Kauno technologijos universitetas

Elektros ir elektronikos fakultetas

Programuojamos logikos taikymas duomenų apdorojimo spartinimui daiktų interneto pakraščio sluoksnyje

Baigiamasis magistro projektas

Mindaugas Punys

Projekto autorius

Prof. dr. Žilvinas Nakutis

Vadovas

Kaunas, 2024

Kauno technologijos universitetas

Elektros ir elektronikos fakultetas

Programuojamos logikos taikymas duomenų apdorojimo spartinimui daiktų interneto pakraščio sluoksnyje

Baigiamasis magistro projektas

Elektronikos inžinerija (6211EX012)

Mindaugas Punys

Projekto autorius

Prof. dr. Žilvinas Nakutis

Vadovas

Doc. dr. Mindaugas Knyva

Recenzentas

Kaunas, 2024

Kauno technologijos universitetas

Elektros ir elektronikos fakultetas

Mindaugas Punys

Programuojamos logikos taikymas duomenų apdorojimo spartinimui daiktų interneto pakraščio sluoksnyje

Akademinio sąžiningumo deklaracija

Patvirtinu, kad:

1. baigiamąjį projektą parengiau savarankiškai ir sąžiningai, nepažeisdamas kitų asmenų autoriaus ar kitų teisių, laikydamasis Lietuvos Respublikos autorių teisių ir gretutinių teisių įstatymo nuostatų, Kauno technologijos universiteto (toliau – Universitetas) intelektinės nuosavybės valdymo ir perdavimo nuostatų bei Universiteto akademinės etikos kodekse nustatytų etikos reikalavimų;

2. baigiamajame projekte visi pateikti duomenys ir tyrimų rezultatai yra teisingi ir gauti teisėtai, nei viena šio projekto dalis nėra plagijuota nuo jokių spausdintinių ar elektroninių šaltinių, visos baigiamojo projekto tekste pateiktos citatos ir nuorodos yra nurodytos literatūros sąraše;

3. įstatymų nenumatytų piniginių sumų už baigiamąjį projektą ar jo dalis niekam nesu mokėjęs;

4. suprantu, kad išaiškėjus nesąžiningumo ar kitų asmenų teisių pažeidimo faktui, man bus taikomos akademinės nuobaudos pagal Universitete galiojančią tvarką ir būsiu pašalintas iš Universiteto, o baigiamasis projektas gali būti pateiktas Akademinės etikos ir procedūrų kontrolieriaus tarnybai nagrinėjant galimą akademinės etikos pažeidimą.

Mindaugas Punys

Patvirtinta elektroniniu būdu

Punys, Mindaugas. Programuojamos logikos taikymas duomenų apdorojimo spartinimui daiktų interneto pakraščio sluoksnyje. Magistro baigiamasis projektas / vadovas prof. dr. Žilvinas Nakutis; Kauno technologijos universitetas, Elektros ir elektronikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Elektronikos inžinerija, inžinerijos mokslai.

Reikšminiai žodžiai: Programuojama logika, FPGA, HLS, Daiktų interneto pakraščio sluoksnis.

Kaunas, 2024. 63 p.

Santrauka

Šiame darbe yra pristatomas atliktas programuojamos logikos tyrimas realizuojant daiktų interneto pakraščio sistemą ir lyginant įterptinių sistemų ir programuojamos logikos greitaveiką ir našumą. Pirmoje dalyje yra pristatoma atlikta literatūros analizė: daiktų interneto pakraščio sąvoka ir taikymo būdai, palyginamosios studijos tiriančios įterptinių sistemų, programuojamos logikos ir debesų kompiuterijos taikymą daiktų internetui. Antra dalis yra skirta pristatyti tyrimo metodologiją, taikymo pavyzdį bei tyrimo uždavinių sprendimus. Trečioje darbo dalyje yra apžvelgiami esminiai etapai realizuojant taikymo pavyzdį. Šie etapai apima dalelių spiečiaus optimizavimo algoritmo realizavimą, Furjė transformacijos pritaikymą, programos optimizavimo etapus bei realizuotos programos architektūros apžvalgą. Ketvirtame skyriuje yra pateikiami tyrimo rezultatai: atliktas įrangos greitaveikos, kainos ir energetinio efektyvumo palyginimas. Galiausiai paskutiniame skyriuje yra aptariama tyrimo apimtis bei alternatyvūs tyrimo testavimo metodai. Taip pat šiame skyriuje yra apžvelgiama pagrindiniai iššūkiai kuriant programas aukšto lygio sinteze.

Punys, Mindaugas. Application of FPGA to Accelerate IoT Data Processing at the Edge Layer. Master's Final Degree Project / supervisor Prof. Dr. Žilvinas Nakutis; Faculty of Electrical and Electronics Engineering, Kaunas University of Technology.

Study field and area (study field group): Electronics Engineering, Engineering Sciences

Keywords: Programmable logic, FPGA, HLS, IoT Edge.

Kaunas, 2024. 63 pages.

Summary

This thesis presents a study of programmable logic in an IoT Edge system and compares the speed and performance of embedded systems and programmable logic. In the first part literature analysis is presented: the concept and applications of the IoT edge, comparative studies investigating the application of embedded systems, programmable logic and cloud computing for the IoT Edge. The second part is dedicated for research methodology, case study and research objectives overview. The third part of the thesis provides an outline of the essential steps in the realization of the case study. These steps include the realization of the particle swarm optimization algorithm, the application of the Fourier transform, steps taken to optimize the example application and an overview of the architecture of the developed application. Chapter 4 presents the results of the study: a comparison of the speed, cost and energy efficiency of the hardware that was used. Finally, the last section discusses the scope of the study and alternative testing methods. Also in this chapter, the main challenges in developing applications in high-level synthesis are reviewed.

Turinys

Paveikslų sąrašas

Santrumpų ir terminų sąrašas

Santrumpos:

FPGA (angl. *Field Programmable Gate Arrays*) – programuojamos logikos įrenginiai, programuojamų loginių vartų masyvai;

IoT (angl. *Internet of Things*) – daiktų internetas;

HMI (angl. *Human Machine Interaction*) – vartotojo ir mašinos sąsaja;

IIoT (angl. *Idustrial Internet of Things*) –pramoninis daiktų internetas;

M2M (angl. *Machine to Machine*) – tarp-mašininė sąsaja;

CPS (angl. *Cyber-Physical System*) – kibernetinė–fizinė sistema;

PSO (angl. *Particle Swarm Optimization*) – dalelių spiečiaus optimizavimo algoritmas;

DSP (angl. *Digital Signal Processing*) – skaitmeninių signalų apdorojimas;

HDL (angl. *Hardware Descriptive Language*) – aparatūrinės įrangos aprašymo kalbos;

HLS (angl. *High Level Synthesis*) – aukšto lygio sintezė;

CNN (angl. *Convolutional Neral Networks*) – konvoliuciniai neuroniniai tinklai;

WSL (angl. W*indows Subsystem for Linux*) – *Linux* posistemė *Windows* operacinei sistemai;

Įvadas

Daiktų interneto sistemos yra fundamentali šiuolaikinės infrastruktūros dalis, sukurianti sąsają tarp interaktyvių įrenginių ir interneto. Augant kompiuterių skaičiavimo resursams, dažnai daiktų interneto prietaisai ne tik kaupia, bet ir apdoroja informaciją prieš siunčiant ją į debesų kompiuteriją. Šis daiktų interneto sluoksnis, apdorojantis duomenis, yra vadinamas informacijos apdorojimo pakraščiu (angl. *edge*). Duomenų apdorojimas pakraštiniame daiktų interneto sluoksnyje sumažina skaičiavimo trukmes, internetu siunčiamų duomenų srautą ir bendrą energijos suvartojimą.

Kylantis poreikis vis dažniau integruoti mašininį mokymą ir sudėtingus skaičiavimus į kasdieninius išmaniuosius daiktus skatina ieškoti procesorių ir technologijų su vis didesniais skaičiavimo ištekliais, tačiau išlaikyti įterptinėms sistemoms keliamus energetinius, ekonominius ir fizinius reikalavimus. Taigi yra aktualu analizuoti įvairius įterptinių sistemų ir daiktų interneto realizavimo metodus bei gilintis, kokie yra šių technologijų privalumai bei trūkumai.

Šiame darbe yra analizuojama ir tiriama, kaip pritaikyti programuojamos logikos įrenginius – programuojamų loginių vartų masyvus – *FPGA* matricas (angl. *Field Programmable Gate Arrays*). Šiame darbe yra pateikia sukurta sistema, skirta daiktų interneto pakraščio sluoksnio duomenų apdorojimui su *FPGA* komponentais. Ši sistema yra lyginama su įprasta „*Linux*“ įterptine sistema bei debesų kompiuteriją atitinkančia sistema. Darbe yra lyginama sistemų sparta, resursų sąnaudos ir energetinis efektyvumas.

Tyrimo tikslas – įvertinti programuojamos logikos privalumus ir trūkumus atliekant daiktų interneto pakraščiui tipinių skaičiavimų spartinimą. Palyginti programuojamą logiką su įterptinių mikrovaldiklių ar kompiuterinių sistemų panaudojimu, siekiant sumažinti debesų serveriuose atliekamų apdorojimų apimtis.

Projekto uždaviniai:

1) realizuoti tipinę pakraščio skaičiavimų spartinimo įterptinę sistemą, panaudojant programuojamą logiką ir įterptinį mikrovaldiklį;
2) optimizuoti programuojamos logikos sistemos realizaciją, atsižvelgiant į skaičiavimų greitaveiką ir resursų sąnaudas;
3) įvertinti ir palyginti programuojamos logikos ir įterptinių sistemų mikrovaldiklių greitaveiką;
4) įvertinti programuojamos logikos privalumus ir trūkumus pagal resursų sąnaudas ir energetinį efektyvumą.

1. Mokslinės literatūros apžvalga

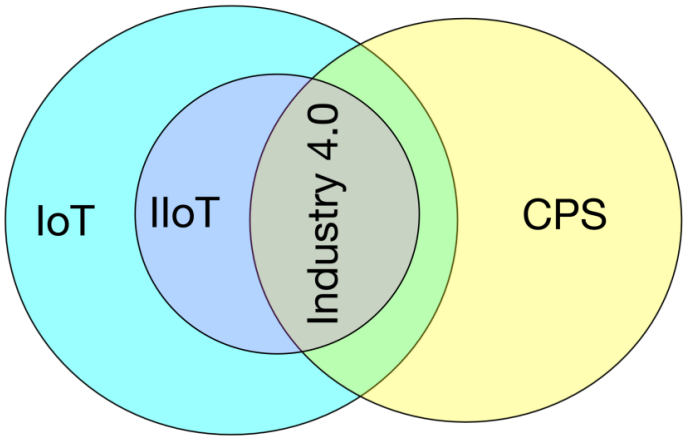
Šiame skyriuje apžvelgiama daiktų interneto ir daiktų interneto pakraščio sluoksnio sąvokos bei tendencijos. Analizuojami paplitę taikymo metodai bei pritaikymo sritys, kuriose yra realizuoti modernūs daiktų interneto pakraščio sprendimai.

1.1. Daiktų internetas

Daiktų internetas (angl. *IoT*) tai konceptas, apibūdinantis kasdieninius daiktus, kurie yra neatskiriamai susieti su internetu ir taip tampa išmaniais prietaisais, padedančiais rinkti ir apdoroti informaciją bei tobulinti paslaugų kokybę [1]. Pats daiktų interneto terminas yra gana senas, įvardintas 1999 metais Kevino Aštono, tačiau išgarsėjęs tik 2010 metų vasarą [2]. 2022 metais paskelbtoje statistikoje buvo nurodyta, kad jau 2021 metais *IoT* prietaisų buvo virš 11 milijardų ir šis skaičius turėtų pasiekti 17 milijardų prietaisų per 2024 metus [3]. Daiktų internetas yra itin svarbus, nes suteikia galimybes automatizuoti įvairius procesus, pagerinti gyvenimo kokybe bei realizuoti išmanius sprendimus tokius kaip: išmanieji miestai, išmanusis transportas, oro taršos sekimas ir mažinimas bei sveikatos apsaugos kokybės tobulinimas [1].

Daiktų internetas paprastai yra skaidomas į keletą grupių pagal daiktų pritaikymą:

- **vartotojams skirtas daiktų internetas** (vadinamas tiesiog *IoT*). Šie daiktai yra skirti ir kuriami galvojant apie žmogų, siekiant pagerinti žmogaus gyvenimo kokybę ir sutaupyti laiko. Daiktai įprastai turi vartotojo ir mašinos (angl. *HMI – Human Machine Interaction*) bei vartotojo ir serverio sąsajas. Šiai sričiai priskiriami išmanių namų prietaisai, dėvimi bei išmanūs buities pritaisai;
- **pramoninis daiktų internetas** (*IIoT* – angl. *Idustrial Internet of Things*). Daiktai skirti susieti pramonines mašinas su valdymo sistemomis ir informacinėmis technologijomis. Šie prietaisai leidžia rinkti duomenis ir juos analizuoti siekiant optimizuoti gamybą. Šiai sričiai yra būdinga tarp-mašininė sąsaja (angl. *M2M – Machine to Machine*). Šiai sričiai priskiriamos išmaniosios valdymo sistemos bei įvairios prevencinės gedimų šalinimo sistemos;
- **pramonė 4.0** – bendras konceptas, kurio tikslas yra integruoti išmaniuosius prietaisus nuo vartotojo iki gamybos proceso;
- ***CPS* (angl. *Cyber-Physical System*)** – kibernetinė–fizinė sistema. Tai bendras konceptas, apimantis *IoT*, *IIoT* ir Pramonę 4.0, apibūdinantis fiziškai ir internetu susietas sistemas.



1 pav. Daiktų interneto sistemų skirstymas [1]

Tikimasi, kad daiktų internetas ateityje smarkiai pakeis ir patobulins žmonių gyvenimo kokybę, tačiau jau dabar yra konkrečių ir išmatuojamų privalumų, kuriuos teikia daiktų internetas:

- **daiktų internetas tobulina ir skatina pramonės produktyvumą ir efektyvumą**. Pavyzdžiui „*Black & Decker*“ įmonei gamybos linijoje pritaikius išmanias kokybės vertinimo sistemas, produktyvumas padidėjo apie 10 % ir defektų skaičius sumažėjo apie 16% [2];
- **įrangos stebėjimas leidžia iš anksto nuspėti kritinius gedimus ir išvengti didelių nuostolių**. UK vandens tiekėjo instaliuoti išmanūs vandens skaitliukai vartotojams leidžia sutaupyti apie 12% vandens ir iš anksto pašalinti gedimus, kurie aptinkami pas 3% vartotojų [4];
- **išmanūs prietaisai tobulina sveikatos apsaugą**. Išmanūs *CAALYX* prietaisas padeda sekti sergančiųjų sveikatą ir iškviečia pagalbą seniems žmonėms nukritusiems ant žemės [5];
- **išmanūs prietaisai padeda užtikrinti darbuotojų saugumą**. Išmanūs jutikliai sekantys statybininkų aukštį bei aplinkos vaizdą padeda išvengi nelaimingų atsitikimų [6].

