_Wynik = uchwytGPU.Allocate<float>(Wynik);

        uchwytGPU.CopyToDevice<float>(A, gpu_A);
        uchwytGPU.CopyToDevice<float>(B, gpu_B);

        dynamic startKernel = uchwytGPU.Launch(new dim3(liczbaDanychWjednymWymiarze /
    liczbaWątkówWjednymWymiarze, liczbaDanychWjednymWymiarze / liczbaWątkówWjednymWymiarze),
                    new dim3(liczbaWątkówWjednymWymiarze, liczbaWątkówWjednymWymiarze));

        uchwytGPU.StartTimer();
```

```
    startKernel.iloczynSchura(gpu_A, gpu_B, gpu_Wynik);
    uchwytGPU.Synchronize();
    float upłynęłoCzasu = uchwytGPU.StopTimer();

    uchwytGPU.CopyFromDevice(gpu_Wynik, Wynik);
    Console.WriteLine("Macierz A:");
    WyświetlElementy(A);
    Console.WriteLine("Macierz B:");
    WyświetlElementy(B);
    Console.WriteLine("Wynik obliczeń na GPU:");
    WyświetlElementy(Wynik);
    Console.WriteLine();
    Console.WriteLine("Czas wykonania kernela: {0} ms", upłynęłoCzasu);

    uchwytGPU.Free(gpu_A);
    uchwytGPU.Free(gpu_B);
    uchwytGPU.Free(gpu_Wynik);

    Stopwatch stopWatch = new Stopwatch();
    stopWatch.Start();
    iloczynSchuraNaCPU(A, B, Wynik);
    stopWatch.Stop();
    Console.WriteLine("Wynik obliczeń na CPU:");
    WyświetlElementy(Wynik);
    TimeSpan ts = stopWatch.Elapsed;
    Console.WriteLine();
    Console.WriteLine("Czas wykonania obliczeń na CPU: {0} ms", ts.TotalMilliseconds);

    Console.WriteLine();
    Console.WriteLine("Przyśpieszenie obliczeń w stosunku do CPU: {0} ",
ts.TotalMilliseconds / upłynęłoCzasu);

    Console.ReadKey();
}
public static void iloczynSchuraNaCPU(float[,] macierzA, float[,] macierzB, float[,]
wynik)
{
    for (int i = 0; i < macierzA.GetLongLength(0); i++)
        for (int j = 0; j < macierzA.GetLongLength(1); j++)
        {
            wynik[i, j] = macierzA[i, j] * macierzB[i, j];
        }
}
```

```
file:///c:/Users/Tomek/Desktop/CUDA w .NET - książka/Iloczyn_Schura/Iloczyn_Schura/bin/Releas...
Macierz A:
0 1 2
1 2 3
2 3 4
Macierz B:
0 1 2
1 2 3
2 3 4
Wynik obliczeń na GPU:
0 1 4
1 4 9
4 9 16

