UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: Calculatoare și tehnologia informației
SPECIALIZAREA: Tehnologia informației

Algoritmi pentru descoperirea pattern-urilor frecvente în baze de date de mari dimensiuni – Implementări GPGPU

LUCRARE DE DIPLOMĂ

Coordonator științific
Prof. dr. Mitică Craus

Student
Ungureanu Ionuț-Leonaș

Iași, 2019

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII
LUCRĂRII DE DIPLOMĂ

Subsemnatul(a) _____,
legitimat(ă) cu _____ seria _____ nr. _____, CNP _____
autorul lucrării _____
_____
_____
elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iași, sesiunea _____ a anului universitar _____, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data Semnătura

_____

Cuprins

Algoritmi pentru descoperirea pattern-urilor frecvente în baze de date de mari dimensiuni – Implementări GPGPU

Ungureanu Ionuț-Leonaș

Rezumat

În contextul în care, datele digitale sunt pretutindeni și necesitatea prelucrării lor este la ordinea zilei, ne propunem să reducem timpul necesar procesului de obținere a rezultatelor, având în vedere cantitatea enormă de informații din bazele de date la momentul actual.

Data mining este domeniul analizei colecțiilor mari de date, cu scopul descoperirii de cunoștințe noi, unde determinarea pattern-urilor frecvente reprezintă un pas esențial, fără de care nu se pot descoperi cunoștințe ulterioare. Procesul de determinare a pattern-urilor frecvente poate fi costisitor, din punct de vedere al timpului necesar execuției, considerând volumul mare de date stocat în bazele de date și debitul cu care se acumulează, zilnic, noi date. Odată cu creșterea în dimensiune a bazelor de date, crește și timpul necesar procesării acestora, și astfel ne propunem, să cercetăm cum pot fi influențați timpii de execuție dacă implicăm procesorul grafic în procesarea bazei de date.

În procesul de cercetare, sunt paralelizați, la nivel de GPU, algoritmii:

- Apriori – propus de Agrawal și Srikant;
- Fast Item Miner – propus de prof. dr. Mitică Craus.

Algoritmii menționați sunt paralelizați conform framework-ului OpenCL, fiind aplicate procedee de optimizare, astfel încât să se ajungă la rezultate satisfăcătoare, din punct de vedere al timpului.

În urma testării implementării algoritmilor, conform necesităților OpenCL, se observă, prin comparația, în contrast, a rezultatelor obținute pentru algoritmul Apriori implementat secvențial cu rezultatele obținute în urma implementărilor GPGPU, performanța considerabilă obținută.

Introducere

Datele digitale sunt la ordinea zilei, întâlnindu-le la fiecare pas, de la telefonul mobil, chioșcul de după bloc, la marile bănci și corporații, acestea colectând zi de zi informațiile utilizatorilor, mașinilor, printr-o multitudine de aplicații și senzori. Microsoft, Google, aplicațiile Web, dispozitivele mobile, colectează date prin interacțiunile cu utilizatorii, și diferiți senzori. Se estimează, că în fiecare zi sunt create și colectate date de dimensiuni aproximative cu 2,5 quintilioane de octeți. În contextul acestui debit, cu care sunt acumulate datele, este de așteptat ca și dimensiunea bazelor de date să crească zilnic, în ritm alert, a căror parcurgere/procesare devine anevoioasă, greoaie, ce implică intervale de timp consistente.

Toate aceste date, colectate, trebuie prelucrate pentru aflarea unor noi cunoștințe pe baza cărora se pun în practică anumite acțiuni. Astfel, procesul de data mining este necesar pentru determinarea unor pattern-uri, reguli, cunoștințe, pe baza cărora se implementează reguli de afaceri, menite să sporească câștigul unității economice (magazin, bancă, etc.). În procesul de data mining, pasul de determinare a pattern-urilor, care stă la baza descoperirii regulilor de asociere, poate fi costisitor din punct de vedere al timpului necesar obținerii rezultatelor. Pentru baze de date de dimensiuni foarte mari ne așteptăm la un cost, din punct de vedere al timpului, pe măsură, direct proporțional. Pentru algoritmii abordați de lucrarea în consecință, o tratare într-o manieră secvențială se poate dovedi o greșeală, având în vedere numărul mare de scanări la care este supus o bază de date mare, care poate veni totodată și cu multe pattern-uri. De aceea se propune tratarea algoritmilor într-o manieră paralelă pentru a obține timpi de execuție mai mici, deși numărul de scanări a bazei de date este același. Scanând baza de date de același număr de ori, cu cât mai multe scanări pot fi realizate în același timp cu atât mai mic va deveni și timpul de execuție. Considerând acest fapt, ne propunem să realizând o paralelizare cât mai mare, dar CPU-ul platformei de test poate trata până la 8 scanări la un moment dat. Astfel ne orientăm atenția către procesorul grafic (GPU), pentru care știm că numărul de nuclee îl depășește cu mult pe cel al CPU-ului, 1024 pentru platforma de test.

Lucrarea de față își propune să prezinte implementările GPGPU a algoritmilor propuși și concluziile care au fost obținute în urma testelor și comparațiilor rezultatelor.

Fundamentarea teoretică introduce cititorul în necesarul de cunoștințe necesar pentru a avea o idee ce își propun implementările GPGPU să realizeze. Este realizată o introducere generală în „data mining” și pașii standard parcurși de proces, după care sunt introduși algoritmii abordați din punct de vedere teoretic.

Proiectarea prezintă tehnologiile software și hardware-ul platformei de lucru, utilizate în realizarea cercetării și testării implementărilor GPGPU. Cititorul este introdus succint în tehnologia OpenCL de programare a procesorului grafic, tehnologie pe baza căreia au fost realizate implementările.

Implementarea este capitolul efectiv unde sunt prezentate implementările GPGPU a algoritmilor Apriori și Fast Item Miner și modul de abordare, pentru fiecare în parte, ce constituie contribuția autorului. Sunt puse în evidență procedee de optimizare și limitări, menite să îndeplinească scopul cercetării, acela fiind obținerea unor timpi de execuție mici.

Rezultate experimentale este capitolul în care sunt prezentate rezultatele obținute în urma testelor implementărilor GPGPU, comparate cu rezultatele algoritmului Apriori implementat secvențial.

Capitolul 1. Fundamentarea teoretică

1.1. Data mining

Data mining este domeniul analizei colecțiilor mari de date în scopul descoperirii de cunoștințe noi, cunoștințe folosite pentru a determina anumite concluzii și/sau predicții. Conceptul de data mining apare oriunde sunt implicate informațiile digitale, stocate într-o bază de date, indiferent de domeniul de aplicabilitate. Câteva domenii în care este implicat data mining-ul sunt, după [1]:

- Marketing;
- Managementul riscului de credit;
- Detecția fraudei;
- Medicină;
- Filtrarea spam-ului;
- Analiza sentimentelor;
- Elaborarea de date calitative.

Figura 1.1. Aplicații data mining.

Procesul de data mining implică următorii pași:

- Înțelegerea domeniului de aplicabilitate (afaceri) – stabilirea obiectivelor și cum va ajuta data mining-ul la atingerea acestora;
- Înțelegerea datelor – datele sunt colectate de la toate sursele disponibile (baze de date, fișiere, etc.);
- Pregătirea datelor – datele sunt prelucrate, pregătite, pentru procesul de minare;
- Modelarea datelor – modele matematice sunt folosite pentru a găsi pattern-uri în colecțiile de date;
- Evaluarea – rezultatele obținute la pasul anterior sunt evaluate în comparație cu obiectivele domeniului de aplicabilitate pentru a determina importanța lor;
- Implementarea – rezultatele obținute (concluziile, predicțiile) sunt utilizate, implementate, în viața de zi cu zi.

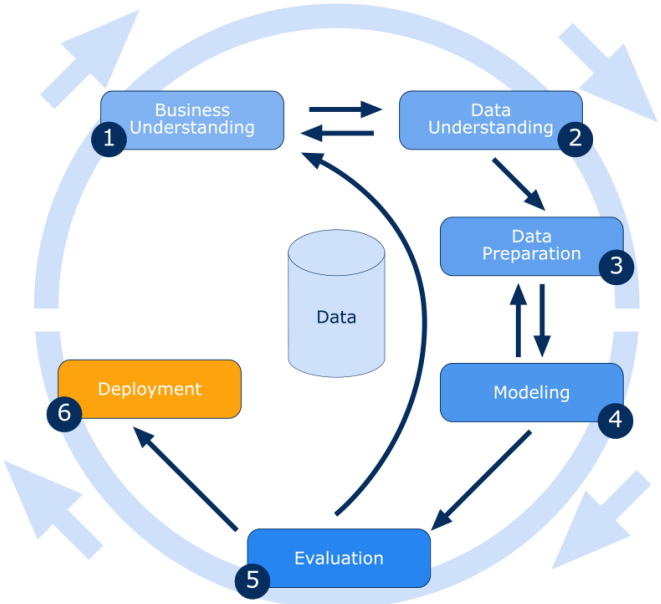

Figura 1.2. pașii acceptați în data mining.

Beneficiile analizei colecțiilor mari de date sunt în număr mare, implicând: luarea de decizii automate – prin analiza continuă a datelor, băncile pot detecta tranzacțiile frauduloase; predicții – magazinele implementează metode de prezicere a cererii pentru produse; reducerea de costuri; informații despre clienți – Disney a implementat așa numitele brățări magice ce au ca scop îmbunătățirea experienței și colectarea datelor activităților clienților.

Informații digitale sunt colectate zi de zi de companiile de pretutindeni, și cu trecerea timpului datele devin din ce în ce mai sofisticate, având drept urmare îngreunarea procesului de data mining. Acest efect se reflectă, îndeosebi, asupra descoperirii de pattern-uri frecvente, proces abordat de lucrarea în consecință.

Procesul de descoperire a pattern-urilor frecvente într-o colecție mare de date este esența data mining-ului. Deși data mining-ul nu constă doar în descoperirea pattern-urilor frecvente, acest proces este pasul de bază în orice operație a data mining-ului, nelipsit, constând în descoperirea itemset-urilor ce apar în colecțiile de date, itemset-uri ce apar într-un număr minim de tranzacții predefinit. Cu alte cuvinte, un pattern frecvent este un pattern (itemset) ce apare frecvent într-o colecție.

Suportul este o măsură a frecvenței absolute, reprezentând partea fracționară (procentul) de tranzacții în care apare un itemset din totalul de tranzacții conținute de o colecție. Un itemset este frecvent dacă suportul lui în colecția dată nu este sub o valoare predefinită, numită și suport minim.

Datorită dimensiunilor mari a colecțiilor de analizat,procesul descoperirii pattern-urilor frecvente poate necesita un interval mare de timp dacă acesta este realizat într-o manieră secvențială. De aceea se obișnuiește realizarea acestui proces într-o manieră paralelă și distribuirea sarcinilor între mai multe mașini de calcul pentru reducerea intervalului necesar obținerii pattern-urilor frecvente.

1.2. Algoritmul Apriori

1.2.1. Descriere

Algoritmul Apriori a fost propus de Agrawal în 1994 [2]. În primul pas, algoritmul Apriori determină toate 1-itemset-urile frecvente, prin scanarea bazei de tranzacții D. În pasul k,

vor fi generate k-itemset-urile candidate, pa baza (k-1)-itemset-urilor frecvente determinate în pasul k-1, după care vor fi reținuți candidații care au suportul mai mare decât suportul minim.

O modalitate de generare a k-itemset-urilor candidate este bazată pe Principiul Apriori [3]: Dacă un itemset este frecvent, atunci toate subset-urile sale sunt frecvente.

Demonstrația se bazează pe proprietatea următoare:

$$\forall\ X, Y : X \subseteq Y \Rightarrow s(X) \geq s(Y) \tag{1}$$

unde X și Y sunt itemset-uri.

În [2] se demonstrează că rezultatele algoritmului Apriori nu sunt influențate de ordinea elementelor (itemi) frecvente.

1.2.2. Pseudocod

Premise:

L_k = mulțimea k-itemset-urilor cu suport mai mare decât suportul minim sau egal cu acesta (itemset-uri frecvente);

C_k = mulțimea k-itemset-urilor candidate pentru a fi frecvente;

D = mulțimea tranzacțiilor, $t \in D$.

```
Apriori(D, Rezultat)
begin
 L1 ← {i | i este item frecvent}
 for k ← 2; Lk-1 ≠ ϕ; k++ do
    Ck ← GenerareCandidati(Lk-1) /* candidați noi */
    for all t∈D do
          Ct ← {c | c∈Ck and c⊂t} /* candidați noi conținuți în t */
          for all c∈Ct do
                c.nr ← c.nr+1
          end for
    end for
    Lk ← {c ⊂Ck | c.nr ≥ suport_minim}
 end for
 Rezultat ← Lk
end
```

În primă fază, algoritmul generează lista itemset-urilor candidate, ce include toate itemset-urile conținute de baza de date. În a doua fază, se calculează suportul fiecărui itemset candidat în baza de date, și se verifică dacă valoarea este mai mică decât un suport minim predefinit. Dacă suportul obținut pentru itemset-ul candidat nu este mai mic, atunci itemset-ul este frecvent.

1.3. Algoritmul Fast Item Miner

1.3.1. Descriere

Ideea de bază este de a determina, pas cu pas, itemset-urile frecvente, prin lărgirea mulțimii din care fac parte elementele (itemii) frecvente, dacă în pasul k fiecare candidat este format din elementele care aparțin uneia dintre mulțimea {i, i+1, …, i+k }, i = 1, 2, ..., n-k, în pasul următor, k+1, fiecare candidat este format din elementele care aparțin uneia dintre mulțimile {i, i+1, …, i+k+1}, i = 1, 2, ..., n-k-1}.

Considerăm n elemente (itemi) frecvente și m tranzacții. În primul pas, candidații vor fi

submulțimi ale mulțimilor {1, 2}, {2, 3}, ..., {n -1, n}. În pasul 2, candidații vor fi submulțimi ale mulțimilor {1, 2, 3}, {2, 3, 4},..., {n- 2, n-1, n}. În ultimul pas, candidații vor fi submulțimi ale mulțimii {1, 2, ..., n}.

Această abordare simplifică procesul de generarea a candidaților şi are potențial de paralelizare mai mare decât algoritmul Apriori [4]

Notăm cu Fi,j mulțimea tuturor itemset-urilor frecvente cu elemente din mulțimea {i, i+1, …, j} și cu Ci,j mulțimea candidaților cu elemente din mulțimea {i, i+1, …, j}.

Formula de calcul pentru Ci,j este dată de relația următoare [4]:

$$Ci,j=\{i \cup Y \vee Y \subset Fi+1,j \wedge j \in Y\} \quad (2)$$

1.3.2. Pseudocod

Premise:

D_i = mulțimea tranzacțiilor asignate unității de procesare Pi.

```
FIM(D, F, Rezultat)
begin
 for k=1 to n-1 do
    for all Pi : i ∈{1,…, n-k} do in parallel
           j ← i+k
           Fi,j ← Fi,j -1 ∪ Fi+1,j
           Ci,j={i∪Y | Y⊂Fi+1,j∧j∈Y )
           for all transactions t ∈ Di do
                  Ct ← {c | c∈Ci,j ∧ c⊂t} /* candidații  conținuți în t */
                        for all c∈Ct do
                               c.nr  ← c.nr+1
                        end for
           end for
           Fi,j ← Fi,j ∪{c ⊂Ci,j | c.nr ≥ suport_minim}
    end for
 end for
 Rezultat ← F1,n
end
```

Algoritmul paralel Fast Item Miner utilizează un patern fix de comunicație. Acesta este marele avantaj al acestui algoritm [5].

Capitolul 2. Proiectarea

2.1. OpenCL

2.1.1. Generalități

OpenCL (Open Computing Language [6]) este un framework pentru dezvoltarea programelor ce se execută pe platforme heterogene consistând în CPU-uri, GPU-uri, DSP-uri, FPGA-uri și alte procesoare sau acceleratoare hardware. Limbajul de dezvoltare OpenCL este bazat pe standardele C99 și C++11, limbaj utilizat în controlul platformelor și executarea programelor pe dispozitivele de procesare, prin intermediul unei interfețe standard pentru calcul paralel utilizând un paralelism bazat pe task-uri și date.