Tačiau išmaniems prietaisams yra keliami išskirtiniai reikalavimai, kurie tampa iššūkiu derinant išmanių daiktų privalumus ir trūkumus:

- **energetinis efektyvumas.** Išmanieji daiktai dažnai yra naudojami neprijungti prie elektros tinklo, todėl reikia alternatyvių energijos šaltinių, kaip baterijos, akumuliatoriai ar saulės elementai. Tai lemia aukštus energetinio efektyvumo reikalavimus ir ribotus skaičiavimo išteklius;
- **realaus laiko sistemos.** Tiek taikant pramonėje, tiek žmonių sveikatos apsaugai, tampa itin aktualus išmaniųjų sistemų gebėjimas greitai vertinti ir reaguoti į aplinką. Šis reikalavimas reikalauja atitinkamų skaičiavimo resursų, kurie lemia aukštesnes energijos sąnaudas;
- **sąveika su kitais prietaisas.** Išmanūs prietaisai privalo veikti kartu su kitomis sistemomis dalinantis informacija ir ją apdorojant;
- **duomenų saugumas.** Surenkami duomenys iš galinių taškų turi būti užšifruoti ir apsaugoti nuo pašalinių akių bei įtakos. Tai reikalauja skaičiavimo resursų koduojant informaciją ir kenkia energetiniam efektyvumui.

1.2. Daiktų interneto taikymo sritys

Daiktų internetas gali būti pritaikytas beveik bet kurioje mūsų gyvenimo srityje. Šiandien didelė infrastruktūros dalis dar vis yra neišmani ir yra mažai intereso ją keisti, kol tenkinami keliami reikalavimai. Tačiau atnaujinant infrastruktūrą ir keliant vis aukštesnius reikalavimus daiktų internetas skinasi kelią kasdieniniame gyvenime ir pramonėje. Apžvalginė studija sudarė daiktų interneto taikymo kategorijas bei privalumus, kuriuos teikia išmanūs sprendimai [7]:

1) **Transportas ir logistika**:
   a) **logistika** – naudojant *RFID* ir *NFC* technologijas galima stebėti ir greitai reaguoti į produktų tiekimo grandinės situaciją. Ši technologija leidžia mažmeninės prekybos įmonėms turėti vos kelių dienų atsargas sandėlyje;
   b) **vairuotojams skirta pagalba** – išmanūs vairuotojams padedantys jutikliai ir vaizdo kameros padeda užtikrinti vairuotojų saugumą, o įmonėms – padeda skatinti ekologišką ir saugų vairavimą;
   c) **viešojo transporto taškai** – *NFC* taškai ir švieslentės tiekia informaciją apie viešojo transporto dienotvarkę ir maršrutus;
   d) **transportuojamų daiktų aplinkos vertinimas** – gendantys daiktai, tokie kaip vaisia ar daržovės reikalauja specialių sąlygų juos transportuojant, siekiant užtikrinti jų kokybę.

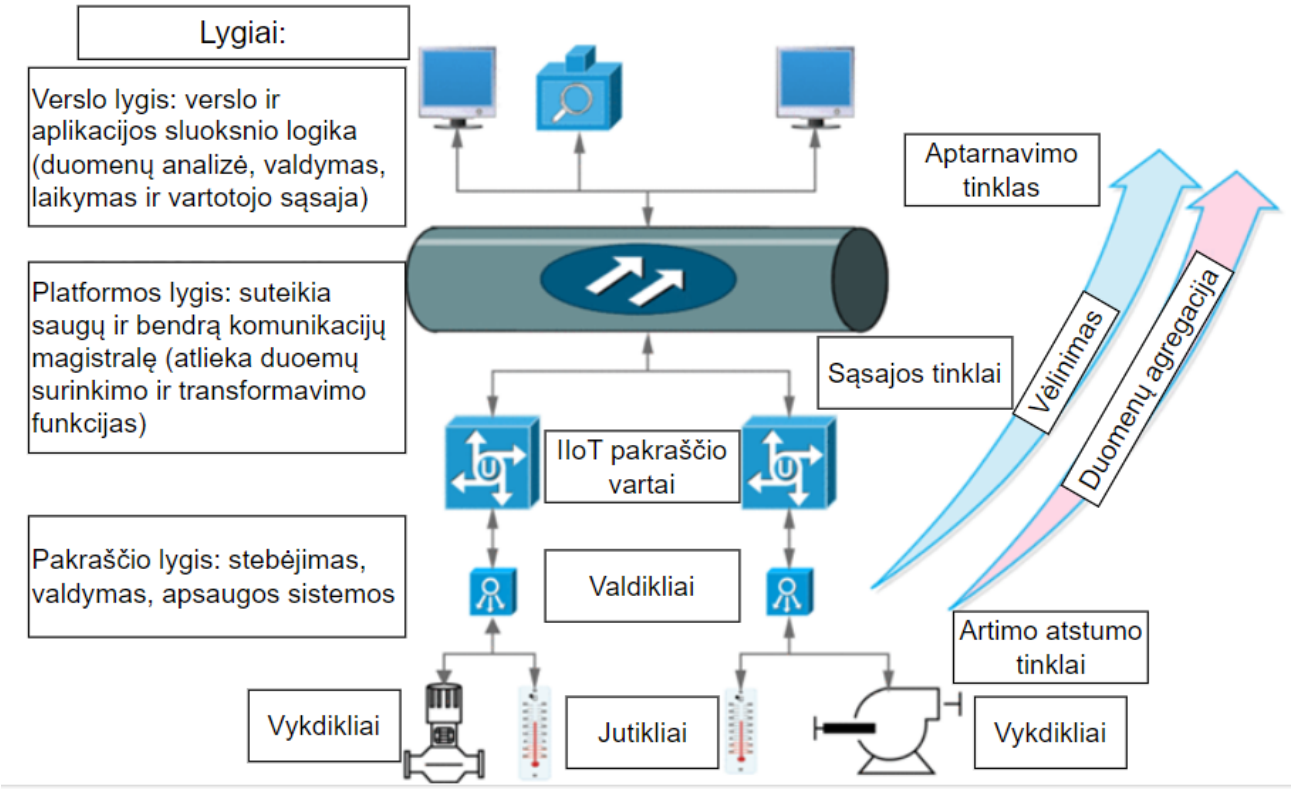
Išmanūs jutikliai padeda sekti šią aplinką ir užtikrinti, kad transportuojami daiktai saugiai pasieks savo tikslą.

e) **Išmanioji žemdirbystė** – duomenų analize ir mokslu paremta žemdirbystė, siekianti optimizuoti išteklių sąnaudas ir produktų kokybę.

2) **Sveikatos apsauga**:
   a) **judėjimo sekimas** – sekant ir analizuojant kaip žmonės juda ligoninėse galima užtikrinti efektyvesnį patalpų išplanavimą;
   b) **identifikacija ir autorizavimas** – išmanūs prietaisai leidžia užtikrinti saugumą ir teisingą bei išsamią pacientų ligos istoriją;
   c) **duomenų rinkimas** – automatizuotas domenų rinkimas ir apdorojimas siekiant mažinti eiles ir agreguoti informaciją;
   d) **pacientų būklės sekimas** – siekiant greitai ir realiu laiku reaguoti į sveikatos problemas.
3) **Išmani aplinka**:
   a) **patogūs namai ir biurai** – išmanūs prietaisai reguliuojantys apšvietimą, šilumą ir drėgmę. Šie prietaisai ne tik užtikrina patogią aplinką, bet ir mažina energijos sąnaudas. Pavyzdžiui išmanus daiktų interneto tinklas su informacijos apdorojimu pakraštyje gali būti pritaikytas dinamiškai reguliuoti aplinką panaudojant dalelių spiečiaus optimizavimo algoritmą – *PSO* (angl. *Particle Swarm Optimization*) [8];
   b) **pramoninės gamyklos** – gamyklos gali būti robotizuotos ir optimizuotos pasitelkiant išmanius jutiklius ir *RFID* ženklelius. Išmanūs jutikliai taip pat padeda atlikti prevencinę įrangos priežiūrą ir išvengti gedimų [9];
   c) **išmanūs muziejai ir sporto salės** – išmanūs muziejai gali suteikti galimybę lengvai ir interaktyviai apžiūrėti ekspozicijas, o išmanios sporto salės suteikia galimybę sekti sveikatos ir kūno parametrus.
4) **Asmeninis ir socialinis gyvenimas**:
   a) **socialiniai tinklai** – išmanūs prietaisai padeda automatiškai atnaujinti informaciją socialiniuose tinkluose;
   b) **užklausų saugojimas** – padeda sekti gyvenimo įvykius ir tendencijas;
   c) **pamesti ir pavogti daiktai** – išmanūs pakabukai padeda sekti ir surasti pamestus bei pavogtus daiktus.
5) **Futuristiniai sprendimai**:
   a) autonominiai taksi;
   b) išmanūs miestai.

1.3. Daiktų interneto pakraščio sluoksnis (*edge*)

Per laiką yra nusistovėjus plačiai priimtina daiktų interneto struktūra, kurią sudaro trys sluoksniai: daiktų interneto pakraščio, platformos ir verslo sluoksniai.



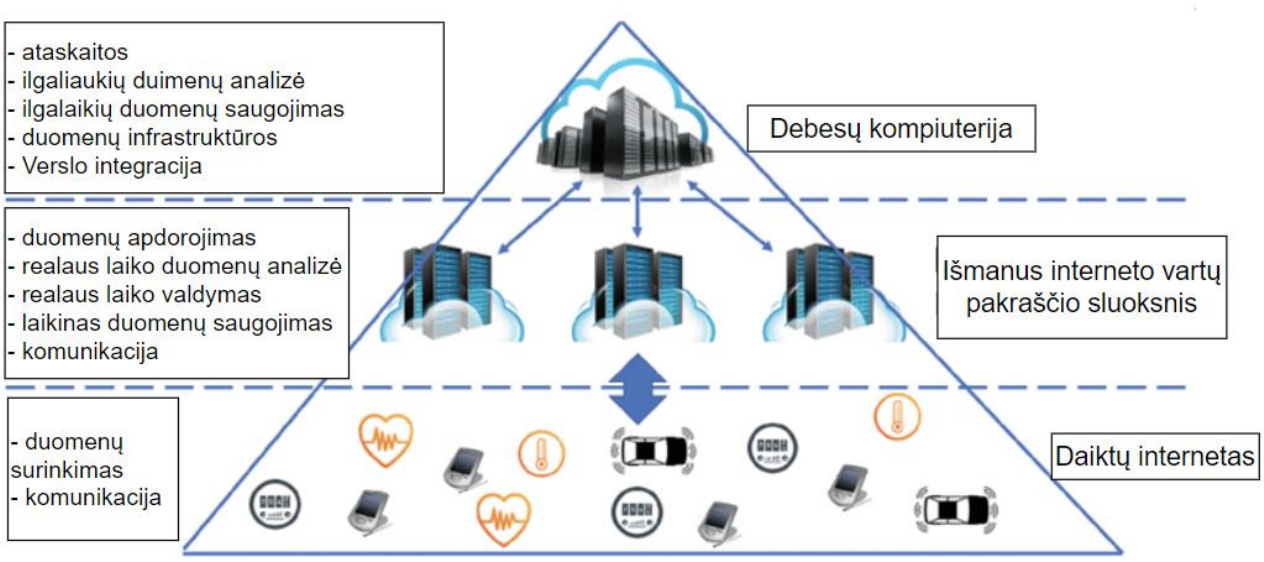
2 pav. Daiktų interneto sistemų skirstymas [1]

Daiktų interneto pakraščio sluoksnis (angl. *edge*) – tai išmanieji prietaisai, esantys ties fizinio pasaulio ir interneto riba, tarpusavyje sąveikaujantys ir renkantys informaciją bei perduodantys į aukštesnį lygmenį. Šį sluoksnį sudaro jutikliai, valdikliai ir įvairios pavaros bei interneto vartai perduodantys informaciją į aukštesnį duomenų apdorojimo lygmenį. Šis daiktų tinklas yra susietas pakraščio vartų su platesniu tinklu – platforma, kuria yra perduodama informacija į serverius, kur informacija yra apdorojama ir atvaizduojama vartotojui.

Kaip matyti **2** paveikslėlyje, standartinėje *IIoT* sistemoje duomenys yra siunčiami aukštyn iš jutikliu ir valdiklių į platformą ir serverį. Kuo toliau siunčiami duomenys, tuo atsiranda vis didesnis vėlinimas. Debesų kompiuterija suteikia praktiškai neribotus resursus, tačiau tai kainuoja neišvengiamą vėlinimą persiunčiant duomenis.

1.4. Informacijos apdorojimas daiktų interneto pakraščio sluoksnyje

Alternatyva įprastam daiktų internetui, paremtam duomenų agregavimu debesų saugyklose, yra duomenų apdorojimas pakraščio sluoksnyje. Ši daiktų interneto struktūra bando išnaudoti pakraščio sluoksnio ir debesų kompiuterijos privalumus: pakraščio sluoksnio kompiuterija teikia greitaveiką, tačiau turi ribotą atmintį ir skaičiavimo resursus, o debesų kompiuterija teikia neribotus resursus greitaveikos kaina. Vienas paprasčiausiai suprantamų ir lengviausiai įgyvendinamų pakraščio sluoksnio kompiuterijos modelių yra hierarchinis [10].



3 pav. Hierarchinis daiktų interneto pakraščio sluoksnio kompiuterijos modelis [10]

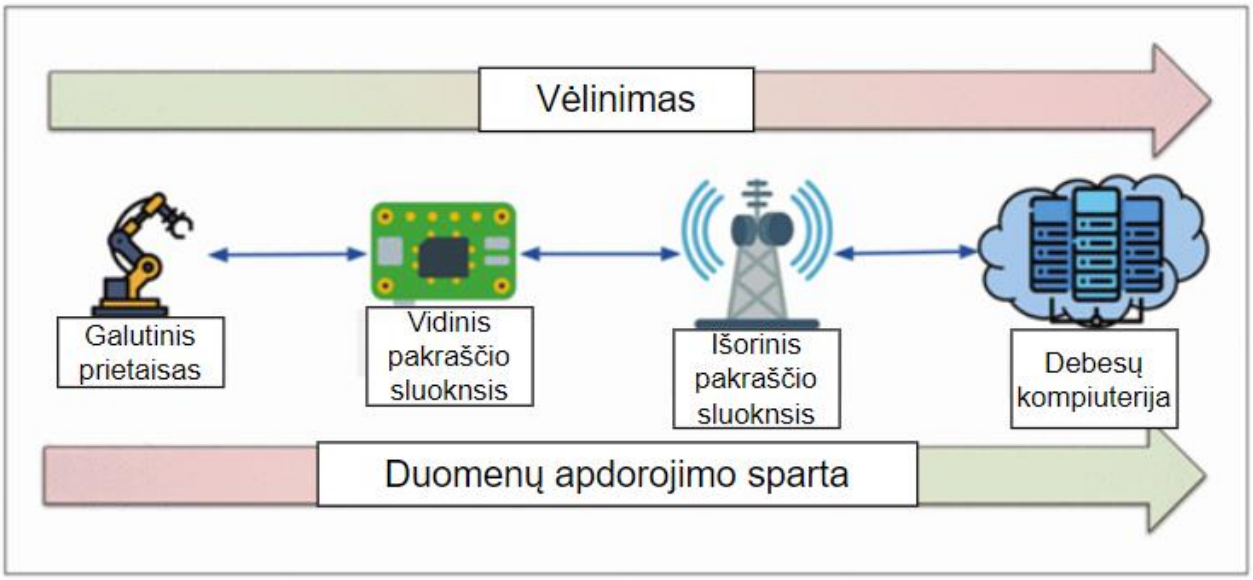
Šis modelis yra sudarytas iš trijų sluoksnių:

- *IoT* prietaisų – jutiklių, išmanaus transporto ar buities prietaisų, kurie teikia informaciją tinklui bei vykdo komandas;
- išmanių interneto vartų ir daiktų interneto pakraščio skaičiavimo stočių – serverių ir skaičiavimo įrangos esančios arti duomenis renkančių prietaisų, ir debesų kompiuterijos serverių. Šis sluoksnis yra atsakingas už duomenų apdorojimą realiu laiku ir realaus laiko ragavimą į situaciją. Šis sluoksnis taip pat gali išnaudoti ribotus resursus laikinam duomenų saugojimui;
- debesų kompiuterija šiame modelyje yra atsakinga už ilgalaikį duomenų saugojimą ir analizę. Šiame lygyje taip pat galima integruoti verslo sluoksnį ir vartotojo sąsają.