Czas wykonania kernela: 2,853184 ms
Wynik obliczeń na CPU:
0 1 4
1 4 9
4 9 16

Czas wykonania obliczeń na CPU: 9,4704 ms

Przyśpieszenie obliczeń w stosunku do CPU: 3,31923915548276
```

Rysunek 17.10. Wyniki działania programu z listingu 17.28. Zależą one mocno od posiadanego sprzętu, a więc zapewne większość czytelników będzie miała zupełnie inne wyniki niż przedstawione na tym rysunku.

Wyniki działania programu na moim systemie, który posiada układ graficzny GeForce GT 130M z 32 rdzeniami oraz dwurdzeniowy procesor Intel Core 2 Duo P8600 2.40GHz przedstawione są na rysunku 17.10. Jak widać przyspieszenie obliczeń wykonywanych na GPU w stosunku do obliczeń wykonywanych na CPU jest około trzykrotne. Wykonywane przez nas obliczenia nie są skomplikowane, dlatego przyśpieszenie nie jest zbyt duże. Poza tym karta graficzna, którą posiadam jest układem przeznaczonym do laptopów. Przyśpieszenie obliczeń jakie można z jej pomocą uzyskać jest siłą rzeczy ograniczone. Z drugiej strony obliczenia na CPU wykonywane są tylko na jednym rdzeniu. Czas jaki został wyznaczony dla jednego rdzenia można ekstrapolować na wiele rdzeni dzieląc go przez ilość rdzeni. Wartość jaką w ten sposób otrzymamy będzie ograniczeniem z dołu czasu wykonywania obliczeń. Czas ten jest oczywiście tylko wartością teoretyczną i w praktyce bardzo rzadko osiąganą. Zauważmy, że szacując czas wykonywania obliczeń dla np. czterech rdzeni CPU, nie uzyskujemy żadnego przyśpieszenia. Wręcz przeciwnie - program będzie wykonywał się wolniej na GPU niż na CPU!

Dostęp zwarty do pamięci globalnej i pamięć współdzielona

Zajrzyjmy do pliku CUDAFYSOURCETEMP.cu i sprawdźmy jak wygląda kernel `iloczynSchura` przetłumaczony na język *C for CUDA*. W szczególności przyjrzyjmy się ostatniej instrukcji w tym kernelu (por. listing 17.29):

```
wynik[(num) * wynikLen1 + ( num2)] = macierzA[(num) * macierzALen1 + ( num2)] *
macierzB[(num) * macierzBLen1 + ( num2)];
```

Zauważmy, że do przechodzenia pomiędzy elementami w danym wierszu macierzy używana jest zmienna `num2`, która jest wyznaczana na podstawie składowej *y* identyfikatora bloku oraz wątku:

```
int num2 = blockIdx.y * blockDim.y + threadIdx.y;
```

Przypomnę, że wątki „znajdujące się obok siebie” są numerowane tą samą wartością składowej *y* i różnymi, zmieniającymi się o 1 wartościami składowej *x*. Przypomnę także, że aby dostęp do pamięci globalnej był jak najbardziej optymalny, należy na niej wykonywać operacje w tzw. zwarty sposób (ang. *coalesced*)[10]. Oznacza to, że w naszym przypadku operacje na macierzach są wykonywane w nieoptymalny sposób.

Listing 17.29. Kernel z listing 17.23 skonwertowany do języka „C for CUDA”. Poniższy kod znajduje się w pliku CUDAFYSOURCETEMP.cu.

[10] O dostępie zwartym do pamięci oraz innych optymalizacjach, które znacznie przyśpieszają obliczenia, można przeczytać w dokumencie *CUDA C BEST PRACTICES GUIDE* dostarczonym przez firmę NVidia razem z SDK.

```
// Iloczyn_Schura.Program
extern "C" __global__ void iloczynSchura(float* macierzA, int macierzALen0, int
macierzALen1, float* macierzB, int macierzBLen0, int macierzBLen1, float* wynik, int
wynikLen0, int wynikLen1);
```