OpenCL oferă multe beneficii în domeniul calculului de înaltă performanță, printre care și portabilitatea. Task-urile codate în OpenCL, numite kernel-e, pot fi executate pe GPU sau CPU de la dezvoltatori precum Intel, AMD, Nvidia și IBM. Pe lângă faptul că OpenCL poate rula pe tipuri diferite de dispozitive, o aplicație poate expedia kernel-e la mai multe dispozitive în același timp.

Cele 5 structuri de date, implicate în comunicarea OpenCL dintre host și dispozitivele de calcul:

- Device – dispozitivul care se ocupă de calculul efectiv;
- Kernel – un task, set de instrucțiuni, care specifică modalitățile de calcul și manipulare a datelor, la nivel de dispozitiv;
- Program – această structură reține task-uri pe care host-ul le expediază către dispozitivele de calcul;
- Command queue – structură prin intermediul căruia host-ul face schimb de date cu dispozitivul;
- Context – permite dispozitivelor ăs primească kernel-e și să transfere date;

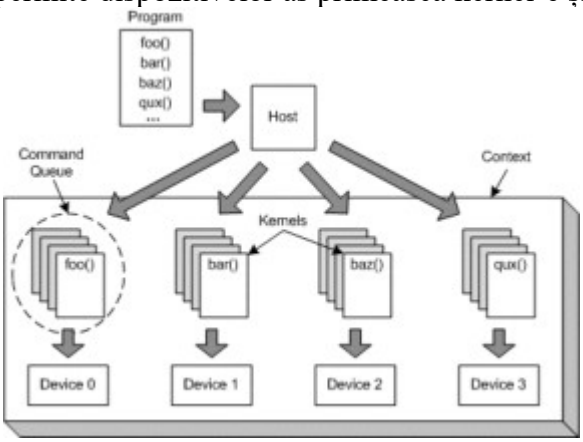

Figura 2.1. Distribuirea kernel-elor dispozitivelor conforme OpenCL [7].

Unul din avantajele OpenCL este faptul că kernel-ele pot fi executate pe dispozitive de calcul de înaltă performanță precum GPU-uri. Pentru exploatarea acestor capabilități trebuie avute în vedere:

- Modelul de execuție OpenCL – Kernel-ele sunt executate de unul sau mai mulți itemi de lucru (similar unui thread). Itemii de lucru sunt grupați în grupuri de lucru, și fiecare grup de lucru este executat pe o unitate de procesare;

- Modelul de memorie OpenCL – Datele unui kernel trebuie alocate în una din cele patru spații de adrese – memorie globală, memorie constantă, memorie locală sau memorie privată. Locația datelor determină viteza cu care acestea sunt procesate.

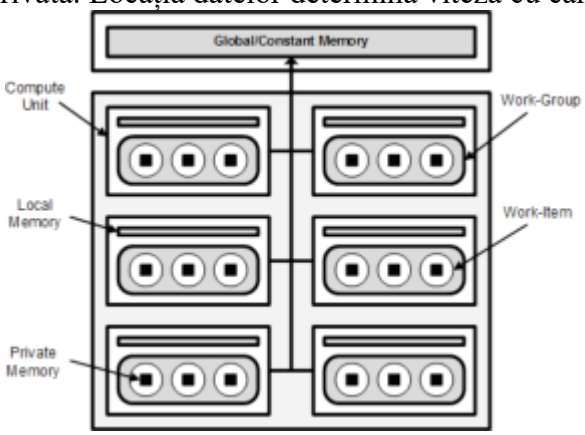

Figura 2.2. Modelul dispozitivelor OpenCL [7].

Fiecare item de lucru este identificat printr-un ID global și unul local. ID-ul global identifică item-ul de lucru printre toți itemii de lucru ce execută kernel-ul. ID-ul local identifică item-ul de lucru în grupul de lucru. Itemii de lucru în grupuri de lucru diferite pot avea același ID local, dar nu o să dețină niciodată același ID global.

Cele 4 spații de memorie:

- Memoria globală – stochează date pentru întregul dispozitiv, vizibilă tuturor itemilor de lucru;
- Memoria constanta – similară cu memoria globală, cu precizarea ca este „read-only”;
- Memoria locală – stochează date accesibile pentru itemii de lucru dintr-un grup de lucru;
- Memoria privată – stochează date accesibile unui singur item de lucru.

Pentru un dispozitiv ce conține M unități de procesare și N itemi de lucru per grup de lucru, la un moment dat doar MxN itemi de lucru vor executa kernel-ul.

2.1.2. Avantaje

În calculul paralel de înaltă performanță, anumite avantaje oferite de OpenCL îi determină pe dezvoltatori în alegerea acestui framework, în ciuda altora existente, din care amintim:

- suport oferit pentru diferite platforme ce vin de la diferiți producători;
- portabilitate – un program realizat pe baza OpenCL poate fi executat pe diferite GPU-uri indiferent de producător și de platforma pe care a fost dezvoltat inițial;
- OpenCL oferă o interfață orientată pe task-uri și date, lucru ce se dovedește foarte util în calculele de înaltă performanță;
- este un framework într-o continuă dezvoltare, primind îmbunătățiri cu fiecare versiune nouă;
- ales din ce în ce de mai mulți dezvoltatori de aplicații comerciale datorită portabilității care este valabilă pentru componentele hardware apărute după 2010/2012;
- același cod dezvoltat în OpenCL este valabil și pentru CPU-uri cât și pentru GPU-uri.

2.2. Componente software

În proiectarea aplicației, s-a considerat utilizarea următoarelor tehnologii:

- System de operare: Windows 10 x64.
- Python 3.6 – Pune la dispoziție biblioteci (numite pachete în Python) scrise în C/C++ instalabile prin pachetul „pip”. În plus, în Python, instrucțiunile necesare, în cazul OpenCL îndeosebi, sunt mai puține, astfel dezvoltatorul își poate concentra atenția pe obiectivul în cauză. S-a considerat limbajul potrivit, având în vedere că biblioteca este obiectivul realizării unui „proof of concept” în vederea evidențierii importanța utilizării GPU-ului în operații de data mining.
- PyCharm – IDE pentru dezvoltarea aplicațiilor realizate în Python.
- PyOpenCL – Un pachet open-source Python scris în C++ pentru dezvoltarea aplicațiilor OpenCL utilizând limbajul Python. Oferă acces la avantajele calculului paralel oferit de OpenCL.

În dezvoltarea algoritmilor de data mining s-a considerat următoarea diagramă de secvență:

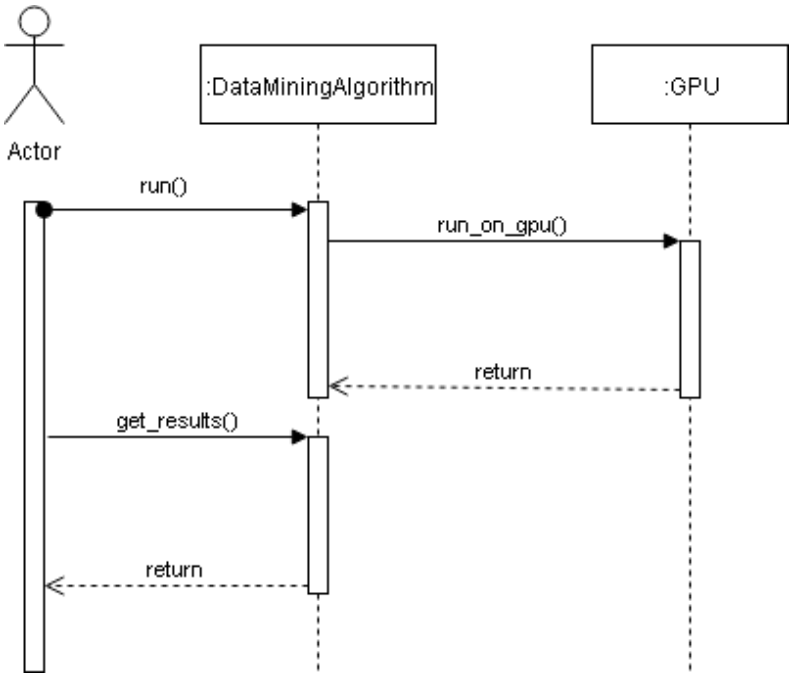

Figura 2.3. Diagrama de secvență generală.

unde:

- actorul este aplicația/programul care apelează metoda obiectului de data mining;
- DataMiningAlgorithm reprezintă clasa superioară, moștenită de clasele algoritmilor de data mining, care operează pe GPU;
- GPU (Graphical Processing Unit) este componenta hardware pe care se realizează calculele prin intermediul OpenCL.

Scopul clasei de bază DataMiningAlgoritm este de a impune o anumită structură claselor derivate, având în vedere faptul că în Python nu există o interfață similară C++ (Programare orientată obiect). Structura impusă arată în felul următor:

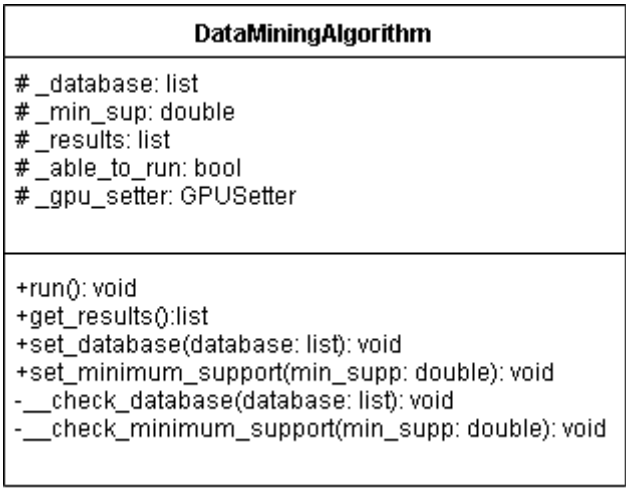

Figura 2.4. Clasa de bază DataMiningAlgorithm.

Pentru determinarea specificațiilor OpenCL și observarea stării GPU-ului s-au utilizat aplicațiile GPU Caps Viewer și TechPowerUp GPU-Z.

2.3. Componente hardware

Platforma îndeplinește următoarele specificații hardware:

- CPU AMD Ryzen 7 2700U cu 8 nuclee logice și frecvența 2.2GHz (3.8GHz cu accelerație);
- RAM DDR4 8GB;
- SSD 500GB;
- GPU dedicat AMD Radeon RX 560x 4GB;
- GPU integrat AMD Radeon RX Vega 10.

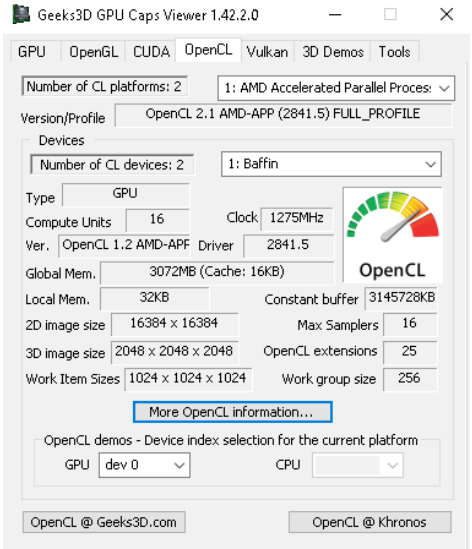

Figura 2.5. AMD Radeon RX 560x OpenCL specificații.

Testele au fost rulate exclusiv pe GPU-ul dedicat a cărui specificații OpenCL sunt prezentate în Figura 2.5.

Capitolul 3. Implementarea

3.1. Modelul hostCPU-workerGPU

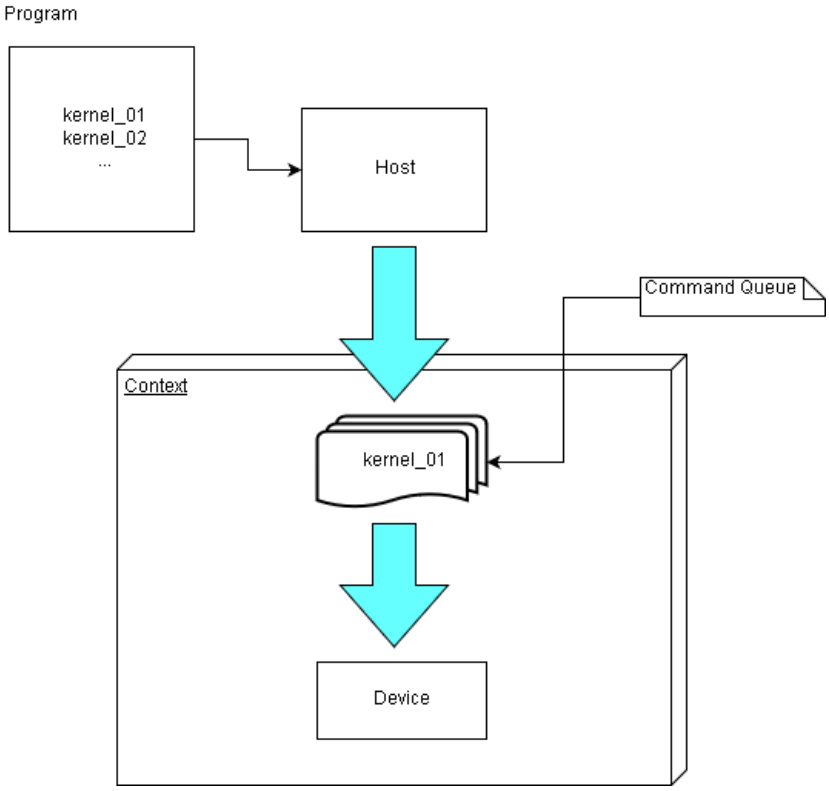

Figura 3.1. Model comunicare hostCPU-workerGPU

Modelul hostCPU - workerGPU, prezentat în Figura 3.1, este un model de comunicare între host și procesorul grafic, adoptat în adaptarea algoritmilor abordați pentru rezolvarea acestora, utilizând paralelismul pus la dispoziție de procesorul grafic. Modelul este similar modelului master-slave 1:1, cu alte cuvinte un master are un singur slave. În cazul de față, master-ul este host-ul (CPU) și slave-ul este procesorul grafic (GPU). În abordarea considerată, host-ul trimite task-uri către GPU și așteaptă până la terminarea și primirea rezultatelor lor.

Program – colecție de unul sau mai multe kernel-e, plus (opțional) funcții/proceduri suplimentare.

Context – grup de device-uri alese pentru a fi folosite; în cazul de față va fi un singur device: procesorul grafic.

Command Queue – obiect ce conține instrucțiuni ce informează care dintre device-urile din context va executa un anumit task și cum.

Kernel – reprezintă o entitate executabilă care pot fi trimise prin intermediul cozii de comandă către un anumit device pentru execuție. Un kernel este dezvoltat în OpenCL C, standard utilizat de OpenCL.

Host – dezvoltat în Python 3.6, se ocupă cu modelarea structurilor de date pentru trimiterea lor către device și implementarea interfețelor algoritmilor.

Device – procesorul grafic la nivelul căruia se vor realiza etapele costisitoare, din punct de vedere al timpului, a algoritmilor, pe baza paralelismului.