Hierarchinis *IoT edge* sluoksnių kompiuterijos modelis išnaudoja fundamentalias daiktų interneto savybes ir pakraščio sluoksnio privalumus, kurie yra analizuojami sekančiuose poskyriuose.

1.4.1. Duomenų perdavimas

Greitas sistemos reakcijos laikas yra vienas iš pagrindinių daiktų interneto pakraščio skaičiavimo privalumų. Programos reakcijos laiką sudaro dvi komponentės: laikas per kurį yra įvykdoma programa ir laikas, kurį užtrunka surinkti ir perduoti informaciją. Programos vykdymo laiką nulemia programos kokybė, duomenų kiekis ir aparatūrinių resursų sparta. Daiktų interneto pakraščio skaičiavimų pritaikymas daugiausiai įtakos turi aparatūriniams resursams. Duomenų perdavimo laiką nulemia tinklo kokybė: vėlinimas, tinklo pralaidumas ir paketų praradimų dažnumas. Pakraščio sluoksnio daiktų internetas suteikia galimybę balansuoti tarp duomenų apdorojimo ir siuntimo laiko, perkeliant dalį ar net visus skaičiavimus į galinius daiktų interneto taškus. Taip yra mažinamas duomenų siuntimo laikas, skaičiavimo laiko sąskaita [10].



4 pav. Duomenų apdorojimo ir perdavimo spartos iliustracija [11]

Yra nemažai studijų ir metodų kaip padalinti skaičiavimo darbus tarp daiktų interneto pakraščio ir debesų kompiuterijos. Pavyzdžiui yra pasiūlyta metodų, kaip išlaikant tą pačią paslaugų kokybę galima pasiekti aukštesnį energijos našumą ir žemesnį vėlinimą perkeliant dalį skaičiavimų į tinklo pakraštį ir naudojant dinaminį įtampos valdymą [12].

Kita vertus didėjant išmanių prietaisų skaičiui – surenkamų duomenų kiekis auga eksponentiškai. Siųsti visus duomenis į debesų kompiuteriją jų neapdorojus tampa neracionalu. Tai lemia ilgus duomenų siuntimo laikus, reikalauja tinklo išteklių ir lemia dalies duomenų praradimą. Taigi, daiktų interneto pakraštyje duomenys suspaudžiami ir agreguojami prieš siunčiant į debesų kompiuteriją. Mažesni duomenų paketai sumažina ryšio išlaidas ir suteikia galimybę naudoti specialius daiktų internetui skirtus ryšio protokolus, kaip *Zigbee*, *Lora-WAN* ar *NB-IoT*.

Daiktų interneto pakraščio taškai atliekantys skaičiavimus padeda išspręsti tai, kad siunčiant mažus paketus išauga pridėtinių duomenų skaičius. Šie duomenys gali sudaryti iki pusę visos siunčiamos informacijos.

1.4.2. Duomenų saugojimas

Daiktų interneto pakraščio sluoksnis gali būti panaudotas siekiant sukurti paskirstytą duomenų saugojimo sistemą. Šis paskirstytas duomenų saugojimas taip pat sumažina siunčiamų duomenų kiekį ir leidžia saugoti duomenis taškuose, kurie yra arti duomenų generavimo ir vartojimo taškų.

Paskirstytos duomenų saugojimo sistemos turi privalumą, nes suteikia galimybę dubliuoti duomenis ir taip užtikrina duomenų prieinamumą, net sugedus vienam iš skaičiavimo taškų. Lokalus duomenų saugojimas taip pat užtikrina, kad laikui kritiški duomenys bus prieinami net sutrikus interneto tiekimui.

Ypač daug įtakos gali turėti trumpalaikių duomenų saugojimas daiktų interneto pakraštyje. Informacijos talpinimo metodikos gali sumažinti sistemos uždelsimą iki 95% [11]. Saugant daiktų interneto užklausas bei jas prognozuojant pakraščio taškuose ir gražinant informaciją galima smarkiai padidinti sistemos greitaveiką ir paslaugų kokybę.

1.4.3. Duomenų apdorojimas

Paskirstyti daiktų interneto pakraščio resursai leidžia ne tik apdoroti lokalius duomenis, tačiau ir paskirstyti šių duomenų apdorojimą per kelis daiktų interneto taškus. Šios paskirstytos sistemos gali užtikrinti greitą programos veikimą ir nuolat prieinamus resursus.

Sekančiuose poskyriuose yra analizuojami straipsniai, kuriuose tiriama daiktų interneto pakraščio efektyvumas ir greitaveika su *Linux* ir *FPGA* įterptinėmis sistemomis. Šie kriterijai vertinami renkant ir apdorojant įvairią informaciją, atliekant komunikacijos ir valdymo uždavinius bei taikant mašininio mokymosi algoritmus.

1.5. Duomenų apdorojimo greičio vertinimas

Tinkamos įrangos pasirinkimas yra fundamentali daiktų interneto architektūros dizaino dalis. Lyginant ir analizuojant, kurios technologijos yra pranašesnės bei vertinant, ar verta perkelti skaičiavimus į daiktų interneto pakraštį reikia atlikti lyginamąsias studijas. Siekiant įvertinti ir palyginti kompiuterių skaičiavimo spartą buvo sukurti standartizuoti testai. Pirmasis standartizuotas testas (angl. *benchmark*) - *Whetstone* buvo sukurti jau 1976 metais [13]. Ši programa buvo skirta testuoti kompiuterio gebėjimus atlikti skaičiavimus su slankiojančio kablelio skaičiais, tačiau neatliko praktiškos užduoties. Šiais laikais yra stengiamasi testuoti prietaisus pagal jų taikymo pobūdį.

2014 metais publikuota studija įvardina, kad dažnai yra kuriami *IoT* sprendimai naudojant *Linux* įterptines sistemas, tačiau neatsižvelgiant į prietaisų resursus, vertinant juos kaip juodą dėžę [14]. Tačiau net ta pačia operacine sistema paremtų prietaisų sparta ir parametrai gali smarkiai skirtis. Studija lygino tris plačiai paplitusius ir vartotojams bei tyrėjams prieinamus prietaisus: *BeagleBone*, *BeagleBone Black* ir *Raspberry Pi*. Šie prietaisai paprastai yra naudojami kaip mokymosi platformos ir yra pritaikomi kaip interneto vartai bei valdikliai surenkantys duomenis iš aplinkinių jutiklių, indikuojantys informaciją ar valdantys prietaisus. Buvo matuojama prietaisų sparta atliekant skaičiavimo, duomenų rašymo bei siuntimo operacijas. Studija įvardina, kad būtent komunikacija sudaro didžiąją dalį programos vykdymo laiko, o brangesnė ir greitesnė atmintis neturi didelės įtakos. Autoriai pabrėžia, kad būtent energijos suvartojimas, naudojimo paprastumas ir universalumas yra pagrindiniai kriterijai, kuriuos reiktų vertinti renkantis daiktų interneto vartų prietaisą.

Programuojama logika, kitaip vadinama tiesiog *FPGA* yra šiek tiek senesnė technologija, už įterptines *Linux* sistemas ir iš pradžių taikyta daugiausiai signalų apdorojimui. Tačiau *FPGA* technologija po truputį randa nišą daiktų interneto srityje. *FPGA* padeda efektyviai ir greitai apdoroti iš jutiklių surenkamą informaciją ir užtikriną laikui kritiškų programų funkcionalumą. *FPGA* itin naudinga technologija naudojant daiktų internetą sveikatos priežiūrai. Atlikta studija atrado, kad pritaikius programuojamą logiką *ECG* (*Elektrokardiogramos*) diagnozavimui buvo padidinta programos greitaveika iki 90% ir sumažintas bendras energijos suvartojimas iki 70% [15].

Toliau bus apžvelgta *Linux* įterptinių sistemų ir *FPGA* pritaikymo scenarijai bei gauti rezultatai taikant šias sistemas daiktų interneto pakraščio informacijos apdorojimui.

1.5.1. Programuojama logika

FPGA – tai integrinis grandynas, sukurtas taip, kad naudotojas galėtų jį konfigūruoti po pagaminimo, todėl jis vadinamas programuojama logika [16]. *FPGA* sudaro programuojamų loginių blokų masyvas ir perkonfigūruojamų jungčių hierarchija, leidžianti šiuos blokus sujungti tarpusavyje.

Loginiai blokai gali būti programuojami sudėtingoms kombinacinėms funkcijoms atlikti. Be to, *FPGA* turi atminties elementų, kurie gali būti paprasti *flip-flop* elementai arba sudėtingesni atminties blokai. Ši itin lanksti ir konfigūruojama architektūra leidžia įgyvendinti specifines aparatinės įrangos funkcijas nekuriant lusto nuo pat pradžių. Tai gerokai sutrumpina elektroninių sistemų kūrimo laiką ir sumažina išlaidas.

FPGA yra universalūs įrenginiai, kurie dėl savo programavimo galimybių ir efektyvumo naudojami įvairiose srityse ir ypač paplitę taikymo srityse, kuriose reikia greito apdorojimo ir didelio lygiagretumo. Dažniausiai šie įrenginiais naudojami skaitmeniniam signalų apdorojimui – *DSP* (angl. *Digital Signal Processing*), kur jie pagreitina garso ir vaizdo apdorojimo algoritmus. Taip pat *FPGA* taikoma telekomunikacijose, įgyvendinant įvairius protokolus arba apdorojant signalus, automobilių sistemose valdant jutiklius ir realizuojant saugos funkcijas realiuoju laiku. Be to, *FPGA* vis dažniau naudojama neuroniniuose tinkluose, skirtuose dirbtinio intelekto programų ir mašininio mokymosi skaičiavimams spartinti, užtikrinant reikiamą greitį ir pritaikomumą, reikalingą šioms daug skaičiavimų reikalaujančioms užduotims.

Programuojant *FPGA* paprastai naudojamos aparatūrinės įrangos aprašymo kalbos – *HDL* (angl. *Hardware Descriptive Language*), pavyzdžiui: *VHDL* ar *Verilog*. Šios kalbos leidžia aprašyti skaitmeninės grandinės struktūrą ir elgseną aukštesniu abstrakcijos lygmeniu nei tradicinis grandynų projektavimas. *HDL* aprašytas projektas kompiliuojamas, patalpinamas ir įkeliamas į *FPGA* architektūrą, aukšto lygio aprašymą paverčiant konfigūracijos failu, kuriame programuojami *FPGA* loginiai blokai ir jungtys. Jau 2009 metais aprašyta aukšto lygio sintezė – *HLS* (angl. *High Level Synthesis*) kaip itin potenciali priemonė išplečianti *FPGA* technologijos prieinamumą. *HLS* leidžia programuoti *FPGA* su *C* arba *C++* specialiais kompiliatoriais, galinčiais tam tikrus programinės įrangos algoritmus tiesiogiai versti į *FPGA* konfigūracijas.

FPGA paremtas aparatūrinės įrangos papildymas yra dažnai siūloma alternatyva daiktų interneto pakraščio sluoksniui dėl itin didelės spartos ir žemų energijos sąnaudų atliekant konkrečius, sudėtingus skaičiavimo uždavinius. *FPGA* sistemos dažnai naudojamos tokioms užduotims kaip kompiuterinei regai, duomenų šifravimui ar signalų filtravimui. Yra atlikta ne vienas tyrimas rodantis *FPGA* pranašumą naudojant kompiuterinę regą lyginant su debesų kompiuterija.

Atliktas tyrimas parodė, jog *FPGA* sistema veikia geriausiai lyginant tiek su programa veikiančia daiktų interneto pakraščio procesoriuje, tiek su debesyse atliekamais skaičiavimais [17]. Atliekant skaičių, objektų ir veidų atpažinimo uždavinius, programuojama logika įvykdė užduotis 1,7 (skaičių atpažinimas); 4,1 (objektų atpažinimas) ir 1,6 (veidų atpažinimas) kartus greičiau, negu tiesiog perkeliant duomenų apdorojimą į debesų kompiuteriją. Įdomu pastebėti, kad pakraštyje veikiantis *Intel Core i7-7700* 3,10 GHz procesorius įvykdė užduotis greičiau už debesyse veikiantį *Intel Xeon E5-2650* procesorių. *FPGA* sprendimai gali užtikrinti daug greitesnį vykdymo laiką, 22 % ir 35 % sumažindamas objektų ir veidų atpažinimo energijos sąnaudas, palyginti su debesų kompiuteriją naudojančiais sprendimais. Tačiau tyrime pripažįstama, kad šiuos rezultatus sunku pasiekti dėl sudėtingo *FPGA* programavimo pobūdžio.

FPGA sistema taip pat gali suteikti žymiai geresnius rezultatus, nei *Arduino* sistema už panašią kainą [18]. Šiame tyrime buvo lyginami *DE10-lite FPGA* ir *Arduino Mega*, naudojami kaip orų duomenų rinkimo stotys. Stotis turi surinkti duomenis iš 4 jutiklių, sudaryti orų prognozę, pagrįstą sprendimų medžio skaičiavimo algoritmu, ir užšifruoti duomenis prieš siųsdama juos į serverį. *FPGA* sistema

turi apie 5 kartus trumpesnį duomenų apdorojimo laiką nei *Arduino* sistema ir panaudojo tik dalį savo išteklių. Tyrimas taip pat parodė, kad naudojant *FPGA* pagrįstą krašto sluoksnį, siunčiamų duomenų kiekis sumažėjo daugiau nei 3,5 karto. Straipsnio autoriai taip pabrėžė, kad *FPGA* išnaudoja vos 20% resursų, todėl lieka vietos plėsti ir tobulinti egzistuojančia oro stoties sistemą. Nors šis tyrimas analizavo ir lygino tik programos veikimo laiką ir siunčiamų duomenų kiekį, tačiau galime teigti, kad buvo atliktas ir ekonominis vertinimas naudojant panašios kainos produktus: *DE10-lite* ir *Arduino Mega*.

Itin aktualius šiam darbui straipsnis buvo atliktas apjungiant visas tris technologijas: debesų kompiuteriją, įterptinę *Linux* sistemą ir *FPGA* matricą [19]. Šio tyrimo metu buvo sukurta realiu laiku konfigūruojama *FPGA* signalų filtravimo sistema. Tyrimas parodė, kad *FPGA* pagrįsta sistema, naudojanti „*Xilinx XC6SLX4*“, pranoko debesų pagrindu veikiančią sistemą ir įterptinę *Raspberry Pi* sistemą.

Šiais laikais vis svarbesne tampa informacijos apsauga. Kompanijos turi pareigą institucijoms ir vartotojams apsaugoti duomenis ir pasirūpinti atitinkamu duomenų kodavimu. *FPGA* technologija puikiai tinka tiek kodavimo, tiek dekodavimo užduotims. Straipsnio autoriai realizavo plačiai taikomus kriptografijos algoritmus, tokius kaip *DES, AES, RSA* ir *SHA256* [20]. Tyrimo metu buvo naudota *Zedboard* plokštė, kuri turi dviejų branduolių *ARM* procesorių ir *Zynq FPGA* matricą. Buvo lyginama programos sparta ir elektros sąnaudos realizuojant programą tik su *ARM* procesoriais ir naudojant *ARM* procesorius kartu su *FPGA IP* šerdimis. Gauti rezultatai nustebino: pritaikius *FPGA, RSA* kodavimo greitis pagreitėjo 71 kartą, o dekodavimo greitis – net 2983 kartus. Atitinkamai tai lėmė 6,5 karto mažesnes energijos sąnaudas koduojant ir 4033 kartus mažesnes energijos sąnaudos dekoduojant. Panašūs rezultatai buvo gauti ir naudojant *SHA256 hash* funkcijas: kodavimo sparta padidėjo 6,6 kartus, o energijos sąnaudos sumažėjo net 4,6 kartus.

