Łukasz MACIURA

Uniwersytet Rzeszowski, Katedra Informatyki

Dyskretyzacja z nadzorem tablic danych przy użyciu wielordzeniowego procesora karty graficznej (GPU)

Streszczenie. *Niniejszy artykuł opisuje opracowany algorytm do dyskretyzacji tablic, polegający na masowym zrównolegleniu wyliczania optymalnego ciecia, poprzez jednoczesne badanie bardzo wielu atrybutów za pomocą wielordzeniowego procesora karty graficznej (GPU) oraz procesora (CPU). Jest to możliwe dzięki zastosowaniu technologii NVIDIA CUDA. Artykuł również porównuje prędkość działania tradycyjnego i zrównoleglonego algorytmu*

Abstract*. This paper describes the developed algorithm for discretization of arrays, consisting of a mass parallelization of calculating the optimal cut by simultaneous examination of a large number of attributes using a multi-core graphics card processor (GPU) and central processing unit (CPU). This is possible by using NVIDIA CUDA technology. Paper also compares the speed of traditional and parallelised algorithm (**Supervised discretization of data arrays using multicore GPU**).*

Słowa kluczowe: dyskretyzacja, bioinformatyka, zrównoleglenie, *CUDA*
Keywords: discretization, bioinformatics, parallelization, *CUDA*

doi:10.12915/pe.2014.05.26

Wstęp

Dyskretyzacja z nadzorem jest ważną metodą eksploracji danych, potrzebną miedzy innymi do tworzenia klasyfikatorów. Istnieją algorytmy, które działając sekwencyjnie wyliczają cięcie dyskretyzacji w czasie rzędu *O(m *n * log n)*, gdzie *n* to liczba obiektów i *m* to liczba atrybutów warunkowych. Działanie tych algorytmów polega na zachłannym wyznaczeniu najlepszego cięcia na danym etapie działania algorytmu, przy czym dla jego wyznaczenia trzeba każdy z atrybutów badać algorytmem o złożoności rzędu *O(n * log n)*.

Istnieje potrzeba praktycznych zastosowań związana z dyskretyzacją tablic danych o bardzo dużej liczbie atrybutów. Na przykład analiza danych bioinformatycznych dotyczących ekspresji genów wymaga analizy danych liczących często ponad 60000 atrybutów. Z powodu powolnej pracy dotychczasowych algorytmów dla takich tablic istnieje potrzeba przyspieszenia ich działania.

Niniejszy artykuł opisuje opracowany algorytm do dyskretyzacji tablic, polegający na masowym zrównolegleniu wyliczania optymalnego ciecia, poprzez jednoczesne badanie bardzo wielu atrybutów za pomocą wielordzeniowego procesora karty graficznej (GPU) w technologii *NVIDIA CUDA* (*ang. Compute Unified Device Architecture*). W artykule porównano również prędkość działania tradycyjnego i zrównoleglonego algorytmu dla przykładowych zbiorów danych.

