Dodatek A.
CUDA

Trzy ostatnie rozdziały książki poświęcone są zagadnieniom związanym z programowaniem równoległym. Skłoniła nas do tego wszechobecność maszyn wieloprocesorowych. Nawet niektóre notebooki wyposażane są w wiele procesorów, a przynajmniej w procesory wielordzeniowe. W tej perspektywie warto zaznaczyć obecność kilku technologii, które na razie nie mają może szans wejść do powszechnego użytku, ale z pewnością już niedługo zagoszczą w instytutach naukowych, centrach obliczeniowych i wszędzie tam, gdzie szybkość obliczeń ma decydujące znaczenie. Mam na myśli technologię CUDA firmy NVIDIA oraz ATI Stream jej największego konkurenta. W poniższym rozdziale skupimy się na tym pierwszym rozwiązaniu.

O co tak naprawdę chodzi? Technologie CUDA (z ang. *Compute Unified Device Architecture*) i ATI Stream pozwalają na wykorzystanie mocy obliczeniowej procesorów montowanych na współczesnych kartach graficznych (GPU). Chodzi zarówno o specjalne karty dedykowane do obliczeń naukowo-inżynierskich, w których nie ma w ogóle wyjścia wideo (w szczególności układy NVIDII z procesorem Tesla), jak i o „zwykłe" karty np. z serii NVIDIA G8x (GeForce i Quadro). Oczywiście idea użycia procesorów GPU do obliczeń nie związanych z grafiką 3D[1] nie pojawił się znikąd. Już od pewnego czasu możliwe jest modyfikowanie działania kart graficznych. Polega to na przygotowywaniu niewielkich programów (shaderów), które zastępowały niektóre elementy potoku przetwarzania grafiki. W szczególności dotyczy to niestandardowego przetwarzania werteksów i pikseli, co pozwala na stosowanie ciekawych efektów trudnych do otrzymania na poziomie kodu programu korzystającego tylko z OpenGL lub DirectX/XNA. Pierwsze tego typu programy przygotowywane były w asemblerze, ale szybko w NVIDII powstał język Cg (od ang. *C for graphics*), a potem GLSL (OpenGL) i HLSL (DirectX). Wszystkie one dzięki wzorowaniu na języku C umożliwiły programowanie kart graficznych szerszej grupie programistów przyzwyczajonych do języków wyższego poziomu. Wykorzystanie ogromnych możliwości zrównoleglenia obliczeń na GPU nie mogło nie doczekać się prób stosowania tych rozwiązań do zagadnień nie związanych bezpośrednio z grafiką komputerową. I tu właśnie pojawiła się technologia CUDA, która zrewolucjonizowała wykonywanie dowolnych obliczeń na GPU dzięki modelowi programowania, który nie wymaga biegłej znajomości grafiki komputerowej. Aplikacje powstające z wykorzystaniem tej technologii mogą być uruchamiane na którejkolwiek z setek tysięcy maszyn, które są wyposażone w karty firmy NVIDIA. Korzystając z CUDA możemy przyspieszyć niektóre fragmenty programu nawet stukrotnie. Aby uzyskać taki wynik zadanie musi jednak „chętnie" poddawać się zrównolegleniu, przy czym najlepiej by wydajność algorytmu była ograniczona „przez obliczenia", a nie „przez pamięć". Jednak nawet jeżeli tak nie jest, nie stoimy na straconej pozycji. Zwykle udaje się uzyskać dziesięciokrotne przyspieszenie obliczeń, a dzięki zastosowaniu optymalizacji ten współczynnik można jeszcze zwiększyć.

W poniższym dodatku zapoznamy się m.in. z architekturą współczesnych kart graficznych. Wyewoluowała ona do obecnej postaci dzięki ogromnemu zapotrzebowaniu na realistyczne efekty graficzne w grach komputerowych. Specyfika takich obliczeń wymaga identycznych operacji na pokaźnych ilościach danych. Lektura tego rozdziału pozwoli także opanować podstawy interfejsu programistycznego (API) CUDA oraz umożliwi napisanie pierwszych programów korzystających z tej technologii. Uwadze Czytelnika nie powinny również ujść fragmenty opisujące podstawowe techniki optymalizacyjne. Początkowo mogą się wydać one dość trudne, jednak z czasem stają się naturalne. Od czytelnika wymagana jest znajomość podstawowych zasad programowania równoległego, czyli przynajmniej uważna lektura trzech ostatnich rozdziałów książki. Wskazane jest posiadanie karty graficznej firmy NVIDIA wspierającej CUDA, ale nie jest to warunek konieczny. Programy można bowiem uruchamiać w trybie emulacji. Wiąże się to jednak z pewnymi ograniczeniami..

Należy również zastrzec, że nie tylko NVIDIA I ATI zajmują się obecnie zagadnieniami GPGPU. Także inni producenci procesorów coraz uważniej przyglądają się rynkowi związanemu z obliczeniami prowadzonymi na kartach graficznych. Przykładem jest Intel, który niedawno zapowiedział procesor Larrabee. W przeciwieństwie do procesorów ATI i NVIDIA, Larrabee powstał na bazie procesora CPU, a konkretnie popularnego Pentium, które zostało przystosowane do wykonywania obliczeń na potrzeby grafiki 3D. Aktywność w tej dziedzinie przejawia również Microsoft, który zapowiada interfejs programistyczny DirectX 11 Compute (jako składnik DirectX będzie obecny w każdej edycji Windows 7). Podobnie, jak technologie ATI i NVIDII także, ona będzie umożliwiała wykorzystanie procesorów karty graficznej do dowolnych obliczeń. Jednak dzięki temu, że jej

[1] Stosowany jest w tym kontekście skrót GPCPU (od ang. *general-purpose computing on graphics processing units*).

DRAFT

obecność będzie gwarantowana na wszystkich nowych komputerach zawierających DirectX 11, masowo korzystać z niej będą zapewne twórcy gier komputerowych. Alternatywą jest OpenCL (ang. *Open Computing Language*) rozwijane przez konsorcjum Khronos Group, zajmujące się nadzorowaniem biblioteki OpenGL. Jednak dołączenie DirectX 11 Compute do systemu operacyjnego, podobnie jak było w przypadku „pojedynków" Direct3D vs. OpenGL, ADO vs. BDE i Internet Explorer vs. Netscape Navigator może dać tej technologii zdecydowaną przewagę w walce z konkurencją. Należy jednak wyraźnie zaznaczyć, że nie jest to konkurencja dla rozwiązań sprzętowych, a jedynie pomiędzy różnymi metodami ich programowania.

Instalacja

Niezbędne pakiety instalacyjne możemy pobrać z oficjalnej strony projektu[2]. W dziale *Downloads* możemy wybrać interesującą nas platformę sprzętową. Opiszę instalację w 32-bitowym systemie Windows XP[3] najnowszej publicznie dostępnej wersji *CUDA 2.2*. W tym celu musimy pobrać i zainstalować:

1. Najnowszy sterownik firmy NVIDIA dostępny dla naszej karty graficznej zawierający sterownik CUDA (*CUDA Driver*).
   Możemy go pobrać ze strony CUDA lub bezpośrednio z witryny NVIDIA[4]. Przed instalacją nowego sterownika musimy pamiętać o usunięciu obecnego (możemy tego dokonać w aplecie *Dodaj lub usuń programy* z *Panelu sterowania*); Po tym zabiegu, podobnie jak i po instalacji nowego sterownika będziemy zmuszeni do ponownego uruchomienia komputera.
2. Pakiet narzędziowy CUDA (*CUDA Toolkit*).
   W trakcie instalacji można zmienić domyślny folder instalacji pakietu. Ponieważ z jego programów będziemy korzystać z linii komend, warto zadbać, aby ścieżka katalogu nie była zbyt długa. W poniższych przykładach zakładamy, że pakiet zainstalowany jest w podkatalogu *toolkit* katalogu sterowników CUDA. Pakiet narzędziowy zawiera m.in. kompilator NVCC, niezbędne biblioteki, pliki nagłówkowe oraz dokumentację.
3. Ostatni element jest opcjonalny, niemniej jednak wart rozważenia. Moim zdaniem Czytelnik powinien zainstalować także pakiet narzędzi programistycznych CUDA (*CUDA SDK*).
   Instalacji najlepiej dokonać w podfolderze *sdk* katalogu, w którym zainstalowane są sterowniki CUDA. W skład tego zestawu narzędzi wchodzą m.in. liczne przykłady użycia CUDA (w celu weryfikacji poprawności instalacji wskazane jest uruchomienie kilku skompilowanych przykładów znajdujących się w folderze *CUDA\sdk\bin\win32\Release*) oraz dodatkowe narzędzia.

