Politechnika Poznańska
Wydział Informatyki i Zarządzania
Instytut Informatyki

Opracowanie, implementacja i przetestowanie algorytmu wykorzystującego procesor karty graficznej do odszukiwania zbiorów częstych

PRACA MAGISTERSKA

Kierunek: Automatyka i Zarządzanie
Autor: inż. Mikołaj Łosiak
Promotor: dr inż. Witold Andrzejewski

Poznań 2009

1. Wstęp

Prostota konstruowania baz danych oraz akceptowalny poziom cen systemów bazodanowych przyczyniają się do popularyzacji systemów informatycznych gromadzących dane. Znajdują one zastosowanie w wielu dziedzinach życia, od techniki przez medycynę, astronomię i ekonomię po szeroko rozumiany biznes.

Człowiek, który chciałby bez zastosowania wspomagających narzędzi informatycznych odkryć zależności istniejące między zgromadzonymi danymi, musiałby poświęcić na to bardzo dużo czasu. Przy złożonych systemach koszt czasowy jest na tyle duży, że ręczna analiza danych staje się niemożliwa.

Aby ułatwić proces wyszukiwania zależności między informacjami zgromadzonymi w bazie danych powstały algorytmy eksploracji danych, które automatyzują to zadanie. Najbardziej powszechne są rozwiązania wyszukujące zbiory częste, tworzące reguły asocjacyjne, wzorce sekwencji, reguły decyzyjne oraz korzystające z sieci neuronowych lub grupujące dane.[1]

Szczególną wartość praktyczną mają reguły asocjacyjne. Pozwalają one w jasny sposób określić zależności między badanymi obiektami, a algorytmy stosowane do ich znajdowania są stosunkowo proste.[2] Duża wartość użytkowa reguł asocjacyjnych sprawia, że są one chętnie stosowane np. do przeprowadzania tak zwanej „analizy koszykowej”[2], czyli badania powiązań między produktami kupowanymi przez klientów danego sklepu.

Przykładem reguły asocjacyjnej odkrytej w bazie danych supermarketu spożywczego może być reguła $mleko \wedge płatki \rightarrow kakao$. Reguła ta reprezentuje fakt, że klienci kupujący mleko i płatki, z dużym prawdopodobieństwem nabędą również kakao.[3]

Algorytmy wyszukujące reguły asocjacyjne są bardzo wolne. Jest to bezpośrednią przyczyną podjęcia badań nad opracowaniem rozwiązania szybszego. Jedną z możliwości przyspieszenia działania algorytmu jest zrównoleglenie operacji, które on wykonuje.

W niniejszej pracy przedstawiono implementację algorytmu, w której operacje zostały zrównoleglone przy użyciu procesora karty graficznej.

1.1. Struktura pracy

W rozdziale 2 przedstawiono charakterystykę wybranych algorytmów dotychczas stosowanych w eksploracji danych.

Rozdział 3 omawia możliwości zrównoleglenia obliczeń przy wykorzystaniu karty graficznej

Rozdział 4 zawiera definicje pojęć używanych w dalszej części pracy.

Rozdział 5 to opis implementacji opracowanych algorytmów.

Rozdział 6 przedstawia plan eksperymentów oraz omawia ich wyniki.

Rozdział 7 stanowi podsumowanie pracy.

1.2. Cel i zakres pracy

Celem niniejszej pracy jest opracowanie algorytmu wyszukiwania zbiorów częstych za pomocą procesora karty graficznej. Na cel składają się następujące cele szczegółowe:

1. Zapoznanie się z literaturą na temat eksploracji danych, a w szczególności odkrywania zbiorów częstych.
2. Zapoznanie się z literaturą dotyczącą technik GPGPU (General Processing on GPU).
3. Opracowanie algorytmu odkrywania zbiorów częstych wykorzystującego procesor karty graficznej.
4. Implementacja opracowanego algorytmu na GPU i CPU.
5. Implementacja algorytmu FP-Growth na CPU. Z celu zrezygnowano decyzją promotora ze względu na czasochłonność prac związanych z implementacją algorytmu równoległego na GPU.
6. Przeprowadzenie eksperymentów porównujących wydajność zaimplementowanych algorytmów.
7. Opracowanie wyników eksperymentów.

2. Dotychczasowe osiągnięcia

W rozdziale omówiono dwa wybrane algorytmy wyszukiwania zbiorów częstych – Apriori oraz FP - growth. Przedstawiono ideę opisywanych rozwiązań oraz nakreślono ich słabe strony.

2.1. Algorytm *Apriori*

Algorytm *Apriori* jest pierwszym, który wykorzystał własność monotoniczności miary wsparcia do ograniczenia przestrzeni poszukiwań zbiorów częstych. Jest algorytmem iteracyjnym, który w kolejnych krokach szuka zbiorów 1,2,3, ..., k - elementowych.

2.1.1. Opis algorytmu

Pierwszym krokiem jest wyodrębnienie z bazy danych wszystkich zbiorów jednoelementowych oraz sprawdzenie, które z nich są częste (posiadają wsparcie co najmniej *minsup*). W kolejnym kroku w oparciu o jednoelementowe zbiory częste algorytm generuje dwuelementowe zbiory kandydujące. Potencjalnie każdy z wygenerowanych zbiorów może być zbiorem częstym. Dla każdego wygenerowanego zbioru algorytm oblicza wsparcie. Jeżeli obliczona wartość jest większa lub równa minimalnej przyjętej mierze wsparcia, zbiór trafia do puli zbiorów częstych i zostanie użyty w kolejnej iteracji do generowania zbiorów kandydujących trzyelementowych. W kolejnych krokach zbiory częste trzyelementowe zostaną użyte do generowania zbiorów kandydujących czteroelementowych, czteroelementowe do generowania pięcioelementowych itd. Proces ten będzie trwać do czasu, kiedy nie można już będzie wygenerować kolejnych zbiorów kandydujących. Istotny jest fakt, że w celu obliczenia wsparcia zbiorów kandydujących, w każdym kroku algorytmu należy odczytać całą bazę danych. Wynikiem działania algorytmu jest suma k – elementowych zbiorów częstych (k = 1,2,3...).

Formalny zapis algorytmu przedstawiono w sekcji 2.1.2.

2.1.2. Zapis algorytmu[4]

L_1 = {zbiory częste 1 – elementowe};

for (k = 2; $L_{k-1} \neq \emptyset$; k++) do

begin
    C_k = **apriori_gen**(L_{k-1});
    For each trancakcji t∈T **do**
    begin
        C_t = **subset**(C_k, t);
        For each zbioru kandydującego c∈C_t **do**
        c.count++;
    end;

$L_k = \{c \in C_k \mid c.count \geq minsup\}$

end;

Wynik = $\cup k L_k$;

Algorytm 1.1. Algorytm Apriori

W opisie zastosowano następujące oznaczenia:

C_k - rodzina zbiorów kandydujących k – elementowych,

L_k - rodzina zbiorów częstych k – elementowych,

c.count – licznik zliczający liczbę transakcji wspierających zbiór elementów c,

apriori_gen() – funkcja generująca zbiory kandydujące,

subset() – funkcja, która dla danej transakcji zwraca wszystkie zbiory kandydujące wspierane przez t.

Funkcja apriori_gen

function apriori_gen(C_k)

insert into C_k

select $p.item_1$, $p.item_2$, ...,

$p.item_{k-1}$, $q.item_{k-1}$

from L_{k-1} p, L_{k-1} q

where $p.item_1 = q.item_1$, ...,

$p.item_{k-2} = q.item_{k-2}$

$p.item_{k-1} = q.item_{k-1}$;

forall itemsets $c \in C_k$ **do**

forall (k-1) – subsets s **of** c **do**

if ($s \in L_{k-1}$) **then**

delete c **from** C_k ;

endfunction;

Algorytm 1.2. Funkcja *apriori_gen*

2.1.3. Wady algorytmu

W czasie działania algorytm *Apriori* generuje bardzo dużą liczbę zbiorów kandydujących, które potencjalnie mogą się okazać zbiorami częstymi.[5]

Poza tym algorytm wymaga wielu dostępów do bazy danych[5] (w każdym kroku algorytmu należy odczytać całą bazę danych w celu obliczenia wsparcia zbiorów kandydujących).

Przy użyciu algorytmu *Apriori* utrudnione jest znajdowanie długich wzorców.[5] Wynika to z bardzo dużej liczby małych kandydatów (zbiorów o niewielkiej liczbie elementów).

2.2. Algorytm *FP-Growth*

Znajdowanie zbiorów częstych za pomocą algorytmu FP-Growth jest realizowane w trzech krokach.

W pierwszym dochodzi do kompresji bazy danych, w drugim do transformacji do FP-drzewa. Trzeci polega na analizie FP-drzewa w celu znalezienia zbiorów częstych.

2.2.1. Kompresja bazy danych

W pierwszym etapie znajdowane są wszystkie 1 – elementowe zbiory częste występujące w bazie danych.

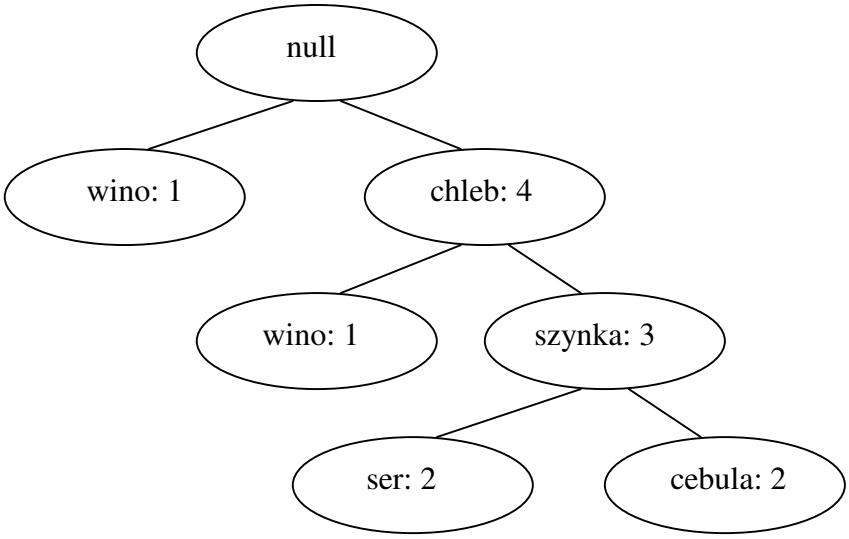
Następnie z każdej transakcji istniejącej w bazie danych usuwane są wszystkie elementy, które nie są częste. Na ogół baza danych otrzymana w wyniku usunięcia nieczęstych elementów ma znacznie mniejszy rozmiar niż wyjściowa baza danych.

Elementy każdej transakcji zostają posortowane według malejącej wartości ich wsparcia. Posortowane transakcje są następnie transformowane do FP-drzewa.

2.2.2. Transformacja do FP-drzewa

FP – drzewo jest ukorzenionym, etykietowanym w wierzchołkach grafem acyklicznym. Korzeń grafu posiada etykietę *null*. Pozostałe wierzchołki grafu, zarówno wierzchołki wewnętrzne jak i liście, reprezentują 1 – elementowe zbiory częste.

Z każdym wierzchołkiem grafu, z wyjątkiem korzenia, związana jest etykieta reprezentująca 1 – elementowy zbiór częsty oraz licznik transakcji określający liczbę transakcji wspierających dany zbiór.



Rys. 2.1. Przykład FP-drzewa

Pierwszy krok to utworzenie korzenia i nadanie mu etykiety *null*.

Następnie odczytuje się skompresowaną bazę danych i dla pierwszej transakcji tworzy się ścieżkę w FP-drzewie. Początkiem ścieżki jest korzeń drzewa.

Kolejność występowania elementów w posortowanej transakcji jest taka jak kolejność wierzchołków w ścieżce odpowiadającej danej transakcji. Licznik transakcji każdego wierzchołka jest początkowo równy 1.

W kolejnym kroku działania algorytmu tworzy się ścieżki odpowiadające następnym transakcjom.

Jeżeli lista elementów danej transakcji posiada wspólny prefiks ze ścieżką już istniejącą w FP-drzewie, elementy należące do wspólnego prefiksu nie tworzą nowych wierzchołków drzewa, lecz współdzielą istniejącą ścieżkę.

Elementy transakcji, które nie należą do wspólnego prefiksu, tworzą nowe wierzchołki. Początkiem nowej ścieżki jest wierzchołek odpowiadający ostatniemu elementowi we wspólnym prefiksie.

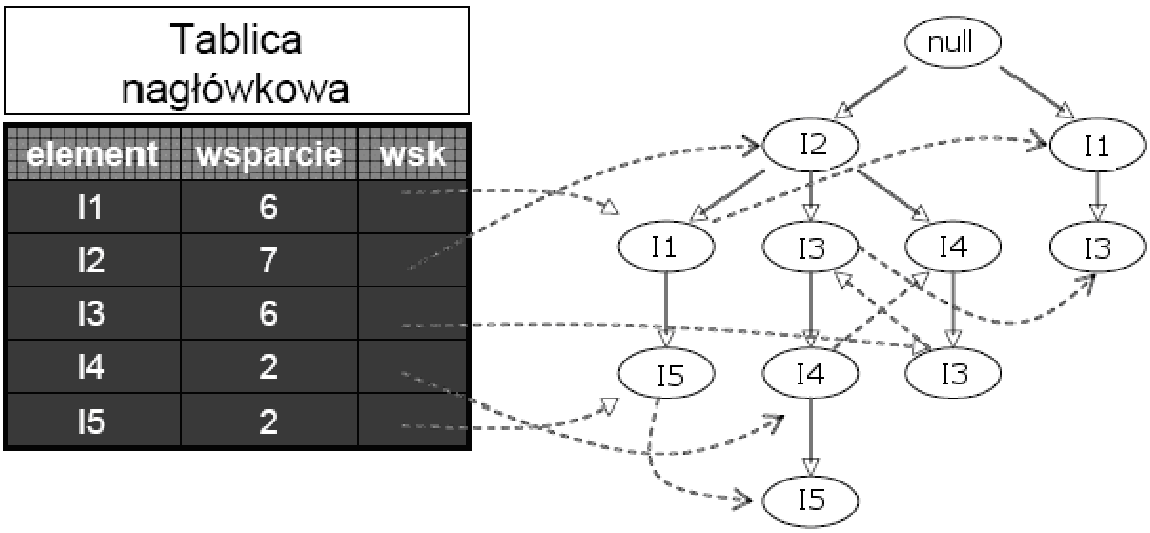
Wskutek współdzielenia ścieżek pojedyncza ścieżka w FP-drzewie, która rozpoczyna się w korzeniu drzewa, odpowiada zbiorowi transakcji, które zawierają identyczne elementy.

Licznik transakcji ostatniego wierzchołka należącego do danej ścieżki określa liczbę transakcji, które wspierają zbiór elementów odpowiadających wierzchołkom grafu należącym do tej ścieżki.

Tablica nagłówkowa

Aby przyspieszyć i ułatwić przeszukiwanie FP-drzewa, tworzy się strukturę pomocniczą, zwaną tablicą nagłówkową. Pełni ona rolę katalogu. Dla każdego elementu wskazuje jego lokalizację w FP-drzewie.

W przypadku, gdy dany element występuje w FP-drzewie wielokrotnie, wskaźnik do wierzchołków odpowiadających danemu elementowi tworzy listę wskaźników.



Rys. 2.2. Przykład tablicy nagłówkowej powiązanej z FP – drzewem[4]

2.2.3. Eksploracja FP-drzewa

Kiedy FP-drzewo zostanie utworzone, należy przeprowadzić jego analizę w celu znalezienia wszystkich zbiorów częstych.

Eksploracja FP – drzewa bazuje na obserwacji, że dla każdego 1 – elementowego zbioru częstego α wszystkie częste nadzbiory zbioru α są reprezentowane w FP – drzewie przez ścieżki zawierające wierzchołek (wierzchołki) α.

