Biuletyn WAT
Vol. LX, Nr 4, 2011

Zastosowanie technologii CUDA w rozpoznawaniu wzorców nieregularnych

WITOLD ŻORSKI, MICHAŁ MAKOWSKI

Wojskowa Akademia Techniczna, Wydział Cybernetyki, Instytut Teleinformatyki i Automatyki,
00-908 Warszawa, ul. S. Kaliskiego 2, wzorski@ita.wat.edu.pl, mmakowski@wat.edu.pl

Streszczenie. W artykule przedstawiono implementację techniki rozpoznawania wzorców nieregularnych przy zastosowaniu technologii CUDA. Zasygnalizowano możliwości współczesnych procesorów graficznych firmy NVIDIA o architekturze Fermi. Przytoczono podstawowe reguły programowania w CUDA. Dokonano wyboru metody segmentacji wzorcami nieregularnymi opartej na transformacie Hougha, jako odpowiedniej do wykorzystania potencjału procesora graficznego. Opisano kluczowe fragmenty implementacji. Dokonano weryfikacji działania w zakresie szybkości i poprawności obliczeń.
Słowa kluczowe: technologia CUDA, programowanie współbieżne, transformata Hougha, widzenie komputerowe, wzorce nieregularne, segmentacja

1. Wprowadzenie

Podstawowym warunkiem zastosowania danej techniki rozpoznawania w systemie widzenia komputerowego [4, 18] jest czas obliczeń [9]. Złożoność obliczeniowa niektórych algorytmów segmentacji [16] uniemożliwia ich stosowanie w systemach widzenia komputerowego, od których oczekiwana jest reakcja na bieżąco [3, 15, 21]. Niekiedy stosuje się środki pozwalające zmniejszyć czas obliczeń, jednak zazwyczaj skutkują one obniżeniem jakości działania systemu. Skrajnym przypadkiem jest zmniejszanie rozdzielczości obrazów wejściowych lub redukcja poziomów jasności.

Przykładem techniki widzenia komputerowego, którą cechuje wyjątkowo duże zapotrzebowanie na czas obliczeń, jest transformata Hougha [7, 8]. Wielką zaletą tej techniki jest jednak jej skuteczność, nawet w przypadku wykrywania „słabych" obiektów.

W przypadku transformaty Hougha [5, 12] przez wiele lat stosowano różne metody skracające czas obliczeń. Można tutaj wyróżnić: podejście hierarchiczne [14], podejście probabilistyczne [11], randomizację [19], niejednorodny podział przestrzeni parametrów [13], podział w przestrzeni obrazu [23], analizę histogramową [20].

Drugą poważną wadą transformaty Hougha jest znaczne zapotrzebowanie na pamięć w przypadku, gdy liczba parametrów jest większa od trzech [24]. Ten problem stracił jednak na znaczeniu wraz z rozwojem pamięci półprzewodnikowych.

Zaletą transformaty Hougha jest możliwość zrównoleglenia obliczeń, gdyż wartości w przestrzeni parametrów (akumulatorze) nie są od siebie obliczeniowo zależne. Współczesne procesory posiadają kilka (2-6) rdzeni fizycznych, a niektóre też wielowątkowość[1], co mogłoby wydawać się drogą do rozwiązania problemu. Niestety pisanie oprogramowania wykorzystującego wielordzeniowość procesorów opartych na architekturze x86 jest trudne[2] (tylko nieliczne programy, najczęściej gry, wykorzystują więcej niż dwa rdzenie). Od roku 2007, za sprawą firmy NVIDIA, można wykorzystać potencjał dziesiątek lub nawet setek rdzeni procesorów graficznych GPU (ang. *Graphics Processing Unit*) [10]. Wiele rozwiązań funkcjonuje już w zakresie akceleracji obliczeń naukowych, medycynie, finansach i wielu innych dziedzinach. Z konkretnymi przykładami można zapoznać się w materiałach „GPU Technology Conference"[3].

W niniejszym opracowaniu przedstawiono przykład zastosowania technologii CUDA [17] do zrównoleglenia obliczeń [2, 6]. Wybrano przypadek rozpoznawania wzorców nieregularnych techniką zaproponowaną przez Ballarda [1], będącą w istocie uogólnieniem transformaty Hougha na obiekty nieposiadające opisu analitycznego. Technikę Ballarda z powodzeniem można stosować w przypadku wykrywania obiektów nieregularnych na obrazach w poziomach szarości [21, 22], a nawet bezpośrednio w obrazach kolorowych [25].