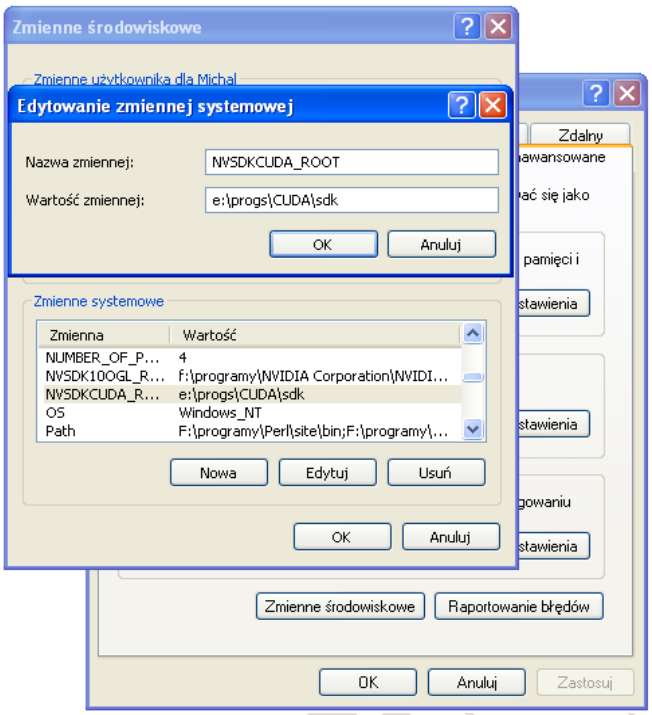
W podkatalogu *CUDA SDK\projects\template* znajduje się szablon projektu przygotowany przez NVIDIA. Niestety wymaga on żebyśmy tworzone na jego podstawie własne projekty umieszczali w folderze z przykładami zawartymi w SDK[5]. Możemy ten problem rozwiązać na kilka sposobów. Jedną z możliwości jest dodanie do ustawień projektu bezwzględnej lokalizacji katalogu *CUDA\sdk\common\inc*. Druga opcja to utworzenie zmiennej środowiskowej zawierającej tę ścieżkę. Skorzystamy z drugiej możliwości. Ponadto będziemy korzystać ze zmodyfikowanego projektu dołączonego do niniejszej książki. W tym celu dodajemy zmienną środowiskową o nazwie `NVSDKCUDA_ROOT` wskazującą na katalog zawierający pakiet *CUDA SDK*. Żeby tego dokonać musimy kliknąć prawym przyciskiem myszy na *Mój komputer*, wybrać *Właściwości*, następnie w zakładce *Zaawansowane* klikamy na *Zmienne środowiskowe*. Klikamy na *Nowa* i wprowadzamy nazwę nowej zmiennej oraz jej wartość (rysunek A.1).

DRAFT

[2]Oficjalna strona NVIDIA CUDA znajduje się pod adresem *http://www.nvidia.com/object/cuda_home.html*
[3]Instalacja na innych platformach nie powinna nastręczać dużo większych trudności. Przygotowane są również pakiety instalacyjne dla większości najpopularniejszych dystrybucji Linuksa.
[4]Należy pamiętać, iż wersja sterownika o wyższym numerze nie zawsze zawiera najnowszy stos CUDA. Dotyczy to szczególnie wersji beta sterowników. Dla Windows XP i CUDA 2.2 zalecany jest w tej chwili sterownik w wersji 185.85. Wersje sterowników na inne platformy maja nieznacznie zmodyfikowaną numerację.
[5]Przeniesienie projektu do innego katalogu powoduje błąd kompilacji, ponieważ szablon szuka potrzebnych bibliotek korzystając ze ścieżek względnych.

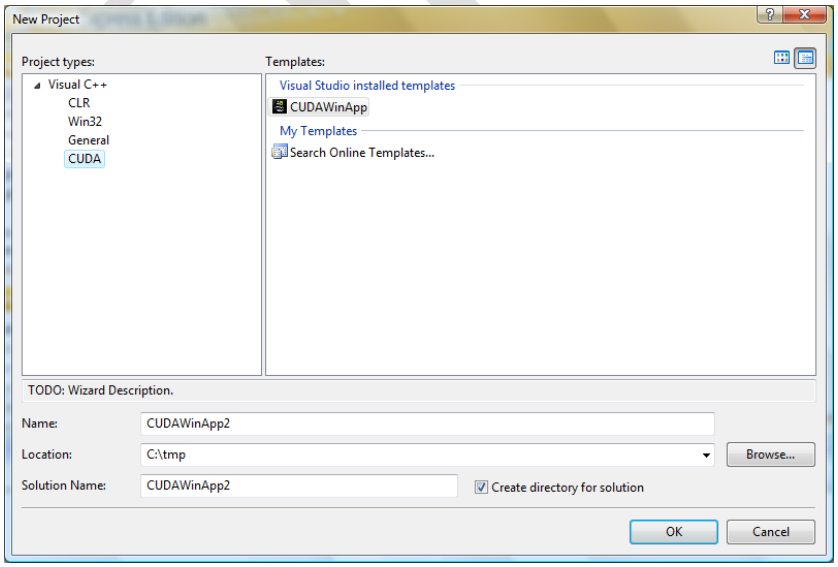
Rysunek A.1.

Ustawienia zmiennych środowiskowych.



Integracja z Visual C++[6]

Integracji narzędzi CUDA ze środowiskiem programistycznym Visual C++ możemy dokonać na dwa sposoby: „ręcznie” lub pobierając z Internetu odpowiedni program, który wykona za nas całą pracę. Bez wątpienia należy wybrać ten drugi sposób, który oszczędzi nam mnóstwo pracy i tylko ten sposób będzie tu opisany. Po zainstalowaniu pakietu narzędzi CUDA oraz środowiska Visual C++ w wersji 2005 lub 2008, pobieramy odpowiedni program *CUDA VS2005 Wizard* ze strony *http://forums.nvidia.com/index.php?showtopic=80572*. Rozpakowujemy go, uruchamiamy i postępujemy zgodnie z wyświetlanymi komunikatami, co tak naprawdę, sprowadza się do zaakceptowania licencji oraz wciśnięcia przycisku *Dalej*. Po zakończeniu działania tego instalatora, w Visual C++ dostępny będzie nowy typ projektu o nazwie *CUDAWinApp* (rysunek A.2.)

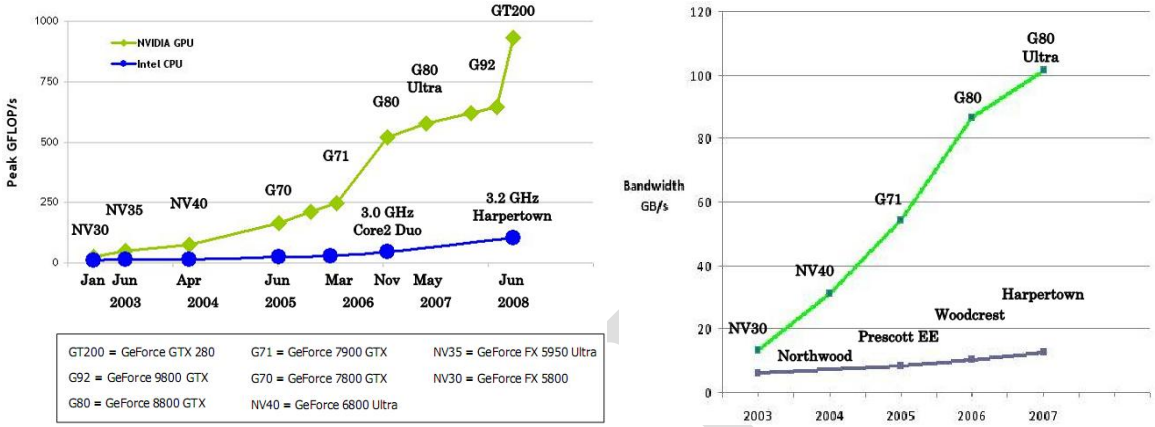


[6] Autorem tego podrozdziału jest Tomasz Dziubak

Rysunek A.2. Nowy typ projektu oraz wzorzec dostępny w Visual C++ po zainstalowaniu aplikacji CUDAWin32App

Architektura

Dzięki ogromnemu zapotrzebowaniu na wysokiej jakości efekty w grach komputerowych rozwój kart graficznych postępował błyskawicznie (rysunek A.3) Obecnie są one wysoce równoległymi, wielowątkowymi, wieloprocesorowymi kartami o wielkiej mocy obliczeniowej i wysokiej przepustowości pamięci.

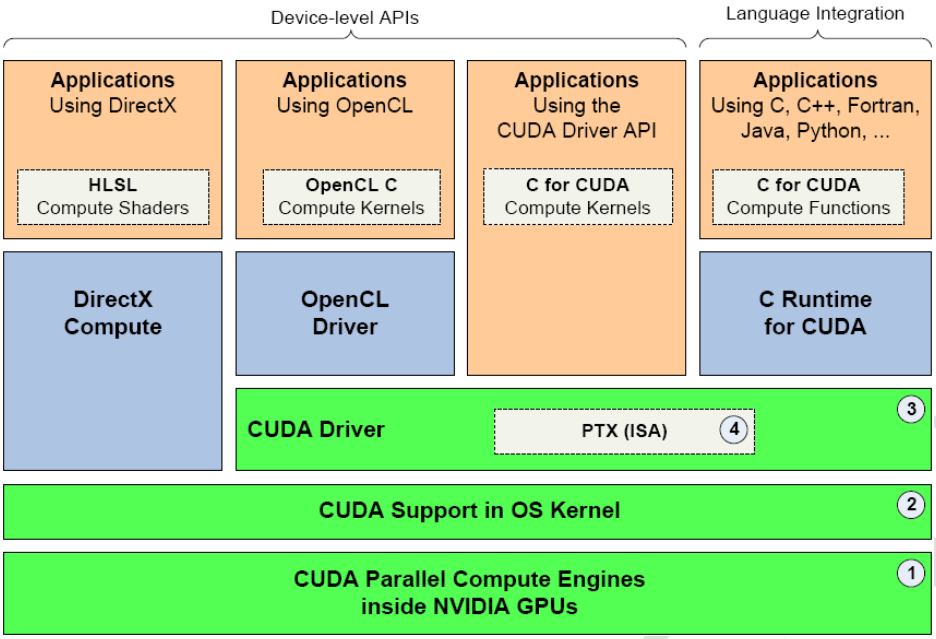


Rysunek A.3. Zmiennopozycyjna moc obliczeniowa oraz przepustowość pomiędzy kartą graficzną a procesorem (ten i kolejne rysunki pochodzą ze strony *nvidia.com*, wykorzystane za zgodą firmy NVIDIA)

Widoczna na wykresach A.3 różnica teoretycznej wydajności między CPU, a GPU wynika z tego, iż GPU specjalizuje się w intensywnych obliczeniowo, wysoce równoległych obliczeniach, czyli dokładnie w takich obliczeniach, jakie są realizowane w ramach potoku przetwarzania grafiki trójwymiarowej. Procesory graficzne składają się z większej ilości tranzystorów przeznaczonych do przetwarzania danych kosztem jednostek odpowiedzialnych za buforowanie danych oraz kontrolę przepływu instrukcji. Ściślej mówiąc karta graficzna jest przeznaczona do wykonywania zadań, które można określić jako zbliżone do architektury SIMD (od ang. *Single Instruction, Multiple Data* – jedna instrukcja, wiele danych) znanej z komputerów wektorowych. Oznacza to, że ten sam program uruchamiany jest równolegle dla różnych danych. Dzięki temu nie jest potrzebny wyszukany system kontroli przepływu instrukcji. Takie podejście pozwala również na rezygnację z rozbudowanych buforów, gdyż opóźnienia w dostępie do pamięci można maskować wykonując w tym czasie operacje w innym wątku.