Algorytm dyskretyzacji z nadzorem

Opisywane w tej pracy metody obliczeniowe służą do eksploracji zbiorów danych, przy czym dane są przygotowane w postaci prostokątnej tablicy zwanej tablicą decyzyjną, w której wiersze (zwane obiektami), reprezentują np. pacjentów, a kolumny (zwane atrybutami) reprezentują cechy lub parametry tych pacjentów (używamy tu aparatu pojęciowego stosowanego z ogólnie znanej teorii zbiorów przybliżonych zaproponowanej przez prof. Zdzisława Pawlaka [1]). Ostatnia kolumna w tablicy decyzyjnej jest specjalnym atrybutem, który nazywamy atrybutem decyzyjnym. Przechowuje on wartości związane z ustalonym problemem decyzyjnym, które zostały zaobserwowane dla danych pacjentów i mogą być wykorzystane do maszynowego uczenia się rozpoznawania takich wartości dla nowych pacjentów. Wartości atrybutu decyzyjnego nazywamy wartościami decyzji. Zbiór obiektów mających te same wartości decyzji nazywamy klasą decyzyjną. W praktycznych zastosowaniach, atrybut decyzyjny często ma tylko dwie wartości, co oznacza, że są tylko dwie klasy decyzyjne. Natomiast pozostałe atrybuty z tablicy decyzyjnej (poprzedzające atrybut decyzyjny) nazywamy atrybutami warunkowymi. Opisują one cechy pacjentów, które mogą być użyte do opisywania danych pacjentów (zwanych treningowymi pacjentami) mających określone wartości decyzji. Tego rodzaju opis (model) umożliwia późniejszą predykcję wartości decyzji dla nieznanych pacjentów (zwanych pacjentami testowymi), czyli predykcje przynależności pacjentów do poszczególnych klas decyzyjnych. Algorytm, który potrafi generować wartości atrybutu decyzyjnego dla pacjentów testowych na podstawie utworzonego modelu, w oparciu o wartości atrybutów warunkowych, nazywany jest klasyfikatorem. Tworzenie klasyfikatora odbywa się na podstawie automatycznej analizy dostępnej informacji o danych pacjentach, z których każdy reprezentowany jest za pomocą odpowiadającego mu wektora atrybutów warunkowych wraz z wartością atrybutu decyzyjnego. Dla przykładu załóżmy, że w tablicy decyzyjnej mamy 4 kolumny (atrybuty) *a, b, c i d*, z których kolumna *d* jest atrybutem decyzyjnym mającym dwie wartości *c1 i c2*. Załóżmy także, że na podstawie analizy tablicy stwierdzono, że w tej tablicy prawdziwa jest reguła decyzyjna *(a=2) AND (b<3) to (d=c1)* to oznacza, że dla każdego pacjenta, jeśli wartość dla tego pacjenta atrybutu a jest równa 2 i jednocześnie wartość atrybutu *b* jest mniejsza od 3, to zawsze pacjent ten należy do klasy decyzyjnej związanej z wartością decyzji *c1*. Reguła taka może być wykorzystywana do klasyfikowania (przypisywania) pacjentów testowych (nieznanych podczas wyszukiwania reguły), którzy zostali przez tę regułę rozpoznani, czyli dla których wartość atrybutu *a=2* i wartość atrybutu *b<3*. Oczywiście w przypadku, gdy pacjent testowy rozpoznany jest przez więcej niż jedną regułę, które klasyfikują go do różnych klas decyzyjnych, powstaje konflikt, który musi zostać rozstrzygnięty za pomocą określonej strategii. Jednak w przypadku drzew decyzyjnych wykorzystywanych w tej pracy, reguły decyzyjne (reprezentowane za pomocą drzewa) są tak konstruowane, że pacjent testowy może być rozpoznany tylko przez jedną regułę.

Jednym z największych wyzwań związanym z utworzenia efektywnego w praktyce klasyfikatora, jest skonstruowanie odpowiednich cech pacjentów (atrybutów warunkowych), które pozwolą na utworzenie takich modeli (opisów) pacjentów należących do odpowiednich klas decyzyjnych, które umożliwią poprawną klasyfikację nowych pacjentów na przynależność do odpowiednich klas decyzyjnych. Istnieje wiele metod konstruowania takich