3.2. Apriori

3.2.1. Schema logică

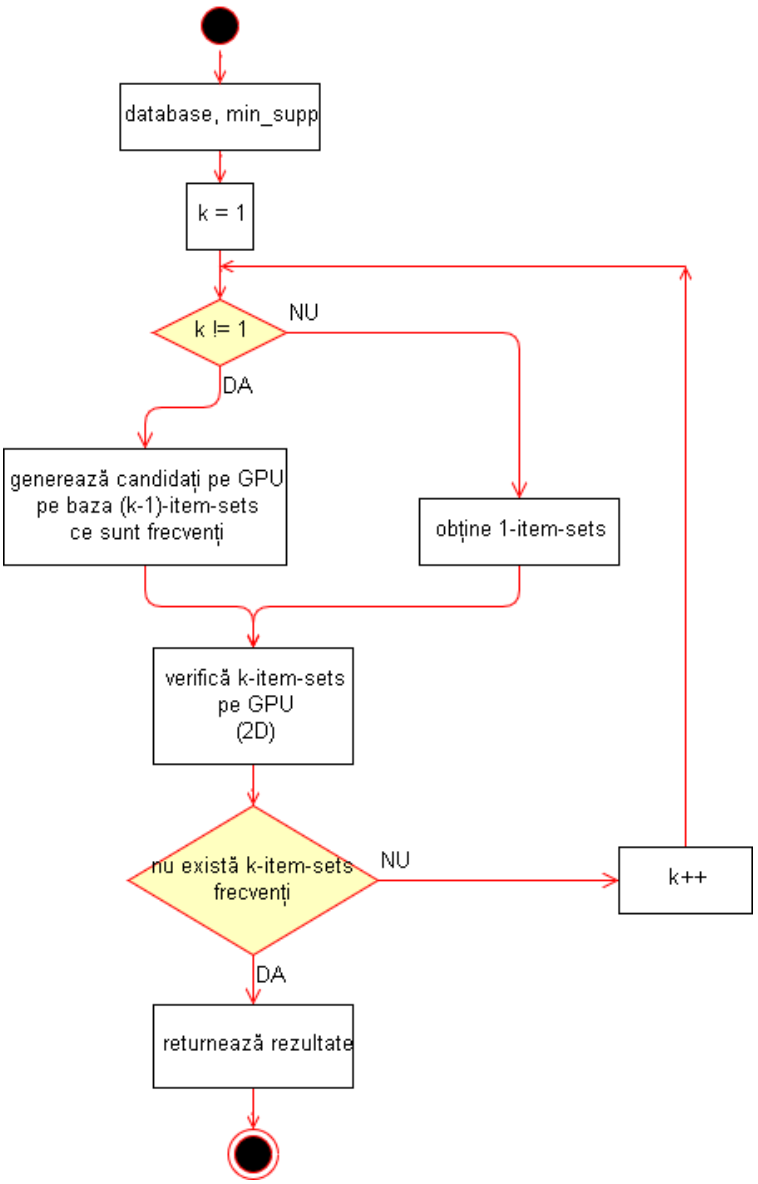

Figura 3.2. Implementarea algoritmului Apriori

Algoritmul Apriori determină k-itemset-urile frecvente în colecții mari de date prin executarea repetitivă a două mari etape:

- Etapa de generare a candidaților;
- Etapa de identificare a itemset-urilor frecvente;

Etapa de generare a candidaților presupune crearea de noi itemset-uri de dimensiune k, despre care nu se știe dacă sunt frecvente sau nu, pe baza itemset-urilor frecvente de dimensiune k-1, deci itemset-urile frecvente de la pasul anterior.

Etapa de identificare a itemset-urilor frecvente presupune verificarea condiției de frecvență pentru fiecare itemset candidat în parte și luarea deciziei dacă acesta din urmă este sau nu un itemset frecvent.

Executate într-o manieră secvențială, aceste două etape se pot dovedi atât consumatoare

de memorie, cât și consumatoare de timp, astfel compromisul „consum mare de memorie pentru un timp de execuție scăzut” fiind invalidat. Pentru păstrarea compromisului, introducem implementările GPGPU, al căror paralelism, pus la dispoziție de GPU, propune rezolvarea execuției algoritmului într-un interval mare de timp.

După cum este prezentat în Figura 3.2, algoritmul Apriori necesită execuția a n pași pentru găsirea itemset-urilor frecvente, unde n reprezintă dimensiunea ultimilor candidați generați, cu alte cuvinte, candidații de dimensiune maximă care au fost generați și dintre care pot fi găsiți sau nu frecvenți.

3.2.2. Generarea candidaților pe GPU

În cazul algoritmului Apriori, generarea candidaților presupune crearea itemset-urilor de dimensiune k pe baza itemset-urilor frecvente de dimensiune k-1. Deci, itemset-urile candidate sunt create pe baza itemset-urilor frecvente identificate la pasul anterior. Se numesc „candidate” deoarece nu se cunoaște starea acestor itemset-uri nou create, nu se știe dacă sunt frecvente sau nu, dar vor fi testate pentru verificarea condiției de frecvență.

Crearea unui itemset candidat de dimensiune k presupune realizarea operației matematice de uniune între doua itemset-uri frecvente de dimensiune k-1. Execuția secvențială a etapei de generare a candidaților ar presupune crearea a câte un candidat la un moment dat, lucru care în cazul unui număr mare de candidați de creat, de exemplu câteva milioane, ar necesita un timp mare de execuție.

O abordare paralelă în generarea candidaților poate reduce timpul necesar obținerii lor, având în vedere că la un moment dat ar fi generați mai mulți candidați, și nu doar unul. Totuși, un CPU este limitat în acest sens la numărul de coruri logice pe care le deține. Pentru un CPU cu 8 coruri logice s-ar genera 8 itemset-uri candidate la un moment dat, ceea ce este un avantaj, dar nu este îndeajuns. De aceea, abordăm problema propusă din perspectiva procesorului grafic, care după cum știm suportă un paralelism mult mai ridicat față de GPU, având un număr mult mai mare de coruri (unele au până la 4000 de coruri).

Generarea unui itemset candidat este realizată pe baza operației matematice de uniune, în intermediul căreia se realizează și o sortare a elementelor în candidatul nou generat, similar procesului de unire din algoritmul „Merge Sort”. Această sortare se realizează pentru a nu avea itemset-uri candidate cu aceleași elemente, dar în altă ordine, lucru ce ar introduce ambiguitate și drept urmare păstrarea ambelor itemset-uri candidate. Fiecare itemset candidat este unic, deține o particularitate care îi este specifică doar lui.

Fiind necesară generarea itemset-urilor candidate de dimensiune k, și având la dispoziție o listă de itemset-uri frecvente de dimensiune k-1, generarea se realizează în felul următor: fiecare itemset frecvent din listă este unit cu fiecare itemset frecvent ce îl urmează, astfel fiind create itemset-urile candidate, dar nu și cu cele ce îl preced. Această abordare este cunoscută sub numele de „principiul de tăiere (pruning)”, care are ca scop reducerea atât a memoriei necesare cât și a timpului, prin evitarea generării de itemset-uri candidate care sunt evident dubluri ale altora deja existente. După cum s-a menționat, interesul este de a avea itemset-uri candidate unice. Principiul de tăiere este prezentat pe scurt în Figura 3.3.

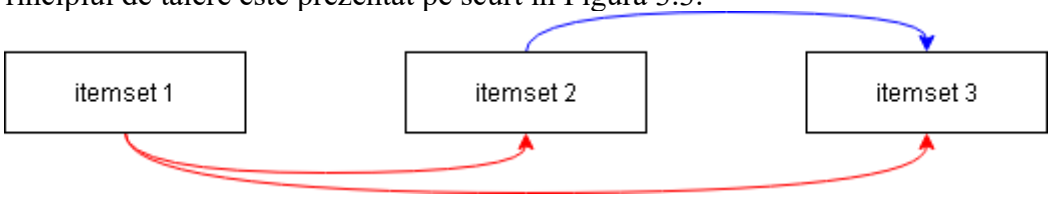

Figura 3.3. Principiul de tăiere (pruning).

Figura 3.3 prezintă succint modul de formare a itemset-urilor candidate. Itemset-ul 1

formează doi candidați cu itemset-ul 2 și itemset-ul 3, iar itemset-ul 2 formează un candidat prin uniunea cu itemset-ul 3. Se observă că itemset-ul 2 nu a fost unit cu itemset-ul 1, deoarece operația de uniune a fost deja făcută anterior, într-o altă ordine.

Fiecare itemset candidat creat trebuie validat astfel încât să nu apară itemset-uri de dimensiuni greșite în urma operației de uniune. Prin itemset-uri de dimensiuni greșite referim acele itemset-uri candidate care în urma operației de uniune între două itemset-uri frecvente, de dimensiune k-1, rezultă candidați de dimensiune mai mare strict decât k, când itemset-urile candidate ar trebui să aibă dimensiunea k. Implementarea propusă, validează candidații prin verificarea dimensiunii lor după operația de uniune în felul următor: dacă procesul de uniune formează un itemset candidat de dimensiune k atunci acesta este valid, altfel, dacă dimensiunea este mai mare decât k, itemsetul nu este valid. O alternativă, la această metodă, este testarea dimensiunii rezultatului operației de intersecție, realizată pe itemset-urile frecvente propuse pentru operația de uniune. Dacă dimensiunea intersecției este egală cu d-1, unde d este dimensiunea itemset-urilor frecvente, atunci itemset-ul candidat este valid, altfel nu este valid.

În comunicarea hostCPU – workerGPU, host-ul trimite itemset-urile frecvente de la pasul anterior, de dimensiune k-1, și kernelul de executat, deja existent, către GPU, pentru procesare, și așteaptă să primească înapoi itemset-urile candidate de dimensiune k. În funcție de numărul de itemset-uri frecvente, host-ul cere procesorului grafic maximul de putere de calcul pe care îl poate oferi și care totodată este necesar. În cazul în care procesorul grafic nu poate satisface cerințele în privința puterii de calcul, host-ul împarte lista de itemset-uri frecvente de la pasul anterior în bucăți mai mici, care să poată fi tratate separat de procesorul grafic, de dimensiuni egale cu numărul de elemente de lucru de pe o dimensiune. După terminarea procesării tuturor bucăților, listele mici, acestea sunt adunate într-o singură listă de itemset-uri candidate.

La nivel de GPU, generarea itemset-urilor candidate este realizată într-o manieră paralelă, pe mai multe dimensiuni, în felul următor:

- dimensiunea 1 – pe această dimensiune sunt tratate itemset-urile frecvente de dimensiune k-1;
- dimensiunea 0 – pe această dimensiune sunt tratate itemset-urile frecvente de dimensiune k-1, următoare itemset-ului frecvent curent tratat de dimensiunea 1 pentru grupul în cauză;
- dimensiunea 2 – o dimensiune necesară pentru a crea dinamic tablouri unidimensionale necesare pentru stocarea temporară a itemset-urilor frecvente de procesat.

Cu alte cuvinte, informația de mai sus poate fi exprimată în felul următor: „Pentru fiecare itemset frecvent este asignat un item de lucru (fir de execuție), care la rândul său are mai mulți itemi de lucru, pentru tratarea itemset-urilor frecvente următoare itemset-ului corespunzător master-ului”. După cum se poate înțelege, dimensiunea folosită în plus nu are legătură cu tratarea itemset-urilor frecvente, ci cu crearea dinamică de tablouri unidimensionale odată cu crearea firelor de execuție, având în vedere faptul că, pentru OpenCL, acestea nu pot fi create în interiorul firelor de execuție dinamic, prin funcția „malloc”.

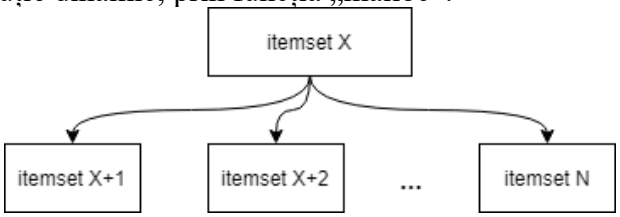

Figura 3.4. Dimensionarea generării candidaților pentru algoritmul Apriori.

3.2.3. Identificarea itemset-urilor frecvente

Având o listă de itemset-uri candidate de dimensiune k pregătită, se dorește obținerea itemset-urilor frecvente de dimensiune k, existente în baza de date pusă la dispoziție. Pentru aceasta, pentru fiecare itemset candidat se verifică condiția de frecvență, și dacă este îndeplinită atunci itemset-ul candidat este frecvent și se păstrează.

Realizată secvențial, această etapă ar presupune ca itemset-urile candidate să fie verificate în baza de date pe rând, unul după altul, ceea ce , pentru colecțiile mari de date, devine foarte costisitor din punct de vedere al timpului. Deci baza de date ar fi scanată de atâtea ori câte itemset-uri candidate sunt. Presupunem că, pentru o bază de date de dimensiune mare, scanarea unui itemset candidat de dimensiune k ar necesita o secundă, și avem de testat un milion de itemset-uri candidate. Rezultă că în total sunt necesare un milion de secunde pentru scanările bazei de date, ceea ce în ore înseamnă aproximativ 277 ore. Acesta este un exemplu fictiv, dar 11 zile pentru obținerea rezultatelor este un interval de timp mare.

Abordată într-o manieră paralelă pe CPU, această etapă ar presupune verificarea mai multor itemset-uri candidate la un moment dat, de exemplu 8 pentru un CPU cu 8 nuclee logice. Folosind procesorul grafic se poate obține un paralelism mult mai ridicat, având în vedere faptul că procesoarele grafice au mult mai multe nuclee.

Pe lângă paralelizarea verificării itemset-urilor candidate, implementarea propune și paralelizarea scanării bazei de date. Aceasta presupune împărțirea bazei de date în grupuri de tranzacții pentru a fi scanate separat și paralel. Pentru fiecare master, fir de execuție ce verifică un itemset candidat propriu, există o mulțime de itemi de lucru care scanează baza de date, pentru fiecare item de lucru având un grup de tranzacții. Grupurile de tranzacții pot diferi ca dimensiune, dar nu mai mult de o unitate, acestea fiind echilibrate astfel încât și procesarea la nivelul item-ilor de lucru să fie echilibrată. În acest fel, un item de lucru poate scana cu o tranzacție în minus sau în plus, dar nu mai mult, față de un alt item de lucru. Baza de date este împărțită în numărul maxim de elemente care pot compune un grup de lucru pe GPU, în OpenCL. Pentru fiecare astfel de grup de tranzacții, se verifică existența itemset-ului candidat în fiecare tranzacție, prin căutarea elementelor itemset-ului în tranzacție. Dacă se trage concluzia că pentru tranzacția în cauză itemset-ul candidat este prezent, atunci tranzacția este contorizată, contor ce semnifică numărul de apariții a itemset-ului în baza de date. Căutarea elementelor itemset-ului se realizează prin algoritmul de „căutare prin interpolare”.

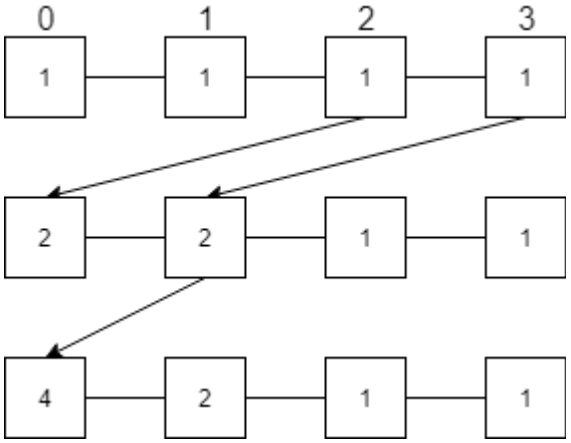

Figura 3.5. Reducere paralelă.

Pentru fiecare grup de tranzacții format este reținut un contor pentru a număra de câte ori apare itemset-ul candidat în acel grup. Se folosește un tablou unidimensional de tip local memory shared de dimensiune egală cu numărul de itemi de lucru la care este împărțită baza de

date. Fiecare item de lucru contorizează propriul grup de tranzacții și salvează acest contor în tabloul unidimensional de tip local shared memory, la indicele corespunzător. Toți acești contori sunt însumați prin algoritmul de „reducere paralelă” la poziția 0 a tabloului unidimensional obținând astfel numărul total de apariții a itemset-ului candidat în baza de date (vezi Figura 3.5).

Având numărul de apariții a itemset-ului candidat în baza de date și numărul total de tranzacții, se calculează suportul itemset-ului în cauză după (3).

$$Suport(I) = \frac{număr\ apariții\ I}{număr\ total\ tranzacții} \tag{3}$$

După aflarea suportului itemset-ului candidat, se testează condiția de frecvență prezentată în (4).

$$Suport(I) \geq Suport\ minim \tag{4}$$

Dacă suportul itemset-ului candidat respectă condiția prezentată în (4) atunci itemset-ul este frecvent și se păstrează.