W listopadzie 2006 roku NVIDIA po raz pierwszy przedstawiła CUDA, czyli architekturę umożliwiającą równoległe wykonywanie dowolnych obliczeń. W jej skład wchodził nowy model programowania równoległego wraz z zestawem instrukcji niskopoziomowych dla karty graficznej. Technologia CUDA obejmuje środowisko programistyczne, które umożliwia wykorzystanie C jako języka wysokiego poziomu. Jak widać na rysunku A.4 inne języki będą również wspierane w niedalekiej przyszłości.

DRAFT



Rysunek A.4. *Architektura CUDA.*

NVIDIA deklaruje wsparcie dla wszystkich języków i interfejsów programistycznych (API), które umożliwiają programistom korzystanie z mocy obliczeniowej kart graficznych. Dlatego oprócz opracowanego przez NVIDIA języka *C for CUDA* można używać także innych modeli programowania. W niniejszym dodatku przedstawione są jednak informacje dotyczące tylko tego najbardziej obecnie dopracowanego i popularnego rozwiązania. Pozostałe przedstawione na rysunku A.4, a mianowicie DirectX Compute oraz OpenCL są dostępne w chwili pisania tego modułu tylko dla grupy wybranych osób, choć jak zaznaczyłem we wstępie potencjalna popularność pierwszego z nich może być ogromna. Różnice miedzy tymi modelami nie są zresztą zbyt duże. Opanowanie modelu NVIDIA *C for CUDA* umożliwi w razie potrzeby migrację do innego środowiska. Najbardziej podobne do naszego środowiska jest OpenCL, z którego specyfikacją można się już zapoznać. Różnice między nim, a przedstawianym w tym dodatku *C for CUDA* są w zasadzie kosmetyczne. Na podstawie prezentacji przedstawionej przez przedstawicieli NVIDIA można mieć nadzieję, że podobnie będzie w przypadku DirectX Compute.

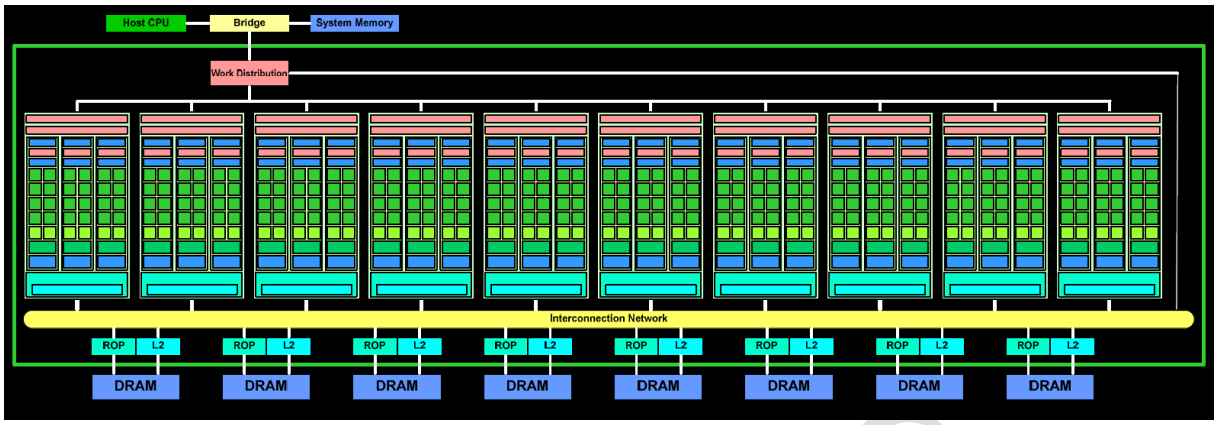
W skład architektury CUDA wchodzi:

1. Równoległy silnik obliczeniowy umieszczony na karcie graficznej.
2. Wsparcie na poziomie jądra systemu dla zainicjowania sprzętu, konfiguracji, itp.
3. Sterownik w trybie użytkownika, który dostarcza niskopoziomowe API dla programistów.
4. Niskopoziomowa maszyna wirtualna PTX[7] (z ang. *parallel thread execution*) oraz zbiór instrukcji asemblera (ISA).

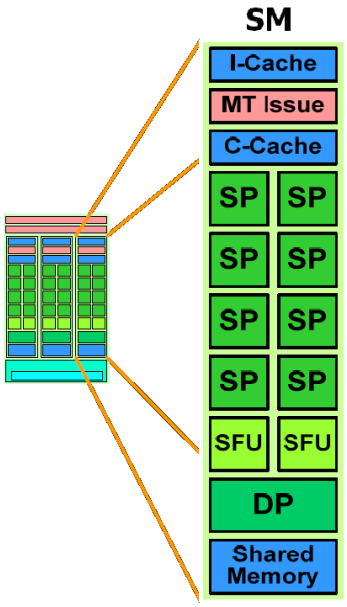
Najnowsza karta graficzna firmy NVIDIA jest zbudowana z multiprocesorów strumieniowych (SM). Karta z rysunku A.5 składa się 30 multiprocesorów. Schematyczną budowę multiprocesora można zobaczyć na rysunku A.6.

[7] Zob. *CUDA\toolkit\doc ptx_isa_1.4.pdf.* ~~*http://www.nvidia.com/object/io_1195170102263.html*~~. Pliki w formacie PTX są plikami zawierającymi kod pośredni (bardzo podobny do instrukcji asemblera), który podczas uruchamiania aplikacji jest tłumaczony na kod maszynynowy odpowiadający posiadanej karcie graficznej. Doświadczeni programiści mogą się pokusić o ich ręczną modyfikacjęmogą być modyfikowane lub tworzone „ręcznie”. Jednak jest to zadania trudne. Na początek ciekawe może być przegladanie tych plików i sprawdzanie, np. które zmienne zostały umieszczone przez kompilator w rejestrach, a które się tam nie zmieściły.

DRAFT



Rysunek A.5. Architektura kart serii 10



Rysunek A.6. Multiprocesor strumieniowy

Każdy SM składa się z:

- *I-Cache* - bufor instrukcji
- *MT Issue* - rozdziela zadania dla SP i SFU (skróty wyjaśnione w następnych punktach)
- *C-Cache* - 8KB bufor stałych, który przyspiesza odczyt z obszaru pamięci stałej (ang. *constant memory*).
- 8 x SP - 8 jednostek obliczeniowych (nazywanych w tradycyjnych kartach graficznych *stream processors*, stąd skrót), które wykonują większość obliczeń pojedynczej precyzji (każdy zawiera własne 32-bitowe rejestry)
- 2 x SFU (ang. *special function units*, jednostki specjalne). Ich zadania to m.in. obliczanie funkcji przestępnych np. trygonometrycznych, wykładniczych i logarytmicznych, czy interpolacja parametrów. Sprzętowo zaprogramowane funkcje są obliczane w 16 cyklach. Pozostałe funkcje potrzebują co najmniej 32 cykli (np. pierwiastek lub funkcja wykładnicza).
- DP – jeden procesor podwójnej precyzji
- SM (ang. *shared memory*) - pamięć współdzielona (16KB). Mają do niej dostęp wszystkie wątki z danej kraty.

Multiprocesor tworzy, zarządza i wykonuje równolegle wątki praktycznie bez opóźnień wywołanych przez planistę. Warto zwrócić uwagę na to, że bariera potrzebuje tylko jednej instrukcji procesora. Te cechy pozwalają znacznie rozdrabniać zadania. Można na przykład przydzielać jeden wątek do każdego piksela na obrazku. W celu zarządzania tak dużą ilością, często wykonujących różne zadania, wątków multiprocesory stosują nową

