<<AUTOR>>

Michał Matuszak, Jacek Matulewski

<</AUTOR>>

<<TYTUL>>

CUDA i czyny

<</TYTUL>>

<<TOP>>

Technologia NVIDIA CUDA

<</TOP>>

<<LEAD>>

W zeszłomiesięcznym numerze SDJ w artykule pt. *Czyń cuda* opisaliśmy architekturę procesorów i pamięci kart graficznych wspierających technologię CUDA oraz strukturę programów, które mogą korzystać z tej technologii, w szczególności kod umożliwiający uruchamianie jąder (ang. *kernel*), czyli programów przesyłanych do pamięci kart graficznych i uruchamianych na jej procesorach. W tym artykule skupimy się na programowaniu jąder, a przede wszystkim omówimy kilku dróg do jego optymalizacji.

<</LEAD>>

<<Ramka: Z artykułu dowiesz się>>

GPGPU to skrót, który na ustach informatyków pojawia się coraz częściej. Oznacza *general-purpose computing on graphics processing units*, czyli możliwość przeprowadzania dowolnych silnie zrównoleglonych obliczeń na procesorach kart graficznych, których spora moc była do tej pory wykorzystywana jedynie do generowania grafiki trójwymiarowej, czyli w wielu przypadkach okazjonalnie. Z drugiej części artykułu dowiesz się jak przygotowywać tzw. jądra wykonywane przez procesory kart graficznych i jak w sposób optymalny przekazywać dane do pamięci karty graficznej.

<<Ramka: Powinieneś wiedzieć>>

Od czytelnika wymagana jest znajomość C++ oraz podstawowych zasad programowania równoległego. Potrzebna będzie również przedstawiona w pierwszej części artykułu znajomość architektury kart graficznych zgodnych z CUDA oraz umiejętność uruchamiania jąder CUDA.

<<Ramka: Więcej w książce>>

Artykuł jest fragmentem dodatku dołączonego do książki *Visual C++. Gotowe rozwiązania dla programistów Windows*, które wkrótce ukaże się nakładem Wydawnictwa Helion.

<<Ramka: Więcej w sieci>>

- *http://www.nvidia.com/object/cuda_home.html* - Oficjalna strona NVIDIA CUDA
- *http://developer.nvidia.com/object/gpu_computing_online.html* - materiały edukacyjne dotyczące CUDA
- *http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf* - duży (145 stronicowy) przewodnik dla programistów korzystających z CUDA
- *http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf* - dokumentacja wszystkich funkcji CUDA i towarzyszącego mu pakietu pomocniczego
- *http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf* - zbiór tzw. najlepszych rozwiązań (ang. best practices) w programowaniu z użyciem CUDA

<<Ramka: O autorach>>

Michal Matuszak - doktorant na Wydziale Matematyki i Informatyki Uniwersytetu Mikołaja Kopernika w Toruniu. Jego główne zainteresowania to sieci bayesowskie i gaussowskie oraz programowanie równoległe.

Jacek Matulewski - Fizyk zajmujący się na co dzień optyką kwantową i układami nieuporządkowanymi na Wydziale Fizyki, Astronomii i Informatyki Stosowanej Uniwersytetu Mikołaja Kopernika w Toruniu. Jego specjalnością są symulacje ewolucji układów kwantowych oddziaływujących z silnym światłem lasera. Od 1998 interesuje się programowaniem dla systemu Windows. Ostatnio zainteresowany platformą .NET i językiem C#, a także XNA. Wierny użytkownik kupionego w połowie lat osiemdziesiątych "komputera osobistego" ZX Spectrum 48k.

CUDA i czyny

W zeszłomiesięcznym numerze SDJ w artykule pt. Czyń cuda opisaliśmy architekturę procesorów i pamięci kart graficznych wspierających technologię CUDA oraz strukturę programów, które mogą korzystać z tej technologii, w szczególności kod umożliwiający uruchamianie jąder (ang. kernel), czyli programów przesyłanych do pamięci kart graficznych i uruchamianych na jej procesorach. W tym artykule skupimy się na programowaniu jąder, a przede wszystkim omówimy kilku dróg do jego optymalizacji.

Przykład programu korzystającego z CUDA

Przygotujemy 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 listingu 1. Funkcję tą należy umieścić w pliku *zad2.cu* z drugiego projektu startowego umieszczonego na płycie dołączonej do tego wydania SDJ.

Listing 1. Funkcja wykonująca rozmycie wektora na CPU

<LISTING lang=C/C++>

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

</LISTING>

Stałą `RADIUS` definiujemy w pliku *zad2_kernel.cu*. Odpowiada ona za ilość sąsiadów, których uwzględniamy przy uśrednianiu. Definiujemy tam również ilość wątków w bloku. Na razie ustawimy dość małą wartość, co ułatwi nam znalezienie ewentualnych błędów w kodzie.

