VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

GPU panaudojimas ultra greitam įvykių apdorojimui

Ultra - fast complex event processing using a GPU

Magistro baigiamasis darbas

Atliko: Marius Balčaitis

Darbo vadovas: docentas dr. Antanas Lenkevičius

Recenzentas: Denis Lebedenko

Vilnius – 2017

Santrauka

Kadangi sudėtingų įvykių apdorojimo procesas pasižymi savybe atlikti daug tų pačių operacijų skirtingiems duomenims, tai darbe yra nagrinėjama galimybė pritaikyti grafinius procesorius ultra sparčiam primityvių įvykių apdorojimui, t. y. apdorojimui su kuo mažesne delsa tarp šakninio primityvaus įvykio ir jo sugeneruotų sudėtingų įvykių gavimo. Darbe pristatomi grafinio procesoriaus architektūriniai skirtumai lyginant su centriniu procesoriumi, pateikiamos įvykių apdorojimo sistemos, skirtos vykdymui su grafiniu procesoriumi, realizacijos detalės bei iššūkiai, analizuojant gautus eksperimentinio tyrimo rezultatus, pristatoma kokioms sąlygoms esant geriausia naudoti grafinio procesoriaus vykdomą sistemą, kokio apdorojimo spartos padidėjimo galima tikėtis priklausomai nuo sudėtingus įvykius aprašančių taisyklių skirtingų parametrų reikšmių, bei pateikiamos rekomendacijos kaip su grafiniais procesoriais reikėtų siekti didesnės įvykių apdorojimo spartos.

Summary

Complex event processing systems have a tendency to do same operations for multiple different data, so in this thesis we are investigating possibility to use Graphics Processing Units (GPUs) for ultra-fast event processing, i.e. for processing with the least possible delay between receiving a root primitive event and obtaining complex events that are generated by the former primitive event. In this research we are presenting differences between GPU and CPU architectures, implementation details of complex event processing system designed to be run on GPUs. Also in the thesis we are analyzing our testing results, presenting cases in which the use of GPUs for complex event processing is most beneficial and providing recommendations and insights on ways to achieve greater processing performance while using GPUs.

Turinys

Įvadas ... 6
Temos aktualumas bei naujumas ... 6
Darbo tikslas ... 10
Uždaviniai ... 11
Laukiami rezultatai ... 11
1. Analitinė dalis ... 12
1.1. Sudėtingų įvykių apdorojimo sistemos apibrėžimas ... 12
1.1.1. Tęstinių užklausų duomenų srautams sistemos apibrėžimas ... 12
1.1.2. Trigeriai ... 17
1.1.3. Sudėtingų įvykių apdorojimo sistemos (CEP) modeliavimas ... 18
1.2. CPU ir GPU architektūrų skirtumai ... 19
1.2.1. CPU fizinė architektūra ... 19
1.2.2. GPU fizinė architektūra ... 20
1.2.3. CPU ir GPU resursų palyginimas ... 24
1.3. OpenCL modelis ... 25
1.3.1. Branduolio funkcija ... 25
1.3.2. Darbo vienetas ir darbo vienetų grupės ... 26
1.3.3. Darbo vienetų grupės ir bangos frontai ... 26
1.3.4. Darbo vienetų sinchronizacija ... 26
1.3.5. Atminties modelis ... 27
1.3.6. Tipinis programų vykdymo šablonas ... 27
1.4. Sudėtingų įvykio apdorojimo sistemos realizacija ... 28
1.4.1. Kalbos modelis trigeriams apibrėžti ... 28
1.4.2. Įvesties srauto sekų skaidymas ... 29
1.4.3. CEP apdorojimo algoritmai ... 30
1.4.3.1. Nedeterminuotais baigtiniais automatais paremtas apdorojimas ... 30
1.4.3.2. Stulpeliais paremtas uždelstas apdorojimas ... 32
1.5. Skyriaus išvados ... 33
2. Tiriamoji dalis ... 34
2.1. GPU modelio realizacija ... 35
2.1.1. GPU atminties valdymas ... 36
2.1.2. Primityvių įvykių procesorius ... 37
2.2. Eksperimentinio tyrimo rezultatai ... 54
2.2.1. Taisyklių skaičius ... 56

2.2.2. Primityvių įvykių skaičius 57
2.2.3. Atributų skaičius 58
2.2.4. Parametrizuotų atributų skaičius 59
2.2.5. Agregatinių atributų skaičius 60
2.2.6. Langų dydis 61
2.2.7. Daugelio išrinkimo primityvių įvykių stulpelių skaičius 62
2.3. Skyriaus išvados 63
3. Išvados ir pasiūlymai 64
Šaltinių sąrašas 65
Santrumpos ir sąvokų apibrėžimai 68

Įvadas

Temos aktualumas bei naujumas

Kompiuterių ir interneto paplitimas suteikė žmonėms galimybę daug greičiau ir produktyviau atlikti įvairius skaičiavimus, perduoti informaciją tolimais atstumais, naudojant įvairius sensorius, aptikti pokyčius aplinkoje ar stebėti kaip kinta kažkokio objekto būseną nusakantys atributai. Taip pat didėjant kompiuterių skaičiavimo pajėgumams bei sudėtingėjant programų sistemoms, atsiranda galimybės realizuoti sistemas, leidžiančias aptarnauti daug užklausų vienu metu: ar tai būtų bankininkystės sistema, ar prekyba vertybinių popierių biržoje, ar tiesiog elektroninė parduotuvė. Vienas iš svarbiausių šių naujų galimybių padarinių yra tas, jog per labai trumpą laiką didelės sistemos gali sugeneruoti labai daug naujų duomenų ar labai sparčiai keisti sistemos būseną, o tai savo ruožtu yra iššūkis, nes tinkamai ir laiku nesureagavus į pokyčius gali būti patiriami dideli nuostoliai.

Tam, kad laiku būtų sureaguojama į svarbius įvykius, reikia sugebėti tuos įvykius prognozuojančius atributus atrinkti, išanalizuoti ir atpažinti iš įvairių informacijos šaltinių gaunamame informacijos sraute. Kad būtų lengviau išgauti informaciją apie vykstančius procesus ar būsenas įvairiose sistemose ar jų dalyse, didžioji dauguma jų yra projektuojamos kaip įvykių valdomos sistemos, kuriose signalai gaunami iš fizinių įrenginių, kitų programų sistemų, tinklo servisų, vartotojo veiksmų ar kitų šaltinių yra interpretuojami kaip nauji įvykiai, kurie yra perduodami aukštesnio lygio sistemos dalims tolesniam apdorojimui, taip suformuojant įvykių apdorojimo hierarchijas, kurių aukščiausiame lygyje generuojami įvykiai apibūdina jau ne techninius fizinių įrenginių ar programų sistemos lygio gautus signalus, o aptarnaujamos srities lygio įvykius, kurie yra aktualūs ir kuriuos gali suprasti tos srities specialistas [Luc01]. Tačiau žemiausiame lygyje iš šaltinių gaunami įvykiai yra tiesiog primityvios duomenų struktūros, dažniausiai turinčios vyksmo laiko žymę, keletą atributų apibūdinančių pastarąjį bei susietos su kitais tokiais įvykiais laiko, priežastingumo ir susietumo ryšiais [Luc01]. Kad tokius įvykius pavyktų paversti sudėtingesniais bei artimesniais sprendžiamos problemos dalykinei sričiai, reikia atrinkti iš per tam tikrą laiką gauto tokių pranešimų srauto aibę tokių, kurie tenkina iš anksto nustatytus laiko, eiliškumo bei turinio apribojimus, leidžiančius nustatyti, jog įvyko tam tikras aukštesnio lygio įvykis, pavyzdžiui tam tikrų akcijų vertė smarkiai sumažėjo.

Šiai problemai spręsti yra naudojamos srautinės įvykių apdorojimo sistemos, kurios skaido gautus duomenis pagal tam tikrus žymenis į sudėtines dalis, kurias, jungiant į tam tikrus poaibius, tenkinančius tam tikrą įvykį signalizuojančius apribojimus, sekas ar šablonus, yra nustatoma, jog įvyko atitinkamas įvykis, kur kiekvienas įvykis aprašomas tam tikra užklausa,

nusakančia apribojimus ar šabloną [Hir12]. Daug tokių sistemų remiasi įvykių sekų atrinkimu iš srauto panašiais principais kaip atliekamos duomenų užklausos duomenų bazėse, kuriose visi duomenys yra lentelėse [FJL+01], tačiau tokia realizacija reikalauja duomenų srautą iš anksto apdoroti prieš ieškant atitikmenų, t. y. atlikti gaunamų duomenų skaidymą bei filtravimą dėl ko prarandama galimybė kuo didesnę proceso dalį atlikti lygiagrečiai. Taip pat sudėtingėjant sistemoms ir daugėjant srautų iš kurių gaunama informacija apie įvykius, daugėja dominančių ir norimų stebėti įvykių skaičius, o tai savo ruožtu didina atliekamų šablonų atpažinimo skaičių [DGP+07]. Net ir prie šiuolaikinių duomenų apdorojimo centrinių procesorių branduolių skaičiaus gausėjimo, stebint įvykius didelius duomenų srautus generuojančiose sistemose , jų skaičius yra labai mažas palyginus su norimų atlikti užklausų skaičiumi, todėl nemaža dalis duomenų yra apdorojama nuosekliai ir įvykių atpažinimo procesas užtrunka daugiau laiko [Hir12].

Kadangi dauguma įvykių apdorojimo sistemų realizuotos kaip tam tikrų atributų paieška sraute, lyg tai būtų užklausa duomenų bazėje, tai kiekvienam įvykiui aprašyti yra naudojama tam tikra užklausų kalba, kuria galima formuluoti predikatus ir ryšius tarp jų, nusakančius dominantį įvykį ar jo dalį. Predikatus atitinka duomenų sraute gaunamos duomenų struktūros, o ryšius tarp jų operatoriai, kurie įvykių apdorojimo sistemoje yra realizuojami kaip funkcijos, transformuojančios predikatus į tolimesniems operatoriams reikalingus predikatų poaibius ar naujus sudėtinius įvykius [ABB+04]. Kadangi kiekviena užklausa įvykiui atpažinti yra predikatų ir operatorių ryšių sistema, tai įvykių apdorojimo sistemoj kiekvieną įvykio atpažinimo užklausą galima atvaizduoti, kaip operatorių medį, kurio lapai yra predikatų įvesčiai, aukštesniuose lygiuose esančių viršūnių įvestis yra žemesnių viršūnių išvestis, o išvestis aukščiau esančių viršūnių įvestis. Šaknis išveda apdorotų įvykių ar sąryšių rezultatus. Toks modelis aiškiai atskleidžia, kad predikatai turi būti apdorojami tam tikra tvarka, kad būtų gaunami norimi rezultatai, tačiau esant ribotiems skaičiavimo ištekliams tik dalis operatorių gali vykdyti duomenų apdorojimą vienu metu. Todėl kiekvieno vykdomo operatoriaus išvesties srautas yra kaupiamas eilėje, kaip kito operatoriaus įvesties srautas [ABB+04], tokiu būdu išnaudojant daugiau atminties. Siekiant tokia sistema apdoroti daug duomenų su daug užklausų, kyla dvi problemos – kaip padidinti sistemos įvykių apdorojimo spartą bei kaip neišnaudoti daugiau operatyviosios atminties nei turi fizinė mašina, nes pastaruoju atveju dalis operatyviosios atminties sektorių būtų perkeliami ir keičiami kietajame diske, kas sistemą labai smarkiai sulėtintų.

Iki šiol šios problemos buvo sprendžiamos keliais būdais – siekiu kuo efektyviau išnaudoti procesorių spartinančiąją atmintį, bandant iš anksto nuspėti ir užkrauti atminties įrašus, kurių

turėtų prireikti tolimesniuose skaičiavimuose [FJL+01], išvengti tokių pat operatorių atliekamo darbo dubliavimo skirtingoms užklausoms [MSW07] bei bandymu parinkti optimalų operatorių vykdymo eiliškumą, kuris leistų sumažinti naudojamos operatyviosios atminties kiekį [BBD+03]. Pirmuoju ir antruoju atvejais iš visų užklausų siekiama surinkti vienodus rezultatus duodančius operatorius ir jų gaunamus rezultatus panaudoti atliekant skaičiavimus tik vieną kartą bei iš anksto prašant užkrauti predikatų duomenis į procesoriaus spartinančiąją atmintį, kad būtų išvengta procesoriaus prastovų duomenų persiuntimo metu. Trečiuoju atveju yra nustatoma, kurie operatoriai ar jų grandinės sugeba apdoroti daugiausiai duomenų per tam tikrą laiko vienetą ir siekiama, teikti tiems operatoriams vykdymo pirmumą tuo atveju, kai yra susidariusios eilės, tokiu būdų sumažinant naudojamos atminties kiekį. Tačiau šie sprendimai turi didelių trūkumų, t. y. jie daro įvykių apdorojimo sistemą žymiai sudėtingesne bei savo ruožtu reikalauja papildomų mechanizmų, kurie atliktų užklausų analizę bei parengtų atitinkamą apdorojimo vykdymo planą. Kuo daugiau įvykių siekiama apdoroti, tuo daugiau užklausų reikia išanalizuoti – tai savo ruožtu reikalauja papildomų skaičiavimo ir atminties resursų, dėl kurių efektyviausio plano parengimo sistema konkuruoja su įvykių apdorojimo sistema, siekiant rasti pusiausvyrą tarp laiko sugaištamo analizavimui bei sunaudojamos atminties kiekio.

Deja, tokiose srityse kaip prekyba vertybinių popierių biržoje, atominių reaktorių valdymo sistemose ar kitose, kuriose laiku nesureagavus į kritinius įvykius padariniai būtų labai nuostolingi ar net katastrofiški, atsiranda poreikis kuo sparčiau, t. y. ultra greitai atlikti įvykių apdorojimą bei pranešti apie susidariusią situaciją, kitaip tariant, įvykiai turėtų būti apdorojami su kiek įmanoma mažesne delsa tarp šakninio primityvaus įvykio gavimo ir jo sugeneruotų sudėtingų įvykių grąžinimo. Esant dideliam duomenų srautui ir daug norimų stebėti įvykių, didesnis duomenis apdorojančių procesorių skaičius išspręstų pastarąsias problemas, todėl duomenų apdorojimui pasitelkus vaizdo apdorojimo procesorius galima būtų tikėtis žymiai didesnės spartos, nes pastarieji yra suprojektuoti atlikti daug lygiagrečių skaičiavimų vienu metu [TS12]. Tačiau vaizdo apdorojimo įrenginių architektūra smarkiai skiriasi nuo centrinių procesorių architektūros, t. y. jos realizuoja vienos instrukcijos daugeliui duomenų aibių (angl. single instruction, multiple data – SIMD) ar vienos instrukcijos daugeliui gijų (angl. single instruction, multiple threads – SIMT) lygiagretaus vykdymo modelius [WPS+10]. Didžiosios daugumos šiuo metu kuriamų vaizdo apdorojimo procesorių architektūra paremta daugelio mažesnių, paprastesnių bei efektyvesnių branduolių jungimu į blokus, leidžiančius formuoti gijų hierarchijas, kurios lygiagrečiai atliktų tokias pat operacijas su skirtingomis aibėmis duomenų. Plačiai paplitusių gamintojų, tokių kaip AMD ar Nvidia, vaizdo apdorojimo procesorių architektūros yra panašios tuo, kad abiejose gijų hierarchijos skirstomos į tinklus (angl. grid)

sudarytus iš blokų (angl. block), kurie savo ruožtu susideda iš vienu metu vykdomų gijų aibių (angl. warps), kurios vykdymo metu yra perskirstoms skaičiavimo elementams, siekiant pastaruosius kuo labiau apkrauti, taip kuo efektyviau išnaudojant [WPS+10]. Viena iš ryškiausių tokios architektūros savybių yra ta, jog visos gijų aibę sudarančios gijos vienu metu yra vykdomos susietai, t. y. kartu ir atlieka vieną ir tą pačią komandų dekoderio iškoduotą instrukciją, tačiau kadangi kiekvienas srautinio procesoriaus apdorojimo elementas turi atskirus instrukcijos argumentų bei adresavimo registrus, tai ta instrukcija yra atliekama su skirtingomis duomenų aibėmis. Šios savybės dėka yra pasiekiamas didelis informacijos srautų apdorojimo lygiagretumas bei sparta, tačiau, lyginant su centriniais procesoriais, tokia architektūra turi ir silpnybių, kurios reikalauja atitinkamai priderinti programinę įrangą, kad sparta nebūtų prarandama. Svarbiausi tokių trūkumų pavyzdžiai būtų šie:

- neturi tokių sinchronizacijos galimybių tokių kaip centriniai procesoriai, todėl skaičiavimo rezultatų sinchronizacija tarp gijų gali būti vykdoma tik atlikus visas instrukcijas, kitu atveju visos gijos blokuojasi ir prarandama sparta [WPS+10];
- neturi spartinančiosios atminties, todėl norint maksimaliai išnaudoti turimus fizinius resursus reikia stengtis pilnai užpildyti leistinų gijų aibių skaičių, kad kol viena aibė laukia resursų, fiziniai resursai būtų išnaudojami kitų aibių instrukcijoms atlikti, nes, jei bent vienai gijai iš aibės trūksta argumento, visos gijos blokuojasi, kadangi vykdomos gali būti tik visos kartu [WPS+10];
- netolygus sąlyginių kodo blokų vykdymo išsišakojimas taip pat mažina spartą, nes kadangi visos gijos turi būti vykdomos vienu metu, tai tos gijos, kuriose sąlyginis blokas turi būti vykdomas, vykdys instrukcijas, o kitos – susietos blokuosis, atlikdamos bereikšmes instrukcijas, kol visos sąlyginio bloko instrukcijas vykdančios gijos baigs darbą [WPS+10].

Tam, kad pastarieji apribojimai nebūtų smarkaus informacijos apdorojimo spartos sumažėjimo priežastimi, programos, kurios bus vykdomos tokios architektūros procesorių turi būti atitinkamai suprojektuotos, kad duomenų srautų apdorojimui reiktų kuo mažiau sinchronizacijos, kad apdorojamus duomenis būtų galima suskaidyti į kuo optimalesnį kiekį darbo paketų, kurie galėtų būti paskirstomi optimaliam skaičiui gijų aibių bei kad sąlyginis vykdymas išsišakotų taip, jog kuo daugiau gijų vienu metu vykdytų instrukcijas.

Kaip programos vykdymo sparta priklausytų nuo duomenų apdorojimo procesoriaus architektūros, atliekant tarpusavyje nepriklausomas operacijas su duomenimis labai aiškiai galima pamatyti panagrinėjus šiuos dvejų matricų daugybos pavyzdžius:

```
// CPU matrix multiply C = A * B
void matMul (float* A, float* B, float* C, int dim) {
  for (int row = 0; row < dim; row++) {
    for (int col = 0; col < dim; col++) {
      // Dot row from A with col from B
      float val = 0;
      for (int i = 0; i < dim; i++)
        val += A[row * dim + i] * B[i * dim + col];
      C[row * dim + col] = val;
    }
  }
}
```

1 pav. Dvejų matricų sandaugos realizacija, naudojant centrinio procesoriaus branduolį [TS12]

```
// OpenCL Kernel for matrixmultiply. C = A * B
__kernel void
matrixMul (__global float* C,
           __global float* A,
           __global float* B,
           int wA, int wB) {
  int tx = get_global_id (0); // 2D Thread ID x
  int ty = get_global_id (1); // 2D Thread ID y

  // Perform dot-product accumulate into value
  float value = 0;
  for (int k = 0; k < wA; ++k) {
    value += A[ty * wA + k] * B[k * wB + tx];
  }
  C[ty * wA + tx] = value; // Write to device memory
}
```

2 pav. Dvejų matricų sandaugos realizacija, naudojant GPU duomenų apdorojimo procesoriaus elementus [TS12]

Iš pateiktų pavyzdžių matyti, kad 1 pavyzdyje vienas branduolys sinchroniškai ims reikšmes iš kiekvienos matricos A eilutės ir daugins su kiekvieno matricos B stulpelio reikšme, paskui gautas reikšmes sudės ir gaus atitinkamoje eilutėje ir stulpelyje esančią matricos C reikšmę. Tuo tarpu 2 – ame pavyzdyje matyti, kad kiekviena gija sudaugina vienos matricos A eilutės reikšmes su vieno matricos B stulpelio reikšmėmis, gautas sandaugas sudeda ir gautą suma patalpina atitinkamame matricos C kintamajame [TS12]. Matyti, kad turint daug vaizdo apdorojimo procesoriaus branduolių, vykdomos gijos lygiagrečiai skaičiuotų kiekvieną matricos C reikšmę, tuo tarpu centrinio procesoriaus branduolys tai darytu nuosekliai, taip sugaišdamas daugiau laiko, jei skaičiavimo operacijos atliekamos per nereikšmingai skirtingą laiko tarpą.

Taigi, matant vaizdo apdorojimo procesorių lygiagretaus vykdymo privalumus bei siekiant didesnės duomenų srautų apdorojimo spartos, darbe yra tiriama, kaip įvykių apdorojimui gali būti panaudoti šiuolaikiniai vaizdo apdorojimo procesoriai, t. y. didelio lygiagretumo apdorojimo galimybėmis pasižyminti techninė įranga, bei veiksniai lemiantys jų pranašumus ar trūkumus, lyginant su centriniais procesoriais.

Darbo tikslas

Šiais laikais dauguma sudėtingų įvykių apdorojimo sistemų yra realizuotos atliekant skaičiavimus su bendros paskirties centrinio procesoriaus branduoliais, tačiau esant dideliems duomenų srautams ir norint stebėti daug sudėtingų įvykių, jų skaičius gali būti per mažas, kad duomenys būtų apdorojami per norimą laiko intervalą.

Šio darbo tikslas – pritaikyti sudėtingų įvykių atpažinimo principus, realizuojant sistemą, pritaikytą vykdymui su labai daug lygiagrečių skaičiavimų galinčia atlikti technine įranga, t. y. grafinio vaizdo apdorojimo procesoriais, bei ištirti kaip būtų galima patobulinti sudėtingų įvykių

apdorojimo sistemą ar kokių veiksnių reikėtų vengti, norint padidinti lygiagrečių skaičiavimų efektyvumą bei sumažinti vykdymo laiką.

Uždaviniai

Išsikeltiems darbo tikslams pasiekti bus sprendžiami šie uždaviniai:

- realizuoti sudėtingų įvykių apdorojimo sistemos modelį, palaikantį pagrindinius šiuolaikinių sistemų įvykių užklausų operatorius ir leidžiantį atlikti reprezentatyvius bandymus bei vykdomą tiek centrinio procesoriaus branduolių, tiek vaizdo apdorojimo procesoriaus, optimizuojant realizaciją taip, kad būtų išnaudojami atitinkamo procesoriaus architektūriniai pranašumai;
- ištirti OpenCL karkaso pritaikymo galimybę vaizdo apdorojimo procesoriaus vykdomo modelio realizacijai;
- nusistačius pagrindinius spartą įtakojančius parametrus, atlikti įvykių apdorojimo bandymus su įvykių apdorojimo sistema vykdoma centrinio ir grafinio procesorių, palyginti gautus spartos rezultatus ir nustatyti kada geriausia naudoti atitinkamą procesorių;
- pateikti išvadas ir rekomendacijas, gautas realizuojant grafinio procesoriaus vykdomą sistemą bei išanalizavus gautus spartos palyginimo rezultatus.

Laukiami rezultatai

Ištyrus kaip kinta sudėtingų įvykių apdorojimo sistemos sparta priklausomai nuo įvairių parametrų ir vykdomosios aparatinės įrangos, tikimasi pateikti konkrečius veiksnius, smarkiausiai įtakojančius sistemos spartą, bei rekomendacijas kaip sistemą būtų galimą patobulinti, siekiant išvengti neigiamų veiksnių bei padidinti spartą, ir kokiais atvejais viena aparatinė įranga būtų labiau tinkama užduočiai atlikti, nei kita.

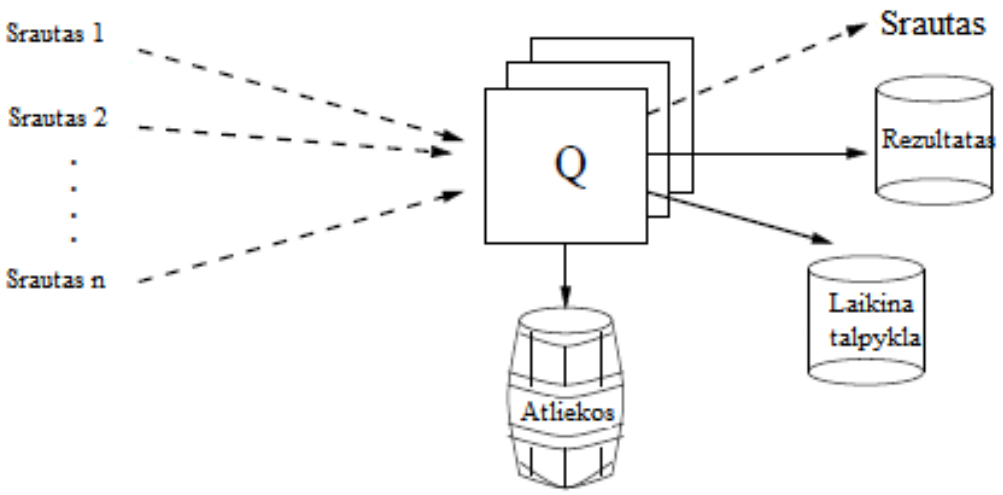
1. Analitinė dalis

1.1. Sudėtingų įvykių apdorojimo sistemos apibrėžimas

Sudėtingų įvykių apdorojimo sistema – tai sistema, kuri gaunamuose įvesties duomenų srautuose ieško šablonų, kurie nusako, jog įvyko tam tikras sudėtingesnis įvykis [Hir12]. Patys duomenų srautai – tai kas kažkokį, neapibrėžtą laiko tarpą gaunamos duomenų sekos, kurios gali būti sudarytos iš reikšmių, kurios sekoje išdėliotos pagal tam tikrą iš anksto nustatytą struktūrą, arba iš žymių ir reikšmių porų, kaip įrašai duomenų bazėse, kur viena srauto duomenų seka atitiktų vieną lentelės eilutę, o žymės identifikuotų lentelės stulpelį. Kitaip sakant, duomenų srautus galima būtų įsivaizduoti kaip duomenų bazes, kuriose galima tik viena operacija, t. y. duomenų įrašų pridėjimas. Savo ruožtu šablonai yra nusakomi tam tikra kalba aprašomais duomenų atributų filtrais, kurie detalizuoja kokioms atributų reikšmėms esant galima laikyti, jog įvykis įvyko ir kaip sukonstruoti naujo sudėtinio įvykio atributus. Filtrų aprašymai yra analogiški duomenų bazių užklausų aprašymui, o įvykių apdorojimas atitinka tokių užklausų įvykdymą duomenų bazėje. Tarp šių dviejų analogijų skirtumas yra tik tas, jog duomenų bazėje užklausos yra įvykdomos vieną kartą, pritaikant jas statinei duomenų aibei, esančiai duomenų bazėje lentelių pavidalu, ir gaunamas baigtinis rezultatas po apibrėžto laiko tarpo, o vykdant sudėtingų įvykių atpažinimą gaunamuose duomenų srautuose – duomenų aibės nėra statinės, apibrėžto dydžio, o dėl to ir vykdomos užklausos bei jų rezultatai nėra baigtiniai, t. y. kol yra gaunama naujų duomenų, tol esamos užklausos gali generuoti naujus rezultatus. Todėl sudėtingų įvykių apdorojimo sistemą galima vadinti specializuota tęstinių užklausų duomenų srautams sistema.