cech oraz wiele metod późniejszej ich analizy celem skonstruowania klasyfikatora. Stosujemy tu podejście oparte na tworzeniu tzw. drzewa decyzyjnego lokalnej dyskretyzacji [2,3]. Takie drzewo tworzone jest za pomocą podziałów danego zbioru danych na dwie grupy obiektów (pacjentów) za pomocą wartości wybranego atrybutu. Np. dla atrybutu *a* o wartościach symbolicznych (nienumerycznych, z niewielką liczbą wartości) podział pacjentów może odbywać się za pomocą jakiegoś atrybutu *a* oraz jego wartości *v* w taki sposób, że do jednej grupy należą pacjenci którzy mają wartość atrybutu *a* równą *v*, a do drugiej grupy należą pacjenci którzy mają wartość atrybutu *a* różną od *v*. Natomiast dla atrybutu *b* o wartościach numerycznych (z dużą liczbą wartości) podział pacjentów może odbywać się za pomocą jakiegoś atrybutu *b* oraz jego wartości *v* w taki sposób, że do jednej grupy należą pacjenci którzy mają wartość atrybutu *b* większą lub równą *v*, a do drugiej grupy należą pacjenci którzy mają wartość atrybutu *b* mniejszą od *v*. Sposób wybrania atrybutu oraz jego wartości (zwanej cięciem), które wykorzystujemy do podziału jest kluczowym elementem omawianej metody budowy klasyfikatora i powinien wiązać się z analizą wartości atrybutu decyzyjnego dla pacjentów treningowych. Jako miarę jakości ciecia, można np. wykorzystywać liczbę par pacjentów rozróżnianych przez cięcie i mających różne wartości atrybutu decyzyjnego. Np. jeśli pewne cięcie *c* (cięcie to wartość dla jakiegoś wybranego atrybutu) dzieli pacjentów na dwie grupy o liczebności *M* i *N* oraz w pierwszej z tych grup mamy *M0* i *M1* pacjentów odpowiednio z klasy decyzyjnej *C0* i *C1*, a w drugiej grupie mamy *N0* i *N1* pacjentów odpowiednio z klasy decyzyjnej *C0* i *C1*, to liczba par pacjentów rozróżnianych przez cięcie *c* wynosi: *M0*N1+M1*N0*. Jeśli któraś grupa po podziale zbioru wejściowego nie jest czysta, tzn. zawiera pacjentów zarówno z klasy decyzyjnej 0 jak i 1, to opisany wyżej proces podziału musi zostać powtórzony, ale tylko dla tej grupy. W ten sposób tworzona jest hierarchiczna struktura drzewa binarnego, w którego wierzchołkach znajdują się atrybuty i ich wartości wykorzystane do podziału. Na danym poziomie hierarchii drzewa każdy wierzchołek jest połączony z dwoma wierzchołkami niższego poziomu drzewa. Dane odgałęzienie drzewa kończy się w liściu, który zawiera pacjentów tylko z jednej klasy decyzyjnej (jest czysty). Wytworzone w ten sposób binarne drzewo decyzyjne reprezentuje szereg reguł decyzyjnych, które mogą być wykorzystane do klasyfikacji pacjentów.

Łatwo zauważyć, że w opisanej wyżej procedurze tworzenia drzewa decyzyjnego lokalnej dyskretyzacji, miejscem krytycznym jest wyszukiwanie optymalnego cięcia rozdzielającego zbiór danych na danym etapie obliczeń. Wyznaczenie takiego cięcia wymaga przebadania potencjalnych cieć ze wszystkich atrybutów warunkowych oraz zachłannego wybrania jednego optymalnego cięcia.
W tym celu każdy z atrybutów warunkowych może być badany algorytmem o złożoności rzędu *O(n * log n)*, gdzie *n* jest liczbą obiektów w tablicy. Algorytm ten po prostu sortuje obiekty (wiersze) względem wartości ustalonego atrybutu, po czym przegląda poszczególne wiersze zawierające posortowane wartości atrybutu, wyznaczając miary jakości wszystkich potencjalnych cięć dla badanego atrybutu [2].