```
// Iloczyn_Schura.Program
extern "C" __global__ void iloczynSchura(float* macierzA, int macierzALen0, int
macierzALen1, float* macierzB, int macierzBLen0, int macierzBLen1, float* wynik, int
wynikLen0, int wynikLen1)
{
  int num = blockIdx.x * blockDim.x + threadIdx.x;
  int num2 = blockIdx.y * blockDim.y + threadIdx.y;
  wynik[(num) * wynikLen1 + ( num2)] = macierzA[(num) * macierzALen1 + ( num2)] *
macierzB[(num) * macierzBLen1 + ( num2)];
}
```

Postaramy się to zmienić. Utwórzmy kopię naszego kernela i nazwijmy go `iloczynSchuraPamiecWspoldzielona`. Parametry wywołania tego kernela są takie same jak jego poprzednika. W celu zapewnienie zwartego dostępu do pamięci globalnej musimy utworzyć dwie macierze w pamięci współdzielonej (tego typu pamięć jest dostępna dla wszystkich wątków w tym samym bloku). Do tych macierzy będziemy kopiować dane z pamięci globalnej. Wynik obliczeń również będzie przechowywany w pamięci współdzielonej. W ostatnim kroku zsynchronizujemy wątki oraz skopiujemy wyniki z pamięci współdzielonej do pamięci globalnej karty graficznej (zoptymalizujemy w ten sposób zapis danych wyznaczonych przez kernela do pamięci globalnej karty).

Zastanówmy się w jakiej kolejności powinniśmy wyznaczać elementy macierzy z wynikami, aby operacje wykonywane na pamięci odbywały się w sposób zwarty. Otóż należy je obliczać w taki sposób aby wątki o kolejnych identyfikatorach `threadIdx.x` wyznaczały kolejne elementy macierzy `wynik` w danym wierszu.

Z początku to wszystko może wydawać się dość trudne, ale zapoznanie się z kodem kernela `iloczynSchuraPamiecWspoldzielona` przedstawionym na listingu 17.30 powinno ułatwić zrozumienie problemu. Do alokacji pamięci współdzielonej używamy metody `AllocateShared`, składowej klasy `GThread` (listing 17.30). Przyjmuje ona trzy argumenty: nazwę bufora (w naszym przypadku macierzy) oraz jego rozmiary (dwie wartości ponieważ bufor jest macierzą). Rozmiar pamięci współdzielonej musi być znany już w momencie kompilacji programu. Z tego powodu zmieniłem definicję zmiennej `liczbaWątkówWjednymWymiarze` na `const int liczbaWątkówWjednymWymiarze = 16;`. Przeniosłem ją również poza funkcję `Main` tak, aby była dostępna także w kernelu.

Listing 17.29. Wykorzystanie pamięci współdzielonej do zaimplementowania dostępu zwartego do pamięci globalnej karty graficznej.

```
[Cudafy]
public static void iloczynSchuraPamiecWspoldzielona(GThread thread, float[,] macierzA,
float[,] macierzB, float[,] wynik)
{
    float[,] cacheMacierzA = thread.AllocateShared<float>("cache",
liczbaWątkówWjednymWymiarze, liczbaWątkówWjednymWymiarze);
    float[,] cacheMacierzB = thread.AllocateShared<float>("cache",
liczbaWątkówWjednymWymiarze, liczbaWątkówWjednymWymiarze);


    int xIndex = thread.blockIdx.x * thread.blockDim.x + thread.threadIdx.x;
    int yIndex = thread.blockIdx.y * thread.blockDim.y + thread.threadIdx.y;


    cacheMacierzA[yIndex, xIndex] = cacheMacierzA[yIndex, xIndex] *
cacheMacierzB[yIndex, xIndex];
    thread.SyncThreads();
    wynik[yIndex, xIndex] = cacheMacierzA[yIndex, xIndex];
}
```

Proszę zauważyć, że zamieniłem miejscami indeksy yIndex oraz xIndex. Teraz yIndex indeksuje wiersze a xIndex kolumny. Dzięki temu wątki o kolejnych numerach identyfikowanych przez threadIdx.x i tym samym threadIdx.y będą wyznaczały wiersze macierzy wynik. Wszystkie powyższe modyfikację powodują, że kernel iloczynSchuraPamiecWspoldzielona wykorzystuje dostęp zwarty do pamięci globalnej komputera. Przekonajmy się o tym wywołując nowy kernel i sprawdzając o ile szybciej wykonuje obliczenia (listing 17.30).

Listing 17.30. Modyfikacje jakie należy wykonać w programie aby uruchomić i sprawdzić czas wykonania kernela iloczynSchuraPamiecWspoldzielona. Listing przedstawia tylko fragmenty programu, które uległy zmianie