1.1.1. Tęstinių užklausų duomenų srautams sistemos apibrėžimas

Tęstinių užklausų duomenų srautams sistema turėtų spręsti problemas, kurios kyla, bandant pritaikyti tradicines duomenų bazių valdymo sistemas užklausų vykdymui dinaminiams duomenų srautams, t. y. tradicinės duomenų bazių valdymo sistemos yra pritaikytos vykdyti tas pačias užklausas daug kartų baigtinei duomenų aibei, tačiau kai siekiama užklausas vykdyti duomenų srautams, tų pačių užklausų vykdymas daug kartų yra nepraktiškas, nepageidautinas ir neįgyvendinamas, nes duomenų srautų dydis yra neapibrėžtas ir dažnai net negali ir neturi prasmės būti saugomas duomenų kaupikliuose [BW01]. Todėl pati sistema galėtų būti modeliuojama kaip programų sistema, kuri, iš kurio nors srauto gavusi naują duomenų seką, vykdytų kiekvieną iš tam tikro baigtinio skaičiaus užklausų, kurios savo ruožtu išsaugotų duomenis apdorojimui ateityje ar formuotų atsakymus, atlikdamos tam tikrus su užklausa susijusius veiksmus su tam tikrais apribojimais. Tokios sistemos architektūros modelis pateikiamas 3 paveikslėlyje.



3 pav. Tęstinių užklausų duomenų srautams sistemos architektūros modelis [BW01]

Sistemos veikimo principą geriausiai apibrėžtų 4 paveikslėlyje pateiktos tęstinės užklausos vykdymo pavyzdys, kurio vykdymo schema yra pateikiama 5 schemoje.

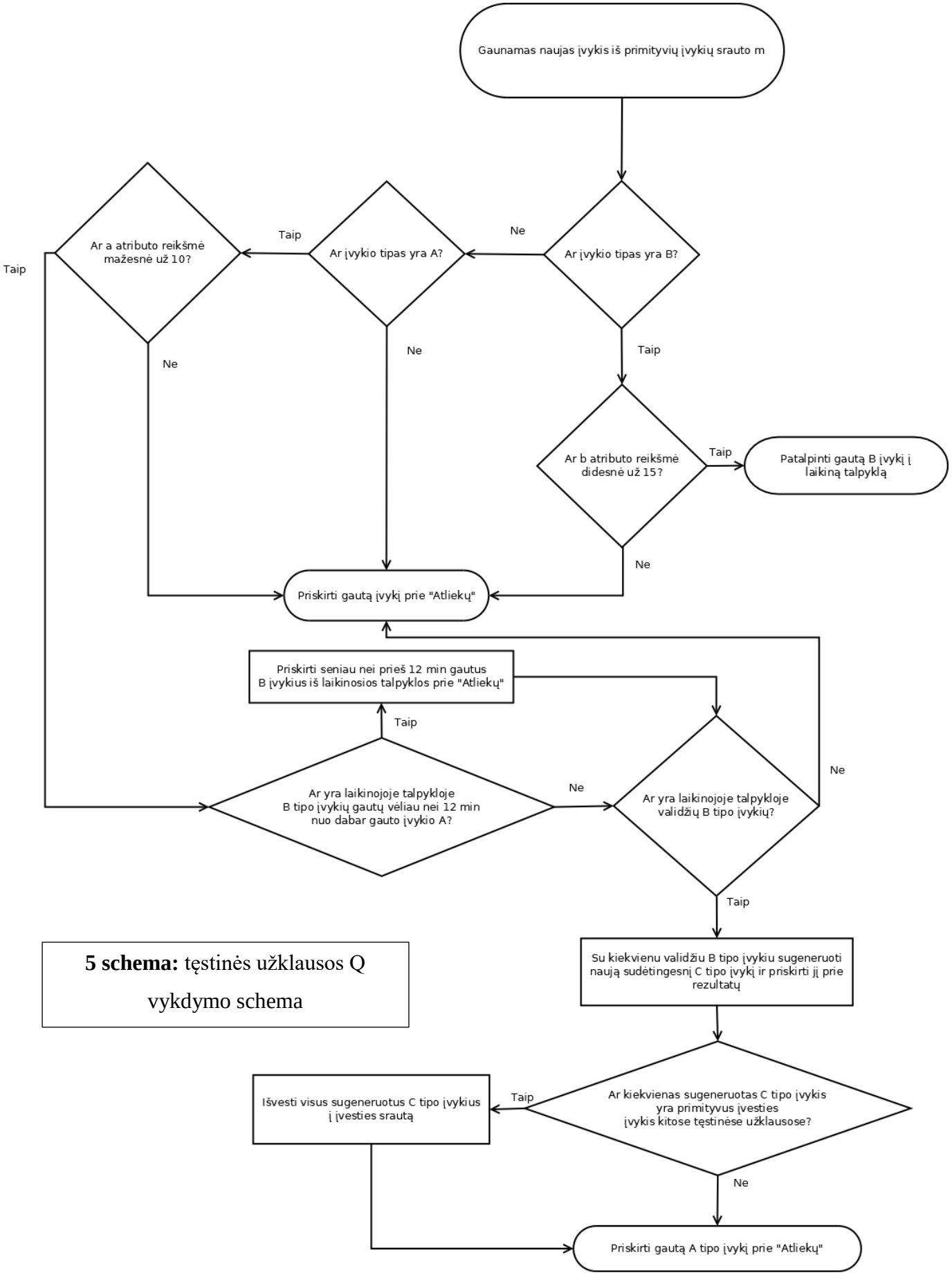
```
Q:

    EVENT C

    WHERE A.a < 10 AND
          B.b > 15 AND
          EACH B WITHIN 12 min FROM A

    SELECT C.c = B.b
```

4 pav. Tęstinės užklausos aprašas

Nagrinėjant schemą, galima pastebėti jog gavus įvykį iš bet kurio srauto iš eilės vienas po kito yra atliekami tokie žingsniai:

1. Jei įvykio tipas yra *B*, t. y. jis nėra šakninis, vykdomas 2 žingsnis, kitu atveju vykdomas 3 žingsnis.
2. Jei įvykio atributų reikšmės tenkina taisyklėje aprašytas sąlygas, t. y. *b* atributo reikšmė yra didesnė nei 15, įvykis yra talpinamas į laikinąją talpyklą ir vykdomas 13 žingsnis, kitu atveju vykdomas 12 žingsnis.
3. Jei įvykio tipas yra *A*, t. y. jis yra šakninis, vykdomas 4 žingsnis, kitu atveju vykdomas 12 žingsnis.
4. Jei įvykio atributų reikšmės tenkina taisyklėje aprašytas sąlygas, t. y. *a* atributo reikšmė yra mažesnė nei 10, vykdomas 5 žingsnis, kitu atveju vykdomas 12 žingsnis.
5. Jei laikinojoje talpykloje yra įvykių senesnių nei 12 min. nuo gautojo *A* įvykio, jie priskiriami prie atliekų, t. y. atlaisvinami jų naudojami resursai. Vykdomas 6 žingsnis.

6. Jei laikinojoje talpykloje yra įvykių *B* vykdomas 7 žingsnis, kitu atveju vykdomas 12 žingsnis.
7. Iš laikinosios talpyklos imamas dar nepanaudotas, seniausias įvykis *B.*
8. Panaudojant 7 žingsnyje pasirinktą primityvų įvykį *B* ir iš srauto gautą, apdorojamą įvykį *A*, sugeneruojamas naujas sudėtingas įvykis *C*.
9. 8 žingsnyje sugeneruotas įvykis *C* yra talpinamas į rezultatų talpyklą.
10. Jei įvykis *C* yra primityvus įvykis kitoje tęstinėje užklausoje, jis išvedamas į įvesties srautą.
11. Jei laikinojoje talpykloje dar yra nepanaudotų primityvių įvykių *B,* vykdomas 7 žingsnis, kitu atveju vykdomas 12 žingsnis.
12. Sraute gautas ir apdorojamas įvykis yra priskiriamas prie atliekų, t. y. atlaisvinami jo naudojami resursai.
13. Laukiama kol bus gautas sekantis primityvus įvykis.



5 schema: tęstinės užklausos Q vykdymo schema

Siekiant aiškiau apibrėžti su kokiomis problemomis susiduriama atliekant tęstines užklausas duomenų srautams pagal 3 paveikslėlyje pateiktą modelį, galima būtų panagrinėti dar ir tokį bendresnį pavyzdį: tarkime, kad turime vieną tęstinį duomenų sekų srautą ir norime pastarajam srautui įvykdyti vieną užklausą *Q* taip, kaip buvo aptarta 4 paveikslėlyje aprašytos užklausos atveju. *Q* yra tęstinė užklausa, kuri yra vieną kartą paleidžiama ir kuri generuoja tęstinį atsakymą, kai tik srautu gaunama nauja duomenų seka. Laikant, kad duomenų srautu gaunamos tik naujos sekos, t. y. negalimi prieš tai buvusių sekų pakeitimai ar išmetimai, yra keletas skirtingų būdų vykdyti užklausą, iš kurių kiekvienas kelia tam tikrus iššūkius [BW01]:

1. Norint visą laiką saugoti ir padaryti užklausos rezultatą pasiekiamą, reiktų srauto naują duomenų seką saugoti „Rezultatų“ talpykloje, jei pastaroji tenkina užklausos sąlygas. Tačiau esant neriboto dydžio duomenų srautams, rezultatų talpykla taip pat turėtų būti neriboto dydžio [BW01].
2. Renkantis, vietoje naujos srauto sekos išsaugojimo, iš jos pagaminti naujas duomenų srauto sekas, kaip naujus elementus, ir pastaruosius išvesti kaip naują išvesties duomenų srautą, kuris taip pat būtų neribojamas, būtų išsprendžiama pirmame punkte aptarta neriboto dydžio talpyklos poreikio problema, bet tik tuo atveju, jei atliekamos operacijos nereikalauja visų prieš tai gautų sekų (pavyzdžiui rezultatų iš sekų kaip duomenų bazių lentelės jungimo su pačia savimi) [BW01].
3. Nors duomenų srautas gali tik pridėti naujas duomenų sekas, tačiau atliekant tam tikras duomenų bazių užklausų operacijas, tai vis tiek galėtų reikalauti jau sugeneruotų rezultatų keitimo ar elementų pašalinimo (pavyzdžiui grupavimo pagal tam tikrą atributą operacija). Tokiu atveju kiltų poreikis kažkokiu būdu atnaujinti ar keisti rezultatus išvestus išvesties duomenų sraute [BW01].
4. Jei būtų nagrinėjamas pats bendriausias duomenų bazių valdymo sistemų galimybių variantas, t. y. duomenų sraute būtų galimos ir atnaujinimo bei išmetimo operacijos, tai įvesties duomenų srautai turėtų būti įsivaizduojami kaip duomenų bazės lentelės, kurių elementai gali būti ne tik pridėti, bet ir atnaujinti bei pašalinti, kas vėlgi reikalautų galimybės arba visus gaunamus duomenis saugoti, arba kažkokiu būdu atnaujinti bei šalinti elementus jau išvestus išvesties duomenų sraute [BW01].

Iš aptarto pavyzdžio matyti, jog tokios užklausos kokios yra naudojamos tradicinėse duomenų bazėse yra neįgyvendinamos su tęstinėmis užklausomis duomenų srautams, todėl yra du variantai šioms problemoms spręsti, t. y. apriboti užklausų operatorių aibę [BW01], taikyti

apribojimus rezultatų aibės dydžiui, arba nustatyti maksimalų dydį duomenų talpyklai, kuri yra papildomai reikalinga laikinų duomenų saugojimui [TGN+92].

3 paveikslėlyje pavaizduotame modelyje matyti, kad kiekviena užklausa, duomenų sraute gavus naują duomenų seką *t*, gali pastarąją apdoroti atlikdama kažkuriuos iš žemiau pateiktų veiksmų:

1. Užklausa gali nustatyti, kad dėl *t*, bus sugeneruota nauja atsakymo seka *a*. Jei yra žinoma, jog *a* visada liks atsakyme, tai užklausa gali išvesti naująją seką tiesiai į išvesties *„Srautą“* [BW01].
2. Jei yra nustatoma, jog *a* turėtų būti užklausos atsakyme, tačiau yra numatoma, kad gali taip nutikti, jog ateityje ji gali užklausos reikalavimų nebetenkinti, tuomet ji talpinama į laikiną *„Rezultatų”* talpyklą. Kitaip sakant, modelyje apibrėžtas išvesties *„Srautas“* ir *„Rezultatas“* yra skirti formuoti užklausos rezultatui. Jei tikslas yra sumažinti užimamos vietos kiekį, tuomet reikėtų stengtis kuo daugiau rezultatų išvesti iškart į išvesties srautą [BW01].
3. Dėl gautos duomenų sekos *t,* kai kurios *„Rezultatų“* talpykloje esančios atsakymų sekos gali būti modifikuojamos arba ištrinamos. Taip pat iš *„Rezultatų“* talpyklos duomenų sekos gali būti išvedamos į išvesties *„Srautą“* [BW01].
4. Gauta seka *t* gali būti išsaugota *„Laikinoje talpykloje”* tuo atveju, kai *t* arba duomenys gauti iš *t* yra reikalingi, kad užklausos rezultatai galėtų būti generuojami su po *t* ateinančiomis sekančiomis duomenų sekomis. Duomenų sekos taip pat gali būti perkeliamos iš *„Rezultatų“* talpyklos į *„Laikinąją talpyklą”* [BW01].
5. Tuo atveju, kai yra nustatoma, kad *t* nėra reikalinga nei atsakymo generavimui einamuoju momentu, nei bus reikalinga ateityje, *t* turi būti talpinama į atliekų talpyklą, kas paprastai realizuojama tiesiog duomenų sekos ištrynimu (nebent tokios reikšmės nori būti archyvuojamos) [BW01].
6. Gavus *t* gali būti nustatyta, jog anksčiau išsaugotos *„Rezultatų“* ar *„Laikinojoje”* talpyklose duomenų sekos yra nebereikalingos ir jos gali būti perkeliamos į *„Atliekų”* talpyklą. Ir šiuo atveju, jei yra siekiama minimizuoti papildomai sunaudojamos talpyklų vietos kiekį, reiktų stengtis nereikalingas duomenų sekas ištrinti kaip galima anksčiau [BW01].

1.1.2. Trigeriai

Trigeriai, dar žinomi kaip įvykio-sąlygos-veiksmo taisyklės, duomenų bazėse yra skirti stebėti tam tikrus iš anksto apibrėžtus įvykius bei sąlygas ir automatiškai atlikti tam tikrus veiksmus, kai tik apibrėžtos situacijos yra aptinkamos [WC96]. Trigeriai tradicinėse duomenų

bazėse gali būti apibrėžti kaip tęstinės užklausos aktyvioms duomenų lentelėms, t. y. kiekviena nauja duomenų seka atitinka įvykį – duomenų keitimą duomenų bazės lentelėje. Kai tik nauja duomenų seka yra pridedama į vieną iš aktyvių duomenų bazės lentelių, kiekviena tęstinė užklausa, atitinkanti vieną iš trigerių, susijusių su modifikuojama lentele, yra įvykdoma ir atitinkamas, su pastaruoju trigeriu susijęs veiksmas yra atliekamas kiekvienai trigerio užklausos rezultate sugeneruotai duomenų sekai, tokiu būdu užtikrinant leistiną duomenų bazės lentelių būseną ar leidžiant atlikti papildomus veiksmus, tinkamus aptiktai situacijai [BW01].

Tokiu būdu apibrėžtus trigerius tradicinėje duomenų bazėje, nesunku realizuoti ir 1.1. poskyryje aptartame tęstinių užklausų duomenų srautams modelyje, t. y. įvykiai, kuriuos norima stebėti ir tikrinti ar jie tenkina tam tikras trigerio sąlygas, gali būti suvokiami kaip tęstinių užklausų duomenų srautams naujai sugeneruotos duomenų sekos [BW01]. Tuomet patys trigeriai yra taip pat tęstinės užklausos, kurios vykdomos generuojamoms išvesties srauto duomenų sekoms ir kurios atlieka tam tikrus veiksmus, jei išvesties duomenų sraute yra randama sekų, kurios tenkina tam tikras trigerio sąlygas. Nors patys trigeriai yra realizuojami kaip tęstinės užklausos, tačiau jie negeneruoja naujų rezultatų, t. y. pagrindinis jų tikslas stebėti duomenų srautų pokyčius ir pastebėjus tam tikras sąlygas atitinkančią situaciją ar kitų tęstinių užklausų išvestą rezultatą, atlikti priskirtus veiksmus.

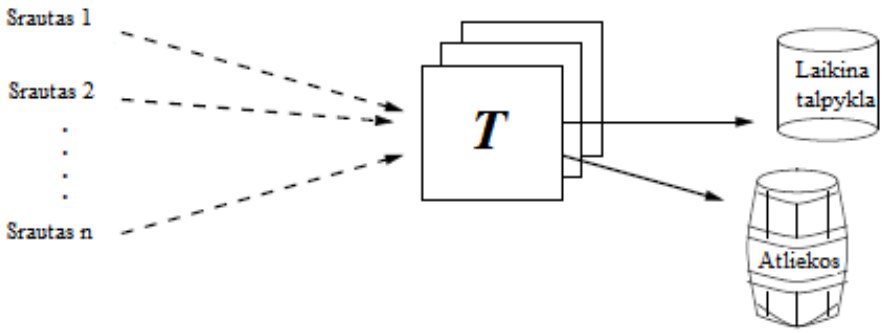
Patys veiksmai gali įtraukti jau sugeneruotų rezultatų modifikavimą, laikinojoje talpykloje laikomų duomenų pašalinimą, pranešimų generavimą ar bet kokius kitus veiksmus, kuriuos būtų galima realizuoti atitinkama paprograme.

1.1.3. Sudėtingų įvykių apdorojimo sistemos (CEP) modeliavimas

Kadangi 1.1. poskyryje nagrinėtas tęstinių užklausų duomenų srautams modelis iš tiesų yra bendresnis sudėtingų įvykių apdorojimo sistemos atvejis, tai visai nesunku pastebėti, kad pastarasis modelis puikiai tinka ir sudėtingų įvykių apdorojimo sistemoms modeliuoti. Kaip anksčiau buvo minėta CEP sistemos pagrindinis tikslas yra ne vykdyti tęstines užklausas, kurios iš įvesties srauto duomenų sekų generuotų išvesties srauto duomenų sekas, bet aptikti tam tikrus duomenų sekose pasitaikančius šablonus, kurie leistų nustatyti jog įvyko kažkoks įvykis, ir atitinkamai arba sugeneruoti tą įvykį atitinkančią naują duomenų seką ir ją perkelti atgal į įvesties duomenų srautą tolesniam aukštesnio lygio įvykių užklausom apdoroti, arba atlikti tam tikrus veiksmus, kurie skirti tam tikrų sistemų ar žmonių informavimui apie nustatyto įvykio aptikimą ir su juo susijusios informacijos pateikimui.

Kadangi CEP sistemos užklausos negeneruoja rezultatų, o tik sukuria arba informuoja apie pastebėtus įvykius, panaudodamos įvykį nusakančius duomenis, tai tokios užklausos atitinka 1.2 poskyryje aptartų trigerių apibrėžimą, t. y. CEP sistemos modelio tęstines užklausas atitiks

trigeriai, kurie stebės įvesties srautuose pasirodančias naujas duomenų sekas ir atitinkamai arba iš pastarųjų generuos naujas duomenų sekas, kurias galėtų apdoroti kiti tos pačios sistemos trigeriai, arba atliks informavimo veiksmus. Taip pat bendruoju atveju CEP sistemoje nėra poreikio atlikti tradicinėse duomenų bazėse naudojamų operatorių, tokių kaip grupavimas, lentelės jungimas su savimi ir kitų, kurie reikalautų tarpinių rezultatų saugojimo, t. y. užtektų *„Laikinosios talpyklos“* su įvykiu susijusių duomenų laikinam saugojimui, kol gaunamos visos duomenų sekos, reikalingos identifikuoti ir generuoti naują sudėtingesnį įvykį. Dėl pastarųjų priežasčių CEP modelyje nėra reikalingi išvesties *„Srautas“* ir *„Rezultatų”* talpykla, o CEP sistemos modelį sudarytų įvesties duomenų srautai, baigtinė aibė trigerių, sudarytų iš tęstinių užklausų, kurios apibrėžtų sąlygas, leidžiančias nustatyti tam tikro įvykio atsitikimą, bei atitinkamų veiksnių, susijusių su įvykio apdorojimu, ir *„Laikinoji talpykla“*, kaip pavaizduota 6 paveikslėlyje.



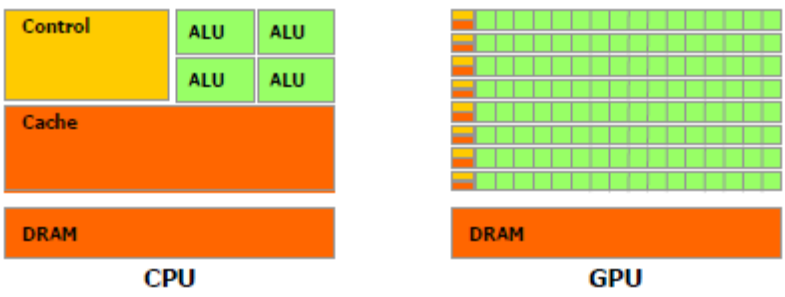
6 pav. Įvykių apdorojimo sistemos architektūros modelis

1.2. CPU ir GPU architektūrų skirtumai

Centriniai procesoriai (CPU) ir grafiniai procesoriai (GPU) yra sukurti vadovaujantis skirtingomis filosofijomis, skirtingo tipo darbams, todėl ir jų fizinės architektūros turi didelių skirtumų, į kuriuos būtina atsižvelgti, norint kuo efektyviau išnaudoti procesorių skaičiavimo resursus.

1.2.1. CPU fizinė architektūra

Centriniai procesoriai yra suprojektuoti bendros paskirties, kuo įvairesnėms programoms vykdyti, koncentruojantis į kuo



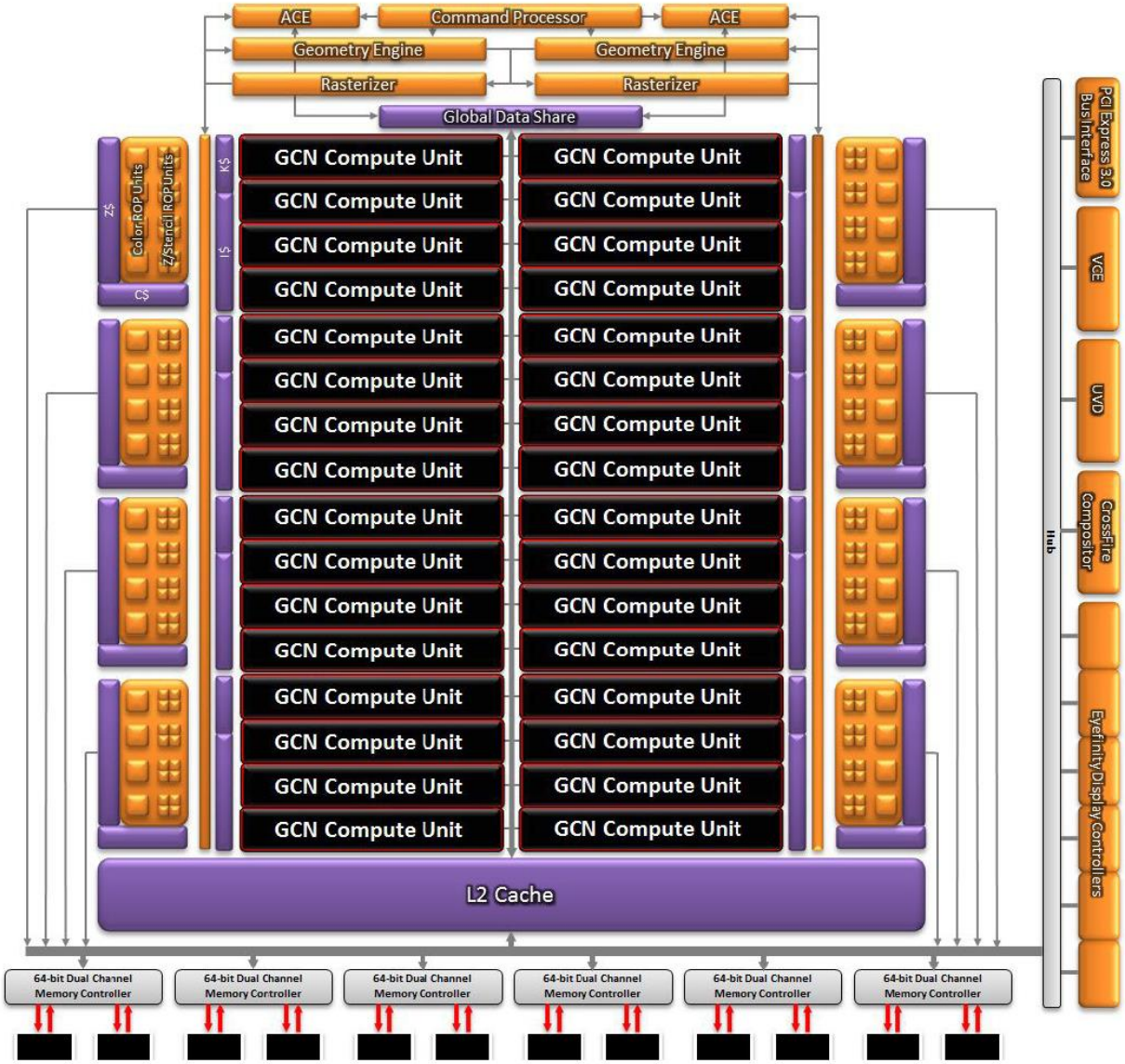
7 pav. Tranzistorių kiekio paskirstymas mikroprocesorių dalim [Nvi09]

greitesnį atsako laiką vienai užduočiai. Todėl tokių mikroprocesorių spartos didėjimą lemia pažanga architektūriniuose sprendimuose, tokiuose kaip sąlyginio valdymo nuspėjimas, operacijų vykdymas, neišlaikant jų tvarkos (angl. out-of-order execution), bei kelių lygių spartinančiosios atminties hierarchijos [LKC+10]. Tačiau šių mechanizmų realizacija didina mikroprocesoriaus sudėtingumą, reikalauja papildomo didelio kiekio tranzistorių, kurie savo ruožtu papildomai užima vietą, eikvoja energiją bei išskiria šilumą. Dėl šių priežasčių mikroprocesoriuose gali būti ribotas, t. y. nedidelis kiekis branduolių, kurie savyje turi nedidelį skaičių aritmetinių loginių įrenginių lyginant su grafiniais procesoriais. Tačiau pastarieji, lyginant su GPU, turi greitesnį atsako laiką, gali vykdymo metu keisti programų vykdymo kontekstus, pasižymi didesne sparta vykdant vieną giją per branduolį bei turi lankstesnį lygiagretaus gijų vykdymo sinchronizacijos mechanizmą. Tranzistorių kiekis skiriamas atitinkamų mikroprocesorių mechanizmų realizacijai, lyginant CPU su GPU, pavaizduotas 7 paveikslėlyje.

