

POLITECHNIKA POZNAŃSKA
WYDZIAŁ INFORMATYKI
INSTYTUT INFORMATYKI

PRACA DYPLOMOWA INŻYNIERSKA

Implementacja algorytmu eksploracji danych z użyciem CUDA API

Marcin Jabłoński

Łukasz Kosiak

Piotr Kurzawa

Marek Rydlewski

Promotor:
dr inż. Witold ANDRZEJEWSKI

Poznań, 2017 r.

*„Coś się popsło“
Zbigniew Stonoga*

Spis treści

1	Wstęp	4
1.1	Wprowadzenie	4
1.2	Cel i zakres pracy	5
1.3	Charakterystyka źródeł	5
1.4	Struktura pracy	6
1.5	Podział pracy	6
2	Podstawy teoretyczne	7
2.1	Charakterystyka danych przestrzennych	7
2.1.1	Modelowanie danych przestrzennych	7
2.1.2	Źródła danych przestrzennych	7
2.1.3	Relacje	7
2.2	Metody eksploracji danych przestrzennych	8
2.2.1	Grupowanie przestrzenne	8
2.2.2	Klasyfikacja przestrzenna	8
2.2.3	Odkrywanie trendów przestrzennych	9
2.2.4	Przypadki osobliwe w danych przestrzennych	9
2.2.5	Asocjacje przestrzenne	9
2.2.6	Kolokacje przestrzenne	10
2.3	Odkrywanie kolokacji przestrzennych	10
2.3.1	Cecha przestrzenna	11
2.3.2	Podstawowe definicje	11
2.3.3	Miary kolokacji	12
2.3.4	Problem	13
2.4	Przegląd algorytmów odkrywania wzorców kolokacji przestrzennych	13
2.4.1	Co-location Miner	14
2.4.2	Multiresolution Co-location Miner	14
2.4.3	Joinless	14
2.4.4	iCPI-tree	15
3	Wprowadzenie do technologii CUDA	16
3.1	CPU a GPU	16
3.2	Architektura CUDA	17
3.3	Programowanie wielowątkowe w języku CUDA C	17
3.4	Model pamięci	18
3.5	<i>Thrust</i>	18
4	Algorytm <i>SGCT</i>	20
4.1	Generowanie tabeli instancji kolokacji o rozmiarze 2	20
4.2	Obliczanie miary powszechności	20
4.3	Generowanie kandydatów na kolokacje maksymalne	20
4.4	Proces odcinania	22
5	Implementacja CPU	24
5.1	Wersja sekwencyjna	24
5.1.1	Generowanie instancji w oparciu o próg odległości	24
5.1.2	Filtrowanie sąsiadów w oparciu o próg minimalnej powszechności	25

5.1.3	Generowanie kandydatów na kolokacje maksymalne	25
5.1.4	Generowanie skondensowanych drzew instancji oraz filtrowanie kandydatów na podstawie progu powszechności	25
5.2	Wersja wielowątkowa	26
5.2.1	Generowanie sąsiadów na podstawie progu odległości	27
5.2.2	Filtrowanie sąsiadów na podstawie progu minimalnej powszechności	28
5.2.3	Generowanie kandydatów na kolokacje maksymalne	28
5.2.4	Generowanie skondensowanych drzew instancji oraz filtrowanie kandydatów na podstawie progu powszechności	28
6	Implementacja GPU	30
6.0.1	Przygotowanie danych w pamięci GPU	30
6.0.2	Generowanie tabeli instancji o rozmiarze 2	30
6.0.3	Przygotowanie struktury szybkiego dostępu do informacji o instancjach należących do relacji o zadanych typach	32
6.0.4	Generowanie przewalentnych relacji sąsiedztwa pomiędzy typami połączeń	33
7	Testy efektywnościowe	35
8	Zakończenie	35
	Bibliografia	36
A	Opis klas i struktur pomocniczych	39
A.1	Opis architektury wersji CPU	39
A.2	Opis struktur & klas pomocniczych i ich implementacji	40

1 Wstęp

1.1 Wprowadzenie

Informatyzacja życia codziennego, jaka dokonała się w ostatnich latach sprawiła, że każdego dnia konsumenci często nieświadomie zostawiają po sobie wiele informacji na swój temat. Nawet z pozoru niewinne dane o ludzkich przyzwyczajeniach typu "z której półki bierzemy bułki w sklepie" są zapisywane w systemach informatycznych. Należy do tego oczywiście dodać inne usługi, które wybierane są przez użytkowników świadomie np. zapisywanie lokalizacji przez prywatny telefon komórkowy.

Wbrew pozorom, taka błaża na pierwszy rzut oka informacja może mieć jednak istotne znaczenie dla funkcjonowania przemysłu piekarskiego. Nic nie stoi na przeszkodzie, aby spróbować z tych danych odczytać preferencje bądź przyzwyczajenia przeciętnego Kowalskiego na temat jego codziennych zakupów, które mogą w przyszłości zaprocentować - zarówno dla właściciela, jak i klienta. Jest to oczywiście tylko przykład, ale oddaje doskonale fakt przydatności z pozoru nie mających znaczenia prostych czynności człowieka, jakie często przypadkiem rejestrują działające wokół konsumentów systemy.

Pozostaje jednak problem przetworzenia takich danych w celu otrzymania interesującej informacji, która byłaby potencjalnie użyteczna. Trzeba pamiętać, że rozmiar takich danych nierzadko sięga terabajtów i w praktyce skuteczna analiza takich danych przez człowieka nie jest możliwa. Musi on zatem w tym celu skorzystać z dobrodziejstw, jakie przynosi mu współczesna technologia.

Problem efektywnego przetwarzania zdążył urosnąć do rangi oddzielnego działu w informatyce. W pracy [1] zasugerowano utworzenie nowej dyscypliny mającej na celu opracowanie technik obliczeniowych rozwiązujących takie problemy, zwanej odkrywaniem wiedzy w bazach danych (ang. KDD – *Knowledge Discovery in Databases*). Techniki te mają na celu odnajdywanie prawidłowych, nietrywialnych i potencjalnie użytecznych wzorców w dużych zbiorach danych.

Wspominane wyżej techniki w dużej mierze zależą od rodzaju bazy, a ściślej mówiąc - charakteru danych w niej występujących. W przypadku danych zawierających informację o położeniu zazwyczaj mowa jest o odkrywaniu wiedzy w bazach danych przestrzennych (ang. *spatial data mining*). Takie systemy mogą zawierać atrybut lokalizacji obiektu w danym obszarze, jego opis w formie geometrycznej (np. w postaci wielokątów), a także inne atrybuty nieprzestrzenne. Okazuje się, że tradycyjne metody analizy danych przestrzennych zazwyczaj nie radzą sobie z nimi na tyle efektywnie, by było opłacalne ich użycie w praktyce [2], dlatego też zaczęto szukać nowych sposobów na odkrywanie wiedzy w takich bazach.

W pracy [3] zaproponowano odkrywanie *wzorców kolokacji przestrzennych* (lub krócej: *kolokacji*), czyli zbioru cech przestrzennych występujących w niewielkiej odległości od siebie. Łatwo to można sobie wyobrazić na przykładzie przyrody, gdzie osobniki (gatunki) o podobnych cechach zazwyczaj trzymają się razem. Rozumowanie to działa również dla bliższych współczesnemu człowiekowi cech przestrzennych, np. punktach o podobnej funkcji - stacje, kina, piekarnie, itd. Wraz z rosnącą popularnością obliczeń na kartach graficznych (w dużej mierze spowodowana wprowadzeniem technologii *CUDA* autorstwa firmy NVIDIA) pojawiło się wiele gotowych rozwiązań, pozwalających na efektywne wyszukiwanie kolokacji nawet w bardzo rozbudowanych bazach danych. Przegląd niektórych z nich można znaleźć w pracy [4].

Ostatni rok przyniósł kolejną metodę efektywnego przeszukiwania baz danych w

celu odnalezienia kolokacji [5]. Wykorzystuje ona autorski algorytm wyszukiwania maksymalnych klik w grafie rzadkim oraz skondensowane drzewa instancji przechowywujące kliki instancji dla każdego kandydata do kolokacji (patrz Rozdział 4) w celu zmniejszenia czasu obliczeń oraz ograniczenia wymagań co do pamięci operacyjnej. Algorytm ten jest przedmiotem badań niniejszej pracy zbiorowej.

1.2 Cel i zakres pracy

Celem niniejszej pracy jest analiza wydajności zaproponowanych w pracy [5] rozwiązań z zakresu odkrywania kolokacji przestrzennych dla GPU i CPU.

Zakres pracy obejmuje następujące zadania szczegółowe:

1. **Zapoznanie się z literaturą.** Zapoznanie się z podstawowymi pojęciami dotyczącymi odkrywania danych w bazach danych przestrzennych oraz wyszukiwania wzorców kolokacji przestrzennych jest niezbędne do stworzenia działającej implementacji powyższego algorytmu. Dodatkowo należy zwrócić uwagę na dodatkowe zagadnienia związane z teorią grafów.
2. **Opracowanie wersji równoległej algorytmu eksploracji danych.** Konieczne jest przemyślenie wykorzystania algorytmów pomocniczych dla poszczególnych kroków całego rozwiązania oraz zaproponowanie możliwie najkorzystniejszego rozwiązania biorąc pod uwagę dostępną pamięć operacyjną, czas przetwarzania i przesyłania danych między pamięcią operacyjną a pamięcią karty graficznej.
3. **Implementacja wersji sekwencyjnej i równoległej ww. algorytmu.** Rozwiązanie podane w punkcie drugim powinno zostać zaimplementowane w technologii NVIDIA CUDA dla wersji GPU oraz biblioteki PPL w przypadku odmiany dla CPU.
4. **Przeprowadzenie eksperymentów wydajnościowych.** Analiza wyników testów wydajnościowych implementacji z punktu 3 jest głównym celem tej pracy. Należy zbadać efektywność obu rozwiązań pod względem czasu wykonywania oraz zapotrzebowania na dostępną pamięć.

1.3 Charakterystyka źródeł

Jak już wspomniano, niniejsza praca w dużej mierze opiera się o algorytm zaprezentowany w dokumencie [5]. Do jej opracowania była wymagana wiedza zawarta w innych źródłach, często również o charakterze naukowym.

Głównym źródłem wiedzy na temat kolokacji przestrzennych była rozprawa doktorska dr inż. Pawła Boińskiego [4], która w dużym przekroju omawia ideę kolokacji zaprezentowaną przez Shekhara i Huanga w pracy [3], a także prezentuje najpopularniejsze techniki ich odkrywania (metody *Co-location Miner*, *iCPI-tree*). Część rozwiązań wykorzystanych w tych technikach została wykorzystana w trakcie realizacji algorytmu.

Oddzielną kwestią jest literatura książkowa, wykorzystana do zapoznania się z technologią CUDA oraz przyjęcia dobrych praktyk optymalizacyjnych i programistycznych. Tutaj szczególnie należy wymienić popularną pozycję *CUDA w przykła-*

dach autorstwa Shane’a Cooke’a [6], a także *Professional CUDA C Programming* [7] będącą również podstawą do wstępu teoretycznego w Rozdziale 3.

1.4 Struktura pracy

Rozdział 2 stanowi wstęp teoretyczny do niniejszej pracy - prezentuje podstawowe pojęcia związane z odkrywaniem kolokacji przestrzennych, a także prezentuje najprostsze algorytmy wyszukujące wzorce kolokacji przestrzennych.

Rozdział 3 zawiera wprowadzenie do technologii CUDA. Przedstawia także podstawowe różnice w programowaniu na CPU i GPU oraz wyjaśnia podstawowe pojęcia związane z programowaniem na procesory graficzne.

Rozdział 4 poświęcony jest algorytmowi *SGCT* będącym głównym tematem pracy. Zawiera wyjaśnienia poszczególnych kroków algorytmu oraz niezbędne definicje.

Implementacja CPU (w wersji sekwencyjnej, jak i równoległej) jest głównym tematem **Rozdziału 5**. Koncentruje się on w szczególności na przedstawienie zastosowanych technik optymalizacyjnych poprawiających wydajność algorytmu w środowisku CPU. Rozdział zawiera także krótki wstęp do biblioteki *PPL* użytej w równoległej implementacji algorytmu.

Rozdział 6 zawiera omówienie implementacji algorytmu dla procesorów graficznych (GPU).

Porównanie wydajności zaimplementowanych rozwiązań zostało zawarte w **Rozdziale 7**.

Ostatni **Rozdział 8** stanowi podsumowanie całego projektu, uwagi zespołu oraz możliwe dalsze kierunki rozwoju.

W **Dodatku A** zawarty jest krótki opis klas zaimplementowanych w implementacji dla CPU i GPU.

1.5 Podział pracy

Podział zadań w ramach niniejszego projektu wyglądał następująco:

- **Marcin Jabłoński** - równoległa implementacja CPU, testy;
- **Łukasz Kosiak** - implementacja GPU, testy;
- **Piotr Kurzawa** - tekst pracy inżynierskiej, testy;
- **Marek Rydlewski** - sekwencyjna i równoległa implementacja CPU, testy.

2 Podstawy teoretyczne

2.1 Charakterystyka danych przestrzennych

2.1.1 Modelowanie danych przestrzennych

Sposób reprezentacji danych przestrzennej w dużej mierze zależy od zastosowań, niemniej najczęściej przybiera jedną z następujących form:

- *model pól* - ma formę funkcji, której dziedziną należy do modelowanej przestrzeni, a jego wynikiem jest cecha przestrzenna;
- *model obiektowy* - dla każdego zjawiska jest tworzony nowy obiekt z odpowiednimi właściwościami (etykietami, atrybutami przestrzennymi i nieprzestrzennymi).

W praktyce model pól używany jest przede wszystkim w metodach opartych na dokonywaniu pomiarów z powietrza - takie dane mają wtedy charakter rastrowy (reprezentacja w postaci pikseli). Model obiektowy stosowany jest natomiast w przypadkach, gdzie występuje duża liczba dodatkowych atrybutów nieprzestrzennych.

2.1.2 Źródła danych przestrzennych

Najogólniej źródła danych przestrzennych można podzielić ze względu na ich format.

Pierwotne źródła danych są opracowane w jednym ze standardowych formatów źródeł (najczęściej dla konkretnego systemu) i nie wymagają jakichkolwiek transformacji. Mają one zazwyczaj postać cyfrową i pochodzą z automatycznych pomiarów dokonanych przez specjalizowane systemy wyposażone w odbiorniki GPS czy tachimetry.

Wtórne dane źródłowe nie zostały zebrane z myślą o wykorzystaniu w *systemach informacji geograficznej* (ang. *Geographic Information System*, GIS) i dlatego wymagają one odpowiedniej transformacji oraz cyfryzacji (jeżeli są one analogowe). Procedury te są one obciążone pewnym ryzykiem, ponieważ istnieje możliwość wystąpienia błędów w trakcie konwersji i w konsekwencji przekłamaniami w danych wynikowych, które należy ręcznie poprawić.

2.1.3 Relacje

Określenie zachodzących relacji między obiektami w źródłach danych przestrzennych jest ważnym elementem przetwarzania danych przestrzennych. Sposób ich określenia zależy od zastosowanego modelu danych.