```
class Program
{
    const int liczbaWątkówWjednymWymiarze = 16;

    static void Main(string[] args)
    {
        ...
         uchwytGPU.LoadModule(modułCuda);

        int liczbaDanychWjednymWymiarze = 512;
        int liczbaWątkówWjednymWymiarze = 16;
        ...
        Console.WriteLine("Czas wykonania kernela: {0} ms", upłynęłoCzasu);

        uchwytGPU.StartTimer();
        startKernel.iloczynSchuraPamiecWspoldzielona(gpu_A, gpu_B, gpu_Wynik);
        uchwytGPU.Synchronize();
        upłynęłoCzasu = uchwytGPU.StopTimer();
        WyświetlElementy(Wynik);
        Console.WriteLine();
        Console.WriteLine("Czas wykonania kernela (wyk. pamięci współdzielonej): {0}
ms", upłynęłoCzasu);

        uchwytGPU.Free(gpu_A);
        ...
    }
}
```

```
file:///c:/Users/Tomek/Desktop/CUDA w .NET - książka/Iloczyn_Schura/Iloczyn_Schura/bin/Releas...
Macierz A:
0 1 2
1 2 3
2 3 4
Macierz B:
0 1 2
1 2 3
2 3 4
Wynik obliczeń na GPU:
0 1 4
1 4 9
4 9 16

Czas wykonania kernela: 3,12912 ms
0 1 4
1 4 9
4 9 16

Czas wykonania kernela (wyk. pamięci współdzielonej): 0,227552 ms
Wynik obliczeń na CPU:
0 1 4
1 4 9
4 9 16

