Térbeli együttes előfordulási minták GPU-val gyorsított felismerése

A GPU-CM algoritmus

Gyenes Csilla
Sallai Levente
Szabó Andrea

Absztrakt

Az egybefüggő minták felfedezése egy érdekes része a térbeli adatbázisok adatbányászásának. Olyan térbeli objektum típusok kereséséből áll, melyek gyakran együtt állnak a térbeli szomszédságban. Ezen minták néhány alkalmazási területe: biológia, földrajz, marketing és meteorológia. Ahhoz hogy megbirkózhassunk az ilyen nagy mennyiségű adatokkal, felhasználhatjuk a (nagy teljesítményű) programozható grafikus kártyákat (GPU). A GPU-król már bebizonyosodott az utóbbi időben, hogy rendkívül hatékonyan képesek felgyorsítani számos már meglévő algoritmust. Ebben a cikkben bemutatjuk a GPU-CM-et, ami egy GPU-gyorsított verziója az iCPI-fára alapozott algoritmusnak együttes előfordulási minták felfedezésére. A legjobb teljesítmény érdekében, speciálisan megtervezett struktúrákat és eljárásokat vezetünk be, hogy kiaknázhassuk az SIMD végrehajtási modell lehetőségeit. Kísérleti jelleggel összehasonlítjuk az iCPI-fa módszerek GPU-s és párhuzamos implementációit. A kapott eredmények bizonyítják, hogy a CPU-s algoritmushoz képest a GPU-s nagyságrendekkel gyorsabb.

1. Bevezetés

A térbeli adatbázisok hatalmas növekedése már meghaladja az emberi feldolgozás lehetőségét. Ezért van szükség az automatikus felismerésre, ezeket az eljárásokat Knowledge Discovery in Databases - KDD-nek (“tudás felfedezés adatbázisokban”) nevezik [8]. Ennek legérdekesebb része az úgynevezett adatbányászás és bizonyos minták felfedezésére speciálisan felépített algoritmusok alkalmazásából áll.

Ebben a munkában elsősorban a térbeli objektum osztályok (térbeli tulajdonságok) felfedezésével foglalkozunk, melyek gyakran csoportosulnak. Minden térbeli tulajdonság definiálható a tér egy adott pontbeli tulajdonságaként. Tipikus példák: fajok, üzleti típusok, érdekes pontok (pl.: kórházak, repterek, látnivalók, stb.). Vegyünk például egy mobiltelefon hálózatot üzemeltető céget, ami számos szolgáltatást nyújt; érdekelhetik, hogy milyen kapcsolat van azon tényezők között, amik alapján az ügyfelek szolgáltatást választanak. A tudósok számára érdekes lehet az ökológiában és meteorológiában együttesen előforduló természeti jelenségek[12].

A térbeli egybefüggő minta (röviden csoport/egybefüggő) olyan térbeli tulajdonságok halmaza, amelyek a térben egymáshoz közel állnak. Az ilyen minták azonosításához fontos, hogy felkutassuk a minták összes előfordulását. Számos csoport minta felfedező algoritmust írtak már [11-14, 16, 15]. De még nem terjesztettek elő egy olyan megoldást sem, amely hardver támogatást vett volna igénybe.

2. Definíciók

D1: Legyen *f* egy térbeli tulajdonság. Az *x* objektum az *f* egy példánya, ha *x f* egy típusa és van helyzete és egyedi azonosítója. Legyen *F* térbeli tulajdonságok halmaza és *S* ezek példányainak halmaza. Adott szomszédsági reláció, *R*, mellett azt mondjuk, hogy a *C* **csoport minta** részhalmaza a térbeli tulajdonságoknak $C \subseteq F$, melynek példányai $I \subseteq S$, egy klikket alkotnak *R* relációval.

D2: Az f_i tulajdonság **részvételi rátája** (participation ratio) $Pr(C, fi), C = \{f_1, \ldots, f_k\}$ csoportban, az f_i tulajdonságot képviselő objektumok töredéke a példányok csoportjának szomszédságában $C - \{f_i\}$. $Pr(C, f_i)$ az f_i *C*-beli példányainak száma osztva a összes f_i tulajdonságú elemmel. A **részvételi indexe** (participation index) $P_i(C)$, egy $C = \{f_1, \ldots, f_k\}$ csoportnak: $P_i(C) = \min_{f_i \in C}\{Pr(C, f_i)\}$.

T1: A részvételi ráta és a részvételi index monoton csökken a C csoport méretének növekedésével.

D3: Csoport minta bányászat:

1. adott a térbeli tulajdonságok halmaza $F = \{f_1, \ldots, f_k\}$
2. azok példányai $S = \bigcup_1^k S_i$, ahol S_i (1 < i < k) az f_i tulajdonságú példányok halmaza, és minden *S*-beli példány tartalmaz információt a típusáról, egyedi azonosítójáról és helyzetéről
3. adott egy szomszédsági reláció a helyzetek felett értelmezve: *R*
4. adott egy minimum gyakorisági küszöb (min_prev).

Ekkor hatékonyan találunk helyes és teljes csoport mintákat, ahol a részvételi index min_prev.

3. Kapcsolódó munkák

3.1 Az iCPI-fa bázisú csoport minta keresés

Az egybefüggő minták bányászatának egy általános megközelítése bemutatásra került [11]-ben, és három fő lépésből áll. Az első lépésben egy jól ismert apriori stratégiát [1] használunk, hogy lehetséges csoportokat generáljuk a gyakorisági mérték (P_i) monotonitását (↘) kihasználva. A következő lépésben meghatározzuk az így kapott csoportok példányait. Végül kiszámítjuk a részvételi indexet minden csoporthoz. Azon jelölteket, amik a minimum gyakorisági küszöb alattiak, elhagyjuk.

Ugyan nem minden kutatás követi az előbb bemutatott apriorisztikus stratégiát (pl.: maximális csoport minták [16], csoport minták sűrűség alapján [14]), az általános megközelítés még mindig a legnépszerűbb. Az általános megközelítések közül a legjelentősebb eljárások a Co-Location Miner (csoportosulás bányász)[11], Joinless (kapcsolat nélküli)[15] és az iCPI-fa eljárás jelenlegi állapota [13]. A következőkben röviden ismertetjük az algoritmus mögötti ötletet.