W modelu pól relacje determinowane są przez operacje pól (ang. *field operations*, [12]), mogące przybierać różne formy w zależności od zastosowań, natomiast w modelu obiektowym rodzaje relacji przestrzennych zależą od definicji przestrzeni. Według standardu organizacji OGC (*Open Geospatial Consortium*) istnieją trzy najpopularniejsze rodzaje związków przestrzennych między obiektami:

- *Relacje metryczne* - wyrażane w postaci predykatów typu "w odległości nie większej niż 10 metrów", oparte na odległości;

- *Relacje kierunkowe* - położenie określone jest względem globalnych kierunków dla przestrzeni (np. na północ, na południe - są to relacje bezwzględne) lub względem innego obiektu/obserwatora (nazywamy takie relacjami względnymi);
- *Relacje topologiczne* - najbardziej skomplikowane, wyrażone przez zależności typu pokrywanie, zawieranie, styczność.

W systemach typu GIS stosuje się głównie relacje topologiczne. Mają one postać predykatów przestrzennych dla operacji filtrowania i połączenia przestrzennego w językach zapytań działających na danych przestrzennych. Najczęściej wykorzystuje się je w tzw. *modelu dziewięciu przecięć* [13], za pomocą którego określa się możliwe relacje zachodzącą dla pary obiektów.

Dla każdego obiektu wyznacza się jego wnętrze, granicę i zewnątrz. Następnie, dokonuje się operacji przecięcia dla danej pary obiektów dla każdej z możliwych kombinacji elementów tego obiektu (np. granica pierwszego obiektu z wnętrzem drugiego). Takich relacji w dwuwymiarowej relacji można wyznaczyć osiem, należących do nich np. rozłączność, styczność, częściowe i całkowite pokrycie itd.

Istnieje również rozszerzenie modelu dziewięciu przecięć, zwanym DE-9IM (ang. *Dimensionally Extended nine-Intersection Model*, [14]), które rozróżnia rodzaj obiektu uzyskanego w wyniku przecięcia (mogą być puste, bezwymiarowe, jednowymiarowe i dwuwymiarowe).

2.2 Metody eksploracji danych przestrzennych

Specyfika danych przestrzennych, a w szczególności fakt, że własności obiektu w danych przestrzennych mogą zależeć od cech jego sąsiadów, powoduje, że stosowanie klasycznych metod eksploracji danych może doprowadzić do nieprawidłowych wyników [15] [16] - stąd też istnieje konieczność korzystania z metod eksploracji dedykowanych dla danych przestrzennych. Wiele z nich jest tak naprawdę rozwinięciem metod opracowanych dla klasycznych zbiorów danych.

2.2.1 Grupowanie przestrzenne

Metoda grupowania przestrzennego (ang. *spatial clustering*) zakłada istnienie przestrzeni m -wymiarowej, w której znajdują się punkty odpowiadające obiektom. Przestrzeń ta ma rozkład niejednorodny, a każdy z obiektów jest opisany przez m atrybutów. Celem grupowania jest poszukiwanie gęstych obszarów punktów używając w tym celu metryki Euklidesowej jako funkcji podobieństwa (bliskości).

Grupowanie przestrzenne w największym stopniu spośród wszystkich metod eksploracji danych przestrzennych jest podobna do swojego klasycznego odpowiednika - wiele algorytmów grupowania opracowanych dla klasycznych zbiorów danych działa również dla danych przestrzennych.

2.2.2 Klasyfikacja przestrzenna

Klasyfikacja przestrzenna (ang. *spatial classification*) działa podobnie jak jego odmiana dla danych klasycznych - przewiduje klasy nowych obiektów w oparciu o tzw. zbiór uczący, składający się ze wcześniejszych obserwacji.

W celu dostosowania klasyfikacji dla danych przestrzennych zaproponowano [17] wykorzystanie *grafu sąsiedztwa* będącego reprezentacją relacji przestrzennych między obiektami, w którym wierzchołki stanowią obiekty przestrzenne, a relacje są krawędziami. Następnie w takim grafie wyznaczane są wszystkie ścieżki, których początkiem jest analizowany obiekt. Dalej analiza przebiega zgodnie z algorytmem *ID3* [18].

Z pojęciem klasyfikacji przestrzennej wiąże się także predykcja położenia (ang. *location prediction*), czyli przewidywanie zdarzeń we wskazanym miejscu w przestrzeni, przy uwzględnieniu autokorelacji przestrzennej. Przydaje się ono np. w określaniu regionów o wysokim ryzyku wystąpienia klęsk żywiołowych czy awarii.

2.2.3 Odkrywanie trendów przestrzennych

Trend przestrzenny definiuje się [19] jako regularną zmianę co najmniej jednego atrybutu nieprzestrzennego obiektów wraz z oddalaniem się od innego obiektu. Odkrywanie trendów sprowadza się zazwyczaj do analizy regresji, gdzie odległość od danego obiektu jest zmienną niezależną, natomiast różnica wartości atrybutów do obserwacji - zmienną zależną.

Trendy dzieli się na globalne i lokalne. Pierwsze wskazują na zwiększanie (bądź zmniejszanie) wartości obserwowanych atrybutów przy rozpatrywaniu wszystkich obiektów znajdujących się na ścieżkach wychodzących z punktu początkowego. Typowym przykładem jest wzrost bezrobocia wraz z oddalaniem się od centrum miast. Trend lokalny jest reprezentowany przez pojedyncze ścieżki wykazujące inny kierunek zmian na danym atrybucie niż na sąsiednich ścieżkach.

2.2.4 Przypadki osobliwe w danych przestrzennych

Czasem w danych przestrzennych można znaleźć obiekty, których atrybuty nieprzestrzenne są niespójne z innymi obserwacjami dokonanymi w ich otoczeniu. Noszą one miano *przypadków osobliwych* [20].

Wyszukiwanie takich zjawisk jest trudne, szczególnie gdy istnieje więcej atrybutów nieprzestrzennych - odwzorowanie ich w n -wymiarowej przestrzeni może skutkować *przekleństwem wielowymiarowości* (ang. *curse of dimensionality*) [21], utrudnionym rozróżnianiem obiektów podobnych do siebie.

2.2.5 Asocjacje przestrzenne

Problem odkrywania asocjacji został pierwszy raz zdefiniowany w pracy [22] i w ogólności polega na analizie dostępnych transakcji (zbiorów obiektów, np. koszyka zakupów) oraz wykryciu występujących w nich regularności występowania elementów (typu: klient, który kupując bułki wybrał także masło).

Najczęściej reguły charakteryzuje się miarą *wsparcia* (ang. *support*) i *ufności* (ang. *confidence*). Pierwsza z nich wyraża stosunek występowania transakcji zawierającą lewą i prawą stronę reguły do ilości wszystkich transakcji. Ufność z kolei wskazuje na procentowy udział transakcji zawierających lewą i prawą stronę reguły we wszystkich transakcjach, które zawierają jego lewą stronę (jest to tzw. prawdopodobieństwo warunkowe). W celu ograniczenia ilości wykrytych wzorców wprowadzono także pojęcie *zbioru częstego* (ang. *frequent itemsets*) - zbioru elementów, dla

których wyznaczone wsparcie przekracza pewien ustalony przez użytkownika próg minimalnego wsparcia.

Z pracy [22] pochodzi także popularny algorytm wykorzystywany w wielu metodach odkrywania asocjacji i kolokacji, czyli metoda *Apriori*. Wykorzystuje on ważną cechę miary wsparcia, jaką jest *antymonotoniczność*. Wynika z niej, że zbiór może być zbiorem częstym tylko w przypadku, kiedy jego podzbiory są również zbiorami częstymi.

Na początku algorytmu generowane są jednoelementowe zbiory częste. Następnie iteracyjnie wykonywane są następujące kroki:

- tworzenie zbiorów częstych $(i + 1)$ -elementowych na podstawie zbiorów o długości i ,
- filtrowanie zbiorów kandydujących w oparciu o miarę wsparcia,
- dodanie kandydatów do zbioru wynikowego.

Generowanie kandydatów polega na łączeniu wszystkich par zbiorów częstych o identycznych elementach początkowych, a następnie usuwaniu tych, które nie są zbiorami częstymi w oparciu o własność antymonotoniczności. Algorytm kończy się, gdy zbiór kandydatów będzie pusty.

W celu dostosowania metody odkrywania asocjacji do danych przestrzennych wprowadzono pojęcie *przestrzennej reguły asocjacyjnej* [23]. Zakłada ona istnienie predykatów przestrzennych (mogących wyrażać informacje o odległości czy kierunku), które mogą występować zarówno w części warunkującej (poprzedniku), jak w warunkowanej (następniku). Następnie podczas procesu odkrywania przestrzennych reguł asocjacyjnych dane umieszczone w ciągłej przestrzeni są zamieniane na zbiór transakcji. Metoda ta jest zaliczana do modelu zorientowanego na cechę referencyjną [4]. Istnieje też inne podejście, zwane *odkrywaniem zbiorów częstych klas sąsiadów*, opisane w pracy [24].

2.2.6 Kolokacje przestrzenne

Przedstawiony w pracy [3] problem *odkrywania przestrzennych reguł kolokacyjnych* powstał w odpowiedzi na niedoskonałości asocjacji (w szczególności konieczność wyboru cechy referencyjnej) i zakłada istnienie równorzędnych cech przestrzennych.

Praca wprowadza pojęcie *wzorca kolokacji przestrzennej* (zwanego także kolokacją przestrzenną lub krócej - kolokacją), zbioru cech przestrzennych, których instancje często występują we wzajemnym sąsiedztwie [4]. Stanowi on swego rodzaju odpowiednik zbiorów częstych w asocjacjach przestrzennych. Również miara wsparcia została zastąpiona przez *miarę powszechności* (ang. prevalence), które eliminują wymaganie wiedzy o transakcjach.

Kolokacje przestrzenne są przykładem modelu zorientowanego na zdarzenie (ang. *event-centric model*).

2.3 Odkrywanie kolokacji przestrzennych

Niniejszy rozdział zawiera opisy i definicje pojęć niezbędnych do zrozumienia algorytmu zawartego w rozdziale 3.

2.3.1 Cecha przestrzenna

Kluczową kwestią w procesie odkrywania kolokacji jest odpowiednia klasyfikacja obiektów występujących w bazie danych. Każdy zbiór danych przestrzennych, oprócz informacji o lokalizacji obiektu i opisujących go danych nieprzestrzennych powinien zawierać także właściwość pozwalającą na sklasyfikowanie danego obiektu do określonej klasy. Takie przypisanie nazywane jest cechą przestrzenną (ang. spatial feature) lub rzadziej klasą obiektu (ang. object class).

Jako typowy przykład cechy przestrzennej można podać etykietę przypisaną do obiektu na mapie (np. kościół, szkoła, strzelnica). Pozwala ona na jednoznaczne określenie własności przestrzeni w punkcie, gdzie znajduje się obiekt.

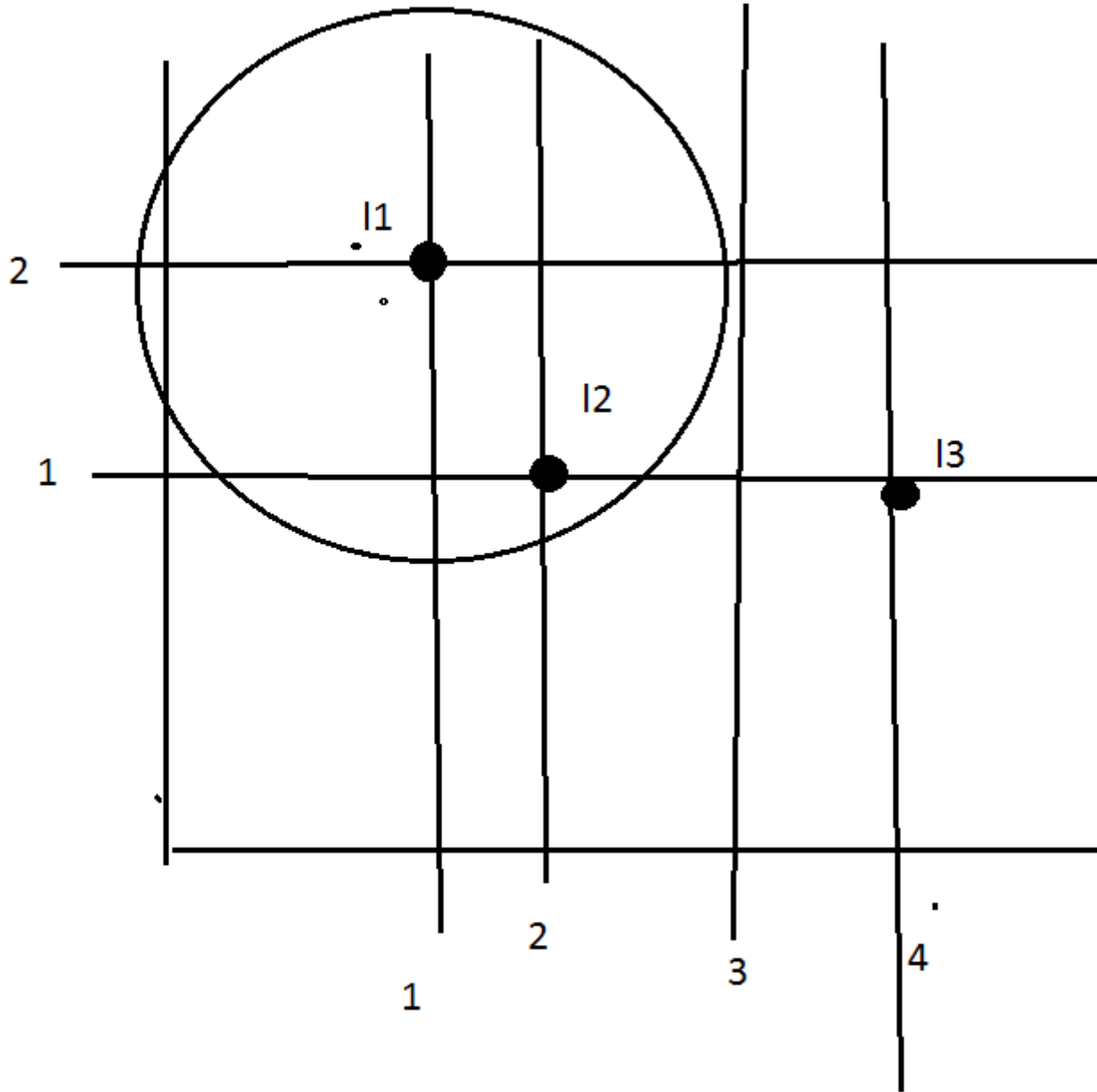
2.3.2 Podstawowe definicje

Definicja 1 (Relacja sąsiedztwa) Dany jest graf $G \in (E, V)$. Relacja sąsiedztwa R to relacja istniejąca pomiędzy wierzchołkami w grafie, taka, że jeżeli para (u, v) stanowi krawędź grafu G , to wierzchołek v jest sąsiedni do wierzchołka u .

Definicja 2 (Instancja cechy przestrzennej) Niech f będzie cechą przestrzenną. Mówimy, że obiekt x jest instancją cechy przestrzennej f , wtedy i tylko wtedy, gdy obiekt x jest typu f oraz jest opisany przez lokalizację i identyfikator.