1.5.2. *Linux* įterptinės sistemos

Šiame poskyryje bus nagrinėjamas konkrečios *Linux* įterptinės sistemos pritaikymas ir parametrai: *Raspberry Pi*. Ši įranga buvo pasirinkta būtent dėl jos paprastumo, populiarumo ir skaičiavimo resursų už konkurencingą kainą.

Siekiant įvertint *Raspberry Pi* informacijos apdorojimo pajėgumus *IoT edge* sluoksnyje naudojant dirbtinius neuroninius tinklus ir gilųjį mokymąsi buvo atliktas tyrimas [21]. Šio tyrimo tikslas buvo įvertinti ar *Raspberry Pi* gali būti taikoma realaus laiko vaizdo apdorojimo tikslams ir kokia yra tiriamų sistemų sparta, kadrų per sekundę atžvilgiu. *Raspberry Pi* gali būti pritaikyta kartu su kitais įrenginiais, kurie pagreitina skaičiavimo pajėgumus. Atliktame tyrime buvo naudojamas *Intel Movidius Neural Compute Stick* (*NCS*) kaip papildomas aparatūrinis skaičiavimų greitintuvas, naudojant gilųjį mokymą. Tyrimas buvo atliktas lyginant *Raspberry Pi 3 model B* su *Linux (Ubuntu)* asmeniniu kompiuteriu. Gauti rezultatai parodė, kad naudojant tik *Raspberry Pi* procesorių su *Google's MobileNets* konvoliuciniais neuroniniais tinklais – CNN (angl. *Convolutional Neral Networks*) vaizdo apdorojimas pasiekdavo vos iki 12 kadrų per sekundę greitį, tačiau su *NCS* – iki 70 kadrų per sekundę. Lyginant kito eksperimento rezultatus buvo pastebėta, kad naudojant 1 *NCS* akseleratorių *Raspberry Pi 3* pasiekia 5,7, o asmeninis kompiuteris – 9,3 kadrų per sekundę apdorojimo greitį. Šiame tyrime buvo pastebėta, kad didžiausias *Raspberry Pi 3* trūkumas buvo *USB 2.0* jungtis ribojanti akseleratoriaus greitį. Tyrimas įrodė, kad įterptinės sistemos gali būti naudojamos

mašininio mokymo pritaikymui pakraščio sluoksnyje realaus laiko uždaviniams, tačiau reikalauja papildomų įterptinių sistemų skaičiavimų pagreitinimui.

5 pav. *Intel Movidius Neural Compute Stick (NCS)* [22]

Kitas tyrimas, kuriame buvo naudojamas *Raspberry Pi 3 B* ir lyginami įvairūs objektų atpažinimo algoritmai [23], parodė, kad geriausi rezultatai buvo pasiekti naudojant smulkų *CNN* algoritmą, kuris naudoja palyginti mažiau skaičiavimo resursų ir atminties nei kiti algoritmai, tokie kaip *SSD GoogleNet* ar *Haar Cascaded* algoritmai. Tyrimo autoriai sugebėjo pasiekti 1,8 kadrų per sekundę vidutinį vaizdo apdorojimo greitį atliekant žmonių aptikimo kadre uždavinį.

6 pav. Smulkaus *CNN* žmonių atpažinimo rezultatai [23]

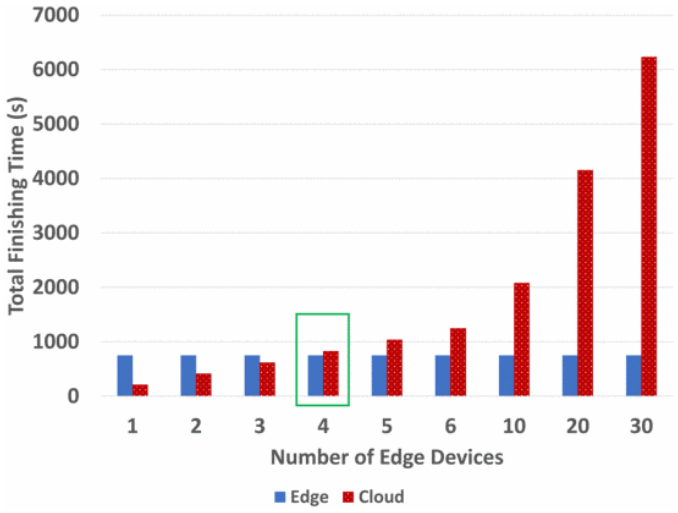
Abu tyrimai pateikia neginčijamų įrodymų ir metodų, kaip pasiekti objektų atpažinimą realiuoju laiku naudojant *Raspberry Pi*.

Raspberry Pi gali būti naudojamas kartu su kitais ištekliais, norint sukurti *IoT edge* sluoksnį įprastai centralizuotame tinkle. Atliktame tyrime, *Raspberry Pi* buvo naudojamas kaip savarankiškai organizuotas užduočių planavimo mechanizmas valdantis saldainių pakavimo mašinas [24]. Šio tyrimo tikslas buvo įvertinti konkrečią ekonominę naudą pritaikius daiktų interneto pakraščio principus. Buvo nustatyta, kad savarankiškai organizuotos mašinos pasiekia iki 22,8 % greitesnį darbą, kai dirbo su dideliais užsakymų kiekiais. Taip pat buvo nustatyta, kad *IoT* pakraščio tinklas sumažino siunčiamų duomenų kiekį 3 kartus.

1.5.3. Kiti *IoT* pakraščio vertinimo metodai

Gana paprastas daiktų interneto pakraščio sistemos lyginimo ir analizės metodas yra naudojant iš anksto apmokytus konvoliucinius neuroninius tinklus objektų atpažinimui. Taip buvo pasielgta kelete iš nagrinėtų tyrimų [17, 21, 23]. Tačiau tai be abejo yra lengvesnė užduotis *FPGA* matricoms, kurios lengvai atlieka iš karto numatytus skaičiavimus ir yra nesąžininga *Linux* įterptinių sistemų atžvilgiui.

Viename tyrime, lygindami debesų kompiuteriją ir daiktų interneto pakraščio sluoksnio architektūrą su keliais *Raspberry Pi*, tyrėjai naudojo *Smith-Waterman* genų sekos derinimo algoritmą ir *Tesseract* metodą optiniam simbolių atpažinimui [25]. Straipsnyje aprašomas sistemų palyginimo metodas pagal laiką, kurio reikia duomenims apdoroti ir perduoti į serverį. Tyrimo metu buvo matuojamas bendras užduočių užbaigimo laikas, kai iteracijų skaičius didinamas su kiekvienu pridėtu krašto sluoksnio įrenginiu. Tyrėjai nustatė, kad didinant duomenis proporcingai krašto įrenginių skaičiui, bendras laikas krašto skaičiavimo sistemose išlieka pastovus. Nors ši metodika neatskleidžia aiškaus debesų kompiuterijos ir tinklo paribio kompiuterijos palyginimo, nes sukurta sistema labiau primena lygiagrečios ir nuoseklios programos palyginimą, joje siūlomas įdomūs tinklo paribio kompiuterijos sistemos našumo matavimo metodai: *Smith-Waterman* ir *Tesseract* objektų atpažinimo metodai.



7 pav. Daiktų interneto ir debesų kompiuterijos programos veikimo laikas taikant *Tesseract* metodą [25]

Norėdami objektyviai išbandyti ir palyginti tinklo paribio kompiuterijos sistemas, taip pat galima naudoti jau sukurtus įrankius, tokius kaip *EdgeBench* [26]. *EdgeBench* programinė įranga matuoja skaičiavimo laiką, siuntimo laiką, laiką kurį užtrunka perduoti duomenis nuo jų sukūrimo iki apdorojimo, naudingos informacijos dalį, procesoriaus ir atminties išnaudojimą. *EdgeBench* duomenų apdorojimui palyginti naudoja tris metodus: kalbos vertimą į tekstą, objektų atpažinimą, skaliarinį jutiklį (jutiklio emuliaciją). Pagrindinė straipsnio ir programinės įrangos idėja yra palyginti skirtingas debesų paslaugas, skirtas IoT pakraščio diegimui, kaip pavyzdžiui *AWS Greengrass* ir *Microsoft Azure IoT Edge*. Išmatuoti parametrai taip pat gali būti naudojami norint palyginti skirtingas daikt interneto pakraščio skaičiavimo platformas. *EdgeBench* programinė įranga suteikia patogų, standartišką ir įvairias užduotis realizuojantį metodą leidžiantį palyginti pakraščio sluoksnių sistemas.

Taip pat yra sukurta įrankių skirtų įvertinti daiktų interneto struktūros efektyvumą ir jį optimizuoti pateikiant optimalius parametrus. Pauliaus Tervydžio (ir kitų autorių) parašytame straipsnyje yra pristatomas sukurtas *CloudEdgeAssetOptimizer* įrankis skirtas debesų kompiuterijos ir daiktų

interneto pakraščio tinklui optimizuoti [27]. Ši programa skirta tinklo išteklių optimizavimui pagal įvairius parametrus, kaip pavyzdžiui duomenų apdorojimo uždelsimas, baterijos talpa ir ekonomiškumas. Programa naudoja laukimo eilių teorijos principus siekiant modeliuoti ir analizuoti darbų eiles įvairiuose tinklo taškuose. Tai suteikia įžvalgų apie duomenų apdorojimo ir laukimo laiką, įrenginių baterijos iškrovimą ir serverių apkrovas. Šis įrankis gali atlikti tiek rankinį, tiek automatinį parametrų optimizavimą.

Naudojant *CloudEdgeAssetOptimizer* galima įvertinti maksimalų duomenų apdorojimo užklausų atvykimo dažnį, kuri sistema gali apdoroti neviršydama nurodyto vidutinio laukimo laiko. Programa taip pat gali būti naudojama siekiant optimizuoti daiktų interneto prietaisų ir serverių skaičių. Šie parametrai yra parenkamai taip, kad duomenų apdorojimo laikas neviršytų kritinės ribos, taip pat atsižvelgiant į prietaisų kainą ir baterijos talpą. Ši funkcija leidžia surasti ekonomiškai optimalų sprendimą, tenkinantį paslaugų kokybę. Galiausiai, programinė įranga leidžia modeliuoti tinklo topologiją. Keičiant tinklo mazgų ir jų ryšių skaičių, programa padeda suprasti tinklo architektūros įtaką efektyvumui, laukimo laikui ir kainai.

2. Tyrimo metodika

Šio tyrimo tikslas yra realizuoti įterptinėms sistemoms būdingą duomenų apdorojimo modelį su programuojama logika ir įvertinti programuojamos logikos greitaveiką bei resursų efektyvumą. Tai pasiekiama realizuojant įterptinėms sistemoms būdingus skaičiavimus su programuojama logika, kitomis įterptinėmis sistemomis ir asmeniniu kompiuteriu. Skaičiavimų modelis yra optimizuojamas ir tobulinamas, kol pasiekiamas tenkinantis sistemos optimizavimo lygis. Programuojamos logikos skaičiavimų sparta analizuojama ir lyginama su kitomis technologijomis. Be skaičiavimų spartos taip pat atliekama resursų ir jų sąnaudų analizė.

Šiame skyriuje yra apžvelgiami tyrimo uždaviniai bei priimti sprendimai realizuojant ir testuojant programuojamos logikos greitaveiką ir resursų efektyvumą. Taip pat šiame skyriuje yra apžvelgiamas pasirinktas tyrimo taikymo pavyzdys ir skaičiavimai. Apžvelgiama pasirinktos priemonės realizuojant tyrimo modelį ir etapai, kuriais buvo atliekama modelio įgyvendinimas.

2.1. Tyrimo uždavinių sprendimas

Projekto metu buvo apibrėžti šie tyrimo uždaviniai:
1) realizuoti tipinę pakraščio skaičiavimų spartinimo įterptinę sistemą, panaudojant programuojamą logiką ir įterptinį mikrovaldiklį;
2) optimizuoti programuojamos logikos sistemos realizaciją, atsižvelgiant skaičiavimų greitaveiką ir resursų sąnaudas;
3) įvertinti ir palyginti programuojamos logikos ir įterptinių sistemų mikrovaldiklių greitaveiką;
4) įvertinti programuojamos logikos privalumus ir trūkumus pagal resursų sąnaudas ir energetinį efektyvumą.

Pakraščio sluoksnio realizavimo scenarijų analizė buvo atlikta literatūros analizės metu, **1.4** skyriuje. Apžvelgus literatūros analizę galima teigti, kad daiktų interneto pakraščio taikymai yra įvairūs ir apima nuo skaitmeninių signalų analizės, duomenų šifravimo iki mašininio mokymosi algoritmų. Pasirenkant taikymo pavyzdį buvo ieškoma gerai ištirtų sistemų. Taip buvo pasirinkta, nes šio tyrimo tikslas – realizuoti egzistuojančią sistemą su programuojama logika. Šiam tikslui puikiai tinka jau ištirtas ir žemės ūkyje pritaikytas modelis aprašytas **2.2** skyriuje. Šie skaičiavimai apima skaitmeninių signalų analizę, Furjė transformaciją bei iteracinio mašininio mokymo algoritmų pritaikymą. Taigi šis modelis apima platų daiktų interneto pakraštyje naudojamų metodų spektrą ir atvaizduoja realią aplinką.

Tyrimo uždaviniai apima pasirinkto taikymo pavyzdžio realizavimą programuojamoje logikoje. Pirmiausia pasirinktas tymo pavyzdys yra realizuojamas programuojamoje logikoje, nes ji turi daugiau apribojimų ir sudėtingesnį projektavimo ciklą negu programavimas su įterptinėmis *Linux* sistemomis. Projektavimo metu nėra pasirenkama konkretaus komponento, nes užbaigus *FPGA* programos projektavimo ir optimizavimo etapą galima pasirinkti tinkamiausią, poreikius atitinkantį ir proporcingai kainuojantį komponentą.

Suprojektuoto programuojamo logikos projekto optimizavimas prieš testuojant ir lyginant sukurtą programą su kitomis sistemomis yra vienas iš projekto uždavinių. Programos optimizavimas yra būtinas siekiant užtikrinti optimalią *FPGA* projekto struktūrą. *FPGA* greitaveika ir resursų našumas itin priklauso nuo pasirinktos architektūros, kuri aprašyta **3.4** skyriuje. Tik atlikus reikalingus

optimizavimo veiksmus ir galimybių analizę yra pereinama prie tyrimo pavyzdžio realizavimo kitose sistemose.