Analiza rozpoczyna się od znalezienia dla każdego 1 – elementowego zbioru częstego α wszystkich ścieżek w FP-drzewie, których końcowym wierzchołkiem jest wierzchołek odpowiadający zbiorowi α.

Pojedyncza ścieżka, na której końcu znajduje się wierzchołek α w dalszej analizie będzie nazywana ścieżką prefiksową wzorca α.

Z każdą taką ścieżką związany jest licznik częstości ścieżki. Jego wartość to wartość licznika transakcji wierzchołka końcowego ścieżki, która odpowiada zbiorowi α.

Zbiór wszystkich ścieżek prefiksowych wzorca tworzy warunkową bazę wzorca. Warunkowa baza wzorca służy do konstrukcji tzw. warunkowego FP-drzewa wzorca α, oznaczanego *Tree* - α. Drzewo to jest rekursywnie eksplorowane w celu znalezienia wszystkich zbiorów częstych zawierających zbiór α.

2.2.4. Wady algorytmu

Krytycznym etapem algorytmu FP - growth jest tworzenie FP – drzewa. Ceną jaką trzeba zapłacić za prostą analizę gotowej struktury danych jest czas potrzebny na jest stworzenie.[6]

Należy zwrócić uwagę na to, że wsparcie poszczególnych zbiorów może zostać obliczone dopiero, gdy cały zestaw danych zostanie dodany do FP – drzewa.[6] Poza tym algorytm działa wolno dla baz danych zawierających długie wzorce.

3. Zrównoleglenie obliczeń za pomocą GPU

W rozdziale przedstawiono opis architektury sprzętowej karty graficznej oraz środowiska CUDA, które wykorzystano do zrównoleglenia obliczeń niezbędnych do realizacji algorytmu prezentowanego w niniejszej pracy. Szczególną uwagę poświęcono wielowątkowemu przetwarzania danych za pomocą GPU.

3.1. Możliwości kart graficznych

Historia pecetowych kart graficznych sięga lat siedemdziesiątych, a konkretnie roku 1975, kiedy to firma IBM rozpoczęła sprzedaż swojego pierwszego komputera osobistego, IBM 5100. Urządzenie miało 16 kB pamięci operacyjnej, dane pobierało z taśmy, zaś z użytkownikiem komunikowało się wyświetlając na monitorze koślawe literki: 16 wierszy, po 64 znaki w każdym wierszu. IBM 5100 kosztował [...] "jedyne" 9000 USD.[7]

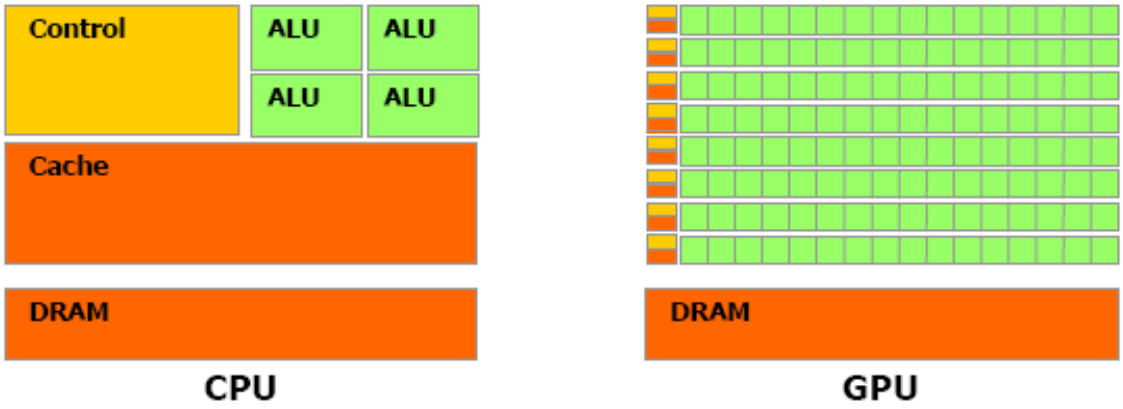
Od tego czasu wiele się zmieniło. Karty graficzne stały się wyspecjalizowanymi układami do przetwarzania danych związanych z obrazem. Rynek gier komputerowych wymusił powstanie zaawansowanych technologicznie urządzeń do wykonywania obliczeń związanych z trójwymiarową grafiką. W efekcie aktualnie dostępne na rynku karty graficzne przewyższają swoją mocą obliczeniową tradycyjne procesory.

Specyfika gier komputerowych wymaga, aby wiele obliczeń było wykonywanych bardzo szybko. Dlatego też w układach obliczeniowych kart graficznych stosuje się rozwiązania wspomagające równoległe przetwarzanie danych. Ponadto producenci kart graficznych musieli zadbać o to, aby ceny ich urządzeń były atrakcyjne dla końcowego odbiorcy. Dzięki temu mamy dzisiaj do dyspozycji układy o dużej mocy obliczeniowej oferowane w stosunkowo niskiej cenie.

Duży potencjał kart graficznych skłania do implementowania na nich algorytmów, które w klasycznym podejściu wymagają dużych zasobów czasowych do ukończenia powierzonego im zadania. Dlatego też stworzono i przedstawiono w niniejszej pracy program wykorzystujący możliwość wielowątkowego przetwarzania na karcie graficznej.

3.2. Architektura GPU i CUDA

Specyfika operacji wykonywanych przez karty graficzne sprawia, że są one projektowane w taki sposób, aby efektownie potrafiły wykonywać równolegle operacje. Dlatego też największa liczba tranzystorów w układach graficznych jest przeznaczona na przetwarzanie danych. Mniej przypada na pamięć podręczną oraz kontrolę przepływu danych.[8] Na poniższym rysunku przedstawiono schematycznie stosunek liczby tranzystorów przeznaczonych do wykonywania opisanych zadań w klasycznym procesorze oraz na karcie graficznej.



Rys. 3.1. Porównanie architektury procesora i karty graficznej[8]

Na powyższym rysunku użyto następujących oznaczeń:

Control – blok logiczny kontrolujący przepływ danych

ALU – jednostka arytmetyczno – logiczna

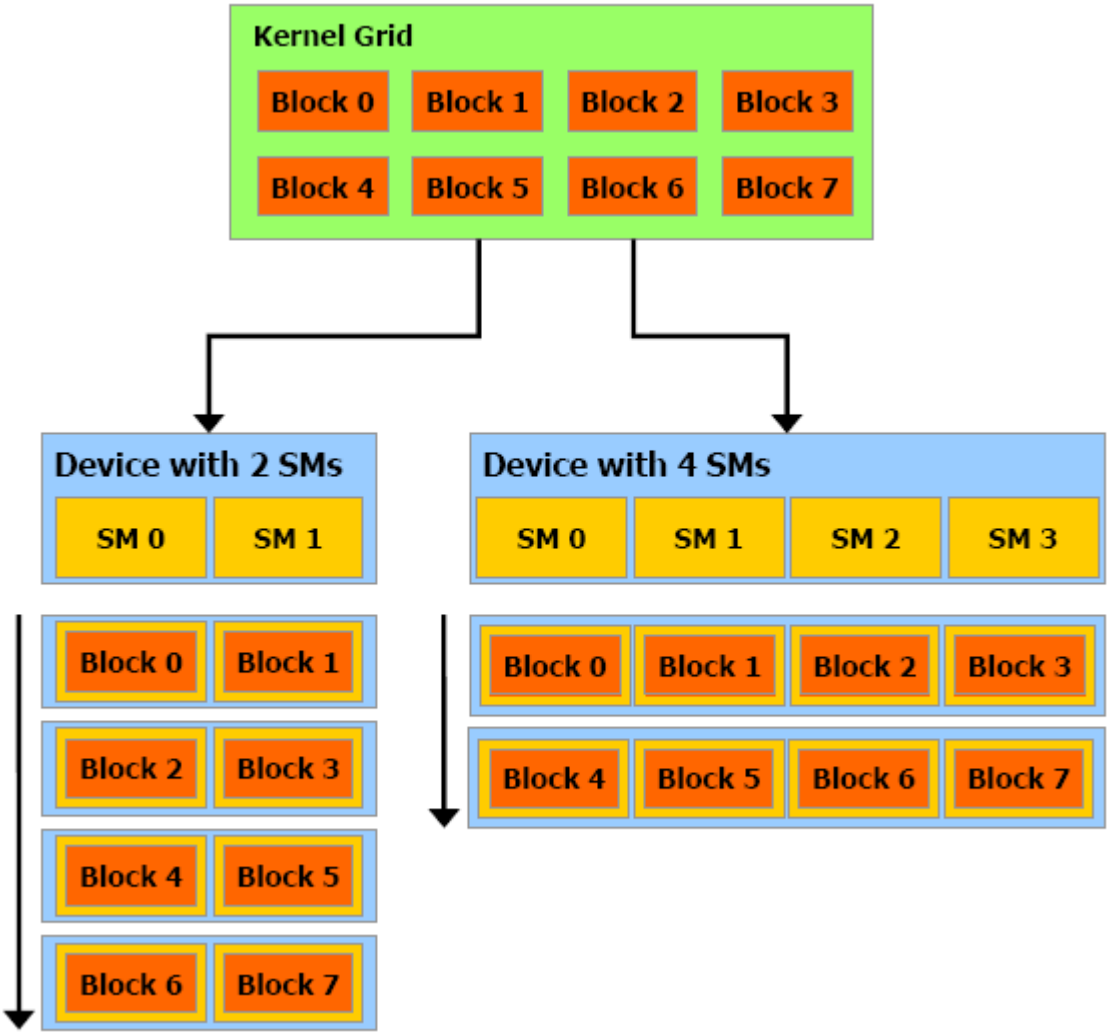
Cache – pamięć podręczna

DRAM – pamięć dynamiczna (*Dynamic Random Access Memory)*

Jednostka przetwarzania grafiki (GPU) nadaje się szczególnie do zadań, które można rozwiązać za pomocą przetwarzania równoległego. Przy takim podejściu te same fragmenty programu są wykonywane na różnych danych. Z racji wielokrotnego używania tych samych instrukcji blok kontrolujący przepływ danych może być mniej rozbudowany niż ten istniejący w klasycznym procesorze.

Architektura CUDA (*Compute Unified Device Architecture*) została zbudowana wokół skalowalnej tablicy wielowątkowych procesorów strumieniowych (*Streaming*

Multiprocessors - SMs). Gdy program hostujący wywołuje siatkę kerneli, bloki są rozmieszczane na multiprocesory według schematu przedstawionego na rysunku 3.2.[9].



Rysunek 3.2. Urządzenie z większą liczbą multiprocesorów automatycznie wykona siatkę kerneli szybciej niż urządzenie z mniejszą liczbą multiprocesorów.[9]

Na powyższym rysunku użyto następujących oznaczeń:

Kernel grid – siatka kerneli

Block 1 : 7 – Blok 1 : 7

Device with 2 (4) SMs – urządzenie z dwoma (czterema) multiprocesorami

Multiprocesor składa się z ośmiu procesorów skalarnych (ang. Scalar Procesor – SP). Jego zadaniem jest tworzenie zarządzanie i wykonywanie wątków.

3.3. Środowisko CUDA

NVIDIA CUDA to model programowania równoległego oraz środowisko programowe, które pozwala wykorzystać moc obliczeniową *GPU* (*graphics processing unit*) do zadań nie związanych z przetwarzaniem grafiki.

CUDA została oparta na kilku podstawowych założeniach. Po pierwsze zawiera ona kilka rozszerzeń języka C, które pozwalają na prostą implementację algorytmów równoległych. Dzięki temu zamiast spędzać wiele czasu nad implementacją programista może skupić się na projektowaniu odpowiedniego rozwiązania.

Poza tym CUDA korzysta z modelu, w którym jeden program zawiera elementy wykonywane przez CPU oraz takie, które są uruchamiane na GPU. Sekwencyjne fragmenty aplikacji z powodzeniem mogą zostać powierzone klasycznemu procesorowi. Karta graficzna wykonuje te zadania, dla których opracowano równoległe plany wykonania. Dzięki temu możliwe jest powierzenie procesorom karty graficznej wykonania fragmentów kodu istniejących programów, do tej pory uruchamianych tylko na CPU. Ponadto przy tworzeniu nowych programów możliwe jest stworzenie wersji działającej na klasycznym procesorze i sukcesywne przenoszenie kolejnych elementów algorytmu na GPU. Umożliwia to testowanie i porównanie wydajności różnych wersji aplikacji.

CPU oraz GPU są traktowane jako oddzielne urządzenia, które posiadają własne przestrzenie pamięci. Z tego faktu wynika cena jaką trzeba zapłacić za możliwość równoległego przetwarzania danych na karcie graficznej. Mowa tu o opóźnieniach, które pojawiają się przy transferach danych z CPU do GPU i z powrotem, a które nie występują przy klasycznym podejściu charakterystycznym dla przetwarzania sekwencyjnego.

4. Definicje

W rozdziale przedstawiono definicję pojęć używanych w całej pracy.

Wsparcie (sup) – określa liczbę wystąpień danego zbioru w bazie danych.

Minimalny próg wsparcia (minsup) –określa minimalną wartość wsparcia zbioru częstego. Jest wielkością ustalana przez analityka.

Zbiór częsty – zbiór, którego wsparcie jest większe lub równe minimalnemu progowi wsparcia.

Reguła asocjacyjna – sposób zapisu wiedzy o asocjacjach w bazie danych; Reguły w najbardziej naturalny sposób przedstawia się za pomocą logicznej implikacji.

W każdej regule można wyróżnić dwa zbiory wartości: warunkujące oraz warunkowane. Regułę o części warunkującej $X \subset Z$ i części warunkowej Y ? Z zapiszemy więc w sposób następujący:

X => Y

i będziemy ją interpretować jako stwierdzenie, że wartości atrybutów ze zbioru X często pociągają za sobą wartości atrybutów ze zbioru Y. Inaczej – w wielu przykładach, w których występują wszystkie wartości zbioru X występują również wszystkie wartości ze zbioru Y.[10]

5. Opracowany algorytm

W rozdziale przedstawiono opis opracowanego algorytmu. Omówiono ideę projektowanego rozwiązania oraz szczegóły implementacji.

Ponadto opisano etapy rozwijania algorytmu z uwzględnieniem przyczyn zmian oraz ich wpływem na ostateczny kształt implementacji.

5.1. Idea algorytmu

Konstrukcja algorytmu opiera się na kilku głównych fundamentach. Tworzą one spójne rozwiązanie, które pozwoliło na opracowanie wydajnej metody wyszukiwania zbiorów częstych. Poniżej przedstawiono wspomniane elementy koncepcyjne algorytmu.

5.1.2. Monotoniczność miary wsparcia

Przy tworzeniu algorytmu wykorzystano obserwację na temat monotoniczności miary wsparcia, z której czerpie także algorytm Apriori.

„Miara wsparcia zbioru elementów ma własność monotoniczności. Monotoniczność miary wsparcia zbioru oznacza, że zbiór X jest zbiorem częstym wtedy i tylko wtedy, gdy wszystkie podzbiory zbioru X są również zbiorami częstymi. Innymi słowy, jeżeli zbiór X nie jest zbiorem częstym, to żaden nadzbiór zbioru X nie jest zbiorem częstym. Oznacza to, że nie musimy rozważać wsparcia zbioru X, którego podzbiór nie jest zbiorem częstym.”[4]

Własność tę wykorzystano do zawężenia przestrzeni poszukiwań zbiorów częstych poprzez usuwanie ze zbioru danych przeznaczonego do dalszej analizy zbiorów, które nie są częste.

5.1.3. Przedstawienie danych w postaci bitmapy

Etapem wstępnym do wyszukiwania zbiorów częstych jest przedstawienie danych wejściowych w postaci struktury nadającej się do efektywnego przetwarzania w kolejnych krokach algorytmu.