Kezdetben minden térbeli tulajdonság egy elemű csoportként van értelmezve. A következő lépésben két elemű jelölteket alkotunk belőle az apriorisztikus stratégiával. Ahhoz hogy kiszámíthassuk a gyakoriságukat, szükség van a példányaik listájára. Az iCPI-fa eljárásban a csillag szomszédság fogalmát (eredetileg [15]-ben bemutatott) alkalmazzuk.

D4: Minden objektumra a térben, az olyan szomszédok listáját, amelyek térbeli tulajdonságai nagyobbak, mint az adott objektumé, csillag szomszédságnak nevezzük.

Ezt az információt tároljuk iCPI-fa formájában. A gyökér minden gyermeke egy részfa, amely egy bizonyos térbeli tulajdonság példányainak szomszédjait tárolja. A részfák a szomszédok térbeli tulajdonságait tároló csúcsokból, és a szomszéd példányoknak megfelelő levelekből áll. Egy példa adathalmazt és a hozzátartozó iCPI-fát az *1. ábrán* láthatjuk.

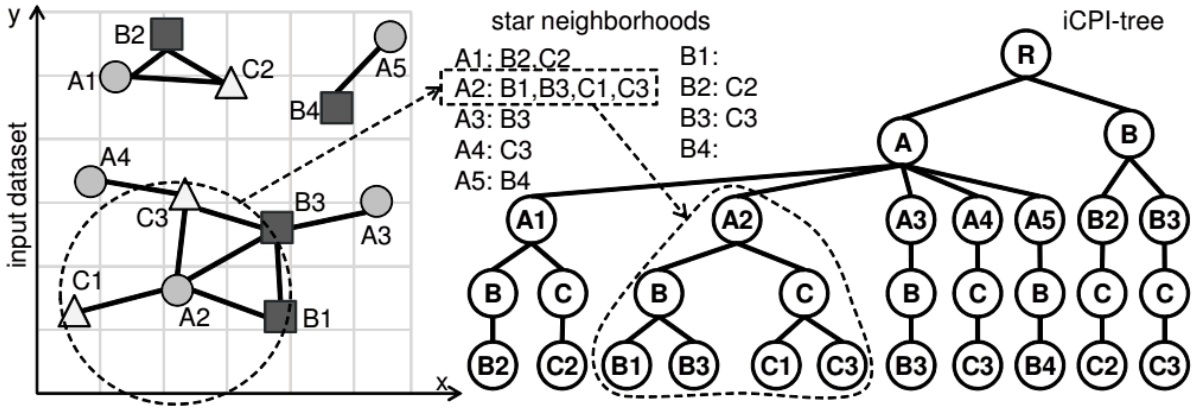


Fig. 1. Sample dataset and the corresponding iCPI-tree

Például A2 csillag szomszédsága B1, B3, C1, C3 példányokból áll, melyet egy részfa szimbolizál az A csúcs alatt az iCPI-fában. Egy jelölt csoport példányainak meghatározásához, pl. {B, C}-hez könnyen le tudjuk olvasni B példányainak összes C tulajdonságú szomszédját (B2, C2 és B3, C3).

Az algoritmus n-dik lépésében n méretű jelöltek jönnek létre. n = 2-re minden a fából generált példány egy csoport (D1). n > 2-re a következő eljárással generáljuk a csoportok példányait: hogy beazonosítsuk az n méretű csoport jelöltek példányait, a példányok n-1-dik lépésbeli halmazát használjuk. Minden n méretű jelölt, annak az n-1 méretű csoportnak a kibővítése, amely túlnyomó mértékben tartalmazza az első n-1 tulajdonságot a jelöltből. Csak a közös szomszédokat használhatjuk minden csoport bővítésére. Például, adott jelölt {A, B, C}-re vegyük egymás után a példányokat {A, B} csoportból, pl. {A2, B3} példány. Megpróbáljuk kibővíteni C-beli példányokkal. Hogy megkapjuk a csoport példányait {A, B, C} jelöltnek, keressük azon C típusú példányokat, amelyek szomszédjai A2-nek és B3-nak is egyszerre. A kapott iCPI-fát felhasználva láthatjuk, hogy A2-nek szomszédjai C1 és C3, míg B3-nak csak C3 a szomszédja. Ezért az egyetlen közös szomszéd C3 és így a jelölt egy új példánya {A2, B3, C3}. Ezt az eljárást ismételjük minden jelöltre.

Az iCPI-fa eljárás további részleteiért lásd a [13]-as cikket.

3.2 Általános feldolgozás a GPU-n

A GPU a számítási feladatokat ún. kernelek formájában kapja. D5: A kernel olyan feladat, amely műveletek sorozatából áll, amiket több konkurens szálon kell végrehajtani. A szálakat a programozó azonos méretű blokkokra osztja. D6: A blokk egy 1, 2 vagy 3 dimenziós tömb legfeljebb 1024 szállal (vagy 512-vel, a grafikus kártya architektúrájától függően), ahol minden egyes szálat egyedileg azonosíthatunk a tömbben elfoglalt helyével. A blokkok halmaza egy ún. számítási hálót alkot. A szálak egy blokkon belül úgy kommunikálnak, hogy ugyanazt a részét használják az ún. megosztott memóriának, amely fizikailag a csippen található, ezért nagyon gyors. Más blokkbeli szálakkal a kommunikáció a grafikus kártya lassú, globális memóriájában történik. Különböző memória típusoknak, különböző hatékonyságú elérési követelménye van. A szálak szinkronizációs lehetőségei korlátozottak. Szálak egy blokkon belül szinkronizálhatóak; bár a globális szinkron is elérhető, de csak költséges megoldással. Egy blokk szálait 32 szálas SIMD csoportokban hajtják végre, amit kötegnek (csomagnak) nevezünk (ezeknek a szálaknak ugyanazt az utasítást kell elvégezniük egyidejűleg = SIMD). A programozónak figyelembe kell vennie mindezeket az alacsony szintű GPU korlátokat, hogy az algoritmus a leghatékonyabb lehessen.