Siekiant atlikti tyrimą ir palyginti programuojamos logikos greitaveiką buvo naudojami Error! R eference source not found. skyriuje aprašyti įrankiai. Tyrimui buvo naudojama *IP* šerdies sintezavimo skaičiavimų duomenys, nes sintezuojama *FPGA* architektūra yra deterministinio laiko ir teikia pakankamai tikslius rezultatus. **4** skyriuje yra aprašomi tyrimo rezultatai, kur programuojama logika yra lyginama su tipine įterptine sistema ir nešiojamuoju kompiuteriu. *Raspberry Pi 4B* buvo pasirinkta kaip tipinė *Linux* įterptinė sistema gebanti atlikti skaičiavimams imlius uždavinius daiktų interneto pakraštyje. Tyrimui buvo pasirinkta naudoti nešiojamąjį kompiuterį dviem scenarijais: kai kompiuteris veikia maksimaliu pajėgumu ir yra įjungtas maitinimo šaltinis bei, kai kompiuteris veikia galios taupymo režimu ir yra atjungtas nuo energijos šaltinio. Šie scenarijai reprezentuoja dvi skirtingas pritaikymo paskirtis. Maksimaliu greičiu veikiantis kompiuteris teikia rezultatus reprezentuojančius debesų kompiuterijoje atliekamus skaičiavimus. Kita vertus nešiojamas kompiuteris, veikiantis galios taupymo režimu, reprezentuoja daiktų interneto pakraštyje laikomą nešiojamą kompiuterį, skirtą duomenų surinkimui ir analizei. Šių keturių scenarijų palyginimai suteikia visapusišką programuojamos logikos palyginimą su alternatyviomis skaičiavimo technologijomis ir sprendimais.

Šio tyrimo metu buvo analizuojami ir lyginami su kitomis technologijomis šie programuojamos logikos apsektai:

- greitaveika;
- resursų panaudojimo efektyvumas;
- technologijų energetinis našumas.

Greitaveika gali būti vertinama lyginant instrukcijų skaičių arba laiką, kuris yra reikalingas atlikti skaičiavimams. Instrukcijų skaičius yra patogus vertinti skirtingų architektūrų *FPGA* greitaveiką. Tačiau lyginant procesorių ir programuojamos logikos greitaveiką, instrukcijų skaičių naudoti yra neteisinga, nes programuojama logika ir procesoriai veikai itin skirtingu taktiniu dažniu. Todėl lyginant skirtingų technologijų greitaveiką yra naudojami profiliavimo įrankiai siekiant įvertinti ir palyginti tikslų skaičiavimo laiką.

Vertinant procesorių greitaveiką taip pat svarbu atsižvelgti į sukurtos programos architektūrą. Nuo programos architektūros priklauso ar vertinama procesoriaus vieno branduolio sparta ar visų procesoriaus branduolių sparta. Vertinant visų procesoriaus branduolių spartą galima geriau įvertinti sistemos greitaveiką su konkrečiu procesoriumi. Tačiau šie rezultatai gaunami specifiniai būtent tam procesoriui. Kita vertus vieno procesoriaus branduolio

Vertinant greitaveiką taip pat svarbu atsižvelgti ir į procesoriams skirtos programos architektūrą. Pagal tai kokia yra sukurta architektūra priklauso, kiek programa panaudoja procesoriaus branduolių. Siekiant įvertinti konkretaus procesoriaus greitaveiką puikiai tinka naudoti architektūrą panaudojančią visus procesoriaus branduolius. Tačiau siekiant išmatuoti bendrą procesorių greitaveiką ir nekeisti programos struktūros šeimai tyrime yra naudojama vienam branduoliui pritaikytą programą.

Kitas svarbus aspektas, kuris yra lyginamas šio tyrimo metu yra resursų panaudojimo efektyvumas. Tyrime yra lyginamos panaudojamų technologijų kainos ir greitaveikos palyginimas. Taip pat yra vertinama, kiek efektyviai yra panaudojami esami resursai, kokia jų dalis yra panaudojama.

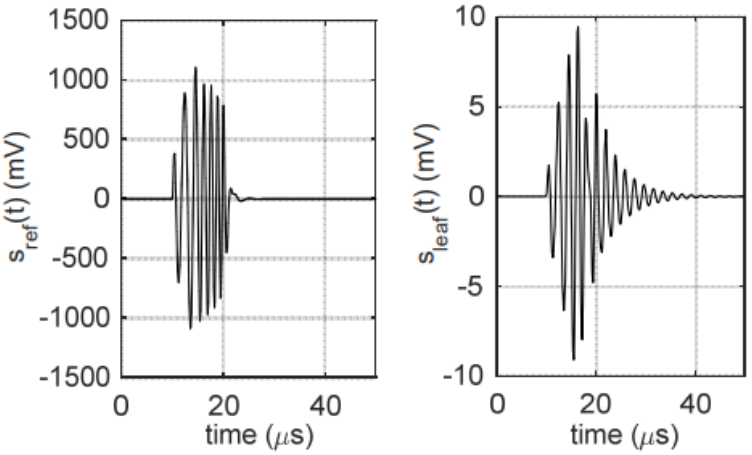
Programuojamos logikos privalumas – lygiagretinti ir spartinti skaičiavimus panaudojant konvejerines struktūras. Tačiau procesoriai naudoja kitus principus – naudoja kelis procesoriaus branduolius atlikti skaičiavimams lygiagrečiai arba vykdyti kelis procesus vienu metu. Taigi programuojamos logikos privalumas – išnaudoti maksimalų resursų kiekį vieno proceso greitaveikos optimizavimui, yra lyginamas su procesoriaus privalumu – vykdyti kelis procesus lygiagrečiai.

Galiausiai šiame tyrime yra vertinama technologijų energetinis efektyvumas ir energijos sąnaudų įtaka finansiniam sprendimų tinkamumui, taikant programuojamą logiką duomenų apdorojimui. Siekiant palyginti įrangos energetinį efektyvumą yra naudojami programuojamos logikos modeliavimo duomenys ir procesorių specifikacijos dokumentacija.

2.2. Taikymo pavyzdys

Analizuojant daiktų interneto pakraščio efektyvumą ir lyginant rezultatus yra svarbu turėti realų tyrimo objektą ir pritaikymo sritį. Šiam tikslui buvo pasirinktas inversinio uždavinio, naudojamo ultragarsinėje spektroskopijoje, sprendimas [28]. Šios uždavinio sprendimas apima: dalelių spiečiaus optimizavimo algoritmo pritaikymo, ultragarsinių bangų sklidimo modelio parametrų nustatymo, signalų apdorojimo naudojant kompleksinius skaičius bei tiesioginės ir atvirkštinės Furjė transformacijos pritaikymo. Taigi šis pritaikymo atvejis atspindi platų spektrą skaičiavimo algoritmų.

L. Svilainio ir Ž. Nakučio (bei kitų) straipsnyje yra aprašomas neinvazinis augalų lapų tyrimo metodas, naudojant ultragarsinę rezonansinę spektrometriją siekiant nustatyti Brekhovskikh modelio parametrus [28, 29]. Pirmiausia duomenys yra surenkami išmatuojant signalo sklidimą tarp ultragarsinių keitiklių išmatuojant atskaitinį signalą s_{ref} be augalo lapo ir signalą s_{leaf} su įdėtu lapu tarp dviejų ultragarsinių keitiklių. Gautas matavimų rezultatas yra pateikiamas **8** pav.



8 pav. Ultragarsinių matavimų rezultatai: atskaitos signalas s_{ref} ir s_{leaf} [29]

Ultragarsinio signalo perdavimo per augalo lapą atsakas yra išreiškiamas šių signalų spektrų santykiu kampinių dažnių srityje:

$$T(\omega) = \frac{S(\omega)}{R(\omega) * e^{j\omega\frac{h}{c_1}}}, \quad (1)$$

R(ω) – atskaitinio signalo s_{ref} spektras, S(ω) – signalo su lapu s_{leaf} spektras ir $e^{j\omega h/c}$ – eksponentė skirta įvertinti lapo išstumtą orą.

Ultragarsinio signalo perdavimas per augalą taip gali būti aprašytas šiomis funkcijomis:

$$T(\omega) = \frac{-Z_1 Z_2}{-2Z_1 Z_2 \cos(k'h) + j(Z_1^2 + Z_2^2)\sin(k'h)}, \quad (2)$$

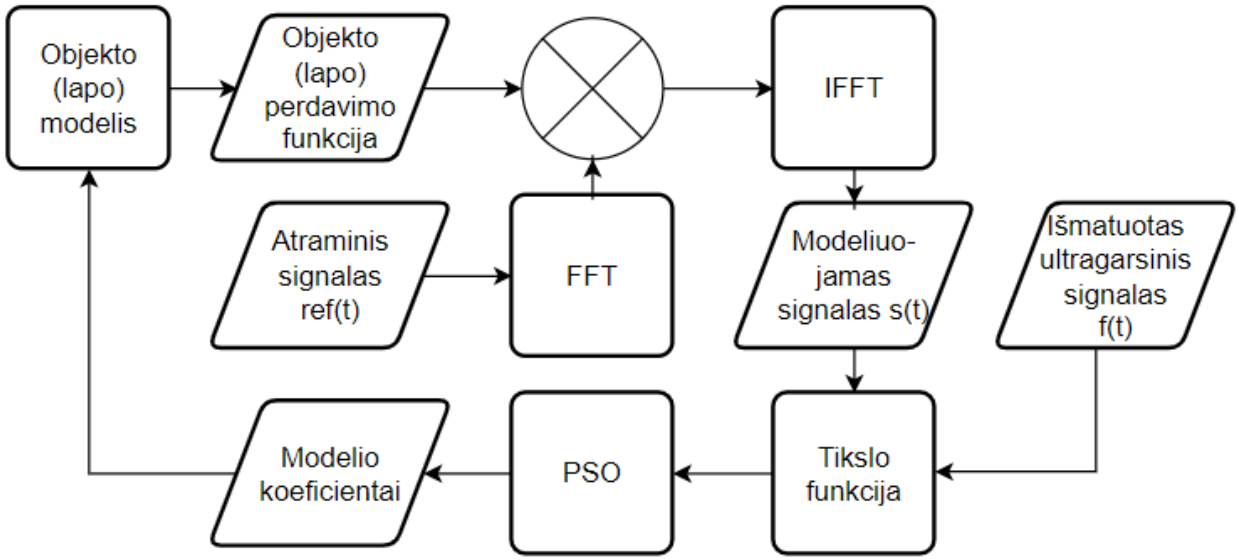
$$k' = \frac{\omega}{c_2} - j\alpha, \quad (3)$$

$$\alpha = \alpha_0 * \left(\frac{f}{f_0}\right)^n, \quad (4)$$

$$Z_{1,2} = c_{1,2} * \rho_{1,2}, \quad (5)$$

$Z_{1,2}$ yra akustinis oro ir lapo kompleksinė akustinė varža, h – lapo storis, k‘ – kompleksinis bangos skaičius, c_2 – ultragarso greitis lapo ertme, α – akustinis lapo slopinimo koeficientas, f_0 - normalizavimo dažnis, kuris buvo naudojamas kaip ultragarsinio keitiklio rezonansinis dažnis ir $\rho_{1,2}$ – atitinkamai aplinkos ir lapo tankis.

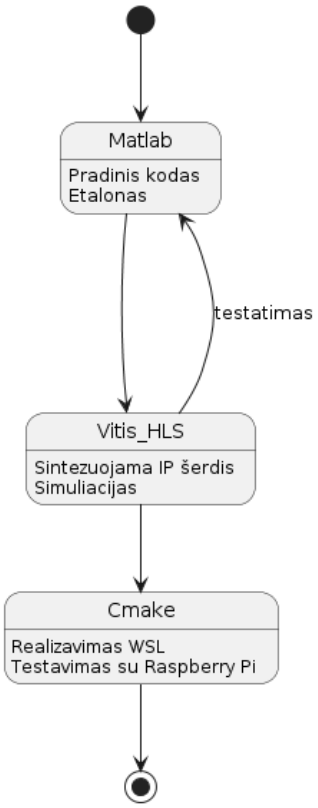
Išsprendžiant 1–5 lygčių sistemą ir pritaikant *PSO* algoritmą įmanoma įvertinti augalo lapo parametrus ρ_2, c_2, h, α_0, n, f_0 iš kurių galima apskaičiuoti kitus parametrus įvertinančius lapo ir augalo būklę. Šio tyrimo objekto apibendrintas algoritmas yra pateikiamas blokine diagrama **9** pav. Šio tyrimo objektas yra būtent šių skaičiavimų įvertinimas naudojant įterptines sistemas.



9 pav. Augalo parametrų įvertinimo algoritmo blokinė schema

2.3. Duomenų apdorojimo algoritmo realizavimas

Atliekant tyrimą ir kuriant programas buvo naudojamasi šiais įrankiais: *Matlab, Vitis HLS, Linux* posisteme *Windows* operacinei sistemai – *WSL* (angl. W*indows Subsystem for Linux*), *Cmake* ir *Raspberry Pi*. Pirmiausia buvo sukurtas prototipas su *Matlab*, toliau buvo pereita prie projekto kūrimo ir testavimo su *Vitis HLS* ir galiausiai projektas buvo pritaikytas *Linux* operacinei sistemai naudojant *WSL*, *Cmake* projektą. Toliau šiame skyriuje bus apžvelgta kiekviena iš šių technologijų ir kodėl jos buvo pasirinktos bei kaip pritaikytos.



10 pav. Tyrimo sistemos struktūra, eigos seka

Tyrimas buvo pradėtas nuo *Matlab* kodo, kuris buvo parašytas L. Svilainio ruošiant straipsnį apie naudojamą tyrimo pavyzdį [**29**]. *Matlab* turi privalumų: lengvai apdorojami signalai ir galima naudoti integruotas bibliotekas atlikti Furjė transformaciją bei veiksmus su kompleksiniais skaičiais. *Matlab* tinka duomenims analizuoti taikant mašininį mokymą, nes turi plačia algoritmų ir mašininio mokymo biblioteką. Taip pat didelis *Matlab* privalumas – nesudėtingas duomenų atvaizdavimas. Pradinis programos kodas buvo modifikuojamas ir pritaikomas perkelti į *Vitis HLS* projektą. Projekto kodas buvo tobulinamas gerinant skaitomumą, derinant ir taisant klaidas. Pradinės *Matlab* programos rezultatai buvo laikomi etalonu kituose etapuose.

Antras tyrimo etapas – sukurti *FPGA* programą, kuri atliktų reikalingus skaičiavimus ir apdorotų duomenis. Šio tyrimo pavyzdinis pritaikymas reikalauja gana sudėtingų skaičiavimų – naudojami trupmeniniai ir kompleksiniai skaičiai, *FFT* ir *IFFT* transformacijos. Taigi *Verilog* ir *VHDL* programavimo kalbos yra atmetamos, nes būtų itin sudėtinga realizuoti sudėtingas operacijas, ypač su paminėtais skaičių tipais. Šiam tyrimui puikiai tinka *Vitis HLS* įrankis, nes leidžia naudoti slankaus ir fiksuoto kablelio operacijas, naudoti kompleksinius skaičius ir turi platų bibliotekų palaikymą. Naudojant *Vitis HLS* yra sukuriamas projektas, kuris iš *C++* kalbos sintezuoja *IP* šerdis, kurios gali būti integruojamos į *FPGA* komponentus.

Naudojant *Vitis HLS* galiam nesudėtingai derinti projektą lygiagrečiai rašant programą ir testavimą. Tačiau *Vitis HLS* leidžia lyginti ir spausdinti vertes, tačiau negalima jų atvaizduoti. Naudojant skaitmeninių signalų apdorojimą yra itin svarbu įvertinti signalų kokybę vizualiai. Šiam tikslui buvo pasirinkta integruoti *Vitis HLS* rezultatus su *Matlab*. Metodologijos principas – atspausdinti kintamųjų vertes taip, kad juos būtų nesudėtinga nuskaityti su *Matlab* ir lyginti. Šis lygiagretus *Vitis HLS* ir *Matlab* derinimo metodas leidžia vizualiai vertinti skaičiavimų tinkamumą.