Definicja 3 (Wzorzec i instancja kolokacji) Załóżmy F jako zbiór cech przestrzennych $F = \{f_1, f_2, \dots, f_m\}$, a $FI = FI^{f_1} \cup FI^{f_2} \cup \dots \cup FI^{f_m}$ niech będzie zbiorem ich instancji. Niech $>_F$ oznacza dowolną relację porządku zdefiniowaną dla zbioru F . Niech f_i oznacza i -tą cechę przestrzenną (ze względu na relację $>_F$), zatem $\forall i, j \in 1, \dots, m$ $f_i <_F f_j \Leftrightarrow i < j \wedge f_i, f_j \in F$. Mając daną relację sąsiedztwa R (zwrotną i przechodnią) mówimy, że wzorzec kolokacji przestrzennej (w skrócie "kolokacja") jest podzbiorem cech przestrzennych $c \subseteq F$, których instancje $I \subseteq FI$ tworzą klikę ze względu na relację R . Zbiór wszystkich instancji kolokacji przestrzennej c jest oznaczany przez CI^c . Przez długość kolokacji należy rozumieć liczbę elementów w zbiorze cech przestrzennych, który tworzy tę kolokację.

Definicja 4 (Sąsiedztwo) Mając daną zwrotną i symetryczną relację sąsiedztwa R , sąsiedztwem lokalizacji l nazywamy zbiór lokalizacji $L = \{l_1, l_2, \dots, l_n\}$, gdzie l_i jest sąsiadem l , tzn. zachodzi $R(l, l_i) \forall i \in 1, \dots, n$.



Rysunek 1: Relacja sąsiedztwa

2.3.3 Miary kolokacji

Definicja 5 (Współczynnik uczestnictwa) *Współczynnik uczestnictwa (ang. participation ratio) cechy f_i w kolokacji c jest równy procentowemu udziałowi wszystkich instancji cechy f_i w instancjach kolokacji c :*

$$pr(f_i, c) = \frac{|\pi^{f_i}(CI^c)|}{FI f_i} \quad (1)$$

gdzie $\pi^{f_i}(CI^c)$ oznacza projekcję relacyjną zbioru instancji CI^c względem cechy f_i (z usuwaniem duplikatów).

Definicja 6 (Indeks uczestnictwa) *Indeks uczestnictwa (ang. participation index) kolokacji c jest równy najmniejszemu ze współczynników uczestnictwa wyzna-*

czonych dla każdej cechy przestrzennej $f_i \in c$:

$$pi(c) = \min_{f_i \in c} pr(f_i, c) \quad (2)$$

Indeks uczestnictwa najczęściej określany jest w literaturze mianem miary powszechności lub krótko powszechności kolokacji.

Definicja 7 (Maksymalny wzorzec kolokacji przestrzennej) Niech będzie dana wartość min_prev oznaczająca pewien minimalny próg powszechności. Jeżeli $c = \{f_1, \dots, f_m\}$ jest kolokacją powszechną (tzn. $pi(c) \geq min_prev$) i nie istnieje żaden nadzbiór c taki, że powszechność dla tego nadzbioru jest równa co najmniej min_prev , kolokacja c nazywana jest kolokacją maksymalną.

2.3.4 Problem

Definicja 8 (Prawdopodobieństwo warunkowe) Prawdopodobieństwem warunkowym $cp(c_1, c_2)$ reguły kolokacyjnej $c_1 \rightarrow c_2$ nazywamy stosunek liczby instancji wzorca c_1 w sąsiedztwie instancji wzorca c_2 do liczby wszystkich instancji wzorca c_1 :

$$cp(c_1, c_2) = \frac{|\pi^{c_1}(CI^{c_1 \cup c_2})|}{CI^{c_1}} \quad (3)$$

gdzie $\pi^{c_1}(CI^{c_1 \cup c_2})$ oznacza projekcję relacyjną instancji wzorca $CI^{c_1 \cup c_2}$ względem wzorca c_1 (z usuwaniem duplikatów).

Definicja 9 (Problem odkrywania kolokacji) Problem odkrywania kolokacji przestrzennych jest zdefiniowany w następujący sposób. Mając dane:

- zbiór cech przestrzennych $F = \{f_1, f_2, \dots, f_m\}$
- zbiór obiektów $FI = FI^{f_1} \cup FI^{f_2} \cup \dots \cup FI^{f_m}$, gdzie $FI^{f_i}, (0 < i \leq m)$ jest zbiorem instancji cechy f_i , przy czym każda instancja jest opisana przez lokalizację i identyfikator,
- symetryczną i zwrotną relację sąsiedztwa R ,
- próg minimalnej powszechności min_prev oraz próg minimalnego prawdopodobieństwa warunkowego min_cond ,

znajdź wszystkie poprawne reguły kolokacyjne z powszechnością nie mniejszą niż min_prev i prawdopodobieństwem warunkowym nie mniejszym niż min_cond .

2.4 Przegląd algorytmów odkrywania wzorców kolokacji przestrzennych

W tym podrozdziale zostaną zaprezentowane skrótowo najważniejsze algorytmy odkrywania kolokacji przestrzennych.

2.4.1 Co-location Miner

Wraz z wprowadzeniem pojęcia kolokacji autorzy pracy [3] zaprezentowali także podstawowy obecnie algorytm rozwiązujący problem odkrywania wzorców kolokacji przestrzennych, zwany *Co-location Miner*. W algorytmie tym wyróżnia się następujące fazy:

- generowanie kandydatów na kolokacje przestrzenne (o długości i),
- wyznaczanie instancji dla wygenerowanych kandydatów,
- usuwanie kandydatów, których powszechność wynosi mniej niż przyjęty próg minimalnej powszechności.

Pozostali kandydaci trafiają do zbioru wynikowego, a następnie na ich podstawie są tworzone reguły kolokacyjne. Same reguły również podlegają filtracji - usuwane są te reguły, których prawdopodobieństwo warunkowe jest poniżej określonego progu.

W następnej iteracji algorytm wykonuje dokładnie te same kroki, przy czym generowani kandydaci są o długości o jeden większej. Całość kończy się, gdy nie jest możliwe już wygenerowanie nowych kandydatów.

2.4.2 Multiresolution Co-location Miner

Korzystanie z oryginalnego algorytmu *Co-location Miner* wiąże się niestety z dużymi kosztami obliczeniowymi, głównie ze względu na pracochłonny krok generowania kandydatów na kolokacje. Dlatego też niedługo później w pracy [8] autorzy zaproponowali drobną modyfikację oryginalnego algorytmu, dodając dodatkowy krok filtrowania w oparciu o przybliżoną reprezentację zbioru wejściowego.

W algorytmie *Multiresolution Co-location Miner* zbiór wejściowy zostaje podzielony na obszary (mniejsze fragmenty). Zanim rozpocznie się faza wyznaczania instancji dla wygenerowanych kandydatów, następuje szacowanie ich powszechności na podstawie sąsiadujących instancji cech przestrzennych w ramach obszarów. W przypadku zbyt niskiej wartości szacowanej powszechności kandydata, można go wykluczyć z dalszego przetwarzania i tym samym oszczędzić zasoby niezbędne na wyznaczenie jego instancji.

Dalsze kroki przebiegają identycznie jak w przypadku oryginalnego *Co-location Miner*.

2.4.3 Joinless

Celem autorów pracy [9] było stworzenie algorytmu, który omijałby konieczność tworzenia kosztownych połączeń przestrzennych na etapie wyznaczania instancji kandydatów na kolokacje (tak jak np. w przypadku rodziny algorytmów *Co-location Miner*). Nosi on nazwę algorytmu bezpołączeniowego (ang. *joinless*).

Główną różnicą w porównaniu do wcześniejszych algorytmów jest sposób generowania instancji kolokacji. Są one generowane na podstawie sąsiedztw typu gwiazda - zbiorów obiektów, w którego skład wchodzi rozpatrywany obiekt oraz jego sąsiedzi posiadający większą cechę przestrzenną. Wyznacza się je na podstawie oddzielnych algorytmów (np. *plane sweep*), lub korzysta z specjalnych struktur ułatwiających wykrywanie sąsiadów typu *R-drzewo*.

Wygenerowane instancje muszą zostać dodatkowo zweryfikowane (poprawne instancje powinny być kliką, czego nie gwarantuje sąsiedztwo typu gwiazda), a następnie - podobnie jak w algorytmie *Multiresolution Co-location Miner* - dokonuje się ich wstępnego filtrowania pod kątem progu minimalnej powszechności.

2.4.4 iCPI-tree

Drzewo iCPI (*improved Co-location Pattern Instance*, [11]) stanowi zmodyfikowaną odmianę drzewa CPI zawartego w pracy [10]. Struktura ta zawiera informacje o wszystkich zachodzących relacjach sąsiedztwa.

iCPI-tree posiada następującą strukturę:

- Poziom 1 - korzeń drzewa (oznaczony etykietą *NULL*),
- Poziom 2 - cechy elementów centralnych, czyli cechy przestrzenne obiektów centralnych *sąsiedztw typu gwiazda*;
- Poziom 3 - instancje elementów centralnych, dla których ma zostać przechowywana informacja o sąsiadach;
- Poziom 4 - cechy sąsiadów,
- Poziom 5 - instancje sąsiadów.

Sąsiedzi uporządkowani są według rosnącej cechy przestrzennej, a w przypadku instancji tej samej cechy - zgodnie z rosnącym identyfikatorem. Takie uporządkowanie nosi nazwę *uporządkowanego zbioru sąsiadów*.

Powyższa struktura drzewiasta jest wykorzystana w algorytmie w celu generowania instancji coraz dłuższych kandydatów w kolejnych iteracjach. Dokonuje się tego poprzez systematyczną ich rozbudowę o kolejne elementy. Na początku wszystkie instancje są jednoelementowe, a w kolejnych iteracjach są one rozbudowywane poprzez wyszukiwanie sąsiadów z odpowiednią cechą i weryfikowane (należy sprawdzić, czy nowo dodany obiekt do instancji jest sąsiadem każdego z obiektów należących do tej instancji).

Pozostałe kroki algorytmu (generowanie kandydatów i reguł, filtrowanie według powszechności) są podobne jak w metodach *Co-location Miner* i *joinless*.

3 Wprowadzenie do technologii CUDA

Dziedzina informatyki, jaką są obliczenia ogólnego przeznaczenia na układach GPU (ang. *general-purpose computing on graphics processing units*, w skrócie *GPG-PU*) należy do stosunkowo świeżych technik programowania - jej właściwy początek można datować na okolice 2007 roku. Jeszcze do niedawna wśród programistów panowało przekonanie, że karty graficzne powinny być odpowiedzialne tylko za rysowanie obrazu, a cała odpowiedzialność za niezbędne obliczenia powinna spadać na CPU, jako "serce" komputera. Były to czasy, kiedy pojęcie równoległego wykonywania zadań nie było szeroko rozpowszechnione - rdzenie procesora były jedynie nowinką, a cały rozwój procesorów CPU szedł w zwiększanie częstotliwości taktowania.

Szybko się okazało, że nie da się tego robić w nieskończoność. Podnoszenie częstotliwości spowodowało, że procesory zaczęły pobierać spore ilości energii, a także wydzielać ogromne ilości ciepła, którego nie dało się okiełznać bez korzystania z specjalnego chłodzenia. Rozwiązaniem okazało się skorzystanie z wielordzeniowości - wielu procesorów połączonych ze sobą specjalnymi magistralami, zamkniętymi w jednej obudowie. Wymagało to także zmiany podejścia do programowania. Równoległość daje olbrzymie możliwości, o ile potrafi się z nich skorzystać w odpowiedni sposób - bez tego programista nie uzyska zauważalnego wzrostu szybkości obliczeń, jakie dają współczesne układy wielordzeniowe.

Wielu osobom w trakcie tej "równoległej rewolucji" umknął pewien fakt - kiedy swoje triumfy świecił procesor *Pentium 4* firmy *Intel* o częstotliwości taktowania rzędu 3 GHz, na rynku istniały już rozwiązania równoległe, na których można było uzyskać znacznie lepsze wyniki. Były to oczywiście karty graficzne, które od dawna działały w sposób równoległy.

Dopóki nie powstały pierwsze specyfikacje bibliotek wspierających obliczenia na kartach graficznych, programiści próbowali wykorzystywać potencjał brzmiący w procesorach graficznych w oparciu o rendering 3D. Było to dosyć karkołomne zadanie i nie każdy algorytm można było rozwiązać w ten sposób. Karty graficzne musiały odczekać jeszcze parę lat, aby dało się wykorzystać w pełni ich możliwości obliczeniowe.

Spośród wszystkich technologii umożliwiających przeprowadzanie obliczeń na kartach graficznych zdecydowanie największą popularność osiągnęła pierwsza z nich, czyli CUDA (ang. *Compute Unified Device Architecture*). Została ona zaprezentowana po raz pierwszy w 2007 roku przez firmę NVIDIA i jest dedykowana wyłącznie produktom tej firmy (w porównaniu do konkurencyjnej technologii OpenCL).

3.1 CPU a GPU

Jak już wspomiano, procesory graficzne, mające w pierwotnym założeniu wyłącznie wspomagać generowanie obrazu, od początku były projektowane jako układy przetwarzające dane w sposób równoległy. Procesory zawarte w kartach graficznych posiadają znacznie więcej rdzeni niż CPU - mają one jednak mniejsze taktowanie, a także są gorzej wyposażone (w szczególności brakuje im rozszerzeń typowych dla CPU, typu *SSE2* czy *MMX*). W związku z tym, nie wszystkie algorytmy nadają się dobrze do implementacji w środowisku GPU. Jeżeli jednak istnieje możliwość zrównoleglenia jakiegoś problemu zgodnie z paradygmatem SIMD, to wyniki osiągnięte na GPU będą w większości wypadków lepsze niż uzyskane na procesorze CPU.

Obecne procesory CPU z kolei potrafią działać sekwencyjnie, jak i równolegle, za pośrednictwem wielu mechanizmów wspierających równoległość (np. wątki, mutexy, wsparcie dla specjalnych struktur).

Oddzielną kwestią jest przydział tranzystorów tworzących poszczególne układy do pełnienia poszczególnych funkcji. W przypadku CPU większość tranzystorów stanowi pamięć podręczną oraz układy sterowania - za wykonywanie obliczeń jest odpowiedzialna ok. 1/4 tranzystorów wbudowanych w procesor. Natomiast w przypadku GPU gros tranzystorów wykorzystanych jest do budowy jednostek arytmetyczno-logicznych oraz zmiennoprzecinkowych. Z tego powodu duża część zadań związana z przepływem danych, uruchamianiem wątków czy podziałem zadań na GPU spoczywa na programiście, co dosyć mocno komplikuje kod.

3.2 Architektura CUDA

Procesor graficzny kompatybilny z CUDA zbudowany jest z wielu multiprocesorów strumieniowych (ang. *streaming multiprocessors*, w skrócie SM). Na każdy z nich składa się określona ilość rdzeni CUDA, które mogą przetwarzać dane w sposób równoległy - w każdym z nich jest umieszczona dodatkowo jednostka arytmetyczno-logiczna (ALU, umożliwia obliczenia na liczbach całkowitych) oraz jednostka zmiennoprzecinkowa (FPU). Do tego w każdym SM znajdują się jednostki funkcji specjalnych - w nich wyznaczane są wartości funkcji matematycznych typu pierwiastkowanie czy funkcje trygonometryczne. Liczba powyższych elementów nie jest stała - zależą one od miary *potencjału obliczeniowego CUDA* (ang. *CUDA Compute Capability*), który obsługuje dany sprzęt.