La nivel de GPU, verificarea itemset-urilor candidate, pentru frecvență, este realizată paralel, dimensionarea fiind următoarea:

- dimensiunea 1 – pe această dimensiune sunt tratate itemset-urile candidate, care vor fi verificate pentru frecvență;
- dimensiunea 0 – pe această dimensiune sunt tratate grupurile de tranzacții, în care este împărțită baza de date, paralel.

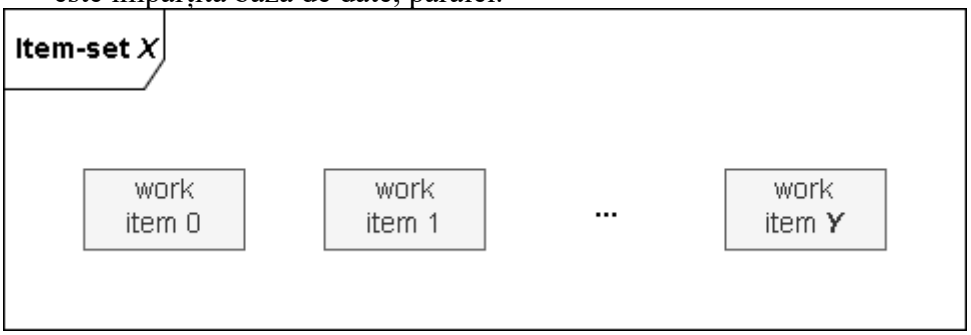

Figura 3.6. Dimensionare locală algoritm Apriori.

În Figura 3.6, itemset-ul X este atribuit item-elor de lucru, cu indecși între 0 și Y, acestea din urmă având pus la dispoziție un tablou unidimensional de tip local shared memory. Analogia master-slave poate fi interpretată în felul următor, pentru modelul de dimensionare din OpenCL:

- master – grupul de lucru din care fac parte item-ele de lucru care împărtășesc aceeași zonă de memorie (tabloul unidimensional);
- slave – fiecare item de lucru din grupul de lucru are propriile lui particularități și se ocupă cu contorizarea numărului de apariții în grupul de tranzacții din baza de date ce i-a fost atribuit, salvând contorul în tabloul unidimensional pe care îl împărtășește cu celelalte elemente de lucru.

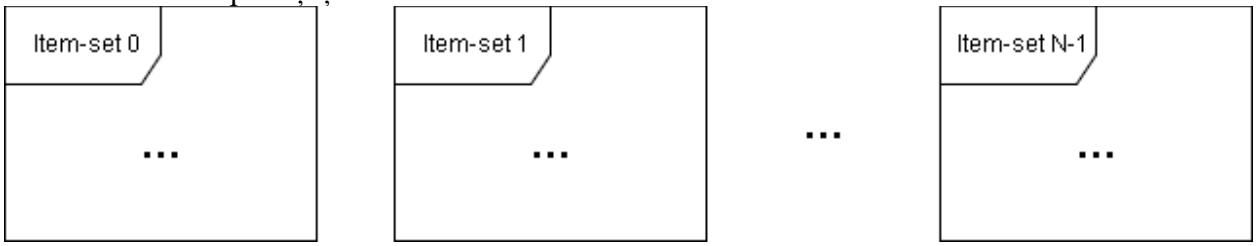

Figura 3.7. Dimensionare globală algoritm Apriori.

Se înțelege faptul că termenul de master este fictiv, în cazul GPU-ului, și semnifică nu un thread/proces efectiv, palpabil, ca în cazul CPU-ului, ci un grup de iteme de lucru care lucrează

împreună, paralel, pentru obținerea unui rezultat. În Figura 3.7 sunt înfățișate grupurile locale, fiecare având ca obiectiv testarea propriului itemset candidat atribuit, pentru a descoperi dacă respectivul itemset este frecvent sau nu.

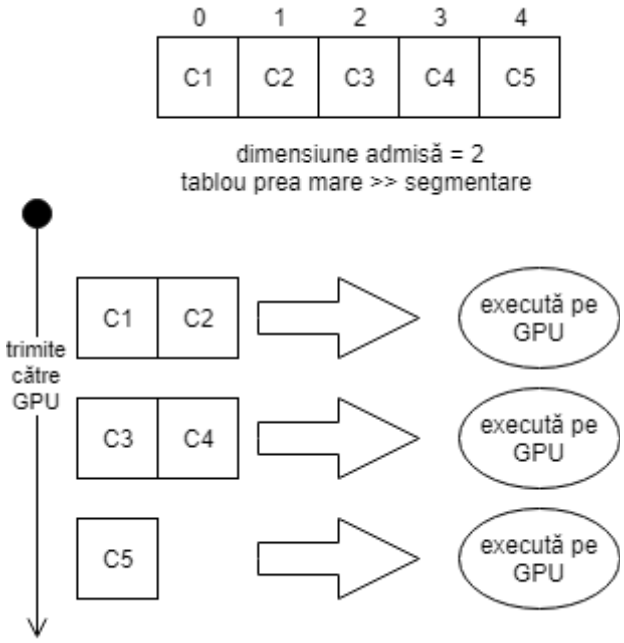

Figura 3.8. Partiționare listă itemset-uri candidate în caz de dimensiune prea mare.

În caz că numărul de itemset-uri candidate este prea mare pentru procesarea integrală pe GPU, host-ul partiționează lista de itemset-uri candidate în liste mai mici de dimensiuni mai mici sau egale cu numărul maxim de elemente de lucru admis de o dimensiune pe GPU, după cum este figurat în Figura 3.8. Trimiterea listelor mici se realizează secvențial datorită faptului că GPU-ul poate prelucra un singur task (kernel) la un moment dat.

3.2.4. Optimizări

Procesorul grafic oferă avantajul unui grad de paralelism ridicat, cu mult mai mare față de CPU, însă odată cu acesta vin și anumite limitări precum:

- dimensiunea memoriei împărtășită între elementele de lucru (local memory);
- manipularea memoriei prin instrucțiunea „malloc” nu este oferită;
- comunicarea între elementele de lucru se realizează în principiu prin intermediul memorie locale care este „shared” doar între elementele de lucru ale unui grup, fiind incluse „pipe-urile” odată cu OpenCL 2.0;
- viteza de acces la memorie având cele patru tipuri de memorie: globală, constantă, locală, privată;
- nu există importuri.

Toate aceste constrângeri trebuie luate în calcul în momentul dezvoltării kernel-ului pentru a obține rezultatele dorite. Pentru a îmbunătăți rezultatele, s-au luat în considerare anumite optimizări la nivel de cod, instrucțiuni, pentru a reduce numărul de instrucțiuni parcurse, în anumite cazuri, și totodată pentru a economisi timp de execuție. În continuare vor fi prezentate optimizările implementate care au crescut performanța și au adus beneficii în privința timpului necesar obținerii rezultatelor.

3.2.4.1. Căutarea

O căutare simplă, realizată doar prin instrucțiuni repetitive, fără vreun artificiu, ar oferi o complexitate $O(n^2)$, complexitate care atrage totodată și un timp mai mare de execuție, mai ales în cazul unui tablou de dimensiune mare. Cum o tranzacție este tratată ca un tablou unidimensional, și dimensiunile acesteia pot varia până la unele foarte mari, s-a luat în considerare utilizarea unui algoritm de căutare care să ofere o complexitate timp mult mai mică. O idee ar fi utilizarea algoritmului de „căutare binară" cu complexitatea O(log n), dar având în vedere că tranzacțiile sunt sortate, o idee mai bună a fost considerată căutarea prin „interpolare" cu complexitatea O(log log n). Căutarea a fost implementată în felul următor:

```
bool InterpolationSearch(__global int* database, int begin, int end, int elem)
    {
        while (database[begin] <= elem && elem <= database[end] && begin != end)
        {
            int mid = begin + ((elem - database[begin]) * (end-begin) /
(database[end] - database[begin]));

            if (database[mid] == elem)
                return true;
            else if (database[mid] < elem)
                begin = mid + 1;
            else
                end = mid;
        }

        return false;
    }
```

Căutarea unui itemset candidat, în baza de date, presupune cătarea tuturor elementelor itemset-ului, în fiecare tranzacție, unul câte unul. Un itemset candidat este considerat găsit în tranzacție dacă toate elementele sale au fost găsite în acea tranzacție, fără excepție.

Pentru a îmbunătăți viteza de determinare a numărului de apariții a itemset-ului candidat în baza de date, s-a considerat că dacă un element dintr-un itemset nu este găsit, într-o tranzacție, să se oprească căutarea acelui itemset în tranzacția în cauză. Astfel sunt evitate executarea unor instrucțiuni care nu aduc nici un beneficiu.

3.2.4.2. Sortarea

Pentru ca complexitatea timp a algoritmilor de căutare să fie cât mai mică, aceștia trebuie să realizeze căutarea în condițiile cele mai avantajoase, cu alte cuvinte, căutarea trebuie realizată în tranzacții sortate crescător. Pentru algoritmul căutării prin interpolare, în cel mai bun caz obținem o complexitate de timp O(log log n). Marele avantaj al acestei căutări este că dacă elementul căutat nu are valori între elementul de început al tranzacției și elementul de final, nu se mai realizează căutarea pe tranzacția respectivă, ceea ce salvează timp. Pentru ca acest lucru să funcționeze tranzacția trebuie sortată crescător.

Sortarea tranzacțiilor din baza de date este realizată în momentul în care aceasta este atribuită obiectului de tip Apriori. Asupra acesteia se rulează verificări, automat, pentru verificarea corectitudinii tranzacțiilor și totodată este realizată și sortarea. Sortarea este realizată prin metoda built-in a mediului de dezvoltare pentru host, Python, de sortare a unui liste, a cărei viteză de execuție a fost testată.

3.2.4.3. Eliminarea elementelor care nu sunt frecvente în faza de pre-procesare

Scanarea bazei de date poate necesita un interval de timp mare chiar și în cazul unei căutări optimizate datorită dimensiunilor mari. De aceea, după aflarea itemset-urilor frecvente de dimensiune 1, se realizează o parsare a bazei de date cu scopul de a elimina elementele din tranzacții care nu sunt frecvente. Această eliminare poate reduce dimensiunile bazei de date consistent, în anumite cazuri, astfel următoarele scanări vor fi realizate pe dimensiuni mai mici față de cele inițiale. Eliminând elementele ce nu sunt frecvente putem obține tranzacții de dimensiuni mai reduse, evitând scanarea unor elemente care sigur nu vor aduce nici un rezultat. În felul acesta este redusă dimensiunea bazei de date în lățime. Uneori, eliminarea elementelor ce nu sunt frecvente poate reduce o tranzacție la dimensiunea 0, caz în care aceasta este eliminată. Astfel, dimensiunea bazei de date este redusă în înălțime.

Această pre-procesare a bazei de date este realizată o singură dată, deoarece, în cazul bazelor de date de mari dimensiuni, efectuarea multiplă a acesteia poate atrage cost suplimentar din punct de vedere al timpului necesar obținerii rezultatelor.

3.2.5. Condiții de stop

Ideea de bază a algoritmului Apriori este aceea de a genera itemset-uri candidate și verificarea acestora pentru a identifica care itemset-uri sunt frecvente. Acest proces se realizează de mai multe ori, până se ajunge într-un pas în care nu au fost găsite itemset-uri frecvente, caz în care rezultatul este reprezentat de itemset-urile frecvente de la pasul anterior. Uneori însă, algoritmul poate ajunge într-un pas în care s-a obținut un singur itemset frecvent. Continuarea ar fi inutilă, deoarece nu pot fi generați candidați dintr-un singur itemset frecvent. Deci, condiția de stop principală este ca lungimea liste ce conține itemset-urile frecvente să respecte:

$$lungime \leq 1 \tag{5}$$

Trebuie avută în vedere și posibilitatea ca la un moment dat etapa de generare a itemset-urilor candidate să nu producă nici un itemset candidat. Deci, lungimea listei ce conține itemset-urile candidate, după generare, trebuie să respecte condiția:

$$lungime > 0 \tag{6}$$

În caz de nerespectare a condiției, se oprește executarea algoritmului și se returnează itemset-urile frecvente.

3.3. Fast Item Miner

3.3.1. Schema logică

Algoritmul Fast Item Miner este destinat, exclusiv, unei implementări paralele, datorită modului de distribuire a volumului de lucru și modului de comunicare între unitățile de procesare. După cum se poate observa în Figura 3.9, în obținerea rezultatelor, algoritmul realizează două etape de bază:

- identificarea itemset-urilor frecvente inițiale, de dimensiune 1;
- obținerea rezultatelor prin aplicarea algoritmului „Fast Item Miner”, efectiv, plecând de la itemset-urile frecvente inițiale.

Identificarea itemset-urilor frecvente inițiale este similară cu un executarea pasului algoritmului Apriori pentru aflarea itemset-urilor frecvente de dimensiune 1.

Determinarea itemset-urilor frecvente presupune găsirea itemset-urilor frecvente prin executarea efectivă a algorimului Fast Item Miner, algoritm care parcurge un număr predefinit de pași, indiferent de cât de repede sunt găsite rezultatele finale.

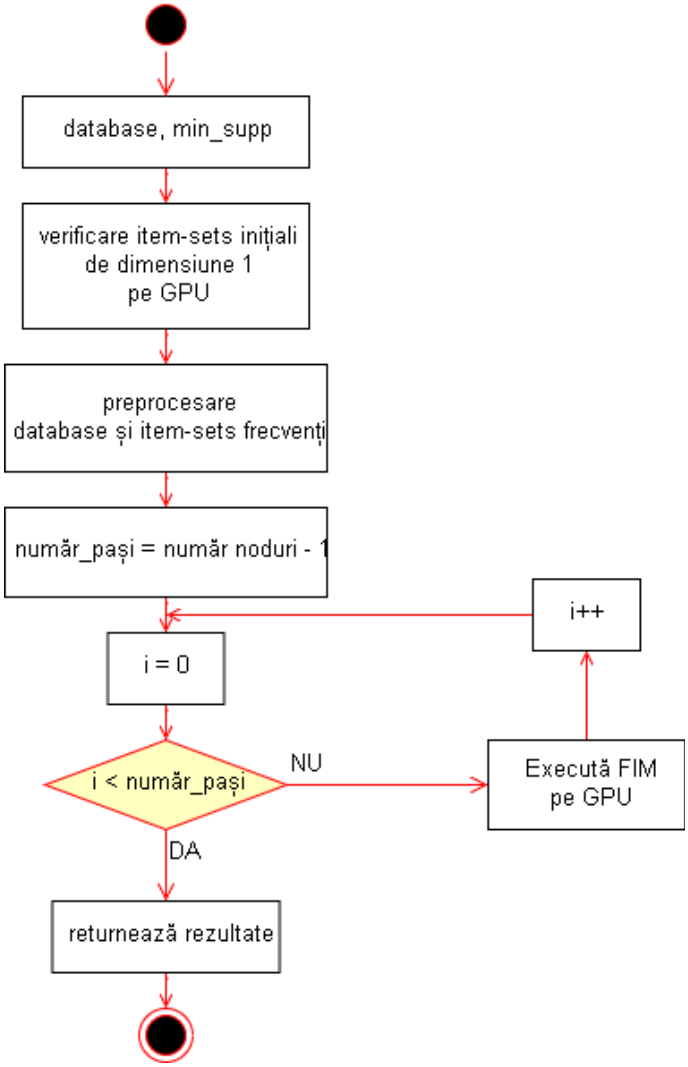

Figura 3.9. Implementarea algoritmului Fast Item Miner

Implementarea este abordată din perspectiva algoritmului Fast Item Miner generalizat, pentru a putea continua dezvoltarea implementării pe viitor. La momentul dezvoltării lucrării, implementarea acoperă funcționalitatea algoritmului simplu, a căror limtări vor fi prezentate în sub-subcapitolul corespunzător.