Hogy megkönnyítvén a párhuzamos számításokat végző programok készítését (nem feltétlen GPU) számos párhuzamos primitív létezik. A továbbiakban a következőket fogjuk használni: inclusive és exclusive scan, compact, sort, unique, reduce és reduce by key. A legtöbbet ezek közül implementálták GPU-ra is például a Thrust könyvtárban [3]. A mi megvalósításunkban ezt a könyvtárat használjuk majd, bár megírtuk a saját verziónkat is a tömörítő algoritmusra. Alább röviden bemutatjuk a primitíveket.

D7: Adott a tömbre az **belső vizsgálat** (inclusive scan) talál egy olyan b tömböt, amely hossza megegyezik a-éval és minden elemére teljesül: bi=k = 1iai. A **külső vizsgálat** (exclusive scan) hasonlóan működik, de b minden elemére: bi=k =1i-1ai. Bármilyen asszociatív bináris műveletet használhatunk a summa helyett.

D8: A **tömörítő algoritmus** (compact) adott a tömbből kiveszi azon elemeket, amelyek teljesítenek bizonyos feltétel(eke)t. Leggyakrabban egy további flag tömböt használnak 0-ákkal és 1-esekkel (eltávolít/marad) erre a célra.

D9: A **rendezés** bármely tömböt rendezi. A rendezés történhet további kulcs tömb alapján is.

D10: Az **egyedi algoritmussal** (unique) kiválasztjuk az eltérő elemeket egy felhasználó által megadott adat tömbből. A Thrust implementáció megköveteli, hogy a tömb legyen rendezett.

D11: A redukáló valamely asszociatív bináris művelettel redukálja a felhasználó által megadott tömböt, pl.: összegzi a tömb minden elemét. A kulcs alapján történő redukció sokkal kifinomultabb verziója a redukálónak. Két adott tömbre, k-ra és v-re, ahol k a kulcsok tömbje, v az értékeké, a redukáló azon elemeket redukálja, melyek kulcsa azonos. A Thrust implementáció megköveteli, hogy az adatok a kulcs alapján rendezve legyenek.

4. A GPU-CM algoritmus

A GPU-CM algoritmus felteszi, hogy az iCPI-fa már felépült. Erre már léteznek hatékony algoritmusok [13, 15]. Továbbá ez az építő algoritmus csupán apró töredékét teszi ki a teljes adatbányász folyamatnak [4].

4.1 Adatstruktúrák

A [4]-ben bemutatott megoldást követve az iCPI-fát hashtáblaként ábrázoljuk. A hashtábla minden eleme egy listát tárol egy adott f1 térbeli tulajdonság példányaival, melyek mind szomszédjai egyazon i példánynak egy másik f2 tulajdonságból. Ez a struktúra a GPU-n a következő képpen néz ki:

A példányok szomszéd listái egy memória blokkban foglalnak helyet. Hatékonysági okokból minden listának konstans L hossza van, amely kettő bármely hatványa lehet (32-ig). Ha szomszédok száma kisebb mint L, akkor néhány hely üresen marad. Ezért a memória blokk mérete L-szer a listák száma. A lista minden eleme egy példányt jelöl. Minden tulajdonság példányt 32 bit hosszú szóként ábrázolunk, ahol a legkisebb helyiértékű 16 biten tároljuk a példány számot, a következő 8 biten a tulajdonság számát. A legnagyobb helyiértékű 8 bitet pedig nullákra állítjuk. Az üres listaelemet az FFFFFFFFh hexadecimális számmal jelöljük. Minden lista folytonos L×4 bájtnyi memóriát foglal el. Minden listát rendezünk. Ezt az adatstruktúrát példány szomszéd puffernek nevezzük.

Egy hashtáblát két tömbbel ábrázolunk: egy kulcs és egy érték tömbbel. Minden kulcs tárolja az f1 és az f2 tulajdonságok számait és i példány számát is. Ez egy 32 bites szóban van lekódolva, ahol a legkisebb helyiértékű 16 biten i példány számát jegyezzük, a következő 8 biten az f2 tulajdonság azonosítóját, a legnagyobb helyiértékű 8 biten pedig f1-ét. A kulcs tömb minden eleme egy kulcsot tárol, vagy üres (FFFFFFFFh). A nyitott címzéses sémát használjuk a hasításhoz [2]. Az érték tömb, minden a kulcs tömbben található kulcsra, pointereket tartalmaz, ami a megfelelő lista elejére mutat, melyet feljebb leírtunk. Ezt a hashtáblát példány szomszéd hashtáblának nevezzük.

A csoport minták bányászása közben az algoritmus eltárolja a minták (vagy minta jelöltek) példányait lista formájában. Hogy felgyorsítsuk egy adott minta példányainak kikeresését, egy másik hashtáblát használunk. Ez a struktúra sokkal bonyolultabb az iCPI-fa reprezentálása, és az alábbi módon valósítjuk meg:

A tulajdonságok számosságától függően, minden mintát 32 bites szavak sorozatával írunk le, ahol minden bit egy tulajdonságnak felel meg. A minta “szavak” számát BL-lel jelöljük. Minden mintát egyetlen memória blokkban tárolunk, mely BL-szer a minták száma méretű. Az együtt álló minták bányászása közbeni elérések miatt, a minták nem tudnak egybefüggően helyet foglalni a memóriában. Ehelyett az egymást követő szavak a memóriában, minden minta első szavai, aztán a második szavak, és így tovább. A mintákat lexikografikus sorrendben tároljuk. Ezt a struktúrát minta puffernek nevezzük.

A minta példányokat hasonlóan tároljuk egy külön álló memória blokkban. Minden minta példányt 32 bites szavak sorozatával írunk le, ahol minden szó egy tulajdonság példány, ami a feljebb leírt módon néz ki. Következés képpen ennek a memória blokknak a mérete megegyezik az összes minta példány száma szorozva a minta hosszával. A mintákhoz hasonlóan, a memóriában egymást követő részek a minta példányok első, majd a második, és így tovább, szavait tárolják. Egy minta példányai szomszédos helyeket foglalnak el a memóriában, tehát a minta példányok nem folynak egybe.