Baza danych z analizowanymi informacjami zostaje odwzorowana na dwuwymiarową tablicę zawierającą wartości binarne. Wiersze tablicy odpowiadają

pojedynczym elementom istniejącym w bazie danych (potencjalnym jednoelementowym zbiorom częstym).

Kolumny tablicy odpowiadają kolejnym transakcjom. Sposób tworzenia bitmapy opisuje poniższy przykład.

Przykład

Niech dana będzie baza danych zawierające następujące transakcje:

1. mleko, wino, szynka, chleb
2. cukier, chleb,
3. szynka, wino,
4. mleko
5. mleko, szynka, chleb
6. mleko, wino, szynka, chleb

Określony zestaw danych zawiera pięć różnych produktów (mleko, wino, szynka, chleb, cukier) oraz sześć transakcji. W związku z tym utworzona bitmapa będzie miała wymiary 5x6.

Zakładając, że wartość ‘1’ oznacza, że dany produkt występuje w określonej transakcji, a wartość ‘0’, że nie występuje, tworzymy następujące odwzorowanie:

transakcja / produkt	**1**	**2**	**3**	**4**	**5**	**6**
mleko	1	0	0	1	1	1
wino	1	0	1	0	0	1
szynka	1	0	1	0	1	1
chleb	1	1	0	0	1	1
cukier	0	1	0	0	0	0

Tabela 5.1. Przykładowa bitmapa

W implementacji przedstawionej w niniejszej pracy zastosowano dodatkowy zabieg, który powoduje znaczne poprawienie wydajności programu wyszukującego zbiory częste.

Otrzymana w opisywanym kroku algorytmu bitmapa nie jest reprezentowana w pamięci jako tablica zmiennych binarnych (typ *bool* w jezyku *C*), tylko jako tablica liczb całkowitych bez znaku (typ *unsigned int*). Szczegóły rozwiązania przedstawiono w podrozdziale 4.2.1

5.1.4. Operacja AND na wierszach bitmapy i jej interpretacja

Każdy z wierszy bitmapy utworzonej w sposób omówiony w punkcie 5.1.3. zawiera opis występowania unikalnego, jednoelementowego zbioru w poszczególnych transakcjach bazy danych.

Weźmy pod uwagę dwa kolejne wiersze bitmapy. Oznaczmy je A i B. Należy zauważyć, że bitowa operacja *AND* wykonana na wierszach A i B da nam w rezultacie strukturę danych opisującą występowanie dwuelementowego zbioru (zawierającego zbiór A oraz B) w poszczególnych transakcjach.

Logiczny iloczyn tak otrzymanej struktury z kolejnym wierszem (C) bitmapy da nam w rezultacie informację o występowaniu zbioru ABC w poszczególnych transakcjach bazy danych.

Warto zauważyć, że częstość występowania zbioru w bazie danych (wsparcie) można obliczyć sumując jedynki, które zawiera struktura danych opisująca ten zbiór.

Zobrazowaniu przedstawionych spostrzeżeń służy przedstawiony poniżej przykład.

Przykład

Niech dane będą trzy kolejne wiersze bitmapy utworzonej w sposób przedstawiony w punkcie 5.1.3.:

A:

1	0	0	1	1	1

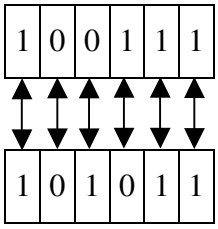
B:

1	0	1	0	1	1

C:

1	0	1	0	1	0

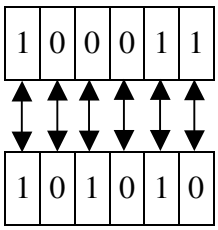
W wyniku iloczynu logicznego wiersza A i B otrzymujemy:



AND => 1 0 0 0 1 1 AB:

Wiersz AB zawiera jedynkę na trzech pozycjach. Otrzymany wynik oznacza występowanie zbioru AB w transakcji pierwszej piątej oraz szóstej. Wsparcie zbioru AB można obliczyć sumując jedynki występujące w wierszu AB. Dla analizowanego przypadku wynosi ono 3.

W wyniku iloczynu logicznego wiersza AB oraz C otrzymujemy:



AND => ABC: 1 0 0 0 1 0

Wiersz ABC zawiera jedynkę na dwóch pozycjach. Otrzymany wynik oznacza występowanie zbioru ABC w transakcji pierwszej oraz piątej. Wsparcie zbioru ABC można obliczyć sumując jedynki występujące w wierszu ABC. Dla analizowanego przypadku wynosi ono 2.

5.1.5. Rekurencja

Główny etap algorytmu odkrywania zbiorów częstych opisywanego w niniejszej pracy wykorzystuje własność bitmap opisanych w podrozdziale 5.1.3. pozwalającą mu na odkrywanie zbiorów zawierających konkretny podzbiór.

W pierwszym kroku program pracuje na strukturze danych utworzonej zgodnie z opisem z punktu 5.1.3. Załóżmy, że pierwotna bitmapa (przyjmijmy oznaczenie B1) ma wymiary M x N. Tworząc logiczne iloczyny wiersza pierwszego z kolejnymi otrzymamy bitmapę (B21) o wymiarach (M-1) x N. Będzie ona zawierać charakterystykę występowania w bazie danych dwuelementowych zbiorów zawierających element, do którego odnosi się pierwszy wiersz pierwotnej bitmapy. Każdy wiersz bitmapy B2 zawiera informacje, które zbiory zawierają podzbiór złożony z elementu reprezentowanego przez pierwszy wiersz bitmapy B1.

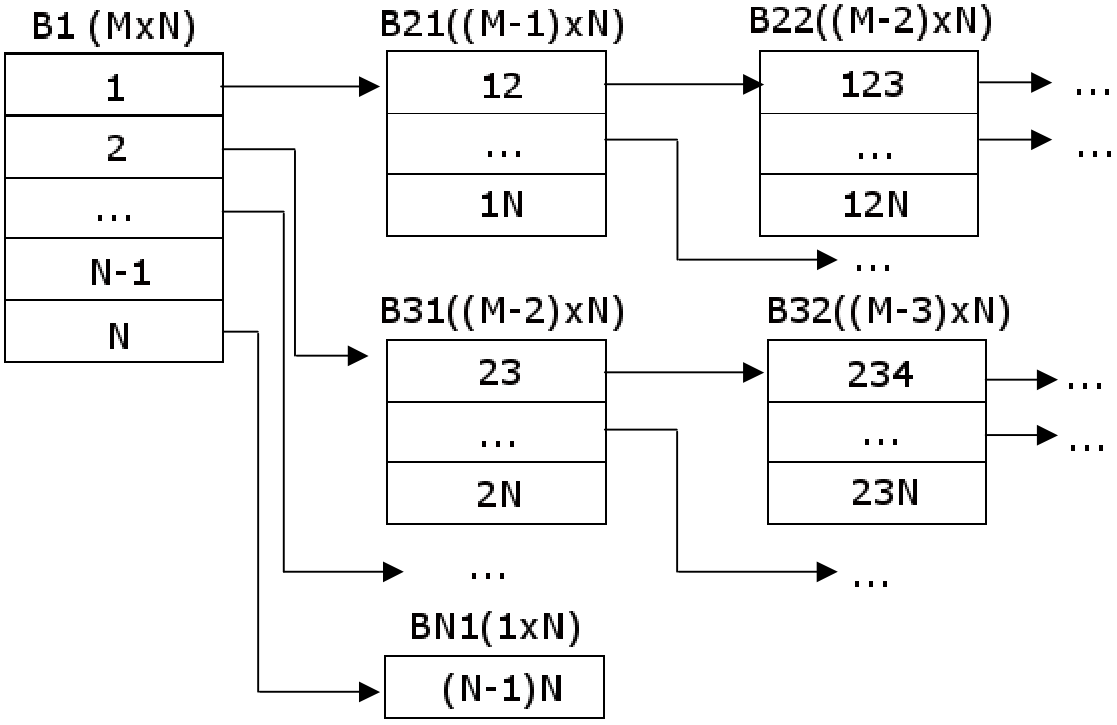
W kolejnym kroku kontynuujemy przeglądanie bitmapy B1 tworząc logiczne iloczyny wiersza drugiego z trzecim, czwartym itd.. W wyniku otrzymujemy bitmapy

B31, B41 itd.. Proces trwa do momentu wykonania logicznej operacji AND przedostatniego i ostatniego wiersza bitmapy B1, którego wynikiem jest bitmapa BN1 o wymiarach 1 x N. Po otrzymaniu bitmap B21, B31, ..., BN1 usuwamy z nich wiersze, które nie opisują zbiorów częstych, tzn. zawierają mniej jedynek, niż określa to minimalna dopuszczalna wartość wsparcia. W wyniku otrzymujemy bitmapy B21', B31', ..., BN1'.

Aby odkryć zbiory częste o liczebności elementów większej niż dwa należy na bitmapach B21', B31', ..., BN1 przeprowadzić operacje analogiczne do tych wykonanych na bitmapie B1 i zgodne z interpretacją z punktu 5.1.4. Przykładowo bitmapa B32 będzie zawierać wiersze wskazujące na zbiory złożone z trzech elementów, z których pierwsze dwa są reprezentowane przez wiersze 1 i 2 bitmapy B1, a pozostały element jest inny w każdy wierszu bitmapy B22.

Widać wyraźnie, że prezentowany algorytm ma postać rekurencyjną. Warto także zauważyć, że warunkiem koniecznym do dalszego analizowania otrzymanej bitmapy jest to, aby po usunięciu z niej elementów opisujących zbiory nieczęste zawierała ona co najmniej dwa wiersze.

Schemat opisanego procesu przedstawia poniższy rysunek. Dla uproszczenia zapisu pominięto krok usuwania zbiorów nieczęstych.



Rys. 5.1. Rekurencyjne wyszukiwanie zbiorów częstych

5.2. Implementacja na CPU

Implementacja prezentowanego w niniejszej pracy algorytmu była realizowana etapami. Po stworzeniu pierwszej wersji programu przeprowadzono wstępne eksperymenty, które ukierunkowały kierunek dalszego rozwoju. Określono etapy, których wydajność można poprawić. Następnie wprowadzano do programu nowe rozwiązania i przeprowadzono eksperymenty weryfikujące.

W rozdziale przedstawiono kluczowe elementy programu wyszukującego zbiory częste wraz z opisem zmian wprowadzanych w czasie jego rozwoju.

5.2.1. Stworzenie bitmapy

W początkowym etapie prac nad programem bitmapa była przechowywana w pamięci jako dwuwymiarowa tablica zmiennych typu *bool*. Rozwiązanie to na pierwszy rzut oka wydawało się najbardziej naturalne i pozwoliło na szybkie stworzenie działającego prototypu, który następnie należało udoskonalić.

Poniżej zaprezentowano podstawowy algorytm tworzenia bitmapy.

For each unikalnego elementu $i \in I$ **do**
begin
    For each transakcji $t \in T$ **do**
    begin
        B_{kl} = **check_item**(i_k , t_l);
    end;
end;

Algorytm 5.1 Podstawowy algorytm tworzenia bitmapy

W opisie zastosowano następujące oznaczenia:

I - zbiór unikalnych elementów występujących w bazie danych,

T - zbiór transakcji

B – bitmapa

k, l – indeksy elementu bitmapy

Funkcja **check_item()** sprawdza czy dany element występuje w aktualnie analizowanej transakcji i jest określona następująco:

function check_item((i_k , t_l))
    if($i_k \in t_l$)
        return true;
    else
        return false;
endfunction;

Algorytm 5.2. Funkcja check_item

Wersja int bitmapy

Wykonanie operacji AND na dwóch wierszach bitmapy zawierającej wartości typu *bool* sprowadza się do wykonania N operacji AND na elementach wierszy, przy czym N określa liczbę elementów. Procesor komputera umożliwia wykonanie jako jednej operacji bitowego iloczynu nie tylko wartości binarnych, ale także liczb całkowitych. Stąd pomysł, aby przedstawić wartości binarne na kolejnych bitach zmiennych typu *unsigned int*. Zakładając 32 – bitowy rozmiar pojedynczej zmiennej typu *int* oraz to, że każda zmienna *bool* reprezentuje jedną wartość binarną, wykonanie logicznej operacji AND na dwóch liczbach całkowitych odpowiada wykonaniu 32 operacji na zmiennych typu *bool*. Potencjalny zysk wydajnościowy jest więc widoczny na pierwszy rzut oka.

Proces transformacji dwuwymiarowej tablicy zawierającej wartości *bool* do tablicy z liczbami *int* został zrealizowany przy pomocy następującego algorytmu:

For each wiersza w ∈ B **do**
begin
    B' $_k$ **= make_int_vector**(B $_k$ **);**
end

Algorytm 5.3 Algorytm konwersji tablicy zmiennych *bool* do *int*

W opisie zastosowano następujące oznaczenia:

B – tablica 2D z wartościami typu *bool*

B' – tablica 2D z wartościami typu *int*

Funkcja make_int_vector() tworzy z jednowymiarowej tablicy wartości typu *bool* jednowymiarową tablicę wartości *int* i ma następującą postać:

X = **get_bool_packets**();
for each paczki ∈ X **do**
    for each wartości ∈ paczki **do**
        B $_k$ ' += **add_value**(wartość);
    end;
end;

Algorytm 5.4. Opis funkcji *get_bool_packets*

Funkcja **get_bool_packets()** dzieli wiersz bitmapy B na 32 - elementowe pakiety. Funkcja **add_value()** zwraca zero w przypadku, gdy wartość zmiennej *bool* jest ustawiona na *false*. W przeciwnym razie zwracana wartość jest i – tą potęgą liczby 2,

przy czym i określa pozycję zmiennej *bool* w analizowanej paczce (0 dla najmłodszego bitu, +1 dla każdego kolejnego bitu).

5.2.2. Usunięcie z danych wejściowych zbiorów nieczęstych

Aby ograniczyć przestrzeń poszukiwań zbiorów częstych jeszcze przed wykonaniem pierwszej logicznej operacji AND, tuż po utworzeniu bitmapy usuwane są z niej wiersze odpowiadające zbiorom, które nie są częste. Proces ten jest realizowany przez funkcję określoną w następujący sposób:

```
function get_frequent_sets(B)
        if(sup(w) < minsup)
                delete w;
endfunction;
```

W opisie zastosowano następujące oznaczenia:

w – wiersz bitmapy B

Algorytm 5.5. Algorytm usuwania zbiorów nieczęstych

Usuwanie zbiorów nieczęstych z bieżącej przestrzeni poszukiwań zostało zrealizowane w opisany powyżej sposób także w późniejszych krokach algorytmu (patrz punkt 5.2.5).