Datorită complexității algoritmului și modelului de comunicare care nu poate fi suportat integral pe procesorul grafic, la momentul realizării implementării, nu s-a reușit realizarea unei implementări, a procesului de găsire a itemset-urilor frecvente, integrale în OpenCL. De aceea, implementarea algoritmului a fost abordată din perspectiva comunicării hostCPU-workerGPU, prezentată anterior, în felul următor:

- determinarea itemset-urilor inițiale pe GPU;
- preprocesarea bazei de date;
- identificarea numărului de pași pe care algoritmul îi va parcurge în funcție de numărul de noduri pe care sunt distribuite itemset-urile inițiale;
- lansare pas pentru procesarea pe GPU;
- primirea și modelarea datelor pentru următorul pas;
- continuarea executării pașilor pe GPU până la finalizarea acestora;
- validarea itemset-urilor și returnarea acestora.

În continuarea vor fi prezentate etapele necesare obținerii rezultatelor într-o ordine

cronologică a acțiunilor.

3.3.2. Identificarea itemset-urilor frecvente inițialele

În documentația de referință, algoritmul Fast Item Miner pornește procesarea de la existența unor itemset-uri frecvente inițiale, itemset-uri de dimensiune egală cu 1. De aceea, în primă fază, implementarea propusă determină itemset-urile frecvente inițiale.

Această etapă este asemănătoare executării unui pas a algoritmului Apriori, pentru determinarea itemset-urilor frecvente de dimensiune 1. Astfel, se identifică elementele distincte în baza de date ce vor constitui itemset-urile candidate de dimensiune 1, itemset-uri ce vor fi verificate la nivelul procesorului grafic, pentru a se determina care dintre ele sunt frecvente. După identificarea itemset-urilor candidate, de dimensiune 1, se determină dacă se pot verifica toate itemset-urile odată la nivelul GPU-ului, sau este necesară împărțirea listei de itemset-uri în liste mai mici, ce pot fi prelucrate de GPU. Această partiționare este prezentată în Figura 3.8, unde o listă mare, a cărei dimensiune este prea mare pentru a fi procesată pe GPU odată, este împărțită în liste mai mici, care pot fi procesate integral pe GPU, și verificarea itemset-urilor candidate, conținute de acestea, într-o manieră secvențială.

Fiecare itemset candidat este atribuit unui grup de lucru, grup a cărui itemi de lucru scanează, concomitent, porțiuni unice destinate fiecăruia. Deci, un item de lucru scanează un grup de tranzacții din baza de date care îi este destinat încă de la nivelul host-ului. Host-ul împarte baza de date în grupuri unice de tranzacții, pe baza numărului de itemi de lucru posibili într-un grup și numărul total de tranzacții, astfel încât volumul de lucru la nivelul procesorului grafic să fie eficient echilibrat.

La nivel de GPU, verificarea itemset-urilor candidate, pentru frecvență, este realizată paralel precum în figurile Figura 3.6 și Figura 3.7, dimensionarea fiind următoarea:

- dimensiunea 1 – pe această dimensiune sunt tratate itemset-urile candidate, care vor fi verificate pentru frecvență;
- dimensiunea 0 – pe această dimensiune sunt tratate grupurile de tranzacții, în care este împărțită baza de date, paralel.

După determinarea itemset-urilor frecvente de dimensiune 1, baza de date este supusă unei faze de pre-procesare, fază ce va fi detaliată ulterior.

3.3.3. Containerul datelor

În comunicarea hostCPU-workerGPU, host-ul trimite kernel-ul/task-ul și datele aferente acestuia, către GPU, pentru procesare. Datele trimise pot avea tipuri de date native (int, double, etc.) sau pot fi tablouri unidimensionale. Structurile complexe, sau tablourile bidimensionale și superioare, nu pot fi trimise către procesorul grafic în forma curentă. Tablourile de date trebuie transformate în tablouri unidimensionale, fie că sunt ele tablouri bidimensionale, tridimensionale, sau de grad mai mare.

Transformarea în tablouri unidimensionale nu reprezintă o problemă în sine, însă parsarea sau scanarea acestora, la procesorul grafic, pot întâmpina probleme dacă dimensiunile tablourilor unidimensionale, din tabloul bidimensional, diferă. Un tablou bidimensional, este format din mai multe tablouri unidimensionale. În implementarea propusă, aceste tablouri unidimensionale sunt reprezentate de liste, care în momentul transferului datelor vor fi transformate în array-uri (vectori). Baza de date este transpusă într-un tablou bidimensional, cu alte cuvinte, este o listă de liste, în care listele interioare reprezintă tranzacțiile, care pot varia în dimensiuni. Baza de date este trimisă sub forma unui tablou unidimensional, împreună cu încă două tablouri unidimensionale:

- tablou unidimensional ce conține indecșii de start pentru fiecare tranzacție în baza de date transpusă într-o singură dimensiune;
- tablou unidimensional ce conține lungimile tranzacțiilor.

Aceste tablouri sunt necesare pentru identificarea tranzacțiilor în parte, în momentul scanării lor, la nivelul procesorului grafic. Baza de date nu e singura structură care trebuie trimisă sub forma 1D. În cazul algoritmului Apriori, itemset-urile candidate sunt tablouri unidimensionale, toate având aceeași dimensiune k, reținute într-o listă, de unde și forma de tablou bidimensional. Lista de itemset-uri candidate trebuie transformată în tablou unidimensional și trimisă la procesorul grafic, însă, în cazul acesta, nu sunt necesare alte tablouri care să ajute la scanare, deoarece toate itemset-urile candidate au aceeași dimensiune k.

Datorită arhitecturii algoritmului Fast Item Miner, este necesară manipularea tablourilor tridimensionale. Cele trei dimensiuni sunt necesare în felul următor:

- o dimensiune pentru itemset-urile distribuite fiecărui nod;
- o dimensiune pentru nod-urile ce for fi atribuite item-ilor de lucru;
- o dimensiune pentru reținerea tuturor nod-urilor.

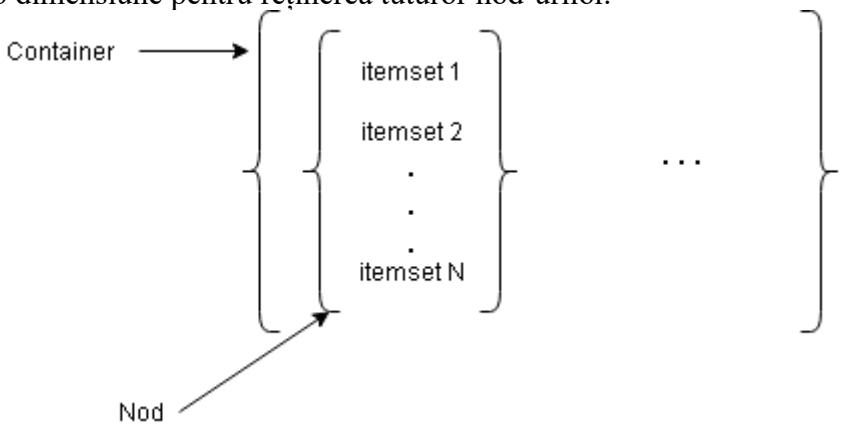

Figura 3.10. FIM - schema datelor principale utilizate.

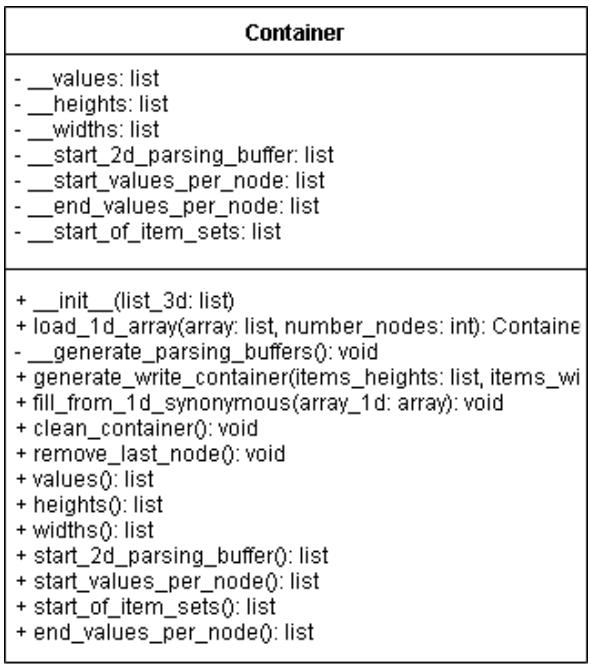

Figura 3.11. Clasa Container.

Procesarea unui tablou tridimensional este o sarcină dificilă, având în vedere faptul că structurile interioare au dimensiuni variabile, una față de alta, și de aceea sunt necesare tablouri suplimentare pentru suport. Cum tabloul tridimensional este transformat în unul unidimensional, tablourile suplimentare pot stoca diferite informații precum:

- lungimea unui itemset;
- numărul de itemset-uri dintr-un nod;
- indexul de start al unui nod;
- și alte informații considerate necesare.

În Figura 3.10, container-ul nu definește efectiv structura clasei „Container”, ci o listă care conține toate nodurile cu informațiile aferente. În clasa „Container”, structura prezentată în Figura 3.10 este stocată în atributul, de natură privată, „__values”. Clasa „Container”, a cărei structură este prezentată în Figura 3.11, oferă avantajul unei gestiuni ușoare a informațiilor, stocând tabloul de date tridimensional sub formă unidimensională, și odată cu asta, și tablourile necesare pentru procesarea datelor.

O funcție suplimentară a obiectelor de tip Container, este faptul că acestea pot genera noi obiecte de tip Container, necesare scrierii noilor informații obținute în urma procesării datelor containerului generator, pe baza aproximării a volumului maxim de date ce ar fi necesar de stocat.

3.3.4. Determinarea itemset-urilor frecvente

La momentul realizării acestei etape, baza de date este deja pre-procesată, în conformitate cu itemset-urile frecvente de dimensiune 1. Pre-procesarea bazei de date este procesul de eliminare a itemset-urilor, de dimensiune 1, care nu sunt frecvente, cu scopul obținerii unor scanări performante. Deci, baza de date o să fie deja redimensionată, conținând doar elementele frecvente, a căror suport va fi calculat pe baza numărului total de tranzacții inițial.

Etapa de determinare a itemset-urilor frecvente, itemset-uri cu dimensiunea mai mare decât 1, reprezintă algoritmul Fast Item Miner efectiv, care pleacă de la itemset-urile frecvente de dimensiune 1, pentru a obține rezultatele finale. Algoritmul Fast Item Miner nu generează candidați, după care îi verifică, cum e cazul algoritmului Apriori. Fapt ce poate fi considerat un avantaj, având în vedere că nu mai este necesară generarea apriori a posibilelor itemset-uri frecvente. Implementarea propune verificarea pe loc a itemset-urilor, și în caz că sunt frecvente acestea sunt reținute într-un tablou unidimensional, altfel nu.

Algoritmul Fast Item Miner parcurge un număr predefinit de pași pentru obținerea rezultatelor care este egal cu N-1, unde N reprezintă numărul de noduri. În construcția algoritmului, un nod este echivalent cu o unitate de procesare, fie aceasta proces sau fir de execuție. În cazul implementării propuse, datele sunt structurate după cum au fost explicate în 3.3.3, fiecare structură de tip nod fiind atribuită unui grup local de lucru, deci pentru algoritmul Fast Item Miner, s-a considerat ca și echivalent al unui nod, un grup de lucru.

Fiecare pas este realizat în stilul modelului hostCPU-workerGPU, astfel, host-ul trimite kernelul cu datele aferente către procesorul grafic pentru procesare, acesta din urmă returnând rezultatele obținute. La nivel de host, sunt gestionate informațiile primite și pe baza acestora se trece la următorul pas. Aceste acțiuni se realizează de un număr predefinit de ori determinat de host, numărul de noduri (grupuri de lucru) fiind egal cu numărul de itemset-uri frecvente inițiale. Algoritmul va executa numărul de pași integral, chiar dacă rezultatele finale sunt determinate la un pas anterior. Acest comportament se explică prin faptul că, toate itemset-urile frecvente găsite sunt purtate la dreapta, cu câte o poziție pentru fiecare pas, către nodul 0, de unde sunt preluate informațiile de către host. Host-ul va citi întotdeauna informațiile de pe nodul 0, salvându-le într-un dicționar, după dimensiunea itemset-urilor, în felul următor: un itemset frecvent de

dimensiune k, va fi salvat într-o listă stocată la cheia cu valoarea k. Astfel la finalul algoritmului vor fi returnate itemset-urile frecvente pentru k maxim, k maxim reprezentând cheia cu valoarea maximă dintre toate cheile dicționarului.

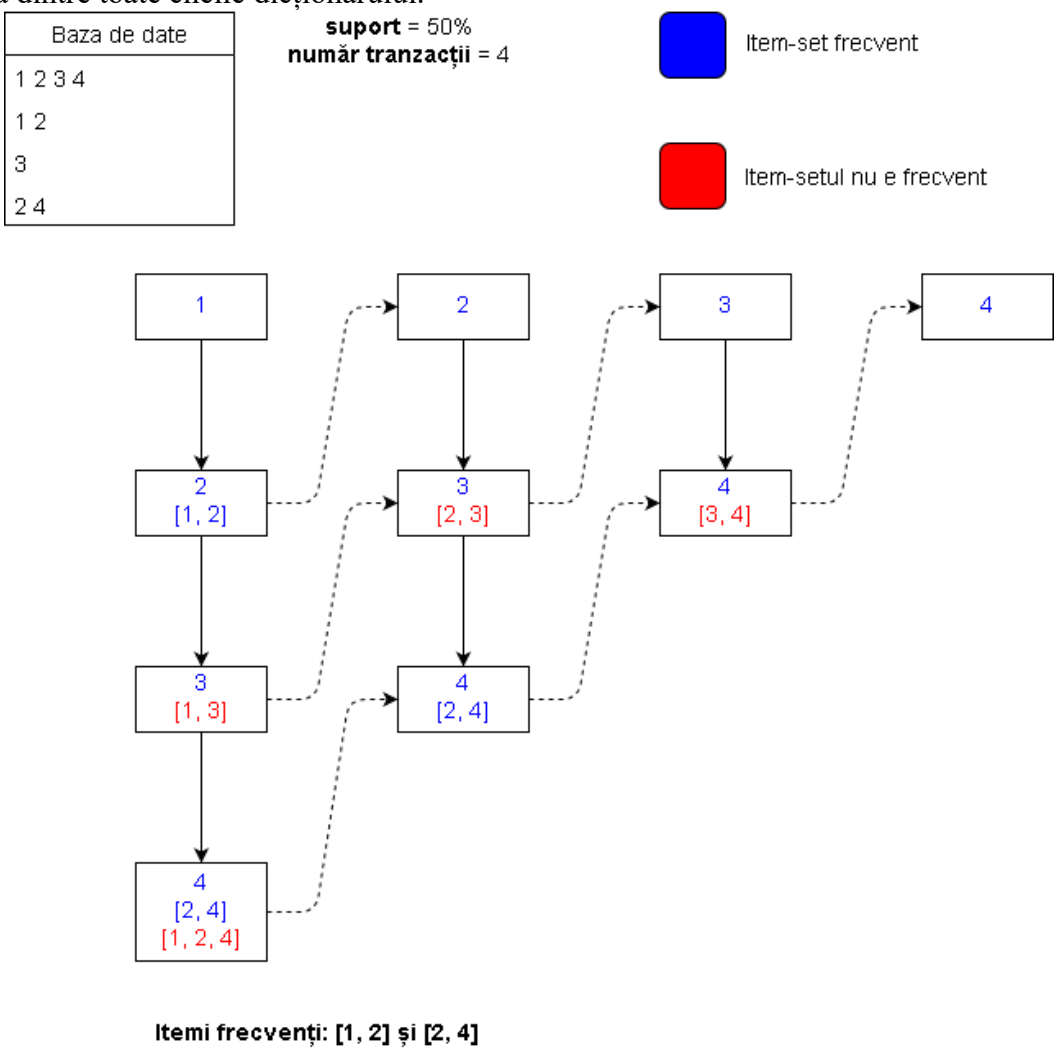

Figura 3.12. FIM - exemplu de funcționare.