2. Technologia CUDA i architektura Fermi

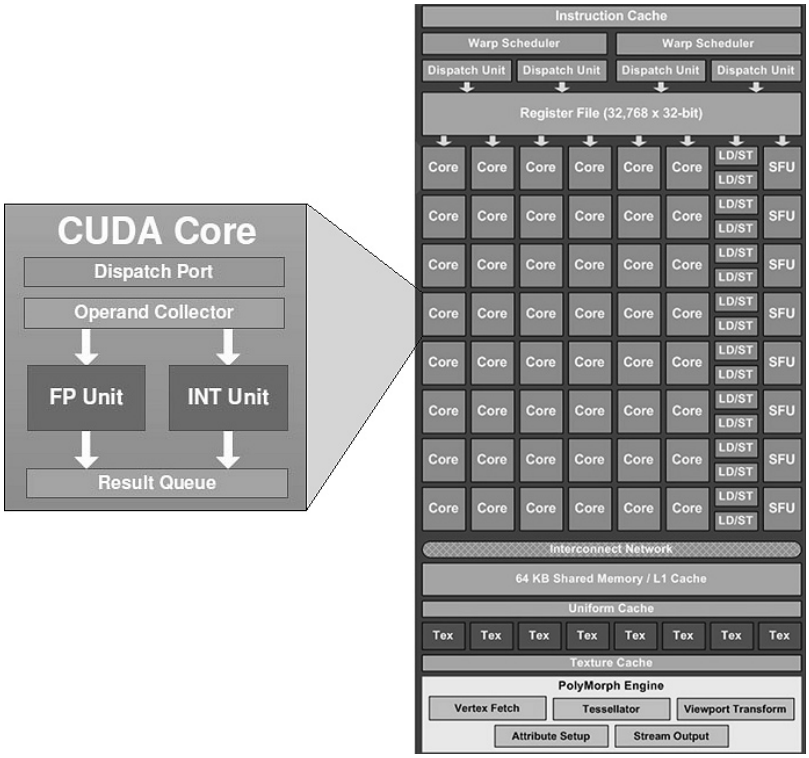
W 2006 roku firma NVIDIA wprowadziła na rynek rodzinę układów graficznych GeForce 8, które jako pierwsze były kompatybilne z DirectX 10. Już w lutym 2007 roku NVIDIA zaproponowała dla tych układów technologię CUDA (ang. *Compute Unified Device Architecture*), która pozwalała na pisanie oprogramowania wykorzystującego moc obliczeniową procesora GPU. Widząc rosnące zainteresowanie

[1] HT (ang. *Hyper-Threading technology*) w przypadku niektórych procesorów firmy Intel.

[2] Intel w 2010 roku wysłał wersje inżynierskie 50-rdzeniowych procesorów „Knights Corner" do różnych ośrodków naukowych w celu prowadzenia badań nad nowymi technikami programowania współbieżnego.

[3] http://www.nvidia.com/object/gtc2010-presentation-archive.html

rynku nową technologią, NVIDIA zdecydowała się na opracowanie całkiem nowej architektury procesorów graficznych o kodowej nazwie Fermi. Wyzwanie okazało się niezwykle ambitne i kosztowne, gdyż pociągnęło za sobą wykonanie układu GPU zawierającego ok. 3 mld tranzystorów. Przy zastosowaniu dostępnego procesu technologicznego (40 nm) zadanie to było ledwie realizowalne. Po roku od premiery Fermi największą popularnością cieszą się układy: GTX 460 i GTX 560 Ti (nadal wytwarzane w 40 nm), zawierające odpowiednio: 336 i 384 rdzenie CUDA (ang. *CUDA cores*). Znaczenie technologii CUDA może podkreślić fakt zastosowania 7168 modułów Tesla M2050[4] (obok 14336 zwykłych procesorów) w najszybszym superkomputerze świata Tianhe-1A. Współczesne procesory graficzne dysponują mocą obliczeniową w okolicach 1 TFLOPS-a[5] (np. Tesla M2050).



Rys. 1. Budowa pojedynczego rdzenia CUDA i struktura bloku multiprocesora

[4] Moduł Tesla M2050 zawiera 448 rdzeni CUDA.

[5] Moc obliczeniowa 1 TFLOPS-a (ang. *Tera FLoating point Operations Per Second*) została po raz pierwszy osiągnięta w 1997 roku za sprawą superkomputera ASCI Red (9152 procesory Pentium Pro 200MHz).

Na potrzeby niniejszego opracowania wykorzystano układ[6] GTX 460, którego wewnętrzna struktura składa się z 8 bloków[7] SM (ang. *Streaming Multiprocessor*). Każdy multiprocesor SM zawiera 48 procesorów CUDA. Budowa bloku SM i pojedynczego rdzenia CUDA ukazana jest na rysunku 1. Układ GTX 460 jest zgodny z potencjałem obliczeniowym[8] (ang. *Compute capability*) w wersji 2.1, co daje obecnie najszersze możliwości w zakresie tworzenia oprogramowania. Komputer, w którym zainstalowana jest karta graficzna obsługująca CUDA, nazywany jest hostem, a sama karta urządzeniem CUDA (lub po prostu urządzeniem).