Czas wykonania obliczeń na CPU: 9,283 ms

Przyśpieszenie obliczeń w stosunku do CPU: 40,7950716069329
```

Rysunek 17.11. Wyniki działania programu z listingu 17.30. Czerwonymi kwadratami oznaczyłem najważniejsze wyniki.

Rysunek 17.11 prezentuje uzyskane wyniki. Najciekawsza jest wartość przyspieszenia obliczeń wykonywanych przez kernela `iloczynSchuraPamiecWspoldzielona` w stosunku do obliczeń z wykorzystaniem CPU (ostatnia linia). W przypadku mojego systemu to przyśpieszenie wynosiło prawie 41! Myślę, że ten wynik powinien usatysfakcjonować każdego. W porównaniu do poprzedniej wersji kernela, która wykonywała te same obliczenia nie dbając o optymalny dostęp do pamięci globalnej, jest to ogromne, prawie czternastokrotne przyśpieszenie. Porównując uzyskany czas obliczeń za pomocą kernela `iloczynSchuraPamiecWspoldzielona` do teoretycznego czasu wykonywania obliczeń przez przykładowe cztery rdzenie CPU przyśpieszenie jakie uzyskaliśmy jest dziesięciokrotne. Ten wynik również jest imponujący.

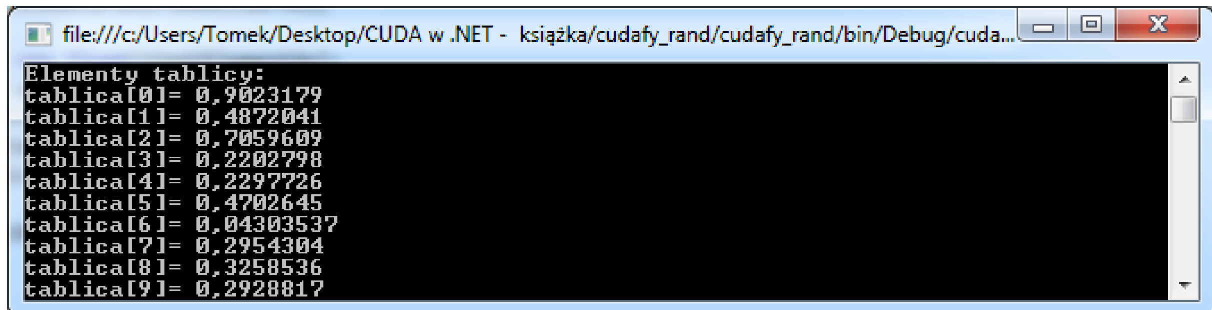
Powyższy przykład pokazuje jak ważne jest dbanie o prawidłowy dostęp do pamięci globalnej karty graficznej. W porównaniu ze standardowymi obliczeniami wykonywanymi za pomocą CPU, tego typu problemy wymuszają na programiście znacznie lepszą znajomość sprzętu, na którym wykonywane są obliczenia.

Generator liczb pseudolosowych

Firma NVidia udostępniła bibliotekę *cuRAND*, która umożliwia generowanie liczb pseudolosowych za pomocą karty graficznej. Dostępnych jest kilka algorytmów. Dokumentacja dla tej biblioteki znajduje się pod adresem *http://docs.nvidia.com/cuda/curand/index.html*. Biblioteka CUDAfy.NET również umożliwia wykorzystanie zalet *cuRAND* w aplikacji C#. Pokażę to w tym podrozdziale.

Stwórzmy nowy projekt o nazwie *cudafy_rand*, a następnie dodajmy do niego referencję do biblioteki CUDAfy.NET. Jak zawsze dodajemy także odpowiednie przestrzenie nazw oraz tworzymy instancje niezbędnych klas. Oprócz poznanych wcześniej przestrzeni nazw dodajemy także `Cudafy.Maths.RAND`. Zawiera ona klasy, które umożliwiają generowanie liczba za pomocą GPU. W funkcji `Main` tworzymy za pomocą metody `Create` instancję klasy `GPGPURAND` o nazwie `generujLiczby`. W jednej ze swych wersji, której będziemy używać, metoda `Create` przyjmuje trzy argumenty: uchwyt reprezentujący kartę graficzną, parametr typu wyliczeniowego `curandRngType` oraz parametr typu `bool`. Pierwszy parametr jest już nam znany z poprzednich programów. Drugi umożliwia wybór jednej z metod generowania liczb pseudolosowych. Ja wybiorę `CURAND_RNG_PSEUDO_XORWOW`, ale zachęcam czytelnika do przetestowania również innych. Trzeci argument mówi o tym czy wygenerowane liczby mają być skopiowane do pamięci RAM komputera (wartość `true`) czy też mają być przechowywane w pamięci karty graficznej (wartość `false`). Domyślnie wartość tego parametru jest ustawiona na `false` i taką ją pozostawmy.

Kolejnym krokiem jest ustawienie *ziarna*. Ziarno to wartość określająca stan początkowy generatora. Dla tej samej wartości ziarna wygenerowane liczby pseudolosowe są takie same. Ziarno możemy ustawić za pomocą metody `generujLiczby.SetPseudoRandomGeneratorSeed`. Ja ustawiłem je na 30 000.
Następnym krokiem jest stworzenie tablic typu `float`, w których będziemy przechowywać wygenerowane liczby. Do nich zapiszemy liczby wygenerowane metodą `generujLiczby.GenerateUniform`. Ta metoda generuje liczby z przedziału [0, 1]. Oprócz niej dostępne są także inne metody, które umożliwiają generowanie liczb pseudolosowych np. o typie całkowitym (metoda `Generate`), czy też posiadające rozkład normalny (metoda `GenerateNormal`). Na koniec pozostaje już nam tylko skopiować dane z karty graficznej do pamięci komputera i wyświetlić je na monitorze komputera (rysunek 17.12).



Rysunek 17.12. Wyniki działania programu cudafy_rand.

Cały kod programu *cudafy_rand* przedstawiony jest na listingu 17.31. Warto wspomnieć, że jeśli w metodzie `Create` ostatniemu parametrowi nadamy wartość `true`, a więc chcemy, aby dane były automatycznie skopiowane do pamięci komputera, do metody `generujLiczby.GenerateUniform` w pierwszym argumencie należy przekazać tablicę, która zaalokowana jest w pamięci RAM komputera.

Listing 17.31. Program cudafy_rand generujący dziesięć liczb pseudolosowych dla ziarna równego 30 000 i wyświetlający je na monitorze komputera

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Cudafy;
using Cudafy.Host;
using Cudafy.Translator;

using Cudafy.Maths.RAND;

namespace cudafy_rand
{
    class Program
    {
        const ulong ziarno = 30000;
        const int ilośćElementów = 10;

        static void Main(string[] args)
        {
            CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_11);
            GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Cuda);
            uchwytGPU.LoadModule(modułCuda);
```

```
            GPGPURAND generujLiczby = GPGPURAND.Create(uchwytGPU,
curandRngType.CURAND_RNG_PSEUDO_XORWOW);
            generujLiczby.SetPseudoRandomGeneratorSeed(ziarno);

            float[] tablicaLiczbCPU = new float[ilośćElementów];
            float[] tablicaLiczbGPU = uchwytGPU.Allocate<float>(tablicaLiczbCPU);
            generujLiczby.GenerateUniform(tablicaLiczbGPU);
            uchwytGPU.CopyFromDevice(tablicaLiczbGPU, tablicaLiczbCPU);
            WyświetlWartości(tablicaLiczbCPU);

            uchwytGPU.Free(tablicaLiczbGPU);

            Console.ReadKey();
        }

        static void WyświetlWartości(float[] tablica)
        {
            Console.WriteLine("Elementy tablicy: ");
            for (int i = 0; i < tablica.Length; i++)
                Console.WriteLine("tablica[" +i + "]= " + tablica[i]);
        }
    }
}
```