Każdy SM zawiera 16 jednostek wejścia i wyjścia, mających na celu pośredniczenie w przekazywaniu danych między różnymi typami pamięci. Przez nie odbywa się transfer danych do pamięci RAM karty graficznej (zlokalizowanej poza procesorem graficznym). W multiprocesorze są dostępne następujące typy pamięci podręcznej:

- pamięć rejestrów - najszybsza pamięć, w której wątek może przechowywać swoje zmienne,
- pamięć wspólna (ang. *shared memory*) - pamięć, poprzez którą może odbywać się wymiana danych między wątkami;
- pamięć podręczna pierwszego poziomu (ang. *L1 cache*).

Oprócz tego multiprocesor wyposażony jest w układy sterujące wywołaniem instrukcji (ang. *Instruction Dispatch Unit*) odpowiadające za wykonywanie obliczeń na zgrupowanych rdzeniach.

3.3 Programowanie wielowątkowe w języku CUDA C

Środowisko programistyczne *CUDA Toolkit* umożliwia programowanie w języku *CUDA C* - stanowi on podzbiór języka C++ i jest z nim w dużym stopniu kompatybilny. Kod w języku *CUDA C* da się łączyć z standardowym kodem C++ kompilowanym pod procesory CPU - w tym przypadku fragmenty kodu wywoływane po stronie karty graficznej są kompilowane przez specjalny kompilator *NVCC* dołączany do środowiska CUDA.

Funkcje wywoływane przez hosta, a wykonywane przez procesor graficzny w sposób równoległy przez wiele wątków są nazywane funkcjami jądra (ang. *kernel*). Wszystkie wątki uruchomione poprzez wywołanie jądra wykonują identyczny kod. Oczywiście da się je odróżnić poprzez unikalny identyfikator dostępny w specjalnej zmiennej. Zgodnie z modelem przetwarzania SIMD, na jednym multiprocesorze uruchomione wątki są grupowane w tzw. *warpy* (ang. *warps*) - zbiory 32 wątków wykonujących tą samą instrukcję.

Wątki można organizować w większe grupy. Podstawową grupą wątków jest *blok*, w ramach którego można uformować wątki w jednym, dwóch, a nawet trzech wymiarach. Bloki grupowane są z kolei w siatkę obliczeniową (ang. *computation grid*), która również może składać się z trzech wymiarów. Przy wywołaniu kernela programista określa liczbę równoległych bloków oraz ilość wątków na blok, za pomocą których urządzenie będzie przetwarzać równolegle kernel. Odpowiednie ustawienie tych wartości jest kluczowe dla wydajności przetwarzania - zbyt mała/wielka ilość przyznanych bloków może spowodować odwrotny skutek do zamierzonego. Maksymalne ilości wątków na blok i bloków na siatkę określa miara *potencjału obliczeniowego CUDA*.

3.4 Model pamięci

Karty graficzne obsługujące technologię CUDA posiadają następujące typy pamięci, dostępne dla programisty:

- *rejestry* - zdecydowanie najszybsza i najwygodniejsza pamięć, umieszczona w procesorze graficznym, mała, ograniczona żywotność do czasu życia wątku;
- *pamięć lokalna* - umiejscowiona w pamięci karty graficznej, wolna, będąca alternatywą do rejestrów;
- *pamięć wspólna* - dostęp do niej możliwy jest ze wszystkich wątków pracujących w ramach jednego bloku, mała (umiejscowiona w chipie GPU), lecz o szybkim czasie dostępu;
- *pamięć globalna* - duża (rzędu paru gigabajtów), wolna, o dużym czasie dostępu, dostęp z poziomu każdego wątku oraz hosta;
- *pamięć stała* - dostępna dla wszystkich wątków w trybie tylko do odczytu, buforowana, posiada unikalną możliwość rozgłaszania danych (ang. *broadcast*);
- *pamięć tekstur* - również tylko do odczytu, przeznaczona z myślą o przechowywaniu danych w postaci macierzy.

3.5 Thrust

Biblioteka *Thrust* stanowi odpowiednik popularnej *Standard Template Library* dla języka CUDA C. Od wersji 1.4.0 jego wersja produkcyjna jest dołączana wraz z środowiskiem *CUDA Toolkit 4.0*.

Podobnie jak jego odpowiednik dla języka C++, Thrust zawiera bogatą kolekcję wbudowanych algorytmów i operacji dostosowanych do równoległego środowiska CUDA. W szczególności dostarcza równoległe operacje skanowania, sortowania czy

redukcji, zwalniając programistę od implementowania własnych, potencjalnie niewydajnych rozwiązań.

Thrust zawiera struktury znane z biblioteki STL, takie jak wektory (w wersji dla hosta oraz karty graficznej), iteratory oraz wiele innych typów generycznych ułatwiających korzystanie z możliwości dostarczanych przez środowisko CUDA C. Użycie ich jest analogiczne jak w "tradycyjnym" STL-u.

Oprócz tego, biblioteka *Thrust* oferuje następujące operacje:

- *Inclusive scan* – wyznaczenie tablicy w oparciu o tablicę o rozmiarze k , takiej, że $B[i] = \sum_{j=0}^i A[j], i \in (0, k - 1)$;
- *Exclusive scan* – wyznaczenie tablicy w oparciu o tablicę o rozmiarze k , takiej, że $B[0] = 0 \cap B[i] = \sum_{j=0}^{i-1} A[j], i \in (1, k - 1)$;
- *Compact* – usuwanie pozycji z tablicy wejściowej spełniającej podany warunek, często na podstawie innej tablicy;
- *Unique* – wyznaczenie unikalnych wartości z (posortowanej) tablicy wejściowej,
- *Sort* – sortowanie tablicy podanej na wejściu, również na podstawie klucza sortowania będącego oddzielną tablicą.

4 Algorytm *SGCT*

Niniejszy rozdział ma za zadanie przybliżenie algorytmu będącego tematem tej pracy - metody odkrywania maksymalnych kolokacji przestrzennych w oparciu o graf rzadki i skondensowane drzewo instancji (ang. *sparse-graph and condensed tree-based maximal co-location algorithm*) przedstawionej w pracy [5].

Poszczególne kroki pierwotnego algorytmu *SGCT* zostaną opisane w kolejnych podrozdziałach. Szczegóły implementacji wraz z zaproponowanymi usprawnieniami znajdują się w Rozdziale 4.

4.1 Generowanie tabeli instancji kolokacji o rozmiarze 2

Pierwszy krok algorytmu jest podobny do metody *Co-location Miner* i polega na wygenerowaniu 2-elementowych kandydatów na kolokacje.

Kolokacje o rozmiarze 2 tworzone są na podstawie wygenerowanych w oparciu o cechy przestrzenne jednoelementowych kolokacji. Nie jest do tego wykorzystywana jednak metoda *Apriori*, ponieważ udowodniono w pracy [3], że dla kandydatów dwuelementowych lepszą wydajność można uzyskać w oparciu o algorytm *spatial join*. Wykorzystany został zatem algorytm *sweeping-based spatial join* [25] z dodatkową modyfikacją, usuwającą pary instancji o tej samej cesze przestrzennej.

Przykład 1 Przykładowe zapytanie tworzące kandydatów na kolokacje o rozmiarze 2 przedstawia się następująco:

```
select  $p', p''$   
from  $\{p_1, \dots, p_{12}\}p', \{p_1, \dots, p_{12}\}p''$   
where  $p'.feature \neq p''.feature, p' \neq p'', (p', p'') \in R$ 
```

Na podstawie wygenerowanych kolokacji oraz tzw. *progu odległości* (ang. *distance threshold*) tworzona jest dwuwymiarowa tablica z haszowaniem (ang. *hash table*). Jest ona indeksowana cechami przestrzennymi. Każdy element tablicy zawiera wskaźnik do listy zawierającej instancje kandydatów o odpowiadających indeksom cechach przestrzennych. Instancja zostanie dodana do tej listy tylko wtedy, gdy odległość między instancjami nie przekracza dopuszczalnego progu odległości między nimi.

4.2 Obliczanie miary powszechności

Również krok obliczania powszechności dla kandydatów nie różni się od tego znanego z *Co-location Miner*.

Dla każdego kandydata w tabeli wyliczany jest współczynnik uczestnictwa (ang. *participation index*). Dokonuje się tego poprzez wybieranie wszystkich unikalnych instancji cech przestrzennych, która są ujęte w danej kolokacji. Następnie zgodnie z definicją miary powszechności z tabeli instancji są usuwani kandydaci, dla których obliczona miara powszechności jest mniejsza niż zadany próg minimalnej powszechności *min_prev* [3].

4.3 Generowanie kandydatów na kolokacje maksymalne

Krok ten wprowadza nową strukturę, zwaną *grafem kolokacji o rozmiarze 2* (ang. *size-2 co-location graph*). Jego definicja brzmi następująco:

Definicja 10 (Graf kolokacji o rozmiarze 2) Jeżeli przyjąć relacje sąsiedztwa między kolokacjami o rozmiarze 2 jako krawędzie $E = \{e_1, \dots, e_u\}$, a cechy przestrzenne występujące w kolokacjach jako wierzchołki $V = \{v_1, \dots, v_\lambda\}$, gdzie u i λ są odpowiednio liczbą krawędzi i liczbą wierzchołków, to graf kolokacji o rozmiarze 2 można zamodelować jako graf nieskierowany $G = (V, E)$, przechowywany w listowej strukturze danych uporządkowanej rosnąco. Zbiór N jest zbiorem sąsiedztw wierzchołka i definiuje się go następująco:

$$N(v_i) = \{W | v_i, w \in E\} \quad (4)$$

Zadaniem tego kroku jest wyszukanie w takim grafie maksymalnych klik, określanych jako *kandydaci na maksymalne kolokacje* (ang. *candidate maximal co-location*).

Definicja 11 (Kandydat na maksymalną kolokację) Kandydat na maksymalną kolokację C_m składa się z uporządkowanych cech przestrzennych o następujących właściwościach: każda para cech w C_m jest ze sobą połączona krawędzią, a żadne dodatkowe cechy nie mogą być dodane do C_m bez zachowania ich kompletnego połączenia.

Autorzy pracy [5] udowodnili, że graf kolokacji o rozmiarze 2 można traktować jako graf rzadki. Umożliwia to efektywne korzystanie z algorytmu Brona-Kerboscha [27] do wyszukiwania maksymalnych klik w grafie nieskierowanym. Wprowadzone zostały do niego pewne modyfikacje uwzględniające rozproszenie grafu oraz wybieranie *pivotu* w celu usprawnienia wyszukiwania kandydatów na kolokacje (patrz Algorytm 1).

Wejście: $G = (E, V)$
Wyjście: CP_m

```

1  $CP_m \leftarrow \emptyset; X \leftarrow \emptyset; P \leftarrow \emptyset;$ 
2 foreach  $v_i^*$  in lista  $v_1^*, v_2^*, \dots, v_\lambda^*$  uporządkowanych wg. degeneracji do
3    $P \leftarrow N(V_i^*) \cup \{v_{i+1}^*, \dots, v_\lambda^*\};$ 
4    $X \leftarrow N(V_i^*) \cup \{v_1^*, \dots, v_{i-1}^*\};$ 
5    $BK\_Pivot(P, \{v_i^*\}, X);$ 
6 end
7 Procedure  $BK\_Pivot(M, K, T)$ 
8   if  $M \cup T = \emptyset$  then  $\{CP_m \leftarrow CP_m \cup K\};$ 
9   wybór pivotu  $u \in M \cup T$ ; % do maksymalizacji  $|M \cap N(u)|$ ;
10  foreach  $v_i \in M \setminus N(u)$  do
11     $BK\_Pivot(M \cap N(V_i), K \cup \{v_i\}, T \cap N(V_i));$ 
12     $M \leftarrow M \setminus \{v_i\};$ 
13     $T \leftarrow T \cup \{v_i\};$ 
14  end

```

Algorytm 1: Generowanie maksymalnych kandydatów na kolokacje

Pierwsza z modyfikacji oryginalnego algorytmu dodaje mechanizm *pivoting selection* opisany pierwotnie w pracy [28] (linia 9). Jak wykazano w pracy [29] wybranie wierzchołka zmniejszającego liczbę rekurencyjnych wywołań algorytmu znacząco

zmniejsza ogólny czas wykonania. Wybiera on wierzchołek będący osią podziału zbioru (tzw. *pivot*) w oparciu o rozmiar unii sąsiadów tego wierzchołka i wierzchołków kandydatów. Każda maksymalna klika musi zawierać albo wierzchołek u , albo niesąsiadujące z nim wierzchołki - jeżeli nie zawiera, zostanie on dodany (linie 11-13). W związku z tym, tylko wierzchołek u i jego nie-sąsiedzi muszą być przetestowani (linia 10).

Druga modyfikacja opiera się o pojęcie rozproszenia grafu. Opisuje się je miarą *degeneracji grafu* [31]:

Definicja 12 (Degeneracja grafu) *Degeneracja grafu G jest najmniejszą wartością k , taką, że każdy niepusty podgraf G zawiera wierzchołki o stopniu co najwyżej k . Oznacza to, że wielkość maksymalnej kliki nie może przekroczyć $k + 1$.*

Definicja 13 (Uporządkowanie wg. miary degeneracji) *Uporządkowanie według miary degeneracji wierzchołków grafu G to takie uporządkowanie, które minimalizuje stopień degeneracji grafu. Taka uporządkowanie gwarantuje m.in. optymalną kolejność kolorowania w problemie kolorowania wierzchołków.*

Wierzchołki w zewnętrznej rekurencji są uporządkowane według stopnia degeneracji (linia 2). W ciele rekurencji (linie 3-5) liczba wierzchołków czekających na weryfikację nie przekroczy k . Tym sposobem ograniczono liczbę zewnętrznych rekurencji. Dla grafów o małym stopniu degeneracji obserwuje się duży wzrost wydajności [26].

Autorzy pracy [30] udowodnili, że zbiór wynikowy kandydatów na kolokacje zawsze będzie zawierał wszystkie maksymalne kliki spełniające próg powszechności. Jest to jeden z warunków poprawności algorytmu generowania maksymalnych kandydatów na kolokacje.

4.4 Proces odcinania

W ostatnim kroku w oparciu o tzw. *prunning framework* [30] następuje otrzymywanie końcowych maksymalnych kolokacji spośród kandydatów wyznaczonym w poprzednim procesie. Na początku dla każdego z nich uruchamiany jest algorytm wyszukiwania klik instancji. Do jego zrozumienia niezbędne jest wprowadzenie poniższych definicji.

Definicja 14 (Uporządkowana klika instancji) *Dana jest tabela instancji kolokacji o rozmiarze 2 ($InsTable_2$) oraz kandydat na maksymalną kolokację C_m . Jego uporządkowana klika instancji $InsC_m$ jest grupą instancji przestrzennych spełniających następujące warunki:*

- rozmiar $InsC_m$ jest równy rozmiarowi C_m , a cechy w odpowiadających sobie instancjach w $InsC_m$ i C_m są takie same;
- instancje każdej pary instancji w $InsC_m$ sąsiadują ze sobą w przestrzeni i można je znaleźć w tabeli $InsTable_2$.