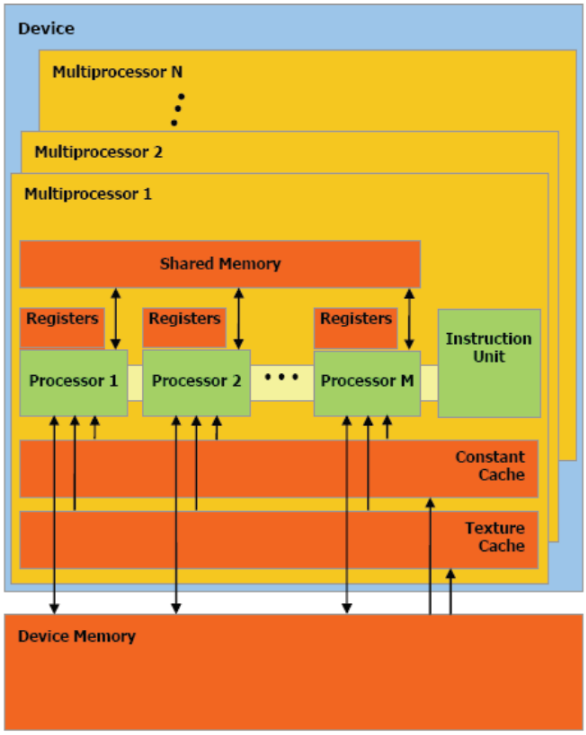
DRAFT

architekturę zwaną SIMT (ang. *single-instruction, multiple thread*). Multiprocesor przydziela każdy wątek do jednego procesora skalarnego. Każdy wątek ma własny zestaw instrukcji oraz stan rejestrów. Multiprocesor zarządza wątkami w grupach liczących po 32 wątki zwanych warpami[8]. Definiuje się również half-warpa, który składa się z pierwszej bądź drugiej połowy warpa. Pojedyncze wątki wewnątrz warpa startują z tymi samymi instrukcjami, jednak każdy wątek może się rozgałęzić i podążać własną ścieżką instrukcji. Wskazane jest, aby wszystkie wątki wewnątrz warpa wykonywały te same instrukcje. Gdy dochodzi do rozgałęzienia, to wszystkie wątki z warpa muszą wykonywać instrukcje z obu rozgałęzień, a część z nich po prostu nie będzie miała żadnego efektu. Każdy wątek ma dostęp do następujących pamięci (rysunek A.7):

- Rejestry procesora
  Każdy multiprocesor posiada 8192 (wersja do 1.1) lub 16384 (w wersjach od 1.2) 32-bitowych rejestrów. Zwykle dostęp do rejestru następuje bez dodatkowych opóźnień, chyba, że próbujemy odczytać wartość z rejestru, do którego dopiero co pisaliśmy. Aby uniknąć wynikających z tego opóźnień należy przydzielić co najmniej 192 aktywne wątki na każdy multiprocesor. W celu uniknięcia konfliktów należy ilość wątków na blok ustawić jako wielokrotność liczby 64.
- Pamięć lokalna
  Dostęp do niej trwa 400-600 cykli. Nie jest buforowana. Odczyt z niej jest zawsze złączony (ang. *coalesced*), bo z definicji pamięć ta(?) jest zdefiniowana dla wątku. Przechowuje duże struktury, duże tablice, tablice, które nie są indeksowane stałymi.
- Pamięć współdzielona
  Tak szybka jak rejestry, gdy nie ma konfliktów.
- Pamięć stała
  Jest to pamięć tylko do odczytu, buforowana. Koszt odczytu rośnie, gdy potrzebna wartość nie znajduje się w buforze. Wówczas dochodzi do odczytu z pamięci karty.
  Gdy cały half-warp czyta ten sam element z bufora, to czas odczytu odpowiada dostępowi do rejestru. Koszt rośnie liniowo wraz z ilością odczytywanych adresów.
- Pamięć tekstur
  Tylko do odczytu, buforowana. Koszt odczytu rośnie, gdy potrzebna wartość nie znajduje się w buforze. Wówczas dochodzi do odczytu z pamięci karty. Bufor jest zoptymalizowany do odczytywania bliskich adresów tzn. osiąga najlepsze rezultaty, gdy wątki z warpa chcą odczytać adresy znajdujące się blisko siebie.
- Pamięć globalna
  Dostęp do niej trwa 400-600 cykli. Nie jest buforowana.

DRAFT

[8] Nazwa wywodzi się z tkactwa, gdzie ang. *warp* oznacza osnowę.



Rysunek A.7. Schemat pamięci

Sygnatura jądra

Skupimy się na modelu używającym NVIDIA *C for CUDA* wraz z wysokopoziomowym API[9]. Niskopoziomowe API wymaga więcej kodu, jest trudniejsze w używaniu i debugowaniu, ale oferuje większą kontrole i umożliwia wykorzystywanie innych języków programowania. W niniejszym module jednak decydujące znaczenie miał fakt, iż API niskopoziomowe nie może korzystać z trybu emulacji, co znacznie utrudnia napisanie poprawnego programu początkującemu użytkownikowi CUDA. Siłą rzeczy uniemożliwia także naukę osobom nieposiadającym najnowszych kart firmy NVIDIA.

Na początek zdefiniujmy pojęcie jądra (ang. *kernel*), czyli funkcji, która będzie uruchamiana równolegle na karcie graficznej. Może być wywoływany tylko przez wątek działający na CPU. W kodzie można go rozpoznać po słowie kluczowym **__global__** poprzedzającym jego deklaracje. Wywołanie jądra z poziomu programu wygląda podobnie jak wywołanie zwykłej funkcji C, z tym że uzupełnione jest o dodatkowe parametry:

```
kernel<<<dim3 grid, dim3 block, size_t Ns, cudaStream_t S>>>(parameter s)
```

Parametry umieszczane wewnątrz nawiasów <<< >>> są charakterystyczne dla *C for CUDA* i mają następujące znaczenie:

- `grid` - ustala wymiar i rozmiar kraty. Iloczyn `grid.x*grid.y` odpowiada sumarycznej liczbie bloków, które zamierzamy uruchomić. Ponieważ parametr `grid` jest typu `dim3`, a więc zawiera trzy wymiary, to `grid.z` jest domyślnie ustawiany na 1.
- `block` - ustala wymiar i rozmiar każdego bloku. Iloczyn współrzędnych tego wektora odpowiada liczbie wątków wewnątrz każdej kraty.
- `Ns` - jest opcjonalnym argumentem odpowiadającym za ilość dynamicznie przydzielonej pamięci współdzielonej dla każdego z bloków. Domyślnie wynosi 0.
- `S` - również jest opcjonalnym argumentem o domyślnej wartości równej 0. Zrozumienie znaczenia tego parametru wymaga znajomości pewnych faktów nie omówionych dotychczas; tymczasem wspomnę tylko, że parametr ten dotyczy strumienia.

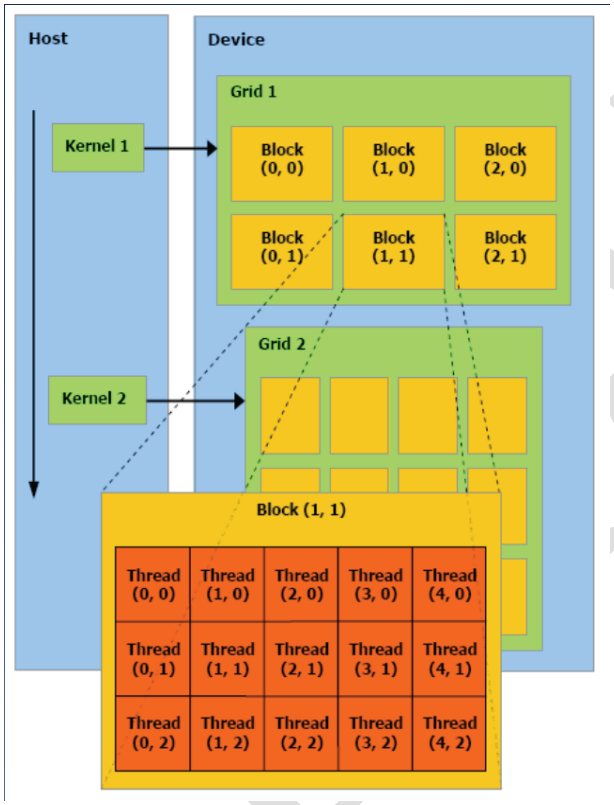
[9] w oryginale nazywa sie je *runtime API*

DRAFT

Poza tymi specjalnymi parametrami do jądra przekazywane są również „zwykły” argument typu `parameters`. Jest to ciąg parametrów, którego długość jest ograniczona do 512 bajtów.
Każdy wątek uruchomiający jądro (*kernel*) posiada unikalny identyfikator. Dostęp do tego numeru z poziomu jądra można uzyskać dzięki wbudowanej zmiennej `threadIdx`, która jest typu `dim3` dzięki czemu możemy bez trudu ustalić pozycje wątku w kracie.

Funkcje jądra muszą respektować kilka ograniczeń: zawsze zwracają tym `void`, mają dostęp **tylko** do pamięci karty graficznej, muszą mieć stałą ilość argumentów, nie mogą posiadać wywołań rekurencyjnych i nie posiadają zmiennych statycznych. Ponadto należy pamiętać, że wywołanie jądra jest asynchroniczne, co oznacza, że program, który go wywołuje natychmiast odzyskuje kontrolę (prawdopodobnie przed zakończeniem działania funkcji jądra).

Aby lepiej zrozumieć numeracje wątków, bloków i krat przyjrzyjmy się bliżej rysunkowi A.8. Widać na nim, że jądro o nazwie *Kernel 1* został uruchomiony dla kraty o rozmiarze 3 x 2, w której każdy blok ma rozmiar równy 5 x 3 x 1. Na czerwonym tle widzimy identyfikator wątku, który możemy odczytać za pomocą zmiennej `threadIdx`, a na pomarańczowym wartość zmiennej `blockIdx`, czyli identyfikator obecnego bloku. Pośród zmiennych, do których wątek ma dostęp z poziomu jądra ważne są jeszcze `blockDim` i `gridDim`, które przechowują odpowiednio wymiar bloku i kraty. Kolejna ważna zmienna to `warpSize`, jednak jest ona używana zdecydowanie rzadziej niż wyżej wymienione zmienne. Jej znaczenie zostanie wyjaśnione w rozdziale 5.



Rysunek A.8. Rozmieszczenie wątków na sieci

Uruchamianie jądra i korzystanie z pamięci karty graficznej

Pierwsze zadanie polega na uruchomieniu bardzo prostego programu korzystającego z architektury CUDA. W tym celu należy otworzyć dostępny w materiałach dołączonych do książki projekt startowy dla tego zadania. Zgodnie z zapowiedzią będziemy korzystać z wysokopoziomowego API zawartego w bibliotekach CUDA.