FFT na GPU

Oprócz biblioteki *cuRAND* NVidia dostarcza także bibliotekę *cuFFT*, która umożliwia wykonywanie szybkiej transformaty Fouriera (FFT) korzystając z GPU. FFT jest operacją bardzo często używaną w nauce, inżynierii, czy obróbce grafiki. Biblioteka CUDAfy.NET także umożliwia wykorzystanie *cuFFT* w programach pisanych w C#. W tym podrozdziale pokażę w jaki sposób można tego dokonać modyfikując program *cudafy_rand*.

Spróbujemy policzyć transformatę FFT na danych, które wygenerowaliśmy w poprzednim podrozdziale za pomocą generatora liczb pseudolosowych. Po pierwsze musimy utworzyć instancję klasy `GPGPUFFT`. Nazwijmy ją `gpuFFT`. Tworzymy ją wywołując, podobnie jak w przypadku `GPGPURAND`, metodę `Create`. Pobiera ona jeden parametr będący uchwytem reprezentującym kartę graficzną. Następnie tworzymy instancję klasy `FFTPlan1D`. Ta klasa przygotowuje bibliotekę *cuFFT* pod względem optymalizacji obliczeń jednowymiarowego FFT, uwzględniając własności sprzętu, na którym obliczenia są wykonywane. Oprócz `FFTPlan1D` biblioteka dostarcza także klasy `FFTPlan2D` i `FFTPlan3D`, które umożliwiają obliczanie szybkich transformat Fouriera na danych dwu- i trójwymiarowych. Instancję klasy `FFTPlan1D` tworzymy wywołując metodę `gpuFFT.Plan1D`, która przyjmuje trzy argumenty. Pierwszy jest typu `eFFTType` i określa typ wykonywanego FFT. My wykonamy FFT na danych rzeczywistych a wynik będzie liczbami zespolonymi (`eFFTType.Real2Complex`). Drugi parametr oznacza typ danych. W naszym przypadku jest to `eDataType.Single`. Ostatni argument określa ilość danych, na których chcemy wykonać FFT (w tworzonym programie jest to `ilośćElementów`). W następnym kroku tworzymy dwie tablice przechowujące wyniki operacji FFT: jedną w pamięci karty graficznej i jedną w pamięci operacyjnej komputera. Obydwie tablice mają rozmiar `ilośćElementów` i są typu `ComplexF`. Typ `ComplexF` jest strukturą dostarczoną przez bibliotekę CUDAfy.NET. Reprezentuje on typ natywny `cufftComplex` dostępny w języku *C for CUDA*. Po stworzeniu tablic możemy wykonać operację FFT na przykładowych danych. Służy do tego metoda `fft1D.Execute`. Pobiera ona trzy argumenty. Pierwsze dwa to tablice: przechowująca dane oraz wynik. Ostatni parametr jest typu `bool` i określa czy ma być wykonana transformata w przód (wartość `false`) czy odwrotna (wartość `true`). Po wykonaniu FFT pozostaje nam skopiować dane do pamięci RAM oraz wyświetlić je na monitorze. Proszę zauważyć, że liczba danych