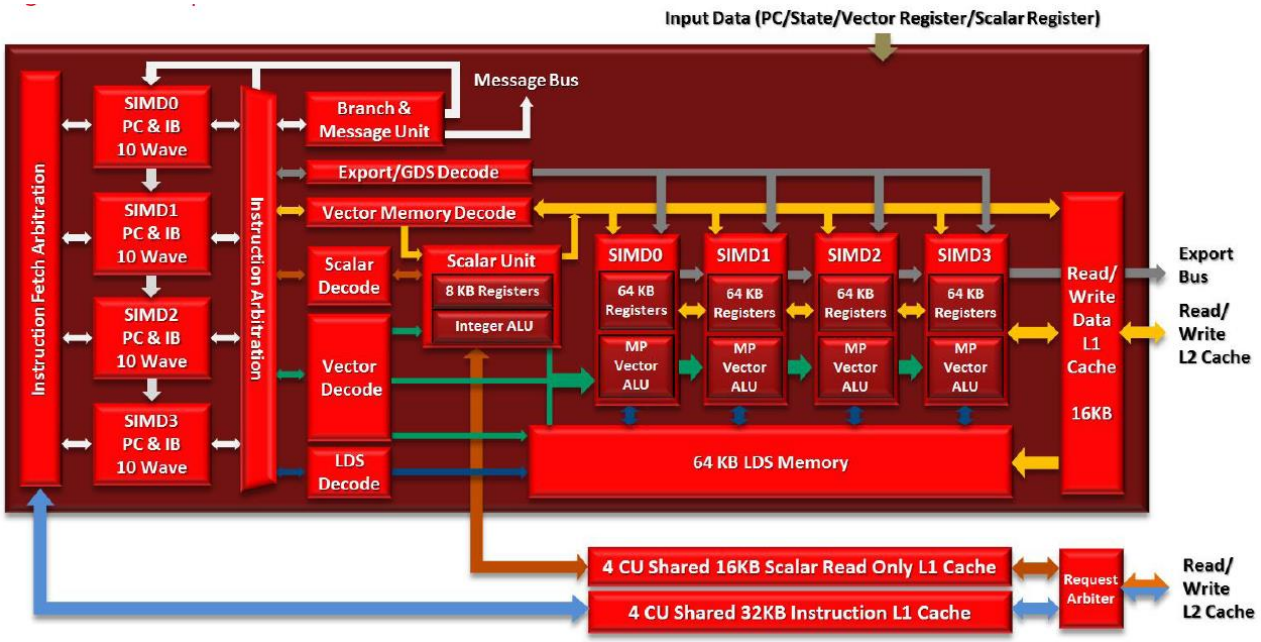
1.2.2. GPU fizinė architektūra

Grafiniai procesoriai pirmiausia buvo sukurti su vienu tikslu – grafinių vaizdų generavimu, t. y. koncentruojantis į lygiagrečius skaičiavimus, atliekant vieną, tą pačią instrukciją daugeliu gijų, kurių kiekviena apdoroja skirtingas duomenų aibes – pikselius. Todėl didžioji dalis tranzistorių grafiniuose procesoriuose yra skirti aritmetinių loginių įrenginių realizacijai. Kadangi architektūra suprojektuota dideliam duomenų paralelizmui, lyginant GPU su CPU mikroprocesoriais, GPU architektūra yra gerokai labiau tolerantiška delsoms, atsirandančioms dėl laiko sugaištamo reikalingų duomenų persiuntimui iš fizinės mašinos operatyviosios atminties į GPU vidinę atmintį. Kol reikiamų duomenų nėra, GPU gali naudoti skaičiavimo branduolius kitų, einamuoju momentu esančių spartinančiojoje atmintyje, duomenų aibių apdorojimui, todėl nėra tokio didelio poreikio daugelio lygių didelėms spartinančiosios atminties hierarchijoms ir vienos gijos sparta gali būti gerokai mažesnė nei CPU, nes dėl didelio skaičiavimo branduolių kiekio, apdorojančio duomenis paraleliai, visuminė duomenų apdorojimo sparta išlieka labai didelė [LKC+10].

Detalesnė GPU architektūros schema pateikta 8 paveikslėlyje, kur vaizduojama AMD GCN (angl. Graphics Core Next).

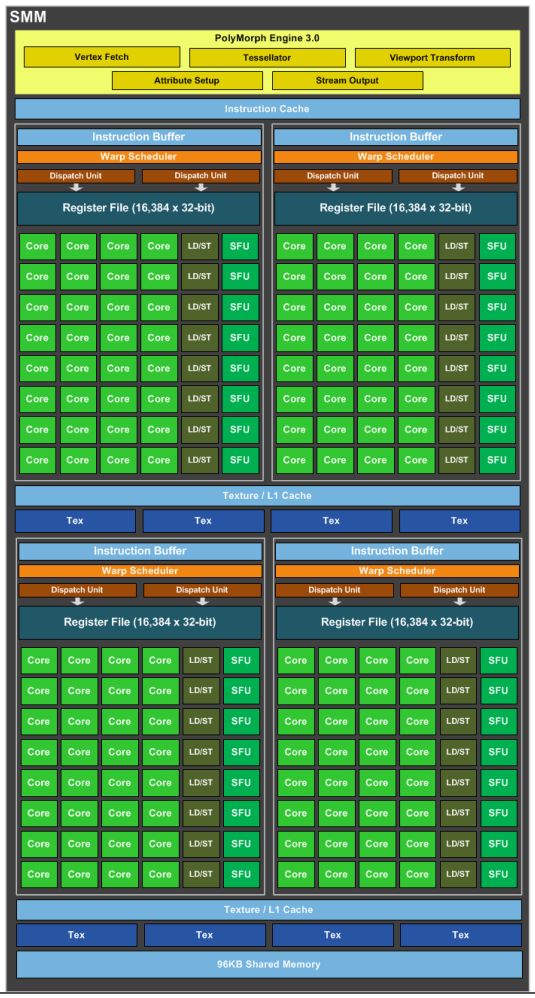


8 pav. GCN 1.0 architektūros schema (HD7970) [AMD12]



9 pav. GCN 1.0 architektūros skaičiavimo vienetas (CU) [AMD12]

Analizuojant schemą, nesunku pastebėti, kad visas mikroprocesorius sudarytas iš 32 vienodų blokų, kuruos AMD vadina skaičiavimo vienetais (angl. Compute Unit, CU). Vienas skaičiavimo vienetas yra smulkiausias autonomiškai galintis veikti modulis, turintis visus reikalingus valdiklius instrukcijų vykdymui. Pats skaičiavimo vienetas, pavaizduotas 9 paveikslėlyje, yra sudarytas iš instrukcijų dekoderių, instrukcijų spartinančiosios atminties, lokalios duomenų apsikeitimo atminties, vieno skaliarinio skaičiavimo branduolio ir 4 vektorinių skaičiavimo branduolių, kurių kiekvienas turi 16 gijų, vykdančių tą pačią vieną instrukciją [AMD12]. Taigi idealiu atveju vienas skaičiavimo vienetas gali tuo pat metu vykdyti 64 gijas, o visas mikroprocesoriaus lustas net 2048 gijų.



10 pav. Nvidia Maxwell architektūros srautinis mikroprocesorius (SMM) [Nvi14]

Kitos gerai žinomos grafinių procesorių projektuotojos ir gamintojos Nvidia inžinieriai naujausius grafinius lustus projektuoja remdamiesi labai panašia filosofija, t. y. visas branduolys susideda iš atskirų nepriklausomų blokų – srautinių mikroprocesorių (angl. Streaming Multiprocessor, SMM). Kiekvienas nepriklausomas blokas, kurio

schema pavaizduota 10 paveikslėlyje, sudarytas iš 4 SIMD procesorių, iš kurių kiekvienas turi po 32 gijas [Nvi14].

Tačiau tokia architektūra turi tam tikrų apribojimų, bandant ją pritaikyti bendros paskirties skaičiavimams. Pirmiausia ji turi aiškiai atskirtus atminties regionus, t. y. fizinės mašinos operatyvioji atminties, globali GPU atminties, lokalios vieno skaičiavimo modulio atminties bei privačios vienos gijos atminties. Toks suskirstymas į regionus turi keletą neigiamų pasekmių:

- duomenų persiuntimas iš operatyviosios atminties į gijos privačiąją atmintį turi nemažą uždelsimą, nes duomenims reikia peršokti per keletą atskirų atminties regionų [YF11];
- pakeistų duomenų dalinimasis tarp SIMD branduolių tame pačiame modulyje reikalauja papildomų operacijų, o dalinimasis tarp modulių yra dar sudėtingesnis, nes turi būti naudojama globali grafinio procesoriaus atmintis, todėl šios operacijos delsos atžvilgiu yra dar ir brangesnės nei duomenų dalinimasis tame pačiame modulyje [YF11];
- norint, kad uždelsimai nenulemtų labai didelio duomenų apdorojimo spartos sumažėjimo, reikia gerai apgalvoto algoritmo, kuris būtų pritaikytas tokiam atminties modeliui, o tai daro programavimo uždavinį sudėtingesniu [AMD13].

Kita tokios architektūros savybė, daranti bendros paskirties programavimą grafiniams procesoriams žymiai sudėtingesnį, yra sinchronizacija tarp gijų. Gijos vykdomos vieno SIMD procesoriaus skaičiavimo vienete yra surištos, t. y. yra garantuojama, kad jos tą pačią instrukciją atliks visos kartu, tačiau tarp SIMD procesorių yra reikalinga papildoma operacija – barjeras, kuri gali sueikvoti daug laiko, kol visos su barjeru susijusios gijos pasieks pastarąjį. Sinchronizacija tarp skaičiavimo vienetų yra negalima, todėl reikalinga naudoti papildomus mechanizmus, tokius kaip branduolio funkcijų darbų eilės ar įvykių mechanizmas, t. y. sinchronizacija tarp skaičiavimo vienetų turi būti atliekama centrinio procesoriaus [AMD13].

Paskutinė svarbi architektūros savybė yra ta, jog 8 paveikslėlyje pavaizduotos architektūros gijų, galinčių vienu metu apdoroti duomenis, skaičius yra teorinis, t. y. jis toks yra tik idealiu atveju, tačiau jei yra programuojama bendros paskirties skaičiavimams ir branduolio funkcija kažkurias instrukcija atlieka tik su sąlyginiu perdavimu, tai susiklosčius situacijai, kuomet vieno SIMD branduolio gijų instrukcijų vykdymo keliai išsišakos dėl skirtingų duomenų, vienos gijos neatliks jokių reikšmingų instrukcijų, kol kitos neįvykdys sąlyginių instrukcijų [AMD13]. Dėl to sumažėja efektyvus gijų skaičius, o kartu ir duomenų apdorojimo sparta.

1.2.3. CPU ir GPU resursų palyginimas

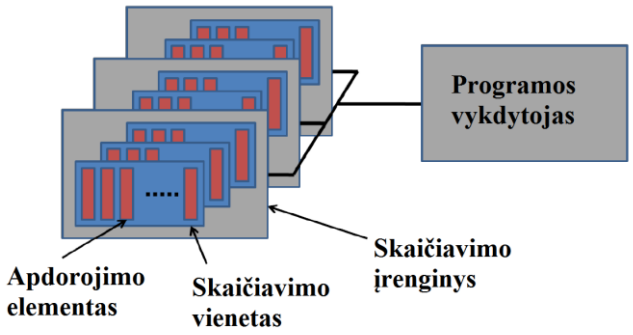
Siekiant apibendrinti 2.1. ir 2.2 skyriuose aptartus skirtumus, 1 lentelėje yra pateikiami techniniai duomenys centrinio ir grafinio procesorių, su kuriais bus atliekamas sudėtingų įvykių apdorojimo sistemų spartos tyrimas, techninių parametrų palyginimas.

		CPU	GPU
Modelis		Intel Core i5-3570K	AMD R290
Branduolių skaičius		4	2560 srautinių branduolių: 40 CU; 4 SIMD/CU; 16 srautinių procesorių/SIMD.
Gijų skaičius		4	2560
Branduolio dažnis		4,2 Ghz	1,050 Ghz
1 lygio spartinančioji atmintis	Duomenų	128KB	16KB (vienam CU)
	Instrukcijų	128KB	32 KB (vienam CU)
2 lygio spartinančioji atmintis		1 MB	1 MB
3 lygio spartinančioji atmintis		6 MB	-
1 lygio spartinančiosios atminties kiekis vienai gijai	Duomenų	32 KB	256 B
	Instrukcijų	32 KB	512 B
Paskutinio lygio spartinančiosios atminties kiekis vienai gijai		1,5 MB	409,6 B
Lokali atmintis (CU)		-	64KB
Teorinė duomenų apdorojimo sparta (32 bit)		4x122 GFLOPS	4848.6 GFLOPS
Teorinė duomenų apdorojimo sparta (64 bit)		-	606.1 GFLOPS

1 lentelė: CPU ir GPU techninių parametrų palyginimas

1.3. OpenCL modelis

Grafiniai procesoriai buvo pradėti naudoti bendros paskirties skaičiavimams palyginti neseniai ir viena iš svarbiausių to priežasčių yra ta, jog skirtingų gamintojų gaminamų grafinių procesorių architektūros nors ir yra panašios, tačiau realizacijos detalės reikalauja pritaikyti programą konkrečiai architektūrai, naudojant specifines žemo abstrakcijos lygio instrukcijas, kas labai apsunkindavo tiek probleminės srities uždavinio sprendimą, tiek reikalaudavo daug laiko bei pastangų. OpenCL (angl. Open Compute Language) technologija buvo pristatyta kaip šios problemos sprendimas. OpenCL yra programų kūrimo sąsaja, turinti savo programavimo kalbos specifikaciją bei vykdymo aplinkos sąsają pastarąja kalba parašytoms programoms. Pagrindinis OpenCL tikslas yra paslėpti nuo programuotojų konkretaus gamintojo mikroprocesoriaus architektūros realizacijos detales, pateikiant abstrakcijas, kurios atitinka GPU architektūrą, bei leisti programuoti naudojantis jau pažįstama programavimo kalba bei aplinka, todėl OpenCL kalbos specifikacija paremta ANSI C kalbos pagrindais [AMD13].



11 pav. OpenCL platformos modelis [TS12]

OpenCL platformos modelis, pavaizduotas 11 paveikslėlyje, susideda iš programos vykdytojo bei keleto skaičiavimo įrenginių. Programos vykdytojas gali būti bet koks kompiuteris turintis centrinį procesorių, operacinę sistemą bei vieną ar daugiau skaičiavimo įrenginių [TS12]. Šio tyrimo atveju skaičiavimo įrenginius atitiks grafinis procesorius, skaičiavimo vienetus – grafinio procesoriaus skaičiavimo vienetai aptarti 2.2. poskyryje, o apdorojimo elementus – viena SIMD procesoriaus gija.

1.3.1. Branduolio funkcija

OpenCL kalba parašyta programa vadinama branduoliu (angl. kernel). Branduolys – tai nedidelis vykdymo komandų vienetas – funkcija, suprojektuota lygiagrečiam vykdymui, su aiškiai apibrėžtu vienu tikslu. Gerai suprojektuotą branduolį turėtų būti galima vykdyti su kiekvienu įvesties srauto elementu, atitinkančiu vieną darbo vienetą, ir atitinkamai kiekvienam įvesties elementui generuoti išvesties elementą. Tokio branduolio pavyzdys pateiktas 2 paveikslėlyje [AMD13].

1.3.2. Darbo vienetas ir darbo vienetų grupės

Vienas branduolio funkcijos vykdymas su tam tikra duomenų aibe yra vadinamas darbo vienetu [TS12]. Vykdant branduolio funkciją reikia kažkokiu būdu adresuoti tam branduoliui skirtus duomenis. Tam tikslui OpenCL suskaido visą duomenų aibę į dviejų ar trijų dimensijų erdvę, kurią galima indeksuoti sveikaisiais skaičiais. Šie indeksai yra perduodami branduolio funkcijoms, vykdomoms konkrečių gijų, vykdymo metu argumentų pavidalu, kuriais naudojantis branduolio funkcijos gali iš globalios atminties pasiimti būtent joms priskirtus duomenis, su kuriais turėtų būti atliekamos kažkokios vieno darbo vieneto operacijos [TS12].

Siekiant prisitaikyti prie grafinių procesorių architektūrinių duomenų dalinimosi ir sinchronizacijos apribojimų, aptartų 2.2. skyrelyje, OpenCL suteikia galimybę grupuoti darbo vienetus į darbo grupes. Kiekvienos darbo grupės dydis yra apibrėžiamas pagal lokalų darbo grupei priklausančių darbo vienetų indeksaciją. Visi darbo vienetai priklausantys vienai darbo grupei yra vykdomi viename skaičiavimo įrenginio skaičiavimo vienete [AMD13].

1.3.3. Darbo vienetų grupės ir bangos frontai

Kadangi viena darbo grupė turi būti vykdoma vieno skaičiavimo vieneto, o pastarasis turi ribotą skaičių apdorojimo elementų, tai, esant situacijai, kai darbo grupėje yra daugiau darbo vienetų nei apdorojimo elementų skaičiavimo vienete, darbo grupės elementai turi būti skaidomi į poaibius. Geriausiai ir efektyviausiai grafinio procesoriaus resursai yra išnaudojami tuomet, kai tokių poaibių dydis atitinka gijų skaičių, kuriam SIMD procesorius susietai įvykdo vieną instrukciją. Šis skaičius vadinamas bangos frontu. Esant situacijai kai darbo vienetų skaičius yra mažesnis nei bangos fronto dydis arba gijų vykdymo keliai išsiskiria dėl sąlyginio vykdymo, SIMD procesorius reikiamas instrukcijas atlieka tik su dalimi gijų, o likusios susietos gijos tampa nepanaudotais resursais [AMD13].

1.3.4. Darbo vienetų sinchronizacija

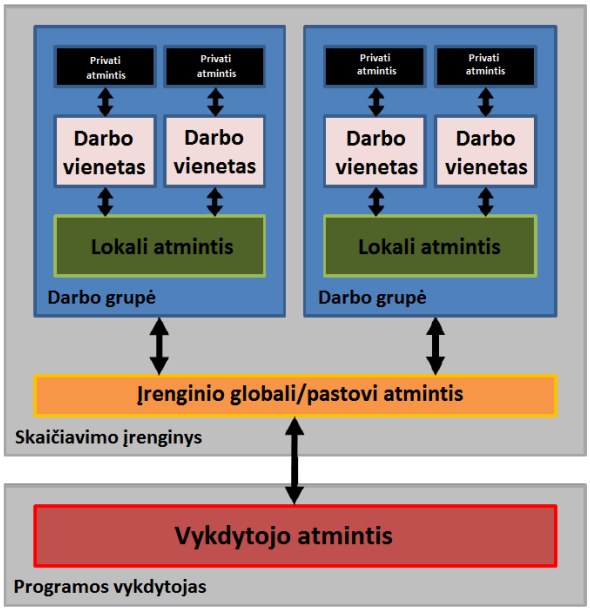
OpenCL darbo vienetų sinchronizavimo galimybės apima grafinių procesorių sinchronizacijos galimybes, kurios buvo aptartos 2.2. skyrelyje, bei praplečia jas suteikiant papildomas galimybes kontroliuoti darbo vienetų atlikimo tvarką panaudojant vykdymo eiles, t. y. darbo vienetai gali būti sinchronizuojami:

- panaudojant barjero komandas branduolio funkcijoje, jei darbo vienetai priklauso vienai darbo grupei [AMD13];
- jų vykdymo prašymus dedant į tvarką išlaikančią vykdymo eilę ir joje naudojant barjerą [AMD13];

- jų vykdymą koordinuojant įvykiais, t. y. įdėjus prašymą vykdyti darbo vienetą į vykdymo eilę, yra grąžinamas įvykio objektas, kurio metodai grąžiną jį atitinkančio darbo įvykio būseną, kuri gali būti naudojam sinchronizacijai [AMD13].

1.3.5. Atminties modelis

OpenCL atminties modelis pavaizduotas 12 paveikslėlyje. Iš jo matyti, kad modelis atitinka 2.2. skyrelyje aptartą grafinių įrenginių atminties hierarchiją, t. y. modelį sudaro 5 sritys:



12 pav. OpenCL atminties modelis [TS12]

- globali atmintis – tai regionas, į kurį gali rašyti ir iš kurio gali skaityti visi darbo vienetai, priklausantys vienam skaičiavimo įrenginiui, bei programos vykdytojas [TS12];
- pastovi atmintis – tai regionas, kuriame duomenys nesikeičia visą branduolio vykdymo laiką, darbo vienetai iš jo gali tik skaityti, o vykdytojas – skaityti ir rašyti [TS12];
- lokali atmintis – tai regionas, naudojamas darbo vienetų priklausančių vienai darbo grupei. Jame darbo vienetai gali ir rašyti ir skaityti [TS12];
- privati atmintis – tai regionas, kuris yra prieinamas tik darbo vieneto skaitymui ir rašymui [TS12];
- vykdytojo atmintis – tai regionas, kuris paprastai yra pasiekiamas tik vykdytojo, todėl darbo vienetai priėjimo prie jo neturi, o duomenys iš šio regiono turi būti perkopijuojami į skaičiavimo įrenginio atmintį pačio vykdytojo prieš pradedant darbo vienetų vykdymą [TS12].

1.3.6. Tipinis programų vykdymo šablonas

Daugumos OpenCL programų vykdymo tvarka remiasi įprastu šablonu, kuris susideda iš tokių operacijų:

- naudojantis OpenCL programavimo sąsaja, surandami bei išsirenkami norimi skaičiavimo įrenginiai;
- pasirinktiems įrenginiams sukuriamas vykdymo kontekstas;

- naudojantis vykdymo kontekstu kiekviename skaičiavimo įrenginyje atliekamos šios operacijos:
  - sukuriama viena ar daugiau vykdymo komandų eilių;
  - sukuriamos programos vykdymui skaičiavimo įrenginyje;
  - sukuriamos branduolio funkcijos sukurtose programose;
  - skaičiavimo įrenginyje išskiriami globalios atminties regionai;
  - duomenys rašomi į skaičiavimo įrenginyje išskirtą atminties regioną;
  - branduolio funkcijos įdedamos į skaičiavimo įrenginio komandų eilę;
  - duomenys perkopijuojami iš skaičiavimo įrenginio globalios atminties į vykdytojo atmintį.

1.4. Sudėtingų įvykio apdorojimo sistemos realizacija

Atsižvelgiant į 2 skyriuje aptartus centrinio bei grafinio procesorių skirtumus bei 1 skyriuje suformuotą sudėtingų įvykių apdorojimo sistemos modelį, šiame skyriuje bus aptariami modelio realizacijai reikalingi sprendimai bei algoritmai.

1.4.1. Kalbos modelis trigeriams apibrėžti

Kadangi 1 skyriuje aptarta sistema modeliuojama kaip tęstinių užklausų vykdymas duomenų srautams, tai reikalingas būdas tęstinėms užklausoms aprašyti. Tam labai tinka tradicinių duomenų bazių užklausom aprašyti naudojama SQL kalba, tačiau, siekiant, kad ji atitiktų duomenų srautams vykdomų užklausų reikalavimus, reiktų ją šiek tiek modifikuoti kaip tai buvo padaryta, kuriant SASE įvykių kalbą [WDR06].

CEP modelio trigeriams nereikia visų SQL kalbos žodžių, nes jie negeneruoja užklausos atsakymo duomenų srautų, o tik arba sukuria naują aukštesnio lygio įvykį, arba įvykdo apibrėžtą veiksmą, todėl užtenka tik poaibio, leidžiančio apibrėžti predikatus, nusakančius tam tikro įvykio atsitikimą, kurių kaip šablono kiekvienas CEP modelio trigeris ieškos įvesties duomenų srautų gaunamose sekose. Kiekviena sraute gaunama duomenų seka, nusako vieną įvykį, todėl pats įvykis gali būti aprašytas duomenų struktūra, kaip pavaizduota 13 paveikslėlyje, kurios duomenų laukų reikšmės atmintyje bus tam tikrais poslinkiais atidėtos bitų aibės, atitinkančios kiekvieną įvesties duomenų srautuose gaunamą duomenų seką [WDR06]. Kiekvienas trigeris žinos įvykio, kurio duomenų reikšmės jam reikalingos, tipą, o pagal tai galės iš gautos duomenų sekos pasiimti to įvykio tipo duomenų laukų reikšmes pritaikęs atitinkamus poslinkius. Svarbu, kad kiekvieną įvykį atitinkanti duomenų seka įvesties srautuose galėtų būti identifikuojama kaip tam tikro tipo įvykis, kuris galėtų būti priskirtas atitinkamo trigerio apdorojimui, bei turėtų laiko

žymę, kuri nusakytų kada įvykis buvo užfiksuotas, bet šios problemos lengvai išsprendžiamos nustatant reikalavimą, kad pirmieji duomenų sekos 4 baitai yra natūralusis skaičius identifikuojantis įvykio tipą, o dar 4 sekantys baitai skirti laiko žymės reikšmei.

13 pav. Įvykių tipų apibrėžimai

```
struct A                struct B
    {                       {
    int _id;                int _id;
    time_t _laikas;         time_t _laikas;
    char* _a                int _b1
    };                      char* _b2
                            };
```

14 pav. Trigerio apibrėžimas

```
EVENT C

WHERE A._a = 'Padidėjo'    AND
        B._b2 = 'Sumažėjo' AND
        [FIRST\LAST\EACH]B WITHIN 12min FROM A

SELECT B._b1

EXECUTE [Vartotojo funkcijos apibrėžimas]
```

Taigi trigerių apibrėžimo kalbos modeliui užtektų šių raktažodžių:

- EVENT – nusako įvykio tipą;
- WHERE – nusako predikatų aibę, kuri signalizuoja, kad aprašytasis įvykis įvyko:
  - AND/OR/[kiti predikatų operatoriai] – atitinka SQL operatorių funkcionalumą;
  - [FIRST\LAST\EACH] X WITHIN Y FROM A – apibrėžia, kad predikatuose naudojamas įvykis B turėtų būti įvykęs ne anksčiau ir ne vėliau nei Y laiko vienetų po arba prieš įvykį A. Jei tokių įvykių įvyko daugiau nei vienas, tai FIRST – tikrina predikatus tik pirmajam, LAST – paskutiniam, o EACH – kiekvienam tame intervale įvykusiam įvykiui B.
- SELECT – nurodo kurios žemesnio lygio įvykių atributų reikšmės turėtų būti išrenkamos ir perduodamos kaip argumentai trigerio funkcijai;
- EXECUTE – nurodo trigerio funkciją, kuri turėtų būti vykdoma, radus predikatus tenkinančius įvykius įvesties sraute.

Tokia kalba apibrėžtas trigeris, naudojantis 13 paveikslėlyje apibrėžtus paprastesnių įvykių tipus, yra pavaizduotas 14 paveikslėlyje. Vartotojo apibrėžta trigerio funkcija gautų SELECT išraiškoje apibrėžtas reikšmes masyve ta pačia tvarka, kokia jos yra išvardintos, ir galėtų arba sukonstruoti nauja sudėtingesnį įvykį C, kuriam turėtų būti apibrėžimas kaip įvykiams 13 paveikslėlyje ir kuris būtų grąžinamas į įvesties srautą tolesniam apdorojimui, arba atlikti kažkokius papildomus veiksmus.

1.4.2. Įvesties srauto sekų skaidymas

Kadangi CEP sistema taiko tęstines užklausas duomenų srautams, kurie neturi dydžio apribojimų, t. y. gali būti begaliniai, o 3 skyriuje buvo aptarta, kad CEP sistemos programos vykdytojas turi išskirti tam tikra atminties kiekį skaičiavimo įrenginyje ir tik tokio dydžio

duomenų aibę gali perkopijuoti, tai atsiranda poreikis duomenų srautuose gaunamus duomenis skaidyti baigtinėmis, apibrėžto dydžio porcijomis, kurios galėtų būti persiunčiamos skaičiavimo įrenginiui su ribotu kiekiu atminties apdoroti. Pagal tai kokiais kriterijais remiantis yra skaidomos duomenų srautais atkeliaujančios duomenų sekos yra išskiriamos trys strategijos:

- duomenų sekų skaičiumi paremta strategija – tai strategija, kuri skaido duomenų srauto elementus į vienodo dydžio aibes, todėl yra žinoma, kad kiekvienoje duomenų porcijoje bus fiksuotas skaičius elementų, tačiau laiko intervalai per kuriuos duomenys bus pateikiami apdorojimui yra kintantys [MBS15];
- laiko intervalais paremta strategija – yra atvirkštinė aukščiau aprašytai, t. y. yra žinoma, kad duomenys bus apdorojami kas fiksuotą laiko tarpą, tačiau nežinoma kiek elementų bus duomenų aibėje [MBS15];
- konkretaus elemento laukimu paremta strategija – tai strategija, kai duomenų sraute yra laukiama terminalinio elemento, kuris žymi, jog reikia pradėti duomenų apdorojimą. Šiuo atveju nei laiko intervalai, nei duomenų elementų skaičius porcijoje yra nežinomi, kintantys dydžiai [CM12].