Mintánként a minta példányok lexikografikus sorrendben vannak. Ezt a struktúrát minta példány puffernek hívjuk.

Egy hashtábla, amit egy minta példányainak keresésére használunk, két tömbből áll: kulcs és érték tömbből. A kulcs tömbben pointereket tárolunk, ami a megfelelő minta kezdő pozíciójára mutat a minták memória blokkjában, vagy null ha üres. Az érték tömb minden kulcshoz egy rekordot tárol, ami tartalmazza a minta példányok számát, a minta gyakoriságát és egy pointert az első minta példány első szavára a minta példányok memória blokkjában. Nyitott címzési sémát használunk a hasításra. Ezt a minta hashtáblának nevezzük.

Észrevehetjük, hogy a minta hashtábla kulcs tömbje tárolhatná akár a mintákat is, ahelyett, hogy mintákra mutató pointereket tárol. A jelenlegi megoldást az atomi beszúrások miatt használjuk. Általában a [2]-ben bemutatott megoldást követjük. Ezt a megoldást az “összehasonlítás és csere” sémára alapozzuk (sajnos az atomicCAS CUDA függvény csak 32 vagy 64 bites szavakon képes végre hajtani az “összehasonlítás és csere”-t).

4.2 A bányászat indítása

A bányászat a kezdeti iCPI-fa olvasásával kezdődik, és felépítjük az iCPI-fa hashtábláját. Ezután a példány szomszéd puffer elkészítése következik. További adatok mint például a tulajdonságok példányainak száma is ekkor számítódik. Sőt, egy ideiglenes tömbben eltároljuk az összes példány szomszéd hashtáblájának kulcsait és értékeit (pointerek listákra). Végül az összes ilyen kulcs, érték párt párhuzamosan beillesztjük a példány szomszéd hashtáblába.

A példány szomszéd puffer és hashtábla felépítése után a minta puffer, a minta példány puffer és a minta hashtábla készül el 1 hosszú mintákra. Mivel párhuzamos algoritmust írtunk ezekre a lépésekre, eltekintünk a részletektől, a feldolgozási ideje ennek a lépésnek elhanyagolható.

4.3 Jelöltek generálása - minta illesztés

Az n méretű jelöltek generálása felhasználja az n-1 méretű minták pufferét. Mivel a minta puffert lexikografikus sorrendben tároljuk, minden n-1 méretű minta, amit beilleszthetünk egy n méretű mintába, sok egymást követő, illeszthető minták együtteseit alkotják. Ezeket az együtteseket illeszkedési csoportnak nevezzük. Az algoritmus utolsó lépésében az illeszkedési csoportokat eredmény csoportokká konvertáljuk. Egy eredmény csoport több egymást követő minta csoportja, melyet egy illeszkedési csoporton belüli összes minta illesztésével kapunk. A csoportok sorrendjét megtartja, tehát adott A és B illeszkedési csoport, ha A megelőzi B-t az input minta pufferen, akkor a megfelelő eredmény csoportok is így követik egymást az output minta pufferen (kivéve ha az illeszkedési csoport csak egy mintát tartalmaz, mert ekkor ez nem hoz létre illeszkedési eredményt). Ha mintákat az eredmény csoporton belül lexikografikusan sorba rendezzük, akkor ez a tulajdonság biztosítja az eredmény minta puffer teljes rendezettségét.

A minta illesztési algoritmus sok egymást követő lépésből áll, de az egyes lépések párhuzamos műveleteket tartalmazhatnak. Az első lépésben azonosítani kell az illeszkedési csoportokat, meg kell találni a számukat (legyen k_{JG}), ki kell számítani a méretüket és a hozzájuk tartozó eredmény csoport méretét (kettes alapú hatványok). Sok fontos tömb jön létre:

- *groupSizes*: egy k_{JG} hosszú tömb, ami az illeszkedési csoportok méreteit tárolja.

- *joinCounts*: egy k_{JG} hosszú tömb, ami az illeszkedési csoportokhoz tartozó eredmény csoportok méreteit tárolja.
- *positions*: egy k_{JG} hosszú tömb, ami minden illeszkedési csoporthoz számon tartja az utolsó mintája input minta pufferbeli indexét.
- *scannedJoinCounts*: egy k_{JG} hosszú tömb, ami a joinCounts tömbön végre hajtott külső vizsgálat eredményét tárolja.

A következő lépésekben egy további segéd tömböt, *scannedJoinFlags*, számolunk, aminek a hossza az eredmény minták számával egyezik meg (k_P). A kapott tömbben, minden mintára egy eredmény csoportból, tároljuk a hozzátartozó illeszkedési csoport referencia számát.

Az utolsó lépésben alakul ki a végső minta illesztés. Minden illeszkedési csoportot eredmény csoporttá alakítunk. Először létre hozunk egy k_P× BL méretű minta puffert (BL a minta kódolására használt 32 bites szavak száma). Ez után $k_P$szálat indítunk. Minden szál alapja a *scannedJoinFlags* tömb, ami meghatározza a megfelelő illeszkedési csoportok referencia számát. Ezen értékek segítségével a szál meg tudja határozni a *scannedJoinCounts* megfelelő pozíciójáról, hogy a hozzátartozó eredmény csoport hol kell kezdődjön az eredmény minta pufferben. A szál kiszámolja a globális szám és a kapott pozíció különbségét a hozzátartozó eredmény csoporton belüli elhelyezkedés, pos, meghatározásához. Ezt a számot aztán ketté bontjuk a megfelelő illeszkedési blokkon belül két sorozat számaivá, az alábbi formulával: $p_1 = bs - 1 - \lceil 0{,}5\left(\sqrt{8(jc - 1 - \mathrm{pos}) + 9} - 1\right)\rceil$ és $p_2 = pos - 0{,}5p_1(2bs - p_1 - 3) + 1$, ahol *bs* a megfelelő illeszkedési csoport mérete a groupSizes tömbből, *jc* pedig a megfelelő eredmény csoport mérete a joinCounts tömbből.