3. Istota programowania w CUDA [27]

Tworzenie oprogramowania wykorzystującego potencjał CUDA wymaga znajomości podstaw modelu programistycznego związanego z tą technologią. CUDA rozszerza możliwości języka C o definiowanie funkcji zwanych kernelami. Wywołany kernel jest wykonywany równolegle N razy w N różnych wątkach (ang. *threads*). Funkcja jest deklarowana jako kernel poprzez użycie słowa kluczowego __global__. Do każdego wywołania kernela wymagane jest podanie konfiguracji wykonania (ang. *execution configuration*).

Konfiguracja wykonania określa wymiary siatki (ang. *grid*) oraz bloków (ang. *blocks*), które zostaną użyte przy danym wywołaniu kernela. Każdy wątek ma dostęp do wbudowanych zmiennych:

- gridDim — wymiary siatki,
- blockDim — wymiary bloku,
- blockIdx — współrzędne bloku, w którym znajduje się wątek,
- threadIdx — współrzędne wątku w bloku.

Każda z powyższych zmiennych to struktura zawierająca trzy wartości typu int:x,y,z, które pozwalają zidentyfikować dany wątek oraz określić, którą porcję danych ma przetwarzać. Wspomniane struktury (siatka, bloki oraz wątki) są zorganizowane w sposób ukazany na rysunku 2. Konfigurację wykonania można określić w kodzie źródłowym na dwa sposoby:

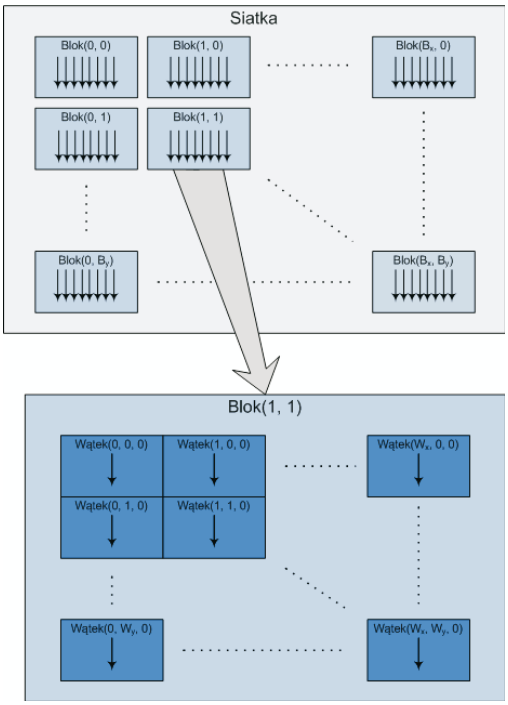
- poprzez użycie specjalnego wyrażenia, którego najprostsza postać wygląda następująco: <<<gridDim,blockDim>>> (ta metoda jest możliwa tylko w językach C/C++);
- poprzez wywołanie odpowiedniej sekwencji funkcji sterownika CUDA.

---

[6] Kartę graficzną Gigabyte GV-N460OC-1GI na procesorze GTX 460.

[7] Niestety w układzie GTX 460 jeden blok jest wyłączony fabrycznie (w GTX 560 Ti wszystkie są aktywne).

[8] Notacja stworzona przez firmę NVIDIA do określania możliwości (architektury) danego układu graficznego.



Rys. 2. Hierarchia struktur: siatka, blok, wątek

Układy NVIDIA korzystają z architektury SIMT (ang. *Single-Instruction, Multiple-Thread*). Jednostki SIMT zarządzają wątkami poprzez łączenie je w tzw. warpy (podział ten nie jest ukazany na rysunku 2). Każdy warp[9] to 32 wątki rozpoczynające wykonywanie programu od tego samego adresu. Możliwe są rozgałęzienia i każdy z wątków w warpie może realizować inną ścieżkę programu. Należy jednak mieć na uwadze, że w ramach warpu może być wykonana tylko jedna wspólna instrukcja. Pełną wydajność uzyskuje się wtedy, gdy wszystkie wątki warpu podążają tą samą ścieżką programu. Jeżeli zdarzy się, że rozgałęzienie wystąpi w ramach warpu, to każda ze ścieżek programu jest realizowana szeregowo, aż do momentu powrotu wszystkich wątków na ścieżkę wspólną. Mówi się wtedy o wystąpieniu dywergencji w warpie.