1.4.3. CEP apdorojimo algoritmai

CEP apdorojimo sistemai svarbu, kad duomenų srautais gaunamos duomenų sekos, atitinkančios įvykius, būtų surūšiuotos laiko žymės didėjimo tvarka. Šiame darbe bus laikoma, kad srautais gaunami įvykiai jau yra surūšiuoti jų įvykimo laiko žymės didėjimo tvarka. Sistemose, kuriose taip nėra, gali būti realizuotas įvesties duomenų valdytojas, kuris savo ruožtu surūšiuotų gaunamus duomenis pagal laiko žymes didėjimo tvarka, o jei tokių nėra pats jas priskirtų, kaip tai aptarta [SW04] darbe.

1.4.3.1. Nedeterminuotais baigtiniais automatais paremtas apdorojimas

Dauguma CEP sistemų sudėtingų įvykių atpažinimui naudoja inkrementinius metodus, paremtus tęstinių užklausų modeliavimu determinuotais baigtiniais automatais, kurie laikinai laiko tarpinius užklausų rezultatus išvedant sudėtingus įvykius iš primityvių [Hir12, ABB+04, CM12].

Pradžioje algoritmas visas tęstines užklausas, naudojamas trigeriui apibrėžti, sumodeliuoja determinuotais baigtiniais automatais tokiu būdu:

- kiekvienas įvykis, naudojamas predikatuose, virsta automato būsena [CM12];

- perėjimas tarp dvejų automato būsenų, atitinkančių įvykius, apibrėžiamas laiko ir įvykio duomenų turinio apribojimais, kuriuos turi tenkinti naujai gautas įvykis, kad automatas pereitų į pastarąjį įvykį atitinkančią būseną [CM12].

Paskui algoritmas sukuria kiekvieno automato modelio objektą, kuris laukia kol bus gautas įvykis, leidžiantis automato objektui pereiti į sekančią būseną. Šis algoritmas remiasi pirmąja 4.2. poskyryje aptarta įvesties duomenų srauto strategija, t. y. srautas yra skaidomas fiksuotu skaičiumi elementų, kurio dydis yra 1, kitaip sakant algoritmas apdoroja kiekvieną duomenų seką vos tik ji yra gaunama. Kai yra gaunamas naujas įvykis *e* algoritmas atlieka tokius žingsnius:

1. suranda visus automatų modelius, kurie gali pakeisti būseną dėl įvykio *e* [CM12];
2. tuomet perrenkami visi kiekvieno 1. žingsnyje surasto automato modelio egzistuojantys objektai, patikrinant ar tenkinami automato modelio apibrėžti apribojimai įvykiui *e*, kad automatas galėtų pereiti į kitą būseną [CM12].

Gali būti keletas automato modelio objektų laukiančių toje pačioje būsenoje ir kiekvienam tokiam objektui gavus *e* algoritmas atlieka šiuos žingsnius:

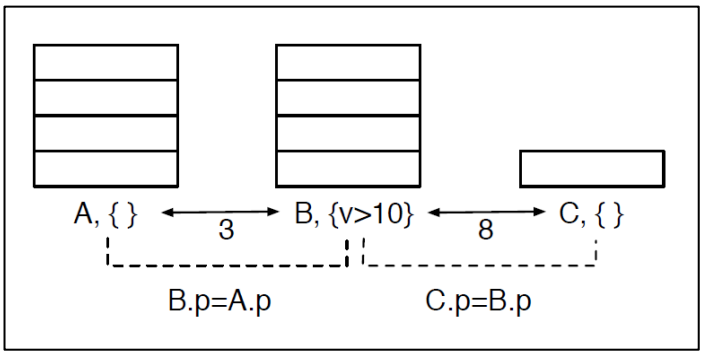
i. patikrina kiekvieno automato modelio objekto laiko apribojimus ir, jei jie nėra tenkinami, sunaikina automato objektą, nes pastarasis jau niekada nebegalės pereiti į kitą būseną (pradinės būsenos objektams šis apribojimas negalioja) [CM12];
ii. patikrina kiekvieno automato modelio objekto turinio apribojimus [CM12];
iii. jei įvykis *e* tenkina visas konkretaus modelio objekto sąlygas, tuomet yra sukuriama šio objekto kopija, kuri panaudodama įvykį *e*, t. y. išsisaugodama gauto įvykio reikiamus duomenis, pereina į įvykį *e* atitinkančią būseną, o originalus objektas lieka, laukdamas kitų įvykių, kurie vėl galėtų jį pervesti į kitą būseną arba kol pats objektas bus sunaikintas dėl laiko apribojimų [CM12];
iv. jei nukopijuotas automato modelio objektas pereidamas į kitą būseną pasiekė galutinę būseną, jis pridedamas į objektų pasiekusių baigtinę būseną sąrašą, kitu atveju naujasis objektas pridedamas į automato modelio objektų sąrašą, kur lauks kitų įvykių [CM12];
v. jei objektų pasiekusių baigtinę būseną sąrašas nėra tuščias, iš kiekvieno sąraše esančio objekto yra sukuriamas tą automato modelį atitinkantis sudėtingesnis įvykis, panaudojant automato modelio objekte išsaugotus duomenis, kurie buvo surinkti keičiant būsenas [CM12];
vi. atlikus v. žingsnį objektų pasiekusių baigtinę būseną sąrašas yra ištuštinamas ir algoritmas laukia kito įvykio *e* [CM12].

1.4.3.2. Stulpeliais paremtas uždelstas apdorojimas

Nors 4.3.1. skirsnyje aprašytas algoritmas gerai tinka realizacijai vykdomai centriniais procesoriais, tačiau ją gan sudėtinga efektyviai realizuoti vykdymui grafiniais procesoriais, todėl šiame skirsnyje bus pateikiamas alternatyvus algoritmas geriau tinkantis grafiniams procesoriams [CM12]. Šis algoritmas naudojasi 4.2. poskyryje aptarta trečiąja duomenų srautų skaidymo strategija, t. y. laukia tam tikro elemento, kad pradėtų duomenų apdorojimą, todėl yra vadinamas uždelstu ir geriau derinasi su duomenų perkopijavimo tarp vykdytojo ir skaičiavimo įrenginio atminčių reikalavimu. Algoritmas yra stulpeliais paremtas, nes kiekvieną tęstinę užklausą modeliuoja stulpelių seka, kurioje kiekvienas stulpelis yra sąrašas vieno tipo įvykiams, kurie naudojami užklausos predikatuose, laikyti, o įvykių laiko ir turinio apribojimai yra modeliuojami sąryšiais tarp stulpelių. 15 paveikslėlyje apibrėžta tęstinė užklausa atitiktų 16 paveikslėlyje pavaizduotą stulpelių seką, rodyklės atitiktų laiko sąlygas tarp stulpelius atitinkančių įvykių, o brūkšninės linijos – turinio sąlygas.

```
EVENT X

WHERE C.p = B.p AND
      C.p = A.p AND
      B.v > 10 AND
      B WITHIN 8min FROM C AND
      A WITHIN 3min FROM B
EXECUTE Z
```

15 pav. Tęstinės užklausos apibrėžimas [CM12]



16 pav. Tęstinės užklausos modeliavimas stulpeliais [CM12]

Kai naujas įvykis *e* yra gaunamas, algoritmas suranda visas užklausų stulpelių sekas, kuriose yra stulpelis, kuris skirtas *e* tipo įvykiams laikyti ir kurio laiko ir turinio sąlygas atitinka gautais įvykis *e*. Tuomet pastarasis įvykis pridedamas į kiekvieną iš surastų sąrašų. Jei tarp rastų stulpelių pasitaiko tokių, kurie yra stulpelių sekos paskutinieji, tai pradedamas iki šiol į stulpelius sudėtų įvykių apdorojimas, t. y. algoritmas kiekvienai iš tokį stulpelį turinčių stulpelių sekų atlieka šiuos žingsnius, eidamas nuo paskutinio stulpelio sekoje iki pirmo [CM12]:

i. yra peržiūrimas kiekvienas stulpelių sekos stulpelis ir iš jo išmetami seni įvykiai, kurie nebetenkina laiko ar turinio sąlygų, apibrėžtų tęstinės užklausos [CM12];
ii. yra konstruojamos dalinės išvedimo sekos. Kiekviename žingsnyje algoritmas lygina jau sugeneruotas dalines išvedimo sekas su įvykiais kiekviename stulpelyje ir kai įvykis ar įvykių aibė tenkina užklausos laiko bei turinio sąlygas yra

sukonstruojama nauja išvedimo seka, įtraukianti pastarąjį įvykį ar įvykių aibę. Naujai sukonstruota seka yra naudojama su tolesniame stulpelyje esančiais įvykiais konstruoti naujas sekas [CM12];

iii. jei sukonstruojama išvedimo seka, kurioje visi iš stulpelių paimti įvykiai tenkina užklausos sąlygas ji dedama į teisingų sekų sąrašą;

iv. su kiekviena seka iš teisingų sekų sąrašo yra konstruojami sudėtingesni įvykiai [CM12].

1.5. Skyriaus išvados

Atlikus analitinės dalies apžvalgą, reiktų išskirti šias pagrindines skyriaus išvadas:

- sudėtingų įvykių apdorojimo sistemų veikimui yra būdingas tų pačių instrukcijų vykdymas skirtingiems duomenims, t. y. tikrinimas ar tas pačias, sudėtingą įvykį aprašančioje taisyklėje apibrėžtas, sąlygas tenkina skirtingų, gautų primityvių įvykių atributų reikšmės;
- grafinių procesorių architektūra yra suprojektuota taip, kad sparčiai būtų atliekama daug tų pačių instrukcijų skirtingiems duomenims lygiagrečiai, todėl, išnaudojant šių procesorių galimybes, galima tikėtis didesnės apdorojimo spartos nei apdorojant įvykius su centriniais procesoriais;
- siekiant išgauti kuo didesnę spartą lygiagrečiai apdorojant skirtingus duomenis, grafinių procesorių architektūriniai sprendimai reikalauja sudėtingesnių sinchronizacijos ir atminties valdymo mechanizmų bei darbo vienetų skirstymo į tam tikro dydžio, nuo grafinio procesoriaus fizinių charakteristikų priklausančias, darbo grupes. Šių uždavinių įgyvendinimui OpenCL karkasas pateikia abstrakčią programavimo sąsają, leidžiančią gauti visą reikiamą informaciją apie fizinę įrangą bei pateikti atitinkamas komandas darbo vienetų skirstymui ir planavimui;
- siekiant atlikti kuo daugiau skaičiavimų lygiagrečiai, labiau tinkama strategija yra gautus įvykius ne iš karto apdoroti, o kaupti atidėtam apdorojimui, todėl grafinių procesorių vykdomoms sistemoms stulpeliais paremtas uždelstas apdorojimas yra tinkamiausias.

2. Tiriamoji dalis

Siekiant ištirti grafinių procesorių panaudojimo galimybes sudėtingų įvykių atpažinimui, buvo pasirinkta atlikti palyginimą tarp dviejų apdorojimo modelių – vieno, kurio apdorojimas realizuotas determinuotų automatų pagrindu su uždelstu apdorojimu ir pritaikytas vykdymui su centriniu procesoriumi, ir kito, kuris realizuotas stulpelių apdorojimo pagrindu su uždelstu apdorojimu ir pritaikytas apdorojimui su grafiniu procesoriumi.

Kadangi šiame darbe norėta gauti kuo reprezentatyvesnius rezultatus, tai palyginimą buvo nuspręsta atlikti lyginant TRex modelį, vykdomą centrinio procesoriaus, su savos realizacijos modeliu, skirtu vykdymui su grafiniu procesoriumi. TRex modelį buvo pasirinkta naudoti dar ir dėl šių priežasčių:

- tai yra gerai žinoma sudėtingų įvykių atpažinimo sistema, kuri savo sparta nenusileidžia komerciniams modeliams, tokiems kaip Esper [CM12];
- naudojant jau realizuotą sudėtingų įvykių apdorojimo sistemą, buvo aiškiai apibrėžti sistemos funkcionalumo reikalavimai;
- naudojant sistemą, kuri jau buvo naudojama spartos palyginimams su komerciniais produktais, buvo galima sulyginti tyrimui realizuotos sistemos generuojamus rezultatus su TRex rezultatais ir įsitikinti, kad yra gaunami korektiški rezultatai.

Abiejų sistemų realizacijos palaiko praktiškai visas sudėtingų įvykių sistemų operatorių galimybes, todėl darbe buvo atrinktos pagrindinės, kurios turi daugiausiai reikšmės apdorojimo spartai, t. y. palyginimas buvo atliekamas keičiant šių kintamųjų reikšmes:

- taisyklių skaičius;
- predikatų skaičius taisyklėje;
- atributų skaičius predikatuose;
- parametrizuotų atributų skaičius taisyklėje;
- agregatinių atributų skaičius rezultate;
- langų dydis;
- daugelio išrinkimo predikatų skaičiaus taisyklėje.

Pagrindinis palyginimo kriterijus – fiksuoto dydžio primityvių įvykių aibės apdorojimo sparta, t. y. laikas sugaištas apdorojant primityvius įvykius ir generuojant sudėtingus įvykius.

2.1. GPU modelio realizacija

GPU modelio realizacijai buvo pasirinktas OpenCL 2.0 karkasas, nes šios versijos ir šią versiją palaikančio grafinio procesoriaus galimybės, tokios kaip bendra virtuali atmintis (angl. shared virtual memory) ir vidinis paralelizmas (angl. nested parallelism), leidžia atlikti papildomus apdorojimo žingsnius su grafiniu procesoriumi, negrąžinant vykdymo kontrolės centriniam procesoriui, tokiu būdu sutaupant laiko, bei apsibrėžti sudėtingas, nuorodomis į kitus atminties blokus užpildytas duomenų struktūras, kurios yra lengviau suprantamos programuotojui ir daro programos išeities kodą lengviau skaitomu ir prižiūrimu. Būtent ši technologija buvo pasirinkta naudojimui šiame darbe dar ir dėl to, kad OpenCL karkasas nepriklauso nuo konkrečios vienos platformos, t. y., priešingai nei alternatyvios technologijos, tokios kaip CUDA ar Metal, jis yra palaikomas visų populiariausių operacinių sistemų ir yra realizuotas bei pateikiamas naudojimui visų populiariausių procesorių gamintojų, įskaitant ir tyrimo eksperimentuose naudojamo grafinio procesoriaus gamintojo. Taip pat OpenCL išnaudojančias programas ir OpenCL branduolio funkcijas realizuojantis kodas yra lengviau pernešamas ir panaudojamas, vykdant jį skirtingose platformose.

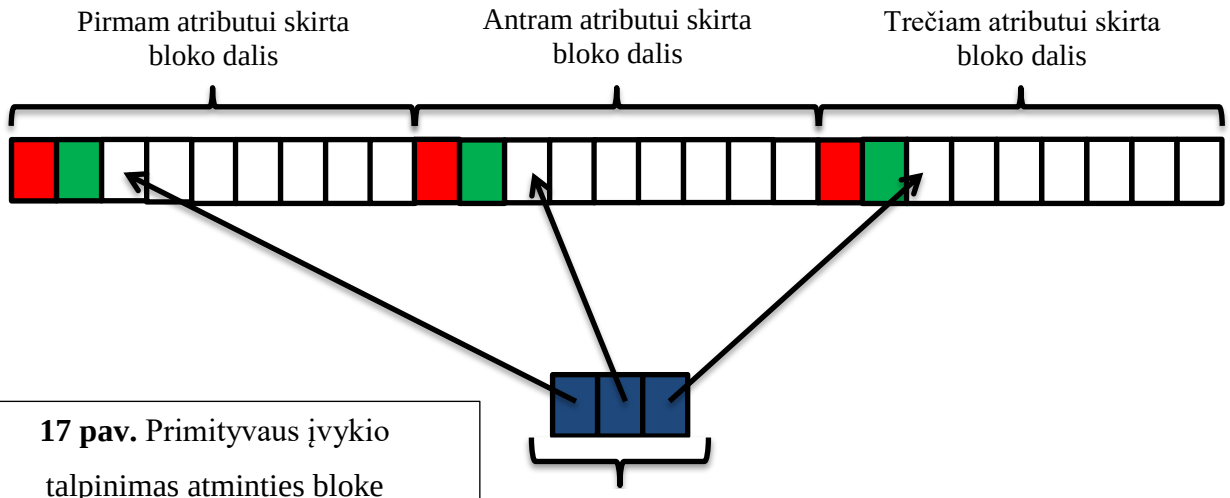
GPU modelį sudaro du pagrindiniai komponentai:

- grafinio procesoriaus atminties valdiklis, atsakingas už visas operacijas, susijusias su grafinio procesoriaus atmintimi, t. y.:
    - taisyklių konvertavimą iš TRex vidinių duomenų struktūrų į grafiniam procesoriui palankesnes struktūras, kuriose mažiau nuorodų į kitus atminties blokus ir atminties skaitymas gali būti susietas tarp gijų, priklausančių vienai gijų bangai;
    - primityvių įvykių konvertavimą į grafiniam procesoriui pritaikytą formą, kurioje visi primityvaus įvykio atributų vardai yra koduojami sveikaisiais skaičiais, o atributų reikšmės yra slankiojo kablelio skaičių tipo;
    - konvertuotų primityvių įvykių nusiuntimą į grafinio procesoriaus atmintį;
    - tarpiniams skaičiavimams reikalingos atminties valdymą bei tarpinių skaičiavimų rezultatų persiuntimą iš pagrindinės sistemos operatyviosios atminties į grafinio procesoriaus atmintį ir atgal;
    - galutinių rezultatų persiuntimą į pagrindinę sistemos operatyviąją atmintį ir jų konvertavimas į TRex sistemos struktūrą.
- primityvių įvykių procesorius, kurio pagrindinės užduotys:
    - sukonfigūruoti ir paleisti atitinkamą OpenCL branduolio funkciją su pradiniais ar tarpiniais skaičiavimo rezultatais;

- gauti ir išsaugoti tolimesnius tarpinius rezultatus;
- vėl sukonfigūruoti ir paleisti atitinkamą branduolio funkciją, kol gaunamas galutinis atsakymas.

2.1.1. GPU atminties valdymas

Grafinio procesoriaus atmintis gijų bangų yra skaitoma susietai, t. y. norint kad gijų banga gautų norimus duomenis iš atminties greitai – per vieną transakciją, reikia, kad arba visos gijos, sudarančios gijų bangą, skaitytų tą patį atminties adresą, arba skaitytų greta esančius adresus. Taip pat gijų bangos, vykdomos skirtingų skaičiavimo vienetų, idealiu atveju turėtų skaityti atmintį esančią skirtinguose atminties bankuose, nes tą patį atminties banką, pasiekiamą per vieną kanalą, vienu metu skaityti gali tik vienas skaičiavimo vienetas. Atsižvelgiant į šiuos reikalavimus, grafinio procesoriaus atmintyje su kiekvienu kiekvienos taisyklės primityviu įvykiu yra susiejamas atminties blokas, skirtas talpinti to tipo primityvių įvykių atributų reikšmes. Atminties blokai yra fiksuoto dydžio, t. y. gali talpinti tam tikrą fiksuotą skaičių įvykių ir yra realizuoti kaip cikliniai buferiai. Taip pat, persiuntus primityvų įvykį į grafinio procesoriaus atmintį, jo atributų reikšmės yra išskaidomos bloke taip, kad gretimose atminties ląstelėse būtų skirtingų įvykių to paties atributo reikšmės, kaip pavaizduota 17 paveikslėlyje.



17 pav. Primityvaus įvykio talpinimas atminties bloke

Tokiu būdu yra išnaudojama daugiau atminties, tačiau padidinama tikimybė, jog gijos apdorojančios skirtingus taisyklės stulpelius, skaitys skirtingus atminties blokus ir galės atminties skaitymą vykdyti paraleliai. Taip pat toks primityvių įvykių atributų reikšmių rašymas užtrunka ilgiau, bet, kadangi TRex realizuoja ir darbe nagrinėjame serverio – kliento modelį bei primityvių įvykių apdorojimas yra uždelstas, t. y. įvykiai yra pradedami apdoroti tik gavus šakninį taisyklės įvykį, tai sugaištas laikas atributų išskaidymui atsiperka, nes tikrinant ar įvykis atitinka taisyklės apribojimus, galima paleisti gijų bangą, kurios dydis sutampa su stulpelyje

esančių primityvių įvykių skaičiumi, visos gijų bangos gijos vykdo tą patį kodą, t. y. tikrina tą pačią sąlygą, bei joms visom reikalingos atminties reikšmės yra greta. Taip pat, primityvaus įvykio siuntimas į grafinio procesoriaus atmintį ir kopijavimas į skirtingus atminties blokus vyksta neblokuojant centrinio procesoriaus ir naudojant vidinį paralelizmą, t. y. persiuntus įvykį į grafinio procesoriaus atmintį, yra paleidžiama gijų banga, kuri pati paleidžia tiek kitų gijų bangų kiek yra atminties blokų į kuriuos reikia perkopijuoti siunčiamą primityvų įvykį. Kadangi pastarosios gijų bangos priklauso atskiroms darbo grupėms, t. y. gali būti vykdomos skirtingų skaičiavimo vienetų, ir atlieka atminties operacijas su skirtingais atminties blokais, dauguma perkopijavimų gali vykti paraleliai.

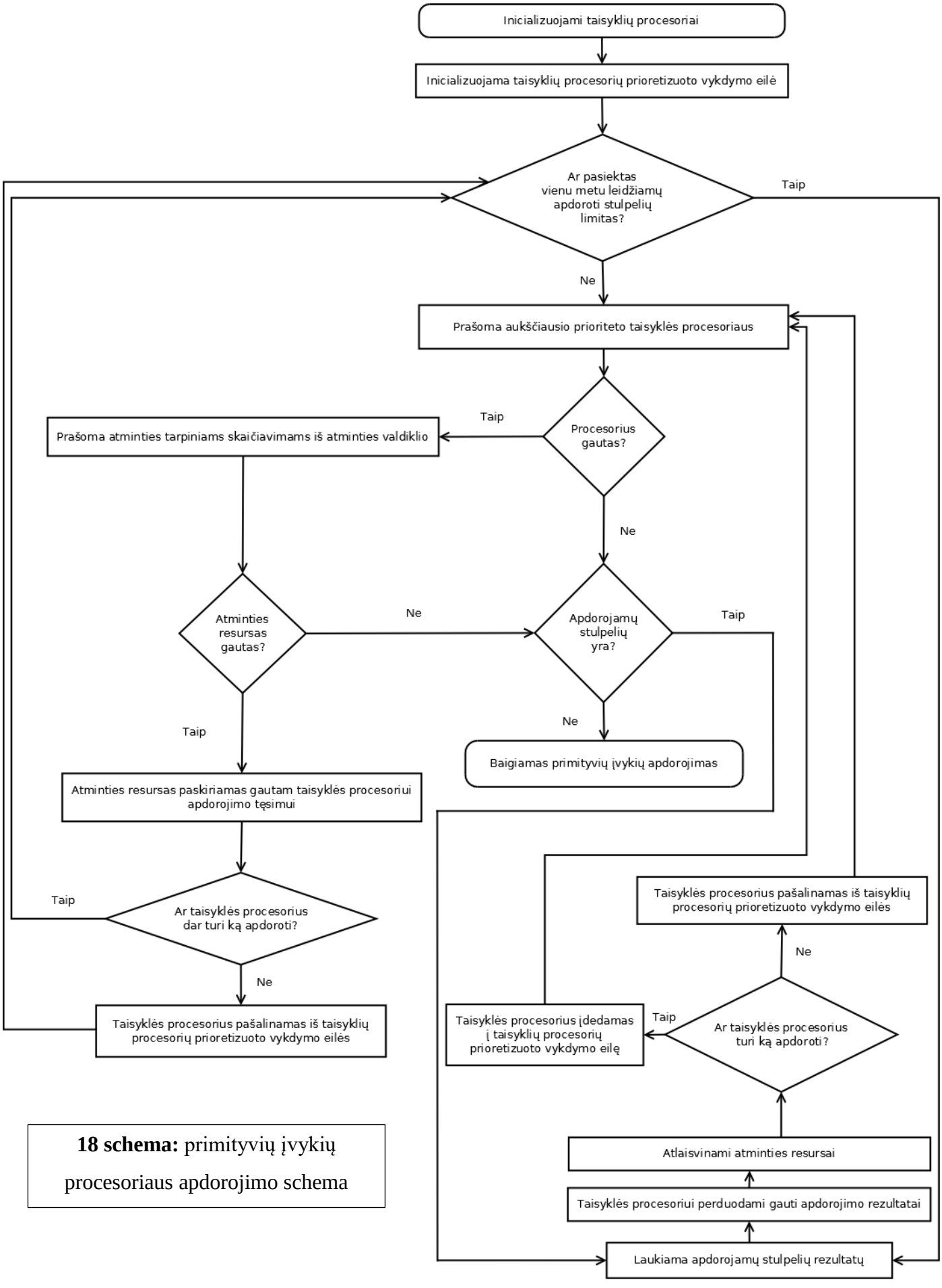
2.1.2. Primityvių įvykių procesorius

Primityvių įvykių procesorius yra atsakingas už primityvių įvykių taisyklių stulpeliuose apdorojimo koordinavimą ir teisingą atminties resursų, gaunamų iš atminties valdiklio, paskirstymą, kad per daug atminties nebūtų sunaudojama tarpinių rezultatų saugojimui. Jį sudaro šie vidiniai komponentai:

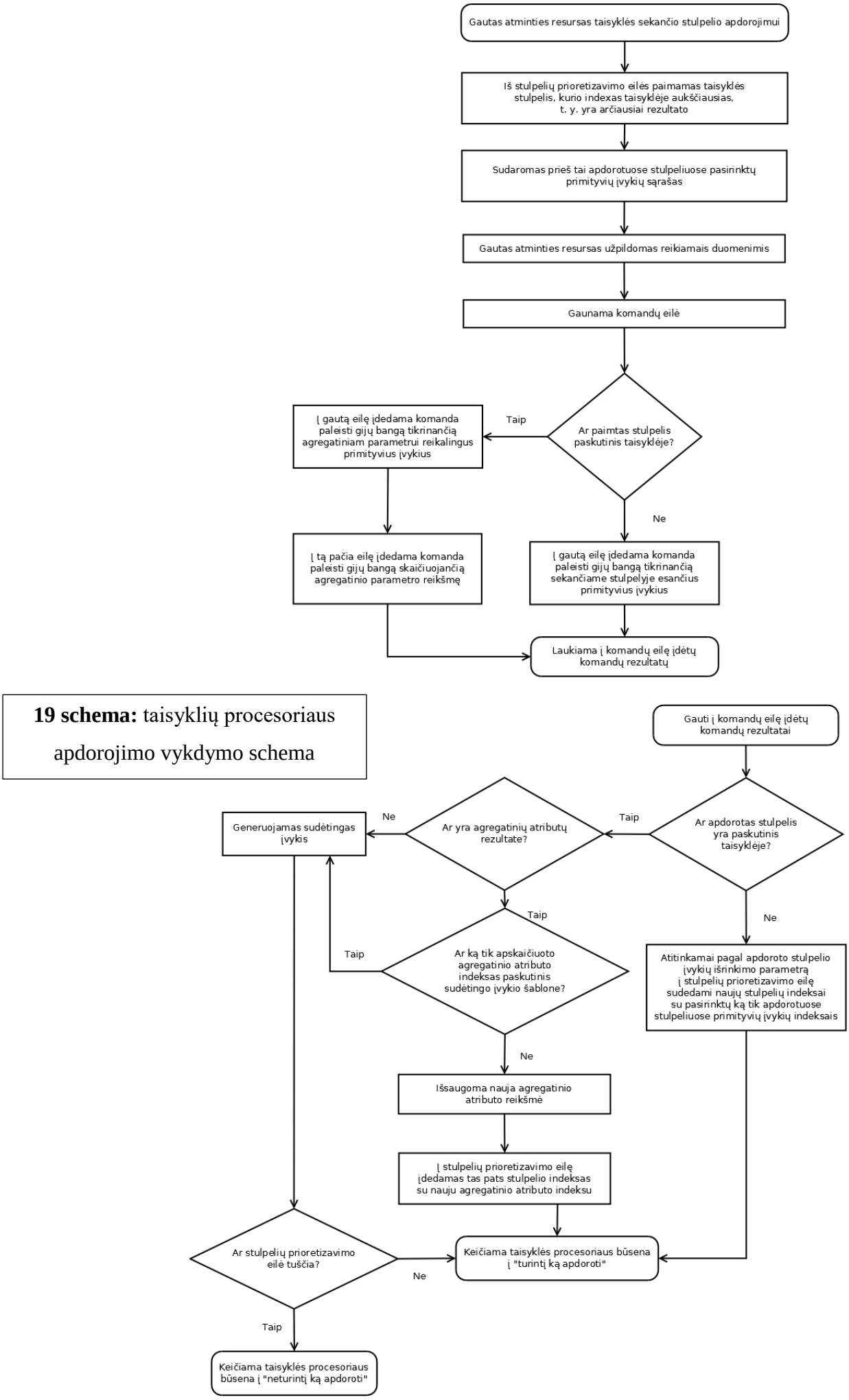
- branduolio funkcijų valdiklis;
- indeksavimo lentelė;
- taisyklių procesoriai;
- taisyklių procesorių prioretizuoto vykdymo eilė;
- OpenCL komandų eilių valdiklis.