5.2.3. Rekurencyjne wyszukiwanie zbiorów częstych

Wywoływana rekurencyjnie funkcja wyszukująca zbiory częste, której idea została opisana w punkcie 5.1.5. ma następującą postać:

```
frequentSets = NULL;

function find_sets(B)
        for each wiersza w ∈ B do
                and_table = get_table_to_bit_and();
                if(and_table.count > 0)
                begin
                        B' = get_row_table_bit_and(w, and_table);
                        B'' = get_frequent_sets(B');
                        addSets(B'');
                        if(B''.count > 1)
                        begin
                                find_sets(B'');
                        end;
```

```
            end;
        end;
    endfunction;
```

Algorytm 5.6. Rekurencyjna funkcja wyszukująca zbiory częste

W opisie zastosowano następujące oznaczenia:

frequentSets – struktura przechowująca zbiory częste

and_table – tablica składająca się z odpowiednich wierszy pierwotnej bitmapy (powstałej według opisu z punktu 5.1.3). Zawiera ona wektory, na których zostanie przeprowadzona logiczna operacja AND z aktualnie analizowanym wierszem.

and_table.count – liczba wierszy tablicy and_table

B' – dwuwymiarowa tablica zawierająca wynik logicznej operacji AND tablicy and_table oraz aktualnie analizowanego wiersza

B'' – jest to bitmapa B' po usunięciu elementów odpowiadających zbiorom nieczęstym

addSets – funkcja dodająca tablicę do struktury wskazującej na zbiory częste

Funkcja **get_table_to_bit_and()** jest określona w następujący sposób:

```
function get_table_to_bit_and()
    next_row_index = current_row.index + 1;
    and_table = select row
                from B
                where row.index ≥ next_row_index;
    return and_table;
endfunction;
```

Algorytm 5.7. Funkcja tworząca bitmapę and_table

W opisie zastosowano następujące oznaczenia:

next_row_index – indeks pierwszego wiersza pierwotnej bitmapy B, który wejdzie w skład bitmapy B'

current_row – wiersz aktualnie analizowany w funkcji *find_sets(B)*

current_row.index – indeks aktualnie analizowanego wiersza

Funkcja *get_row_table_bit_and* jest określona w następujący sposób:

```
function get_row_table_bit_and(w, and_table)
    for each wiersza x ∈ and_table do
```

```
            and_table_k = get_bit_and(x, and_table_k );
        end;
    endfunction;
```

Algorytm 5.8. Funkcja tworząca bitowy AND danego wiersza i bitmapy and_table

przy czym funkcja *get_bit_and* zwraca wynik logicznej operacji AND przeprowadzonej na dwóch wektorach i ma następującą postać:

```
function get_bit_and(x, y)
    for each int_number ∈ x do
        z_k = x & y;
    end;
    return z;
endfunction;
```

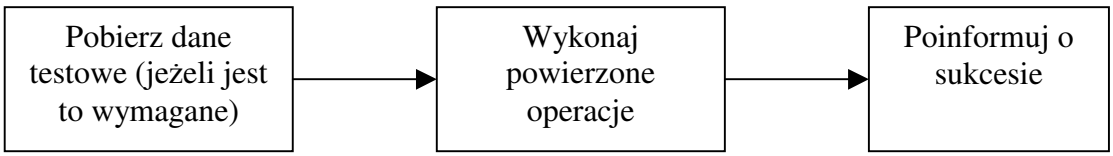
Algorytm 5.9. Funkcja tworząca bitowy AND dwóch wierszy x i y

Funkcja *get_frequent_sets* została opisana w punkcie 5.2.2.

5.2.4. Testowanie modułów programu

W początkowym etapie prac, tuż przed ukończeniem prototypu zauważono, że cała aplikacja jest podatna na błędy związane z wykorzystaniem języka niskiego poziomu (C). Były one spowodowane głównie niewłaściwą alokacją oraz nieodpowiednim zwalnianiem pamięci.

Trudności w precyzyjnym zlokalizowaniu źródeł błędów skłoniły autora programu do stworzenia funkcji testującej dla każdego istotnego modułu programu. Zadaniem każdej takiej funkcji było sprawdzenie czy dany fragment kodu realizuje w stu procentach zadania, które mu powierzono. Ideę działania funkcji testującej przedstawia poniższy schemat:



Rys. 5.2. Idea funkcji testującej

Zaprezentowane podejście cechowała duża pracochłonność. Należy jednak zauważyć, że pozwoliło ono na elementarną analizę tworzonego kodu i wyeliminowanie błędów, które wcześniej były trudne do zdiagnozowania.

Przy okazji tworzenie metod testujących stworzono odpowiednią strukturę projektu, która w dalszych etapach prac ułatwiała zarządzanie kodem. Każdy z modułów programu umieszczono wraz z funkcją testującą w oddzielnym pliku. Poza tym jasno oddzielono wersję CPU i GPU programu.

5.2.5. Algorytmy alokacji pamięci

Początkowo w programie stosowano zasadę, że pamięć jest alokowana w chwili, kiedy jest to konieczne. Innymi słowy tuż przed momentem, w którym mają do niej zostać zapisane jakieś dane. Testowanie poszczególnych modułów programu (patrz punkt 5.2.4.) pokazało, że jest to podejście, które może powodować wiele błędów.

Postanowiono więc, aby najczęściej wykorzystywane struktury danych były reprezentowane w programie na pomocą globalnych wskaźników dostępnych w całym zakresie programu i inicjowanych tylko raz na początku działania algorytmu. Takie podejście poza większą przejrzystością kodu i wyeliminowaniem potencjalnych błędów pojawiających się przy alokacji i zwalnianiu pamięci gwarantuje przyspieszenie działania całego programu. W celu zobrazowania omawianego procesu przytoczono poniższy przykład.

Przykład

Niech dana będzie funkcja *do_work(x, y)* przyjmująca jako parametry dwa wektory (x i y) i realizująca ich logiczny iloczyn. Możliwe są dwa podejścia. Po pierwsze funkcja alokuje pamięć dla wynikowego wektora (z) i zwraca wskaźnik do wyniku. Zgodnie z drugim podejściem funkcja korzysta z dostępnego dla niej wcześniej zaalokowanego obszaru pamięci określonego przez globalny wskaźnik. Przedstawione podejścia obrazują poniższe algorytmy.

```
function do_work(x, y)
        temp = allocate_mem();
        temp = inner_work(x, y);
        return temp;
endfunction;
```

Algorytm 5.10. Realizacja funkcji z użyciem lokalnej alokacji pamięci

```
global_pointer = allocate_mem();
function do_work(x, y)
```

```
        global_pointer = inner_work(x, y);
    endfunction
```

Algorytm 5.11. Realizacja funkcji z użyciem globalnego wskaźnika

W opisie zastosowano następujące oznaczenia:

temp – tymczasowy wskaźnik alokowany lokalnie

allocate_mem – funkcja alokująca pamięć

inner_work – funkcja wykonująca właściwe operacje (w analizowanym przypadku iloczyn logiczny dwóch wektorów).

global_pointer – globalny wskaźnik do pamięci. W podejściu zgodnym z algorytmem 5.11 pamięć na którą wskazuje jest alokowana przed pierwszym wywołaniem funkcji *do_work*.

Zakładając, że w programie głównym funkcja *do_work* zostanie wykonana N - krotnie, przy realizacji zgodnej z algorytmem 5.11. oszczędzamy (w porównaniu z realizacją 5.10) czas potrzebny na (N-1) alokacji pamięci wykonywanych przez funkcję *allocate_mem*.

5.2.6. Obliczanie wsparcia

W czasie prac nad opisywanym w niniejszej pracy programem przygotowano dwie wersje algorytmu obliczającego wsparcie zbioru opisanego przez odpowiednią strukturę danych. Pierwsza wersja wchodziła w skład wspominanego wcześniej prototypu i operowała na bitmapie złożonej ze zmiennych typu *bool*. Druga została przygotowana po zmianie sposobu przechowywania bitmapy w pamięci (tablica *int* zamiast tablica *bool*, patrz punkt 5.2.1.).

Pierwotna wersja obliczania wsparcia sprowadza się do sekwencyjnego przejrzenia aktualnie analizowanego wektora i zliczenia jedynek, które w nim występują. Opisuje ją następujący algorytm:

```
sup = 0;
for each elementu e ∈ W do
    if(e == 1)
        sup = sup + 1;
end;
```

Algorytm 5.12. Pierwotna wersja algorytmu obliczania wsparcia zbioru

W opisie zastosowano następujące oznaczenia:

sup – wsparcie zbioru

W – wektor bitmapy

e – element wektora W

Po zmianie sposobu reprezentowania bitmapy w pamięci należało zmienić sposób obliczania wsparcia. Aby otrzymać prawidłowe wyniki trzeba analizować każdą liczbę *int*, którą zawiera wiersz bitmapy, a następnie zsumować liczby jedynek występujących w każdej z liczb całkowitych wchodzących w skład wiersza. Otrzymany wynik jest wsparciem zbioru określonego przez aktualnie analizowany wektor.

Przy okazji zmiany sposobu obliczania wsparcia wprowadzono dodatkowe ulepszenie znacznie przyspieszające proces obliczeniowy. Mowa tu o zastosowaniu tzw. *tablicy lookup* (ang. *lookup table*). Eksperymenty weryfikujące wydajność zliczania jedynek za pomocą przedstawionych algorytmów opisano w rozdziale 6.1. Sposób obliczania wsparcia z użyciem *tablicy lookup* opisuje poniższy algorytm.

```
sup = 0;
for each elementu e ∈ W do
        sup = sup + count_ones(e);
end;
```

Algorytm 5.13. Obliczanie wsparcia zbiorów z wykorzystaniem tablicy lookup

przy czym funkcja *count_ones* jest określona w następujący sposób:

```
function count_ones(e)
        count = LOOK_UP_TABLE[intValue & 0xff] +
                LOOK_UP_TABLE[(intValue >> 8) & 0xff] +
                LOOK_UP_TABLE[(intValue >> 16) & 0xff] +
                LOOK_UP_TABLE[intValue >> 24];
        return count;
endfunction;
```

Algorytm 5.14. Obliczanie wsparcia zbioru z wykorzystaniem tablicy *lookup*

W opisie zastosowano następujące oznaczenia:

count – wsparcie zbioru (liczba jedynek, które zawiera wektor opisujący dany zbiór)

LOOK_UP_TABLE – tablica zawierająca odpowiednie wartości

Wartości, które zawiera użyta tablica lookup zostały wygenerowane według następującego algorytmu:

```
LOOK_UP_TABLE[0] = 0;
for(k = 0; k < 256; ++k)
```

```
begin
        LOOK_UP_TABLE[k] = (k & 1) + LOOK_UP_TABLE[k / 2];
end
```

Algorytm 5.15. Tworzenie tablicy lookup wykorzystywanej do obliczania wsparcia

Do obliczenia liczby jedynek, które zawiera określony wektor zmiennych typu *int* można zastosować algorytm nie wykorzystujący tablicy *lookup*. Jego podstawowe założenia są takie jak w przypadku algorytmu 5.13. Różnica tkwi w sposobie implementacji funkcji *count_ones*. Rozwiązanie bez tablicy lookup może mieć postać:

```
function count_ones(e)
        count = 0;
        do
                bit = e % 2;
                if(bit == 1)
                        count += 1;
                e = e / 2;
        while(e != 0);
        return count;
endfunction;
```

Algorytm 5.16 Obliczanie wsparcia zbioru bez wykorzystania tablicy *lookup*

W opisie zastosowano następujące oznaczenia:

bit – zmienna określająca resztę z dzielenia modulo 2 liczby e

W trakcie prac nad programem opisywanym w niniejszej pracy początkowo nie zaimplementowano rozwiązania, które opisuje algorytm 5.16. Pominięto je przewidując, że będzie ono wolniejsze od rozwiązania wykorzystującego tablicę *lookup* (algorytm 5.13, 5.14). Testowy program realizujący algorytm 5.16 został opracowany w końcowym etapie realizacji projektu. Celem było obiektywne porównanie wydajności wszystkich rozważanych sposobów obliczania wsparcia. Wyniki pomiarów przedstawiono w punkcie 6.1.

5.2.7. Pomiar czasu

W trakcie prac nad implementacją opisanego w niniejszej pracy algorytmu zaistniała konieczność zmierzenia czasu wykonania programu wyszukującego zbiory częste. Dzięki pomiarom można było porównać szybkość działania wersji CPU oraz GPU programu.

W celu poprawienia wydajności opracowanego rozwiązania niezbędne było także zmierzenie czasów wykonania jego poszczególnych elementów. Pozwoliło to na

określenie części algorytmu, których wykonanie było najbardziej czasochłonne. Dzięki zebranym danym zaprojektowano i zaimplementowano wydajniejsze wersje programu.

Początkowo przy pomiarze czasu wykorzystywano typ *clock_t*. Aby otrzymać wynik w sekundach stosowano następujący sposób pomiaru:

```
clock_t startTime, endTime;
startTime = clock();
do_work();
endTime = clock();
double totalTime = (double)( endTime - startTime ) / (double)CLOCKS_PER_SEC;
```

Algorytm 5.17. Pomiar czasu wykonania programu (metoda pierwsza)

W opisie zastosowano następujące oznaczenia:

clock_t – typ zwracany przez funkcję *clock*

clock – funkcja zwracająca liczbę taktów zegara, które upłynęły od momentu uruchomienia programu[11]

do_work – funkcja, której czas wykonania jest mierzony

CLOCKS_PER_SEC – stała określająca liczbę taktów zegara wykonywanych w ciągu sekundy[11].

Otrzymywane wyniki były wystarczająco precyzyjne mierzenia czasu wykonania całego programu i porównywania wersji CPU i GPU. Szybkość wykonywania operacji składowych algorytmu była jednak na tyle duża, że w celu ich zmierzenia konieczne było użycie innej metody pomiaru. Zastosowano kod mierzący liczbę taktów zegara od momentu włączenia procesora. Poniżej przedstawiono szczegóły rozwiązania.

```
__int64 startTime, endTime;
startTime = GetTicks();
do_work();
__int64 totalTime= endTime – startTime;
```

Algorytm 5.18. Pomiar czasu wykonania programu (metoda druga)

Funkcja GetTicks jest określona następująco:

```
__int64 GetTicks()
{
        __int64 Result;
        __asm
        {
                rdtsc
                mov dword ptr [Result],eax
                mov dword ptr [Result+4],edx
        }
        return Result;
}
```

Przy pomiarze czasu wykonania operacji z wykorzystaniem procesora karty graficznej należy zapewnić zakończenie wszystkich wątków realizujących dane zadanie. W tym celu posłużono się funkcją *cudaThreadSynchronize*. Wzbogacony o ten element algorytm 5.16 ma postać:

```
__int64 startTime, endTime;
startTime = GetTicks();
do_work();
cudaThreadSynchronize();
__int64 totalTime= endTime – startTime;
```

Algorytm 5.19. Pomiar czasu wykonania operacji na karcie graficznej (metoda 2)

5.3. Implementacja na GPU

W rozdziale przedstawiono opis implementacji elementów, które zostały zrealizowane z wykorzystaniem procesora karty graficznej.

5.3.1. Wykonywanie logicznej operacji AND na wektorach

Istotnym elementem procesu wyszukiwania zbiorów częstych (patrz punkt 5.1.5.) jest wykonywanie logicznej operacji AND na wierszach analizowanej bitmapy. Długość wiersza jest określona przez liczbę transakcji, które zawiera analizowana baza danych. W podejściu klasycznym logiczny AND dwóch wektorów (X i Y) jest realizowany według następującego algorytmu:

```
for each k < X.count do
        Z[k] = X[k] & Y[k];
end;
```

Algorytm 5.20 Logiczny AND dwóch wektorów (podejście sekwencyjne)

W opisie zastosowano następujące oznaczenia:

Z – Wektor będący wynikiem bitowego iloczynu wektorów X i Y

k – indeks elementu wektora

X.count – liczba elementów wektora X

Należy pamiętać, że w programie wykonywanym przez CPU opisana pętla jest wykonywana przez jeden wątek. W rezultacie czas wykonania takiej pętli to N x t, przy czym N określa długość wektora X oraz Y, a t do czas potrzebny na wykonanie logicznej operacji AND pojedynczego elementu tablicy X oraz Y.