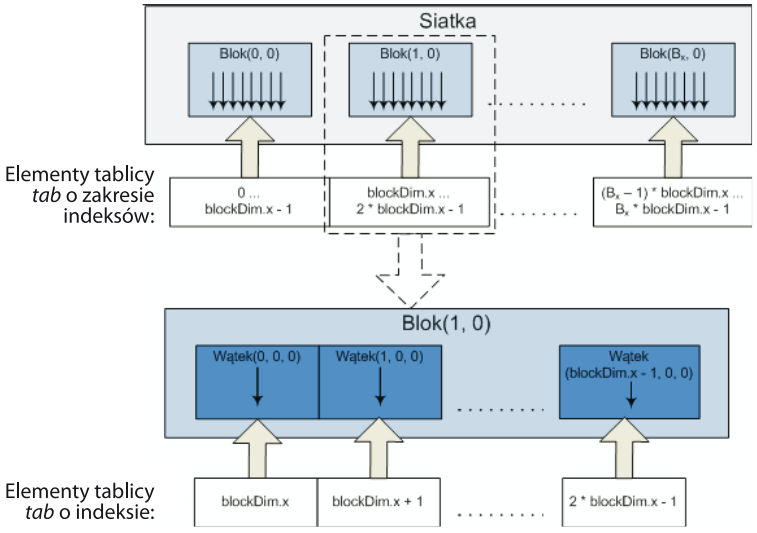
Niech za przykład kernela posłuży trywialna funkcja zwiększająca wartości N-elementowej tablicy tab o 1:

[9] Half-warpem jest pierwsza lub druga połowa warpu (wątki o numerach od 0 do 15 lub od 16 do 31 w warpie).

```
__global__ void incrementArray(int *tab, int N)
{
// obliczenie indeksu wątku
int idx = blockIdx.x * blockDim.x + threadIdx.x;
// inkrementacja pojedynczego elementu tablicy
if (idx < N) tab[idx]++;
}
```

Inkrementacja każdego elementu tablicy musi składać się z dwóch etapów:

1. Ustalenia elementu tablicy, który dany wątek przetworzy.
2. Inkrementacji tego elementu, jeśli indeks wątku nie przekracza rozmiaru tablicy.



Rys. 3. Konfiguracja uruchomienia kernela „incrementArray"

W rozważanym przykładzie każdy wątek oblicza (na podstawie zmiennych wbudowanych) swój bezwzględny numer indeksu w siatce, w której ten kernel został uruchomiony. Indeks ten równocześnie określa, który element tablicy zostanie przez dany wątek zwiększony o 1. Należy zwrócić uwagę, że uwzględniane są jedynie współrzędne blockIdx.x oraz threadIdx.x wątku. Zatem założono, że kernel będzie uruchamiany w jednowymiarowej siatce jednowymiarowych bloków, co pokazano na rysunku 3 (dla uproszczenia przyjęto, że rozmiar tablicy tab jest całkowitoliczbową wielokrotnością zmiennej blockDim.x).

W układach GPU występują istotne ograniczenia dotyczące deklarowanych rozmiarów siatki i bloków, a także liczby aktywnych wątków w bloku. Przykładowo,

dla urządzenia o potencjale obliczeniowym 2.x (np. wspomniany GTX 460) ograniczenia są następujące:

— maksymalna liczba wątków w bloku: 1024,
— maksymalne rozmiary bloku: 1024×1024×64,
— maksymalne rozmiary siatki: 65535×65535×1.

Powyższe zestawienie nie oznacza niestety, że na danym GPU jednocześnie może być przetwarzanych ponad 4 miliardy bloków po 1024 wątki każdy. Gdy wywoływany jest kernel, bloki są numerowane i rozprowadzane do multiprocesorów układu GPU (w GTX 460 jest 7 aktywnych multiprocesorów). W urządzeniu o potencjale obliczeniowym 2.x pojawiają się następujące ograniczenia:

— maksymalna liczba aktywnych bloków na multiprocesor: 8,
— maksymalna liczba aktywnych warpów na multiprocesor: 48,
— maksymalna liczba aktywnych wątków na multiprocesor: 1536.

Wynika stąd, że maksymalna liczba aktywnych wątków na multiprocesor wyraźnie przekracza liczbę fizycznie dostępnych rdzeni CUDA. W przypadku rozważanego potencjału obliczeniowego 2.x mamy dwa „warp schedulery" na multiprocesor, a w każdym cyklu warp scheduler emituje aż dwie instrukcje. Jednak ukrycie latencji na drodze optymalizacji rozlokowania wątków nie jest łatwe.

Wątki mają dostęp do następujących zasobów pamięci:

— globalnej (ang. *global memory*) — wspólnej dla wszystkich wątków,
— lokalnej (ang. *local memory*) — prywatnej pamięci pojedynczego wątku,
— współdzielonej (ang. *shared memory*) — wspólnej dla wszystkich wątków w danym bloku,
— puli 32-bitowych rejestrów (ang. *registers*).

Występują jeszcze dwa obszary pamięci wspólnej (tylko do odczytu z poziomu wątków):

— pamięć wartości stałych (ang. *constant memory*),
— pamięć tekstury (ang. *texture memory*).

4. Wybór techniki widzenia komputerowego

Jak zasygnalizowano we wprowadzeniu, dokonana została implementacja transformaty Hougha dla przypadku wzorców nieregularnych[10] w poziomach szarości. Technika została już opisana detalicznie w publikacjach [20, 24] i tutaj zostanie przedstawiona tylko w zarysie.