În implementarea propusă, nodurile reprezintă structurile în care sunt stocate itemset-urile frecvente inițiale, iar pentru procesare sunt asignate N-1 grupuri de lucru, unde N reprezintă numărul de noduri. În exemplul din Figura 3.12, sunt patru itemset-uri frecvente de dimensiune 1, deci patru itemset-uri inițiale, care sunt stocate în patru noduri. Host-ul determină că sunt necesari trei pași pentru determinarea rezultatelor corecte, fiecare pas fiind realizat pe procesorul grafic, după modelul considerat. La fiecare pas este asignat un număr de grupuri de lucru, care este egal cu N-1, unde N este numărul de noduri ale structurii din care se citește. Se poate observa, din Figura 3.12, că itemset-urile frecvente inițiale, indiferent de dimensiunea lor, sunt purtate către nodul 0, cu fiecare pas.

Fiecare grup de lucru verifică în baza de date itemset-urile de la nodul curent, și acolo unde sunt găsite sunt verificate și itemset-urile de la nodul vecin din stânga. Dacă și acestea sunt găsite, atunci se contorizează, contorul aparținând itemset-ului format din itemset-ul aparținând nodului curent și itemset-ul aparținând nodului vecin. Baza de date este împărțită între mai mulți itemi de lucru pentru scanare, astfel fiecare item de lucru contorizează într-un tablou unidimensional de tip local memory shared, la o poziție proprie. Baza de date este scanată pentru

fiecare itemset inițial compus cu fiecare itemset frecvent din nodul vecin din stânga. Deci, baza de date va fi scanată de NxM ori, unde N este numărul de itemset-uri de la nodul curent, iar M este numărul de itemset-uri din nodul vecin din stânga.

La finalul fiecărei scanări, se realizează o reducere paralelă pentru obținerea contorului pe întreaga bază de date, după cum este reprezentat în Figura 3.5, contor pe baza căruia, după (3), se va determina suportul itemset-ului virtual, am putea spune, datorită faptului că acesta încă nu este efectiv salvat în forma finală, ci este doar o compunere fictivă între două itemset-uri. După determinarea suportului, cu condiția de la (4), se verifică dacă itemset-ul virtual este frecvent sau nu. Dacă itemset-ul este frecvent atunci acesta este salvat într-un tablou unidimensional, altfel se trece mai departe.

La nivel de GPU, paralelizarea se realizează în trei dimensiuni după cum urmează:

- dimensiunea 1 – este alcătuită din grupurile de lucru care tratează nodurile, asemuite cu unitățile de procesare despre care s-a menționat anterior, al căror număr este egal cu N-1, unde N este numărul de noduri (structuri ce conțin itemset-uri);
- dimensiunea 0 – fiecare grup de lucru din dimensiunea 1 este împărțit în mai multe grupuri care tratează itemset-urile inițiale, fiecărui astfel de grup fiind-ui atribuit un itemset;
- dimensiunea 2 – este reprezentată de itemii de lucru, care compun grupurile din dimensiunea 0, ce se ocupă cu scanarea bazei de date, ce este împărțită în grupuri de tranzacții, și contorizarea itemset-urilor.

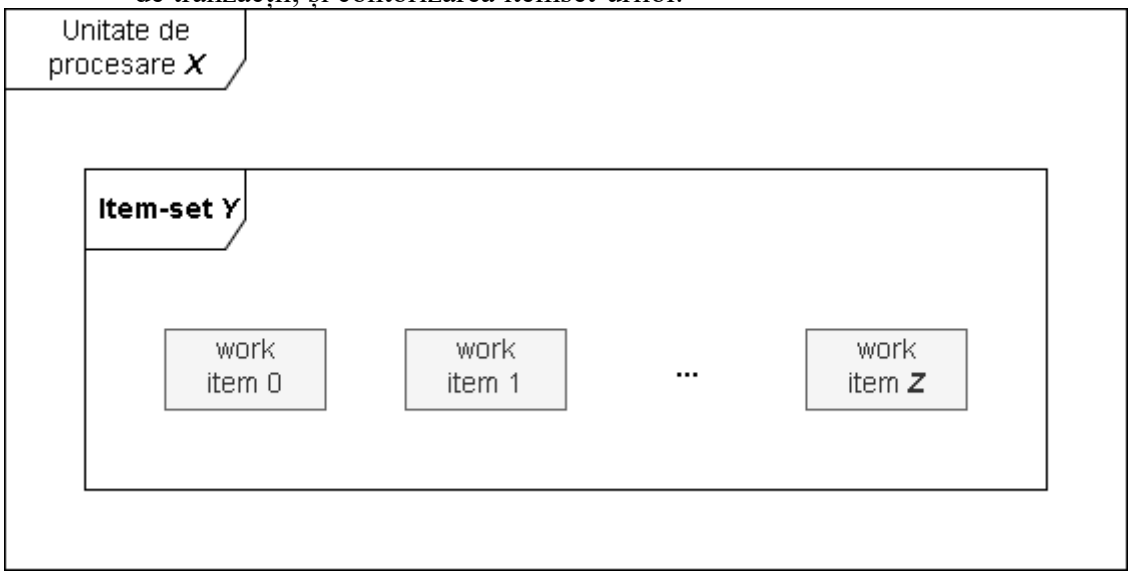

Figura 3.13. FIM - dimensionare locală.

Un grup de lucru, din dimensiunea 1, poate fi văzut ca în Figura 3.13, ce poate fi descrisă în felul următor: „Fiecare unitate de procesare tratează un nod, ce conține un singur itemset, care este tratat de un subgrup, a căror itemi de lucru procesează baza de date, împărțită în grupuri de tranzacții, pentru obținerea rezultatelor”. Un nod are un singur itemset, deoarece, la momentul realizării lucrării, algoritmul Fast Item Miner realizează procesarea pentru un număr de itemset-uri inițiale mai mic sau egal cu numărul de itemi de lucru de pe o dimensiune de paralelism. Implementarea abordează o manieră de paralelizare în trei dimensiuni pentru viitoare contribuiri ce vor „generaliza” algoritmul.

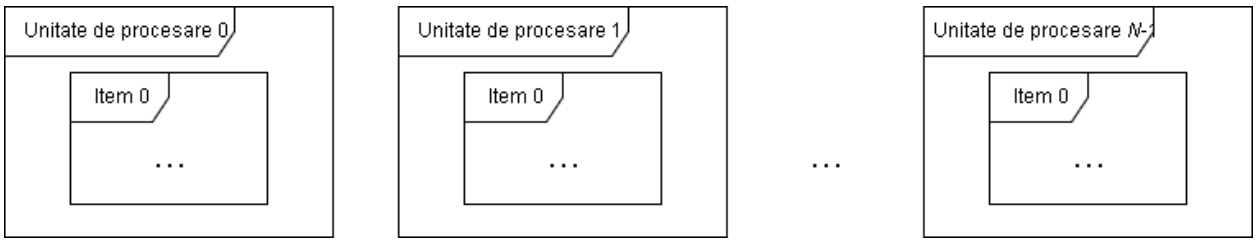

Figura 3.14. FIM - dimensionare globală.

Algoritmul Fast Item Miner nu generează candidați apriori, ci consideră un itemset virtual, itemset care nu există în memorie, ci doar în logica implementării la un moment dat, pe care îl verifică în baza de date pentru a se determina dacă este frecvent sau nu. Dacă itemset-ul virtual este frecvent, atunci acesta este stocat în memorie, într-un tablou unidimensional. Datorită acestui mod de lucru, trebuie parcurși toți pașii, fără vreo excepție, astfel, adesea pot exista cazuri în care algoritmul Apriori poate necesita mai puțini pași față de algoritmul Fast Item Miner, pentru obținerea rezultatelor.

3.3.5. Optimizări

Procesorul grafic oferă avantajul unui grad de paralelism ridicat, cu mult mai mare față de CPU, însă odată cu acesta vin și anumite limitări precum:

- dimensiunea memoriei împărtășită între elementele de lucru (local memory);
- manipularea memoriei prin instrucțiunea „malloc” nu este oferită;
- comunicarea între elementele de lucru se realizează în principiu prin intermediul memorie locale care este „shared” doar între elementele de lucru ale unui grup, fiind incluse „pipe-urile” odată cu OpenCL 2.0;
- viteza de acces la memorie având cele patru tipuri de memorie: globală, constantă, locală, privată;
- nu există importuri.

Toate aceste constrângeri trebuie luate în calcul în momentul dezvoltării kernel-ului pentru a obține rezultatele dorite. Pentru a îmbunătăți rezultatele, s-au luat în considerare anumite optimizări la nivel de cod, instrucțiuni, pentru a reduce numărul de instrucțiuni parcurse, în anumite cazuri, și totodată pentru a economisi timp de execuție. În continuare vor fi prezentate optimizările implementate care au crescut performanța și au adus beneficii în privința timpului necesar obținerii rezultatelor.

3.3.5.1. Căutarea

O căutare simplă, realizată doar prin instrucțiuni repetitive, fără vreun artificiu, ar oferi o complexitate $O(n^2)$, complexitate care atrage totodată și un timp mai mare de execuție, mai ales în cazul unui tablou de dimensiune mare. Cum o tranzacție este tratată ca un tablou unidimensional, și dimensiunile acesteia pot varia până la unele foarte mari, s-a luat în considerare utilizarea unui algoritm de căutare care să ofere o complexitate timp mult mai mică. O idee ar fi utilizarea algoritmului de „căutare binară” cu complexitatea O(log n), dar având în vedere că tranzacțiile sunt sortate, o idee mai bună a fost considerată căutarea prin „interpolare” cu complexitatea O(log log n).

Căutarea unui itemset candidat, în baza de date, presupune cătarea tuturor elementelor itemset-ului, în fiecare tranzacție, unul câte unul. Un itemset candidat este considerat găsit în tranzacție dacă toate elementele sale au fost găsite în acea tranzacție, fără excepție.

Pentru a îmbunătăți viteza de determinare a numărului de apariții a itemset-ului candidat în baza de date, s-a considerat că dacă un element dintr-un itemset nu este găsit, într-o tranzacție,

să se oprească căutarea acelui itemset în tranzacția în cauză. Astfel sunt evitate executarea unor instrucțiuni care nu aduc nici un beneficiu.

3.3.5.2. Sortarea

Pentru ca complexitatea timp a algoritmilor de căutare să fie cât mai mică, aceștia trebuie să realizeze căutarea în condițiile cele mai avantajoase, cu alte cuvinte, căutarea trebuie realizată în tranzacții sortate crescător. Pentru algoritmul căutării prin interpolare, în cel mai bun caz obținem o complexitate de timp O(log log n). Marele avantaj al acestei căutări este că dacă elementul căutat nu are valori între elementul de început al tranzacției și elementul de final, nu se mai realizează căutarea pe tranzacția respectivă, ceea ce salvează timp. Pentru ca acest lucru să funcționeze tranzacția trebuie sortată crescător.

Sortarea tranzacțiilor din baza de date este realizată în momentul în care aceasta este atribuită obiectului de tip Apriori. Asupra acesteia se rulează verificări, automat, pentru verificarea corectitudinii tranzacțiilor și totodată este realizată și sortarea. Sortarea este realizată prin metoda built-in a mediului de dezvoltare pentru host, Python, de sortare a unui liste.

3.3.5.3. Eliminarea elementelor care nu sunt frecvente în faza de pre-procesare

Scanarea bazei de date poate necesita un interval de timp mare chiar și în cazul unei căutări optimizate datorită dimensiunilor mari. De aceea, după aflarea itemset-urilor frecvente de dimensiune 1, se realizează o parsare a bazei de date cu scopul de a elimina elementele din tranzacții care nu sunt frecvente. Această eliminare poate reduce dimensiunile bazei de date consistent, în anumite cazuri, astfel următoarele scanări vor fi realizate pe dimensiuni mai mici față de cele inițiale. Eliminând elementele ce nu sunt frecvente putem obține tranzacții de dimensiuni mai reduse, evitând scanarea unor elemente care sigur nu vor aduce nici un rezultat. În felul acesta este redusă dimensiunea bazei de date în lățime. Uneori, eliminarea elementelor ce nu sunt frecvente poate reduce o tranzacție la dimensiunea 0, caz în care aceasta este eliminată. Astfel, dimensiunea bazei de date este redusă în înălțime.

Această pre-procesare a bazei de date este realizată o singură dată, în cazul algoritmului Fast Item Miner este realizată între etapa de identificare a itemset-urilor inițiale și etapa de minare, deoarece, în cazul bazelor de date de mari dimensiuni, efectuarea multiplă a acesteia poate atrage cost suplimentar din punct de vedere al timpului necesar obținerii rezultatelor.

3.3.5.4. Reducerea numărului de scanări a bazei de date

S-a observat faptul că itemset-urile din noduri sunt așezate de la itemset-urile de dimensiune mică la itemset-urile de dimensiune mare, vezi Figura 3.15.

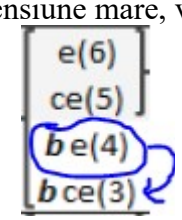

Figura 3.15.
Exemplu.

Odată cu această observare s-a realizat faptul că sunt itemset-uri de dimensiune mai mică care sunt subseturi ale itemset-urilor de dimensiune mai mare, astfel se poate sări peste scanarea acestor itemset-uri mai mici în cazul în care itemset-ul de dimensiune mai mare este frecvent. Toate subseturile dintr-un itemset frecvent sunt itemset-uri frecvente. Implementarea propune parcurgerea tabloului în ordine inversă, și verificarea itemset-urilor ce urmează unui itemset

frecvent pentru a verifica dacă sunt subseturi din cel frecvent. Dacă se găsesc subseturi ale itemset-ului frecvent atunci se consideră și acestea frecvente și nu se mai scanează baza de date. Totuși, această verificare nu este optimă deoarece se oprește când se întâlnește un itemset care nu este subset al itemset-ului frecvent.

S-a observat obținerea de performanță pentru bazele de date dese, în care se întâlnesc itemset-uri de dimensiune mare.

3.3.6. Condiții de stop

Algoritmul Fast Item Miner nu necesită o condiție de stop specifică, o verificare a unei validități, acesta parcurgând un număr de pași, predeterminat și necesar, pentru determinarea rezultatelor. După fiecare pas executat, host-ul salvează noile itemset-uri primite într-un dicționar, în forma cheie-valoare, unde valoarea este o listă ce stochează itemset-uri, iar cheia este dimensiunea itemset-urilor stocate în listă. La fiecare pas, host-ul poate primi itemset-uri de dimensiuni diferite, întotdeauna la nodul 0, de unde se face citirea.

Rezultatele returnate sunt reprezentate de itemset-urile cu dimensiunea maximă din dicționar, deci se identifică cheia cu valoarea ca mai mare și se returnează lista stocată, la cheia respectivă. Algoritmul se poate opri înainte de terminarea pașilor din eventuale erori, ce pot fi reprezentate de depășiri de memorie, în special în cazul memoriei globale dacă procesorul grafic nu îndeplinește un minim de performanță, sau un eventual comportament haotic al procesorului grafic.

3.3.7. Limitări

Implementarea algoritmului Fast Item Miner, la momentul scrierii lucrării, este limitată de numărul de itemi de lucru de pe o dimensiune de paralelism, 1024 în cazul platformei de test. Deci, rezultatele vor fi corecte pentru 1024 de itemset-uri inițiale, deoarece, deși implementarea abordează algoritmul din punctul de vedere al algoritmului generalizat, la momentul menționat, implementarea rezolvă doar cazul particular, urmând, în viitor, generalizarea algoritmului.

Totodată, datorită modului de dimensionare a datelor, tridimensional, ce necesită mai multe tablouri pentru suport în procesarea bazei de date, este necesară mai multă memorie globală, decât în cazul implementării algoritmului Apriori.