A képleteknek két feladata van: (1) egy szál pozícióját a hozzátartozó eredmény csoporton belül, szétbontja két minta kombinációjára a megfelelő illeszkedési csoportból, (2) az illesztett mintákat lexikografikusan tárolja az eredmény csoportban. A p_1 és p_2 pozíciókat átszámítjuk globális pozíciókká az input minta pufferen, úgy, hogy hozzájuk adjuk a megfelelő számot a positions tömbből (+1). Végül a szál összeilleszti a két mintát bitenkénti VAGY művelettel a minták minden egyes szavára és az eredmény eltárolja az eredmény minta pufferben.

4.4 Jelöltek generálása - Jelöltek metszése

Minden illeszkedéssel kapott mintát meg kell vizsgálni, hogy minden részmintájuk gyakori-e vagy sem. Ehhez az n hosszú minta vizsgálathoz felhasználjuk az előző algoritmusban kapott minta puffert és az n-1 hosszú gyakori minták minta hashtábláját. Tegyük fel, hogy k_P mintát kaptunk a minták illesztésével. Először foglalunk k_P méretű helyet a memóriában egy flags tömbnek, aztán indítunk ugyanennyi szálat. Minden szál kiveszi a neki megfelelő mintát az input minta pufferből és egymásután legenerálja az n darab n-1 hosszú részmintát belőle. Minden részmintát ellenőrzünk, hogy létezik-e minta hashtáblában. Ha egy minta minden részmintája megtalálható a minta hashtáblában, akkor a szál egy 1-est ír a flags tömb megfelelő pozíciójára, egyébként pedig 0-t.

A párhuzamos tömörítő algoritmus második lépéseként eltávolítjuk az összes olyan mintát, aminek megfelelő helyen a flags tömbben nem 1 szerepel. Mivel a párhuzamos tömörítő nem változtatja meg a minták sorrendjét, a lexikografikus sorrend megmarad. Az így kapott minta puffer lesz a minta jelölt puffer.

4.5 Példányok generálása

Most bemutatunk egy algoritmust, amely a legidőigényesebb lépése az egybefüggő minták felismerésének - a példányok generálása. Bevezetünk fontos elnevezéseket: legyen a jelölt minták száma k_C. Bármely n-1 hosszú mintát, amely prefixe az n hosszú jelöltnek, jelölt prefix mintának nevezünk. Ennek példányait jelölt prefix minta példánynak nevezzük. A minta jelölt utolsó tulajdonságát bővítő tulajdonságnak nevezzük.

Az algoritmus alapötlete a következő megfigyelésre támaszkodik. Az alap iCPI-fa algoritmusban egy C jelölt f_e bővítő tulajdonsággal a következő képpen található meg: (1) kigyűjtjük az összes P_i jelölt prefix példányt, (2) minden Pi jelöltre keressük az f_e tulajdonságú szomszédjainak listáját az iCPI-fából, (3) közös elemek keresése ezeken a listákon. Mivel a jelölt prefix példányok feldolgozása egymástól független, ezért ez párhuzamosítható. Az algoritmusunk minden jelölt minta prefix példányát párhuzamosan dolgozza fel. Legyen k_I a feldolgozott prefix példányok száma. Vegyük észre, hogy egy jelölt prefix példányt többször is feldolgozhatunk, ha az több jelöltnek (különböző bővítő tulajdonsággal) is prefixe. Minden jelölt prefix példány két számmal (egy lokálissal és egy globálissal) rendelkezhet. A lokális jelölt prefix példány szám, egyetlen jelölt minta jelölt prefix példányainak csoportján belüli számát jelöli. A globális szám, az összes jelölt prefix példány között elfoglalt számát jelenti. Bemenetként az algoritmus kap egy, az előző lépesben megkapott, jelölt minta puffert, amely n hosszú mintákat tartalmaz, egy minta puffert, egy minta példány puffert, egy minta hashtáblát a gyakori n-1 hosszú mintákkal illetve egy példány szomszéd puffert és egy példány szomszéd hashtáblát. A példány generáló algoritmusnak szüksége van segéd tömbökre, amiket szintén tudunk párhuzamosan számítani:

- *listPointers*: egy k_C hosszú tömb, ami minden jelölt mintához egy pointert tárol, ami az első jelölt prefix példányra mutat a minta példány pufferben.
- *instanceCounts*: egy k_C hosszú tömb, ami minden jelölt mintához tárolja a jelölt prefix példányok számát (mennyiség).
- *scannedInstanceCounts*: egy párhuzamos belső vizsgálat eredménye az instanceCounts tömbön.
- *correspondingPatterns* és *extendingFeatures*: egy k_I hosszú tömb, ami melyek összekapcsolják a globális jelölt prefix példány számot a megfelelő jelölt minta számával és a bővítő tulajdonságával, mindkét tömb rendezve van a hozzátartozó minta szerint.

Az első lépésben minden jelölt prefix példányhoz legeneráljunk a közös f_e tulajdonságú példányok listáját. Két tömböt hozunk létre: (1) listSizes k_I hosszú tömböt, amely a szomszéd listák hosszait tárolja, (2) newNeighbours k_I× L hosszú tömböt (L a szomszéd lista hossza a példány szomszéd pufferben), amiben a szomszéd listákat tároljuk. A newNeighbours tömb memóriabeli elhelyezkedése és struktúrája megegyezik a példány szomszéd pufferével. Ehhez a lépéshez k_I× L szálat indítunk. A megfelelő L szál minden csoportja együtt működik egyetlen szomszéd lista legenerálásához a newNeighbours tömbbe. Minden szál az indulásakor meghatározza (1) a jelölt prefix példány globális számát $c \in 0, \dots, k_I - 1$ (ugyanazt minden L egymást követő szálra), (2) a megfelelő pozicíót a szomszéd listában $l \in 0, \dots, L - 1$, (3) a megfelelő jelölt minta számát p, a correspondingPattern tömbből (4) és a bővítő tulajdonságot fe, az extendingFeatures tömbből. Ezután c alapján minden szál meghatározza a lokális jelölt prefix példány számot. Ezt úgy kapjuk, hogy c-ből kivonjuk a scannedInstanceCount[p-1]-et. Ha p = 0, akkor ez a szám a c lesz. Továbbá minden szál lekérdezi a neki megfelelő jelölt mintához, p, tartozó jelölt prefix példányok első elemére mutató pointert a

listPointers tömbből. A lokális jelölt prefix példány szám és a pointer alapján minden szál kiszámítja a feldolgozandó jelölt prefix példány első tulajdonság példány címét. Minden L szál szinkronban keresi az f_e tulajdonságú szomszédok listáinak metszetét, minden olyan tulajdonság példányra, amely része a feldolgozott jelölt prefix példánynak. Az egész eljárás majdnem csak a gyors megosztott memóriát használja a GPU-n. Amikor a szálak végeznek a számításokkal az eredményeiket bemásolják a newNeighbours tömbbe és a kapott listák hosszát a listSizes tömbbe.