Pierwszą czynnością jaką musimy wykonać jest zainicjowanie karty graficznej. Na początku funkcji *main* (plik *zad1.cu*) musimy zatem dopisać[10]:

Listing A.1. Inicjacja karty graficznej

```
14 // wybiera GPU podane z linii komend,
15 // badz te o najwiekszej ilosci Gflops / s
16 if( cutCheckCmdLineFlag( argc, (const char **)argv , "device") )
17       cutilDeviceInit(argc, argv);
18 else
19       cudaSetDevice( cutGetMaxGflopsDeviceId() ); }
```

Linia numer 16 korzysta z funkcji pomocniczej zawartej w pakiecie programistycznym[11] dostarczonym wraz z CUDA i pozwala z linii komend wybrać urządzenie na którym chcemy dokonywać obliczeń. Urządzenie wskazywane jest za pomocą argumentu linii komend, a gdy uruchomimy nasz program bez parametrów, to domyślnie zostanie wybrana ta karta, której estymator ilości GFLOPS (ang. *FLoating point Operations Per Second* - liczba operacji zmiennoprzecinkowych na sekundę) jest największy[12].

Ponieważ dopiero zapoznajemy się z technologią CUDA, użyta w pierwszym przykładzie ilość wątków jest niewielka (listing A.2). W programach przygotowywanych do prawdziwych obliczeń ilość wątków nigdy nie powinna być tak mała.

Listing A.2. Parametry zależne od ilości wątków

```
21 int num threads = 16 ; // podzielne przez 4
22 int mem size = sizeof(int) * num threads;
```

Następnie definiujemy dwa wskaźniki o nazwach `h_data` i `d_data` (listing A.3). Pierwszy (z prefiksem `h_`) będzie służył do przechowywania wyników programu w pamięci komputera, natomiast drugi (ten z prefiksem `d_`) będzie wykorzystywany przez kartę graficzną. Dla `h_data` pamięć przydzielamy klasycznie, identycznie jak to miało miejsce w języku C. Natomiast w przypadku `d_data` skorzystamy z jednej z funkcji służących do alokacji danych na karcie graficznej . Jej sygnatura jest następująca:

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

W `*devPtr` jest zwracany wskaźnik do zmiennej `count` przechowującej ilości bajtów jednolitego bloku zarezerwowanej pamięci. W razie sukcesu funkcja zwraca wartość równą stałej `cudaSuccess`. Należy zwrócić uwagę na to, że funkcja `cudaMalloc` (jak i wiele innych) może zwracać błędy pochodzące z uprzednio wywoływanych funkcji asynchronicznych. Błędy funkcji wykonywanych asynchroniczne nie mogą być zwracane przez nie same – główny wątek nie czeka na ich zakończenie i tym samym nie sczytuje zwracanych przez nie wartości. W zamian program przechowuje pewną zmienna zawierającą kod ostatniego błedu. Jest ona inicjowana wartością `cudaSuccess` i jest nadpisywana za każdym razem, gdy pojawi się bład. Aby go odczytać, a jednocześnie przywrócić jej domyślną wartość, należy użyć funkcji `cudaGetLastError`. Jeżeli chcemy być pewni, że działanie funkcji asynchronicznej zakończyło się sukcesem, konieczne jest ustawienie za nią bariery za pomocą funkcji `cudaThreadSynchronize` i sprawdzenie jaką wartość zwróci ta funkcja. Gdy tego nie zrobimy, to błąd zostanie zgłoszony przez pierwszą funkcję, która wywołuje synchronizację. Należy unikać wstawiania barier po każdym wywołaniu asynchronicznym, gdyż może to znacznie obniżyć wydajność całego programu.

Listing A.3. Zmienne przechowywane w pamięci komputera i w pamięci karty graficznej

```
24 int* h_data = (int*) malloc(mem_size);
25 int* d_data ;
```

---

[10] Dla ułatwienia orientacji w istniejącym już kodzie projektu startowego w listingach umieściłem numery linii

[11] Korzystanie z funkcji pomocniczych zaczynających się na `cut..` lub `cutil..` jest generalnie niewskazane, gdyż nie ma gwarancji ich obecności w różnych edycjach CUDA. Pełnią one tylko funkcje pomocniczą. Później zaprezentuję co tak naprawdę robią i kiedy można z nich zrezygnować. NVIDIA udostępnia ich kod źródłowy, ale bez dokumentacji.

[12] Wartość GFLOPS oblicza się mnożąc ilość multiprocesorów, ilość procesorów na multiprocesorze z częstotliwością taktowania karty. Otrzymany wynik należy pomnożyć przez dwa, gdy mamy do czynienia z kartami w wersji 1.0 lub 1.1, gdyż mogą one wykonywać operacje FMAD (mnożenie i dodawanie: a*b+c) w jednym cyklu. Przez pewien czas twierdzono, że procesor skalarny potrafi wykonać w jednej instrukcji oprócz FMAD również operacje MUL, jednak w wyniku do końca niewyjaśnionego błędu okazało się to niemożliwe. Dopiero karty z serii GT200 (model obliczeniowy 1.3) są w stanie wykonywać takie operacje. Z tego powodu ich teoretyczną moc obliczeniową należy przemnażać nie przez 2, a przez 3.

DRAFT

```
26 cutilSafeCall(cudaMalloc( (void**) &d_data, mem_size));
```

Teraz dochodzimy do najciekawszej części tego programu, czyli uruchomienia jądra. Włączymy go w blokach zawierających po 4 wątki. Jako parametr otrzyma wskaźnik na pamięć zarezerwowaną w zmiennej `d_data` (przechowywanej w pamięci karty graficznej). Pokazuje to listing A.4.

Listing A.4. Uruchomienie jądra

```
28 // ustawia rozmiar kraty i bloku
29 dim3 grid ( num_threads/4, 1, 1);
30 dim3 threads(4, 1, 1);
31
32 // uruchom kernel
33 zad1Kernel<<< grid, threads >>>( d_data );
```

Po wywołaniu funkcji jądra sterowanie natychmiast wraca do głównego programu. Aby poczekać na zakończenie działania jądra musimy uśpić program aż do odebrania odpowiedniego komunikatu. Załatwi to za nas funkcja pomocnicza `cutilCheckMsg` (listing A.5). Funkcja ta jest w zasadzie opakowaniem do innej funkcji dostępnej w interfejsie programistycznym CUDA, a mianowicie do `cudaThreadSynchronize`[13].

Po wykonaniu programu na karcie graficznej i upewnieniu się, że nie powstały żadne błędy, musimy jeszcze skopiować wyniki z powrotem do pamięci komputera. Do tego celu skorzystamy z funkcji `cudaMemcpy`. Pokazuje to również listing A.5.

Listing A.5. Synchronizacja wątków CUDA i kopiowanie wyników do pamięci komputera

```
35 // sprawdz , czy kernel zakonczyl sie sukcesem
36 cutilCheckMsg("Kernel execution failed”);
37
38 // kopiuje wynik z GPU do pamieci komputera
39 cutilSafeCall(cudaMemcpy(h_data, d_data, mem_size,
40                          cudaMemcpyDeviceToHost) );
```

Funkcja `cudaMemcpy` ma następującą składnię:

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

Sądzę, że znaczenia parametrów `dst`, `src` i `count` nie trzeba wyjaśniać. Ciekawy jest natomiast parametr `kind`. Może on przyjmować jedną z trzech wartości:

- `cudaMemcpyHostToDevice` - kopiowanie z pamięci komputera do pamięci GPU.
- `cudaMemcpyDeviceToHost` - kopiowanie z pamięci GPU do pamięci komputera.
- `cudaMemcpyDeviceToDevice` - kopiowanie między dwoma obszarami pamięci na karcie graficznej.

Pozostaje już tylko wyświetlić wynik obliczeń i zwolnić zarezerwowaną pamięć. Pokazuje to listing A.6.Zwolniamy w nim także zasoby zajmowane przez nasz wątek po zainicjowaniu CUDA. Następnie opróżnimy wszystkie bufory.

Listing A.6. Zakończenie funkcji main głównego programu

[13] Jest to dobra okazja, żeby zwrócić uwagę Czytelnika na zachowanie funkcji pomocniczych. Zdarza się, że działają one trochę inaczej w trybie *release*, a inaczej w trybie *debug*. Powyższa funkcja w trybie *release* wcale nie wywołuje `cudaThreadSynchronize`.

DRAFT

```
42 for(int i = 0; i < num_threads; ++i)
43 printf("%d ", h_data[i]);
44
45 //zwalnianie pamieci
46 free( h_data );
47 cutilSafeCall(cudaFree(d_data));
48
49 cudaThreadExit();
50 cutilExit(argc, argv);
```

Możemy teraz spróbować uruchomić program. Jeśli nie popełniliśmy żadnego błędu to w wyniku otrzymamy 16 przypadkowych liczb całkowitych. Jest tak, ponieważ jądro jeszcze nic nie robi (ciało funkcji `zad1Kernel` z pliku *zad1_kernel.cu* jest puste). Zanim to poprawimy wyczyśćmy, obszar pamięci w karcie graficznej wskazywany przez `d_data`. Skorzystamy do tego z funkcji `cudaMemset`, która ustawi każdy bajt wskazanego obszaru pamięci karty graficznej na zadaną wartość:

```
27 cutilSafeCall( cudaMemset( d_data, 0, mem_size ));
```

Dzięki temu po uruchomieniu programu powinniśmy dostać zbiór szesnastu zer.