Definicja 15 (Skondensowane drzewo instancji) *Dany jest kandydat na maksymalną kolokację C_m . Skondensowane drzewo instancji $CInsTree$ jest konstrukcją kompresującą wszystkie uporządkowane kliki instancji C_m .*

Algorytm 2 ma charakter iteracyjny, w którym zmienna i jest zmienną sterującą pętli. Na początku zostaje zainicjalizowane drzewo $CInsTree$ i tworzony jest jego korzeń. Następnie uruchamiany jest proces konstrukcji drzewa, podzielony na dwa etapy. Pierwszy etap wykonywany jest tylko w pierwszej iteracji algorytmu - kolejne iteracje będą wykonywać krok drugi.

W pierwszym kroku (linie 3-11) dla każdej pary instancji pierwszych dwóch cech przestrzennych C_m następuje sprawdzenie, czy pierwszy element aktualnie przetwarzanej pary instancji istnieje na danym poziomie drzewa $CInsTree$ (oznaczonego jako $CInsTree_1$). Jeżeli tak, następuje dodanie drugiego elementu jako węzeł podrzędny odpowiadającego mu węzła na poziomie pierwszym. W przeciwnym wypadku, na podstawie obecnej pary instancji tworzone jest poddrzewo, które następnie jest dołączane do korzenia $CInsTree$.

Drugi etap rozpoczyna się konstrukcją listy El zawierającej instancje cechy $C_m(i + 1)$. Dokonuje się tego dla każdego węzła instancji na poziomie i -tym drzewa $CInsTree$ poprzez skanowanie $InsTable_2(C_m(i), C_m(i + 1))$, gdzie $c_m(i)$ jest i -tą cechą C_m . Następnie dla każdego elementu tej listy następuje sprawdzenie, czy zarówno para instancji składająca się z tego elementu, jak i każdy przodek aktualnego węzła instancji $CInsTree$ znajduje się w tablicy kolokacji o długości 2 - $InsTable_2$. Jeżeli tak, dodaje się ten element jako węzeł podrzędny aktualnego węzła instancji.

Proces działa do czasu, gdy nie będzie żadnego węzła na i -tym poziomie drzewa lub dopóki i nie będzie mniejsze niż $len(C_m) - 1$.

Po zakończeniu algorytmu pozostaje jeszcze wyliczenie indeksów powszechności dla odnalezionych klik instancji. Podobnie jak wcześniej, w przypadku kiedy miara powszechności nie jest mniejsza niż przyjęty na początku próg powszechności, kandydata można przyjąć jako właściwy wzorec kolokacji [3].

5 Implementacja CPU

W tym rozdziale zostaną opisane implementacje rozwiązania problemu [5] wykorzystujące wyłącznie potencjał procesorów CPU (ang. *Central Procession Unit*), bez udziału karty graficznej w obliczeniach. W szczególności zostaną zaprezentowane techniki optymalizacyjne dedykowane dla poszczególnych podejść do rozwiązania problemu.

Do celów porównawczych zostały zrealizowane dwa rozwiązania: sekwencyjne (bez żadnego wsparcia dla równoległości) oraz równoległe (z wykorzystaniem biblioteki *Parallel Patterns Library* firmy *Microsoft*).

5.1 Wersja sekwencyjna

5.1.1 Generowanie instancji w oparciu o próg odległości

Podobnie jak w pierwotnej wersji algorytmu, generowanie kandydatów na kolokacje o rozmiarze 2 następuje na podstawie *łączenia przestrzennego w oparciu o zamykanie* (ang. *sweeping-based spatial join*). Algorytm zamykania (ang. *plane sweep algorithm*) użyty w procesie łączenia został jednak nieznacznie zmodyfikowany.

Na początku wektor będący zbiorem wejściowym jest traktowany algorytmem *sortowania szybkiego* (ang. *quick sort*) dostępnym w bibliotece standardowej C++. Jako kryterium sortowania zostaje wybrana relacja między arbitralnie wybranym wymiarem przestrzeni.

Następnie następuje wyznaczenie progu odległości, na podstawie którego będą dobierani kandydaci na kolokacje. Warto zwrócić w tym miejscu uwagę na fakt, że algorytmowi nie jest potrzebna wiedza o rzeczywistej odległości między dwoma punktami w przestrzeni, a jedynie relacja między tą odległością a zadany progami odległości. Daje to możliwość uniknięcia niepotrzebnych obliczeń (pierwiastkowania) przy wyznaczaniu odległości między punktami. Dlatego też, zamiast bezpośrednio działać w oparciu o zadany próg odległości, wyliczany jest *efektywny próg odległości*, będącym progiem odległości podniesionym do potęgi drugiej.

Pozostało już jedynie filtrowanie obiektów w oparciu o wyliczony efektywny próg odległości. Dla każdego obiektu ze zbioru wejściowego następuje przeszukiwanie we wszystkich kierunkach obiektów sąsiadujących z nim, a następnie sprawdzana jest odległość między nimi. Gdy jest ona mniejsza niż ustalony wcześniej próg odległości obiekt umieszczany jest w specjalnej strukturze *insTable*. Jednocześnie tworzony jest wektor *typeIncidenceCounter*, w którym zliczane są wystąpienia kolejnych cech instancji przestrzennych (informacja o tym przyda się przy wyznaczaniu miary powszechności). Dodatkowo, pętla wewnętrzna jest przerywana, jeżeli wartość bezwzględna różnicy posortowanych współrzędnych poszczególnych obiektów będzie większa niż zadany próg odległości. Ma to na celu ograniczenie zbędnych porównań w przypadku, kiedy istnieje pewność, że żadne kolejne punkty nie spełnią progu odległości - a zatem nie wejdą do zbioru *insTable*.

Struktura *insTable* pełni podobną rolę, jak w oryginalnym algorytmie *SGCT*, nie jest ona jednak dwuwymiarową tablicą z haszowaniem. Zamiast tego zastosowano trójwymiarową mapę wskaźników na wektor liczb, w której wymiarami są kolejno: typ elementu *A*, typ elementu *B* oraz numer instancji przestrzennej *A*. W wektorze przechowywane są kolejne numery instancji przestrzennej *B*. W celu efektywnego odczytywania sąsiadów o danej cesze konkretnej instancji, dane w wektorze

są umieszczane w taki sposób, że zawsze spełniają relację *numer cechy* $A > \text{numer cechy } B$. Dodatkowo, zamiast standardowej mapy (*std::map*) została wykorzystana mapa nieuporządkowana - gwarantuje ona większą wydajność przeglądania (kosztem większego zużycia pamięci).

5.1.2 Filtrowanie sąsiadów w oparciu o próg minimalnej powszechności

Krok ten stanowi zmodyfikowaną wersję algorytmu obliczania miar powszechności znanego z algorytmu *Co-location Miner* (patrz Rozdział 2.4.1).

Na początku wykonywana jest funkcja *countUniqueInstances*. Ma ona na celu zliczenie wszystkich wystąpień par instancji bez duplikatów. Podobnie jak w kroku pierwszym zostały tu użyte mapy nieuporządkowane (*std::unordered_map*) w celu przyspieszenia przetwarzania. Do budowy mapy została użyta własna funkcja mieszająca powstała na bazie *hash_combine* z biblioteki *boost*. Ogranicza ona występowanie kolizji w przypadku wstawiania nowych elementów do tablicy z haszowaniem.

Następnie w oparciu o wektor wynikowy tej funkcji oraz utworzony wcześniej wektor *typeIncidenceCounter* (patrz Rozdział 5.1.1) dla każdego kandydata w tabeli *insTable* wyliczany jest współczynnik uczestnictwa. W przypadku, gdy obliczona miara powszechności jest mniejsza od zadanego progu minimalnej powszechności, kandydat jest usuwany z struktury *insTable*, a użyta przez niego pamięć zostaje zwolniona na poczet dalszych obliczeń.

5.1.3 Generowanie kandydatów na kolokacje maksymalne

Tak jak w przypadku oryginalnego algorytmu, kandydaci na kolokacje maksymalne są generowani w oparciu o *graf kolokacji o rozmiarze 2* (patrz Definicja 10). Tworzony jest na bazie struktury *insTable* za pomocą funkcji *createSize2ColocationsGraph*. Krawędź między elementami pary instancji jest tworzona tylko i wyłącznie wtedy, kiedy numer instancji przestrzennej A jest większy od zera.

Po wygenerowaniu grafu liczona jest jego miara degeneracji (patrz Definicja 12). W tym celu został wykorzystany Algorytm 3 zaprezentowany przez Matulę i Becka w pracy [31]. Działa on w czasie wielomianowym, co pozwala na odciążenie CPU. Funkcja zwraca zarówno miarę degeneracji, jak uporządkowany według niej wektor wierzchołków występujących w grafie.

Dalsze kroki algorytmu wykonywane są dla poszczególnych wierzchołków uporządkowanych według miary degeneracji. Podobnie jak w oryginalnym algorytmie z pracy [28] tworzone są grupy wierzchołków o niższych i wyższych indeksach, a następnie uruchamiana jest funkcja *BK_Pivot*, w której następuje wyszukiwanie maksymalnych klik w oparciu o algorytm Brona-Kerboscha [27].

Aby ograniczyć czas wyszukiwania maksymalnych klik, zostały zastosowane tzw. *wektory sortowane* zamiast zazwyczaj używanych zbiorów (*std::set*) z biblioteki C++. Posiadają one większą wydajność, a do tego pozwalają na skorzystanie z funkcjonalności zarezerwowanej wyłącznie dla zbiorów.

5.1.4 Generowanie skondensowanych drzew instancji oraz filtrowanie kandydatów na podstawie progu powszechności

Na początku wszystkie rozważane maksymalne kliki z poprzedniego kroku są umieszczane w specjalnej strukturze *cliquesToProcess* będącej trójwymiarowym

wektorem, gdzie pierwsza współrzędna stanowi rozmiar kolokacji, a druga o pozycji kliku w kolejce w poszczególnych wierszach. Trzecią współrzędną są oczywiście kolejne elementy danej kliku.

Następnie iteracyjnie przetwarzana jest każda klik, poczynając od największej. Zrezygnowano w tym miejscu z podejścia rekurencyjnego, gdyż prowadziłoby to do niepotrzebnego utrzymywania ogromnego stosu wywołań (ang. *call stack*). W każdej iteracji sprawdzana jest miara powszechności dla kliku instancji - odpowiedzialna jest za to funkcja *isCliquePrevalent*.

Wyliczone kliku są umieszczane w obiektach *CliqueContainer* i *LapsedCliqueContainer*. Pełnią one rolę pamięci podręcznej dla powyższych algorytmów. Dzięki temu nie ma potrzeby ponownego przeliczania tych samych klik instancji, co prowadzi do kilkunastokrotnego wzrostu wydajności obliczeń.

Dla klik o długości większej niż 2 tworzone jest skondensowane drzewo instancji (patrz Definicja 15). Implementacja konstrukcji takiego drzewa zasadniczo nie różni się od pierwotnego algorytmu zaprezentowanego w pracy [5], zastosowano w nim jednak pewne metody optymalizacyjne - wskaźniki do rodziców w celu ograniczenia przeszukiwań drzewa w dół, użycie wektora zawierającego wskaźniki do liści znajdujących się na ostatnim poziomie drzewa, wskaźniki typu *unique_ptr* (więcej szczegółów w Dodatku A). Następnie zliczane są wystąpienia instancji w klikach za pomocą odwróconej pętli i na ich podstawie wyliczana jest miara powszechności.

W przypadku, kiedy miara nie jest mniejsza niż przyjęty na początku próg powszechności, kandydat jest dodawany do zbioru rozwiązań. W przeciwnym wypadku do struktury *cliquesToProcess* dodawane są podkliku, które również będą rozważane jako potencjalni kandydaci na kolokacje.

5.2 Wersja wielowątkowa

Jak już wcześniej wspomniano, rozwiązanie równoległe zostało oparte o multiplatformową, wysokopoziomową bibliotekę *Parallel Patterns Library* firmy *Microsoft*. Została ona pierwszy raz zaprezentowana szerszej publiczności wraz z wydaniem środowiska *Visual Studio 2010* i docelowo ma być konkurencją dla popularnej biblioteki *OpenMP*.

Cechą charakterystyczną tej biblioteki jest podobieństwo składni do tej z biblioteki standardowej C++ (ang. *C++ Standard Library*). Wykorzystuje też wszystkie właściwości języka C++, jakie przynosi standard C++11 i późniejsze (w tym funkcje *lambda*).

Biblioteka *PPL* wprowadza zbiór obiektów zwanych *kontenerami o dostępie równoległym* (ang. *concurrent containers*). Są to odpowiedniki kontenerów z biblioteki standardowej C++, które cechują się równoległymi wersjami funkcji operujących na kontenerach (np. operacji wstawiania czy usuwania). W zdecydowanej większości przypadków ich użycie nie różni się od wersji sekwencyjnych dostępnych w bibliotece STD - wymagane jest jedynie umieszczenie ich w odpowiednim bloku, np. *parallel_for*.

W przypadku, kiedy istnieje potrzeba dostarczenia kopii kontenera dla każdego z wątków (np. żeby nie blokować dostępu innym wątkom do współdzielonego obiektu), istnieje także klasa kontenerów typu *combinable*. Kiedy równoległe przetwarzanie zostanie zakończone, prywatne kopie wygenerowane dla każdego z wątków są wtedy przezroczysto łączone w całość. Oczywiście *PPL* zawiera także tradycyjne metody

zapewnienia równoległości w programie, takie jak zadania (ang. *tasks*) oraz klasyczne mutexy.

Zasadniczą różnicą między *PPL* a *OpenMP* jest zastosowanie dynamicznego planisty (ang. *dynamic scheduler*), co pozwala na lepszą optymalizację równoległego przetwarzania w zależności od aktualnie dostępnych zasobów w systemie. Na dynamicznym planiście zyskują szczególnie problemy o charakterze rekurencyjnym (np. algorytmy sortujące czy szeroko wykorzystywane w niniejszej pracy przeszukiwanie danych) [32]. Do tego technologia *OpenMP* nie zawiera żadnego mechanizmu anulowania, często wymaganego w algorytmach o charakterze równoległym [33].

Powyższe wady środowiska *OpenMP* były powodem, dla których ostatecznie - pomimo ubogiej dokumentacji oraz niskiej popularności - została wybrana biblioteka *Parallel Patterns Library* jako najbardziej optymalna dla algorytmu *SGCT* będącego tematem tej pracy.

5.2.1 Generowanie sąsiadów na podstawie progu odległości

Różnice w równoległej implementacji generowania sąsiadów w stosunku do wersji sekwencyjnej (patrz Rozdział 5.1.1) wynikają w dużej mierze z zastosowania alternatywnych struktur dostępnych w ramach biblioteki *Parallel Pattern Library*.

Dla przykładu, zbiór wejściowy jest sortowany za pomocą wbudowanego w bibliotekę PPL algorytmu *równoległego sortowania buforowanego* (ang. *parallel buffered sort*). Jest to odmiana standardowego sortowania szybkiego wykorzystujące równoległe scalanie (ang. *parallel merge*) w celu zastąpienia pierwotnego kroku dzielenia w algorytmie sortowania szybkiego, którego nie da się zrównoleglić [35] - i podobnie jak *quick sort*, po posortowaniu elementy o tej samej wartości mogą nie występować w tej samej kolejności (przykład tzw. *sortowania niestabilnego*). Dla wielu procesorów pozwala osiągnąć większą wydajność niż standardowy *quick sort* (jego złożoność wynosi $O((n/p) * \log n)$ dla p procesorów). Podejście to wymaga niestety dodatkowej alokacji pamięci o rozmiarze $O(n)$.