Ideą wykonania opisywanego elementu algorytmu na karcie graficznej jest zrównoleglenie operacji AND pojedynczych elementów wektorów X i Y. Przy użyciu CUDA możliwe jest napisanie programu, który będzie realizował powierzoną mu operację równolegle z wykorzystaniem wielu wątków. Pętla z algorytmu 5.9 została zapisana w wersji przeznaczonej do uruchomienia na karcie graficznej w następujący sposób:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;

if (idx < N)
        X[idx] = Z[idx] & Y[idx];
```

Algorytm 5.21 Logiczny AND dwóch wektorów (podejście wielowątkowe)

W pierwszej wersji była to jedyna modyfikacja algorytmu CPU.

5.4. Zmieniony algorytm GPU

Po wykonaniu eksperymentów (punkt 6.2 i 6.3) testujących wydajność programu zgodnego z punktem 5.3 okazało się, że otrzymane wyniki nie są zadowalające. Wersja GPU działała wolniej od analogicznej CPU dla każdego z wykorzystanych w eksperymentach zestawu danych. Szczegółowe pomiary czasu wykonania poszczególnych elementów programu pozwoliły zdiagnozować słabą stronę opracowanego rozwiązania. Była nią duża liczba transferów pamięci pomiędzy CPU a GPU. Częste przesyłanie nawet niewielkiej ilości danych powodowało opóźnienia większe niż czas uzyskany dzięki zrównolegleniu obliczeń. Wyciągnięte wnioski były bezpośrednią przyczyną opracowania przez promotora niniejszej pracy nowego algorytmu pozbawionego opisanych wad. Wspomniane rozwiązanie zostało przedstawione w niniejszym rozdziale. Eksperymenty z nim związane opisano w punkcie 6.3.

Koncepcja przeniesienia elementów algorytmu odkrywające zbiory częste na GPU została przedstawiona w poniższych punktach.

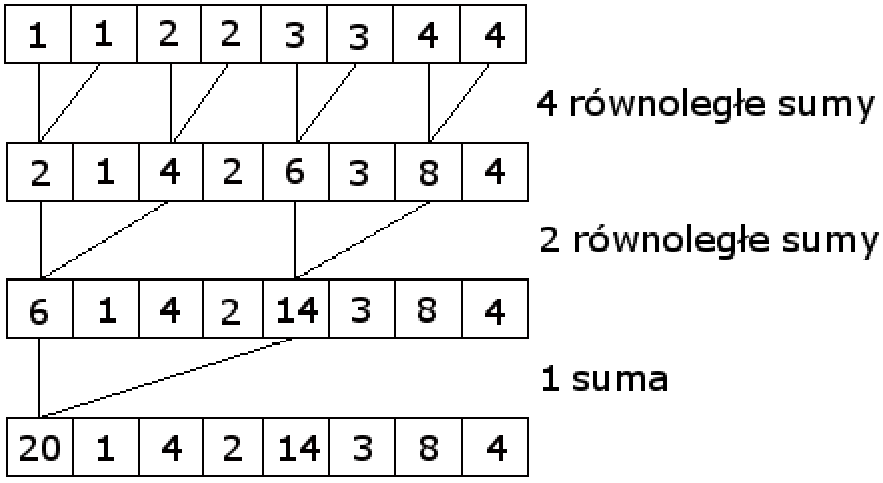
5.4.1. Zliczanie jedynek w wierszach bitmapy

Odbywa się dwuetapowo. Pierwszy etap to niezależne zliczanie jedynek w każdym elemencie wiersza (tak jak w poprzedniej wersji algorytmu). Drugi etap to sumowanie jedynek w wierszu. Liczby jedynek w każdym elemencie otrzymane w poprzednim etapie muszą zostać zsumowane w każdym wierszu. W celu realizacji tego zadania wykorzystano algorytm równoległej agregacji wartości przedstawiony w [12].

Koncepcja algorytmu jest następująca:

- Podziel wiersz na pojedyncze elementy.
- Dodaj każdą parę sąsiednich elementów.
- Dodaj każdą parę elementów występujących co drugą pozycję w tablicy (dodawane mają być elementy, które zawierają sumy obliczone w poprzednim etapie).
- Dodaj każdą parę elementów występujących co 4 pozycje.
- Kontynuuj postępowanie aż w wyniku sumowania pozostanie tylko jeden element.

Powyższy opis został zobrazowany na rysunku 5.3.



Rysunek 5.3. Algorytm równoległej agregacji wartości

Algorytm ten uruchamiany jest równolegle dla wszystkich wierszy. Realizuje go funkcja *sumTable*. Ponieważ wynik sumy zapisywany jest w pierwszym elemencie

sumowanej tablicy, to po wykonaniu opisywanego algorytmu wyniki sum wierszy są rozproszone po tablicy wejściowej. Zachodzi zatem konieczność zebrania wyników sum do osobnej tablicy.

5.4.2. Zebranie wyników sumowania

Ponieważ wiersze mają stałą szerokość, w łatwy sposób można obliczyć, gdzie będą znajdować się obliczone sumy jedynek w wierszach. Przez to w prosty sposób można napisać algorytm równoległy realizujący to zadanie.

Dla każdego elementu tablicy wynikowej (której rozmiar jest równy liczbie wierszy bitmapy) przydzielony jest jeden wątek. Każdy wątek na podstawie położenia w tablicy wynikowej swojego elementu oblicza położenie odpowiedniej sumy w tablicy wynikowej poprzedniego algorytmu. Następnie odczytuje ją i przepisuje do swojego elementu. Algorytm ten zrealizowany jest przez funkcję *gather* (Algorytm 5.28)

Punkty 5.4.1. oraz 5.4.2. tworzą całość algorytmu obliczającego wsparcia w analizowanej bitmapie i są wykonywane za pomocą funkcji *SumOnesInRowsCPU* (patrz punkt 5.6.3.)

5.4.3. Usunięcie wierszy reprezentujących zbiory nieczęste

Kolejnym po obliczeniu wsparcia etapem algorytmu odkrywania zbiorów częstych jest usunięcie z bitmapy wierszy reprezentujących zbiory nieczęste. Zadanie to jest realizowane w kilku etapach i stanowi modyfikację algorytmu kompresji strumieni [13].

W pierwszym etapie dochodzi do sprawdzenia, czy obliczone wsparcia przekraczają minimalny próg. Równoległa realizacja tego etapu polega na uruchomieniu wątku dla każdego elementu tablicy ze wsparciami i sprawdzeniem czy jest on większy od *minsup* czy nie. Wynik tego etapu jest zapisywany do tablicy o rozmiarze o jeden większym niż tablica ze wsparciami (przyczyna takiego stanu rzeczy została opisana w kolejnym punkcie). Do tablicy wynikowej zapisywane jest zero, jeżeli na odpowiadającej pozycji w tabeli wsparcia wartość jest mniejsza od *minsup.* Gdy wartość jest większa lub równa *minsup*, zapisywana jest jedynka. Do nadmiarowego elementu zawsze zapisywane jest zero. Niniejszy punkt realizowany jest przez funkcję ChechValues (Algorytm 5.30).

Kolejnym problemem jest znalezienie indeksów wierszy reprezentujących zbiory częste i zapisanie ich w tablicy. Zadanie to jest realizowane w dwóch etapach. Najpierw znajdowane są indeksy w tablicy wynikowej do której można zapisać poszukiwane indeksy wierszy. W tym celu posłużono się operacją *exclusive scan* [14].

Operację exclusive scan można zdefiniować następująco: mając daną tablicę a[0...n-1] oblicz tablicę b[0...n-1], dla której b[i] = $\sum_{k=0}^{i-1} a_i$. Równoległe algorytmy realizacji tej operacji przedstawiono w [14].

Operacja scan wykonywana jest na tablicy zer i jedynek otrzymanej w poprzednim etapie algorytmu. W wyniku operacji exclusive scan w każdym elemencie tablicy wynikowej otrzymujemy liczbę jedynek, która pojawiła się w tablicy wejściowej na wcześniejszych pozycjach. Przykładowo dla tablicy wejściowej

0	1	0	0	1	1	0	1	0

otrzymamy

0	0	1	1	1	2	3	3	4

Łatwo zauważyć, że na pozycjach, na których w tablicy wejściowej są jedynki (czyli na pozycjach, gdzie w tablicy wsparć są wsparcia większe lub równe minsup) pojawiły się kolejne liczby naturalne, czyli indeksy do których powinny być zapisane indeksy wierszy reprezentujących zbiory częste. Można teraz wyjaśnić, dlaczego w poprzednim punkcie do tablicy zer i jedynek dodano jedną pozycję. Jak można zobaczyć na przykładzie, ostatnia pozycja w tablicy wynikowej (pod warunkiem, że ostatnia pozycja w tablicy wejściowej jest równa zero) zawiera sumaryczną liczbę jedynek, a zatem liczbę zbiorów częstych w bitmapie wejściowej.

Kolejny etap – przepisanie indeksów wierszy do tablicy pomocniczej odbywa się w sposób równoległy. Dla każdego elementu tablicy wsparć uruchamiany jest jeden wątek. Każdy wątek sprawdza, czy w tablicy wsparć na jego pozycji jest wartość większa od minsup. Jeżeli tak, to odczytuje on indeks docelowy z tablicy stanowiącej wynik operacji scan i zapisuje swoją pozycję pod tym indeksem. Zadanie to jest realizowane prze funkcję scatter (Algorytm 5.31).

Ostatnim etapem usuwania „nieczęstych” wierszy jest budowa bitmapy wynikowej na podstawie bitmapy wejściowej i indeksów obliczonych w poprzednim

etapie. Równoległa realizacja tego etapu polega na uruchomieniu dla każdego elementu bitmapy wyjściowej osobnego wątku. Każdy wątek pobiera z tablicy indeksów z pozycji odpowiadającej jego aktualnemu wierszowi numeru wiersza bitmapy, z którego powinien przepisać element z bitmapy wejściowej. Kolumna (numer elementu w wierszu) jest taka sama w obydwu bitmapach (wejściowej i wyjściowej). Po ustaleniu wiersza i kolumny bitmapy wejściowej wątek przepisuje element z tej pozycji do pozycji bitmapy wyjściowej do której został przypisany. Zadanie to realizowane jest przez funkcję gatherRows (Algorytm 5.32).

5.5. Rekurencyjne wyszukiwanie zbiorów częstych

Algorytm według którego wyszukiwano zbiory częste został zmieniony w taki sposób, aby możliwe było ograniczenie czasochłonnych transferów pamięci pomiędzy CPU i GPU. Ideę nowej funkcji rekurencyjnie wyszukującej zbiory częste przedstawiono poniżej.

```
function recMain(level, rows, adjustment)
    if(level < maxRecursion)
        if(level == 0)
            for(i = 0; i < rows –1; ++i)
                frequentItemset[level] = i;
                addToList(level + 1);
                rowAnd(d_data + i * rowSizeIntMultiple,
                        d_data + (i+1)*rowSizeIntMultiple,
                        d_tempData,
                        rowSizeIntMultiple);
                countOnesInIntsCPU(d_tempData,
                                    d_count_Ones,
                                    rows – i – 1);
                sumResult = sumOnesInRowsCPU(rows –i –1);
                resultingNumberOfRows =
                        findFrequentItemsCPU( sumResult,
                                              rows – i – 1,
                                            d_levelsIndexes[level]);
                copyLevelsIndexes();
                gatherRows<1>(d_tempData,
                        NULL,
```

```
                    d_levelsIndexes[level],
                    d_levelsData[level],
                    NULL,
                    rowSizeIntMultiple);
            recMain(level + 1, resultingNumberOfRows, i+1);
        end;
        if(rows >= 1)
            frequentItemset[level] = rows – 1;
            addToList(level + 1);
        end;
else
        copyLevelsRowNumbers();
        for(i = 0; i < rows – 1; ++i)

            frequentItemset[level] =
                h_levelsRowNumbers[level –1][i] + adjustment;
            addToList(level + 1);
            rowAnd(d_levelsData[level – 1] + i*rowSizeIntMultiple,
                d_levels_Data[level–1]+(i+1)*rowSizeIntMultiple,
                d_tempData,
                rowSizeIntMultiple);
            countOnesInIntsCPU(d_tempData,
                d_countOnes,
                rows-i-1);
            sumResult = sumOnesInRowsCPU(rows-i-1);
            resultingNumberOfRows =
                findFrequentItemsCPU(sumResult,
                    rows-i-1,
                    d_leavelsIndexes[level]);
            gatherRows<0>(d_tempData,
                d_levelsRowNumbers[level-1] +(i+1),
                d_levelsIndexes[level],
                d_levelsData[level],
                d_levelsRowNumbers[level],
                rowSizeIntMultiple);
```

```
                    recMain(level + 1, resultingNumberOfRows, adjustment);
                end;
                if(rows >= 1)
                    frequentItemset[level] =
                        h_levelsRowNumbers[level-1][rows-1] + adjustment;
                    addToList(level+1);
                end;
            end;
        end;
endfunction;
```

Algorytm 5.22 Rekurencyjne wyszukiwanie zbiorów częstych według zmienionego algorytmu GPU

W opisie zastosowano następujące oznaczenia:

recMain – funkcja wyszukująca rekurencyjnie zbiory częste

level – zmienna określająca poziom wywołania rekurencyjnego

rows – liczba wierszy aktualnie analizowanej bitmapy

adjustment – zmienna wykorzystywana do prawidłowego obliczania indeksów wierszy w kolejnych wywołaniach rekurencyjnych

maxRecursion – stała określająca maksymalną liczbę wywołań rekurencyjnych. W praktyce jest ona równa maksymalnej długości zbioru częstego znajdującego się w analizowanych danych.

frequentItemset – tablica przechowująca aktualnie analizowany zbiór częsty

rowSizeIntMultiple – stała określająca liczbę wartości *int* w wierszach bitmapy

addToList – funkcja dodająca zbiór częsty do listy wynikowej

rowAnd **–** funkcja obliczająca logiczny AND wiersza ze wszystkimi wierszami reprezentującymi elementy częste

countOnesInIntsCPU – funkcja zliczająca jedynki w poszczególnych elementach wierszy bitmapy

sumOnesInRowsCPU – funkcja zliczająca jedynki w wierszach bitmapy

sumResult – wektor przechowujący wynik działania funkcji *sumOnesInRowsCPU*

resultingNumberOfRows – zmienna określająca liczbę zbiorów częstych uzyskanych jako rezultat działania funkcji findFrequentItemsCPU

findFrequentItemsCPU – funkcja określająca, które wiersze bitmapy reprezentują zbiory częste. Zwraca indeksy tych wierszy.

d_levelsIndexes – tablica alokowana na karcie graficznej przechowująca indeksy wierszy reprezentujących zbiory częste wyszukane przez funkcję *findFrequentItemsCPU*

copyLevelsIndexes – funkcja kopiująca tablicę *d_levelsIndexes* do tablicy *d_levels_RowNumbers*

gatherRows – funkcja scalająca wiersze bitmapy. Przekazywany parametr określa, czy scalać także indeksy (wartość <0> powoduje wywołanie ze scalaniem indeksów, wartość <1> wywołanie bez scalania).

d_levelsData – tablica alokowana na karcie graficznej zawierająca scalone dane na każdym poziomie wywołania rekurencyjnego

copyLevelsRowNumbers – funkcja kopiująca dane z tablicy *d_levelsRowNumbers* alokowanej na karcie graficznej do tablicy *h_levelsRowNumbers* dostępnej dla CPU

Funkcje, które są wykorzystywane w algorytmie 5.22 zostały opisane w punkcie 5.6.

5.6. Funkcje składowe

W podrozdziale przedstawiono opis algorytmów, z których korzystają funkcje wywoływane w trakcie rekurencyjnego wyszukiwania zbiorów częstych według zmienionego algorytmu GPU (Algorytm 5.22).

5.6.1. Funkcja *rowAnd*

Służy do wykonywania logicznej operacji AND wiersza i tabeli przekazanych jako parametry wejściowe. Jest uruchamiana na karcie graficznej algorytmu zamieszczonego poniżej.

```
function rowAnd(inRow, inData, outData, n)
    i = blockDim.x*blockIdx.x + threadIdx.x;
    j = blockIdx.y;
```

```
    if(i < n)
        outData[i+n*j] = inData[i+n*j]&inRow[i];
endfunction;
```

Algorytm 5.23. Funkcja *rowAnd*

W opisie zastosowano następujące oznaczenia:

inRow – wiersz bitmapy

inData – bitmapa wejściowa

outData – bitmapa wynikowa

n – liczba elementów wektora inRow

5.6.2. Funkcja *countOnesInIntsCPU*

Służy do zliczania jedynek w elementach wierszy bitmapy. Wywołuję funkcję countOnesInInts uruchamianą na karcie graficznej. Działa według przedstawionego poniżej algorytmu.