wyjściowych jest równa połowie ilości danych wejściowych plus jeden[11]. W związku z tym przeciążyłem metodę WyświetlWartości. Zadaniem nowej metody jest wyświetlić wynik operacji FFT na danych rzeczywistych w odpowiedni sposób. Listing 17.32 przedstawia wszystkie niezbędne modyfikacje, które należy wykonać, aby obliczyć FFT na danych losowych.

Listing 17.32. Obliczenie szybkiej transformaty Fouriera na danych wygenerowanych przy użyciu generatora liczb pseudolosowych. Zmieniona została również liczba generowanych danych na 32.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Cudafy;
using Cudafy.Host;
using Cudafy.Translator;

using Cudafy.Maths.RAND;
using Cudafy.Maths.FFT;
using Cudafy.Types;

namespace cudafy_rand
{
    class Program
    {
        const ulong ziarno = 30000;
        const int ilośćElementów = 32;

        static void Main(string[] args)
        {
            CudafyModule modułCuda = CudafyTranslator.Cudafy(eArchitecture.sm_11);
            GPGPU uchwytGPU = CudafyHost.GetDevice(eGPUType.Cuda);
            uchwytGPU.LoadModule(modułCuda);

            GPGPURAND generujLiczby = GPGPURAND.Create(uchwytGPU, 
curandRngType.CURAND_RNG_PSEUDO_XORWOW);
            generujLiczby.SetPseudoRandomGeneratorSeed(ziarno);

            float[] tablicaLiczbCPU = new float[ilośćElementów];
            float[] tablicaLiczbGPU = uchwytGPU.Allocate<float>(tablicaLiczbCPU);
            generujLiczby.GenerateUniform(tablicaLiczbGPU);
            uchwytGPU.CopyFromDevice(tablicaLiczbGPU, tablicaLiczbCPU);
            WyświetlWartości(tablicaLiczbCPU);



            GPGPUFFT gpuFFT = GPGPUFFT.Create(uchwytGPU);
            FFTPlan1D fft1D = gpuFFT.Plan1D(eFFTType.Real2Complex, eDataType.Single, 
ilośćElementów);
```

[11] Zob. *http://docs.nvidia.com/cuda/cufft/index.html#topic_4_8*

```
            ComplexF[] fftCPU = new ComplexF[ilośćElementów];
            ComplexF[] fftGPU = uchwytGPU.Allocate<ComplexF>(fftCPU);
            fft1D.Execute<float, ComplexF>(tablicaLiczbGPU, fftGPU, false);
            uchwytGPU.CopyFromDevice(fftGPU, fftCPU);
            WyświetlWartości(fftCPU);

            uchwytGPU.Free(tablicaLiczbGPU);