Zakładamy, że dany jest obraz wejściowy w poziomach szarości $I_G(x, y)$ oraz wzorzec M_p, dla którego poszukiwane są obiekty w obrazie wejściowym. W przypadku

[10] Wzorzec nieregularny można przedstawić jako listę pikseli [1] lub jako obraz (zazwyczaj niewielkich rozmiarów).

gdy zachodzi konieczność rozważania operacji przesunięcia (x_T, y_T) i obrotu (o kąt α) wzorca (rys. 4), transformatę Hougha $H(x_T, y_T, \alpha)$ można zdefiniować następująco:

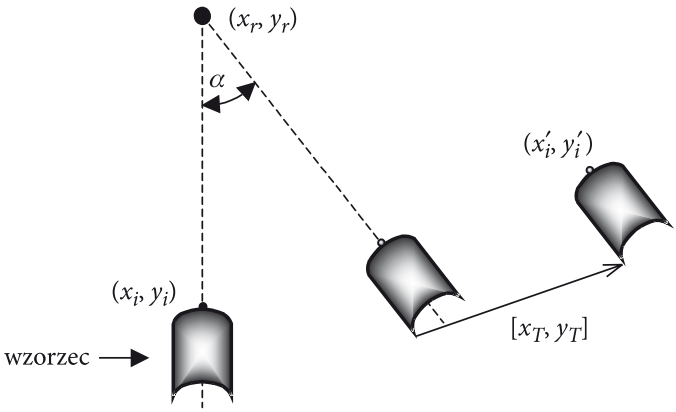
$$H(x_T, y_T, \alpha) = \sum_{(x_i, y_i) \in M_P} h(x_i, y_i, x_T, y_T, \alpha), \tag{1}$$

gdzie funkcja podobieństwa dana jest wzorem:

$$h(x_i, y_i, x_T, y_T, \alpha) = 255 - \left| I_G(x_i', y_i') - M_P(x_i, y_i) \right|, \tag{2}$$

a wartości x_i', y_i' wyznacza się ze wzoru:

$$\begin{cases} x_i' = x_r + (x_i - x_r)\cos(\alpha) - (y_i - y_r)\sin(\alpha) + x_T \\ y_i' = y_r + (x_i - x_r)\sin(\alpha) + (y_i - y_r)\cos(\alpha) + y_T. \end{cases} \tag{3}$$



Rys. 4. Obrót i translacja wzorca względem arbitralnego punktu (x_r, y_r)

W przypadku komputerowej implementacji transformaty Hougha można powyższe wzory stosować bezpośrednio, ale w praktyce lepiej jest zastosować równoważną technikę, którą określić można mianem „stemplowania obrazu wzorcem". W systemie komputerowym należy wygenerować (dla danego wzorca wejściowego) wzorce obrócone (o ustalony kwant $\Delta\alpha = 2\pi / L$ kąta α, L — liczba wzorców obróconych). W celu dopasowania wzorca do obiektu (ang. *template matching*) należy każdy wzorzec (obrócony o kąt α) porównać z każdym możliwym fragmentem obrazu (określonym przez x_T, y_T). Porównanie polega na wykorzystaniu wzoru (2) dla wszystkich pikseli wzorca i fragmentu obrazu. Zastosowanie wzoru (1) pozwoli wyznaczyć wartość $H(x_T, y_T, \alpha)$. W systemie komputerowym zachodzi konieczność przechowywania obliczanych wartości $H(x_T, y_T, \alpha)$ w tablicy, którą

nazywamy akumulatorem. Rozmiary akumulatora są dostosowane do rozważanych zakresów parametrów (x_T, y_T, α). Po zakończeniu procesu akumulacji należy wyznaczyć współrzędne akumulatora dla wartości największej (albo najmniejszej, jeśli we wzorze (2) pozostawimy po prawej stronie tylko moduł), a tym samym pozyskać informację o położeniu i orientacji obiektu najlepiej pasującego do wzorca.

5. Implementacja przy zastosowaniu technologii CUDA

Przedstawiona poniżej implementacja[11] transformaty Hougha, dla przypadku wzorców nieregularnych w poziomach szarości, zrealizowana została w języku C#, jako aplikacja typu Windows Forms, w środowisku Microsoft Visual Studio 2008.

Blok wątków CUDA powinien być przydzielony do obliczenia fragmentu akumulatora w taki sposób, aby zminimalizować liczbę koniecznych dostępów do pamięci związanych z odczytem wartości kolejnych pikseli obrazu. Tak skonstruowany kernel należy uruchomić L razy (liczba ustalonych dopuszczalnych wartości parametru α). Dystrybucję zadań dla wątków CUDA można zatem opisać następująco:

- pojedynczy wątek zajmuje się obliczeniem jednej komórki akumulatora,
- wątki są zorganizowane w bloki o wymiarach 256×1×1 (dla ułatwienia buforowania wartości pikseli obrazu), a każdy blok realizuje obliczenie 256 kolejnych komórek akumulatora znajdujących się w tym samym wierszu,
- bloki są zorganizowane w siatkę o wymiarach $[(W - N_p + 1)/256] \times (H - N_p + 1)$, gdzie: W, H — szerokość i wysokość obrazu wejściowego $I_G(x, y)$, $N_p \times N_p$ — rozmiar wzorca M_p.