Po wyznaczeniu całego drzewa decyzyjnego lokalnej dyskretyzacji, można odczytać wszystkie atrybuty i ich wartości pojawiające się w poszczególnych węzłach drzewa. Dzięki czemu uzyskujemy narzędzie dyskretyzacji (skalowania) wartości atrybutów warunkowych tablicy decyzyjnej. Dla przykładu, jeśli dla numerycznego atrybutu *a* w drzewie pojawiły się węzły z wartościami *v1* i *v2* (*v1*<*v2*), to informacja ta może być użyta do takiego przeskalowania wartości atrybutu *a*, że obiekty tej tablicy będą miały tylko trzy wartości *s1*, *s2*, *s3*, przy czym wartość *s1* odpowiada sytuacji, gdy wartość *a*<*v1*, wartość *s2* odpowiada sytuacji, gdy wartość *a*>=*v1* i *a*<*v2*, oraz wartość *s3* odpowiada sytuacji, gdy wartość *a*>=*v2*.

W przypadku niewielkiej liczby atrybutów tablicy decyzyjnej (np. do kilkuset atrybutów) opisany wyżej algorytm wyznaczana optymalnego cięcia działa bardzo szybko. Jednak w przypadku bardzo dużej liczby atrybutów warunkowych jego szybkość ulega znacznemu spowolnieniu. Sytuacja taka ma miejsce w przypadku danych bioinformatycznych w których często występuje bardzo mała liczba wierszy (maksymalnie kilkaset) oraz bardzo duża liczba atrybutów (np. ponad 60000 atrybutów).

Dane biomedyczne wykorzystane w badaniach

W pracy wykorzystano trzy zbiory danych bioinformatycznych pozyskanych z użyciem mikromacierzy *DNA*, które będziemy nazywać: *ATC, GPE i SPS*. Zbiory te były używane do eksperymentów w pracy doktorskiej [4] i dotyczą problemów decyzyjnych związanych z nowoczesnymi metodami leczenia opartymi na badaniu ekspresji genów. Rozmiary tych zbiorów danych są następujące. Zbiór *ATC* ma *161* wierszy i *61359* kolumn, zbiór *GPE* ma *247* wierszy i *54675* kolumny, natomiast zbiór *SPS* ma *180* wierszy i *54675* kolumn. Atrybuty decyzyjne wszystkich powyższych zbiorów danych mają po dwie wartości.

Technologia NVIDIA CUDA

Technologia *NVIDIA CUDA* [5,6] jest oparta na wielordzeniowym procesorze karty graficznej (*GPU*), który służy do równoległego wykonywania programów. Programy w technologii *CUDA* (w tym tzw. funkcje jądra które są wykonywane równolegle na *GPU*) pisane są w języku *CUDA C*.

Program korzystający z funkcji jądra wykonywanego równolegle zazwyczaj składa się z: inicjalizacji pamięci w karcie graficznej, przesłania danych wejściowych do pamięci karty graficznej, wywołania funkcji jądra, odczytania wyników działania funkcji jądra z pamięci *GPU*, ewentualnych obliczeń końcowych bez użycia *GPU*. Istotnymi elementami wywołania funkcji jądra są tzw. bloki i wątki. Bloki są to równoległe kopie funkcji jądra które można dzielić na wątki. W efekcie można uruchomić jednocześnie nawet miliony wątków – co może uprościć zadanie algorytmiczne. Aby w pełni wykorzystać potencjał obliczeniowy *CUDA* całkowita liczba wątków (zgrupowanych we wszystkich blokach) powinna być co najmniej równa ilości rdzeni *GPU*.

Wąskim gardłem obliczeń równoległych w technologii *CUDA* jest dostęp do pamięci. Algorytm korzystający bardzo często z pamięci będzie działał wolniej niż algorytm wykonujący głównie obliczenia. Jednym ze sposobów by przyśpieszyć działanie takiego algorytmu jest wykorzystanie szczególnych rodzajów pamięci o szybszym dostępie niż pamięć *RAM* karty graficznej są to pamięci:

- pamięć stała (*ang. constant*) – pamięć do zapisywania danych tylko do odczytu przed wywołaniem funkcji jądra – jej całkowita ilość w programie jest ograniczona do 64 kilobajtów)
- pamięć wspólna (ang. *shared*) – pamięć dostępna tylko dla wątków wewnątrz danego bloku. Jest ona zapisana w układzie *GPU*, jest wykorzystywana między innymi do obliczania i zapisywania wyników cząstkowych dla każdego bloku.
- pamięć bitmap – jest to szczególny rodzaj pamięci która oferuje szybszy dostęp tylko w szczególnych przypadkach