```
function countOnesInIntsCPU(in, out, rows)
    blocks = rowSizeIntMultiple*rows;
    blocks = divRoundUp(blocks,512);
    countOnesInInts<<<blocks,512>>>(in,out,rowSizeIntMultiple*rows);
endfunction;
```

Algorytm 5.24. Funkcja countOnesInIntCPU

W opisie zastosowano następujące oznaczenia:
in – tablica wejsciowa
out – tablica wynikowa
rows – liczba wierszy wejściowej bitmapy
rowSizeIntMultiple – stała określająca liczbę elementów wiersza bitmapy

Funkcja *divRoundUp* dzieli wartość parametru x przez wartość parametru d i zaokrągla wynik do najbliższej, większej wartości całkowitej. Formalnie funkcję zdefiniowano następująco:

```
function divRoundUp(x, d)
    return x/d+((x%d)==0?0:1);
endfunction;
```

Algorytm 5.25. Funkcja divRundUp

Funkcja *countOnesInInts* jest wykonywana na karcie graficznej i określona w następujący sposób:

```
function countOnesInInts(in, out, n)
        i=blockDim.x*blockIdx.x+threadIdx.x;
        if (i<n)
                v=in[i];
                out[i]=dc_lookup[v&0xff]+
                        dc_lookup[(v>>8)&0xff]+
                        dc_lookup[(v>>16)&0xff]+
                        dc_lookup[(v>>24)&0xff];
        end;
endfunction;
```

Algorytm 5.26. Funkcja countOnesInInts

W opisie zastosowano następujące oznaczenia:

v – tablica pomocnicza

dc_lookup – *tablica lookup* alokowana na karcie graficznej

5.6.3. Funkcja *sumOnesInRowsCPU*

Służy do sumowania jedynek w wierszach bitmapy i jest określona w następujący sposób:

```
function sumOnesInRowsCPU(sumResult, rows)
        iter = sumTable(d_countOnes,
                        d_countTemp,
                        rowSizeIntMultiple,
                        rows,
                        rowSizeIntMultiple,
                        countTempRowIntMultiple);
        ungatheredSumResult = iter%2 == 1 ? d_countTemp : d_countOnes;
        sumResult = iter % 2 == 0 ? d_countTemp : d_countOnes;

        gather(ungatheredSumResult,
                sumResult,
                rows,
                iter % 2 == 1 ? countTempRowIntMultiple : rowSizeIntMultiple);
endfunction;
```

Algorytm 5.27 Funkcja sumOnesInRowsCPU

W opisie zastosowano następujące oznaczenia:

sumTable – funkcja obliczająca sumę wartości w wierszach tablicy. Wynik sumy zapisywany jest do pierwszego elementu wiersza.

Funkcja *gather* jest określona w nastepujący sposób:

```
function gather(in, out, n, step)
        i=blockIdx.x*blockDim.x + threadIdx.x;
        if (i<n) {
                out[i]=in[i*step];
```

```
endfunction;
```

Algorytm 5.28. Funkcja gather

Funkcja gather stosowana jest do zebrania wyników dodawania wartości za pomocą funkcji sumTable do jednej tablicy.

W opisie zastosowano następujące oznaczenia:

step – odległość między kolejnymi elementami w pamięci przeznaczonymi do scalenia

5.6.4. Funkcja *findFrequentItemsCPU*

Wyszukuje wiersze, których wsparcie jest większe lub równe wartości określonej przez globalną stałą *minsup*. Do tablicy wynikowej zapisuje indeksy tych wierszy.

```
function findFrequentItemsCPU(sumResult,
                              rows,
                              out,
                              resultingNumberOfRows)
    checkValues(sumResult, d_supportScan, minsup, rows);
    scanCpuPart(d_supportScan,d_temp,rows+1);
    scatter (sumResult, minsup, d_supportScan, out, rows);
    memCopy();
endfunction;
```

Algorytm 5.29. Funkcja findFrequentItemsCPU

W opisie zastosowano następujące oznaczenia:

memCopy – funkcja kopiująca wartość z tablicy d_supportScan z pozycji *rows* do zmiennej *resultingNumberOfRows*

Funkcja *checkValues* jest uruchamiana na karcie graficznej i określona w następujący sposób:

```
function checkValues(in, out, treshold, n)
    i=blockDim.x*blockIdx.x+threadIdx.x;
    if (i<n)
        if (in[i]>=treshold)
            out[i]=1;
        else
            out[i]=0;
        end;
    end;
    if (i==n)
        out[i]=0;
endfunction;
```

Algorytm 5.30. Funkcja checkValues

W opisie zastosowano następujące oznaczenia:

treshold – wartość progowa (wartość minimalnego wsparcia)

Funkcja scatter jest uruchamiana na karcie graficznej i określona w następujący sposób:

```
function scatter(in1, treshold, in2, out, n)
        i=blockDim.x*blockIdx.x+threadIdx.x;

        if (i<n)
                if (in1[i]>=treshold)
                        out[in2[i]]=i;
        end;
endfunction;
```

Algorytm 5.31. Funkcja scatter

5.6.5. Funkcja *gatherRows*

Jest uruchamiana na karcie graficznej i określona w następujący sposób:

```
function gatherRows(in, inNumbers, indexes, out, outNumbers, n)
        i=blockDim.x*blockIdx.x+threadIdx.x;
        j=blockIdx.y;
        k=indexes[j];