Branduolio funkcijų valdiklio paskirtis yra rūpintis branduolio funkcijų objektų gyvavimo trukme. Indeksavimo lentelė yra naudojama primityvių įvykių persiuntimo metu, kad greitai pavyktų gauti visus atminties blokus, t. y. stulpelius, į kuriuos turėtų būti perkopijuotas gautas primityvus įvykis. Likę trys komponentai yra skirti apdorojimo koordinavimui ir branduolio funkcijų paleidimui, panaudojant OpenCL komandų eilių valdikį. Proceso schema yra pavaizduota 18 ir 19 schemose. OpenCL komandų eilių valdiklis yra reikalingas, nes OpenCL specifikacija griežtai nereikalauja asinchroninių komandų eilių realizacijos, todėl AMD programinės įrangos kūrimo paketas to nerealizuoja, o norint išnaudoti grafinių procesorių paralelizmo galimybes reikia galimybės paleisti daug gijų bangų, vykdančių tam tikras branduolių funkcijas tuo pat metu. Pavyzdžiui, turint daug taisyklių, daug gijų bangų gali tuo pat metu apdoroti skirtingų taisyklių skirtingus stulpelius, viena kitai netrukdydamos, nes kiekvienos taisyklės kiekvienas stulpelis turi savo atskirą primityvių įvykių atributų reikšmių atminties bloką. Tam tikslui pasiekti OpenCL komandų eilių valdiklis sukuria daug OpenCL komandų

eilių ir komandas, kurios gali būti vykdomos lygiagrečiai, ciklišk ai paskiria skirtingoms komandų eilėms.



18 schema: primityvių įvykių procesoriaus apdorojimo schema

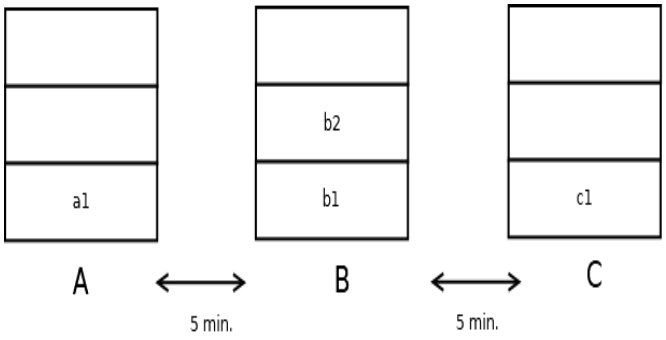


19 schema: taisyklių procesoriaus apdorojimo vykdymo schema

18 schemoje primityvių įvykių procesoriaus naudojama taisyklių procesorių prioretizuoto vykdymo eilė yra standartinė aibė, realizuota kaip raudonai juodas medis. Jos pagrindinis tikslas yra pirmenybę teikti tiems taisyklių procesoriams, kurie jau yra apdoroję daugiausiai taisyklės stulpelių, kad atmintyje nebūtų bereikalingai saugomi tarpiniai skaičiavimų duomenys, kurių prireiks tik gerokai vėliau. Dėl tos pačios priežasties yra naudojama stulpelių prioretizavimo eilė taisyklių procesoriuje. Pats taisyklių procesorius tarpinius rezultatus, t. y. apdoroto stulpelio indeksą su tame stulpelyje pasirinkto primityvaus įvykio indeksu, saugo vienoje duomenų struktūroje – medžio mazge, kuris turi nuorodą į kitą, medžio mazgą, atitinkantį prieš tai apdoroto stulpelio rezultatą. Taigi taisyklių procesoriaus stulpelių prioretizavimo eilėje yra talpinami medžio mazgai, kuriais einant galima sukonstruoti kiekviename stulpelyje pasirinktų primityvių įvykių seką nuo pat šakninio taisyklės įvykio.

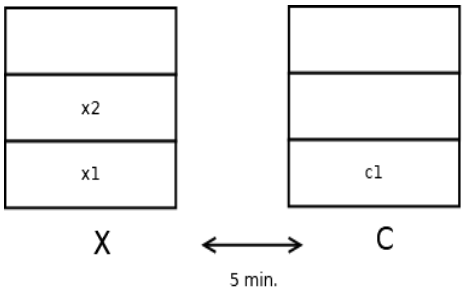
Siekiant aiškiau perteikti 18 ir 19 schemose aprašytus įvykių apdorojimo procesus, toliau darbe pateikiamas elementarus pavyzdys, parodantis kaip kinta skirtingų primityvių įvykių procesoriaus komponentų būsenos vykdant įvykių apdorojimą.

```
EVENT D
WHERE
        EACH B WITHIN 5min FROM C AND
        EACH A WITHIN 5min FROM B
SELECT
        D.d = AVG (B.b) EACH B WITHIN 5min FROM C
```



20 pav. Tęstinės užklausos *T1* apibrėžimas ir užpildyti primityvių įvykių stulpeliai

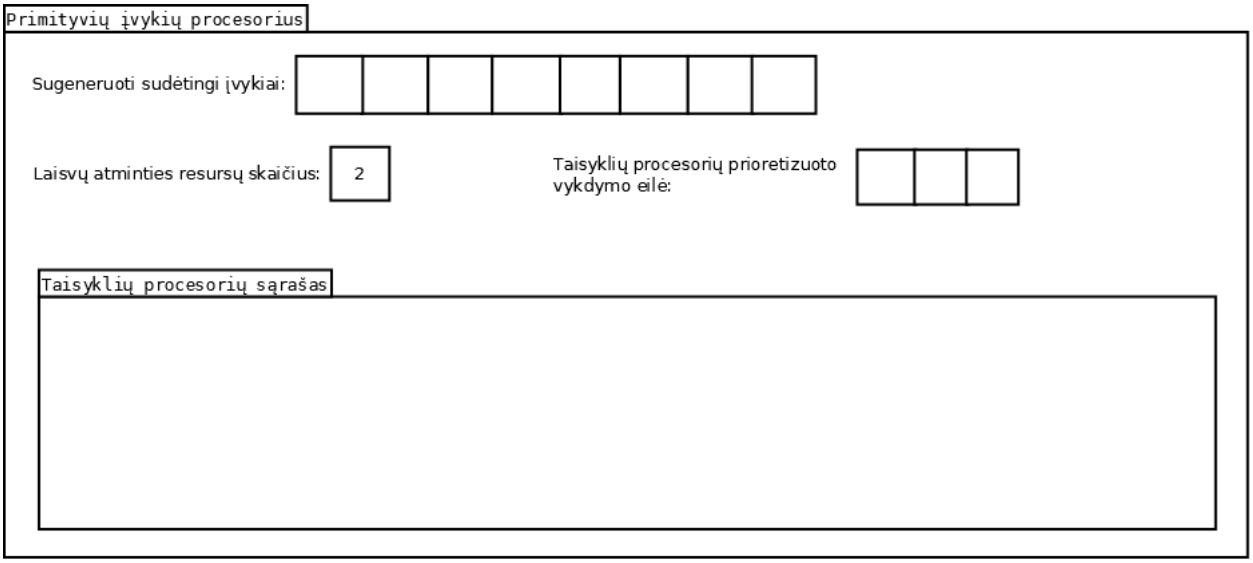
```
EVENT Z
WHERE
        EACH X WITHIN 5min FROM C
SELECT
        Z.z = AVG (X.x) EACH X WITHIN 5min FROM C
```



21 pav. Tęstinės užklausos *T2* apibrėžimas ir užpildyti primityvių įvykių stulpeliai

Pavyzdyje bus pateikiamas 20 ir 21 paveikslėliuose aprašytų taisyklių *T1* ir *T2* apdorojimas. Paveikslėlių dešinėje pusėje atvaizduoti taisyklių primityvių įvykių stulpeliai su juos užpildančiais primityviais įvykiais. Paprastumo dėlei bus laikoma, kad visi įvykiai yra persiųsti į grafinio procesoriaus atmintį ir kad visų primityvių įvykių, esančių taisyklių stulpeliuose, atributų reikšmės ir laiko susietumo apribojimai yra išpildyti. Įvykių apdorojimas

pradedamas gavus *c1* šakninį primityvų įvykį, kuris, naudojant indeksavimo lentelę, signalizuoja, kad yra dvi taisyklės, kurios gali sugeneruoti naujus sudėtingus įvykius. Tad, persiuntus *c1* primityvų įvykį į taisyklių *T1* ir *T2* atitinkamus stulpelius, pradedamas apdorojimas.



22 pav. Primityvių įvykių procesoriaus pradinė būsena

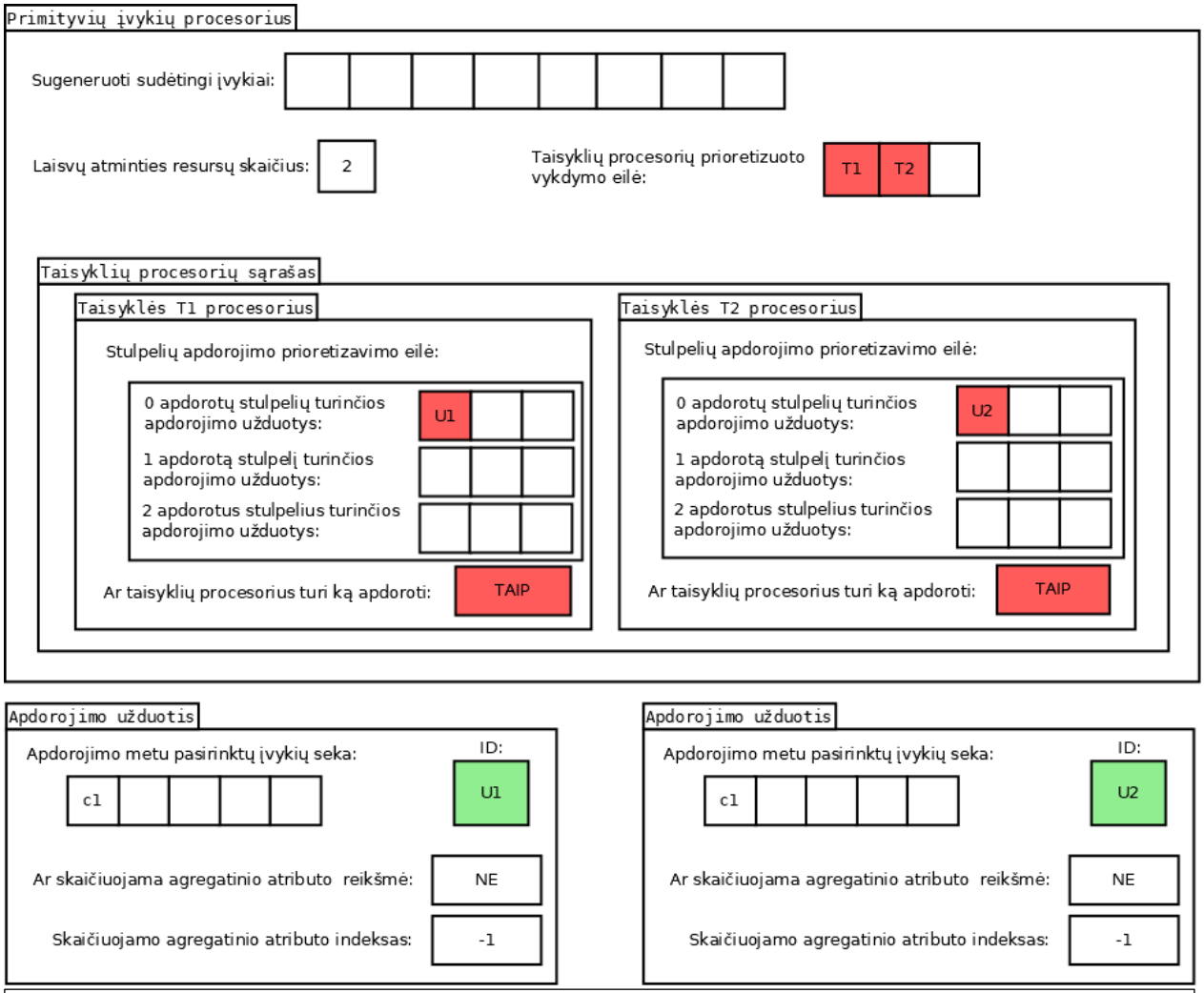
22 paveikslėlyje pateikiama pradinė primityvių įvykių procesoriaus būsena. Gavus norimas apdoroti taisykles *T1* ir *T2*, pastarasis procesorius yra inicializuojamas, t. y. kiekvienai taisyklei yra sukuriamas taisyklės procesorius ir sukurtų taisyklių procesorių indeksai yra patalpinami į taisyklių procesorių prioretizuoto vykdymo eilę. Inicializuoto procesoriaus būsena pateikiama 23 paveikslėlyje. Siekdami atskleisti kaip taisyklių procesoriai ir taisyklių procesorių stulpelių apdorojimo užduotys yra prioretizuojamos, kad būtų efektyviausiai išnaudojami atminties resursai, pavyzdyje laikysime, jog maksimalus laisvų grafinio procesoriaus atminties resursų, reikalingų atlikti vieno taisyklės stulpelio apdorojimui, skaičius yra 2, t. y. vienu metu apdorojami gali būti ne daugiau kaip 2 bet kurios taisyklės stulpeliai.

Inicializavus primityvių įvykių procesorių, yra pradedamas taisyklių stulpelių apdorojimas, kurio metu yra atrenkami stulpelyje esantys įvykiai, kurie tenkina taisyklėje aprašytus laiko susietumo ir atributų reikšmių apribojimus. Iš atrinktų apribojimus tenkinančių primityvių įvykių yra konstruojamos primityvių įvykių sekos, kurios leidžia sugeneruoti sudėtingus įvykius. Taisyklės stulpelių apdorojimas primityvių įvykių procesoriuje atliekamas tokiais žingsniais:

1. Gaunami tarpiniams apdorojimo duomenims saugoti reikalingi atminties resursai, t. y. grafinio procesoriaus atminties blokas, kurio primityvių įvykių procesorius, kaip nurodyta 18 schemoje, prašo iš atminties valdiklio. Atminties valdiklis savo ruožtu

atminties blokus skirsto iš iš anksto išskirtų atminties regionų, taip siekdamas sumažinti delsą, atsirandančią bandant paprašyti atminties per OpenCL sąsają.

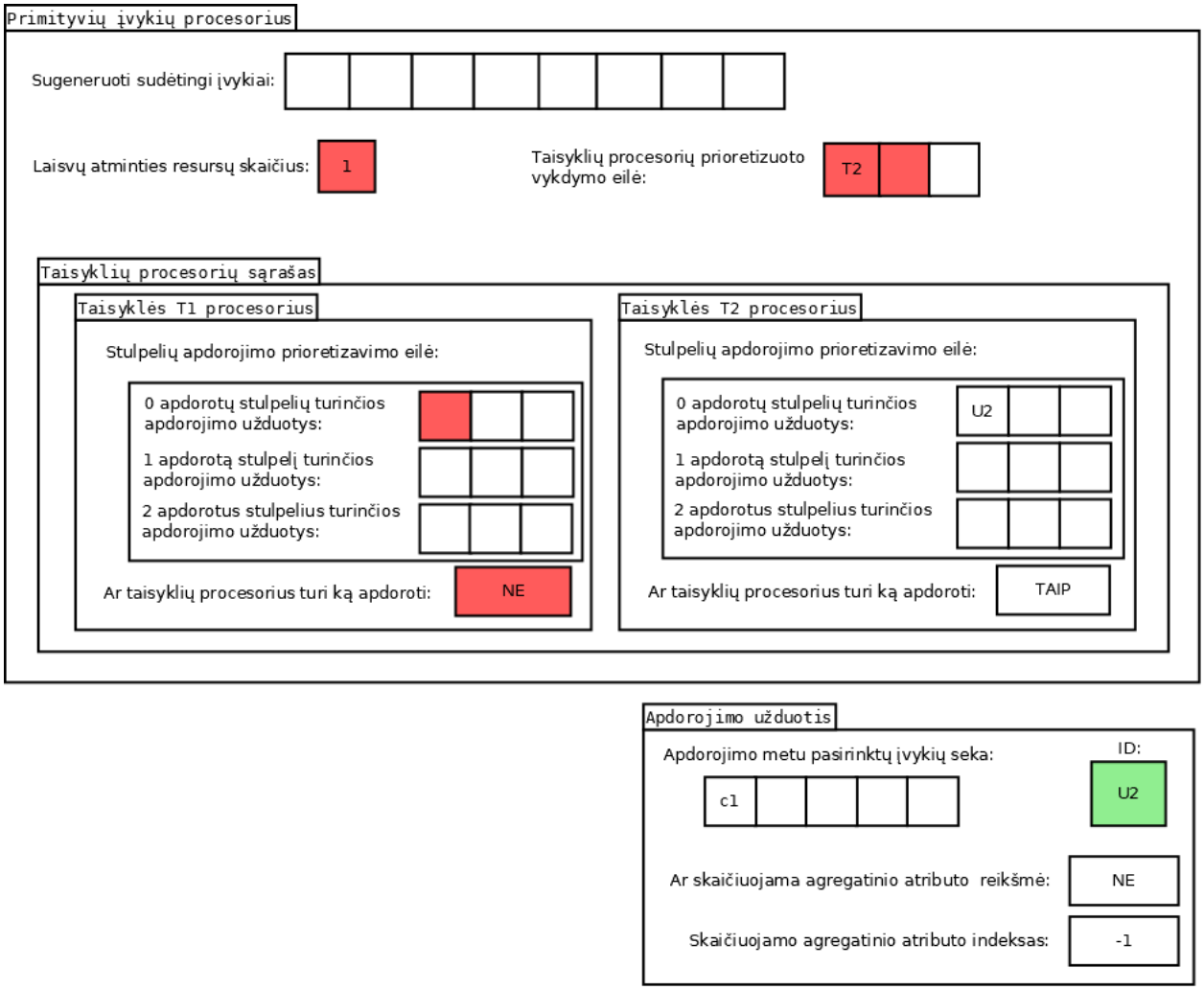
2. Iš taisyklių procesorių prioretizuoto vykdymo eilės, kurioje taisyklių procesoriai talpinami pagal pastovų, unikalų identifikacinį numerį, leidžiantį visuomet surūšiuoti taisyklių procesorius ta pačia tvarka, paimamas pirmasis eilėje esantis taisyklės procesorius.
3. Išrinktam taisyklės procesoriui perduodami atminties resursai, kad pastarasis pradėtų sekančio taisyklės stulpelio apdorojimą.



23 pav. Inicializuoto primityvių įvykių procesoriaus būsena

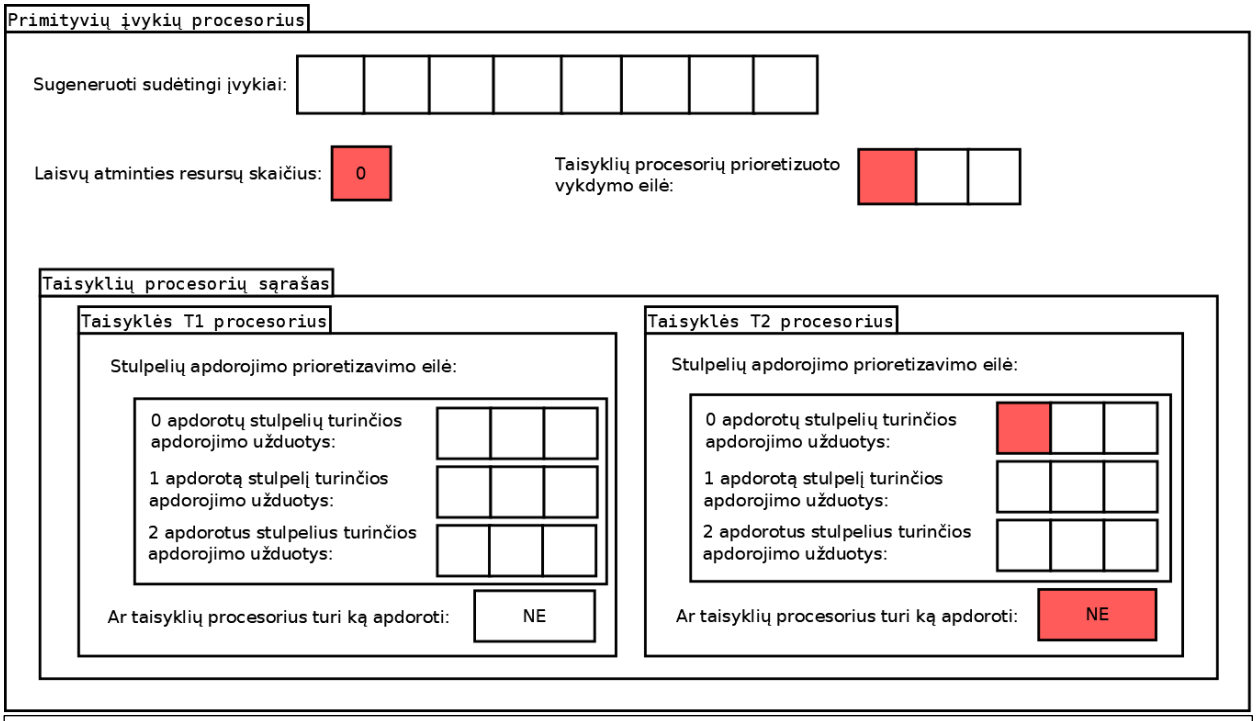
Kaip, vykdant pastaruosius žingsnius, kinta apdorojimo procesorių būsenos pavaizduota 23 – 25 paveikslėliuose, t. y. 23 paveikslėlyje matyti, jog pirmas taisyklių procesorių prioretizuoto vykdymo eilėje yra *T1* taisyklės procesorius, kuris savo stulpelių apdorojimo eilėje turi užduotį *U1*, kurios rezultatas turėtų būti atrinkti, tinkami primityvūs įvykiai iš taisyklės pirmojo stulpelio. Todėl pirmasis laisvas atminties resursas yra paskiriamas *T1* taisyklės procesoriui,

kuris paleidžia *U1* stulpelio apdorojimo užduoties vykdymą ir atnaujina savo būseną kaip pavaizduota 24 paveikslėlyje.



24 pav. Primityvių įvykių procesoriaus būsena, pradėjus vykdyti *U1* stulpelio apdorojimo užduotį

Kadangi primityvių įvykių procesorius turi dar vieną laisvą atminties resursą, o *T1* taisyklės procesorius šiuo metu neturi ką apdoroti, tai sekantis taisyklių vykdymo eilės grąžinamas procesorius yra *T2* ir laisvas atminties resursas suteikiamas jam, kad lygiagrečiai būtų vykdoma *U2* stulpelio apdorojimo užduotis. Atlikus šį žingsnį procesorių būsenas atvaizduoja 25 paveikslėlis, t. y. nėra laisvų atminties resursų ir taisyklių procesorių vykdymo eilė yra tuščia, todėl daugiau užduočių, kurias būtų galima vykdyti lygiagrečiai, paleisti negalima ir lieka laukti iki šiol pradėtų vykdyti primityvių įvykių stulpelių apdorojimo užduočių rezultatų.