W procesie filtrowania obiektów na podstawie wyliczonego progu odległości, podobnie jak w wersji sekwencyjnej, tworzona jest tablica asocjacyjna *insTable* - jednak w podejściu równoległym jest ona typu *combinable*. Dzięki temu każdy wątek posiada swoją prywatną kopię tablicy *insTable*, w której umieszcza przetwarzane przez siebie pary instancji, które nie przekraczają wyliczonego progu efektywnej odległości. Po zakończeniu przetwarzania przez wszystkie wątki lokalne kopie struktury *insTable* są łączone w całość poprzez zastosowanie operacji redukcji oraz sumowania wektorów parami, co eliminuje problem współbieżnego dostępu do struktury w środowisku równoległym. Podobnie sytuacja wygląda w przypadku struktury *typeIncidenceCounter*.

W algorytmie filtrowania został wykorzystany także dynamiczny planista (*auto partitioner*). W przypadku przerwania pętli wewnętrznej algorytmu, planista dynamicznie przydziela bezczynnym wątkom kolejne dane wejściowe niezbędne do dalszego przetwarzania. Po wyjściu z pętli następuje synchronizacja wątków, a tablica *insTable* ulega scaleniu.

Poza powyższymi usprawnieniami, ogólny mechanizm oraz struktury zastosowane w kroku pierwszym pozostały właściwie niezmienione w stosunku do wersji sekwencyjnej.

5.2.2 Filtrowanie sąsiadów na podstawie progu minimalnej powszechności

Jedyną różnicą na tym etapie algorytmu w stosunku do odmiany sekwencyjnej jest dodatkowy krok określenia zakresu pracy dla każdego z wątków biorących udział w liczeniu wystąpień par instancji. Odpowiedzialna za to jest funkcja *getWorkloadForInsTable*.

W porównaniu do poprzedniego etapu, wykorzystywany jest statyczny planista (ang. *static-partitioner*) będący odpowiednikiem funkcji *omp parallel for* z biblioteki *OpenMP*. Dzięki temu każdemu wątkowi przydzielana jest stała ilość par instancji do przetworzenia. Liczba ta jest wyznaczana na podstawie rozmiaru tablicy *insTable*. Daje to pewną oszczędność czasu, gdyż wątki nie tracą czasu na dynamiczne przydzielanie pracy do wykonania.

5.2.3 Generowanie kandydatów na kolokacje maksymalne

Zasadniczą zmianą na tym etapie algorytmu jest wprowadzenie równoległej wersji algorytmu 1 liczenia maksymalnych klik metodą Brona-Kerboscha [27]. Korzysta on ponownie z dynamicznego planisty w celu dynamicznego rozdzielenia pracy dla poszczególnych wątków, ponieważ nieznany jest czas wykonywania algorytmu.

Każdy z wątków uruchamia dla swojej puli wierzchołków standardową wersję algorytmu Brona-Kerboscha wykorzystującą tzw. *pivot*. Podejście to może powodować występowanie duplikatów - dlatego też po zakończeniu działania algorytmów zbiór wynikowy rzutowany jest na kontener *std::set*, likwidując tym samym potencjalnie występujące w zbiorze duplikaty.

Tradycyjnie na końcu lokalne listy maksymalnych klik są scalane w jedną strukturę wynikową za pomocą instrukcji *combinable::combine_each*.

5.2.4 Generowanie skondensowanych drzew instancji oraz filtrowanie kandydatów na podstawie progu powszechności

W porównaniu do wersji sekwencyjnej, konstrukcja skondensowanego drzewa instancji oraz liczenie miary powszechności jest dokonywany w sposób równoległy. Wymagało to zaprojektowania równoległych odpowiedników kontenerów znanych z odmiany sekwencyjnej, czyli *ParallelCliqueContainer* i *ParallelLapsedCliqueContainer* zapewniających bezpieczeństwo przetwarzania równoległego. Niezbędne było także zastosowanie (podobnie jak w kroku pierwszym) dynamicznego planisty (ang. *auto scheduler*) przydzielającego dynamicznie kolejne kliki do przetworzenia do poszczególnych wątków. Niemożliwe było zastosowanie statycznego planisty z uwagi na fakt, że na tym etapie nie można przewidzieć dokładnego czasu konstrukcji skondensowanego drzewa instancji.

Oddzielną kwestią jest struktura tablicy *insTable*. Podobnie jak w wersji sekwencyjnej zawiera trzy wymiary. Wykorzystano wektor pochodzący z standardowej biblioteki C++, którego elementy są wskaźnikami na wektory równoległe (ang. *concurrent vector*) pochodzące z biblioteki *PPL* - same z kolei zawierają wskaźniki na wektory klasyczne. To nieco pokrętne rozwiązanie zapewnia bezproblemowe przetwarzanie równoległe, a także - ze względu na minimalizację fragmentacji zaalokowanej pamięci - przyspiesza działanie algorytmu.

Podobnie jak w innych krokach, również tutaj wykorzystywany jest wektor typu *combinable*. Umieszczany jest w niej ostateczny wynik przetwarzania będący scaleniem lokalnych kopii tablic posiadanych przez każdy wątek, biorący udział w wyznaczaniu ostatecznych kandydatów na kololacje.

6 Implementacja GPU

W tym rozdziale zostaną opisane poszczególne kroki implementacji algorytmu wyszukiwania maksymalnych prewalentnych kolokacji przy wykorzystaniu procesorów CUDA.

Przedstawiona tu implementacja jest silnie zależna od założonej wersji *compute capability* równej 3.0. W wersji tej nie jest dostępna pamięć współdzielona (*Unified Memory*[34]) oraz *Dynamic Parallelism*[36].

6.0.1 Przygotowanie danych w pamięci GPU

Z powodu rozdzielnego modelu pamięci (co zmienia się wraz z wprowadzeniem *unified memory*) dane które zostały wczytane z zewnętrznego źródła (zbiór instancji zawierających typ, identyfikator, położenie w przestrzeni dwuwymiarowej) (//TODO napisać gdzieś o możliwościach rozszerzenia algorytmu) muszą zostać przekazane do pamięci dostępnej dla procesorów CUDA. Operacja przenoszenia danych pomiędzy przestrzeniami pamięci jest dość kosztowna i silnie zależy od wybranych rozwiązań sprzętowych.

6.0.2 Generowanie tabeli instancji o rozmiarze 2

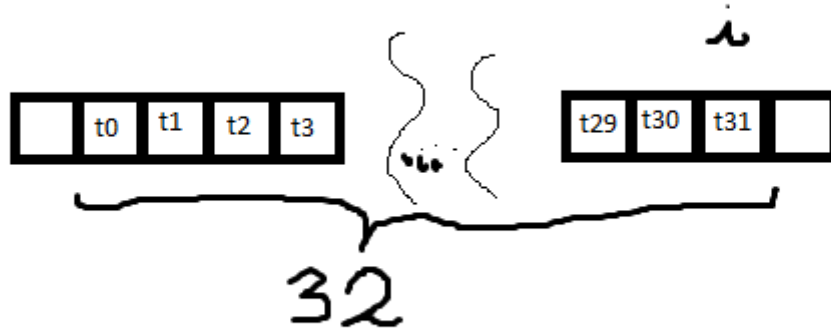
Na wejściu tego kroku oczekujemy wczytanej tablicy danych o rozmiarze n_{dt} zawierającej instancje $\{a_0, a_1, \dots, a_{n_{dt}-1}\}$ oraz parametru *distanceThreshold* zawierającego wartość maksymalnej odległości pomiędzy instancjami dla których ma zostać stwierdzona relacja sąsiedztwa. Na wyjściu otrzymujemy dwie tablice przechowujące znalezione pary instancji $Pairs_A$ oraz $Pairs_B$

Algorytm wybrany do rozwiązywania tego zadania to algorytm *Planesweep* przystosowany do uruchamiania w architekturze masowo równoległej. Sam algorytm podzielić możemy na dwie części, wyznaczanie ilości sąsiadów dla każdej instancji (kernel *countNeighbours*), a następnie alokacja i wpisanie znalezionych sąsiadów (kernel *findNeighbours*) do tabel *InstanceFirst* oraz *InstanceSecond* które służą do przechowywania par instancji będących w relacji sąsiedztwa, gdzie pierwsza tabela zawiera elementy będące pierwsze w parze, a druga drugie. Wartości należące do jednej pary znajdują się pod tymi samymi indeksami. Kolejność w obrębie pary wyznaczana jest przez następujący operator.

Przed wykonaniem samego algorytmu sortujemy zbiór danych wejściowych względem osi odciętych w kolejności rosnącej. Dzięki temu ograniczymy obszar przeszukiwań sąsiadów instancji. Użyta została do tego funkcja sortująca zaimplementowana w bibliotece *Thrust*.

Po zakończeniu sortowania przechodzimy do pierwszego kroku algorytmu *Planesweep*. Tworzymy konfigurację uruchomieniową dla kernel'a *countNeighbours* z ilością wątków równą $n_{dt} * 32$, co daje nam jeden *warp* na instancje. Dodatkowo musimy zaalokować także tablicę T_{counts} wielkości n_{dt} w którą na pozycji i zostanie wpisana ilość sąsiadów instancji a_i . Po przygotowaniu danych uruchamiany jest kernel. Każdy *warp* wykonuje maksymalnie $\lceil i/32 \rceil - 1$ iteracji, w każdej j -tej iteracji sprawdzana jest grupa 32 instancji z zakresu $K = \{i - j * 32; i - (j + 1) * 32\}$. Sprawdzenie polega na obliczeniu odległości euklidesowej pomiędzy instancjami a_i oraz a_k i jeśli jest ona nie większa od założonej maksymalnej odległości pomiędzy instancjami *distanceThreshold* stwierdzenie pomiędzy nimi relacji sąsiedztwa. Każdy

wątek *warp*'u sumuje w tablicy pamięci współdzielonej ilość znalezionych we wszystkich iteracjach sąsiadów, suma ta jest przechowywana przez każdy wątek w osobnej komórce (pozwała to uniknąć stosowania kosztownego mechanizmu synchronizacji zapisu pomiędzy wątkami *warp*'a). Po każdej iteracji sprawdzane jest czy którykolwiek z wątków *warp*'a napotkał instancje która odległa jest na samej osi odciętych o więcej niż odległość *distanceThreshold*, jeśli tak to można przerwać iterowanie po instancjach, ponieważ korzystając z faktu iż zbiór danych został posortowany po tej osi, nie znajdziemy dalej już żadnych instancji leżących nie dalej niż założony *distanceThreshold*. Po zakończeniu iterowania każdy *warp* wykonuje szybką redukcję części tablicy współdzielonej należącej do jego wątków, a wynik wpisuje do tablicy T_{counts} na pozycji i .



Rysunek 2: Poglądowy rysunek przedstawiający iterowanie warpa po instancjach

Po wykonaniu kernel'a *countNeighbours* redukujemy tablicę T_{counts} otrzymując całkowitą ilość znalezionych relacji sąsiedztwa wśród wszystkich instancji, oznaczmy tę wartość jako nC . Alokujemy tablice $Pairs_A$, $Pairs_B$ każda wielkości nC . Dodatkowo alokujemy tablice $Begins$ wielkości $textit{tn}_{dt}$ w której zapisujemy wynik operacji sumy eksklusywnej tablicy T_{counts} , która posłuży do wyznaczania pozycji zapisu relacji w tablicach wynikowych. Konfiguracja uruchomieniowa kernel'a *findNeighbours* jest identyczna z konfiguracją użytą dla *countNeighbours*. Sama funkcja różni się zresztą od poprzedniczki tylko i wyłącznie w miejscu stwierdzenia relacji sąsiedztwa. Mianowicie kernel *findNeighbours* zamiast zliczać relacje sąsiedztwa wpisuje do tablic $Pairs_A$, $Pairs_B$ instancje pomiędzy którymi ją stwierdzono. Przy pomocy wyżej opisanego operatora mniejszości dla obiektów typu *FeatureInstance* (nie zapisujemy nigdzie informacji o współrzędnych instancji, ponieważ nie będą na dalej nigdzie potrzebne), pozycja zapisu w tablicach wynikowych wyznaczana jest na podstawie wartości w tablicy *Begins* na pozycji i , ilości znalezionych przez warp w poprzednich iteracjach relacji sąsiedztwa oraz ilości relacji sąsiedztwa znalezionych w obecnej iteracji przez wątki warpa o mniejszym *id*. Warunki przerwania iterowania są takie same jak w kernel'u *countNeighbours*.

Ostatnią czynnością jest posortowanie par instancji przy użyciu następującego operatora mniejszości


```

1 operator < (FeatureInstance lhs, FeatureInstance rhs)
2 {
3   if a[0].fields.featureId < b[0].fields.featureId then
4     | return True ;
5   else if a[0].fields.featureId == b[0].fields.featureId then
6     | if a[1].fields.featureId < b[1].fields.featureId then
7       | return True ;
8     | else if a[1].fields.featureId == b[1].fields.featureId then
9       | if a[0].fields.instanceId < b[0].fields.instanceId then
10        | return True ;
11        | else if a[0].fields.instanceId == b[0].fields.instanceId then
12        | return a[1].fields.instanceId == b[1].fields.instanceId ;
13 return False ;
14 }

```

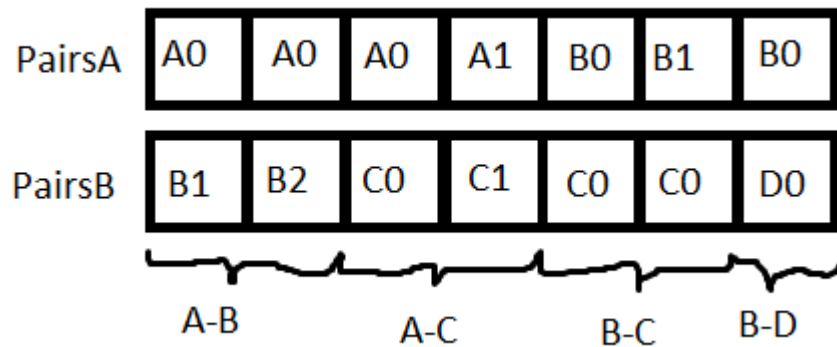
Operator 1: Operator mniejszości dla par instancji typu FeatureInstance

Przyczyna takiego uporządkowania zostanie wyjaśniona w kolejnych punktach opisu algorytmu.

6.0.3 Przygotowanie struktury szybkiego dostępu do informacji o instancjach należących do relacji o zadanych typach

W tym kroku utworzona zostanie struktura *ITMPack* (opisana w dodatku A), która potrzebna nam będzie w kolejnym kroku. Na wejściu przyjmujemy tablice $Pairs_A$ oraz $Pairs_B$, wynikiem działania jest struktura *ITMPack*.