        if (i<n) {
                out[i+n*j]=in[i+n*k];
                if (ignoreNumbers!=1)
                        outNumbers[j]=inNumbers[k];
        end;
endfunction;
```

Algorytm 5.32 Funkcja gatherRows

W opisie zastosowano następujące oznaczenia:

ignoreNumbers – wartość określana dla danego wywołania funkcji, która określa czy oprócz budowy wierszy nowej bitmapy ma być również transferowana tablica numerów wierszy tej bitmapy.

6. Eksperymenty

W rozdziale opisano plan oraz wyniki eksperymentów wydajnościowych. Analizie porównawczej zostały poddane wersje CPU oraz GPU opracowanego algorytmu. Ponadto w punkcie 6.1 przedstawiono porównanie algorytmów obliczania wsparcia zbioru reprezentowanego przez wektor danych. Wyniki każdego z eksperymentów zostały przedstawione na wykresie bądź wykresach oraz opatrzone komentarzem.

Wszystkie opisane eksperymenty zostały przeprowadzone na komputerze o następujących parametrach:

procesor: AMD Athlon 3000+ (1.81 Ghz)

karta graficzna: NVIDIA GeForce 9500 GT

pamięć RAM: 2.00 GB

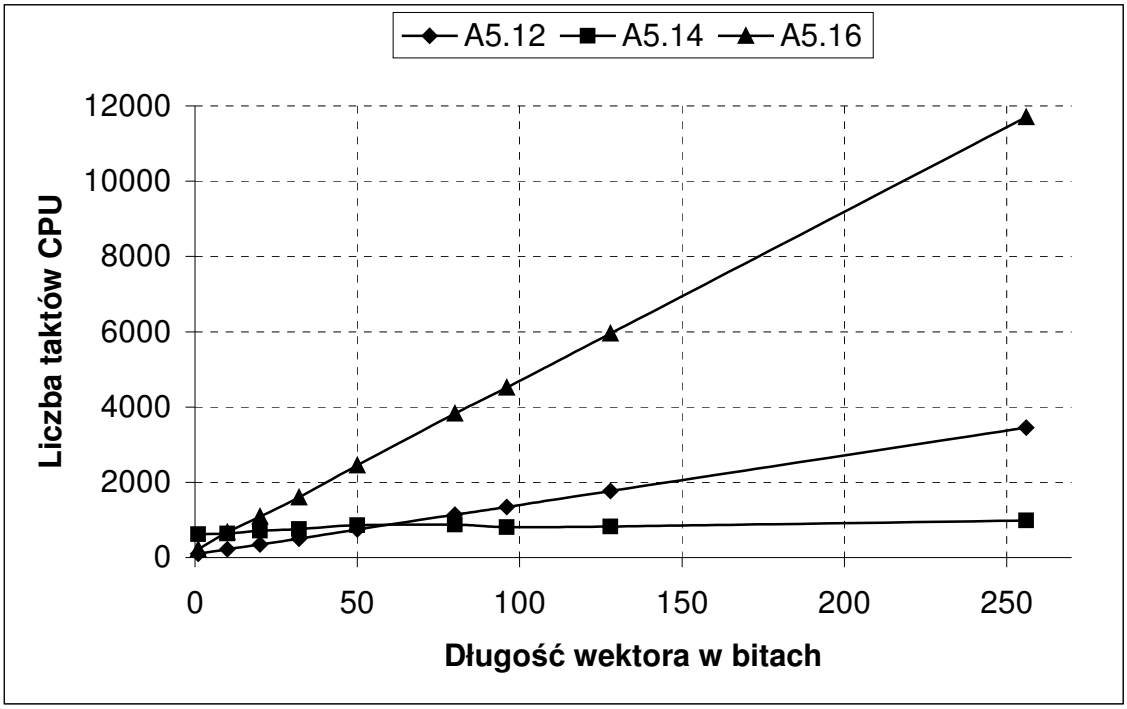
6.1. Algorytmy obliczania wsparcia

Poniżej przedstawiono wydajnościowe porównanie algorytmów 5.12 (zliczanie jedynek w bitmapie reprezentowanej za pomocą zmiennych *bool*) i 5.13 (zliczanie jedynek w bitmapie zawierającej wartości int). Algorytm 4.13 został zrealizowany w dwóch wersjach. Pierwsza wykorzystuje funkcję *count_ones* zaimplementowaną według algorytmu 5.14 (wykorzystanie tablicy lookup), druga zgodnie z algorytmem 5.16 (brak wykorzystania tablicy *lookup*). Dla jasności opisu algorytm działający na tablicy wartości *bool* oznaczono A5.12, natomiast wersje algorytmu wykorzystującego tablicę *int* odpowiednio A5.14 oraz A5.16. Porównanie przeprowadzono dla różnej długości wektora danych. Czas mierzono zgodnie z algorytmem 5.18. Poniżej przedstawiono wyniki pomiarów.

Lp.	**Liczba taktów procesora**			
	N	**A5.12**	**A5.14**	**A5.16**
1	1	101	622	222
2	10	220	635	673
3	20	350	711	1087
4	32	506	758	1603

5	50	747	859	2451
6	80	1137	875	3827
7	96	1343	811	4516
8	128	1770	823	5950
9	256	3450	985	11701

Tabela 6.1 Wyniki pomiarów czasu obliczania wsparcia z użyciem różnych algorytmów



Wykres 6.1 Wyniki pomiarów czasu obliczania wsparcia z użyciem różnych algorytmów

Algorytm A5.12 działający na bitmapie przechowywanej w pamięci z wykorzystaniem zmiennych typu *bool* jest wydajniejszy od algorytmu A5.14 tylko dla małych zestawów danych - długość wektora poniżej 50 bitów. W kontekście niniejszej pracy można mówić o wektorach przechowujących informacje o występowaniu zbiorów w mniej niż 50 transakcjach. Wraz ze wzrostem długości przetwarzanego wektora przewaga algorytmu A5.14 rośnie. Dla 250 – bitowego wektora A5.14 jest około 3.5 razy szybszy od A5.12.

Program realizujący algorytm A5.16 działał najwolniej. Uzyskane przez niego rezultaty należy traktować jako punkt odniesienia do analizy wyników algorytmu A5.14, który także powstał po zmianie sposobu przechowywania bitmapy w pamięci. Rezygnacja z A5.12 była wymuszona. Wyniki przedstawione na wykresie 5.1 pokazują

jednak, że zmiana A5.12 na inne rozwiązanie przyczyniła się do poprawienia wydajności całego programu.

6.2. Porównanie wersji 5.2 i 5.3

Poniżej przedstawiono wyniki eksperymentów przeprowadzonych na programach realizujących założenia opisane w punktach 5.2 (algorytm działający na CPU) i 5.3 (algorytm działający na GPU – pierwsza wersja).

6.2.1 Wpływ liczby transakcji na czas obliczeń

Przygotowano kilka zestawów danych zawierających różną liczbę transakcji. Pozostałe parametry charakteryzujące dane pozostały niezmienione podczas przeprowadzania eksperymentu. Wartości wielkości stałych przedstawiono poniżej.

Liczba różnych elementów w zbiorach: 10

Minimalne wsparcie: 4

Liczba zbiorów częstych: 500

Średnia długość transakcji: 8

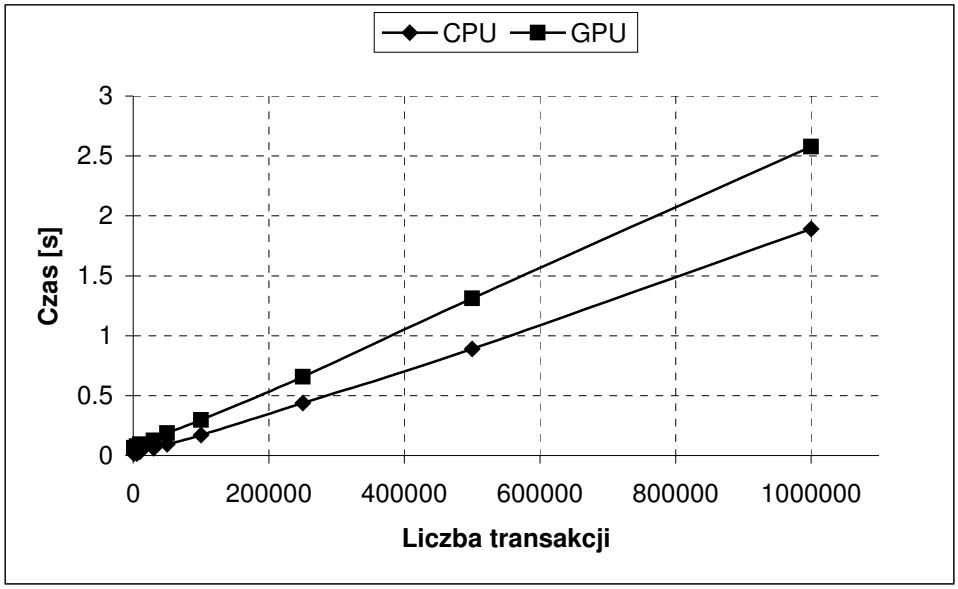
Średnia długość zbioru częstego: 3

Przeprowadzono szereg pomiarów czasu wykonania programu realizującego wyszukiwanie zbiorów częstych według przedstawionego w niniejszej pracy algorytmu. Pomiary wykonano zarówno dla wersji działającej na CPU jak i tej wykorzystującej kartę graficzną. Do zmierzenia czasu wykorzystano metodę zgodną z algorytmem 5.17. Wyniki eksperymentu zestawiono w tabeli oraz przedstawiono na wykresach.

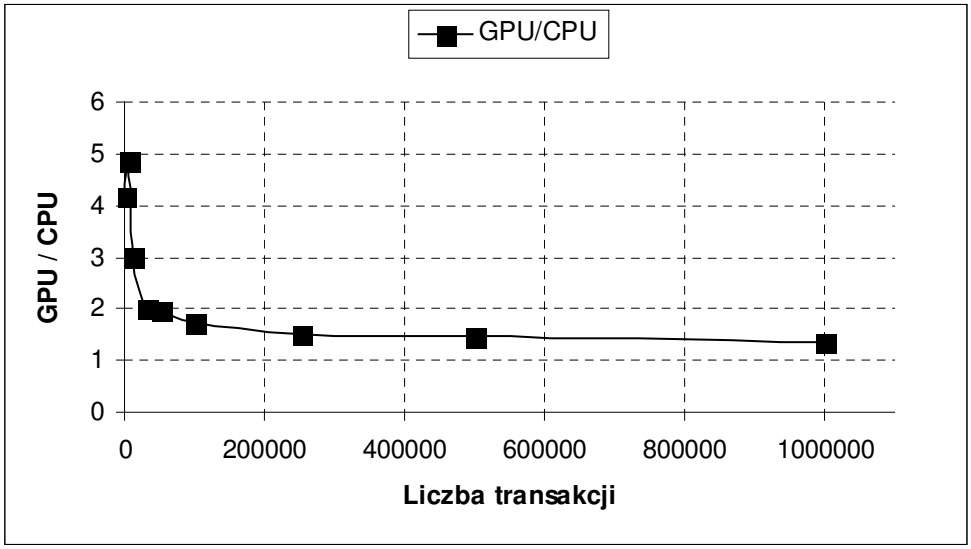
Lp.	**Liczba transakcji**	**Czas wykonania [s]**		$\frac{GPU}{CPU}$
		CPU	**GPU**	
1	1000	0,015	0,063	4,200
2	5000	0,016	0,078	4,875
3	10000	0,031	0,094	3,032
4	30000	0,062	0,125	2,016
5	50000	0,094	0,187	1,989
6	100000	0,172	0,297	1,726

7	250000	0,437	0,657	1,503
8	500000	0,890	1,313	1,475
9	1000000	1,891	2,578	1,363

Tabela 6.2. Czas wykonania programu w zależności od liczby transakcji



Wykres 6.2. Zależność czasu wykonania programu od liczby transakcji



Wykres 6.3. Stosunek czasu wykonania wersji GPU do CPU w zależności od liczby transakcji

Czas wykonania wersji CPU oraz GPU rośnie wraz ze wzrostem liczby transakcji.. Wynika to z faktu, iż większa liczba transakcji oznacza dłuższe wektory

opisujące zbiory znajdujące się w bazie danych. Dłuższe wektory to z kolei dłuższe czasy wykonania na nich logicznych operacji AND.

Wraz ze wzrostem liczby transakcji maleje przewaga wersji uruchamianej całkowicie na CPU. Wynika to z faktu, że dla mniejszych zestawów danych koszt czasowy jaki musimy ponieść na samo uruchomienie funkcji wykonywanej na karcie graficznej oraz transfery pamięci pomiędzy CPU i GPU znacząco niweluje czas zaoszczędzony przez zrównoleglenie obliczeń.

Zestawy danych z większą liczbą transakcji powodują, że wiersze bitmapy mają więcej elementów. Przy wykonywaniu logicznej operacji AND na dłuższych wektorach możliwe jest równoległe wykonywanie większej liczby operacji niż w przypadku wektorów o mniejszej długości. Dzięki temu zysk czasowy otrzymany poprzez zrównoleglenie obliczeń jest bardziej zauważalny.

6.2.2 Wpływ różnorodności bazy danych na czas obliczeń

Przygotowano kilka zestawów danych posiadających różną liczbę unikalnych elementów w zbiorach. Pozostałe parametry charakteryzujące dane pozostały niezmienione podczas przeprowadzania eksperymentu. Wartości wielkości stałych przedstawiono poniżej.

Liczba transakcji: 800

Minimalne wsparcie: 4

Liczba zbiorów częstych: 500

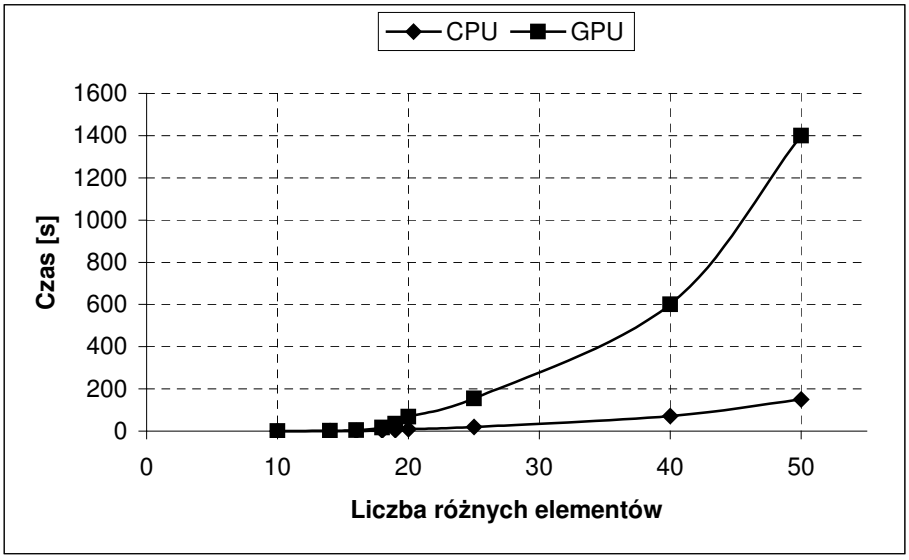
Średnia długość transakcji: 8

Średnia długość zbioru częstego: 3

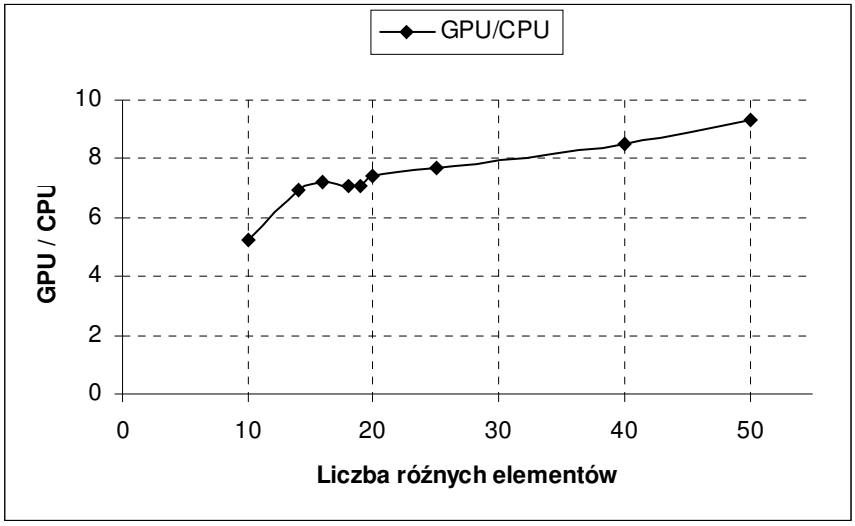
Przeprowadzono szereg pomiarów czasu wykonania programu realizującego wyszukiwanie zbiorów częstych według przedstawionego w niniejszej pracy algorytmu. Pomiary wykonano zarówno dla wersji działającej na CPU jak i tej wykorzystującej kartę graficzną. Do zmierzenia czasu wykorzystano metodę zgodną z algorytmem 5.17. Wyniki eksperymentu zestawiono w tabeli oraz przedstawiono na wykresach.

Lp.	Liczba różnych elementów w zbiorach	Czas wykonania [s]		$\frac{GPU}{CPU}$
		CPU	GPU	
1	10	0,015	0,079	5,267
2	14	0,156	1,079	6,917
3	16	0,609	4,391	7,210
4	18	2,422	17,172	7,090
5	19	4,906	34,656	7,064
6	20	9,219	68,500	7,430
7	25	20,134	155,016	7,699
8	40	70,682	600,233	8,492
9	50	150,026	1400,00	9,333

Tabela 6.3. Czas wykonania programu w zależności od liczby różnych elementów w zbiorach



Wykres 6.4. Zależność czasu wykonania programu od liczby różnych elementów w zbiorach



Wykres 6.5. Stosunek czasu wykonania wersji GPU do CPU w zależności od liczby różnych elementów w zbiorach

Czas wykonania stworzonego programu rośnie wraz ze wzrostem liczby różnych elementów zawartych w bazie danych. Jest to spowodowane faktem, że wraz ze wzrostem liczby elementów w zbiorach rośnie liczba wierszy bitmapy analizowanej w pierwszej iteracji algorytmu. W związku z tym rośnie liczba wywołań funkcji rekurencyjnie wyszukującej zbiory częste. Przyczynia się to do znacznego wydłużenia czasu obliczeń.

Wersja GPU działa wolniej od wersji CPU. Większa liczba wywołań funkcji rekurencyjnie wyszukującej zbiory częste charakterystyczna dla większej liczby różnych elementów bazy danych przyczynia się do zwiększania przewagi wersji CPU w stosunku do wersji GPU wraz ze wzrostem liczby zbiorów jednoelementowych.

6.3. Zmieniony algorytm

Poniżej przedstawiono wyniki eksperymentów przeprowadzonych na programach realizujących założenia z punktu 5.4.

6.3.1 Wpływ liczby zbiorów na czas obliczeń

Przygotowano kilka zestawów danych zawierających różną liczbę zbiorów. Pozostałe parametry charakteryzujące dane pozostały niezmienione podczas przeprowadzania eksperymentu. Wartości wielkości stałych przedstawiono poniżej.

Liczba różnych elementów w zbiorach: 2000

Minimalne wsparcie: 0.1*liczba transakcji

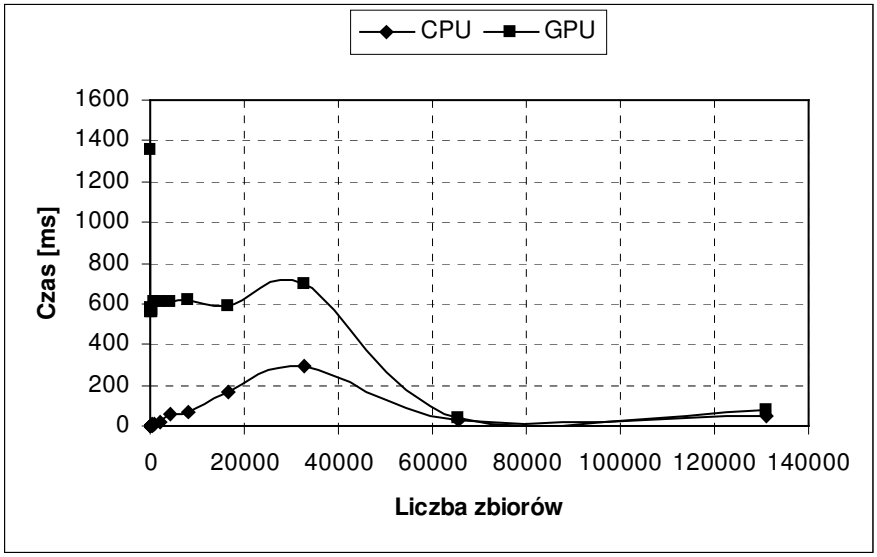
Liczba zbiorów częstych: 50

Wielkość zbioru częstego: 5

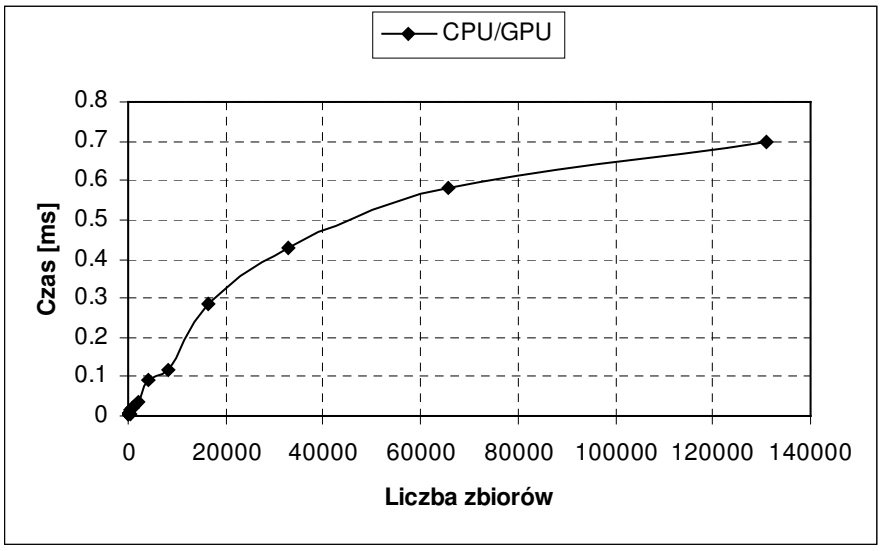
Przeprowadzono szereg pomiarów czasu wykonania programu realizującego wyszukiwanie zbiorów częstych według przedstawionego w niniejszej pracy algorytmu. Pomiary wykonano zarówno dla wersji działającej na CPU jak i tej wykorzystującej kartę graficzną. Wyniki eksperymentu zestawiono w tabeli oraz przedstawiono na wykresach.