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

Tak samo, jak w przykładzie omówionym w pierwszej części artykułu zainicjujemy GPU w funkcji `main` (plik *zad2.cu*) wstawiając do niej polecenia z listingu 2.

<LISTING lang=C/C++>

Listing 2. 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() );
```

</LISTING>

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. Odpowiedni kod pokazuje listing 3. Przygotowujemy też dwie tablice ulokowane w pamięci karty graficznej oraz kopiujemy do jednej z nich wyznaczone w poprzednim kroku dane wejściowe (listing 4). Typ kopiowania wskazany w czwartym argumencie w funkcji `cudaMemcpy` (zob. opis w poprzednim artykule) musimy ustalić w taki sposób, aby kopiować z pamięci komputera do pamięci karty graficznej (czwarty parametru równy `cudaMemcpyHostToDevice`).

<LISTING lang=C/C++>
Listing 3. 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();
```

</LISTING>
<LISTING lang=C/C++>
Listing 4. 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));
```

</LISTING>

Następnie ustalamy wielkość kraty i ilość wątków jakich będziemy używać. Zakładamy, że rozmiar tablicy równy jest wielkości tablicy wejściowej `inputSize` podzielonej przez rozmiar bloku.

<LISTING lang=C/C++>
Listing 5. 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 );
```

</LISTING>

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 6. 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 lang=C/C++>

Listing 6. 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”);
```

</LISTING>

Zauważmy, że tym razem do synchronizacji korzystamy bezpośrednio z funkcji CUDA API `cudaThreadSynchronize` zamiast z funkcji pomocniczej.
Podobnie jak w programie z poprzedniego artykułu, rezerwujemy pamięć komputera oraz kopiujemy do niej wynik z pamięci karty graficznej. Odpowiednie instrukcje widoczne są na listingu 7. Na tym samym listingu widoczne są instrukcje sprawdzające czy wyniki obliczeń przeprowadzonych na karcie graficznej pokrywają się z tymi otrzymanymi tradycyjną metodą. Na ekranie pokazywane są tylko wartości, dla których wyniki uzyskane obydwoma metodami nie pokrywają się ze sobą i jedyne, co pozostaje do zrobienia, to posprzątanie po sobie.

<LISTING lang=C/C++>

Listing 7. 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 ) ) ;


75 // 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]);
80
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);
```

</LISTING>

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. Podobnie, jak w przykładzie z poprzedniej części artykułu pierwszą czynnością w funkcji jądra jest wyznaczenie indeksu wątku (linia 9 w listingu 8). Następnie 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 8).

<LISTING lang=C for CUDA>

Listing 8. 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);
```

</LISTING>

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.
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 7.

Pomiary czasu obliczeń na GPU

Do kodu wykonywanego na CPU dodamy teraz zegary mierzące czas wykonywania poszczególnych części programu. Ułatwi to kontrolę optymalizacji, którą za chwilę będziemy próbowali przeprowadzić, a także pozwoli na porównanie czasu działania klasycznej wersji programu uruchamianej na CPU z naszą implementacją na GPU. Zaczniemy od modyfikacji funkcji `computeGold` wykonywanej na CPU (plik *zad2.cu*). Pokazuje to listing 9.

<LISTING lang=C/C++>

Listing 9. 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));
```

</LISTING>

Użyte w tym fragmencie kodu makro CUT_SAFE_CALL wywołuje podaną jako argument funkcję oraz sprawdza, czy zakończyła się sukcesem. Podobne instrukcje wstawimy w dalszej części naszego kodu. Zmierzymy w ten sposób czas kopiowania danych z pamięci głównej do pamięci karty graficznej (listing 10), czas działania jądra (listing 11) oraz czas kopiowania wyników z pamięci karty graficznej do pamięci głównej komputera (listing 12). Nie możemy zapomnieć o zwolnieniu zasobów zegarów (listing 13).

<LISTING lang=C/C++>

Listing 10. 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));
```

</LISTING>

<LISTING lang=C/C++>

Listing 11. 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));
```

<LISTING lang=C/C++>

Listing 12. 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 ));
```

</LISTING>

<LISTING lang=C/C++>

Listing 13. 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
```

</LISTING>

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 7. Uruchamiając teraz program otrzymamy wynik zbliżony do:

```
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)
```

Powyższe wyniki obliczeń pochodzą z karty GeForce 8800GT.