Miután megtaláltuk a szomszéd listákat, az algoritmus egy módosított tömörítést végez, amely eltávolítja az üres elemeket a kapott listából, de a megmaradt elemekből előáll a teljes jelölt minta példány, tehát a megfelelő jelölt prefix példány kibővítve a megfelelő elemmel. Az eredményül kapott tömb struktúrája és tulajdonságai megegyeznek a minta példány pufferével.

4.6 A gyakoriság számítása

A gyakoriság számításának alapjául több klasszikus párhuzamos algoritmus szolgál, mint például a rendezés, kulcs alapján történő redukció és az egyedi algoritmus (lásd 2. bekezdés). Az eljárás egy érték kiszámításával kezdődik, minden egyes jelölt minta példányhoz tartozó tulajdonság példányra, amely a jelölt példányon belüli pozíciójából, a jelölt minta számából és a példány azonosítóból áll. Ezen értékeket tartalmazó tömböt ezután lexikografikusan rendezzük. Következő lépésként a nem egyedi értékeket távolítjuk el a tömbből az egyedi algoritmus segítségével. Az így kapott tömb minden jelölt mintához minden példányának pozícióján egy egyedi, tulajdonság példányaiból álló listát tartalmaz. Ezután a kapott tömb minden értékből párhuzamosan eltávolítjuk a tulajdonság példány számot (bár a tulajdonság példány azonosítója megmarad). Az így kapott tömbön alkalmazzuk a kulcs alapján történő redukciót, hogy megszámoljuk az eltérő értékeket (a tömböt kulcs tömbnek tekintjük és az értékek 1-esek). A kapott eredmény minden jelölt minta minden egyes pozíciójához eltárolja az egyedi tulajdonság példányok számát, amelyek megjelenek a jelölt minta példányokban. Ezen értékekre alapozva, kiszámítjuk minden jelölt minta részvételi rátáját és gyakoriságát (párhuzamosan).

A nem gyakori jelölt mintákat és a hozzájuk tartozó példányokat eltávolítjuk egy tömörítő algoritmus segítségével. Csak a gyakori minták és a hozzájuk tartozó példányok maradnak (a minta pufferben és a minta példány pufferben). Végül egy új minta hashtáblát építünk (párhuzamosan). Ezek a struktúrák alkotják az együtt álló mintákat felfedező algoritmus következő iterációjának bemenetét.

5. Eredmények

5.1 Megvalósítás és Tesztelési környezetek

A kutatás céljából két megvalósítása készült el az iCPI-fa alapú algoritmusnak: egy GPU-ra és egy CPU-ra. A megoldások GPU-s verzióját a 4. pont írja le részletesen. A CPU verzió a 4.1 pontban leírt adatstruktúrákhoz hasonlóakat használ, de a számítások párhuzamosítása másképp valósul meg. Az SIMD megközelítés helyett, a “többféle utasítás többféle adatra” (multiple instructions multiple data - MIMD) megközelítést alkalmazták. A CPU-ra írt megvalósítás az OpenMP-t [6] használja a párhuzamosításhoz a példány generációnál és a gyakoriság kiszámításánál, több magos CPU-val. Az implementáció annyi szálat indít, ahány magos a CPU és a számítási feladatokat elosztja az indított szálak között. Az L paraméter (a példány szomszéd pufferbeli listák hossza) mindkét megoldásnál 8-ra lett beállítva.

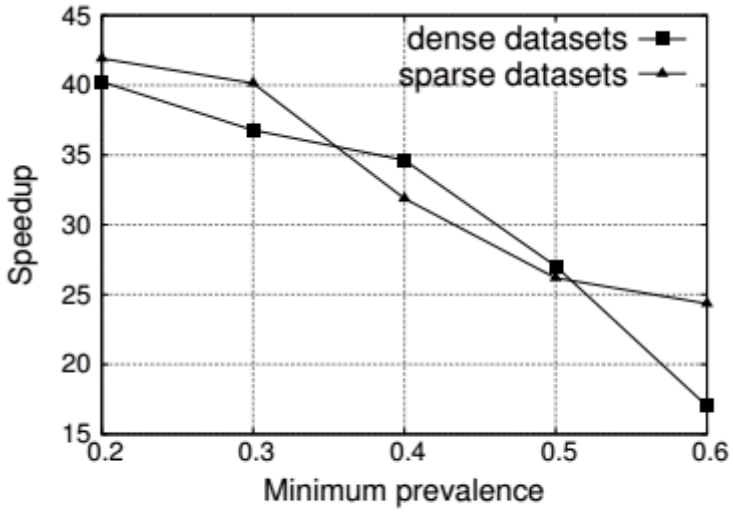
A kísérleteket Core2 Duo 2,1 Ghz-es CPU-val 8GB RAM-mal (CPU implementáció két szállal) és GeForce 580GTX grafikus kártyával 1,5GB RAM-mal rendelkező gépen, Windows 7 operációs rendszer alatt futtatták.

5.2 Adathalmazok és kísérletek

Hogy felbecsülhessék a GPU-ra adott megoldást, 10 mesterséges adathalmazt hoztak létre a teszteléshez. A [15]-ben bemutatott mesterséges generátorhoz hasonlóval készültek ezek a halmazok. Az így kapott objektumok száma 25 ezer és 120 ezer közötti, a térbeli tulajdonságok száma 30-90, és 20-80 százaléka a teljes példány állománynak zajos volt. Két féle adathalmazt használtak: tömöret és ritkát. A tömör adathalmazokat a térbeli szerkezet minden dimenziója mentén való tízszeres csökkentéssel érték el, míg az objektumok számát megtartották (a ritka adathalmazt egy 10000×10000 hálóra generálták).