Lp.	**Liczba zbiorów**	**Czas wykonania [ms]**		$\frac{CPU}{GPU}$
		CPU	**GPU**	
1	32	4,294	1350,791	0,003
2	64	2,332	562,288	0,004
3	128	2,977	580,644	0,005
4	256	4,167	558,083	0,007
5	512	7,305	564,617	0,013
6	1024	11,120	607,894	0,018
7	2048	21,864	604,347	0,036
8	4096	54,502	605,921	0,090
9	8192	72,663	615,906	0,117
10	16384	167,886	585,524	0,287
11	32768	299,071	699,332	0,428
12	65536	25,38	43,575	0,583
13	131072	52,397	74,873	0,700

Tabela 6.4. Czas wykonania programu w zależności od liczby zbiorów



Wykres 6.6. Zależność czasu wykonania programu od liczby zbiorów



Wykres 6.7. Stosunek czasu wykonania wersji GPU do CPU w zależności od liczby zbiorów

Wersja CPU działała szybciej niż GPU dla całego zakresu danych testowych. Warto zauważyć, że dla danych o dużej liczbie zbiorów różnice były niewielkie. Na wykresie 6.7 widać, że wraz ze zwiększaniem liczby analizowanych zbiorów przewaga wersji CPU maleje.

6.3.2 Wpływ różnorodności bazy danych na czas obliczeń

Przygotowano kilka zestawów danych posiadających różną liczbę unikalnych elementów w zbiorach. Pozostałe parametry charakteryzujące dane pozostały niezmienione podczas przeprowadzania eksperymentu. Wartości wielkości stałych przedstawiono poniżej.

Liczba zbiorów: 131072

Minimalne wsparcie: 0.1*liczba zbiorów

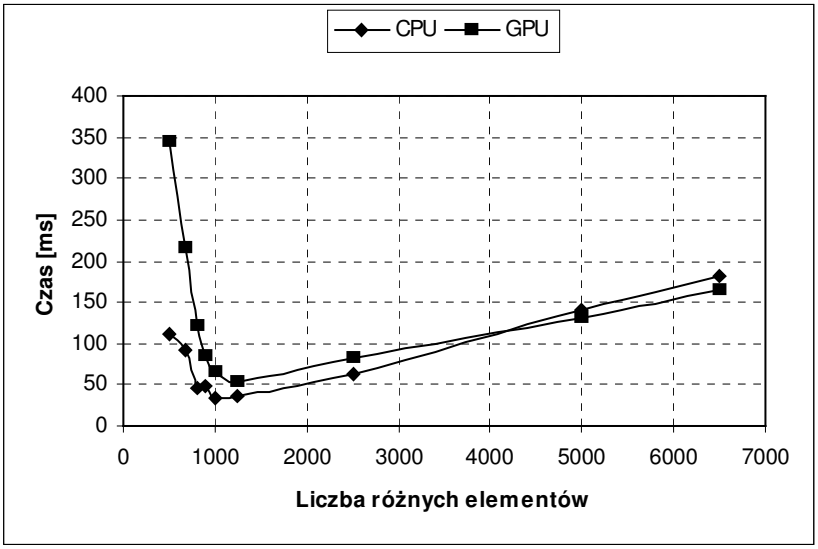
Liczba zbiorów częstych: 50

Wielkość zbioru częstego: 5

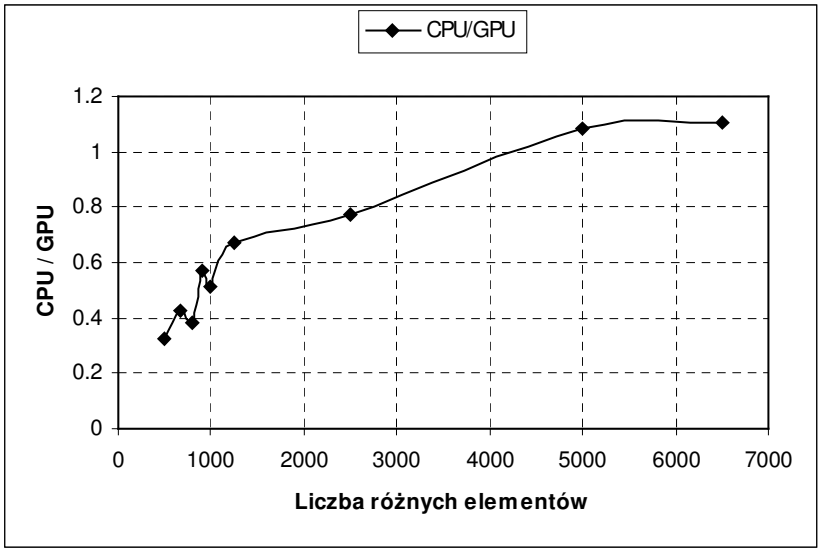
Przeprowadzono szereg pomiarów czasu wykonania programu realizującego wyszukiwanie zbiorów częstych według przedstawionego w niniejszej pracy algorytmu. Pomiary wykonano zarówno dla wersji działającej na CPU jak i tej wykorzystującej kartę graficzną. Wyniki eksperymentu zestawiono w tabeli oraz przedstawiono na wykresach.

Lp.	**Liczba różnych elementów w zbiorach**	**Czas wykonania [s]**		$\frac{CPU}{GPU}$
		CPU	**GPU**	
1	500	112,688	344,379	0,327
2	675	91,899	215,807	0,426
3	800	46,111	120,197	0,384
4	900	47,903	84,290	0,568
5	1000	33,768	65,798	0,513
6	1250	36,316	54,268	0,669
7	2500	63,063	81,375	0,775
8	5000	141,240	160,0179	1,086
9	6500	181,686	164,764	1,103

Tabela 6.5. Czas wykonania programu w zależności od liczby różnych elementów w zbiorach



Wykres 6.8. Zależność czasu wykonania programu od liczby różnych elementów w zbiorach



Wykres 6.9. Stosunek czasu wykonania wersji GPU do CPU w zależności od liczby różnych elementów w zbiorach

Na wykresie 6.8. widać, że wersja CPU działa szybciej niż GPU dla danych o mniejszej liczbie różnych elementów w zbiorach. Dla danych o liczebności różnych elementów większej lub równej ok. 4500 szybsza jest wersja GPU.

Wykres 6.9. obrazuje zmianę stosunku czasu wykonania CPU do GPU w zależności od liczby różnych elementów w zbiorach.

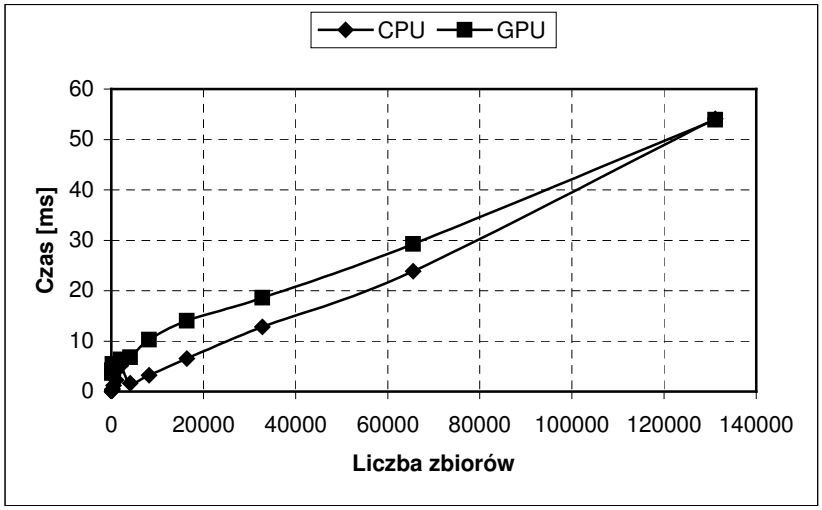
6.3.3. Wpływ liczby zbiorów na czas obliczeń (pomiar bez rekurencji)

Wyniki eksperymentów 6.3.1 oraz 6.3.2 sugerują, że czas wykonania programu był mocno uzależniony od rodzaju danych testowych jakich użyto w eksperymentach. Aby uniezależnić wyniki pomiarów od jakościowej struktury danych testowych przeprowadzono eksperymenty zgodne z założeniami z punktu 6.3.1. i 6.3.2. Różnica

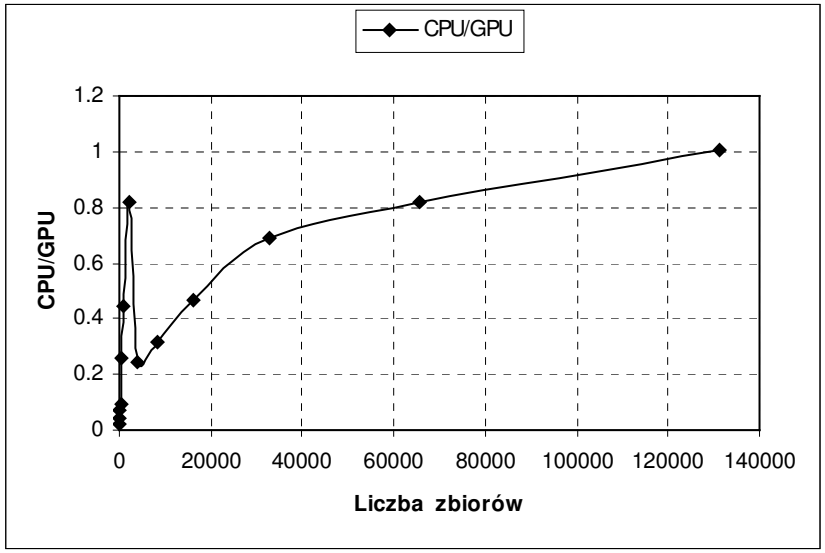
polega na tym, że mierzono czas wykonania programu bez uruchamiania rekurencji. Wyniki eksperymentów przedstawiono odpowiednio w punktach 6.3.3. oraz 6.3.4.

Lp.	Liczba zbiorów	Czas wykonania [ms]		$\frac{CPU}{GPU}$
		CPU	GPU	
1	32	0,086	4,178	0,021
2	64	0,152	3,671	0,041
3	128	0,268	3,737	0,072
4	256	0,518	5,447	0,095
5	512	1,777	4,501	0,261
6	1024	2,268	5,096	0,445
7	2048	5,173	6,319	0,819
8	4096	1,657	6,779	0,244
9	8192	3,238	10,287	0,315
10	16384	6,521	14,045	0,464
11	32768	12,815	18,581	0,690
12	65536	23,876	29,268	0,816
13	131072	54,096	53,930	1,003

Tabela 6.6. Czas wykonania programu w zależności od liczby zbiorów



Wykres 6.10. Zależność czasu wykonania programu od liczby zbiorów



Wykres 6.11. Stosunek czasu wykonania wersji CPU do GPU w zależności od liczby zbiorów

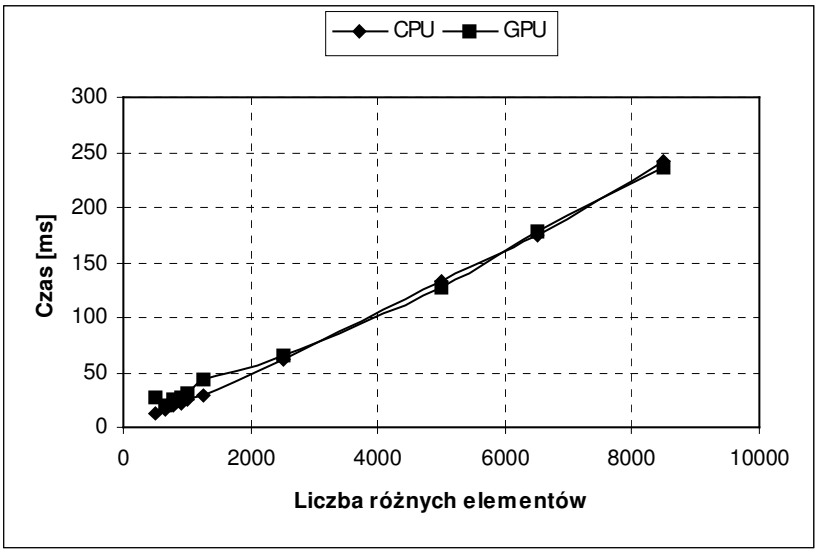
Pomiar czasu realizacji pojedynczej iteracji algorytmu pozwolił na uwidocznienie faktu, iż czas wykonania wersji CPU rośnie szybciej niż czas wykonania wersji GPU dla analogicznych danych. Dla ostatniego punktu pomiarowego (o liczbie zbiorów równej 131072) wersja GPU była nieznacznie szybsza od CPU.

6.3.4 Różnorodność bazy danych, a czas obliczeń (pomiar bez rekurencji)

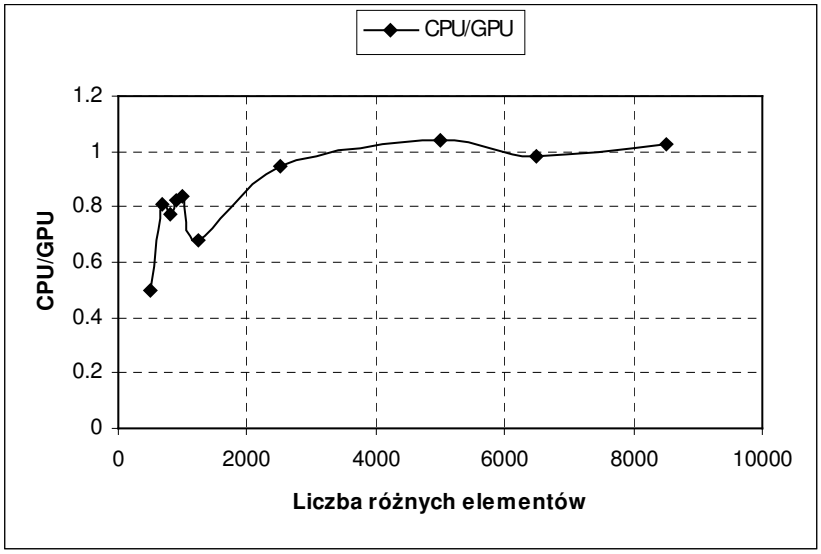
Lp.	Liczba różnych elementów w zbiorach	Czas wykonania [s]		$\frac{CPU}{GPU}$
		CPU	GPU	
1	500	13,426	26,777	0,501
2	675	16,527	20,335	0,813
3	800	19,346	25,081	0,771
4	900	21,839	26,569	0,822
5	1000	25,086	30,03	0,835
6	1250	29,856	44,168	0,676
7	2500	61,927	65,452	0,946
8	5000	132,52	127,106	1,043
9	6500	174,937	178,484	0,980

10	8500	241,79	235,799	1,025

Tabela 6.7. Czas wykonania programu w zależności od liczby różnych elementów w zbiorach



Wykres 6.12. Zależność czasu wykonania programu od liczby różnych elementów w zbiorach



Wykres 6.13. Stosunek czasu wykonania wersji GPU do CPU w zależności od liczby różnych elementów w zbiorach

Czas wykonania pojedynczej iteracji algorytmu rośnie liniowo w zależności od liczby różnych elementów w zbiorach prawie dla całego badanego obszaru danych wejściowych. Dla danych, które posiadają co najmniej ok. 4500 różnych elementów w zbiorach wersja GPU działa szybciej lub porównywalnie szybko jak wersja CPU.

7. Podsumowanie i plany dalszych badań

Uzyskane w punkcie 6.2. wyniki eksperymentów wraz z analizą czasów wykonania poszczególnych elementów algorytmu opisanego w punktach 5.2 i 5.3 pozwoliły stwierdzić, że przy projektowaniu rozwiązań wykorzystujących karty graficzne do obliczeń należy tworzyć algorytmy wymagające minimalnej liczby transferów pamięci pomiędzy CPU i GPU. Transfery te bowiem powodują opóźnienia, które mogą zniwelować zysk czasowy powstały w wyniku zrównoleglenia operacji.

Uzyskane w punkcie 6.3. wyniki potwierdzają wnioski wyciągnięte na podstawie wyników z punktu 6.2. Widać przy tym, że wersja GPU algorytmu działa szybciej od wersji CPU dla dużych zestawów danych (duża liczba zbiorów i/lub duża liczba różnych elementów w bazie danych). Dla małych zestawów danych bardziej wydajna jest wersja CPU.

Eksperymenty opisane w punkcie 6 zostały wykonane z użyciem karty graficznej posiadającej 4 multiprocesory. Aktualnie na rynku dostępne są karty, które mają nawet 30 multiprocesory. Należy spodziewać się, że uruchomienie opisanych w niniejszej pracy programów na sprzęcie z większą liczbą multiprocesorów spowoduje znaczące poprawienie wyników uzyskiwanych przez wersje GPU.

Wnioski wyciągnięte na podstawie przeprowadzonych eksperymentów pozwalają stwierdzić, że optymalnym programem do wyszukiwania zbiorów częstych byłby taki, który analizowałby wstępnie dane wejściowe. Następnie dla małych zestawów danych uruchamiałby wersję CPU opisanego w niniejszej pracy algorytmu, a dla zestawów dużych wersję GPU. Takie podejście pozwoliłoby na wykorzystanie zalet obu rozwiązań przy jednoczesnym wyeliminowaniu ich wad.

W ramach planów dalszych badań proponuje się optymalizację podejścia przedstawionego w niniejszej pracy przy uwzględnieniu wymagań wykonywania wydajnych dostępów do pamięci karty graficznej. Drugim kierunkiem rozwoju może być przeniesienie innych algorytmów odkrywania zbiorów częstych (np. FP-growth) na GPU.

Załączniki

Płyta CD zawierająca:

- kody źródłowe opisanych w niniejszej pracy programów,
- elektroniczną wersję pracy

Bibliografia

1. http://pl.wikipedia.org/wiki/Eksploracja_danych#Techniki_eksploracji_danych, widziane dnia 2009-09-07

2. http://dms.irb.hr/tutorial/tut_assoc_rules.php, widziane dnia 2009-09-07

3. Witold Andrzejewski, Zbyszko Królikowski, Mikołaj Morzy, Tadeusz Morzy „Hierarchiczny indeks bitmapowy wspierający wykonywanie zapytań w bazach danych z atrybutami zawierającymi zbiory"

4. http://mediawiki.ilab.pl/images/c/c3/ED-4.2-m03-1.0.pdf, widziane dnia 2009-09-07

5. http://osilek.mimuw.edu.pl/index.php?title=ED-4.2-m06-1.0-Slajd6, widziano dnia 2009-09-07

6. http://74.125.77.132/search?q=cache:pUzXHzM5K4QJ:www.florian.verhein.com/teaching/2008-01-09/fp-growth-presentation_v1%2520(handout).pdf+FP-growth+disadvantages&cd=2&hl=pl&ct=clnk, widziano dnia 2009-09-10

7. http://pclab.pl/art28888-3.html, widziano dnia 2009-09-23

8. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0

9. NVIDIA CUDA Programming Guide, Version 2.3

10. http://data-mining.wyklady.org/wyklad/305_reguly-asocjacyjne-reprezentowanie-danych-i-hipotez.html, widziano dnia 2009-09-10

11. http://www.cplusplus.com/reference/clibrary/ctime/clock/, widziano dnia 2009-09-21

12. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

13. http://www.amazon.com/exec/obidos/tg/detail/-/0321335597/ref=pd_sim_b_2/102-8196576-8468109?_encoding=UTF8&v=glance

14. mgarland.org/files/papers/nvr-2008-003.pdf