Trečiame tyrimo etape sukurtą *FPGA* projektą reikia pritaikyti įterptinės sistemos procesoriui. Pravartu apgalvoti iš anksto kaip pritaikyti tą patį projektą serveriui, nešiojamam kompiuteriui ir įterptinei sistemai. Pats paprasčiausias būdas tai pasiekti – naudoti platformą leidžiančią programuoti ir kompiliuoti *C++* kodą nepriklausomai nuo platformos. Šiuos reikalavimus patenkina įrankis *Cmake*, kuriuo galima sukompiliuoti kodą tiek įterptinei *Raspberry PI* sistemai, tiek paleisti programą serveryje ar nešiojamame kompiuteryje. Taigi vienoda įrankių grandinė padeda užtikrinti programos suderinamumą su įvairiomis programavimo aplinkomis. Taigi šiame etape sukurtas *FPGA* projektas yra perkeliamas į *Cmake* projektą bei realizuojamas veikti tiek su *Raspberry PI*, tiek su *WSL* nešiojamame kompiuteryje.

3. Taikymo pavyzdžio realizacija

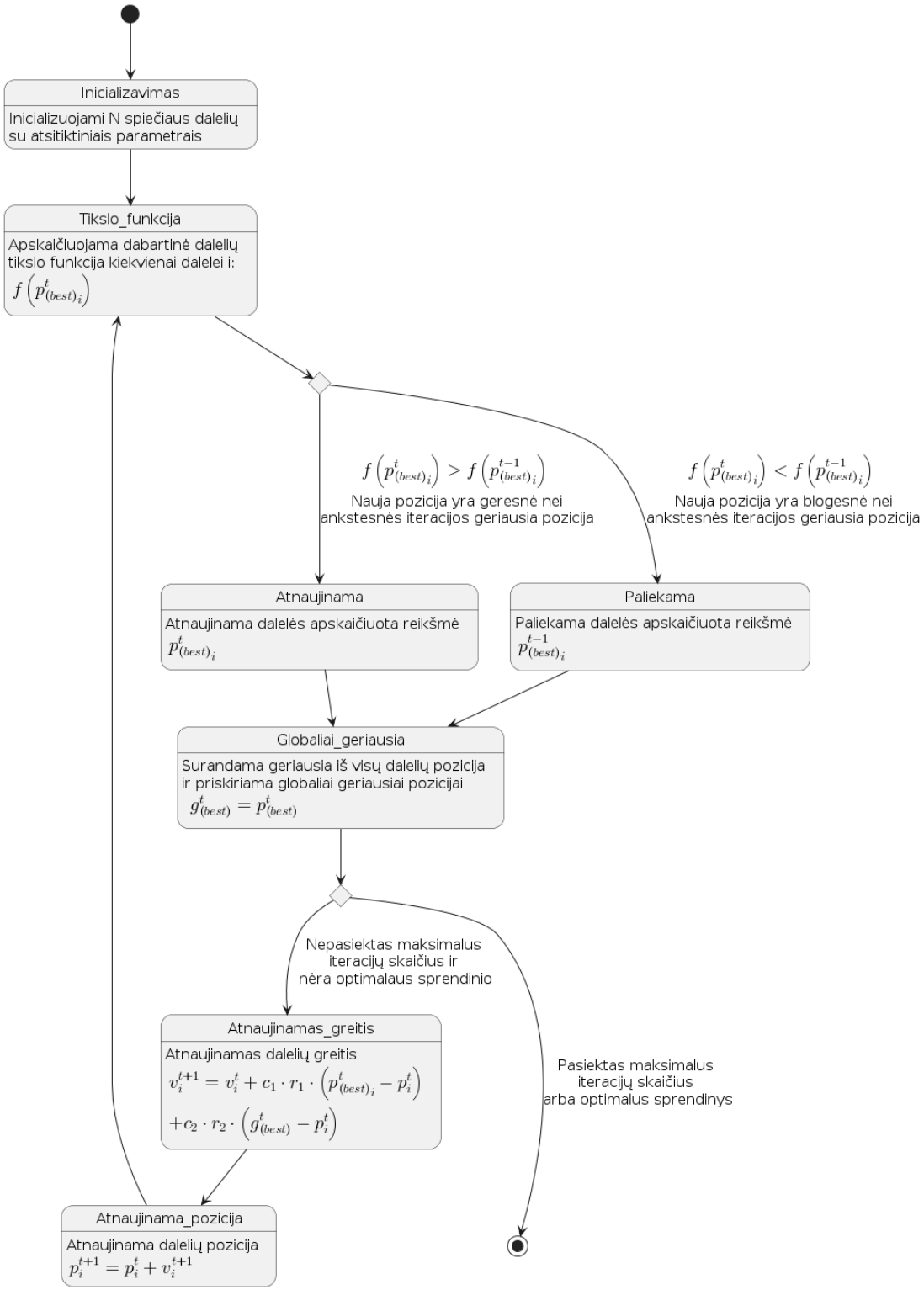
Šiame skyriuje apžvelgiami etapai, kurių imtasi realizuojant tyrimo taikymo pavyzdį (angl. *use case*) bei kokios optimizavimo priemonės yra reikalingos siekiant suprogramuoti sistemą pritaikytą *FPGA* matricoms ir įterptinėms *Linux* sistemoms.

Pirmajame modelio realizavimo etape svarbu pasirinkti, kokia programavimo kalba ir sistema bus realizuota sistema. *FPGA* teikia keletą pasirinkimų, kokiais metodais yra programuojama matrica ir apdorojami duomenys. „*Xilinx*“ įmonės sukurtos matricos yra programuojamos *Verilog*, *VHDL* programavimo kalbomis ir *HLS* metodais. *Verilog* ir *VHDL* priemonės suteikia aukštesnį optimizavimo laipsnį, tačiau smarkiai apriboja galimybes dėl žemo lygio kalbos funkcionalumo ir įrankių trūkumo. Įrankiai, leidžiantys atlikti sudėtingus veiksmus, paprastai suteikia galimybę kurti sudėtingas sistemas. Todėl kuriant *PSO* algoritmą ir realizuojant *FFT* skaičiavimus svarbu gebėti nesudėtingai aprašyti sudėtingus skaičiavimus. Šiam tikslui puikiai tinka aukšto lygio sintezė – *HLS*, gebanti sintezuoti *FPGA* grandymus iš *C/C++* kalbos instrukcijų.

3.1. *PSO* algoritmo realizavimas panaudojant programuojamą logiką

Dalelių spiečiaus optimizavimas (*PSO*) yra stochastinis algoritmas, įkvėptas socialinio gyvūnų elgesio, pavyzdžiui, žuvų ar paukščių grupės. Šis metodas vertina galimus problemos sprendimus kaip daleles, judančias problemos sprendimo erdvėje. Metodas derina istoriškai geriausias vietas, problemos sprendinius ir dabartines pozicijas su kitų spiečiaus narių padėtimis, kad nustatytų jų tolesnius judesius. Per daugelį iteracijų, spiečius kolektyviai artėja prie optimalaus sprendimo. *PSO* privalumas yra lengvas pritaikymas įvairiose sistemose ir nedidelis parametrų skaičius. Tačiau algoritmas taip pat turi trūkumų dirbant su itin dideliais duomenų rinkiniais ir gali užstrigti ties lokaliais minimumais [30].

Algoritmo realizavimo blokinė schema yra pateikta **11** pav. Algoritmas prasideda spiečiaus dalelių inicializavimu suteikiant joms atsitiktines koordinates ir greičio vektorių. Toliau yra vykdomas iteracinis algoritmas. Iteracijos pradžioje yra įvertinamas kiekvienos dalelės esamos pozicijos sprendimas. Jeigu sprendimas yra geresnis už buvusį, tai tos dalelės geriausia pozicija yra atnaujinama į esamą. Toliau yra surandamas ir atnaujinamas globaliai geriausias sprendimas ir tikrinama, ar yra pasiektas maksimalus iteracijų skaičius ar optimalaus sprendimo kriterijai. Jeigu ne, toliau tęsiamas algoritmo sprendimas atnaujinant kiekvienos dalelės greitį atsižvelgiant į esamą greitį, inerciją, dalelės geriausią sprendimą, jo svorį ir globalų sprendimą. Toliau dalelės pozicija yra atnaujinama pagal apskaičiuotą greičio vektorių. Nors šis paaiškinimas yra skirtas dalelių spiečiaus judėjimui dvimatėje erdvėje, tačiau algoritmas nėra apribotas dimensijų skaičiumi.



11 pav. *PSO* algoritmo diagrama [30]

PSO algoritmas buvo realizuotas *Vitis HLS* sistemoje sukuriant algoritmą ir jo testavimo programą. Pilnas algoritmo kodas yra pateikiamas **1** priede ir testavimo kodas – **2** priede. Pagrindinė algoritmo funkcija yra „*PSO_HLS*“, kurioje yra inicializuojamos dalelės naudojant sukurtą funkciją „*swarm_init*“, vykdomos iteracijos atnaujinant globalų sprendimą ir naudojant funkciją „*swarm_update*“ atnaujinama dalelių vektoriai, pozicijos ir sprendimai:

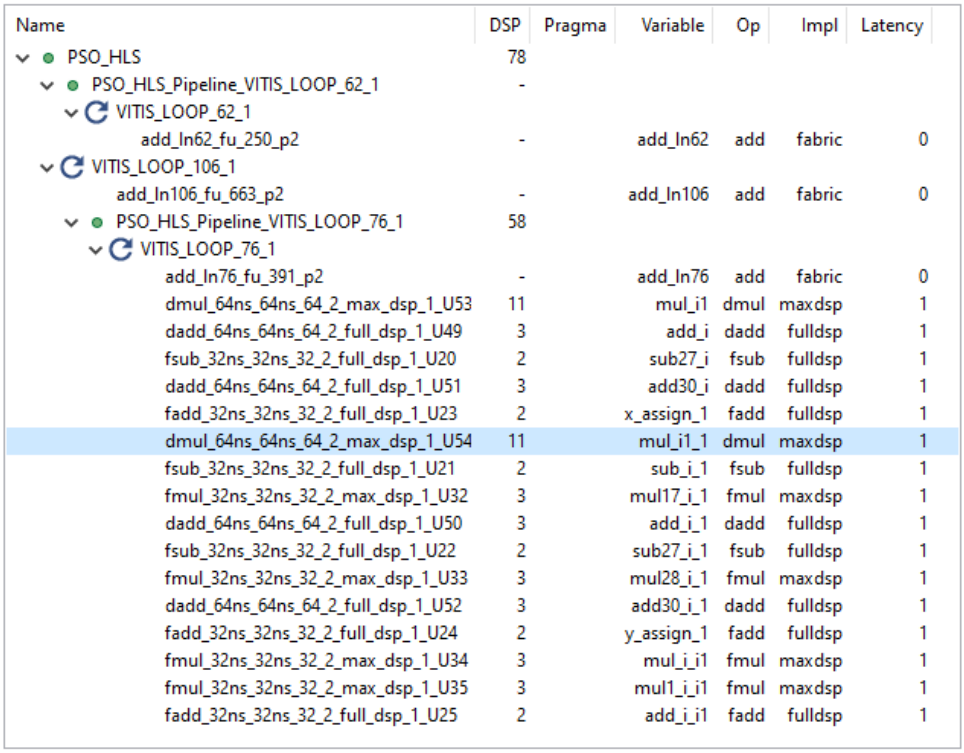
```
void PSO_HLS(float* position_X, float* position_Y, float *fitness, const char
itterations) {
   static sParticle_t swarm[SWARM_SIZE];
   static float global_best[DIMENSION];
   static float global_best_fitness = 9999;

   swarm_init(swarm);

   for (int iter = 0; iter < itterations; iter++) {

      // Update the global best
#pragma HLS unroll
      for (int i = 0; i < SWARM_SIZE; i++) {
         if (swarm[i].best_fitness < global_best_fitness) {
            global_best_fitness = swarm[i].best_fitness;
            for (int j = 0; j < DIMENSION; j++) {
               global_best[j] = swarm[i].position_best[j];
            }
         }
      }
      swarm_update(swarm, global_best);

      /* Update output */
      *position_X = global_best[0];
      *position_Y = global_best[1];
      *fitness = global_best_fitness;
   }

}
```

Vienas iš iššūkių realizuojant *PSO* algoritmą *HLS* yra tai, kad negalima naudoti kai kurių standartinės bibliotekos funkcijų tokių kaip „*rand*“. Atsitiktinių skaičių generavimas yra fundamentalus reikalavimas *PSO* algoritmui, todėl reikia savarankiškai realizuoti greitą ir efektyvų algoritmą generuoti atsitiktiniams skaičiams. Siekiant tai padaryti buvo panaudotas *XOR 32* bitų poslinkio algoritmas skirtas generuoti pseudo-atsitiktinius skaičius:

```
/* Xorshift32 */
static uint32_t RNG_seed = 12345;
float PRNG() {
   uint32_t x = RNG_seed;
   x ^= x << 13;
   x ^= x >> 17;
   x ^= x << 5;
   RNG_seed = x;
   float randomFloat = (float) x / UINT32_MAX;
   return randomFloat;

}
```

3.1.1. Optimizavimas pritaikant fiksuoto kablelio skaičių formatą

Sintezuojant „*PSO_HLS*“ funkciją gauti rezultatai parodė, kad vien *PSO* algoritmui yra sunaudojama itin daug *DSP* modulių. Tai apsunkina skaičiavimų lygiagretinimą ir optimizavimą. Sintezuojant „*PSO_HLS*“ su tikslo funkcija $f(x, y)= x^2 + y^2$ buvo sunaudojama 78 *DSP*, 8572 *FF*, 10836 *LUT* ir 0 *URAM* resursų. Įvertinant, kad *PSO* algoritmo realizavimas sunaudoja reikšmingai daugiau *DSP* resursų negu kitų – tai taptų didele problema toliau plečiant programą. Šios problemos priežastis – slankiojančio kablelio operacijos, kurios naudoja itin daug *DSP* resursų. Čia pateikiama sintezavimo atskaitos ištrauka apibendrinanti *DSP* resursų sąnaudas:



Name	DSP	Pragma	Variable	Op	Impl	Latency
PSO_HLS	78					
PSO_HLS_Pipeline_VITIS_LOOP_62_1	-					
VITIS_LOOP_62_1						
add_ln62_fu_250_p2	-		add_ln62	add	fabric	0
VITIS_LOOP_106_1						
add_ln106_fu_663_p2	-		add_ln106	add	fabric	0
PSO_HLS_Pipeline_VITIS_LOOP_76_1	58					
VITIS_LOOP_76_1						
add_ln76_fu_391_p2	-		add_ln76	add	fabric	0
dmul_64ns_64ns_64_2_max_dsp_1_U53	11		mul_i1	dmul	maxdsp	1
dadd_64ns_64ns_64_2_full_dsp_1_U49	3		add_i	dadd	fulldsp	1
fsub_32ns_32ns_32_2_full_dsp_1_U20	2		sub27_i	fsub	fulldsp	1
dadd_64ns_64ns_64_2_full_dsp_1_U51	3		add30_i	dadd	fulldsp	1
fadd_32ns_32ns_32_2_full_dsp_1_U23	2		x_assign_1	fadd	fulldsp	1
dmul_64ns_64ns_64_2_max_dsp_1_U54	11		mul_i1_1	dmul	maxdsp	1
fsub_32ns_32ns_32_2_full_dsp_1_U21	2		sub_i_1	fsub	fulldsp	1
fmul_32ns_32ns_32_2_max_dsp_1_U32	3		mul17_i_1	fmul	maxdsp	1
dadd_64ns_64ns_64_2_full_dsp_1_U50	3		add_i_1	dadd	fulldsp	1
fsub_32ns_32ns_32_2_full_dsp_1_U22	2		sub27_i_1	fsub	fulldsp	1
fmul_32ns_32ns_32_2_max_dsp_1_U33	3		mul28_i_1	fmul	maxdsp	1
dadd_64ns_64ns_64_2_full_dsp_1_U52	3		add30_i_1	dadd	fulldsp	1
fadd_32ns_32ns_32_2_full_dsp_1_U24	2		y_assign_1	fadd	fulldsp	1
fmul_32ns_32ns_32_2_max_dsp_1_U34	3		mul_i_i1	fmul	maxdsp	1
fmul_32ns_32ns_32_2_max_dsp_1_U35	3		mul1_i_i1	fmul	maxdsp	1
fadd_32ns_32ns_32_2_full_dsp_1_U25	2		add_i_i1	fadd	fulldsp	1

12 pav. PSO algoritmo *DSP* resursų sąnaudų išklotinė

Realizuojant *PSO* algoritmą su *FPGA* vienas iš esminių iššūkių yra pašalinti arba minimizuoti slankiojančio kablelio operacijas. Siekiant pašalinti šią problemą vienas iš būdų yra naudoti fiksuoto kablelio matematines operacijas. Šis metodas yra įgyvendinamas atidedant fiksuotą dalį bitų sveikai ir trupmeninei skaičiaus dalims. Tikrasis iššūkis yra realizuoti sudėtingas matematines operacijas ir pasirūpinti, kad jos būtų teisingai apskaičiuojamos ir rezultatai tilptų į pasirinkto duomenų formato diapazoną.