            Console.ReadKey();
        }

        static void WyświetlWartości(ComplexF[] tablica)
        {
            Console.WriteLine("Elementy tablicy: ");
            for (int i = 0; i < (int)Math.Truncate(tablica.Length / 2.0) + 1; i++)
                Console.WriteLine("tablica[" + i + "]= " + tablica[i]);
        }

        static void WyświetlWartości(float[] tablica)
        {
            Console.WriteLine("Elementy tablicy: ");
            for (int i = 0; i < tablica.Length; i++)
                Console.WriteLine("tablica[" +i + "]= " + tablica[i]);
        }
    }
}
```

BLAS

BLAS to skrót od *Basic Linear Algebra Subprograms*. BLAS jest biblioteką służącą do wykonywania operacji algebraicznych na macierzach i wektorach. Powstało wiele bibliotek implementujących funkcjonalność BLAS w przeróżnych językach programowania. NVidia dostarcza bibliotekę *cuBLAS*[12], która również implementuje funkcjonalność swojego pierwowzoru. Nie będę opisywał wszystkich operacji jakie można wykonać za pomocą tej biblioteki, pokażę tylko w jaki sposób można ją wykorzystać w programie C# przy pomocy biblioteki CUDAfy.NET. Zmodyfikujemy w tym celu program z listingu 17.32.

Modyfikacje, które musimy wykonać są bardzo proste, bowiem jedyną niezbędną czynnością jest utworzenie instancji klasy `GPGPUBLAS`. Z jej pomocą możemy wywoływać metody wykonujące podstawowe operacje algebraiczne na wektorach i macierzach. W przypadku programu z listingu 17.33, wykorzystywaną metodą z tej biblioteki jest `DOT`, która oblicza iloczyn skalarny dwóch wektorów. Z opisem wszystkich funkcji jakie mamy do dyspozycji w *cuBLAS* można zapoznać się pod adresem *http://docs.nvidia.com/cuda/cublas/index.html#topic_6_1*.

Listing 17.33. Wyznaczanie iloczynu skalarnego dwóch wektorów za pomocą metody DOT z biblioteki cuBLAS przy użyciu wrappera z biblioteki CUDAfy.NET

```
static void Main(string[] args)
{
    ...
```

[12] Dokumentacja biblioteki jest dostępna pod adresem *http://docs.nvidia.com/cuda/cublas/index.html*.

```
    uchwytGPU.CopyFromDevice(fftGPU, fftCPU);
    WyświetlWartości(fftCPU);

    GPGPUBLAS gpuBLAS = GPGPUBLAS.Create(uchwytGPU);
    float iloczynSkalarny = gpuBLAS.DOT(tablicaLiczbGPU, tablicaLiczbGPU);
    Console.WriteLine("iloczyn skalarny tablicaLiczbGPU * tablicaLiczbGPU = " +
iloczynSkalarny);


    uchwytGPU.Free(tablicaLiczbGPU);

    ...
}
```

Zadania

1. Napisz kernel, który na dwóch wektorach o rozmiarze 1000 przechowywanych w postaci tablicy wykonuje operację o następującej formule $\mathbf{W}_i=\mathbf{X}_i+a*\mathbf{Y}_i$. Parametr a jest liczbą, przez którą mnożone są składowe wektora **Y**. Elementy wektorów są typu `float`.
2. Wygeneruj tablicę 100 liczb pseudolosowych wszystkimi dostępnymi metodami. Sprawdź jaki jest czas generowania liczb dla każdej z nich.
3. W jednym z poprzednich rozdziałów tworzyliśmy aplikację, która wyznaczała wartość liczby π metodą Monte Carlo. Stwórz aplikację, która tą samą metodą wyznaczy wartość liczby π wykonując jednak obliczenia na karcie graficznej.
4. Stwórz tablicę dwuwymiarową składającą się z liczb pseudolosowych wygenerowanych metodą `CURAND_RNG_QUASI_SOBOL32`. Następnie oblicz dla nich szybką transformatę Fouriera 2D.
5. Wykorzystując metody z przestrzeni `Cudafy.Maths.BLAS` oblicz normę euklidesową wektora **W** z zadania 1.