3.3.8. Viitoare update-uri

Pe viitor se dorește generalizarea algoritmului Fast Item Miner, prin realizarea etapei de „expansiune” a itemset-urilor frecvente inițiale, astfel încât să nu mai existe limitarea în funcție de numărul de itemi de lucru de pe o dimensiune de paralelism.

Se dorește eficientizarea reducerii numărului de scanări a bazei de date, astfel încât să fie verificate toate itemset-urile ce sunt subseturi ale unui itemset frecvent, și nu doar o parte din ei, prin utilizarea unor tablouri unidimensionale de tip local memory shared.

O altă idee ar fi păstrarea numărului de grupuri de lucru pe parcursul tuturor pașilor, fără a le reduce la fiecare pas, și folosirea grupurilor suplimentare, care cu parcurgerea pașilor rămân libere, în ajutorul grupurilor de lucru care au mult de procesat, împărțind astfel munca folosind o metodă de comunicare similară cu cea a hipercubului, în speranța obținerii unui timp de execuție mai mic.

Capitolul 4. Rezultate experimentale

4.1. Rezultate generale

Pentru a pune în evidență performanța obținută prin implementările GPGPU, din punct de vedere al timpului, a fost implementat algoritmul Apriori secvențial, fără nici un fir de execuție, în Python 3.6, utilizând funcții specifice și optimizări, astfel încât să avem o implementare performantă. Rezultatele acesteia din urmă, au fost puse în Tabelul 4.1, împreună cu rezultatele implementărilor GPGPU a algoritmilor Apriori și Fast Item Miner. Implementările au fost testate pentru fiecare bază de date de test considerată, pentru un suport mic.

Baza de date și suportul	Timp rulare pentru Apriori secvențial (s)	Timp rulare pentru Apriori GPGPU (s)	Timp rulare pentru FIM GPGPU (s)
retail.dat 2.2%	875.59	5.05	5.51
T40I10D100K.dat 3.6%	7221.54	61.74	69.10
chess.dat 82%	47.28	3.5	2.7
mushroom.dat 40%	16.51	0.84	1.07

Tabelul 4.1. Rezultate generale.

După cum se poate observa, din Tabelul 4.1, implementările GPGPU au obținut timpi de execuție de peste 100 de ori mai mici față de rezultatele obținute de implementarea secvențială a algoritmului Apriori, în cazul bazelor de date cu un număr mare de tranzacțiile. Diferența între rezultate scade pentru bazele de date cu un număr de tranzacții mai mic, deci putem trage concluzia că diferența între rezultatele implementării secvențiale pentru algoritmul Apriori și rezultatele implementărilor GPGPU este direct proporțională cu numărul de tranzacții. Rezultatele nu depind numai de numărul de tranzacții dintr-o bază de date, ci și de numărul de elemente distincte din baza de date, dimensiunile tranzacțiilor și dispersia elementelor în baza de date. Totuși, se observă performanța semnificativă obținută pentru baze de date mari în privința timpilor de execuție, problemă care s-a propus pentru rezolvat.

Bazele de date pe care s-au efectuat testele și studiul de caz prezintă caracteristicile prezentate în

Nume	Tip	Număr elemente	Număr tranzacții
retail.dat	rară	16470	88162
T40I10D100K.dat	rară	~1000	100000
chess.dat	densă	~75	3196
mushroom.dat	densă	~119	8124
SC_DB.txt	rară	1559	4141

Bazele de date de test au fost descărcate de la adresa http://fimi.uantwerpen.be/data/, întâlnindu-se următoarele două tipuri: rare și dense.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32
33 34 35
36 37 38 39 40 41 42 43 44 45 46
38 39 47 48
38 39 48 49 50 51 52 53 54 55 56 57 58
32 41 59 60 61 62
```

Figura 4.1. Porțiune dintr-o bază de date rară.

Bazele de date rare, Figura 4.1, sunt caracterizate printr-un număr mare de tranzacții, de dimensiuni diferite, număr mare de elemente distincte și dispersie mică a acestora, drept rezultat, itemset-urile frecvente au în general dimensiune relativ mică.