Ieškant ir analizuojant sukurtas bibliotekas nebuvo atrasta tinkamos bibliotekos skirtos atlikti fiksuoto kablelio operacijas *C* kalboje naudojant *HLS*. Siekiant sukurti tokią biblioteką buvo realizuotos šios operacijos:

```c
typedef struct {
    uint16_t decimal;
    uint16_t fractional;
} FixedPoint;
// Function prototypes
FixedPoint fp_add(FixedPoint a, FixedPoint b);
FixedPoint fp_sub(FixedPoint a, FixedPoint b);
FixedPoint fp_mul(FixedPoint a, FixedPoint b);
FixedPoint fp_div(FixedPoint a, FixedPoint b);
FixedPoint fp_pow(FixedPoint base, unsigned int exponent);
FixedPoint float_to_fp(float value);
float fp_to_float(FixedPoint value);
```

Tačiau kuriant biblioteką tapo akivaizdu, kad tai kelia kitas problemas ir ypač sudėtinga realizuoti kompleksinius veiksmus. Siekiant sukurti optimizuotą fiksuoto kablelio skaičių aritmetinių operacijų biblioteką, reikia itin gerai suprasti *FPGA* architektūrą. Itin svaru užtikrinti, kad operacijos būtų ne tik teisingos, bet ir greitos bei efektyviai naudojančios išteklius. Tai reikalautų išsamaus testavimo

įvairiose situacijose. Kita kylanti problema yra fiksuoto kablelio operacijų suderinamumas su kitomis bibliotekomis.

Alternatyva šiai problemai yra tiesiog naudoti *C++* programavimo kalbą ir jai pritaikytas „*Xilinx*“ įmonės bibliotekas. „*Xilinx*“ įmonė, sukūrusi *Vitis HLS* programavimo aplinką ir gaminanti *FPGA* matricas, yra sukūrusi ir pritaikiusi bibliotekas skaičiavimams su *FPGA*. Šiame darbe yra naudojamos tokios bibliotekos kaip „*ap_fixed*“ ir „*hls_math*“. Biblioteka „*ap_fixed*“ skirta būtent fiksuoto kablelio operacijoms, o „*hls_math*“ biblioteka yra skirta optimizuoti standartinės bibliotekos algoritmus. Vienas iš šių bibliotekų privalumų yra tai, kad jos yra suderintos ir su „*complex*“ biblioteka skirta atlikti skaičiavimus su kompleksiniais skaičiais. Tai bus aktualu vykdant *FFT* ir *IFFT* skaičiavimus.

Pritaikius šiuos optimizavimus ir pakeitus projektą į *C++* kalbą, gaunami daug geresni rezultatai. Perrašyta *PSO* programa yra pateikta **3** priede, „*pso*“ ir „*pso_utils*“ failuose. Realizuojant tokios pat tikslo funkcijos sintezavimą su tokiais pat apribojimais yra gaunami geresni rezultatai. Sintezuojant „*PSO_HLS*“ su tikslo funkcija x^2+y^2, su 50 ns taktinio dažnio periodo reikalavimu buvo sunaudojama 37 *DSP*, 1795 *FF*, 7030 *LUT* ir 0 *URAM* resursų. Taigi pritaikytos optimizavimo metodikos padėjo sumažinti *DSP* resursų sąnaudas net du kartus. Taip pat optimizavimas padėjo padidinti *PSO* algoritmo spartą nuo 9809 ciklų iki 2185 ciklų. Taip pat maksimalus leistinas taktinis dažnis padidėjo nuo 29,47 MHz iki 32,68 MHz.

Taigi pakeičiant slankiojančio kablelio operacijas į fiksuoto kablelio operacijas yra įmanoma sumažinti sunaudojamų *DSP* resursų skaičių ir padidinti algoritmo greitaveiką. Tai reiškia, kad galima naudoti paprastesnius ir pigesnius *FPGA* komponentus, turinčias mažiau resursų.

3.1.2. PSO algoritmo pritaikymas daugia-parametrinio uždavinio sprendimui

Šiame tyrime primityvų *PSO* modelį reikia pritaikyti sudėtingesniam uždaviniui – Brekhovskikh tikslo funkcijos optimizavimui. Ši funkcija turi šešis anksčiau minėtus argumentus ρ_2, c_2, h, α_0, n, f_0. Taigi reikia pereiti nuo sprendimo paieškos elementarioje dvimatėje koordinačių sistemoje, prie sudėtingesnės – šešių dimensijų koordinačių sistemos, kurioje kiekvieną dimensiją atitinka vis kitas argumentas. Iš pirmo žvilgsnio elementarus pakeitimas gali sukelti netikėtų sunkumų, nes kintamieji ir kintamųjų masyvai yra skirtingai sintezuojami *HLS*. Šiame poskyryje bus apžvelgta kokie iššūkiai kyla sintezuojant masyvus bei kokie pakeitimai buvo realizuoti.

Keičiant *PSO* algoritmą ir pritaikant Brekhovskikh tikslo funkcijos argumentų paieškai teko pakeisti *PSO* algoritmo dalelių struktūrą. Dalelių pozicija, greitis ir geriausios pozicijos atmintis yra keičiama iš konkrečių x ir y koordinačių į argumentų masyvą. Gali kilti klausimas, kodėl yra renkamasi naudoti masyvą, kuris neapibūdina, koks masyvo elementas atitinka kurį argumentą. Tačiau *PSO* atžvilgiu, veiksmai visiškai nepriklauso nuo to, kokia yra tikslo funkcija, ir kokie yra jos argumentai. Taigi iteruoti per masyvą yra tiesiog paprasčiau ir efektyviau.

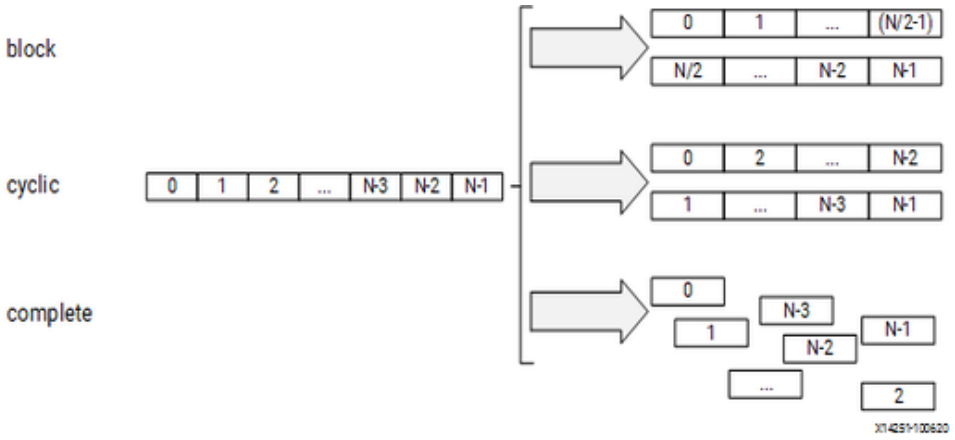
1 lentelė. Pritaikyti pakeitimai *PSO* dalelės struktūrai

PSO dalelės struktūra naudojant atskirus kintamuosius	**PSO dalelės struktūra naudojant kintamųjų masyvus**

`struct sParticle_t {`
`  ap_fixed_64p32 pos_x, pos_y;`
`  ap_fixed_64p32 vel_x, vel_y;`
`  ap_fixed_64p32 fitness_current;`
 	
`  ap_fixed_64p32 pos_best_x, pos_best_y;`
`  ap_fixed_64p32 fitness_best;`
`};`
	`struct sParticle_t {`
	`  ap_fixed_64p32 position[PSO_DIMENSION];`
	`  ap_fixed_64p32 velocity[PSO_DIMENSION];`
	`  ap_fixed_64p32 fitness_current;`
	
	`  ap_fixed_64p32 position_best[PSO_DIMENSION];`
	`  ap_fixed_64p32 fitness_best;`
	`};`

Kuriant programą skirtą procesoriui šis pakeitimas būtų palyginti paprastas. Tačiau įvykdžius pakeitimus ir bandant sintezuoti modifikuotą *PSO* algoritmą yra gaunama klaida, kad nepavyko sintezuoti *IP* šerdies pasirinktam *FPGA* komponentui, nes neįmanoma suplanuoti skaitymo ir rašymo operacijų dėl pasirinkto *FPGA* komponento atminties sąsajos trūkumo [31]. Taip yra todėl, kad masyvai, kurie neturi išorinės sąsajos su programa yra laikomi kaip *RAM* blokai, kurie turi daugiausiai dvi duomenų magistrales. Siekiant išvengti klaidos, masyvą reikia išskaidyti vienu iš trijų būdų naudojant direktyvas:

– *Block* – masyvas skaidomas į nurodytų blokų skaičių. Pavyzdžiui nurodžius parametrą „2“ masyvas skaidomas į du blokus, kuriuose pirmame bloke yra pirma dalis iš eilės einančių elementų, o antrame – likusieji elementai einantys iš eilės.
– *Cyclic* – masyvai kaip ir *Block* būdu skaidomi į nurodytą skaičių masyvų, tačiau masyvo elementai yra skaidomi pakaitomis. Pavyzdžiui į pirmą iš dviejų blokų dedant 0, 2, 4... elementus.
– *Complete* – masyvai yra išskaidomi į atskirus elementus. Elementai yra saugojami registruose.



13 pav. Masyvų elementų skaidymo metodologijos taikomos *HLS* [32]

Taigi atlikus reikalingus pakeitimus ir siekiant gebėti sintezuoti masyvus, programa buvo papildyta direktyvomis, kurios išskaido masyvus į registrus ir leidžia sintezuoti kodą:

```
    sParticle_t swarm[PSO_SWARM_SIZE];
#pragma HLS array_partition variable = swarm->position complete
#pragma HLS array_partition variable = swarm->position_best complete
#pragma HLS array_partition variable = swarm->velocity complete

    sParticle_t global_best;
```

```
#pragma HLS array_partition variable = global_best.position complete

#pragma HLS array_partition variable = global_best.position_best complete
```

Pritaikant *PSO* prie Brekhovskikh tikslo funkcijos taip pat pasikeitė ir tikslai realizuojant *PSO*. Jeigu anksčiau buvo naudojama visai nedaug resursų, nes buvo tik dvi dimensijos ir tikslo funkcija buvo ne sudėtinga. Dabar esant šešiems argumentams ir sudėtingai tikslo funkcija, tikslas nebe kuo labiau išlygiagretinti *PSO* algoritmą ir pasiekti kuo didesnę greitaveiką, bet užtikrinti, kad *PSO* algoritmas pasiekia didelę greitaveiką ir sunaudoja kuo mažiau resursų. Taigi funkcijos skirtos apskaičiuoti ir atnaujinti dalelių greitį ir poziciją yra paverčiamos į konvejerinę struktūrą siekiant panaudoti tuos pačius dauginimo elementus. Tai yra pasiekiama pritaikant „*pipeline*“ direktyvas. Nenaudojant šių direktyvų, yra bandoma nutylėtai lygiagretinti procesą. Tačiau tai lemia išaugančius resursu nuo 32 *DSP* iki 84 *DSP* elementų ir nuo 9,4k *LUT* iki 13,8k *LUT* elementų. Nuoseklus skaičiavimas taip pat užtikrina, kad tuo pat metu nebūtų bandoma skaityti ir rašyti iš masyvo, kodas būtų sintezuojamas.

3.2. *FFT* algoritmo realizavimas su *FPGA*

Realizuojant tyrimo objektą neišvengiamai reiks tiesioginės ir atvirkštinės Furjė transformacijos. Šiam tikslui gali būti panaudota „*Xilinx*“ *IP* šerdis realizuojanti *FFT* ir *IFFT* algoritmą. Kaip naudotis „*hls_fft*“ biblioteka bei nuoroda į pavyzdį yra pateikta dokumentacijoje [33]. Deja dokumentacija bei pavyzdys su testavimo stendu yra neišsamūs bei sudėtingai pritaikomi. Šiame poskyryje bus apžvelgta iššūkiai pritaikant *FFT IP* šerdį ir sukurtas jos apvalkalas skirtas lengvam pritaikymui.

Analizuojant *FFT IP* šerdies dokumentaciją buvo susidurta su klausimu, kodėl *FFT* algoritmas naudoja kompleksinį duomenų formatą tiek įėjimui, tiek išėjimui nepriklausomai ar vykdoma *FFT* ar *IFFT* transformacija. Šio klausimo neatsako nei dokumentacija, nei pateiktas pavyzdys. Šia problemą itin paaštrina tai, kad nėra pateikta metodologijos, kaip buvo sugeneruoti testavimo duomenys naudojami pavyzdžio testavimui. Iš tikrųjų taip yra todėl, kad *FFT* algoritmas gali vykdyti tiek tiesioginę, tiek atvirkštinę *FFT* transformaciją priklausomai nuo pasirinktų parametrų. Taigi vykdant tiesioginę *FFT* transformaciją užtenka užpildyti tik realiąsias įėjimo dalis.

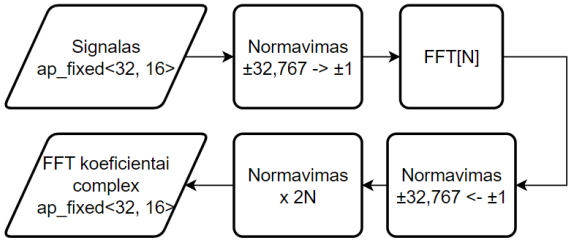
Kita problema kilo dėl nedetaliai aprašytų parametrų ir atliekamo konvertavimo. Vykdant tiesioginę ir atvirkštinę *FFT* transformacijas yra įprasta, kad turėtų būti gaunamas identiškas rezultatas. Tačiau rašant *FFT* apvalkalą buvo akivaizdu, kad gaunamas daug mažesnis rezultatas. Taip yra todėl, kad yra atliekamas duomenų normavimas, kuris nėra aprašytas dokumentacijoje. Duomenys yra normuojami siekiant išvengti rezultato duomenų persipildymo ir sugadinimo dėl fiksuoto kablelio operacijų. Todėl vykdant tiesioginę transformaciją, koeficientai yra gaunami 2N karto mažesni, kur N yra *FFT* ilgis. Kita vertus atliekant atvirkštinę Furjė transformaciją yra gaunamas išėjimas du kartus mažesnis dėl normavimo. Be šių konvertavimų naudojant *FFT* taip pat būtina normuoti įėjimo parametrus nuo -1 iki 1.