Krok ten korzysta z wprowadzonego w pierwszym kroku uporządkowania zbioru mającego tę właściwość, iż wszystkie relacje o tych samych typach zajmują w wejściowych tablicach ciągły obszar. Przykładowe wartości zaprezentowane zostały na poniższym rysunku



Rysunek 3: Przykładowa zawartość tablic $Pairs_A$ oraz $Pairs_B$

Dzięki tej właściwości możemy użyć szybkiego wyznaczenia liczności unikalnych, następujących po sobie elementów. Funkcja ta została zaimplementowana w bibliotece *Thrust* pod nazwą *reduce_by_key*. Jako że w tym przypadku instancje są nierozróżnialne względem wartości ich identyfikatorów instancji, a tylko względem identyfikatorów typów użyty został następujący operator relacji równości

```

1 operator == (FeatureInstanceTuple lhs, FeatureInstanceTuple rhs)
2 {
3   return lhs.get[0].fields.featureId == rhs.get[0].fields.featureId
4   && lhs.get[0].fields.featureId == rhs.get[0].fields.featureId
5 }

```

Operator 2: Operator równości dla par instancji typu *FeatureInstanceTuple*

Gdzie *FeatureInstanceTuple* to obiekt typu para dwóch obiektów *FeatureInstance*, czyli obiekt łączący elementy z tablicy *Pairs_A* z odpowiadającymi im pozycją elementami tablicy *Pairs_B*. Wynikiem działania funkcji *reduce_by_key* są trzy tablice o jednakowym rozmiarze c_r

- *Uniques* zawierająca pary typów dla których istnieje przynajmniej jedna para instancji
- *Counts* zawierająca liczności instancji relacji dla par typów instancji
- *Begins* zawierająca indeksy tabeli instancji relacji oznaczające początki obszarów ciągłych odpowiadającym danym parom typów instancji

Tablice te są częścią struktury *ITMPack*. Na podstawie tych danych uzupełniana jest mapa struktury *ITMPack*, służy do tego kernel uruchamiany z ilością wątków równym c_r , który dla każdego i -tego elementu tablic tworzy strukturę *InstanceTable* uzupełniając ją i -tymi wartościami z *Counts* oraz *Begins*, a następnie wstawia go do tablicy pod kluczem utworzonym z i -tego elementu tablicy *Uniques* (klucz ten ma postać połączonej pary typów).

Tak uzupełniona struktura *ITMPack* jest wynikiem tego kroku.

6.0.4 Generowanie przewalentnych relacji sąsiedztwa pomiędzy typami połączeń

W tym kroku wyznaczane są przewalentne pary typów czyli takie których współczynnik powszechności jest nie mniejszy niż zadany próg powszechności.

Na wejściu wymagane jest podanie informacji o liczności poszczególnych typów, wygenerowanych instancji relacji sąsiedztwa oraz struktury *ITMPack*. Na wyjściu otrzymujemy listę przewalentnych par typów na podstawie których w kolejnym kroku utworzony zostanie graf do generowania kandydatów klik maksymalnych.

Na początku tego kroku tworzymy następujące tablice o rozmiarze N_{ItmUq} równym rozmiarowi tablicy *ITMPack.uniques*.

- *Counts_A* - zawiera liczbę unikalnych wystąpień instancji typu pierwszego elementu relacji na odpowiadającej pozycji w tablicy *ITMPack.uniques*
- *Counts_B* - zawiera liczbę unikalnych wystąpień instancji typu drugiego elementu relacji na odpowiadającej pozycji w tablicy *ITMPack.uniques*
- *TypesCounts_A* - zawiera całkowitą liczbę instancji typu pierwszego elementu relacji na odpowiadającej pozycji w tablicy *ITMPack.uniques*
- *TypesCounts_B* - zawiera całkowitą liczbę instancji typu drugiego elementu relacji na odpowiadającej pozycji w tablicy *ITMPack.uniques*

Rozpoczynamy od obliczenia osobno dla każdej z tablic $Counts_A$ oraz $Counts_B$ liczności unikalnych instancji dla typów równych odpowiednio pierwszym i drugim elementom relacji z tablicy $ITMPack.uniques$, licznosc unikalnych elementów danego typu jest wyznaczana na podstawie tabel $Pairs_A$ oraz $Pairs_B$ w zakresach wyznaczonych z wartości zawierających się w tabelach $ITMPack.begins$ oraz $ITMPack.counts$.

Następnie alokujemy tablice $Mask$ wielkości N_{ItmUq} zawierającą wartości "False". Następnie każdy element z wypełnionej przed chwilą tablicy dzielimy przez odpowiadającą mu wartość z tablicy $TypesCounts_A$, dla wartości z tablic B wykonujemy analogiczne działanie, na końcu sprawdzamy czy minimalna z tych wartości jest nie mniejsza niż zadany próg prewalentności i jeśli tak wpisujemy na odpowiedniej pozycji wartość "True".

Posiadając zbudowaną maskę prewalentności tworzymy dwie tabele, pierwsza wielkości N_{ItmUq} o nazwie $WritePos$, druga wielkości liczby wartości True w tabeli $Mask$ o nazwie $PrevalentPairs$ zawierającą obiekty typu $FeatureTypePair$. Tabela $WritePos$ przechowuje lewostronną sumę eksklusywną tabeli $Mask$ (wartość "True" traktowana jest jako 1, "False" jako 0). Następnie uruchamiamy N_{ItmUq} wątków na GPU, każdy i-ty wątek sprawdza czy na pozycji i w tabeli $Mask$ znajduje się wartość "True" jeśli tak to pobiera i-tą wartość z tabeli $WritePos$ będącej indeksem tabeli $PrevalentPairs$ pod którym zapisuje strukturę $FeatureTypePair$ utworzoną z relacji na i-tej pozycji w tabeli $ITMPack.uniques$.

7 Testy efektywnościowe

8 Zakończenie

Bibliografia

- [1] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, 17:37–54, 1996.
- [2] Harvey J. Miller and Jiawei Han. *Geographic Data Mining and Knowledge Discovery*. Taylor & Francis, Inc., Bristol, PA, USA, 2001.
- [3] S. Shekhar and Y. Huang. Discovering Spatial Co-location Patterns: A Summary of Results. In *SSTD 2001*, pages 236–256, 2001.
- [4] Boliński P., *Przetwarzanie zbiorów przestrzennych zapytań eksploracyjnych w środowiskach z ograniczonym rozmiarem pamięci operacyjnej*, Poznań, 2015.
- [5] Yao X., Peng L., Yang L., Chi T. A fast space-saving algorithm for maximal co-location pattern mining. *W: Expert Systems with Applications Volume 63*, 30 November 2016, strony 310–323.
- [6] Sanders J., Kandrot E., *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley, 2011.
- [7] Cheng J., Grossman M., McKerche T. *Professional CUDA C Programming*, Wrox, 2014.
- [8] Shashi Shekhar and Yan Huang. The Multi-resolution Co-location Miner: A New Algorithm to Find Co-location Patterns in Spatial Dataset. Technical Report 02-019, University of Minnesota, 2002.
- [9] Jin Soung Yoo and Shashi Shekhar. A Joinless Approach for Mining Spatial Colocation Patterns. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1323–1337, 2006.
- [10] Lizhen Wang, Yuzhen Bao, Joan Lu, and Jim Yip. A New Join-less Approach for Co-location Pattern Mining. In Qiang Wu, Xiangjian He, Quang Vinh Nguyen, Wenjing Jia, and Mao Lin Huang, editors, *Proceedings of the 8th IEEE International Conference on Computer and Information Technology (CIT 2008)*, pages 197–202, Sydney, July 2008. IEEE.
- [11] Lizhen Wang, Yuzhen Bao, and Joan Lu. Efficient Discovery of Spatial Co-Location Patterns Using the iCPI-tree. *The Open Information Systems Journal*, 3(2):69–80, 2009.
- [12] Christopher Jones and Mark Hall. A Field Based Representation for Vague Areas Defined by Spatial Prepositions. In *Proceedings of the Workshop on Methodologies and Resources for Processing Spatial Language at 6th Language Resources and Evaluation Conference (LREC 2008)*, 2008.
- [13] Max J. Egenhofer and Robert Franzosa. Point-set topological spatial relations. *International Journal of Geographic Information Systems*, 5(2):161–174, 1991.
- [14] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. A small set of formal topological relationships suitable for end-user interaction. In *Proceedings of the 3rd International Symposium on Advances in Spatial Databases (SSD 1993)*, pages 277–295, London, UK, UK, 1993. Springer-Verlag.

- [15] Harvey J. Miller and Jiawei Han. Geographic Data Mining and Knowledge Discovery. Taylor & Francis, Inc., Bristol, PA, USA, 2001.
- [16] John F. Roddick and Myra Spiliopoulou. A Bibliography of Temporal, Spatial and Spatio-Temporal data Mining Research. ACM SIGKDD Exploration Newsletter, 1(1):34–38, 1999.
- [17] Martin Ester, Hans-Peter Kriegel, and Jörg Sander. Spatial Data Mining: A Database Approach. In Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD 1997), pages 47–66, London, UK, UK, 1997. Springer-Verlag.
- [18] John R. Quinlan. Induction of Decision Trees. Machine Learning, 1(1):81–106, March 1986.
- [19] Martin Ester, Alexander Frommelt, Hans-Peter Kriegel, and Jörg Sander. Algorithms for characterization and trend detection in spatial databases. In Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD 1998), pages 44–50, 1998.
- [20] Shashi Shekhar and Sanjay Chawla. Spatial Databases: A Tour. Prentice Hall, 2003.
- [21] Richard E. Bellman. Adaptive control processes - A guided tour. Princeton University Press, Princeton, New Jersey, U.S.A., 1961.
- [22] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994), pages 487–499, San Francisco, 1994. Morgan Kaufmann Publishers Inc.
- [23] Krzysztof Koperski and Jiawei Han. Discovery of Spatial Association Rules in Geographic Information Databases. In Max J. Egenhofer and John R. Herring, editors, Proceedings of the 4th International Symposium on Advances in Spatial Databases (SSD 1995), volume 951 of Lecture Notes in Computer Science, pages 47–66. Springer Berlin Heidelberg, 1995.
- [24] Yasuhiko Morimoto. Mining Frequent Neighboring Class Sets in Spatial Databases. In Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2001), pages 353–358, New York, NY, USA, 2001. ACM.
- [25] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. Vitter. Scalable Sweeping- Based Spatial Join. In Proc. of the Int’l Conference on Very Large Databases, 1998.
- [26] Eppstein, D. , Löffler, M. , i Strash, D. (2010). Listing all maximal cliques in sparsegraphs in near-optimal time. In O. Cheong, K. Y. Chwa, & K. Park (Eds.), 21st international symposium on algorithms and computation (pp. 403–414). Berlin, Germany: Springer-Verlag.
- [27] Bron, C., & Kerbosch, J. (1973). Algorithm 457: Finding all cliques of an undirected graph. Communications of the ACM, 16 , 575–577.

- [28] Tomita, E., Tanaka, A., & Takahashi, H. (2006). The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363 , 28–42.
- [29] Cazals, F.; Karande, C. (2008), "A note on the problem of reporting maximal cliques" , *Theoretical Computer Science*, 407 (1): 564–568.
- [30] Wang, L., Zhou, L., Lu, J., & Yip, J. (2009). An order-clique-based approach for mining maximal co-locations. *Information Sciences*, 179 , 3370–3382.
- [31] Matula, D. W.; Beck, L. L. (1983), "Smallest-last ordering and clustering and graph coloring algorithms", *Journal of the ACM*, 30 (3): 417–427, doi:10.1145/2402.322385, MR 0709826.
- [32] Comparing the Concurrency Runtime to Other Concurrency Models, Microsoft, 2015 [dostęp: 10-01-2017]. Dostępny w Internecie: <https://msdn.microsoft.com/en-us/library/dd998048.aspx#openmp>
- [33] Microsoft Parallel Patterns Library (PPL) vs. OpenMP, Stack Overflow, 14-03-2012 [dostęp: 10-01-2017]. Dostępny w Internecie: <http://stackoverflow.com/questions/9700088/microsoft-parallel-patterns-library-ppl-vs-openmp>
- [34] <https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6>
- [35] . Sorting in PPL, Parallel Programming in Native Code, vinodsu (Microsoft Developer Network Blogs), 14-01-2011 [dostęp: 10-01-2017]. Dostępny w internecie: <https://blogs.msdn.microsoft.com/nativeconcurrency/2011/01/14/sorting-in-ppl/>
- [36] <https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles>

A Opis klas i struktur pomocniczych

A.1 Opis architektury wersji CPU

Podczas implementacji stworzono dwie główne klasy: `CpuMiningAlgorithmSeq` oraz `CpuMiningAlgorithmParallel` dziedziczące po klasie abstrakcyjnej `CpuMiningAlgorithmBase`. `CpuMiningAlgorithmBase` zawiera cztery metody czysto wirtualne (ang. *pure virtual functions*) odpowiadające za główne kroki algorytmu:

- Filtrowanie sąsiadów na podstawie progu odległości:

```
1 virtual void filterByDistance(float threshold) = 0;
```

- Filtrowanie sąsiadów na podstawie progu minimalnej powszechności:

```
1 virtual void filterByPrevalence(float prevalence) = 0;
```

- Generowanie kandydatów na kolokacje maksymalne:

```
1 virtual void constructMaximalCliques() = 0;
```

- Generowanie skondensowanych drzew instancji kandydatów na kolokacje maksymalne i ich filtrowanie na podstawie progu minimalnej powszechności:

```
1 virtual std::vector<std::vector<unsigned short>>  
2     filterMaximalCliques(float prevalence) = 0;
```

Klasy `CpuMiningAlgorithmSeq` oraz `CpuMiningAlgorithmParallel` zawierają implementację owych metod odpowiednio w wersji sekwencyjnej jak i równoległej.

Oprócz wyżej wspomnianych metod klasa `CPUMiningAlgorithmBase` zawiera implementację metod pomocniczych odpowiadających za:

- Obliczenie odległości między instancjami i porównanie ich z obowiązującym progiem odległości.
- Obliczenie wartości funkcji powszechności dla dwóch elementów.
- Obliczenie wartości funkcji powszechności dla dowolnej liczby elementów, za równo w wersji sekwencyjnej i równoległej.
- Wygenerowanie wszystkich podklik kliku K o rozmiarze n takich, że rozmiar każdej podkliki wynosi $n - 1$.