(gdy wątki odczytują/zapisują dane znajdujące się w pobliżu danych odczytywanych lub zapisywanych przez sąsiednie wątki)

Ważnym elementem programów wykorzystujących *GPU* jest synchronizacja wątków. Wykorzystana jest tu między innymi wspomniana wcześniej pamięć wspólna. Synchronizacja wątków jest przydatna na przykład do obliczania wyników cząstkowych wewnątrz każdego bloku. Niestety przynosi negatywne konsekwencje w postaci spadku szybkości działania funkcji jądra.

Równoległy algorytm dyskretyzacji z nadzorem

W wyniku przeprowadzonych badań zauważono że najwięcej czasu w algorytmie obliczania optymalnego cięcia zajmuje sortowanie w poszczególnych kolumnach a zatem tu należy przede wszystkim szukać sposobów przyśpieszenia algorytmu. Źródła literaturowe [5,6,7] podają że wąskim gardłem obliczeń na *GPU* jest dostęp do pamięci, więc korzystając z pamięci szybszego dostępu można przyśpieszyć czas obliczeń.

W wyniku serii eksperymentów opracowano następującą metodę. Wprowadzono dwie funkcje jądra. Jedna zajmuje się sortowaniem, natomiast druga oblicza optymalne cięcie dla każdej z kolumn. Sortowanie odbywa się kaskadowo 2 metodami. Najpierw każda kolumna jest dzielona na 16 części z których każda jest osobno sortowana algorytmem bąbelkowym. Wątki są rozdzielone w taki sposób, że bloki obsługują kolumny, natomiast wątki wewnątrz bloków, których jest 16 obsługują poszczególne cząstki sortowanej kolumny. Dane kolumny oraz tablicy indeksów która jest sortowana względem danych z kolumny zostają umieszczone w pamięci wspólnej (do której jest szybszy dostęp niż do pamięci *RAM* karty graficznej) do której mają dostęp wątki wewnątrz bloku. Gdy dane wewnątrz kolumny zostaną w ten sposób posortowane (wewnątrz każdej z 16 cząstek) następują kaskadowo scalenia po 2 cząstki do 8 cząstek, następnie do 4, 2, aż zostanie posortowana cała kolumna. Podczas scalania zostaje systematycznie ograniczana liczba działających wątków aż do jednego co powoduje spadek przyśpieszenia (wynikający również z synchronizacji wątków wewnątrz bloków), ale jest on możliwy do zaakceptowania. Warto w tym miejscu podkreślić, że scalanie mniejszych tablic jest nieopłacalne ze względu na koszty wynikające z synchronizacji i redukcji wątków (liczba 16 cząstek została wybrana eksperymentalnie), z tego również względu nie wykonano sortowania w całości metodą scalania, lecz połączono 2 różne metody sortowania.

Aby było możliwe uczciwe porównanie czasów działania na *CPU* i *GPU* oraz obliczenie przyśpieszenia algorytm na *CPU* został zaimplementowany w identyczny sposób, tyle że w sposób sekwencyjny sortuje dane dwiema metodami oraz oblicza optymalne cięcie osobno dla każdej kolumny.