25 pav. Primityvių įvykių procesoriaus būsena, pradėjus vykdyti *U2* stulpelio apdorojimo užduotį

Taisyklės primityvių įvykių stulpelio apdorojimas taisyklės procesoriuje atliekamas tokiais žingsniais:

1. Naudojant paskutiniame, apdorotame primityvių įvykių stulpelyje pasirinkto primityvaus įvykio laiko žymę, dvejetainės paieškos, vykdomos centrinio procesoriaus, pagalba yra atrenkami apdorojamo primityvių įvykių stulpelio įvykių pradžios *s* ir pabaigos *p* indeksai, į kuriuos patenkantys primityvūs įvykiai tenkina taisyklėje apibrėžtus laiko susietumo tarp stulpelius atitinkančių primityvių įvykių apribojimus.
2. Rezultatų buferyje yra loginio tipo kintamųjų masyvas *m*, kurio dydis sutampa su maksimaliu primityvių įvykių skaičiumi, kuris gali būti sutalpintas stulpelio buferyje. Naudojant pirmame žingsnyje gautais indeksais yra paleidžiama branduolio funkcija su tiek darbo vienetų, kiek yra primityvių įvykių apdorojamame stulpelyje tarp pastarųjų indeksų, t. y. *p* - *s*. Paleista branduolio funkcija tokiu būdu inicializuoja rezultatų masyvą, kuris nusako kurie įvykiai tarp gautų indeksų stulpelyje tenkina visus stulpelį atitinkančiam primityviam įvykiui aprašytus atributų reikšmių apribojimus.
3. Kiekvienam atributų reikšmių apribojimui aprašytam stulpelį atitinkančiam primityviam įvykiui taisyklėje yra paleidžiama branduolio funkcija su *p* - *s* globalių darbo vienetų, iš kurių kiekvienas pagal savo globalų identifikacinį numerį pasirenka stulpelyje esantį primityvų įvykį tarp indeksų *s* ir *p* ir patikrina ar to įvykio atributo

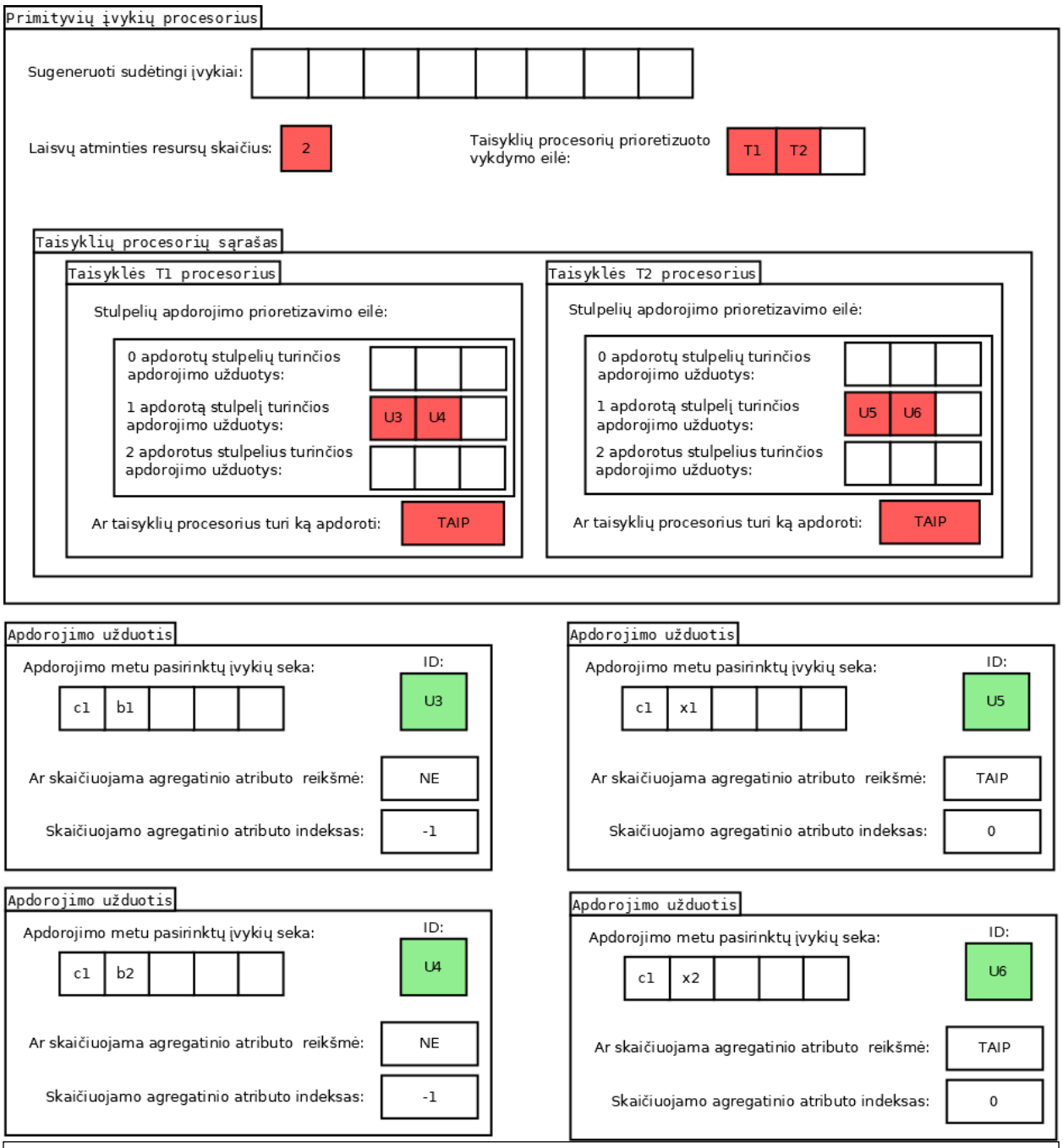
reikšmė tenkina tikrinamą apribojimą. Jei apribojimas netenkinamas masyve *m* tikrinamą įvykį atitinkančiame indekse yra nustatoma neigiama loginė reikšmė, nusakanti, jog įvykis yra netinkamas.

4. Atlikus visų atributų reikšmių apribojimų patikrinimą, iš grafinio procesoriaus atminties masyvas *m* yra persiunčiamas į centrinio procesoriaus operatyviąją atmintį ir nuosekliai peržiūrimas, išrenkant tinkamus įvykius, pagal stulpelio įvykių išrinkimo politiką sukonstruojant naujas taisyklės procesoriaus apdorojimo užduotis tolimesniam stulpeliui apdoroti ir patalpinant pastarąsias užduotis stulpelių apdorojimo prioretizavimo eilėje.

Vykdant taisyklės primityvių įvykių stulpelio apdorojimo užduotį *U1,* pastarieji žingsniai duotų tokius rezultatus:

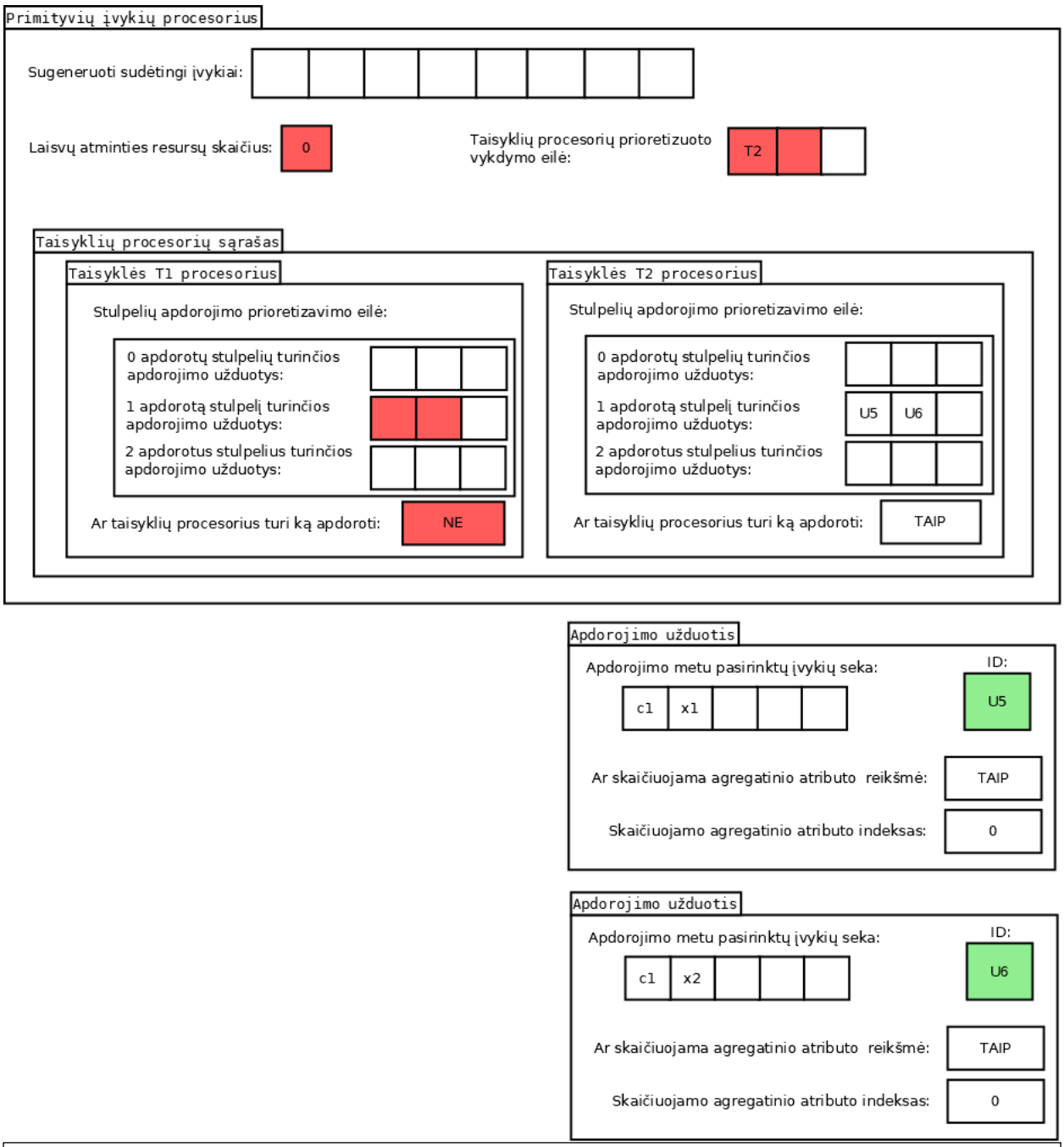
1. Prieš tai pasirinktas įvykis *c1,* apdorojamas stulpelis *B*. Kaip buvo minėta pavyzdyje laikoma, jog visi įvykiai tenkina laiko susietumo apribojimus, todėl atrinktas pradžios indeksas $s = 0$, pabaigos indeksas $p = 2$.
2. $p - s = 2 - 0 = 2$ gijos inicializuoja rezultatų masyvą *m*.
3. Paleidžiama tiek gijų porų, kiek yra atributų reikšmių apribojimų primityviam įvykiui *B*. Kadangi buvo minėta, jog pavyzdyje visos atributų reikšmės tenkins visus atributų reikšmių apribojimus, tai gauname, jog visi įvykiai tarp indeksų 0 ir 2 yra tinkami, t. y. *b1* ir *b2*. Atitinkamai nustatomos rezultatų masyvo *m* reikšmės.
4. Nuosekliai peržiūrėjus rezultatų masyvą *m*, gaunama, kad turėtų būti panaudojami abu primityvūs įvykiai esantys primityvaus įvykio *B* stulpelyje. Kadangi stulpelio išrinkimo politika yra „su kiekvienu“, tai su įvykiais *b1* ir *b2* yra generuojamos naujos *A* primityvaus įvykio stulpelio apdorojimo užduotys *U3* ir *U4* su atitinkamomis pasirinktų primityvių įvykių sekomis. Sugeneruotos užduotys vėliau yra patalpinamos į stulpelių apdorojimo prioretizavimo eilę tolesniam apdorojimui.

Analogiškai yra atliekama ir *U2* apdorojimo užduotis, gaunami tinkami primityvūs įvykiai *x1* ir *x2* iš *X* primityvaus įvykio stulpelio bei sugeneruojamos naujos užduotys *U5* ir *U6*. Apdorojus atliktų užduočių *U1* ir *U2* rezultatus atitinkamai atnaujinama ir primityvių įvykių procesoriaus būsena, t. y. į taisyklių procesorių prioretizuoto vykdymo eilę yra patalpinami *T1* ir *T2* taisyklių procesoriai, atlaisvinami atminties resursai bei atnaujinamos taisyklių procesorių būsenos taip, kaip pavaizduota 26 paveikslėlyje.



26 pav. Primityvių įvykių procesoriaus būsena, gavus *U1* ir *U2* užduočių vykdymo rezultatus

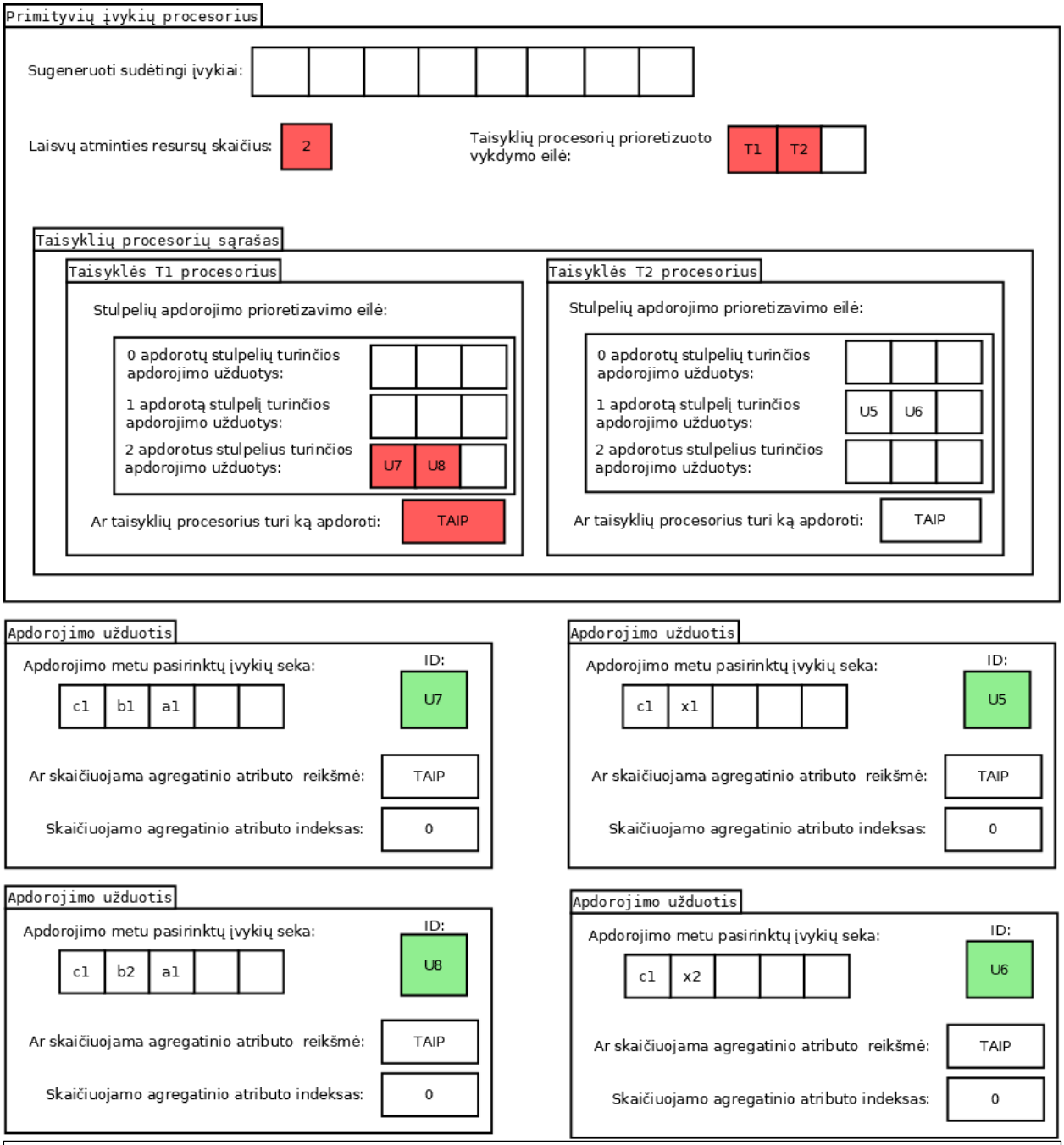
Nagrinėjant primityvių įvykių procesoriaus tolesnį veikimą, nesunku pastebėti, jog ir vėl laisvi atminties resursai pirmiausia bus paskirti *T1* taisyklės procesoriui, tačiau šį kartą jis turi ne vieną stulpelio apdorojimo užduotį, bet dvi. Todėl abu laisvi atminties resursai bus parskirti *T1* procesoriui, kuris juos panaudos *U3* ir *U4* stulpelių apdorojimo užduočių vykdymui. Procesorių būsena po šių dviejų žingsnių yra pavaizduota 27 paveikslėlyje. Tuo tarpu *T2* taisyklės procesorius, negalėdamas gauti apdorojimo užduotims reikalingų atminties resursų, yra paliekamas taisyklių procesorių eilėje ir nėra vykdomas tol, kol nėra reikalingų atminties resursų.



27 pav. Primityvių įvykių procesoriaus būsena, pradėjus vykdyti *U3* ir *U4* stulpelių apdorojimo užduotis

Gavus *U3* ir *U4* stulpelių apdorojimo rezultatus, yra suformuojamos galutinės pasirinktų primityvių įvykių sekos *c1*, *b1*, *a1* ir *c1*, *b2*, *a1*. Jei sudėtingas, sekų generuojamas įvykis *D* neturėtų agregatinių atributų, tai iš gautų sekų, naudojant centrinį procesorių, būtų sugeneruojamas naujas sudėtingas įvykis ir atlaisvinti atminties resursai galėtų būti panaudojami kitos taisyklės *T2* tolesniam apdorojimui. Tačiau šiuo atveju sudėtingas įvykis *D* turi vieną agregatinį atributą, kurio indeksas 0, o reikšmė – visų primityvaus įvykio *B* stulpelyje esančių primityvių įvykių *b* atributų reikšmių vidurkis. Todėl, tam kad būtų sugeneruojami nauji

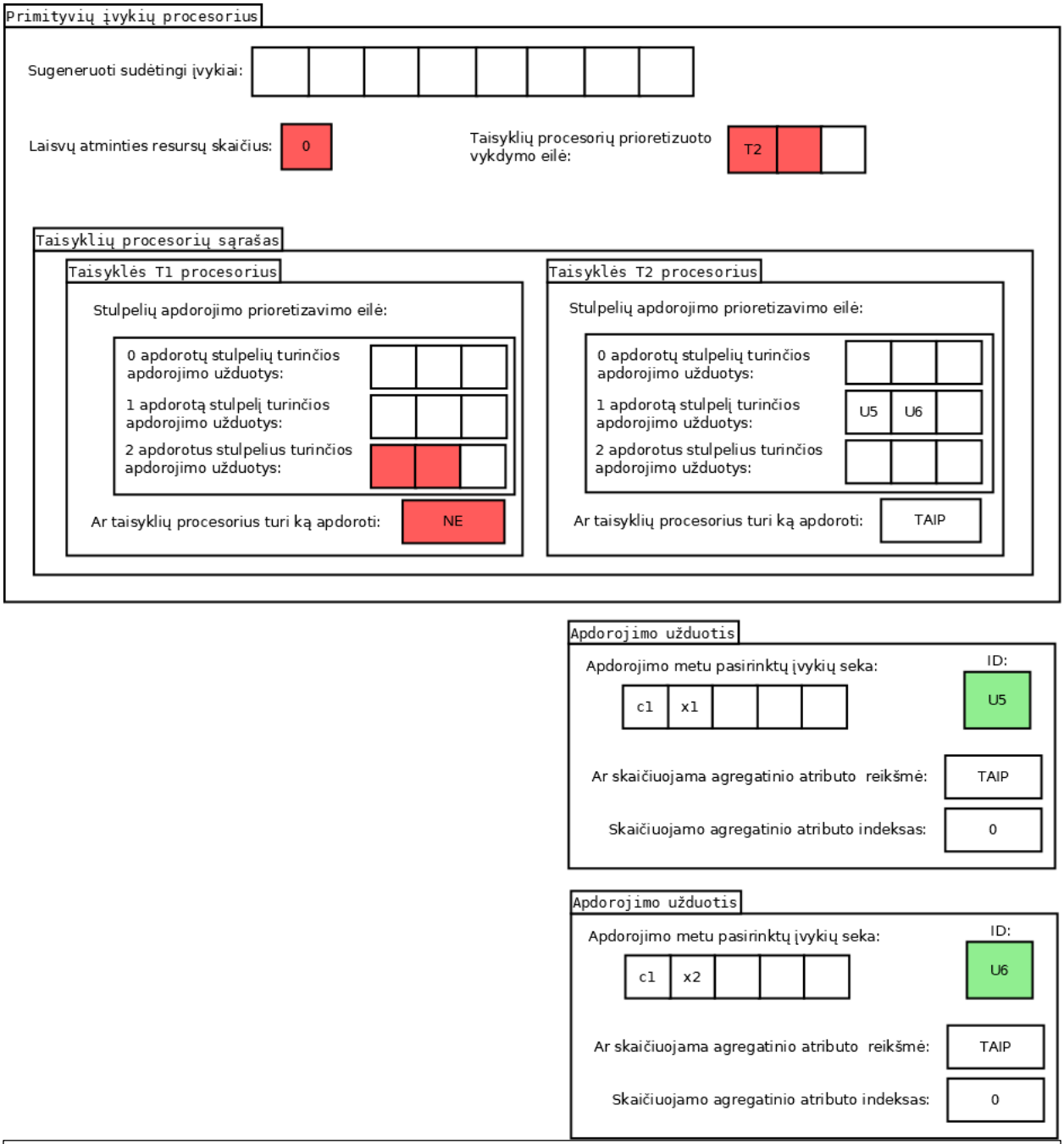
sudėtingi įvykiai, yra sukuriamos naujos stulpelių apdorojimo užduotys *U7* ir *U8*, kaip pavaizduota 28 paveikslėlyje.



28 pav. Primityvių įvykių procesoriaus būsena, gavus *U3* ir *U4* užduočių vykdymo rezultatus

Užduotys *U7* ir *U8* skiriasi nuo ankstesnių stulpelių apdorojimo užduočių tuo, kad jos nebekeičia apdorojimo metu pasirinktų primityvių įvykių sekos ir apdoroja ne taisyklės primityvių įvykių stulpelius, o taisyklės agregatinių atributų įvykių stulpelius ir gautas rezultatas, kurio tikimasi, yra nebe stulpelyje pasirinktas primityvus įvykis, o iš stulpelyje išrinktų primityvių įvykių atributų reikšmių gauta naujo sudėtingo įvykio atributo reikšmė. Šios užduotys nuo ankstesnių yra atskiriamos loginio tipo kintamojo reikšme.

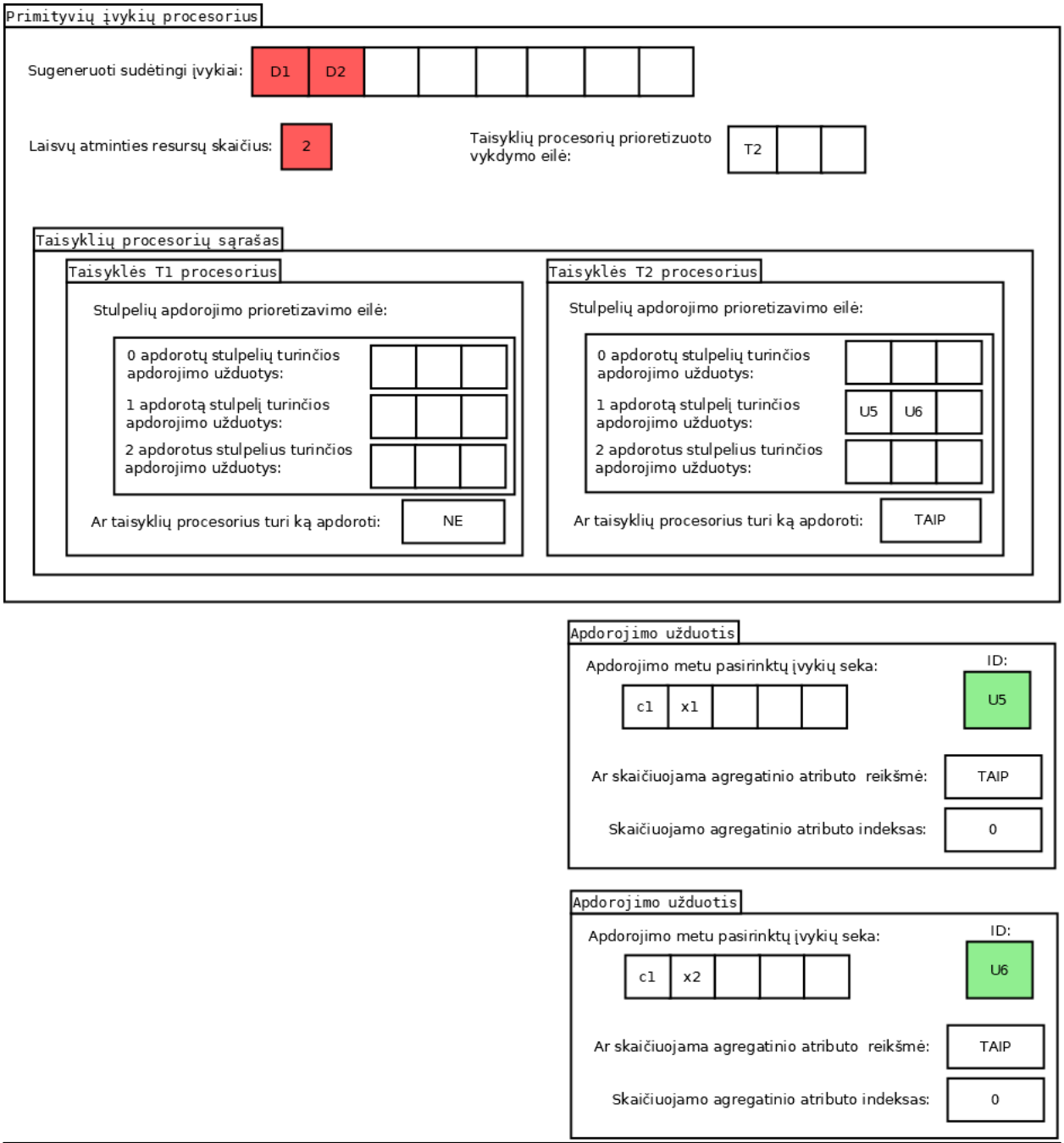
Tęsiant įvykių apdorojimą, kaip demonstruojama 28 paveikslėlyje, pridėtos naujos *U7* ir *U8* stulpelių apdorojimo užduotys taip pat pridėjo taisyklės *T1* procesorių su aukščiausiu prioritetu į taisyklių procesorių prioretizuoto vykdymo eilę. Tad ir vėl abu laisvi atminties resursai yra paskiriami *T1* taisyklės procesoriui, kas aiškiai atvaizduojama 29 paveikslėlyje.



29 pav. Primityvių įvykių procesoriaus būsena, pradėjus vykdyti *U7* ir *U8* stulpelių apdorojimo užduotis

Taisyklės agregatinių atributų įvykių stulpelių apdorojimas nuo įprastų stulpelių apdorojimo skiriasi tik tiek, kad stulpelyje išrinkus tinkamus primityvius įvykius rezultatų masyvas *m* nėra persiunčiamas iš grafinio procesoriaus atminties į centrinio procesoriaus atmintį, kurioje centrinis procesorius atrenka tinkamus įvykius ir juos panaudoja. Vietoje to yra

paleidžiama dar viena branduolio funkcija, kurią vykdo *s* - *p* darbo vienetų, kurie visi priklauso tai pačiai OpenCL darbo grupei ir kurie atrenka tinkamus įvykius pagal reikšmes rezultatų masyve *m* ir naudodami OpenCL darbo grupių mažinimo funkcijas atlieka agregatinių atributų reikšmių skaičiavimus. Tokiu būdu atlikus agregatinio atributo įvykių stulpelio apdorojimą, rezultate yra gaunama viena sudėtingo įvykio agregatinio atributo reikšmė.