Optymalizacje

Widać, że implementacja naszego programu działającego na GPU mimo, że nie zawiera żadnych optymalizacji jest ok. 5 razy szybsza od wersji działającej na CPU. Przy czym same obliczenia korzystające z GPU są aż 11 razy szybsze od analogicznych przeprowadzonych tylko na CPU. Sporo czasu tracimy na kopiowaniu danych do i z karty graficznej. Wydajność transferów poprawimy później. Na razie skupimy się na samym jądrze. Omawiając architekturę kart CUDA w pierwszej części artykułu, zwracałem uwagę na to, że pamięć globalna jest bardzo wolna w porównaniu z szybką pamięcią współdzieloną. Spróbujmy uwzględnić ten fakt w programie. Można zauważyć, że każdy blok używa tylko `BLOCK_SIZE+2*RADIUS` elementów wektora wejściowego. Możemy zatem 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 tak, jak pokazano na listingu 14.

<LISTING lang=C for CUDA>

Listing 14. 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;
```

</LISTING>

Możemy zacząć kopiowanie. Każdy wątek z bloku przekopiuje odpowiadający mu element do tablicy współdzielonej (listing 15). 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. Analogicznie postępujemy obliczając sumy (listing 16).

<LISTING lang=C for CUDA>

Listing 15. 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 }
```

</LISTING>

<LISTING lang=C for CUDA>

Listing 16. 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);
```

</LISTING>

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)
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. Pamięć globalna zachowuje się całkiem 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.

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 17 zawierający wybrane linie z funkcji `main`, plik *zad2.cu*. W instrukcjach na listingu 17 widoczne są dwie nowe funkcje służące do alokacji pamięci: `cudaMallocHost` i `cudaHostAlloc`. Obu użyliśmy z parametrem `cudaHostAllocDefault`. Ich działanie jest identyczne.

<LISTING lang=C/C++>

Listing 17. 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));
```

</LISTING >

Ponownie skompilujmy i uruchommy program. Teraz 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. Należy jednak zwrócić uwagę, że jest to nowa funkcjonalność dodana do CUDA w wersji 2.2. Może dlatego na używanej przez nas karcie GeForce 8800GT nie zanotowaliśmy żadnej różnicy w czasie kopiowania danych. Zauważalnych efektów nie było również na komputerze wyposażonym w kartę GTX260. Na forum CUDA nie znaleźliśmy informacji na ten temat.

Mimo to 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,
49                 mem_size, cudaHostAllocWriteCombined));
```

Następnie 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 spowodowało to pogorszenie czasu działania programu na CPU. Wynika to z omówionych wcześniej własności pamięci *write-combining*.

Pozostał nam do przetestowania jeszcze jeden, prawdopodobnie najciekawszy rodzaj pamięci, a mianowicie pamięć mapowana (ang. *mapped memory* czasami też nazywana *zero-copy memory*). Pozwala ona zarezerwować

w pamięci operacyjnej komputera obszar, 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 tj. w zintegrowany procesor graficzny, który znajduje się na tzw. mGPU (ang. *motherboard GPU*), czyli płytach głównych wyposażonych w zintegrowaną kartę graficzną. 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 (tą serię wskazuje dokumentacja CUDA 2.2 beta; finalna wersja dokumentacji CUDA 2.2 już nie wspomina o ograniczeniu do kart z serii 10, a tylko o tym, że na pewnych kartach ten trik nie działa). 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`. Pokazuje to listing 18.

<LISTING lang=C/C++>

Listing 18. 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));
```

</LISTING>

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

```
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.

Doszliśmy do końca artykułu. Mogliśmy się przekonać, że stworzenie własnego programu wykorzysującego moc kart graficznych jest stosunkowo prostym zadaniem. Jednak, aby w pełni wykorzystać potencjał drzemiący w GPU potrzeba większej wiedzy i doświadczenia. Pisząc kolejny programy musimy pamiętać o kilku podstawowych zasadach tj.na początku należy skupić się na sposobie zrównoleglenia sekwencyjnego kodu. Jest to kluczowy moment i nie powinniśmy lekceważyć tego kroku. W kolejnym kroku powinniśmy minimalizować kopiowanie pamięci pomiędzy pamięcią komputera, a pamięcią karty. Czasami może się okazać, że niektóre

zadania są szybciej obliczane przez CPU, niż przez nasze jądro. Jednak narzut związany z kopiowania pamięci może zniweczyć potencjalny zysk z przeniesienia częsci obliczeń na CPU. Następnie powinniśmy ograniczać korzystanie z pamięci globalnej na rzecz pamięci współdzielonej oraz gdy to jest możliwe nie stosować instrukcji warunkowych mogących rozgałęziać kod wewnątrz warpa.