Na początku programu porównującego algorytm działający w sposób tradycyjny (sekwencyjny) oraz algorytm zrównoleglony poniższa funkcja rekurencyjnie generuje rozmiary poszczególnych cząstek tablic do scalania oraz indeksy tych cząstek wewnątrz kolumny danych. Rozmiary te i indeksy są następnie odczytywane z tablic przez odpowiadające sobie algorytmy działające na *CPU* oraz na *GPU* (wykonywane są odpowiednie kopie tych tablic do pamięci stałej karty graficznej aby były szybciej odczytywane przez GPU). Dzięki temu rozwiązaniu funkcje sortujące cząstki kolumn nie muszą używać rekurencji tylko odczytują odpowiednie indeksy i rozmiary cząstek (możliwe jest to tylko dzięki temu że każda kolumna danych ma taki sam rozmiar – parametr *size* funkcji). Przy wywołaniu tej funkcji parametry *2 – 4* muszą być ustawione na zero.

```
void genSizes(int size, int deep, int ind,
int indstep){
     partsizes[ind] = size;
     partindexes[ind] = indstep;
     if(deep<4){
        genSizes(size/2,deep+1,ind*2+1,indstep);
         genSizes(size- size/2, deep+1,ind*2+2,
         indstep+size/2);
}}
```

Poniżej przedstawiono funkcje sortowania bąbelkowego dla cząstek oraz scalania wywoływane przez funkcję jądra sortującą dane (analogicznie wyglądają odpowiadające im funkcje dla *CPU*)

```
__device__ inline void SortBabelGPU(int start,
int nrows, float *dataSetCol, int *locArray) {
  int i,j,p;
  for(j = start + nrows - 1; j > start; j--){
    p = 1;
    for(i = start; i < j; i++)
    if(dataSetCol[locArray[i]]
            > dataSetCol[locArray[i + 1]]){
       int temp = locArray[i];
       locArray[i] = locArray[i+1];
       locArray[i+1] = temp;
        p = 0;
     }
    if(p) break;
}}

__device__ inline void ScalGPU(int startl, int
startp, int nl, int np, float *dataSetcol,
int *locArray, int *locArray2){
   int indl = startl;
   int indp = startp;
   int ind = startl;
   while((indl < (startl+nl))
              && (indp < (startp+np))){
    if(dataSetcol[locArray[indl]]
              <dataSetcol[locArray[indp]])
       locArray2[ind++] = locArray[indl++];
    else
       locArray2[ind++] = locArray[indp++];
   }
   while(indl < (startl+nl))
       locArray2[ind++] = locArray[indl++];
   while(indp < (startp+np))
       locArray2[ind++] = locArray[indp++];
}
```

Poniżej przedstawiono funkcję jądra dla sortowania obejmującą sortowanie bąbelkowe a następnie scalanie. Wynikiem działania tej funkcji jest uporządkowanie tablicy indeksów względem danych która jest później wykorzystana do obliczania optymalnego cięcia dla każdej kolumny. Wewnątrz tej funkcji stworzono 3 tablice w pamięci wspólnej (*shared*): 2 pomocnicze tablice indeksów do sortowania obraz tablicę z danymi typu *double*. Przeprowadzone badania wykazały że użycie pamięci wspólnej przyśpiesza znacznie działanie obliczeń na *GPU*. Na początku działania te tablice są wypełniane danymi przez wszystkie wątki w bloku a następnie następuje właściwe sortowanie. Pętla *while* sprawia, że dany blok wątków może obsłużyć wiele kolumn sekwencyjnie. Wątki wewnątrz bloku sortują dane wewnątrz kolumny w sposób opisany wcześniej.

Poszczególne warunki służą do stopniowego ograniczania liczby wątków bloku podczas scalania, ponieważ zmniejsza się liczba cząstek tablic które muszą obsłużyć. Pomiędzy tymi sortowaniami i scalaniami użyto synchronizacji wątków wewnątrz bloków aby zapobiec pracy na nieprzetworzonych danych w poprzednim kroku. Przykładowo wątek 1 i 2 scalają na początku 2 podtablice metodą bąbelkową a następnie wątek 1 dokonuje scalanie tych 2 tablic a wątek 2 kończy pracę dla danych z tej