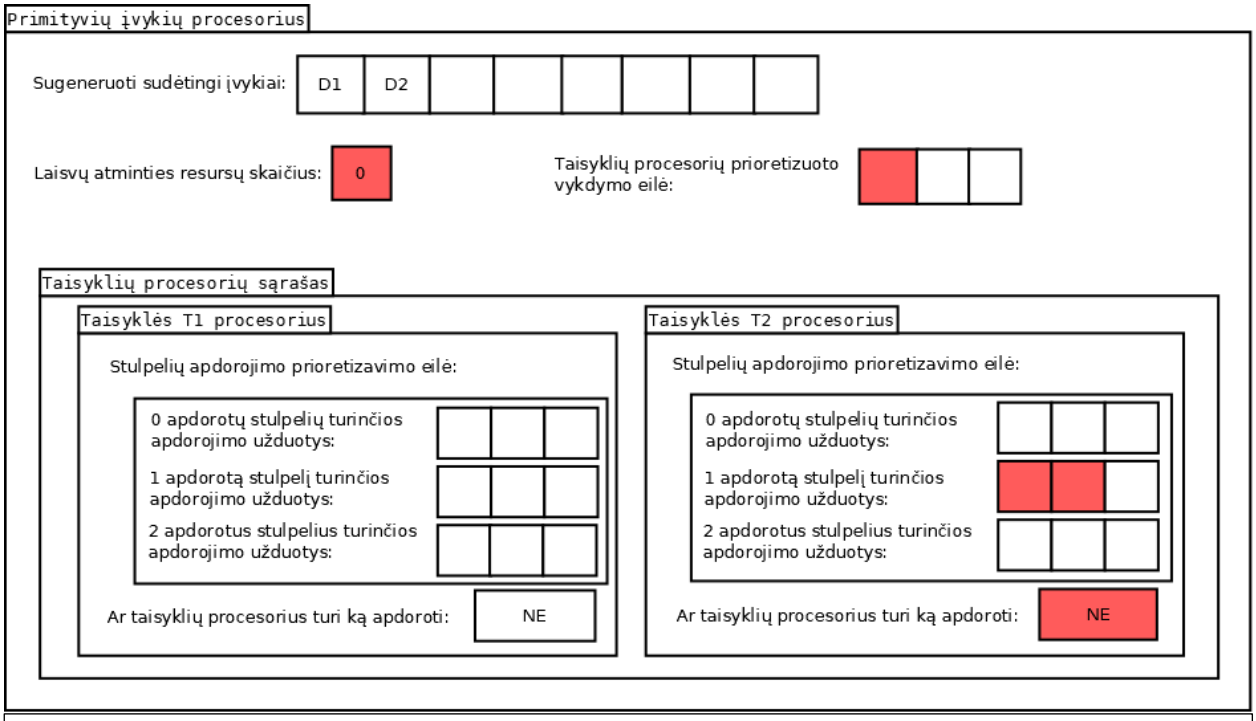


30 pav. Primityvių įvykių procesoriaus būsena, gavus *U7* ir *U8* užduočių vykdymo rezultatus

30 paveikslėlyje vaizduojama, jog gavus *U7* ir *U8* užduočių rezultatus yra sugeneruojami du nauji sudėtingi įvykiai *D1* ir *D2*, pastarieji yra išsaugomi sugeneruotų įvykių eilėje ir yra atlaisvinami atminties resursai. Nagrinėjamame pavyzdyje sudėtingo atributo *D* šablonas turi tik

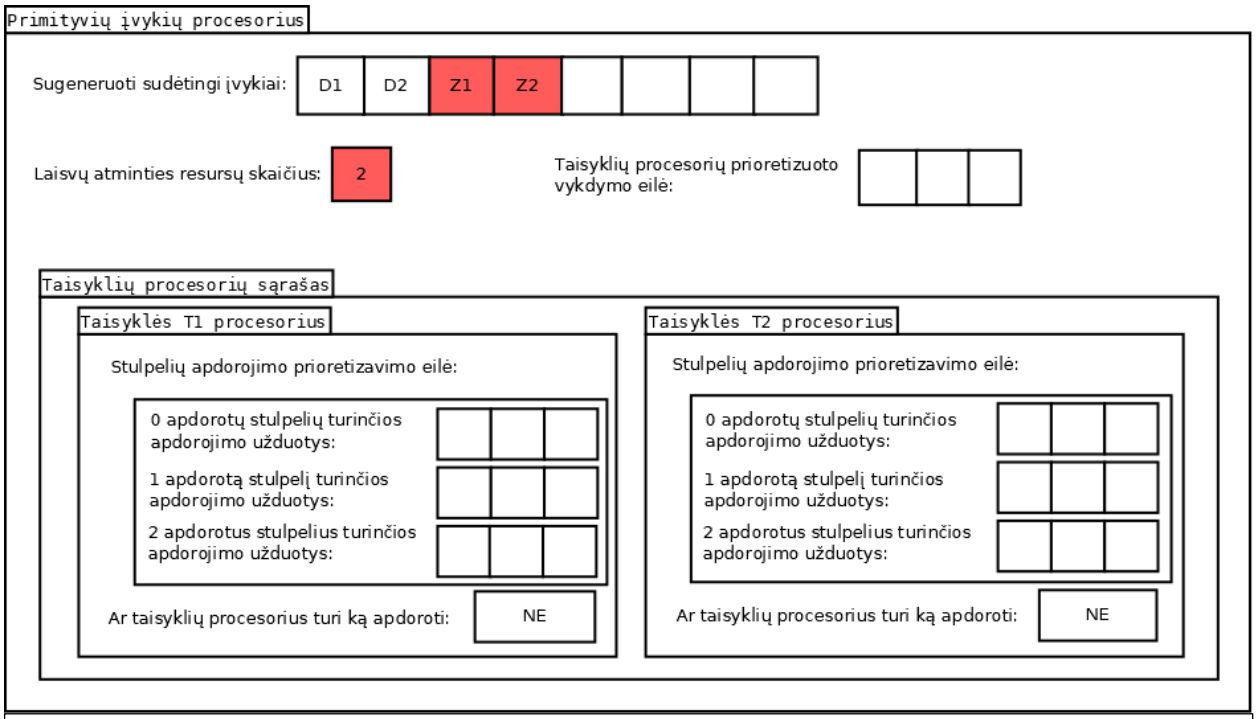
vieną agregatinį atributą, bet jei sudėtingo įvykio šablone jų būtų daugiau, tuomet, gavus nulinių agregatinių atributų reikšmes, jos būtų išsaugomos laikinų reikšmių masyve ir būtų sugeneruojamos naujos agregatinių atributų stulpelių apdorojimo užduotys, nurodant atitinkama skaičiuojamo agregatinio atributo indeksą, iki tol, kol visų agregatinių atributų reikšmės būtų suskaičiuotos ir būtų galima generuoti sudėtingus įvykius.

Toliau nagrinėjant apdorojimo procesą, 30 paveikslėlyje aiškiai matyti, jog taisyklės procesorius *T1*, baigęs generuoti sudėtingus įvykius *D1* ir *D2*, daugiau nebeturi stulpelių apdorojimo užduočių, todėl jis nėra įdedamas į taisyklių procesorių prioretizuoto vykdymo eilę, todėl aukščiausią vykdymo prioritetą einamuoju momentu įgyja taisyklės *T2* procesorius, kuriam ir yra paskiriami abu atsilaisvinę atminties resursai, kuriuos pastarasis procesorius gali panaudoti likusių užduočių *U5* ir *U6* vykdymui. Procesorių būsenos, atlikus šiuos veiksmus, atvaizduotos 31 paveikslėlyje.



31 pav. Primityvių įvykių procesoriaus būsena, pradėjus vykdyti *U5* ir *U6* stulpelių apdorojimo užduotis

Baigus vykdyti *U5* ir *U6* užduotis ir gavus agregatinių atributų reikšmes, taisyklės *T2* procesorius pagal *Z* sudėtingo įvykio šabloną sugeneruoja paskutinius sudėtingus įvykius *Z1* ir *Z2*, patalpina juos į sugeneruotų įvykių eilę ir atlaisvina nebereikalingus atminties resursus. 32 paveikslėlyje yra vaizduojama galutinė primityvių įvykių apdorojimo procesoriaus būsena, pasiekiama baigus apdorojimą, t. y. taisyklių procesorių prioretizuoto vykdymo eilė yra tuščia ir nėra einamuoju momentu vykdomų stulpelių apdorojimo užduočių.



32 pav. Primityvių įvykių procesoriaus būsena, gavus *U5* ir *U6* užduočių vykdymo rezultatus

Galiausiai primityvių įvykių procesorius pereina į 22 paveikslėlyje pavaizduotą pradinę būsena ir grąžina sugeneruotus sudėtingus įvykius, kad pastarieji būtų perduoti jais suinteresuotoms sistemoms.

2.2. Eksperimentinio tyrimo rezultatai

Tyrimo rezultatams gauti buvo nuspręsta pasinaudoti TRex sistemos vidinėmis struktūromis, kurios gautos generuojant ir interpretuojant tekstinio formato užklausas aprašytas Tesla kalba ir suinterpretuotas, naudojant Java Tesla taisyklių transliatorių. Tokios tekstinės užklausos pavyzdys, kuriame reikia trijų primityvių įvykių su trimis atributais ir kuriame yra vienas parametrinis taisyklės atributas bei vienas agregatinis rezultato atributas, pateiktas 33 paveikslėlyje. Gautos TRex sistemos vidinės struktūros vėliau konvertuotos į grafinio procesoriaus vidines struktūras.

```
Assign 1 => P_Event_0_0, 2 => P_Event_0_1, 3 => P_Event_0_2, 4 => C_Event_0
Define
          C_Event_0
                    (
                    c_attribute_0_0 : float,
                    c_attribute_0_1 : float,
                    c_attribute_0_2 : float
                    )
From
          P_Event_0_0
                    (
                    p_attribute_0_0_0 => $parameter_0_0,
                    p_attribute_0_0_1 < 60.0,
                    p_attribute_0_0_2 < 60.0
                    )
and each
          P_Event_0_1
                    (
                    [float]p_attribute_0_1_0 = $parameter_0_0,
                    p_attribute_0_1_1 < 60.0,
                    p_attribute_0_1_2 < 60.0
                    ) within 50 from P_Event_0_0
and last
          P_Event_0_2
                    (
                    [float]p_attribute_0_2_0 = $parameter_0_0,
                    p_attribute_0_2_1 < 60.0,
                    p_attribute_0_2_2 < 60.0
                    ) within 50 from P_Event_0_1
Where
          c_attribute_0_0 := AVG(
                    P_Event_0_1.p_attribute_0_1_0
                              (
                              [float]p_attribute_0_1_0 = $parameter_0_0,
                              p_attribute_0_1_1 < 60.0,
                              p_attribute_0_1_2 < 60.0
                              )
                    ) within 50 from P_Event_0_0,
          c_attribute_0_1 := 10.0 + 10.0 * 4.0,
          c_attribute_0_2 := 10.0 + 10.0 * 4.0;
```

33 pav. Sudėtingo įvykio taisyklės pavyzdys

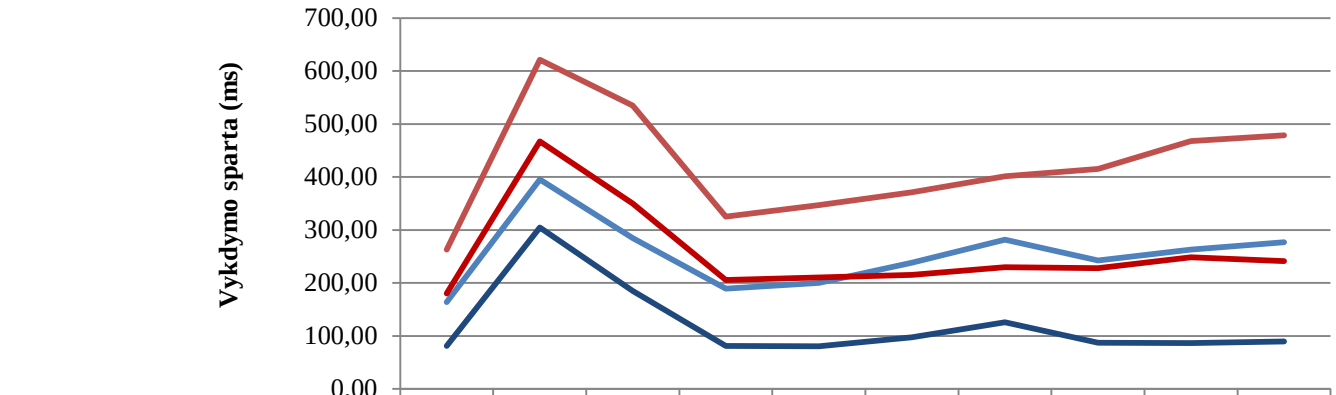
Siekiant išsiaiškinti kaip kiekvienas kintamasis įtakoja apdorojimo spartą, buvo pasirinkta pradinė taisyklė, atitinkanti realaus gyvenimo poreikius finansų rinkose bei kitose srityse, ir buvo

matuojama apdorojimo sparta, keičiant tik vieną iš parametrų tam tikrame reikšmių intervale. Pradinę taisyklę sudarė:

- 10 primityvių įvykių, kurių kiekvienas turi po 10 atributų;
- 2 parametriniai atributai;
- 2 agregatiniai atributai rezultate (skaičiuojantys atributo reikšmių vidurkį);
- 1 primityvus įvykis su daugelio išrinkimų politika;
- 100 milisekundžių trukmės langai tarp primityvių įvykių.

Kiekvienam bandymui buvo sugeneruojama po 10000 primityvių įvykių, siekiant tolygiai užpildyti primityvių įvykių atminties blokus. Kiekvienam primityvaus įvykio atributui buvo priskiriamos atsitiktinės reikšmės intervale nuo 0 iki 100, kur pirma pusė reikšmių tenkina taisyklės reikalavimus, o kita pusė – ne. Kiekvieno matavimo atveju, buvo atliekama 10 iteracijų ir paskaičiuojami vykdymo spartos, įskaitant ir įvykių siuntimą, bei tik apdorojimo spartos vidurkiai.

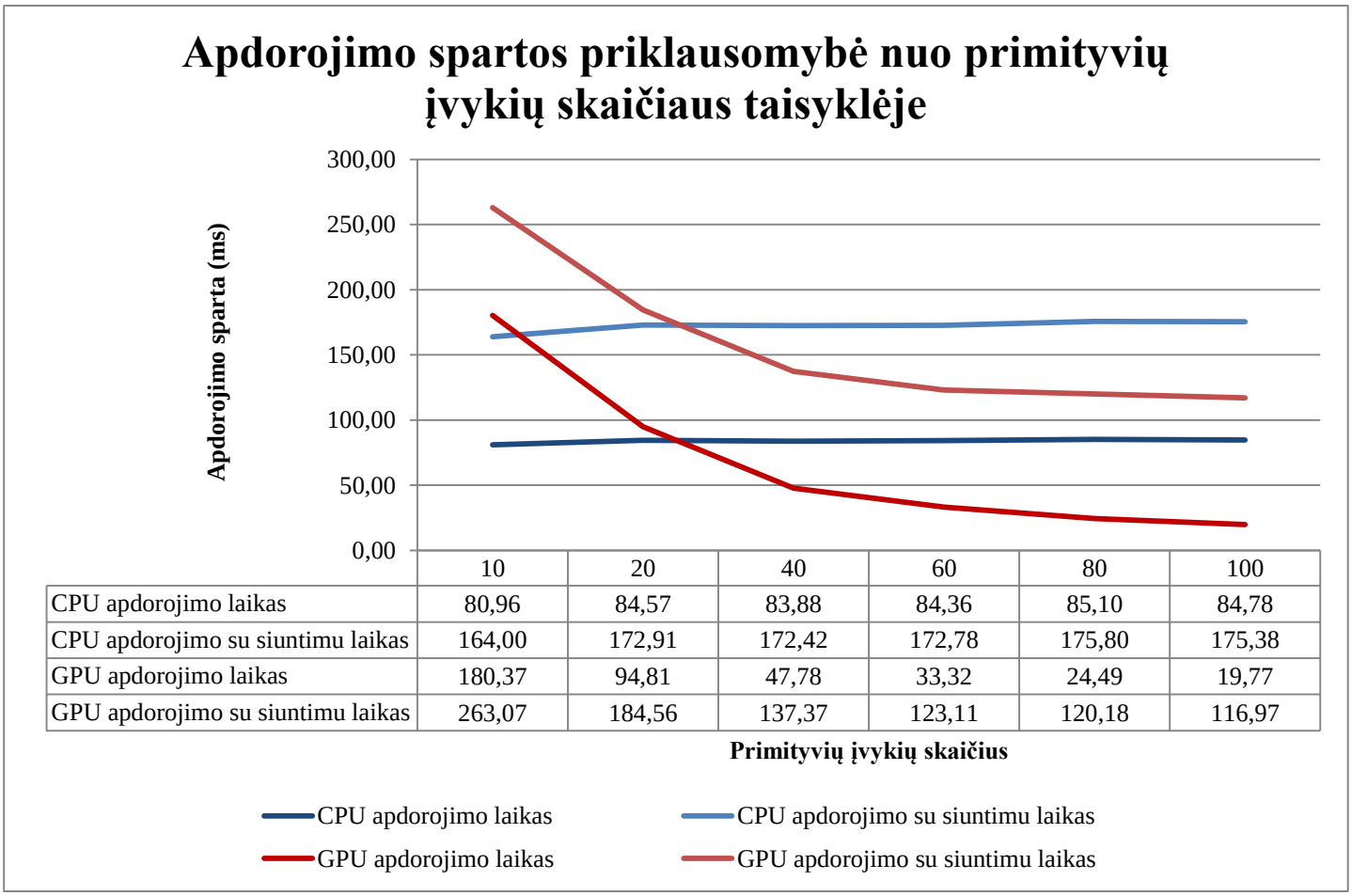
2.2.1. Taisyklių skaičius



	1	11	21	31	41	51	61	71	81	91
CPU apdorojimo laikas	80,96	304,40	185,32	81,16	80,36	97,21	125,87	87,13	86,74	89,51
CPU apdorojimo su siuntimu laikas	164,00	395,28	284,51	189,04	200,05	238,06	281,66	242,22	263,00	276,60
GPU apdorojimo laikas	180,37	467,21	349,74	205,81	210,60	215,40	229,43	228,05	248,34	241,01
GPU apdorojimo su siuntimu laikas	263,07	621,52	534,84	325,48	346,67	370,95	401,27	415,01	467,49	478,52

Nagrinėjant kaip kinta apdorojimo sparta didėjant taisyklių skaičiui, bet primityvių įvykių skaičiui išliekant tokiam pačiam, galima aiškiai matyti kaip neigiamai tai įtakoja tiek apdorojimo, tiek persiuntimo spartą grafinio procesoriaus atveju, tačiau turi nežymią įtaką centrinio procesoriaus atveju. Taip yra todėl, kad grafinis procesorius turi atlikti papildomų žingsnių prieš galėdamas pradėti įvykių apdorojimą, t. y. reikia persiųsti primityvius įvykius į grafinio procesoriaus atmintį ir perkopijuoti atitinkamas atributų reikšmes į atitinkamas vietas kiekvienos taisyklės tam tikro stulpelio atminties bloke. Didėjant taisyklių skaičiui, didėja atminties blokų, į kuriuos reikia perkopijuoti reikšmes skaičius, o tam reikia daugiau atminties operacijų, kurios užima daugiau laiko. Negana to, didėjant taisyklių skaičiui, primityvūs įvykiai yra tolygiai paskirstomi visom taisyklėms, dėl to įvykių stulpelių atminties blokuose susikaupia mažai, o kadangi bangų gijos yra susietos per nagrinėjamą apribojimą skirtingų įvykių tai pačiai atributo reikšmei, tai, esant mažai įvykių, bangų gijos yra nepilnai užpildomos ir jų reikia paleisti daugiau, kad apdoroti visas taisykles. Tai neigiamai veikia ir apdorojimo spartą. Tad grafiniai procesoriai tinka tuo atveju, jei yra proporcingai daug ir taisyklių ir primityvių įvykių, užpildančių tų taisyklių stulpelių atminties blokus.

2.2.2. Primityvių įvykių skaičius

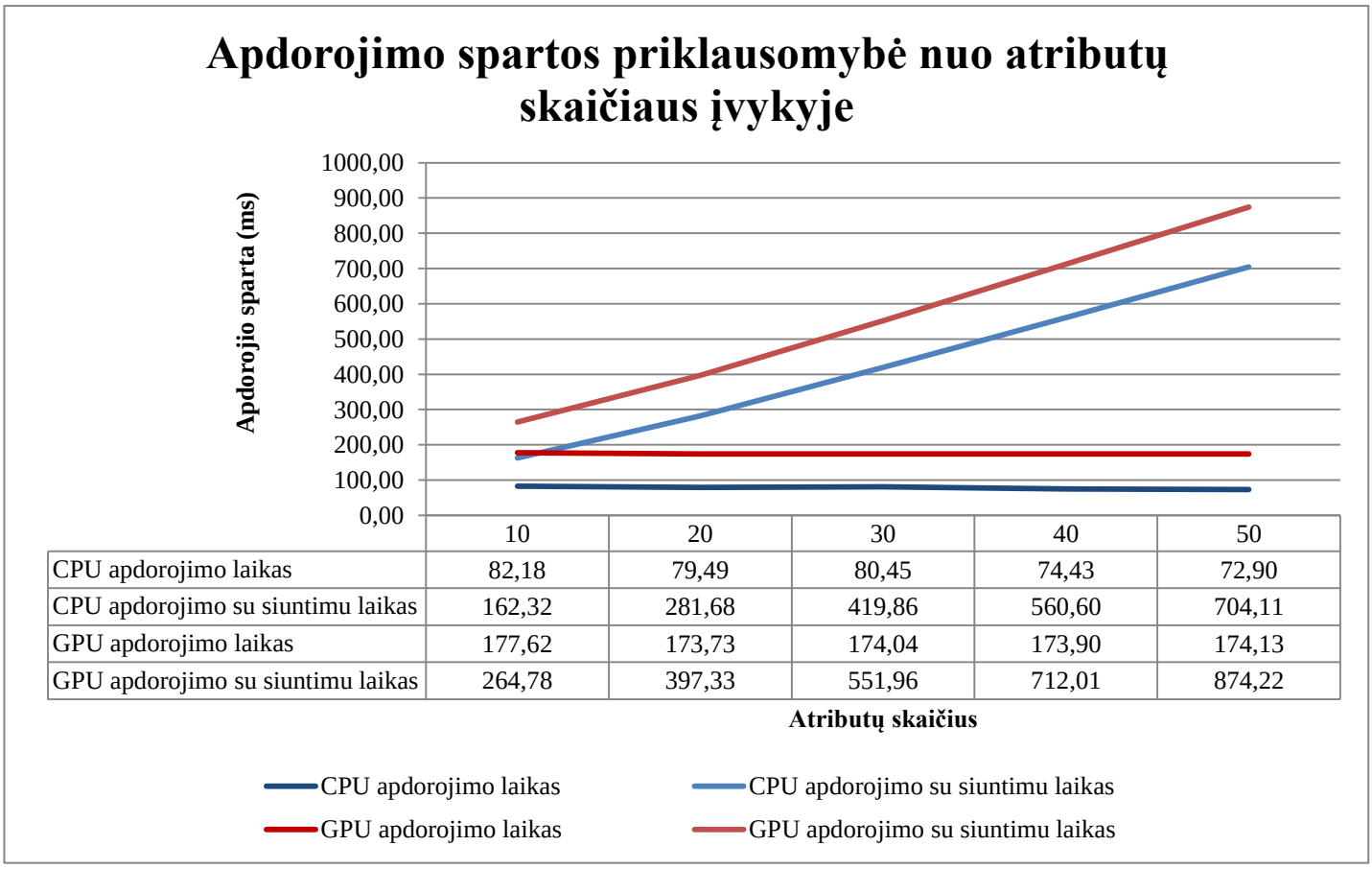


	10	20	40	60	80	100
CPU apdorojimo laikas	80,96	84,57	83,88	84,36	85,10	84,78
CPU apdorojimo su siuntimu laikas	164,00	172,91	172,42	172,78	175,80	175,38
GPU apdorojimo laikas	180,37	94,81	47,78	33,32	24,49	19,77
GPU apdorojimo su siuntimu laikas	263,07	184,56	137,37	123,11	120,18	116,97

Šiuo atveju yra demonstruojami arti idealaus atvejo rezultatai. 10000 primityvių įvykių 100 – tame stulpelių užpildo atminties blokuose po maždaug 100 primityvių įvykių, tai nagrinėto grafinio procesoriaus atveju yra beveik dvi pilnos bangų gijos. Negana to, daugelio išrinkimo stulpeliai generuojami iškart po šakninio įvykio stulpelio, o tai reiškia, kad jau po pirmo stulpelio apdorojimo galima paleisti tiek atskirų bangų gijų, kiek antrame stulpelyje yra apribojimus tenkinančių primityvių įvykių. Visos paleistos bangų gijos gali būti vykdomos skirtingų skaičiavimo vienetų ir, jei tarp jų vykdymo paleidimo yra pakankama delsa, kaip šiuo atveju, tai dauguma skaičiavimo vienetų skaito skirtingus atminties blokus. Tad šiuo atveju stebime puikiai grafinio procesoriaus paralelizmą išnaudojantį atvejį su gan įspūdingu įvykių apdorojimo spartos skirtumu lyginant su centriniu procesoriumi. Dar vienas svarbus reiškinys, kurį pastebėjau vykdant bandymą, yra OpenCL karkaso apribojimas, neleidžiantis pilnai išnaudoti paralelizmo, – asinchroninių komandų eilių palaikymo nebuvimas. Tai galima dalinai išspręsti prikuriant daug atskirų sinchroninių komandų eilių, tačiau su kiekviena sukurta sinchronine eile OpenCL karkasas paleidžia ir susieja atskirą centrinio procesoriaus giją, todėl šitas sprendimas turi daug neigiamų padarinių, t. y., turint mažai fizinių centrinio procesoriaus branduolių, prarandama nemažai spartos sinchronizavimui tarp gijų ir centrinio procesoriaus vykdymų kontekstų

keitimui, prikuriant daug komandų eilių, sudėtingėja sistemos realizacija bei yra nelengva tolygiai paskirstyti komandų krūvį skirtingoms komandų eilėms. Pavyzdžiui, nagrinėto grafinio procesoriaus atveju yra 8 asinchroniniai skaičiavimo varikliai (angl. Asynchronous Compute Engine, ACE), iš kurių kiekvienas gali valdyti ir skirstyti darbus 8 nepriklausomoms komandų eilėms, t. y. būtų galima susikurti 64 komandų eiles, bet su joms OpenCL karkasas sukuria 64 centrinio procesoriaus gijas. Tad šiuo atveju tai yra labai didelis OpenCL karkaso trūkumas.

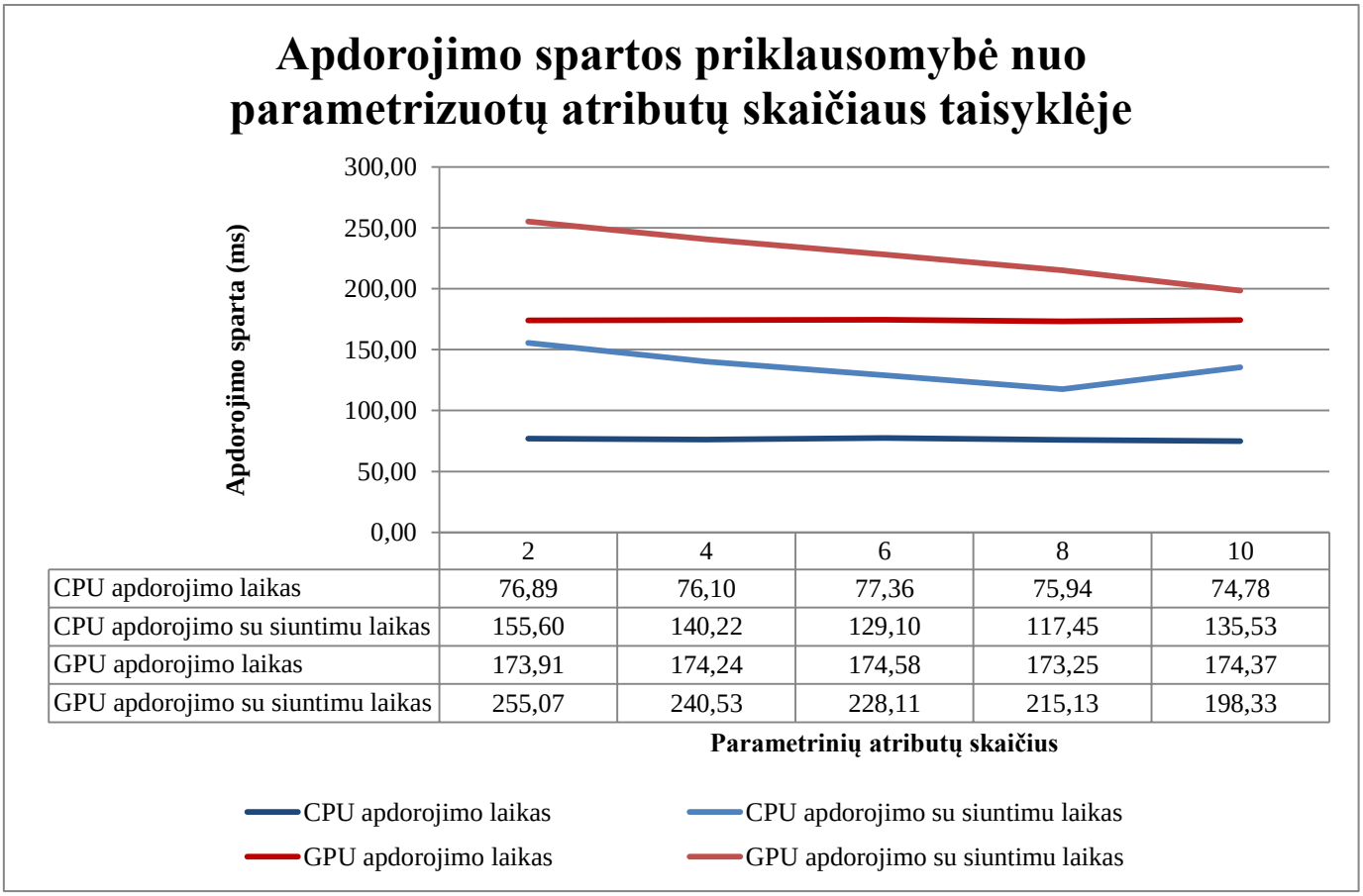
2.2.3. Atributų skaičius



	10	20	30	40	50
CPU apdorojimo laikas	82,18	79,49	80,45	74,43	72,90
CPU apdorojimo su siuntimu laikas	162,32	281,68	419,86	560,60	704,11
GPU apdorojimo laikas	177,62	173,73	174,04	173,90	174,13
GPU apdorojimo su siuntimu laikas	264,78	397,33	551,96	712,01	874,22

Spartos priklausomybė nuo atributų skaičiaus labai nustebino, matyti, kad didėjant atributų skaičiui vykdymo sparta faktiškai nekinta, o persiuntimo laikas, kaip ir galima buvo tikėtis, smarkiai auga, nes su kiekvienu primityviu įvykiu tenka persiųsti daugiau duomenų. Kadangi rezultatas netikėtas, sunku pasakyti kodėl sparta visai neauga, tačiau viena iš priežasčių gali būti ta, jog apdorojant stulpelį yra paleidžiama dviejų dimensijų branduolio funkcija, kurioje konstruojama daug vienos gijų bangos dydžio darbo grupių, kurios tikrina to pačio atributo reikšmės sąlygą taisyklėje, t. y. nors skirtingos darbo grupės skaito skirtingų primityvių įvykių atributų reikšmes, kurios yra greta, bet visos grupės, kurias gali vykdyti skirtingi skaičiavimo vienetai, skaito tą pačią viena atminties dalį, kurioje patalpinta informacija apie apribojimus nagrinėjamo atributo reikšmei, o tokiu atveju atminties skaitymas yra serializuojamas ir netenkama spartos. Tad šitam atvejui reikėtų atlikti papildomą tyrimą.