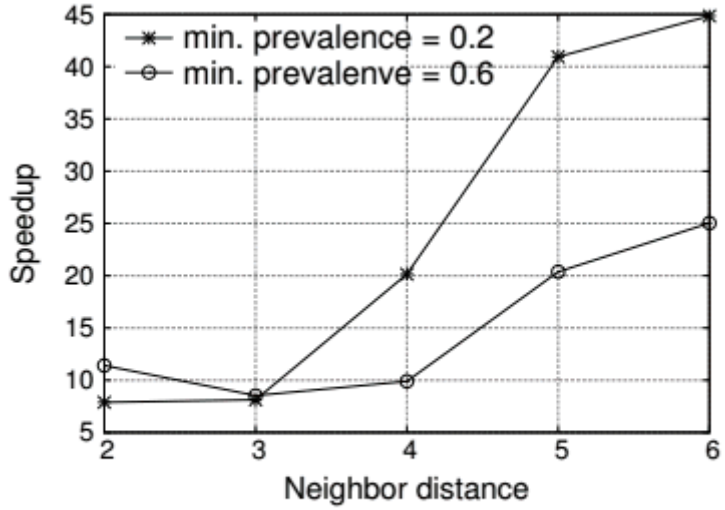
3 kísérlet sorozatot vezettek le. Minden alkalommal mérték a gyorsulást, tehát a CPU és a GPU implementáció végrehajtási idejének a rátáját. Az első sorozatban azt vizsgálták, hogy a minimum gyakorisági küszöb növelésével a gyorsulás miként változik a tömör illetve a ritka adathalmazokon. A második sorozatban a szomszédsági kapcsolatok távolsági küszöbének befolyását vizsgálták két különböző szintű minimum gyakorisági értékre. Végül megnézték miként változik a gyorsulás a bemeneti adathalmaz méretének növelésével.

5.3 Az eredmények értékelése



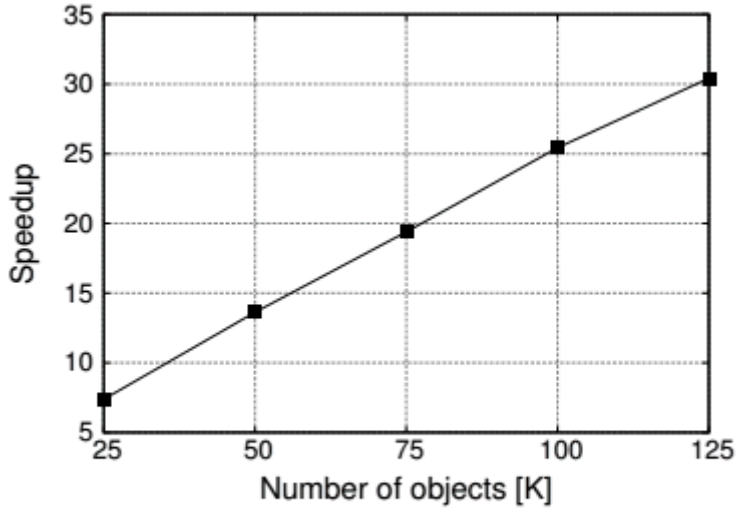
(a)

A *2(a) ábrán* láthatjuk az első kísérlet eredményét, amely a minimum gyakorisági paraméter befolyását vizsgálta. Két érdekes dolgot lehet észrevenni. Először is az adathalmaz sűrűsége nem befolyásolja a gyorsulást, tehát míg a futási idő ugyan változik, a GPU verzió arányaiban körülbelül ugyanannyival gyorsabb a CPU verziónál minden sűrűség esetében ugyanarra a minimum gyakorisági küszöbre. Másodszor, a gyorsulás monoton csökken a minimum gyakoriság növelésével. A kialakított teszt környezetben a 0,2-re állított minimum gyakoriságra kapott gyorsulás majdnem kétszerese a 0,6-ra állított futtatás gyorsulásának. Ez azért van mert a megnövelt gyakorisági küszöbre kevesebb jelölt mintát kapunk és ezzel kevesebb jelölt prefix példányt dolgoz fel az algoritmus. Ez azt jelenti, hogy kevesebb szál indul. A kevesebb szál annyit tesz, mint (1) a memória átvitel nem lesz elfedhető, (2) a GPU multiprocesszora nem biztos, hogy teljes mértében ki lesz használva, (3) a példány generálás lépése kevesebb időbe kerül mint az algoritmus más lépései (mivel korábban az elsődleges szempont ennek a lépésnek az optimalizálása volt, ez okozhatja a megfigyelt gyorsulás csökkenést).



(b)

A *2(b) ábrán* láthatjuk a második kísérlet eredményét, amely a szomszédsági kapcsolatok távolsági küszöbének befolyását tesztelte. Észrevehetjük, hogy minél nagyobb a távolsági küszöb annál nagyobb a gyorsulás. Ennek a jelenségnek a magyarázata hasonló az előző kísérletben tett magyarázathoz. Nagyobb távolsági küszöb esetén minden tulajdonság példánynak több szomszédja lesz. Ennek következménye, hogy a szomszéd listák a példány szomszéd pufferben hosszabbak lesznek és több példány jön létre a példány generálás során. Ahogy azt az előző bekezdésben bemutattuk, minél több példányunk van, annál nagyobb lesz a gyorsulás. Az ábrán látszik, hogy két minimum gyakoriságra végezték el a kísérletet: 0,2-re és 0,6-ra. A hozzájuk tartozó grafikon megerősíti a korábbi megfigyelést, miszerint ha a minimum gyakoriság kisebb, akkor a gyorsulás nagyobb. Sőt, az előzőekhez képest, a gyorsulás itt is kétszeres lesz a két küszöb között.