Programowanie jądra

Przejdźmy teraz do pliku *zad1_kernel.cu* zawierającego funkcję jądra uruchamianą na karci graficznej. Zaczniemy od wyznaczenia numeru wątku korzystając ze zmiennych `blockIdx` i `threadIdx`. Przy użytych przez nas niskich wymiarach bloków i kraty, zadanie to jest stosunkowo łatwe . Następnie do wejściowej tablicy o nowo obliczonym indeksie przypiszmy jakąś wartość, na przykład 17:

Listing A.7. Wypełnianie pamięci wartościami 17

```
5 const unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
6
7 d_data[ tid ] = 17;
```

Jest jasne, że gdy teraz uruchomimy program na ekranie pojawi się ciąg szesnastu liczb 17.

Jeżeli w tablicy zapiszemy wartość zmiennych `tid` (polecenie `d_data[ tid ] = tid;`), `threadIdx.x` i `blockIdx.x` uzyskamy wyniki równe odpowiednio:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

Przykład programu: rozmycie wektora

W tym zadaniu stworzymy nowy program, którego zadaniem będzie rozmycie wektora (ang. *box blur*). Oznacza to, że do każdego elementu wzdłuż tego wektora zapiszemy średnią wartość jego i określonej liczby jego sąsiadów. Elementy znajdujące się na krawędzi pozostawimy bez zmian.

Najpierw napiszemy funkcję kontrolną, która będzie wykonywać to zadanie korzystając jedynie z procesora głównego (CPU) i której potem użyjemy do sprawdzenia poprawności wyników obliczeń przeprowadzonych na karcie graficznej. W tym celu zaimplementujemy funkcję `computeGold` zgodnie ze wzorem z listinguA.8 (plik *zad2.cu* z drugiego projektu startowego).

Listing A.8. Funkcja wykonująca rozmycie wektora na CPU

```
12 float* computeGold(float* h_input, int count) {
13    float* res;
```

DRAFT

```
14    res = (float*)malloc(sizeof(float) *count);
15
16    for(int i = 0; i < RADIUS; i++) {
17       res[i] = h_input[i];
18       res[count-i-1] = h_input[count-i-1];
19    }
20    for(int i = RADIUS; i < count - RADIUS; i++){
21       res[i] = 0.0f;
22        for(int j = -RADIUS; j <= RADIUS; j++)
23          res[i] += h_input[i+j];
24       res[i] /= (2 *  RADIUS + 1.f);
25    }
26
27    return res;
28 }
```

Stałą RADIUS definiujemy w pliku *zad2_kernel.cu*. Odpowiada ona za ilość sąsiadów, których uwzględniamy przy uśrednianiu. Definiujemy tam też ilość wątków w bloku. Na razie ustawimy dość małą wartość, by łatwiej znaleźć ewentualne błędy w programie.

```
5 #define RADIUS 2
6 #define BLOCK_SIZE 16
```

Tak samo, jak w poprzednim przykładzie zainicjujemy GPU w funkcji `main` (plik *zad2.cu*) wstawiając do niej polecenia z listingu A.9.

Listing A.9. Inicjacja karty graficznej na potrzeby obliczeń CUDA

```
31 // wybiera GPU podane z linii komend ,
32 // badz te o najwiekszej ilosci GFLOPS.
33 if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") )
34    cutilDeviceInit(argc, argv);
35 else
36    cudaSetDevice( cutGetMaxGflopsDeviceId() );
```

Następnie deklarujemy zmienną `inputSize`, która będzie odpowiadać za wielkość tablicy wejściowej oraz zmienną pomocniczą `mem_size` przechowującą rozmiar tej tablicy w bajtach.

```
38 int inputSize = 64 ;
39 int mem_size = sizeof(float ) * inputSize;
```

Teraz tworzymy tablicę `h_input` i wypełnimy ją losową zawartością:
Listing A.10. Alokacja pamięci

```
41 float*  h_input = 0 ;
42 h_input = (float*)malloc(mem_size);
43 if(0 == h_input ) {
44    printf("Za malo pamieci operacyjnej.\ n");
45    return 1;
46 }
47 srand(time(NULL));
48 for(int i = 0; i < inputSize; ++i)
49    h_input[i] = (float)rand();
```

DRAFT

Analogicznie, jak poprzednio przygotowujemy też dwie tablice ulokowane w pamięci karty graficznej oraz skopiujemy do jednej z nich wyznaczone w poprzednim kroku dane wejściowe (listing A.11). Typ kopiowania w funkcji `cudaMemcpy` musimy ustalić w taki sposób, aby kopiować z pamięci komputera do pamięci karty graficznej, czyli ostatni jej parametru ustawić na `cudaMemcpyHostToDevice`.

Listing A.11. Alokowanie pamięci karty graficznej i kopiowanie do niej danych wejściowych

```
51 float* d_input ;
52 float* d_output ;
53 cutilSafeCall( cudaMalloc( (void**) &d_input, mem_size));
54 cutilSafeCall( cudaMalloc( (void**) &d_output, mem_size));
55
56 cutilSafeCall( cudaMemcpy( d_input, h_input, mem_size,
57              cudaMemcpyHostToDevice));
```

Ustalimy teraz wielkość kraty i ilość wątków jakich będziemy używać. Zakładamy, że rozmiar tablicy jest podzielny przez rozmiar bloku. Nie jest jednak trudno zrezygnować z tego założenia.

Listing A.12. Ustawianie parametrów kraty multiprocesorów

```
59 // ustawia rozmiar kraty i bloku
60 dim3 grid(inputSize / BLOCK_SIZE, 1, 1);
61 dim3 threads(BLOCK_SIZE, 1, 1 );
```

Wreszcie możemy wywołać jądro (*kernel*), które niebawem napiszemy. Jako, że jest to funkcja asynchroniczna, to po jej wywołaniu umieszczamy polecenia wymuszające synchronizację wątków. Następnie sprawdzamy czy nie wystąpiły jakieś błędy w działaniu jądra. Obie czynności pokazuje listing A.13. Najczęstszym błędem w początkowej fazie pisania programów z wykorzystaniem architektury CUDA jest przekroczenie dopuszczalnej ilości wątków w bloku lub przekroczenie maksymalnej wielkości któregoś z wymiarów kraty. Maksymalna ilość wątków w bloku dla obecnych kart graficznych wynosi 512. Ograniczenie dla konkretnej karty można sprawdzić w dokumentacji bądź skorzystać z funkcji `cudaGetDeviceProperties`, która pozwala sprawdzić także inne parametry karty graficznej. Natomiast maksymalna wielkość każdego wymiaru kraty wynosi obecnie 65535 bloków. Maksymalna wielkość bloku może ulec zmniejszeniu, gdy nasz program potrzebuje wielu rejestrów lub zajmuje sporą ilość pamięci współdzielonej. Aby zdobyć te dane najłatwiej jest skompilować nasz program za pomocą kompilatora `nvcc` z opcjami `-Xptxas -v`.

Listing A.13. Wywoływanie jądra i odczytywanie wyniku jego wykonania

```
63 // uruchom kernel
64 zad2Kernel<<<  grid,threads >>>( d_input, d_output);
65
66 cudaThreadSynchronize();
67 // sprawdz, czy kernel zakonczyl sie sukcesem
68 cutilCheckMsg("Kernel execution failed");
```

Zauważmy, że tym razem do synchronizacji zamiast z funkcji pomocniczej korzystamy bezpośrednio z funkcji CUDA API `cudaThreadSynchronize`.

Podobnie jak w poprzednim programie zarezerwujemy pamięć komputera oraz skopiujemy do niej wynik z pamięci karty graficznej:

Listing A.14. Kopiowanie wyniku do pamięci głównej komputera

```
70 float* h_output = (float*)malloc(mem_size);
71
72 // kopiuje wynik z GPU do pamieci komputera
73 cutilSafeCall ( cudaMemcpy( h_output, d_output, mem_size,
74                              cudaMemcpyDeviceToHost ) ) ;
```

DRAFT

Sprawdzimy jeszcze czy wyniki obliczeń przeprowadzonych na karcie graficznej pokrywają się z tymi otrzymanymi tradycyjną metodą (listing A.15). Na ekranie pokazywane są tylko wartości, dla których wyniki uzyskane obydwoma metodami nie pokrywają się ze sobą.
Listing A.15. Porównanie obliczeń na CPU i GPU

```
76 float* goldRes = computeGold(h_input, inputSize);
77 for(int i = 0; i < inputSize; ++i)
78    if(abs(h_output[i] - goldRes[i] ) >  0.01f)
79       printf("%d = %f != %f \n", i, goldRes[i], h_output[i]);
```

Pozostaje już tylko posprzątać po sobie:
Listing A.16. Zwalnianie pamięci i kończenie wątku

```
81 // zwalnianie pamieci
82 free(h_input);
83 free(h_output);
84 free(goldRes);
85 cutilSafeCall(cudaFree(d_input));
86 cutilSafeCall(cudaFree(d_output));
87
88 cudaThreadExit();
89 cutilExit(argc, argv);
```

W ten sposób zakończyliśmy implementację tej części programu, która wykonywana jest na CPU. Przejdźmy teraz do najważniejszej części programu, a mianowicie do funkcji jądra, która wykonywana będzie na GPU. Ponownie pierwszą czynnością w funkcji jądra jest wyznaczenie indeksu wątku (linia 9 w listingu A.17). Później sprawdzamy, czy nie jesteśmy na krańcu tablicy. Jeżeli tak, to kopiujemy do tablicy wyjściowej odpowiadający mu element tablicy wejściowej i kończymy ten wątek. Wątki nie znajdujące się na krańcach tablicy obliczają sumę elementów oddalonych o co najwyżej wartość przechowywaną w stałej `RADIUS` od ich pozycji, a następnie dzielą ją przez ilość elementów, które sumowały (listing A.17).
Listing A.17. Zasadnicze obliczenia wykonywane przez jądro

```
9 const unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
10
11 if( tid < RADIUS | | tid >= BLOCK_SIZE * gridDim.x - RADIUS) {
12    d_output[tid] = d_input[tid];
13    return;
14 }
15
16 float res = 0;
17 for(int i = -RADIUS; i <= RADIUS; i++)
18    res += d_input[tid + i];
19
20 d_output[tid] = res / (2 * RADIUS + 1.f);
```

Możemy teraz uruchomić program. Jeśli nie popełniliśmy żadnych błędów i program działa prawidłowo, tj. wynik uzyskany za pomocą jądra i funkcji obliczanej na CPU jest identyczny, to na ekranie nie pojawi się żaden napis. Wyniki w dwóch ostatnich wierszach mogą się jednak nieznacznie różnić. Wynika to z precyzji obliczeń na liczbach rzeczywistych.