```
1 3 9 13 23 25 34 36 38 40 52 54 59 63 67 76 85 86 90 93 98 107 113
2 3 9 14 23 26 34 36 39 40 52 55 59 63 67 76 85 86 90 93 99 108 114
2 4 9 15 23 27 34 36 39 41 52 55 59 63 67 76 85 86 90 93 99 108 115
1 3 10 15 23 25 34 36 38 41 52 54 59 63 67 76 85 86 90 93 98 107 113
2 3 9 16 24 28 34 37 39 40 53 54 59 63 67 76 85 86 90 94 99 109 114
2 3 10 14 23 26 34 36 39 41 52 55 59 63 67 76 85 86 90 93 98 108 114
2 4 9 15 23 26 34 36 39 42 52 55 59 63 67 76 85 86 90 93 98 108 115
```

Figura 4.2. Porțiune dintr-o bază de date densă.

Bazele de date dense, Figura 4.2, sunt caracterizate printr-un număr mai mic de tranzacții, de dimensiuni egale, față de bazele de date de tip rar, număr de elemente distincte mai mic și o dispersie mare a acestora, o tranzacție fiind compusă din majoritatea elementelor distincte. Astfel itemset-urile frecvente rezultate au în general dimensiuni mari.

4.2. Rezultate pe baze de date rare

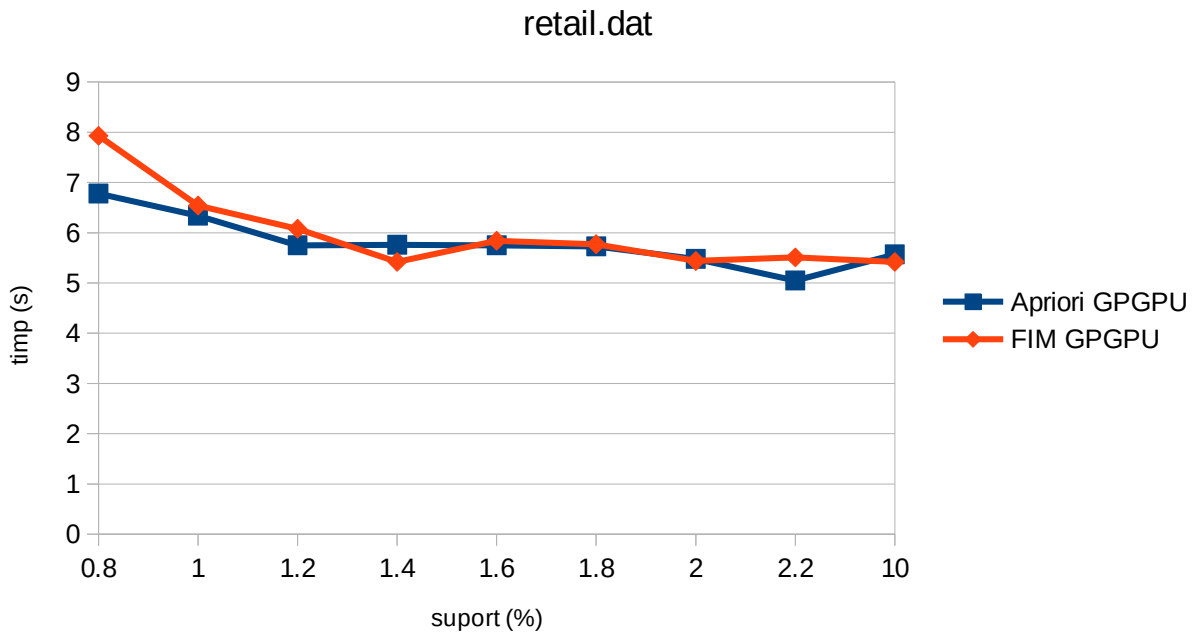

Figura 4.3. Grafic retail.dat

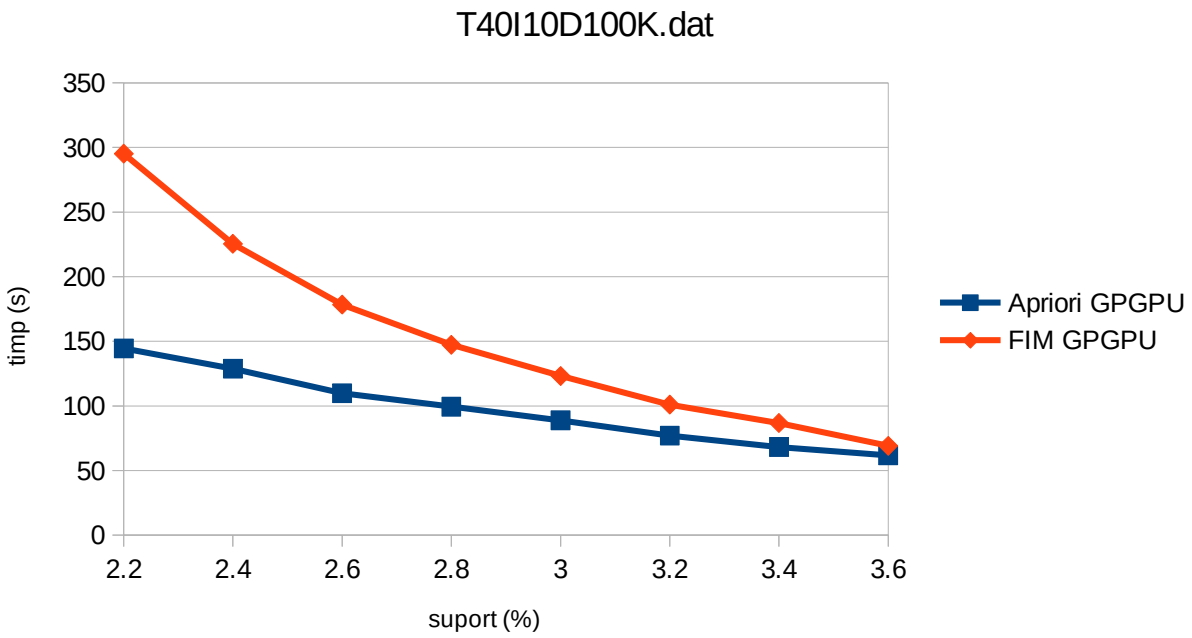

Figura 4.4. Grafic T40I10D100K.dat

4.3. Rezultate pe baze de date dense

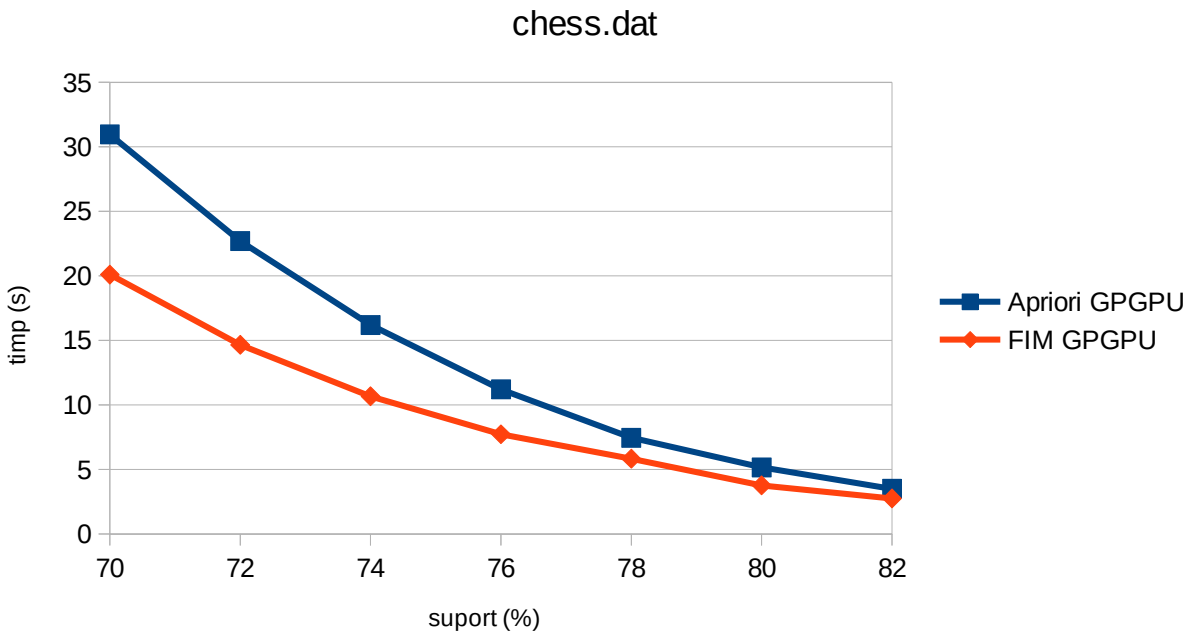

Figura 4.5. Grafic chess.dat

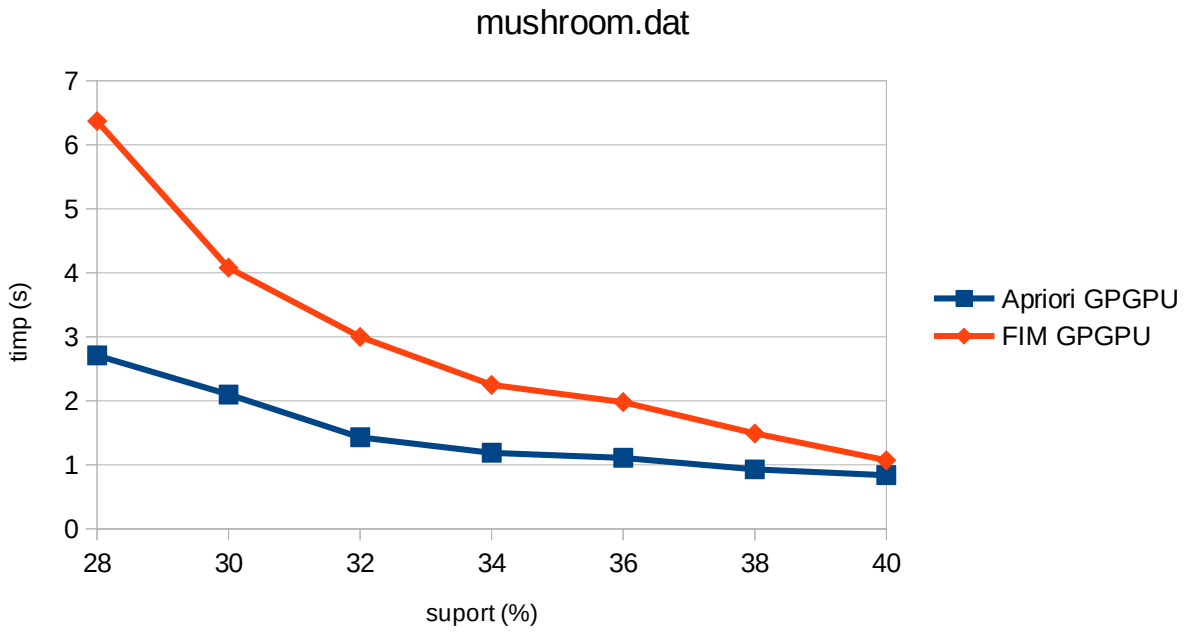

Figura 4.6. Grafic mushroom.dat

4.4. Studiu de caz

Implementările GPGPU, a algoritmilor Apriori și fast Item Miner, au fost puse într-un pachet Python denumit „pyDMLGPU”, ce reprezintă acronimul pentru „Python Data Mining Library GPU”. Pachetul este instalabil prin comanda „pip install pyDMLGPU.whl”, după care poate fi accesat din mediul Python pentru dezvoltarea de aplicații/programe destinate domeniului de data mining.

Pentru a pune în aplicare librăria de algoritmi paraleli, implementați pentru GPU, de descoperire a pattern-urilor frecvente, a fost realizată o aplicație, denumită „Data Mining Tool”, ce folosește implementările GPGPU pentru descoperirea și afișarea elementelor frecvente (de exemplu produse) într-o bază de date, în urma interacțiunii cu utilizatorul. Aplicația este proiectată pentru a fi folosită în orice domeniu în care se pretează, și nu doar pentru anumite domenii precum coșul de cumpărături.

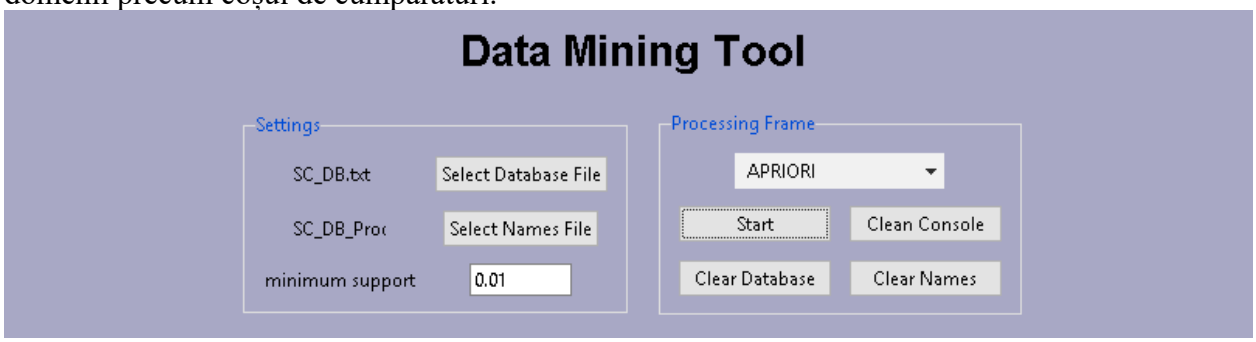

Figura 4.7. Data Mining Tool - Meniu utilizare

Este necesară selectarea bazei de date ce conține tranzacțiile de procesat, obligatoriu, și a bazei de date ce conține asocierile nume – cod. Codul este numărul care apare în tranzacții în loc

de denumirea produsului. După precizarea suportului minim, care este o valoare în procente, și selectarea bazelor de date, se poate realiza procesul de determinare a pattern-urilor frecvente, în urma căruia vor fi afișate rezultatele în fereastra dedicată din aplicație.

La obținerea rezultatelor, dacă există baza de date aferentă asocierilor nume – cod, atunci este realizată o afișare a numelor în funcție de codurile din rezultate, altfel sunt afișate itemset-urile frecvente în forma în care apar în rezultate.

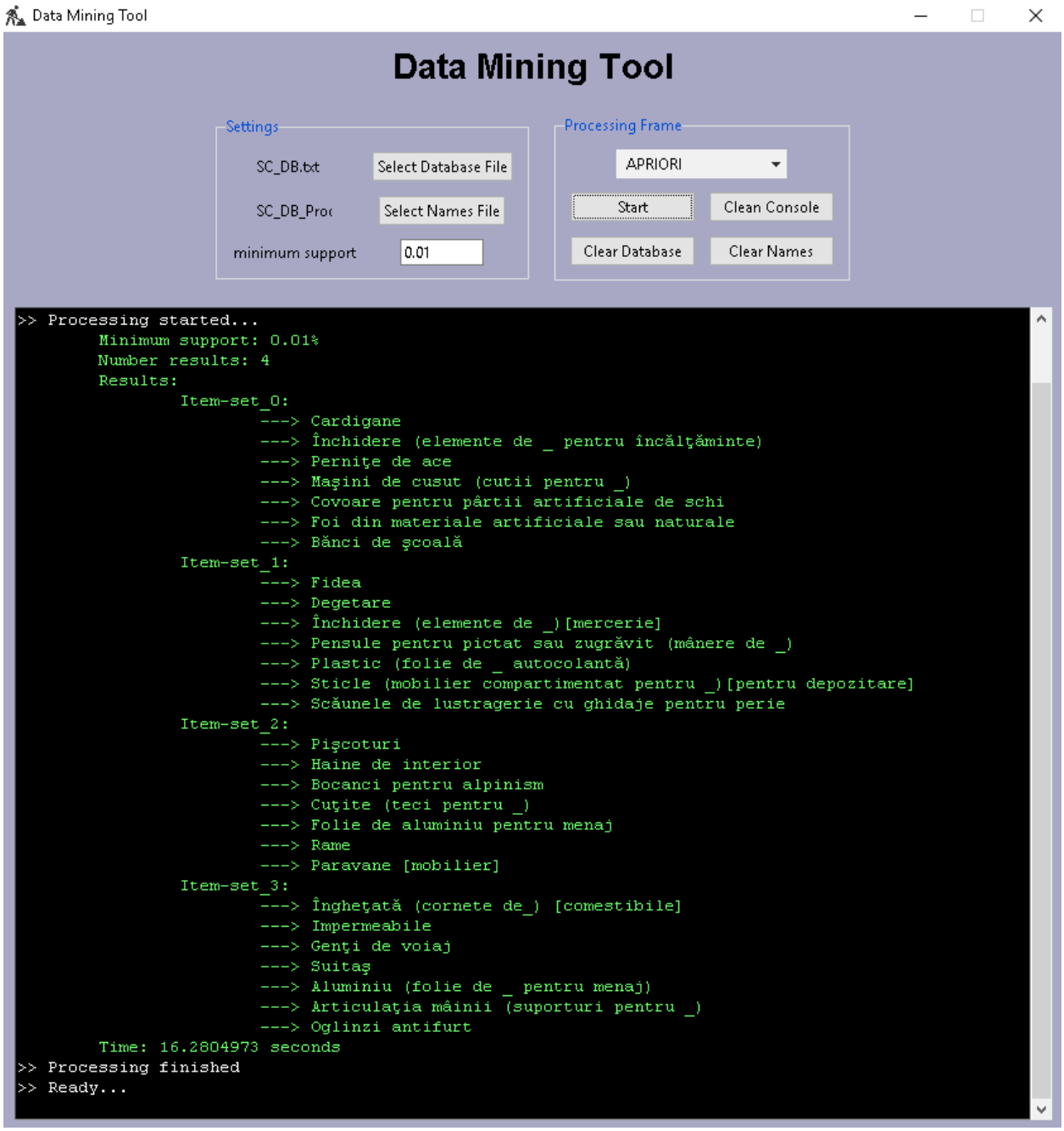

Figura 4.8. Data Mining Tool - Exemplu descoperire pattern-uri frecvente.

Testul din studiul de caz, din Figura 4.8, au fost realizate pe baza de date „SC_BD.txt”, ce conține 4141 de tranzacții aferente clienților unui magazin, bază de date care reprezintă istoricul de vânzări al unui magazin, și pe baza de date „SC_DB_Products.txt”, ce conține asocierile nume – cod realizate fictiv pentru a demonstra scopul aplicației. Deci considerăm

istoricul de vânzări al unui magazin și dorim să descoperim pattern-urile frecvente, ce apar în istoric, compuse din produse.

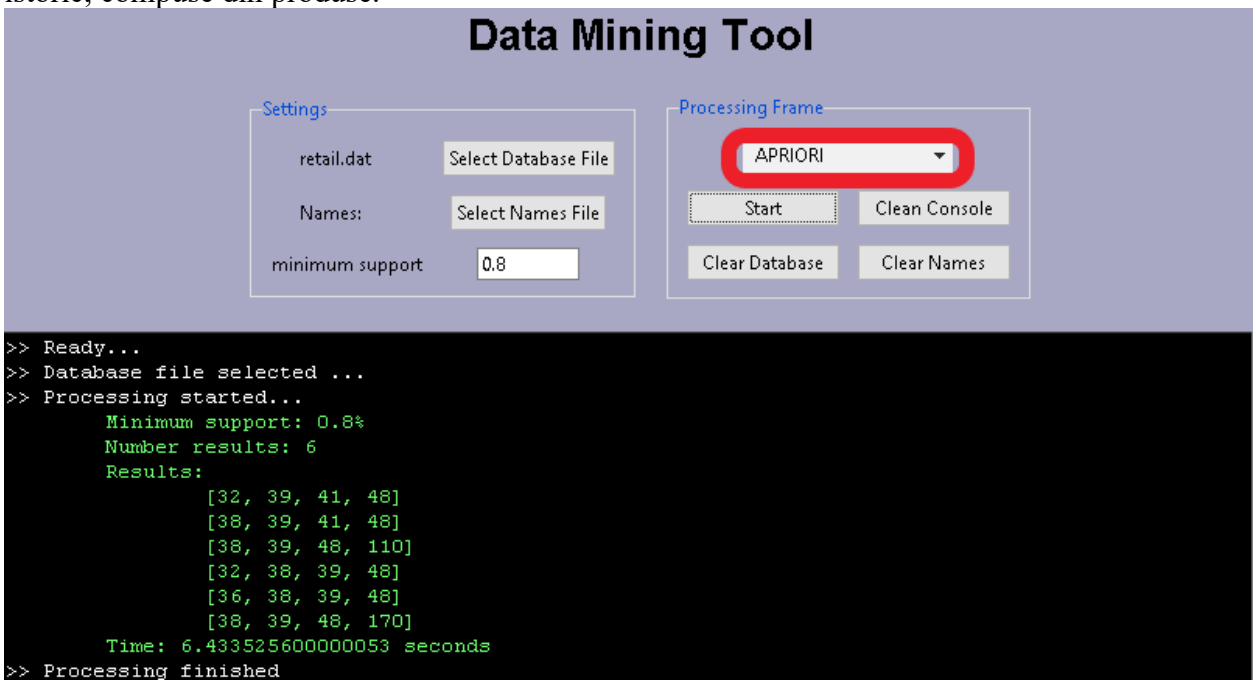

Figura 4.9. Data Mining Tool - Exemplu Apriori.

În Figura 4.9 și Figura 4.10 baza de date cu asocierile nume – cod nu este selectată, și de aceea rezultatele sunt afișate direct, utilizându-se codurile din baza de date.

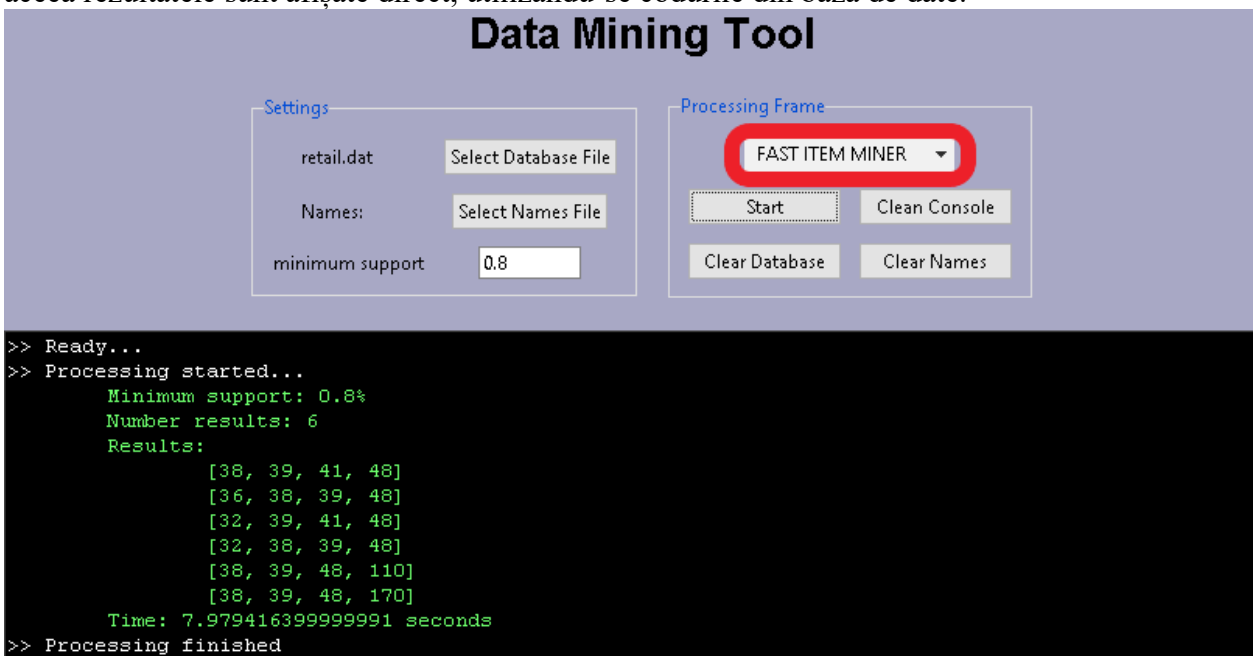

Figura 4.10. Data Mining Tool - Exemplu Fast Item Miner.

Pe baza pattern-urilor descoperite cu o astfel de aplicație, o anumită unitate, fie ea economică sau nu, poate implementa reguli de afaceri sau decizii la nivel de management care pot aduce diferite avantaje, precum beneficiu economic, distribuire de resurse eficientă, evitarea pierderilor datorită eventualelor fraude, în cazul băncilor, și altele.

Concluzii

Dimensiunile bazelor de date cresc de la o zi la alta, ceea ce face procesarea lor din ce în ce mai greoaie și mai costisitoare din punct de vedere al timpului. Din acest motiv se caută constant noi metode care să ofere rezultatele procesării bazelor de date într-un timp mai mic.

Numărul mare de nuclee al procesoarelor grafice și specificațiile OpenCL oferă un grad de paralelism mult mai ridicat, ceea ce poate duce la obținerea unor timpi de execuție mult mai mici, dacă implementarea a fost abordată corect. Trebuie avut în vedere faptul că la nivelul procesorului grafic nu este posibilă execuția integrală a unuia dintre algoritmii propuși, datorită imposibilității la momentul actual de a manipula memoria, din punct de vedere structural. În principiu host-ul trimite datele către GPU în format unidimensional, gestionarea memoriei, precum restructurarea, redimensionarea, fiind realizată la host. Totodată, trebuie luate în calcul resursele de memorie în proiectarea unei implementări GPGPU, în special memoria locală a cărei dimensiuni este foarte mică în comparație cu memoria globală, cât și numărul de elemente de lucru total care pot fi utilizate.

Implementările GPGPU se pot dovedi dificil de realizat, mai ales în cazul algoritmilor dificili, care trebuie abordați în așa fel încât să fie fezabili pentru implementarea lor la nivelul procesorului grafic. Totuși, gradul de paralelism pus la dispoziție de procesoarele grafice, din ce în ce mai mare cu fiecare model nou, oferă timpi de execuție formidabil de mici în comparație cu aceleași implementări realizate secvențial pe GPU, luând în calcul faptul că implementările GPGPU necesită pe partea de host prelucrări ale datelor în avans.

Se observă, din capitolul „Rezultate experimentale”, faptul că implementările GPGPU sunt mult mai performante, din punct de vedere al timpului, față de implementarea secvențială cu care au fost comparate. Gradul de performanță obținut depinde de baza de date pe care se realizează testele, și anume, dimensiunea și numărul de elemente distincte ce apar în baza de date.

Bibliografie

[1] Necunoscut, Data Mining Explained [Online], Disponibil la adresa: https://www.microstrategy.com/us/resources/introductory-guides/data-mining-explained, Accesat: 2019.

[2] R. Agrawal, R. Srikant, „Fast Algorithms for Mining Association Rules in Large Databases", Morgan Kaufmann Publishers Inc. San Francisco, CA, USA ©1994 , Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487-499, 1994.

[3] P.N. Tan, M. Steinbach, V. Kumar, „Introduction to data mining", Addison Wesley, 2005.

[4] Cristian Nicolae Butincu, Mitica Craus, „An improved version of the frequent itemset mining algorithm", Proceedings of the 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), IEEE, pp. , 2015.

[5] Mitica Craus, „A New Parallel Algorithm for the Frequent Itemset Mining Problem", Proceedings of the 2008 International Symposium on Parallel and Distributed Computing, IEEE, pp. , 2008.

[6] Khronos Group, OpenCL Overview [Online], Disponibil la adresa: https://www.khronos.org/opencl/, Accesat: 2019.

[7] Gastón Hillar, Easy OpenCL with Python [Online], Disponibil la adresa: http://www.drdobbs.com/open-source/easy-opencl-with-python/240162614, Accesat: 2013.

Anexe.

Anexa 1.