(c)

A *2(c) ábrán* láthatjuk a harmadik kísérlet eredményeit, amely a tulajdonság példányok (objektumok) számának befolyását vizsgálta a gyorsulásra. Meglepő, hogy a gyorsulás lineárisan nő,

ha az objektumok számát növeljük. Természetesen nyilvánvaló, hogy a gyorsulás nem növekedhet a végtelenbe. Ugyanez a görbe még több példányra aszimptotikusan közelít a maximálisan elérhető sebesség gyorsuláshoz, ami függ a GPU és a CPU relatív teljesítményétől. A monotonitás a korábbiakhoz hasonlóan magyarázható. A több objektum több minta példányhoz vezet és így több jelölt prefix példányhoz. Minél nagyobb az ilyen példányok száma, annál több szálat indít az algoritmus, ezzel jobban kihasználva a GPU által biztosított lehetőségeket.

Sajnos a limitált GPU memória miatt a kísérletek nem tudták elérni a maximális gyorsulást. Az algoritmus memória igénye függ az adatok és a lekérdezés tulajdonságaitól, mint például az objektumok számától, a térbeli tulajdonságok számától, az adatok sűrűségétől és a minimum gyakoriságtól. A 1,5GB memóriájú GPU 120 ezer objektumot tartalmazó tömör adathalmazra, 90 térbeli tulajdonsággal és 0,2-es minimum gyakorisághoz volt elegendő.

6. Összefoglalás és további lehetőségek

Az eddigiekben bemutatott algoritmus korszerű műveleteket alkalmaz az együtt álló minták felfedezésére párhuzamosítva a GPU-n. Összehasonlítva ezt a megoldást a több szálas, CPU feldolgozású implementációval, láthatjuk, hogy a GPU-val nagyságrendekkel jobb gyorsulás érhető el, mint a CPU-s verzióval.

Bár az eredmények nagyon bíztatóak, még sok munka van hátra. A fő probléma a grafikus kártya limitált memóriája. Ez a probléma kétféle képpen kezelhető. Cikk készítőinek első terve módosítani az algoritmust, hogy változó hosszú szomszéd listákkal tudjon számolni. A jelenlegi megoldás pazarolja mind a memóriát és mind a számítási kapacitást (sok szál üres helyeket számol ezeken a listákon). Második tervük megvalósítani az [5] cikkben bemutatott megoldásokat, melyek a korlátozott memóriával való együtt álló minták bányászására vannak kiélezve. Továbbá tervezik egy hatékonyabb algoritmus megalkotását az iCPI-fa felépítésére GPU-n (jelenleg a faépítése a CPU-n zajlik). Végül, bonyolultabb problémák megoldását fontolgatják a maximális együtt álló minták felfedezésére a GPU segítségével.

Hivatkozások

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceeding of the 20th International Conference on Very Large Databases. pp. 487-499. Morgan Kaufmann Publishers Inc., San Fracisco (1994)
2. Alcantara, D.A.F.: Efficient Hash Tables on the GPU. Ph.D. thesis, University of California, Davis (2011)
3. Bell, N., Hoberock, J.: GPU Computing Gems: Jade edition, chap. Thrust: A Productivity-oriented Library for CUDA, pp. 359-371. Morgan-Kaufmann (2011)
4. Boinski, P., Zakrzewicz, M.: Collocation Pattern Mining in Limited Memory Environment Using Materialized iCPI-Tree. In: Cuzzocrea, A., Dayal, U. (eds.) Datawarehousing and Knowledge Discovery, Lecture Notes in Computer Science, vol.7448, pp. 279-290. Springer Berlin Heidelberg (2012)
5. Boinski, P., Zakrzewicz, M.: Partitioning Approach to Collocation Pattern Mining in Limited Memory Environment Using Materialized iCPI-Tree. In: Morzy, T., Haerder, T., Wrembel, R. (eds.) Advances in Databases and Informataion Systems, Advances in Intelligent Systems and Computing, vol. 186, pp. 19-30. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-32741-4_3
6. Chapman, B., Jost, G., Pas, R.v.d.: Using OpenMP: Portable Shared Memory Parallel Programing (Scinetific and Engineering Computation). The MIT Press (2007)
7. Farooqui, N., Kerr, A., Diamos, G., Yalamanchili, S., Schwan, K.: A Framework for Dynamically Instrumenting GPU Compute Applications within GPU Ocelot. In: Proceeding of the 4th Workshop on General Processing on Graphics Processing Unit. pp. 9:1-9:9. GPGPU-4, ACM, New York, NY, USA (2011)
8. Fayyad, U., Piatetsky-Shapiro, G., Smyth, P.: From Data Mining to Knowledge Discovery in Databases. AI Magazine 17, 37-54 (1996)
9. Khronos group: The OpenCL Specification Version 1.2 (2012)
10. NVIDIA Corporation: Nvidia CUDA programming guide (2012)
11. Shekhar, S., Huang, Y.: Discovering Spatial Co-location Patterns: A Summary of Results. In: SSTD 2001. pp. 236-256 (2001)
12. Shekhar, S., Huang, Y.: The multi-resolution co-location miner: A new algorithm to find co-location patterns in spatial dataset. Tech. Rep. 02-019, University of Minesota (2002)
13. Wang, L., Bao, Y., Lu, J.: Efficient Discovery of Spatial Co-Location Patterns Using the iCPI-Tree. The Open Information System Journal 3(2), 69-80 (2009)
14. Xiao, X., Xie, X., Luo, Q., Ma, W.Y.: Density Based Co-Location Pattern Discovery. In: Proceeding of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems. pp. 29:1-29:10. GIS '08, ACM, New York, NY, USA (2008), http://dl.acm.org/citation.cfm?doid=1463434.1463471
15. Yoo, J.S., Shekhar, S., Celik, M.: A Join-Less Approach for Co-Location Pattern Mining: A Summary of Results. In: Proceedings of the IEEE International Conference on Data Mining. pp. 813-816. Washington (2005)
16. Yoo, J., Bow, M.: Mining Maximal Co-location Event Sets. In: Huang, J., Cao, L., Srivastava, J. (eds.) Advances in Knowledge Discovery and Data Mining, Lecture Notes in Computer Science, vol. 6643, pp. 351-362. Springer Berlin Heidelberg (2011), http://link.springer.com/chapter/10.1007%2F978-3-642-20841-6_29