kolumny. Gdyby wątek 1 zakończył sortowanie wcześniej niż wątek 2 to mógłby zacząć scalać nieposortowane dane (które nadal sortowałby wątek 2). Dlatego użyto polecenia `__syncthreads()` które synchronizuje wątki wewnątrz bloku. Rozmiary i indeksy poszczególnych cząstek tablic są odczytywane z pamięci stałej gdzie zostały wcześniej skopiowane z pamięci *RAM* komputera (przed wywołaniem funkcji jądra).

```
extern "C"
__global__ void sortGPU(float *dataSet,
int *locMatrix){
 int ind = blockIdx.x;
 int tid = threadIdx.x;
 int ncols = csizes[0];
 int nrows = csizes[1];
 int ndecs = csizes[2];
 __shared__ int locArrayA[MAXNROWS];
 __shared__ int locArrayB[MAXNROWS];
 __shared__ float dataSetcol1[MAXNROWS];
 while (ind<ncols){
  float *dataSetcol = &dataSet[ind * nrows];
  int i;
  int n = cpartsizes[tid+15];
  int index = cpartindexes[tid+15];
  for(i=0;i<n;i++){
   locArrayA[index+i] = index+i;
   dataSetcol1[index+i] = dataSetcol[index+i];
  }
SortBabelGPU(cpartindexes[tid+15],
cpartsizes[tid+15],&dataSetcol1[0],
&locArrayA[0]);
__syncthreads();
if(tid<8){
 ScalGPU(cpartindexes[tid*2+15],
cpartindexes[tid*2+16],cpartsizes[tid*2+15],
cpartsizes[tid*2+16],&dataSetcol1[0],
&locArrayA[0], &locArrayB[0]);
__syncthreads();
if(tid<4){
 ScalGPU(cpartindexes[tid*2+7],
cpartindexes[tid*2+8],cpartsizes[tid*2+7],
cpartsizes[tid*2+8],&dataSetcol1[0],
&locArrayB[0], &locArrayA[0]);
__syncthreads();
if(tid<2){
ScalGPU(cpartindexes[tid*2+3],
cpartindexes[tid*2+4],cpartsizes[tid*2+3],
cpartsizes[tid*2+4],&dataSetcol1[0],
&locArrayA[0], &locArrayB[0]);
__syncthreads();
int *locArray = &locMatrix[ind * nrows];
if(tid==0){
ScalGPU(cpartindexes[1],cpartindexes[2],
cpartsizes[1],cpartsizes[2],&dataSetcol1[0],
&locArrayB[0], &locArray[0]);
__syncthreads();
}}}}
ind += gridDim.x;
}}
```

Po operacji sortowania wszystkich kolumn, zaczyna działanie kolejna funkcja jądra, która oblicza optymalne cięcie osobno w każdym wątku. Wprowadzenie drugiej funkcji jądra było konieczne, ponieważ nieopłacalne było obliczanie optymalnego cięcia dla każdej kolumny na zredukowanej podczas scalania ilości wątków, dlatego użyto drugiej funkcji jądra z działającymi nowymi wątkami.

Wyniki przyśpieszenia obliczeń

Przeprowadzono eksperymenty dla trzech zbiorów danych biomedycznych uruchamiając algorytm tradycyjny (na *CPU*) i zrównoleglony (na *GPU*). Dla algorytmu działającego na *CPU* obliczono czas sekwencyjnego działania szukania optymalnego cięcia dla wszystkich kolumn (wraz z sortowaniem) oraz czas wyszukiwania najlepszego atrybutu, natomiast dla *GPU* obliczono czas kopiowania danych wejściowych do pamięci karty graficznej i pamięci wspólnej, wywołania funkcji jądra (sortującej oraz obliczającej optymalną wartość cięcia), czas kopiowania wyników do pamięci *RAM* oraz dokończenia obliczeń na *CPU* (wyszukiwania najlepszego atrybutu). Czas dokończenia obliczeń na *CPU* w przypadku tego algorytmu jest bardzo krótki w porównaniu do całości obliczeń, dlatego zdecydowano się nie zrównoleglać go.