Założone rozmiary wzorca M_p, dla których napisany został kernel, wynoszą 32×32. Zakłada się, że obrócone wzorce są przygotowane wcześniej, bez udziału technologii CUDA. Kernel uruchamiany jest dla każdego obróconego wzorca osobno.

Parametrami opisywanego kernela Hough są: uint *A — wskaźnik do zarezerwowanego na akumulator obszaru pamięci, uint *image — wskaźnik na pierwszy piksel obrazu wejściowego, uint *_pattern — wzorzec obrócony o rozważany aktualnie kąt α, uint *_mask — wskaźnik do tablicy zawierającej maskę (wyodrębniającą kołową część wzorca), uint w — szerokość obrazu wejściowego:

[11] Podstawowym warunkiem korzystania z potencjału CUDA jest zainstalowanie w systemie (oprócz sterownika karty graficznej) pakietu CUDA Toolkit http://developer.nvidia.com/object/cuda_3_2_downloads.html).

```
extern „C" __global__ void Hough(uint *A, uint *image, uint *_pattern, uint *_mask,
uint w)
{
```

Deklaracja tablic umieszczonych w pamięci współdzielonej (buforów tablic stanowiących parametry dla kernela Hough) jest następująca:

```
const int sh_size = 32;
 __shared__ byte mask[sh_size * sh_size];
 __shared__ byte pattern[sh_size * sh_size];
 __shared__ int imagebuff[BLOCK_LEN + sh_size];
```

Przygotowanie wartości stałych dla danego wątku:

```
 int thidx = threadIdx.x;
 int picy = blockIdx.y;
 int picx = blockIdx.x * BLOCK_LEN;
```

Kopiowanie danych maski oraz wzorca do tablic współdzielonych:

```
#pragma unroll 4
 for (int i = 0; i < 4; i++)
 {
 int tmp = _mask[thidx + i * BLOCK_LEN];
 mask[4 * thidx + i] = (byte)tmp;
 tmp = _ pattern[thidx + i * BLOCK_LEN];
 pattern[4 * thidx + i] = (byte)tmp;
 }
```

Inicjalizacja wartości Acell dla komórki akumulatora obliczanej przez dany wątek; ze względu na kopiowane krok wcześniej dane do buforów (które muszą stać się widoczne dla wszystkich wątków) występuje synchronizacja funkcją __syncthreads():

```
 int Acell = 0;
 __syncthreads();
```

Wejście do głównej pętli kernela (picy_d oznacza numer przetwarzanego wiersza wzorca):

```
int picy_d = 0;
#pragma unroll 32
for (int point = 0; point < sh_size * sh_size; point++)
{ //for — start
```

Wyznaczenie (na podstawie iteratora point) numeru indeksu danych w tablicach mask oraz pattern (rozłożenie danych jest konieczne, aby uniknąć konfliktów banków przy dostępie do obszarów pamięci współdzielonej):

```
int tmpidx = (point% 256) * 4 + point / 256;
```

Ładowanie do bufora image kolejnego wiersza obrazu:

```
int shx = point% sh_size;
if (shx == 0)
{
int pix = thidx + picx;

if (pix < w)
{
imagebuff[thidx] = image[w * (picy + picy_d) + pix];
if (pix + sh_size — 1 < w && thidx < sh_size) imagebuff[BLOCK_LEN + thidx] =
image[w * (picy + picy_d) + pix + BLOCK_LEN];
}
picy_d++;
}
```

Synchronizacja ze względu na zmianę zawartości bufora imagebuff:

```
__syncthreads();
```

Jeżeli wartość maski (odpowiadająca współrzędnym aktualnie przetwarzanego piksela wzorca) jest równa 1, to następuje akumulacja w Acell (wzory (1) i (2)):

```
if (mask[tmpidx] == 1)
{
int patternpix = pattern[tmpidx];
int imagepix = imagebuff[thidx + shx];
int diff = patternpix — imagepix;
if (diff < 0) diff=-diff; //uwzględniono tylko moduł we wzorze (2)

Acell += diff; //pojedyncza akumulacja dla wzoru (1)
}
```

Synchronizacja jest niezbędna, gdyż na początku kolejnej iteracji głównej pętli następuje ładowanie danych do bufora imagebuff:

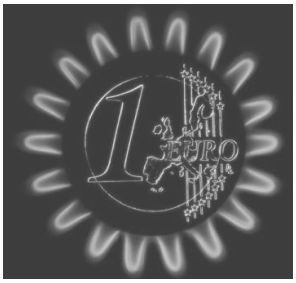
```
__syncthreads();
} //for — koniec
```