DRAFT

Jeśli pusty ekran nie wydaje się nam satysfakcjonujący i chcemy zobaczyć jakieś wyniki, możemy zmodyfikować funkcje `main` w taki sposób, aby prezentowany był obliczony wektor W tym celu wystarczy usunąć instrukcję `if` z linii 78 listingu A.15.

Pomiary czasu obliczeń na GPU

Dodamy teraz zegary mierzące ile czasu zajmują poszczególne części programu. Umożliwi to zobaczenie efektów optymalizacji, którą spróbujemy za moment wykonać, a także pozwoli na porównanie czasu działania klasycznej wersji programu na CPU z naszą implementacją na GPU. Zaczniemy od modyfikacji funkcji `computeGold` wykonywanej na CPU (plik *zad2.cu*)

Listing A.18. Makra wstawiane do funkcji computeGold z pliku zad2.cu.

```
16 unsigned int cpuTimer = 0;
17 CUT_SAFE_CALL(cutCreateTimer(&cpuTimer));
18 CUT_SAFE_CALL(cutStartTimer(cpuTimer));

...

29 CUT_SAFE_CALL(cutStopTimer(cpuTimer));
30 printf("CPU processing time: %f (ms)\n",
31       cutGetTimerValue(cpuTimer));
32 CUT_SAFE_CALL(cutDeleteTimer(cpuTimer));
```

Użyte w tym fragmencie kodu makro `CUT_SAFE_CALL` wywołuje podaną jako argument funkcję oraz sprawdza, czy zakończyła się sukcesem. Podobne fragmenty kodu wstawimy w dalszej części naszego programu. Zmierzymy w ten sposób czas kopiowania danych do z pamięci głównej do pamięci karty graficznej:

Listing A.19. Kolejne pomiary

```
62 unsigned int gpuMemcpyHtDTimer = 0 ;
63 CUT_SAFE_CALL( cutCreateTimer(&gpuMemcpyHtDTimer));
64 CUT_SAFE_CALL(cutStartTimer(gpuMemcpyHtDTimer));
65 cutilSafeCall(cudaMemcpy( d_input, h_input, mem_size ,
66                          cudaMemcpyHostToDevice));
67 CUT_SAFE_CALL(cutStopTimer(gpuMemcpyHtDTimer));
68 printf("GPU: Memcpy z pamieci komputera do GPU: %f (ms)\n" ,
69 cutGetTimerValue(gpuMemcpyHtDTimer));
```

W analogiczny sposób mierzymy czas działania jądra:

Listing A.20. Za krótki czas obliczeń jądra możemy zapłacić długim czasem kopiowania danych

```
75 unsigned int gpuProcessingTimer = 0;
76 CUT_SAFE_CALL(cutCreateTimer(&gpuProcessingTimer));
77 CUT_SAFE_CALL(cutStartTimer(gpuProcessingTimer));
78 // uruchom kernel
83 cutilCheckMsg("Kernel execution failed");
84
85 CUT_SAFE_CALL(cutStopTimer(gpuProcessingTimer));
86 printf("GPU: czas obliczen: %f (ms)\n" ,
87     cutGetTimerValue(gpuProcessingTimer));
```

DRAFT

I wreszcie czas kopiowania wyników z pamięci karty graficznej do pamięci głównej komputera:

Listing A.21. Pomiar czasu kopiowania wyników

```
91 unsigned int gpuMemcpyDtHTimer = 0;
92 CUT_SAFE_CALL(cutCreateTimer(&gpuMemcpyDtHTimer));
93 CUT_SAFE_CALL(cutStartTimer(gpuMemcpyDtHTimer));
94 // kopiuje wynik z GPU do pamieci komputera
95 cutilSafeCall(cudaMemcpy( h_output, d_output, mem_size,
96                            cudaMemcpyDeviceToHost));
97
98 CUT_SAFE_CALL(cutStopTimer(gpuMemcpyDtHTimer));
99 printf("GPU: Memcpy z GPU do pamieci komputera: %f (ms)\n" ,
100             cutGetTimerValue(gpuMemcpyDtHTimer ));
101 printf("GPU: czas calkowity: %f (ms)\n",
102        cutGetTimerValue(gpuProcessingTimer )
103        + cutGetTimerValue(gpuMemcpyDtHTimer )
104        + cutGetTimerValue(gpuMemcpyHtDTimer ));
```

Nie możemy zapomnieć o zwolnieniu zasobów zegarów:

Listing A.22. Zwalnianie zasobów zajmowanych przez zegary

```
109 printf("%d = %f != %f \n", i, goldRes[i], h_output[i]);
110
111 CUT_SAFE_CALL(cutDeleteTimer(gpuProcessingTimer));
112 CUT_SAFE_CALL(cutDeleteTimer(gpuMemcpyHtDTimer));
113 CUT_SAFE_CALL(cutDeleteTimer(gpuMemcpyDtHTimer));
114 // zwalnianie pamieci
```

W ostatecznej wersji programy powiększmy tablicę (zmienną `inputSize` zainicjujmy wartością `16*1024*1024`), ustawmy normalny rozmiar bloku (dyrektywa `#define BLOCK_SIZE 512`) oraz wyłączmy drukowanie poprawnych elementów na ekranie przywracając warunek z linii 78 na listingu A.15.

Uruchamiając teraz program otrzymamy wynik zbliżony do[14]:

```
GPU: Memcpy z pamieci komputera do GPU: 28.025410 (ms)
GPU: czas obliczen: 50.064262 (ms)
GPU: Memcpy z GPU do pamieci komputera: 39.373405 (ms)
GPU: czas calkowity: 117.463078 (ms)
CPU: czas obliczen: 555.848328 (ms)
```

Optymalizacje

Widać, że implementacja naszego programu mimo, że nie zawiera żadnych optymalizacji jest ok. 5 razy szybsza od wersji oryginalnej. Przy czym same obliczenia korzystające z GPU są aż 11 razy szybsze od analogicznych przeprowadzonych tylko na CPU. Sporo czasu tracimy jednak na kopiowaniu danych do i z karty graficznej.

[14] Wszystkie wyniki obliczeń pochodzą z karty GeForce 8800GT, jeśli nie zaznaczono inaczej.

DRAFT

Wydajność transferów poprawimy później. Na razie skupimy się na samym jądrze. We wprowadzeniu zwróciłem uwagę, że pamięć globalna jest bardzo wolna w porównaniu z szybką pamięcią współdzieloną. Spróbujemy zatem skorzystać z tego faktu. Można zauważyć, że każdy blok używa tylko `BLOCK_SIZE+2*RADIUS` elementów wektora wejściowego. Możemy skopiować te elementy do pamięci współdzielonej karty graficznej i potem korzystać tylko z niej. Zadeklarujemy w tym celu nową zmienna `sBuf` (listing A.23). Deklarując ją użyliśmy modyfikatora `__shared__` oznaczającego, że zmienna ta będzie przechowywana w pamięci współdzielonej. Musimy też określić sposób konwersji indeksu tablicy wejściowej na indeks odpowiadającego mu elementu tablicy znajdującej się w pamięci współdzielonej. Zmiennej `tid` będzie odpowiadała zmienna `localId`. Po tych zmianach początek naszego jądra wygląda następująco:

Listing A.23. Kopiowanie wektora do pamięci współdzielonej (funkcja jądra z pliku zad2_kernel.cu)

```
9 __shared__ float sBuf[BLOCK SIZE + 2_RADIUS];
10
11 const unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
12 const unsigned int localId = threadIdx.x + RADIUS;
```

Możemy zacząć kopiowanie. Każdy wątek z bloku przekopiuje odpowiadający mu element do tablicy współdzielonej. Dodatkowo wątki o najniższym indeksie w bloku skopiują elementy brzegowe do naszego bufora. Należy pamiętać, że wątki w bloku wykonują się równolegle i nie ma pewności, że wszystkie w tym samym momencie skończą kopiowania. Mogłoby dojść do sytuacji, że czytamy z banku pamięci współdzielonej do którego inny wątek nie zdążył jeszcze wpisać poprawnej wartości. Dlatego po skopiowaniu fragmentów wektora należy ustawić barierę korzystając z funkcji `__syncthreads`. Wymusi ona zatrzymanie wątków do momentu, w którym wszystkie wątki osiągną to polecenie kodu.. Należy pamiętać, aby unikać umieszczania tego typu barier w blokach warunkowych, gdyż w ten sposób można bardzo łatwo zakleszczyć program.