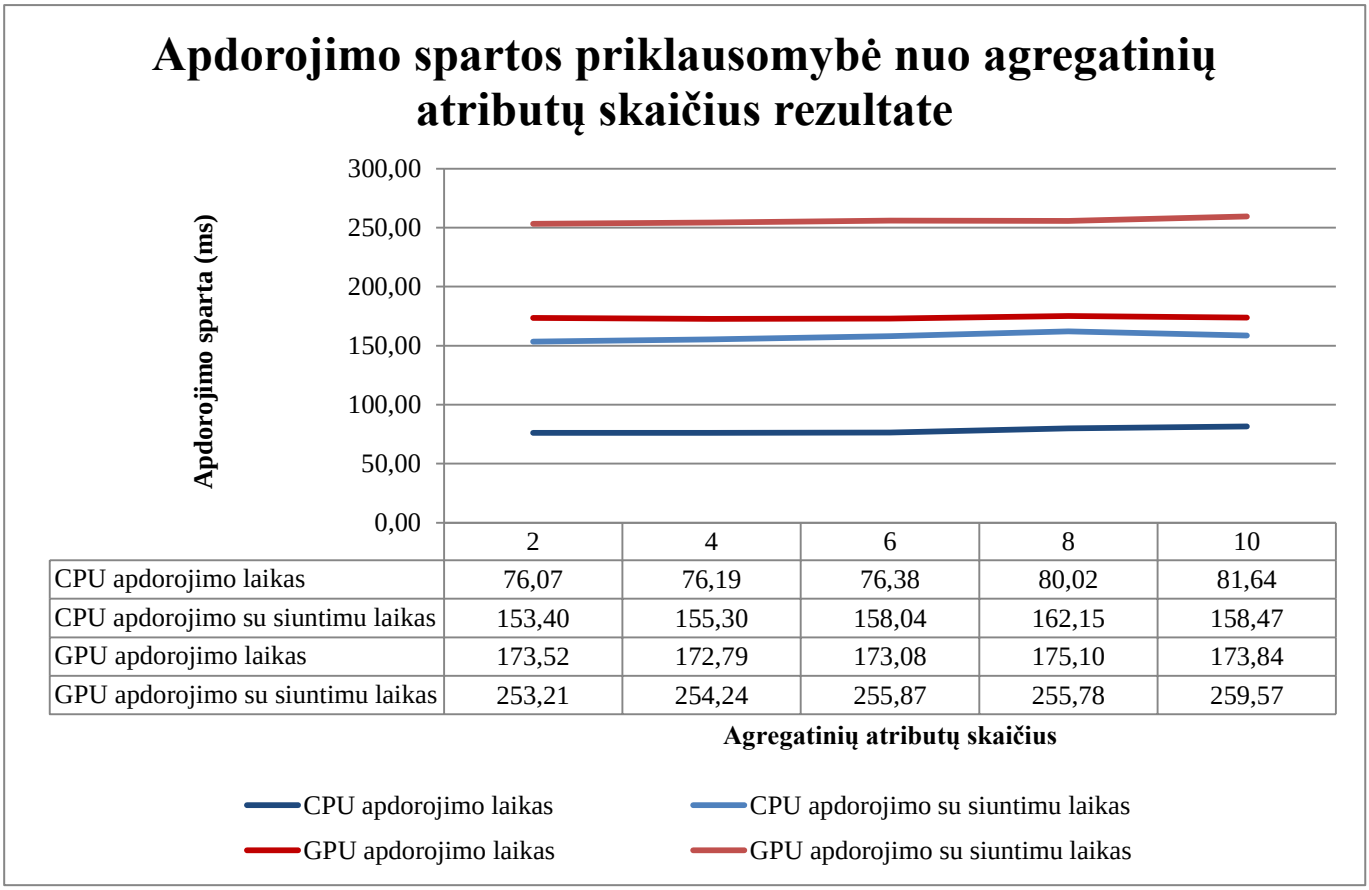
2.2.4. Parametrizuotų atributų skaičius



	2	4	6	8	10
CPU apdorojimo laikas	76,89	76,10	77,36	75,94	74,78
CPU apdorojimo su siuntimu laikas	155,60	140,22	129,10	117,45	135,53
GPU apdorojimo laikas	173,91	174,24	174,58	173,25	174,37
GPU apdorojimo su siuntimu laikas	255,07	240,53	228,11	215,13	198,33

Apdorojimo spartos priklausomybė nuo parametrizuotų atributų skaičiaus iš tiesų nestebina, nes, žinant vidinį apdorojimo procesoriaus veikimo principą, nesunku nuspėti tokį rezultatą, t. y. vykdymo metu vienintelis skirtumas tarp paprasto atributo reikšmės apribojimo ir parametrizuoto yra tas, jog parametrizuotam reikia gauti kažkurio, apdorojant ankstesnį stulpelį pasirinkto įvykio atributo reikšmę, o tai gaunama viena pigia papildoma atminties operacija iš individualaus rezultatų atminties bloko, todėl parametrizuotų atributų skaičius neturėtų turėti didelės reikšmės apdorojimo spartai.

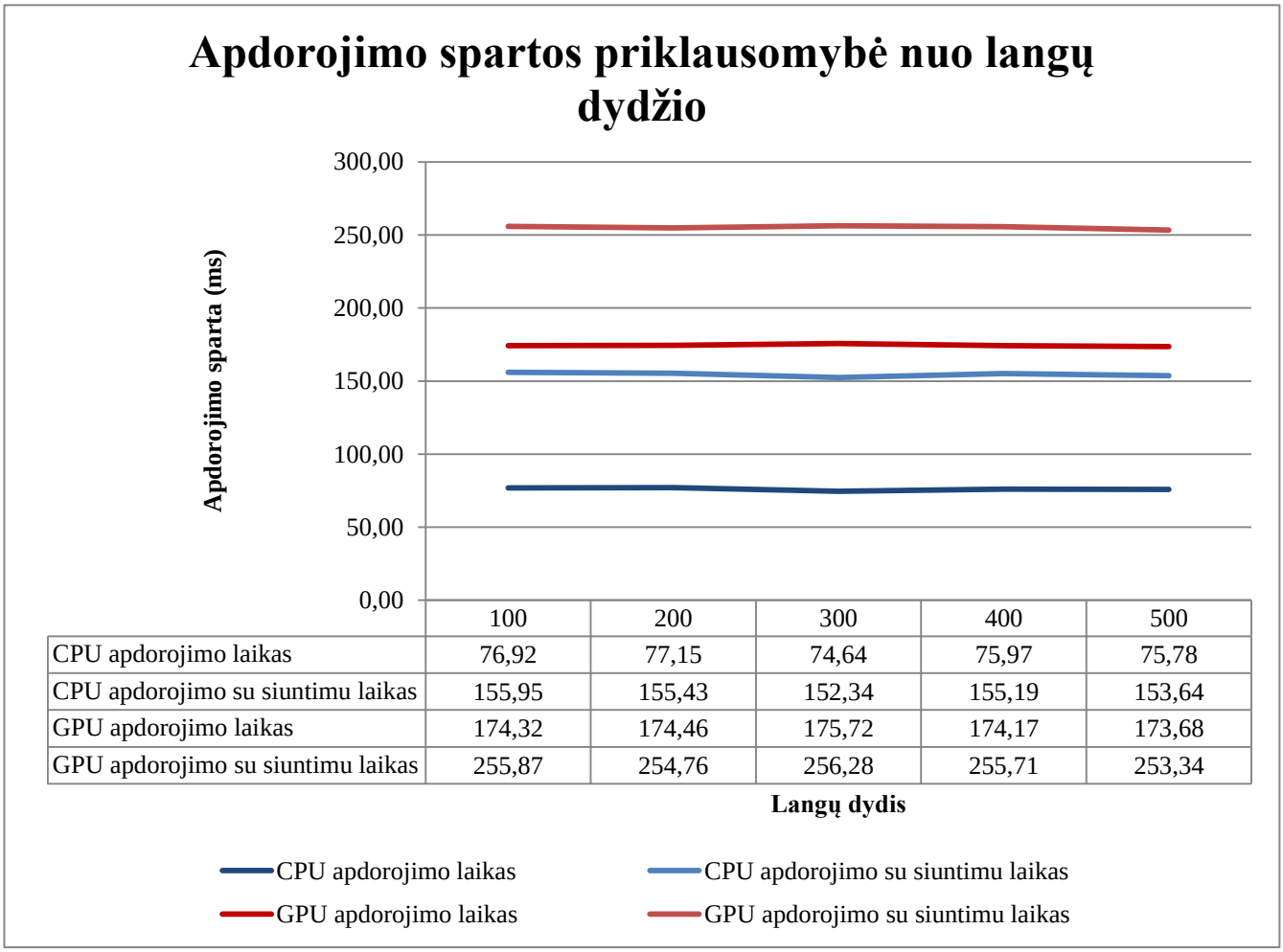
2.2.5. Agregatinių atributų skaičius



	2	4	6	8	10
CPU apdorojimo laikas	76,07	76,19	76,38	80,02	81,64
CPU apdorojimo su siuntimu laikas	153,40	155,30	158,04	162,15	158,47
GPU apdorojimo laikas	173,52	172,79	173,08	175,10	173,84
GPU apdorojimo su siuntimu laikas	253,21	254,24	255,87	255,78	259,57

Iš šio atlikto bandymo galima matyti, jog agregatinių atributų skaičius didelės įtakos apdorojimo spartai nedaro, nors turėtų būti priešingai. Ką galima pastebėti iš šio eksperimento, tai turbūt tiek, kad agregatinių atributų įtaka apdorojimo spartai labai priklauso nuo to, kiek rezultatų yra sugeneruojama. Šiuo atveju, kadangi yra tik vienas daugelio išrinkimų primityvaus įvykio stulpelis ir įvykių langų dydis pakankamai nedidelis, matyti, kad nesugeneruojama pakankamai didelis rezultatų skaičius, kad papildomas laikas, reikalingas apskaičiuoti agregatinių atributų reikšmėms, atsispindėtų rezultatuose. Viena iš priežasčių, kodėl sunku gauti reikšmę atspindinčius rezultatus yra ta, jog, kaip jau buvo minėta, kad išnaudoti paralelizmą reikia prisikurti daug komandų eilių, ypač didelėms primityvių įvykių aibėms, tačiau tokiu atveju buvo pastebėta, kad vykdymo aplinka pasidaro nestabili, t. y. pagrindiniai programos gijai laukiant, kol bus pasignalizuota, jog kažkoks įvykis baigtas, klaidos kodas kyla vienoj iš OpenCL prikurtų gijų.

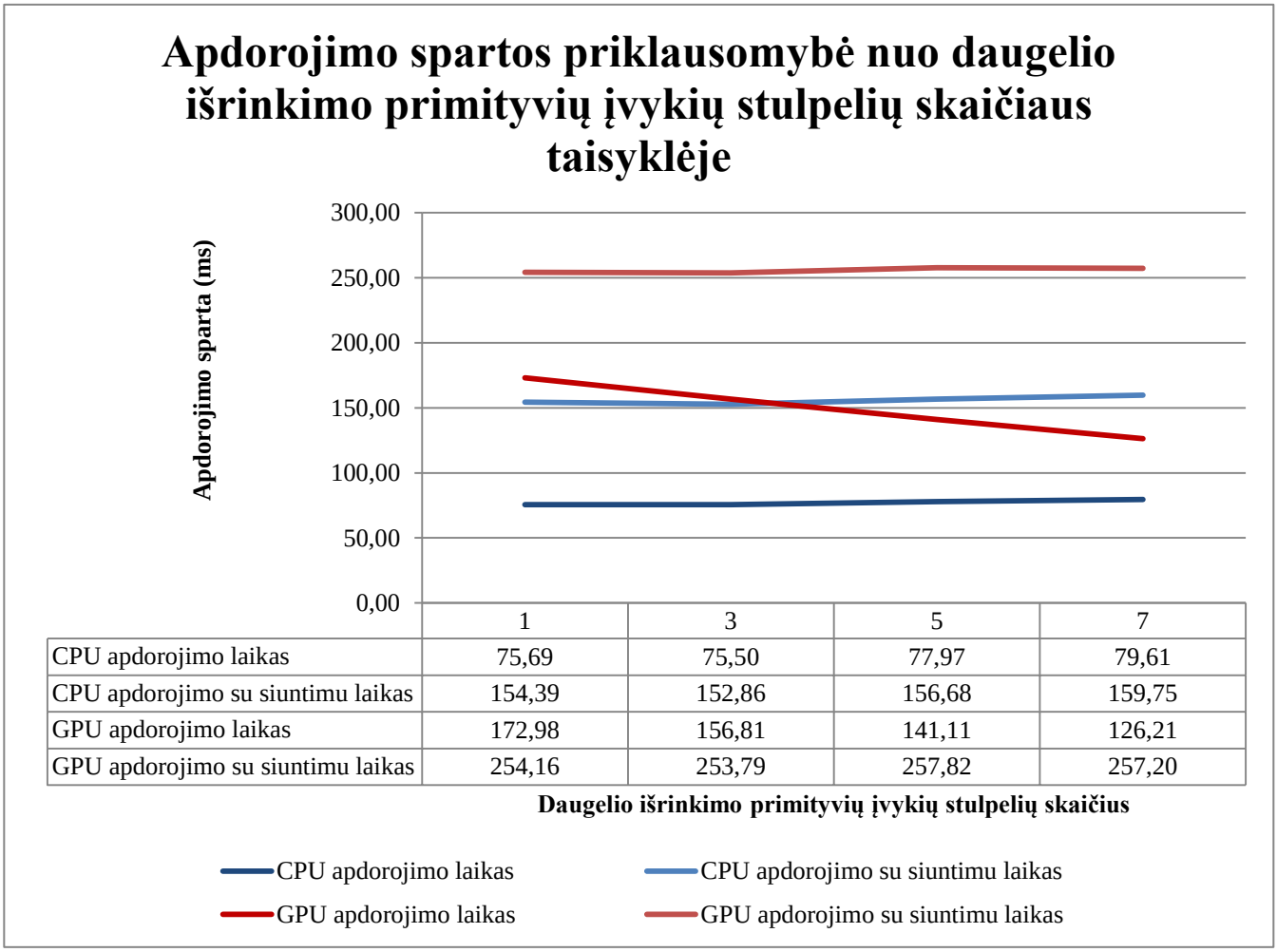
2.2.6. Langų dydis



	100	200	300	400	500
CPU apdorojimo laikas	76,92	77,15	74,64	75,97	75,78
CPU apdorojimo su siuntimu laikas	155,95	155,43	152,34	155,19	153,64
GPU apdorojimo laikas	174,32	174,46	175,72	174,17	173,68
GPU apdorojimo su siuntimu laikas	255,87	254,76	256,28	255,71	253,34

Apdorojimo spartos priklausomybės nuo langų dydžio taip pat pamatyti nepavyko. Galima daryti išvadą, jog taip nutiko todėl, kad pradinė užklausa nėra pakankamai sudėtinga, t. y. net ir paimant visus stulpelyje esančius įvykius nesusidaro pakankamai didelės skaičiavimų apimtys, kurias būtų galima atlikti paraleliai, tam reiktų arba labai smarkiai didinti primityvių įvykių aibės dydį, arba pradinę taisyklę padaryti sudėtingesne, pridedant primityvių įvykių su daugelio išrinkimo stulpeliais.

2.2.7. Daugelio išrinkimo primityvių įvykių stulpelių skaičius



	1	3	5	7
CPU apdorojimo laikas	75,69	75,50	77,97	79,61
CPU apdorojimo su siuntimu laikas	154,39	152,86	156,68	159,75
GPU apdorojimo laikas	172,98	156,81	141,11	126,21
GPU apdorojimo su siuntimu laikas	254,16	253,79	257,82	257,20

Analizuojant daugelio išrinkimo primityvių įvykių stulpelių skaičiaus įtaką apdorojimo spartai galima pastebėti, jog stulpelių skaičiui didėjant apdorojimo sparta mažėja, tačiau ne taip dramatiškai, kaip tai buvo galima stebėti didinant primityvių įvykių skaičių taisyklėje. Tam gali būti keletas priežasčių:

- pradinė taisyklė gali būti nepakankamai sudėtinga, t. y. per mažai primityvių įvykių kad pilnai išnaudotume grafinio procesoriaus paralelizmą;
- langai yra per maži ir išrinktų įvykių skaičius pakankamai neužpildo gijų bangos;
- gijų bangos varžosi tarp savęs dėl priėjimo prie atminties ir taip netenkama paralelizmo.

2.3. Skyriaus išvados

Atlikus eksperimentinį tyrimą ir išanalizavus gautus rezultatus buvo gautos tokios tarpinės išvados:

- siekiant išnaudoti grafinio procesoriaus lygiagretumo galimybes su daug taisyklių, turi būti pakankamai daug primityvių įvykių, kad būtų užpildomos gijų bangos;
- esant sudėtingoms taisyklėms, t. y. su daug primityvių įvykių stulpelių bei daugelio išrinkimo stulpelių galima gauti daugiau kaip keturis kartus didesnę įvykių apdorojimo spartą nei apdorojant su centriniu procesoriumi;
- parametrizuotų atributų skaičius taisyklėje praktiškai jokios įtakos apdorojimo spartai nedaro;
- atvejai nagrinėjantys apdorojimo spartos priklausomybę nuo langų dydžio, atributų skaičiaus primityviame įvykyje bei agregatinių atributų skaičiaus rezultate reikalauja papildomų tyrimų, nes OpenCL karkasas šiuo metu nėra pakankamai stabilus ir nerealizuoja ne pagal eilę vykdomų darbo komandų eilių, todėl nepavyksta atlikti bandymų su pakankamai didelėmis primityvių įvykių aibėmis, kurios parodytų aiškias priklausomybes tarp minėtų parametrų ir apdorojimo spartos.

3. Išvados ir pasiūlymai

Atlikus tyrimą buvo padarytos tokios galutinės išvados:

- išanalizavus atlikto eksperimentinio tyrimo rezultatus, buvo pastebėta, jog tinkamai išnaudotas grafinis procesorius gali padidinti įvykių apdorojimo spartą net daugiau kaip keturis kartus, tačiau tai pasiekti nėra lengva, ypač tokioms abstrakčioms sistemoms kaip buvo nagrinėjama šiame darbe;
- remiantis gautais eksperimentinio tyrimo rezultatais buvo nustatyta, kad grafinius procesorius geriausia įvykių apdorojimui naudoti tuomet kai sudėtingų įvykių atpažinimo taisyklės yra pakankamai sudėtingos, t. y. turi būti daug primityvių įvykių, daugelio išrinkimų stulpelių, daug taisyklių ir pakankamai dideli primityvių įvykių langai, kad apdorojant būtų kuo geriau išnaudojamas grafinio procesoriaus siūlomas paralelizmas;
- atliekant eksperimentinio tyrimo bandymus, buvo nustatyta, kad OpenCL 2.0 karkasas šiuo metu nėra pats tinkamiausias karkasas sudėtingų įvykių apdorojimo sistemoms realizuoti, nes neturi lengvai naudojamo asinchroninių komandų eilių mechanizmo, kurio analogai yra pateikiami naudojimui kitų GPU technologijų tokių kaip CUDA ar Metal, ir OpenCL 2.0 versija yra dar labai nauja, todėl neturi reikiamų programavimo įrankių bei kartais kyla stabilumo problemų, tačiau naujos karkaso suteikiamos galimybės, tokios kaip bendra virtuali atmintis ir vidinis paralelizmas palengvina programavimą, o galbūt ir padidina spartą, tačiau tam reikėtų papildomų tyrimų, kai karkasas taps stabilesnis.

Realizuojant eksperimentiniam tyrimui reikalingą modelį, vykdomą grafinio procesoriaus, buvo išsiaiškinta, jog programavimas grafiniam procesoriui yra gerokai sudėtingesnis nei centriniam ir, norint gauti gerų rezultatų, reikia išmanyti bei programuojant atsižvelgti į fizinės įrangos architektūrą, t. y. siekiant didesnės spartos rekomenduojama stengtis:

- kad paleidžiamos gijų bangos būtų kuo pilniau užpildytos;
- kad gijų bangos, vykdomos skirtingų skaičiavimo vienetų, tuo pat metu skaitytų skirtingus atminties blokus, pasiekiamus skirtingais atminties kanalais;
- kad atminties operacijos, atliekamos gijų, esančių toje pačioje gijų bangoje, būtų atliekamos suderintai – su greta esančiais atminties žodžiais.

Šaltinių sąrašas

[ABB+04] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. [žiūrėta 2015-11-11]. Prieiga per Internetą: <http://ilpubs.stanford.edu:8090/641/1/2004-20.pdf>

[AMD12] AMD. AMD Graphics Core Next (GCN) Architecture. [žiūrėta 2015-11-12]. Prieiga per Internetą: <https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf>

[AMD13] AMD. AMD Accelerated Parallel Processing: OpenCL Programming Guide. [žiūrėta 2015-11-12]. Prieiga per Internetą: <http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf>

[BBD+03] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 253-264, 2003.

[BW01] S. Babu and J. Widom. Continuous Queries over Data Streams. [žiūrėta 2015-12-05]. Prieiga per Internetą: <http://ilpubs.stanford.edu:8090/527/1/2001-9.pdf>

[CM12] G. Cugola and A. Margara. Low Latency Complex Event Processing on Parallel Hardware. Journal of Parallel and Distributed Computing, **72**(2), pp. 205–218, 2012.

[DGP+07] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A General Purpose Event Monitoring System. Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1100-1102, 2007.

[FJL+01] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 115–126, 2001.

[Hir12] M. Hirzel. Partition and Compose: Parallel Complex Event Processing. Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, pp. 191-200, 2012.

[LKC+10] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey.

Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. [žiūrėta 2015-12-05]. Prieiga per Internetą: <http://sbel.wisc.edu/Courses/ME964/Literature/LeeDebunkGPU2010.pdf>

[Luc01] D. C. Luckham. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, 2001.

[MBS15] M. Pinnecke, D. Broneske, and G. Saake. Toward GPU Accelerated Data Stream Processing. [žiūrėta 2015-12-05]. Prieiga per Internetą: <http://ceur-ws.org/Vol-1366/paper15.pdf>

[MSW07] K. Munagala, U. Srivastava, and J. Widom. Optimization of Continuous Queries with Shared Expensive Filters. Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 215-224, 2007.

[Nvi09] Nvidia. OpenCL Programming Guide for the CUDA Architecture. [žiūrėta 2015-12-06]. Prieiga per Internetą: <http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf>

[Nvi14] Nvidia. NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made. [žiūrėta 2015-12-06]. Prieiga per Internetą: <http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF>

[SW04] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 263-274, 2004.

[TGN+92] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. Proceedings of the 1992 ACM SIGMOD Intl. Conf. on Management of Data, pp. 321-330, 1992.

[TS12] J. Tompson and K. Schlachter. An Introduction to the OpenCL Programming Model. [žiūrėta 2015-12-10]. Prieiga per Internetą: <http://www.cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf>

[WC96] J. Widom and S. Ceri. Active Database Systems: Triggers and Rules for Advanced

Database Processing. Morgan Kaufmann, San Francisco, 1996.

[WDR06] E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. Proceedings of the 2006 ACM SIGMOD international conference on Management of data, pp. 407-418, 2006.

[WPS+10] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. ISPASS-2010 2010 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 235-246, 2010.

[YF11] Y. Zhang and F. Mueller. GStream: A General-Purpose Data Streaming Framework on GPU Clusters. [žiūrėta 2015-12-12]. Prieiga per Internetą: <http://moss.csc.ncsu.edu/~mueller/ftp/pub/mueller/papers/icpp11-1.pdf>

Santrumpos ir sąvokų apibrėžimai

CPU – bendros paskirties procesorius (angl. central processing unit).

GPU – vaizdo apdorojimo procesorius (angl. graphics processing unit).

Gija – nepriklausoma, nedaloma, mikroprocesoriaus branduolio vykdoma komandų seka.

Gijų banga – gijų aibė, kurią sudarančios gijos vienu metu yra vykdomos susietai, t. y. atlieka vieną ir tą pačią komandų dekoderio iškoduotą instrukciją su atskirais instrukcijos argumentais bei adresavimo registrais, t. y. ta pati instrukcija yra atliekama su skirtingais duomenimis.

Primityvus įvykis – duomenų struktūra, nusakanti signalus gaunamus iš fizinių įrenginių, kitų programų sistemų, tinklo servisų, vartotojo veiksmų ar kitų šaltinių.

Įvykių srautas – begalinis duomenų šaltinis, tiekiantis primityvius įvykius.

Sudėtingas įvykis – duomenų struktūra, nusakanti aukštesnio lygio konkrečios srities įvykį, ir gaunama analizuojant primityvių įvykių srautą.

Šakninis įvykis – primityvus įvykis, kuris sudėtingą įvykį aprašančioje taisyklėje nėra susietas su kitais primityviais įvykiais laiko apribojimais ir kurio aptikimas duomenų sraute yra žymė, rodanti jog turėtų būti atliekamas taisyklei aktualių primityvių įvykių, sukauptų laikinojoje talpykloje, apdorojimas, siekiant sugeneruoti sudėtingesnį įvykį.

Tęstinių užklausų duomenų srautams sistema – programa, kuri gaunamuose, primityvius įvykius nusakančiuose duomenyse, ieško tam tikrus apribojimus atitinkančių ir tam tikra tvarka išsidėsčiusių duomenų, kurie leidžia sugeneruoti sudėtingesnį įvykį, kurį duomenų sraute leidžia atpažinti ir sugeneruoti vartotojo apibrėžta primityvių įvykių užklausa.