Ostatnim krokiem, wykonywanym przez każdy wątek, jest skopiowanie wartości Acell pod odpowiedni offset tablicy wyjściowej A:

```
if (picx + thidx + sh_size — 1 < w) A[picy * (w — sh_size + 1) + picx + thidx]
= Acell;
} // Hough
```

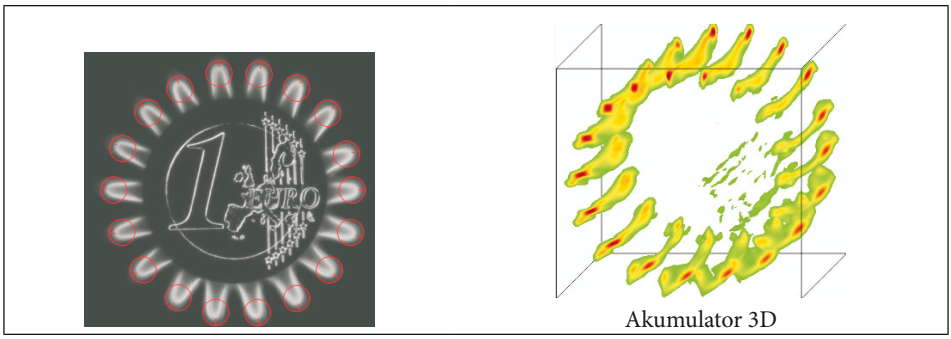
6. Weryfikacja działania

Obraz wejściowy $I_G(x, y)$ oraz wzorzec M_p



Wzorzec M_p (powiększony 3×)

Wynik segmentacji (wykryte obiekty) oraz akumulator



Rys. 5. Wynik działania implementacji CUDA dla przykładowego obrazu i wzorca

Na rysunku 5 przedstawiony został wynik segmentacji przykładowego obrazu wejściowego (o wymiarach 357×338 pikseli) wzorcem nieregularnym (o rozmiarze 32×32). Ponieważ przyjęto L = 36, więc wzorce obrócone tworzone były dla kwantu $\Delta\alpha = 10$ stopni. Identyczny wynik (akumulator) osiągnięty został przy zastosowaniu oprogramowania działającego tylko na CPU komputera, co potwierdza poprawną implementację CUDA. Przy zastosowaniu technologii CUDA i układu GTX 460 obliczenia trwały 515 ms, co oznacza 80-krotne ich przyspieszenie w porównaniu z CPU (kod zarządzany), gdzie osiągnięto czas obliczeń 40,8 s.

7. Podsumowanie

Zasadniczym celem niniejszego opracowania było ukazanie możliwości wykorzystania potencjału technologii CUDA na wybranym przykładzie z zakresu widzenia komputerowego. Zdaniem autorów wiele metod (już dawno opracowanych) z zakresu widzenia komputerowego zyska na znaczeniu dzięki nowym możliwościom technologii komputerowej, a transformata Hougha jest tutaj dobrym przykładem. Trudno ocenić, czy układy GPU nie stracą na znaczeniu wraz z rozbudową układów CPU, w których strukturze zaczęto właśnie integrować procesory graficzne[12] obok standardowych rdzeni. W najbliższych latach pozycja silnych GPU raczej nie będzie zagrożona, a nowe układy (Kepler, Maxwell) wspierające technologię CUDA są oczekiwane z niecierpliwością przez środowiska naukowe i zwykłych użytkowników komputerów. Technologia CUDA jest obecnie rozwijana bardzo dynamicznie, dla różnych środowisk programistycznych.

Artykuł wpłynął do redakcji 25.03.2011 r. Zweryfikowaną wersję po recenzji otrzymano w maju 2011 r.

[12] Czterordzeniowy procesor Core i5 2500K (dostępny od stycznia 2011) posiada zintegrowany układ graficzny HD3000 z 12 SPU (ang. *Streaming Processing Unit*).

LITERATURA

[1] D. H. Ballard, *Generalizing the Hough Transform to Detect Arbitrary Shapes*, Readings in Computer Vision: Issues, Problems, Principles, and Paradigms, Los Altos, CA, 1987, 714-725.

[2] W. Bi, Z. Chen, L. Zhang, Y. Xing, *Fast Detection of 3D Planes by a Single Slice Detector Helical CT*, Nuclear Science Symposium Conference Record (NSS/MIC), IEEE, 2009, 954-955.

[3] W. Bi, Z. Chen, L. Zhang, Y. Xing, Y. Wang, *Real-Time Visualize the 3D Reconstruction Procedure Using CUDA*, Nuclear Science Symposium Conference Record (NSS/MIC), IEEE, 2009, 883-886.

[4] E. R. Davies, *Machine Vision: Theory, Algorithms, Practicalities*, Third Edition, Morgan Kaufmann Publishers, 2005.

[5] S. R. Deans, *Hough transform from the Radon transform*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 3, 2, 1981, 185-188.