Listing A.24. Kopiowanie i synchronizacja

```
14 sBuf[localId] = d_input[tid];
15 if(threadIdx.x < RADIUS) {
16    sBuf[localId - RADIUS] = d_input[tid - RADIUS];
17    sBuf[localId + BLOCK_SIZE] = d_input[tid + BLOCK_SIZE];
18 }
19 __syncthreads( ) ;
20
21 if( tid < RADIUS || tid >= BLOCK_SIZE * gridDim.x - RADIUS) {
22    d_output[tid] = sBuf[localId];
23    return ;
24 }
```

Analogicznie postępujemy z obliczaniem sumy:

Listing A.25. Uśrednianie elementów wektora

```
26 float res = 0 ;
27
28 for(int i = -RADIUS; i <= RADIUS; i++)
29    res += sBuf[localId + i];
30
31 d_output[tid] = res / (2 * RADIUS + 1.f);
```

Uruchamiamy teraz program. Wynik powinien być podobny do widocznego poniżej:

```
GPU: Memcpy z pamieci komputera do GPU: 29.711279 (ms)
GPU: czas obliczen: 5.042140 (ms)
```

DRAFT

```
GPU: Memcpy z GPU do pamieci komputera: 38.995018 (ms)
GPU: czas calkowity: 71.748436 (ms)
CPU: czas obliczen: 555.263977 (ms)
```

Jak widać nowe jądro jest aż dziesięciokrotnie szybsze od poprzedniego. Stąd wniosek, że należy unikać częstych nie łączonych odczytów z pamięci globalnej[15].

Kolejna optymalizacja dotyczyć będzie transferów danych między pamięcią główną i pamięcią karty graficznej. Pierwszą rzeczą, którą zrobimy jest zastąpienie „zwykłej" pamięci pamięcią, która nie może być stronicowana. Pozwala to na szybsze transfery niż pamięć ulegająca stronicowaniu, a kopiowanie pamięci może się odbywać podczas działania jądra. Ponadto pamięć ta może być mapowana na przestrzeń adresową karty graficznej. Pamięć ta musi być jednak przydzielana z rozwagą, gdyż łatwo można wyczerpać zapas całej pamięci komputera. Poprawmy dane wejściowe i wyjściowe. Pokazuje to listing A.26 zawierający wybrane linie z funkcji `main`, plik *zad2.cu*.

Listing A.26. Korekta wybranych linii w funkcji `main`

```
47 float* h_input = 0;
48 cutilSafeCall(cudaHostAlloc((void**) &h_input,
49              mem_size, cudaHostAllocDefault));
90 float* h_output ;
91 cutilSafeCall( cudaMallocHost((void**) &h*output, mem_size));
116 cutilSafeCall(cudaFreeHost( h_input));

118 cutilSafeCall(cudaFreeHost( h_output));
```

W listingu A.26 widoczne są dwie nowe funkcje służące do alokacji pamięci: `cudaMallocHost` i `cudaHostAlloc`. Obu użyliśmy z parametrem `cudaHostAllocDefault`. Są one równoważne.

Włączmy teraz nasz program. W wyniku uzyskamy:

```
GPU: Memcpy z pamieci komputera do GPU: 25.796640 (ms)
GPU: czas obliczen: 5.062041 (ms)
GPU: Memcpy z GPU do pamieci komputera: 33.966850 (ms)
GPU: czas calkowity: 64.825531 (ms)
CPU: czas obliczen: 564.083557 (ms)
```

Uzyskaliśmy poprawę o około jedną ósmą w czasach transferu.

Kolejną rzeczą, której możemy spróbować aby przyspieszyć transfer danych jest pamięć typu *write-combining*. Również ona nie jest stronicowana, a dodatkowo nie jest także buforowana tzn. nie są używane zasoby buforów L1 i L2 procesora. Pozwala to wykorzystać te bufory do innych operacji, co może zwiększać ogólną wydajność systemu. Transfer zawartości takiej pamięci przez magistralę PCI Express może być do 40% szybszy w stosunku do transferu z pamięci tylko nie stronicowanej. Jej największym minusem są długie czasy odczytu z poziomu komputera. Dlatego należy jej używać tylko w sytuacjach, gdy będziemy pisać do takiej pamięci, a odczytem będzie się zajmować wyłącznie karta graficzna[16]. Spróbujemy użyć tej pamięci. Ze względu na jej własności zastosujemy ją tylko do tablicy `h_input`. W tym celu musimy zmienić parametr przekazywany do funkcji `cudaHostAlloc` zastępując wartość `cudaHostAllocDefault` przez `cudaHostAllocWriteCombined`:

```
48 cutilSafeCall( cudaHostAlloc((void**)&h_input,
```

---

[15] Pamięć globalna zachowuje się „przyzwoicie" tylko gdy czytamy kolejne elementy jednego bloku. Szczegółowe omówienie zasad tym rządzących wymaga jednak sporo wyjaśnień i zdecydowanie nie jest to najlepsze na to miejsce.

[16] Ponieważ jest to nowa funkcjonalność dodana do CUDA w wersji 2.2, nie zanotowałem żadnej różnicy w czasie kopiowania danych (karta GeForce 8800GT). Zauważalnych efektów nie było również na komputerze wyposażonym w kartę GTX260. Na forum CUDA nie znalazlem informacji na ten temat.

DRAFT

```
49                mem_size, cudaHostAllocWriteCombined));
```

Ponownie sprawdźmy wyniki drukowane przez program:

```
GPU: Memcpy z pamieci komputera do GPU: 25.459810 (ms)
GPU: czas obliczen: 5.057574 (ms)
GPU: Memcpy z GPU do pamieci komputera: 34.012619 (ms)
GPU: czas calkowity: 64.530004 (ms)
CPU: czas obliczen: 5761.655273 (ms)
```

Widzimy, że pogorszył się tylko czas działania programu na CPU. Wynika to z omówionych wcześniej własności pamięci *write-combining*.

Pozostał nam jeszcze jeden, prawdopodobnie najciekawszy, rodzaj pamięci do przetestowania a mianowicie pamięć mapowana (ang. *mapped memory* czasami też nazywana *zero-copy memory*). Pozwala ona zarezerwować w pamięci operacyjnej komputera obszar pamięci, która nie może być stronicowana, a następnie zmapować ten blok na pamięć karty graficznej. Jej opis można podzielić na dwie części. Pierwsza dotyczy komputerów wyposażonych w układy MCP79[17]. Nowy rodzaj pamięci eliminuje potrzebę bezpośredniego kopiowania bloków pamięci do karty graficznej. MCP79 ma bezpośredni dostęp do pamięci operacyjnej komputera, co eliminuje transfer poprzez magistralę PCI Express. W tej sytuacji nie ma w zasadzie potrzeby stosowania innych rodzajów pamięci. Druga sytuacja to komputery wyposażone w karty serii GT200[18]. Ten przypadek jest trochę bardziej skomplikowany. W tej sytuacji procesory skalarne (SM) będą miały bezpośredni dostęp do pamięci operacyjnej komputera poprzez magistralę PCIe. Nie będzie warstwy pośredniej z wykorzystaniem pamięci globalnej GPU. Jest to technologia podobna do *TurboCache* z kart GeForce serii 6. Ta pamięć jednak też ma pewne opóźnienia. Jej główne zastosowanie w tym przypadku to eliminacja kopiowania danych, które na przykład będą potrzebne tylko raz i kopiując je przed wywołaniem jądra tracilibyśmy czas. Jej użycie prawdopodobnie wydłuży czas działania jądra, ale łączny czas kopiowania danych i działania jądra powinien ulec skróceniu.
Aby skorzystać z pamięci mapowanej musimy najpierw włączyć możliwość korzystania z niej za pomocą funkcji `cudaSetDeviceFlags` z parametrem `cudaDeviceMapHost`:

```
43 cudaSetDeviceFlags(cudaDeviceMapHost);
```

Pamięć będziemy rezerwować podobnie jak wyżej. Zmianie ulegnie tylko flaga alokacji. Tym razem użyjemy `cudaHostAllocMapped`.

```
48 cutilSafeCall(cudaHostAlloc((void**)&h*input,
49    mem_size, cudaHostAllocMapped));
```

Alokacja i kopiowanie danych do `d_input` staje się bezprzedmiotowa. Zamiast tego odczytamy adres bloku pamięci mapowanej pamięci na karcie graficznej za pomocą funkcji `cudaHotGetDevicePointer`:
Listing A.27. Uzyskiwanie adresu pamięci mapowanej w przestrzeni adresowej karty graficznej

```
60 cutilSafeCall( cudaMalloc((void**) &d_input, mem_size));
61 cutilSafeCall( cudaHostGetDevicePointer((void**)&d_input ,
62    h_input, 0));
68 cutilSafeCall(cudaMemcpy(d_input, h_input, mem_size,
69         cudaMemcpyHostToDevice));
```

Również zwalnianie pamięci zajmowanej przez tablicę `d_input` nie ma teraz sensu.

---

[17] Jest to zintegrowany procesor graficzny. Znajduje się na tzw. mGPU (ang. *motherboard GPU*), czyli płytach głównych wyposażonych w zintegrowaną kartę graficzną.

[18] Dokumentacja CUDA 2.2 beta zawierała informacje, że potrzeba jest karta tej serii jednak finalna wersja dokumentacji CUDA 2.2 już nie mówi o ograniczeniu do kart z serii 10, a tylko o tym, że na pewnych kartach ten trik nie działa.

DRAFT

```
122 cutilSafeCall( cudaFree( d_input ));
```

Po uruchomieniu programu powinniśmy otrzymać wynik zbliżony do poniższego:

```
GPU: Memcpy z pamieci komputera do GPU: 0.000687 (ms)
GPU: czas obliczen: 28.628111 (ms)
GPU: Memcpy z GPU do pamieci komputera: 34.062088 (ms)
GPU: czas calkowity: 62.690886 (ms)
CPU: czas obliczen: 564.056335 (ms)
```

Widzimy, że program jest minimalnie szybszy. Gdyby jądro było bardziej ograniczone przez obliczenia, a nie przez dostęp do pamięci, to zysk byłby z pewnością zdecydowanie większy.

DRAFT