A.2 Opis struktur & klas pomocniczych i ich implementacji

- DataFeed - Instancje struktury DataFeed mają za zadanie przechowywać dane wejściowe.

```
1 struct DataFeed
2 {
3     struct Coords
4     {
5         float x;
6         float y;
7         Coords(float x, float y) : x(x), y(y) {}
8     };
9
10    unsigned short type;
11    unsigned short instanceId;
12    Coords xy;
13
14    DataFeed(unsigned short type,
15            unsigned short id,
16            float x,
17            float y)
18        : type(type), instanceId(id), Coords(x,y) {}
19
20    bool operator < (const DataFeed& str) const
21    {
22        return (type == str.type)?(instanceId < str.instanceId)↵
23            :(type < str.type);
24    }
25};
```

Listing A.1: Kod struktury DataFeed

- FeatureInstance - Unia składająca się z pola typu `unsigned int` oraz struktury zawierającej dwa pola typu `unsigned short` służąca do przechowywania identyfikatora typu oraz unikalnego względem typu identyfikatora instancji.

```
1 union __declspec(align(4)) FeatureInstance
2 {
3     unsigned int field;
4
5     struct __inner
6     {
7         unsigned short instanceId;
8         unsigned short featureId;
9     } fields;
10};
```

- InstanceTable - klasa przechowująca informacje o indeksie startowym w tablicach przechowywujących posortowane instancje relacji sąsiedztwa oraz ich ilości.

```

1  class InstanceTable
2  {
3  public:
4      unsigned int count;
5      unsigned int startIdx;
6  };

```

Listing A.2: Kod klasy InstanceTable

- ITMPack - struktura przechowująca informacje potrzebne do szybkiego dostępu do wszystkich instancji relacji o zadanych typach (na przykład wszystkich instancji relacji o typach {A,B}).

```

1
2  struct ITMPack
3  {
4      InstanceTableMapPtr map;
5      thrust::device_vector<unsigned int> begins;
6      thrust::device_vector<unsigned int> counts;
7      thrust::device_vector<FeatureInstanceTuple> uniques;
8      unsigned int count;
9  };

```

Listing A.3: Kod struktury ITMPack

tabele *uniques*, *begins*, *counts* zawierają odpowiednio pary typów będących w relacji, indeks pierwszej instancji pary w tabelach $Pairs_A$ oraz $Pairs_B$ oraz licznosc grupy instancji relacji. Pole map zapewnia dostęp do hash mapy o kluczu będącym parą typów umożliwiającą (w przestrzeni obliczeniowej GPU) na szybki dostęp do informacji o początku i liczności grupy instancji relacji sąsiedztwa danego typu co pozwala wyznaczyć jej zakres.

- FeatureTypePair - struktura przechowująca parę typów instancji, pozwalająca na szybkie pobranie obu wartości

```

1
2  union __declspec(align(4)) FeatureTypePair
3  {
4      unsigned int combined;
5
6      struct
7      {
8          unsigned short b;
9          unsigned short a;
10     } types;
11 };

```

Listing A.4: Kod struktury ITMPack

- CinsTree - Klasa zawierająca implementację skondensowanego drzewa instancji (ang. *condensed instance tree*). Zawiera wskaźnik do korzenia drzewa `std::unique_ptr<CinsNode>root` oraz wektor `std::vector<CinsNode*>lastLevelChildren` zawierający wskaźniki do liści znajdujących się na ostatnim poziomie (wynikającym z rozmiaru aktualnie budowanej klik - w przypadku gdy nie udało się rozbudować drzewa o kolejny poziom lista ta jest pusta) owego drzewa. Takie podejście umożliwia szybki dostęp do finalnych instancji poszczególnych kandydatów na kolokacje.
- CinsNode - Klasa zawierająca implementację pojedynczego węzła skondensowanego drzewa instancji *CinsTree*. Obiekty tej klasy zawierają informacje o cesze `unsigned short type` i numerze `unsigned short instanceId` instancji przestrzennej, wektor wskaźników potomków `std::vector<std::unique_ptr<CinsNode*>>children` oraz wskaźnik pokazujący na rodzica danego węzła `CinsNode* parent`. Klasa zawiera metody umożliwiające m.in.:
 - dodanie potomka.
 - zwrócenie potomka o danej cesze i numerze instancji.
 - zwrócenie listy wszystkich przodków.
- Graph - Klasa implementująca graf nieskierowany, bazująca na macierzy sąsiedztwa (ang. *adjacency matrix*). Oprócz podstawowych metod umożliwiających działanie i budowanie grafu, klasa zawiera również metody pozwalające na:
 - obliczenie maksymalnych klik w grafie za pomocą zmodyfikowanego algorytmu Brona-Kerboscha [5].
 - obliczenie optymalnego *pivotu* [28] dla algorytmu Brona-Kerboscha.
 - obliczenie stopnia degeneracji grafu (ang. *degeneracy*) i uporządkowanie wierzchołków według miary degeneracji (ang. *degeneracy ordering*) [26].

```

1 unsigned short Graph::tomitaMaximalPivot(
2     const std::vector<unsigned short>& SUBG,
3     const std::vector<unsigned short>& CAND)
4 {
5     unsigned short u, maxCardinality = 0;
6     for (auto s : SUBG)
7     {
8         auto neighbors = getVertexNeighbours(s);
9         std::sort(neighbors.begin(), neighbors.end());
10
11         std::vector<unsigned short> nCANDunion(neighbors.size() ←
12             + CAND.size());
13
14         auto itUnion = std::set_union(
15             CAND.begin(),
16             CAND.end(),
17             neighbors.begin(),
18             neighbors.end(),
19             nCANDunion.begin());
20
21         nCANDunion.resize(itUnion - nCANDunion.begin());
22         if (nCANDunion.size() >= maxCardinality)
23         {
24             u = s;
25             maxCardinality = nCANDunion.size();
26         }
27     }
28     return u;
29 }

```

Listing A.5: Kod metody `tomitaMaximalPivot` umożliwiającej wyliczenie optymalnego punktu *pivot*

- `SubcliquesContainer` - Klasa umożliwiająca przechowywanie przetworzonych już kandydatów na kliki maksymalne. W łatwy i wydajny sposób ułatwia sprawdzenie czy dana klika lub klika będąca nadzbiorem danej cliki została już przetworzona - dzięki temu unika się przeprowadzenia wtórnych obliczeń i ewentualnych duplikatów w rozwiązaniu. Główną ideą jest stworzenie mapy wektorów `std::map<short, std::vector<unsigned short>>typesMap`, której kluczami są numery cech występujące w poszczególnych klikach a wartościami wektory kolejnych wartości licznika `unsigned int cliquesCounter` którego bieżąca wartość służy do oznaczania kolejnych klik.

Weryfikację umożliwia algorytm:

```

1 bool SubcliquesContainer::checkCliqueExistence(
2     std::vector<unsigned short>& clique)
3 {
4     assert(clique.size() >= 2);
5
6     std::vector<bool> types(cliquesCounter, false);
7     std::vector<bool> typesNew(cliquesCounter, false);
8
9     for (auto type : typesMap[clique[0]])
10    {
11        types[type] = true;
12    }
13
14    for (auto i = 1; i < clique.size(); ++i)
15    {
16        for (auto id : typesMap[clique[i]])
17        {
18            if (types[id]) typesNew[id] = true;
19        }
20        types = typesNew;
21        std::fill(typesNew.begin(), typesNew.end(), false);
22    }
23
24    if (std::find(types.begin(), types.end(), true) != types.end()↵
25        ())
26        return true;
27    return false;
28 }

```

Listing A.6: Kod metody checkCliqueExistence klasy SubcliquesContainer

- ParallelSubcliquesContainer - Klasa odpowiedzialną za tą samą funkcjonalność co klasa *SubcliquesContainer* jednakże zapewniająca bezpieczeństwo przetwarzania wielowątkowego. Cel ten osiągnięto za pomocą skorzystania z sekcji krytycznych w przypadku inkrementowania licznika jak i posłużenia się współbieżnymi wektorami `concurrency::concurrent_vector<unsigned short>` z biblioteki PPL, co przełożyło się również na zwiększenie efektywności przetwarzania.
- CliquesContainer - Klasa zapewniająca dwie funkcjonalności:
 - Sprawdzenia czy dokładnie taka klika jest już przechowywana, za co odpowiada funkcja `bool checkCliqueExistence(std::vector<↵ unsigned short>& clique)`
 - Sprawdzenie czy taka klika lub jej dowolna podklika jest już przechowywana, odpowiada za to funkcja:

```

1 bool CliquesContainer::checkSubcliqueExistence(
2     std::vector<unsigned short>& clique)
3 {
4     bool isSubclique;
5     for (auto& c : cliques)
6     {
7         if (clique.size() < c.size()) continue;
8         auto it = clique.begin();
9         isSubclique = true;
10        for (auto id : c)
11        {
12            it = std::find(it, clique.end(), id);
13            if (it == clique.end()) {
14                isSubclique = false;
15                break;
16            }
17        }
18        if (isSubclique) return true;
19    }
20    return false;
21 }

```

Listing A.7: Kod metody checkSubcliqueExistence klasy CliquesContainer

- ParallelCliquesContainer - Klasa odpowiedzialna za tą samą funkcjonalność co klasa *CliquesContainer* zapewniająca w tym samym czasie bezpieczeństwo przetwarzania wielowątkowego.
- RandomDataProvider - Klasa zapewniająca losowy generator danych z parametryzowaną liczbą cech, liczbą instancji a także granicami danych przestrzennych.
- SimulatedRealDataProvider - Klasa wykorzystująca gotowe, przygotowane wcześniej dane wczytywane z plików mające symulować dane rzeczywiste.
- pair_hash - Struktura zapewniająca generyczną implementację funkcji hashującej (ang. *hash function*) dla pary `std::pair<T1, T2>` - w przypadku typu zdefiniowanego przez użytkownika konieczne jest własnoręczne przeładowanie operatora `()`. Aby zapewnić odpowiednią wydajność, należy zadbać o właściwą funkcję hashującą tzn. taką która generuje możliwie mało kolizji. Popularną metodą jest skorzystanie z funkcji XOR i zastosowanie jej do dających się pojedynczo hashować elementów pary. Okazało się jednak, że funkcja ta generuje niezadowalająco dużą ilość kolizji, dla tego stworzono bardziej zaawansowany hasher korzystający z funkcji `hash_combine` z biblioteki *boost*:

```

1 struct pair_hash {
2     template <class T1, class T2>
3     std::size_t operator () (const std::pair<T1, T2> &p) const {
4         std::size_t seed1(0);
5         ::hash_combine(seed1, p.first);
6         ::hash_combine(seed1, p.second);
7
8         std::size_t seed2(0);
9         ::hash_combine(seed2, p.second);
10        ::hash_combine(seed2, p.first);
11
12        return std::min(seed1, seed2);
13    }
14 };

```

Listing A.8: Kod struktury pair_hash

Funkcja hash_combine:

```

1 template<typename T>
2 void hash_combine(std::size_t &seed, T const &key) {
3     std::hash<T> hasher;
4     seed ^= hasher(key) + 0x9e3779b9 + (seed << 6) +
5         (seed >> 2);
6 };

```

Listing A.9: Kod funkcji hash_combine

- vector_hash - Struktura umożliwiająca kodowanie mieszające (ang. *hashing*) dla wektorów dowolnych typów dla których istnieje implementacja funkcji hashującej. Również w tym przypadku skorzystano z funkcji hash_combine.

```

1 struct vector_hash {
2     template <class T>
3     std::size_t operator () (std::vector<T> const& vec) const
4     {
5         std::size_t seed = vec.size();
6         for (auto& i : vec) {
7             ::hash_combine(seed, i);
8         }
9         return seed;
10    }
11 };

```

Listing A.10: Kod struktury vector_hash

- Timer - Generyczna klasa umożliwiająca pomiar czasu dla dowolnej funkcji lub funktora z dowolną liczbą argumentów. Zapewnia ustawienie dowolnego typu zegara i dokładności pomiaru. Odpowiada za sprawdzenie czasu wykonywania poszczególnych kroków algorytmu.
- Benchmark - Klasa umożliwiająca przeprowadzenie parametryzowanych testów wydajnościowych całego algorytmu, zapewniając m.in. serializację do

pliku. Wyeksportowane dane mogą zostać zwizualizowane za pomocą wykresów do których tworzenia wykorzystano odpowiedni skrypt w języku Python.

Wejście: $C_m, InsTable_2$
Wyjście: $CInsTree$ - skondensowane drzewo instancji C_m

```

1  $i \leftarrow 1; CInsTree \leftarrow \emptyset$ ; utwórz korzeń drzewa  $CInsTree$ ;
2 while  $i < size(C_m)$  do
3   if  $i = 1$  then
4     foreach para instancji  $InsPair_k \in InsTable_2(C_m(1), C_m(2))$  do
5       if  $InsPair(1) \in CInsTree_0.children$  then
6         dodaj węzeł podrzędny  $InsPair_k(2)$  do
            $CInsTree_1(InsPair_k(1))$ ;
7       else
8         utwórz poddrzewo z  $InsPair_k(1)$  jako korzeniem i  $InsPair_k(2)$ 
           jako pierwszym dzieckiem;
9         dołącz to poddrzewo do korzenia  $CInsTree$ ;
10      end
11    end
12  else
13    foreach węzeł instancji  $ins_k \in CInsTree_i$  do
14      znajdź indeksy elementów równych  $ins_k$  od pierwszej kolumny
         $InsTable_2(C_m(i), C_m(i+1))$ ;
15      przechowaj drugi element odpowiadającej pary instancji w liście
         $El$ ;
16      foreach  $ei_t \in El$  do
17         $flag \leftarrow i - 1$ ;
18         $currIns \leftarrow ins_k.parent$ ;
19        while  $flag \geq 1$  do
20          if  $(currInt, ei_t) \in InsTable_2(C_m(flag), C_m(i+1))$  then
21             $currIns \leftarrow currIns.parent$ ;
22          else
23            break;
24          end
25           $flag \leftarrow flag - 1$ ;
26        end
27        if  $flag = 0$  then dodaj węzeł podrzędny  $ei_t$  do
           $CInsTree_i(ins_k)$ ;
28      end
29    end
30  end
31 end

```

Algorytm 2: Konstrukcja skondensowanego drzewa instancji C_m

Wejście: $G = (E, V)$
Wyjście: k (miara degeneracji), L (lista wierzchołków uporządkowanych według miary degeneracji)

```

1  $L \leftarrow \emptyset$ ;
2  $D \leftarrow \emptyset$ ;
3 foreach  $v_i \in G$  do
4    $D(d_v) \leftarrow$  liczba sąsiadów  $v \notin L$  (początkowo równa stopniowi
   wierzchołka);
5 end
6  $k \leftarrow 0$ ;
7 foreach  $v_i \in G$  do
8   znajdź takie  $i$ , dla którego  $D(i) \neq \emptyset$ ;
9    $k \leftarrow \max(k, i)$ ;
10   $v \leftarrow$  wierzchołek z  $D(i)$ ;
11   $L \leftarrow \{v\} \cup L$ ;
12   $D(i) \leftarrow D(i) \setminus \{v\}$ ;
13  foreach  $w \leftarrow$  sąsiadzi  $v \notin L$  do
14     $d'_w \leftarrow d_w - 1$ ;
15     $D(d_w) \leftarrow D(d_w) \setminus \{w\}$ ;
16     $D(d'_w) \leftarrow D(d'_w) \cup w$ ;
17  end
18 end

```

Algorytm 3: Obliczanie miary degeneracji metodą Matuli i Becka (1983)

Wejście: $G = (E, V)$

```

1  $R \leftarrow \emptyset$ ;  $X \leftarrow \emptyset$ ;  $P \leftarrow V(G)$ ;
2 foreach  $v_i^*$  in lista  $v_1^*, v_2^*, \dots, v_\lambda^*$  uporządkowanych wg. degeneracji do
3    $BK\_Pivot(R \cup \{v_i^*\}, P \cap N(v_i^*), X \cup N(v_i^*))$ ;
4    $P \leftarrow P \setminus \{v_i^*\}$ ;
5    $X \leftarrow X \cup \{v_i^*\}$ ;
6 end

```

Algorithm 1: Równoległy algorytm Brona-Kerboscha

Wejście: a, b – instancje typu *FeatureInstance*
Wyjście: *bool*

```

1 return  $a.field < b.field$  ;

```

Algorytm 4: Operator mniejszości dla instancji typu *FeatureInstance*