Badania przeprowadzono na komputerze z procesorem *Intel Core I3 – 2367M* o taktowaniu 1.4 *GHz*. Do obliczeń równoległych wykorzystano układ *NVIDIA GeForce GT 640M* z 384 rdzeniami *CUDA* i częstotliwością taktowania rdzenia graficznego 625 MHz. Dla każdego zbioru danych (*ATC*, *SPS* i *GPE*) przeprowadzono po 5 testów. Uśrednione za pomocą mediany czasy działania oraz przyśpieszenia pochodzące z tych 5 testów przedstawia Tabela 1. Przyśpieszenie było obliczane na podstawie stosunku czasu działania na *CPU* oraz na *GPU* (wraz z kopiowaniem danych wejściowych i wejściowych oraz dokończeniem obliczeń na *CPU*).

Tabela 1. Uśrednione (za pomocą mediany) czasy obliczeń i przyśpieszenia dla wszystkich rodzajów danych

Typ danych	**ATC**	**SPS**	**GPE**
czas CPU [ms]	2932	3058	4695
czas GPU [ms]	484	484	702
czas jądra [ms]	473,88	472,37	673,62
przyśpieszenie	6,03	6,25	6,69

Wnioski

W wyniku przeprowadzonych badań uzyskano przyśpieszenie działania algorytmu korzystającego z *GPU* w stosunku do algorytmu działającego na *CPU* dla 3 rodzajów danych bioinformatycznych na poziomie powyżej 6 razy. Warto tu zwrócić uwagę, że stosunkowo częste odwoływanie się do pamięci operacyjnej podczas obliczeń równoległych (co jest typowe dla tego rodzaju algorytmów) powoduje niższą wydajność tych obliczeń.

Opracowane oprogramowanie na platformę *CUDA* bardzo dobrze skaluje się na wydajniejsze procesory graficzne. W przypadku wykorzystania profesjonalnych kart *Tesla* lub kart w wersji desktopowej (np. *Tytan*) algorytm działałby zdecydowanie szybciej.

Badania naukowe, których wyniki zostały zaprezentowane w tym artykule, były wspierane przez projekt badawczy numer 2013/09/B/ST6/01568 Narodowego Centrum Nauki.

LITERATURA

[1] Pawlak Z., Skowron A., Rudiments of rough sets, *Information Sciences*, 177 (2007), 3–27.
[2] Bazan J. G., Nguyen H. S., Synak P., Wróblewski J.: Rough set algorithms in classification problems, in: L. Polkowski, T. Y. Lin, S. Tsumoto (Eds.), *Rough Set Methods and Applications: New Developments in Knowledge Discovery in Information Systems*, *Springer-Verlag/Physica-Verlag*, Heidelberg, Germany, *Studies in Fuzziness and Soft Computing*, vol. 56 (2000), 49–88.
[3] Nguyen H. S., Approximate Boolean Reasoning: Foundations and Applications in Data Mining, *Transactions on Rough Sets*, V, LNCS 4100 (2006), 334–506.
[4] Janusz A., Algorithms for Similarity Relation Learning from High Dimensional Data (Algorytmy uczenia się relacji podobieństwa z wielowymiarowych zbiorów danych), *praca doktorska Uniwersytet Warszawski* (2014)
[5] Sanders J., Kandrot E., CUDA w przykładach. Wprowadzenie do ogólnego programowania procesorów GPU, *Helion* (2012)
[6] CUDA Toolkit Documentation, *NVIDIA Corporation* (2013) http://docs.nvidia.com/cuda/index.html
[7] Satish N., Harris M., Garland M., Designing Efficient Sorting Algorithms for Manycore GPUs, *NVIDIA Corp.* (2008)

***Autor**: dr inż. Łukasz Maciura, Uniwersytet Rzeszowski, Katedra Informatyki, ul. Pigonia 1, Rzeszów, E-mail: lmaciura@ur.edu.pl;*