[6] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, N. Guil, *Parallelization of a Video Segmentation Algorithm on CUDA-Enabled Graphics Processing Units*, Euro-Par 2009 Parallel Processing, Lecture Notes in Computer Science, 5704, 2009, 924-935.

[7] P. V. C. Hough, *Method and means for recognizing complex patterns*, U.S. Patent 3,069,654, Dec. 18, 1962.

[8] J. Illingworth, J. Kittler, A *survey of the Hough Transform*, Computer Vision, Graphics and Image Processing, 44, 1988, 87-116.

[9] A. Kanezaki, H. Nakayama, T. Harada, Y. Kuniyoshi, *High-speed 3D Object Recognition using Additive Features in a Linear Subspace*, IEEE International Conference on Robotics and Automation (ICRA 2010), 2010, 3128-3134, http://www.isi.imi.i.u-tokyo.ac.jp/~kanezaki/ICRA2010WeD116.pdf.

[10] B. Kirk, W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, NVIDIA Corporation, Morgan Kaufmann Publishers, 2010.

[11] N. Kiryati, Y. Eldar, A. M. Bruckstein, A *probabilistic Hough transform*, Pattern Recognition, 24, 4, 1991, 303-316.

[12] V. F. Leavers, *Shape Detection in Computer Vision Using the Hough Transform*, Springer, London, 1992.

[13] H. Li, M. A. Lavin, R. J. LeMaster, *Fast Hough transform*, Proceedings of the Third Workshop on Computer Vision: Representation and Control (Bellaire, MI, October 13-16, 1985), IEEE Publ. 85CH2248-3, 75-83.

[14] H. Li, M. A. Lavin, R. J. Lemaster, *Fast Hough transform: a hierarchical approach*, Computer Vision, Graphics, and Image Processing, 36, 1986, 139-161.

[15] K. Nakamura, K. Arimura, T. Yoshikawa, *Recognition of object orientation and shape by a rotation spreading associative neural network*, Neural Networks 2001: Proceedings of IJCNN '01, 1, 2001, 565-570.

[16] P. C. Pedersen, J. D. Quartararo, T. L. Szabo, *Segmentation of speckle-reduced 3D medical ultrasound images*, IEEE International Ultrasonics Symposium (IUS), 2008, 361-366.

[17] J. Sanders, E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, NVIDIA Corporation, Addison-Wesley, 2010.

[18] M. Sonka, V. Hlavac, R. Boyle, *Image Processing: Analysis and Machine Vision*, Third Edition, Thompson Learning, 2008.

[19] L. Xu, E. Oja, P. Kultanen, A *new curve detection method: Randomized Hough Transform (RHT)*, Pattern Recognition Letters, 11, 5, 1990, 331-338.

[20] W. Żorski, B. Foxon, J. Blackledge, M. Turner, *Irregular Pattern Recognition Using the Hough Transform*, Machine Graphics & Vision, 9, 2000, 609-632.

[21] W. Żorski, *Application of the Hough Technique for Irregular Pattern Recognition to a Robot Monitoring System*, Proceedings of the 11th IEEE International Conference MMAR, 2005, 725-730.

[22] W. Żorski, *The Hough Transform Application Including Its Hardware Implementation*, Advanced Concepts for Intelligent Vision Systems: Proceedings of the 7th International Conference, Lecture Notes in Computer Science, Springer-Verlag, 3708, 2005, 460-467, http://www.springerlink.com/content/50yk3q0fw71x1qld.

[23] W. Żorski, *Fast Hough Transform Based on 3D Image Space Division*, Advanced Concepts for Intelligent Vision Systems: Proceedings of the 8th International Conference, Lecture Notes in Computer Science, Springer-Verlag, 4179, 2006, 1071-1079, http://www.springerlink.com/content/6216256332x1p166.

[24] W. Żorski, *Unknown scale objects recognition*, Biul. WAT, 4, 2008, 197-207.

[25] W. Żorski, P. Samsel, *Segmentacja obrazów kolorowych wzorcami nieregularnymi*, Biul. ITA, 26, 2009, 45-64, http://www.ita.wat.edu.pl/.

[26] W. Żorski, *Koncepcja rozpoznawania orientacji obiektów w obrazach 3D*, Biul. ITA, 28, 2010, 3-21, http://www.ita.wat.edu.pl/.

Źródła elektroniczne:

[27] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide*, NVIDIA Corporation 2007, http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

W. ŻORSKI, M. MAKOWSKI

Use of CUDA technology in area of irregular pattern recognition

Abstract. An implementation of an irregular pattern recognition technique with the use of the CUDA technology is presented in the paper. The potential of the contemporary NVIDIA's graphics processing units based on the Fermi architecture is emphasized. Basic rules of the CUDA programming are described. The Hough method for irregular patterns segmentation, as suitable for the implementation, has been chosen. Parts of the written program crucial to the CUDA technology are explained. The implementation has been verified for the sake of speed and correctness.

Keywords: CUDA technology, parallel programming, Hough transform, computer vision, irregular patterns, segmentation