Taigi, siekiant supaprastinti *FFT* funkcijos pritaikymą ir nesudėtingai naudoti *FFT* nesirūpinant apie *FFT* algoritmo konfigūravimą bei duomenų formatavimą buvo sukurtas *FFT IP* šerdies apvalkalas skirtas šiuos veiksmus automatizuoti. Apvalkalo programa pateikta **3** priede, „*fft_wrapper*“ failuose. Čia yra pateikiama pagrindinės *FFT* ir *IFFT* funkcijos skirtos apskaičiuoti transformaciją:

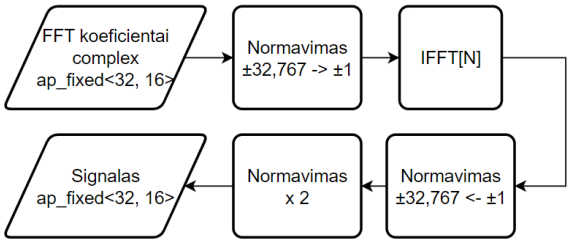
```
void FFT_call(fft_input_t in[FFT_LENGTH], fft_output_t out[FFT_LENGTH]) {
   FFT_input_to_complex(in, fft_in);
   FFT_wrapper(true, fft_in, fft_out);
   FFT_complex_to_output(fft_out, out);
}
```

```
void IFFT_call(fft_output_t in[FFT_LENGTH], fft_input_t out[FFT_LENGTH]) {
   FFT_output_to_complex(in, fft_in);
   FFT_wrapper(false, fft_in, fft_out);
   FFT_complex_to_input(fft_out, out);
}
/* Function to wrap actual FFT call */
void FFT_wrapper(bool direction, fft_complex_t in[FFT_LENGTH],   fft_complex_t
out[FFT_LENGTH]) {

   status_t fft_status;
   config_t fft_config;

   fft_config.setDir(direction);
   fft_config.setSch(0x2AB);

#pragma HLS dataflow
#pragma HLS INLINE recursive
    hls::fft<config1>(in, out, &fft_status, &fft_config);

}
```

FFT algoritmas buvo padalintas į atskiras *FFT* ir *IFFT* funkcijas bei realizuotas atitinkamas konvertavimas keičiant kompleksinius skaičius į nekompleksinius ir juos normuojant. Vykdant tiesioginę *FFT* transformaciją yra vykdomas algoritmas **14** pav. pavaizduotas algoritmas. Pirmiausia duomenys yra normalizuojami iš pasirinkto fiksuoto kablelio formato į kitą pritaikytą *FFT*, kurio vertės yra nuo -1 iki 1. Apskaičiavus *FFT* duomenys yra normalizuojami atgal ir padauginai iš 2N siekiant sugražinti pradinį mastelį. Kviečiant *IFFT* funkciją yra vykdomas algoritmas pavaizduotas **15** pav. kompleksiniai koeficientai yra normuojami į [-1;+1] formatą, o apskaičiavus *IFFT* yra gražinami į pradinį formatą.



14 pav. Įėjimų ir išėjimų normavimas vykdant *FFT* algoritmą



15 pav. Įėjimų ir išėjimų normavimas vykdant *IFFT* algoritmą

Testuojant sukurtą apvalkalą buvo įsitikinta, kad kviečiant *FFT* ir po to *IFFT* funkcijas būtų gaunamas panašus rezultatas. Mažų pokyčių galima tikėtis, nes vis dėlto yra atliekama nemažai dauginimo bei dalybos veiksmų su fiksuoto kablelio skaičiais. Tai reiškia, kad su kiekvienu veiksmu paklaidos akumuliuojasi.

3.3. Tikslo funkcijos realizavimas

Tikslo funkcija yra svarbiausia kiekvieno tyrimo bei optimizavimo uždavinio dalis. Tikslo funkcija apibūdina matematinę priklausomybę, tarp modelio įėjimų ir sprendimo tinkamo. Optimizavimo algoritmų srityje tikslo funkcija yra pagrindinis rodiklis, leidžiantis įvertinti galimų sprendinių tinkamumą tam tikroje problemos sprendimo erdvėje. *PSO* algoritmo kontekste, ši funkcija kiekybiškai įvertina atskirų dalelių pozicijų tinkamumą sprendinių erdvėje, priskirdama joms skaitinę vertę, rodančią, kaip jos atitinka optimizavimo uždavinio tikslus ir apribojimus. Atliekant iteracinį

dalelių padėčių vertinimą pagal tikslo funkciją, *PSO* nukreipia dalelių spiečių į sprendinių erdvės sritis, kurios turi geriausias tikslo funkcijos vertes. Taigi tikslo funkcija nusako būtent koks procesas bus optimizuojamas.

Šiame darbe tikslo funkciją atitinka aptartas Brekhovskikh modelis. Šį modelį galima pritaikyti apibūdinant garso bangos sklidimą dvejose skirtingose ertmėse atitinkamai nuo tų ertmių parametrų. Atitinkamai Brekhovskikh modelį galima paversti į atitinkamą tikslo funkciją, kurios tikslas minimizuoti skirtumą tarp išmatuoto ir sumodeliuoto signalo sklidimo. Tokiu būdu yra optimizuojami modelio parametrai atitinkamai pagal tai, koks yra modelio ir išmatuoto signalų kvadratų skirtumas.

Pagrindinį Brekhovskikh funkcijos išraiška yra aprašyta (2) formulėje. Tačiau skaičiuojant tai su *Matlab* galutinė išraiška yra tokia:

```
T = (2 * Z1 * Z2) ./ ...
    ( (Z1+Z2).^2 .* exp(-1i*k*h) - (Z1-Z2).^2.*exp(1i*k*h) );
```

Z1 ir Z2 – oro ir lapo ertmių akustiniai impedansai yra apskaičiuojami tik kartą. Tačiau kompleksinis bangos skaičius k yra apskaičiuojamas kiekvienai dažnio dedamajai, kurių gali būti net 1024 ar daugiau. Taigi kompleksinis bangos skaičius ir perdavimo funkcijos koeficientai yra skaičiuojami n kartų, tiek kiek yra dažnio dedamųjų naudojamų atvirkštinėje Furjė transformacijoje sumodeliuoti signalo atvaizdą laiko ašyje.

Pilnas programos kodas skirtas modeliuoti ultragarsinį signalą pagal Brekhovskikh modelį yra pateiktas **4** priede.

Tačiau realizuoti šią funkcijos išraiška su *HLS* neįmanoma dėl bibliotekų techninių apribojimų ir trūkumų. Šį *Matlab* kodą reikia ne tik išversti į *C++* kalbą, bet ir optimizuoti taip, kad būtų galima pritaikyti *HLS* bibliotekas ir sintezuoti aprašytą tikslo funkciją.

Pagrindiniai iššūkiai su kuriais yra susiduriama šiame etape yra šie:

- Taikant *HLS* reikia skaidyti operacijas. Nors ir patogu aprašyti visą funkciją viena eilute *Matlab*, tačiau tai yra prastai sintezuojama *HLS*.
- „*Xilinx*“ *HLS* bibliotekos, ypač tos, kurios naudoja fiksuoto kablelio skaičiavimus nepalaiko matematinių operacijų kaip *exp, cos* ar *cosh* su kompleksiniais skaičiais. Taigi šias operacijas reikia pakeisti kitomis.
- Atliekant skaičiavimus su *Matlab* galima pastebėti, kad atliekant tokias operacijas kaip „*4*Z1*Z2*“ ar „*(Z1+Z2)^2*“ skaičiai neretai peržengia 2^{32} ribą, maksimalią „*float*“ tipo kintamojo rezoliuciją.

Toliau bus apžvelgiami etapai kaip buvo sprendžiami šie optimizavimo ir realizavimo iššūkiai bei gauti rezultatai. Galutinis *HLS* programos kodas yra pateiktas **3** priede, „*model*“ failuose.

3.3.1. Optimizavimas skaidant skaičiavimo operacijas

Sintezuojant *HLS* kodą puvo pastebėta *HLS* savybė, kad skaidant kodą į keletą veiksmų vietoj to, kad visas veiksmas būtų aprašytas vienoje eilutėje galima sumažinti sunaudojamų resursų skaičių. Pavyzdžiui, tyrinėjant toliau pateikiamą kodą buvo pastebėta, kad išskaidant veiksmus į keletą etapų galima drastiškai sumažinti naudojamų *DSP* resursų skaičių.

Šiam *PSO* (Rastrigino) tikslo funkcijai sintezuoti yra reikalinga net 84 *DSP* resursai:

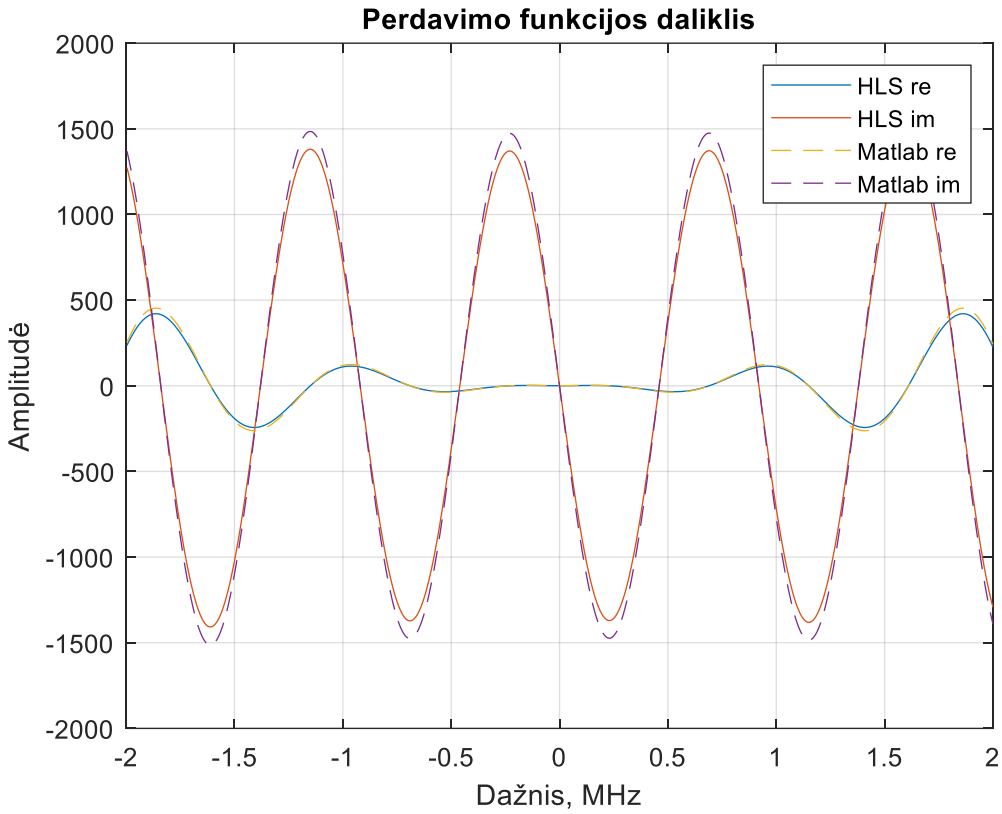
```
float_f Fitness(float_f x, float_f y) {
   /* Sphere Function */
// return x * x + y * y;

   /* Rastrigin Function */
   const float_f A = 20;
   const float_f pi = 3.1415;
   float_f retval = A + x * x - 10 * hls::cos(2 * pi * x) +
                y * y - 10 * hls::cos(2 * pi * y);
   return retval;
}
```

Taip yra todėl, kad nors ir visi skaičiai yra naudojami fiksuoto kablelio formato, tačiau *cos* operacijoms atlikti nutylėtai skaičiai yra vėl konvertuojami į slankiojančio tipo skaičius. Taigi abi *cos* operacijos reikalauja po 36 *DSP* resursų ir dar 12 *DSP* resursų reikalauja sudėti gautus rezultatus.

Pakeitus šią funkciją pagal sekantį kodą galima gauti tuos pačius rezultatus, tačiau sunaudoti vos 28 *DSP* resursus, kurie yra ypač riboti. Taip yra todėl, kad *cos* operacijos sunaudoja vos po 8 *DSP* resursus vietoj 36:

```
float_f Fitness(float_f x, float_f y) {
   /* Sphere Function */
// return x * x + y * y;

   /* Rastrigin Function */
   const float_f A = 20;
   const float_f pi = 3.1415;

   float_f cos_part_x = 2 * pi * x;
   float_f cos_part_y = 2 * pi * y;
   float_f retval = A + x * x - 10 * hls::cos(cos_part_x) +
                y * y - 10 * hls::cos(cos_part_y);
   return retval;
}
```

Taigi skaidant skaičiavimus galima sumažinti *DSP* resursų sąnaudas net iki 4,5 karto užtikrinant, kad tarpiniai rezultatai nebus pagal nutylėjimą konvertuojami į nenumatytus kintamųjų tipus. Taip pat tarpinių rezultatų skaidymas į atskirus veiksmus užtikrina ir kitus privalumus. Pavyzdžiui skaidant veiksmus galima geriau derinti programą, nes įmanoma matyti tarpinius rezultatus ir tikrinti jų tikslumą. Tai buvo pritaikyta sekant trapinių rezultatų tikslumą ir lyginant su gaunamais *Matlab* rezultatais. Atliekant programos derinimą ir optimizavimą buvo sekama *Matlab* ir *HLS* tarpinių rezultatų sutapimas ir lyginami grafikai. Čia yra pateikiamas vienas iš tarpinių rezultatų kompleksinio skaičiaus, perdavimo funkcijos apskaičiavimo daliklio grafikas.



16 pav. Perdavimo funkcijos daliklio *Matlab* ir *HLS* menamųjų ir realiųjų dalių grafikas

Kaip matyti, grafikų forma sutampa, tačiau bendra vidutinė kvadratinė paklaida yra net $5{,}43*10^3$. Šie rezultatai neapibūdina galutinių rezultatų tikslumo, tačiau demonstruoja metodiką naudotą tartinių rezultatų derinimui.

Dar vienas privalumas, kurį suteikia tarpinių rezultatų skaidymas yra tai, kad galima paprasčiau sekti, kiek konkretūs veiksmai atitinkantys kodo eilutes sunaudoja *DSP* resursų. Tai padeda lengviau atkreipti dėmesį į problematiškas vietas, kurias būtina optimizuoti.

3.3.2. Optimizavimas pakeičiant kompleksines operacijas

Tam tikrų operacijų realizuoti *HLS* sinteze yra neįmanoma su standartinėmis *HLS* bibliotekomis, nes jos nepalaiko kai kurių operacijų. Realizuojant Brekhovskikh modelį reikia atlikti skaičiavimus su kompleksiniais veiksmais. Tai yra paprasta, kol apsiribojama kompleksinių skaičių suma, skirtumu, daugyba ar dalyba. Tačiau realizuoti *exp*, *cos* ar kitų aritmetinių funkcijų su įprastomis *HLS* bibliotekomis naudojant fiksuoto kablelio skaičius yra neįmanoma. Tačiau daugelį kompleksinių operacijų galima išskaidyti ir pakeisti kitomis. Šiame poskyryje yra apžvelgiama, kokių veiksmų buvo imtasi optimizuojant, bei pakeičiant kodą taip, kad jis būtų sintezuojamas su *HLS* į *IP* šerdis.

Skaičiuojant Brekhovskikh modelio perdavimo funkcijos koeficientus pagal (2) formulę yra neįmanoma realizuoti vardiklio, kuriame *cos* ir *sin* operacijos yra atliekamos su kompleksiniais skaičiais. Tačiau pritaikius Oilerio kosinuso ir sinuso identitetų pakeitimo formules galima išskaidyti kompleksinius veiksmus taip, kad realiosios ir menamosios skaičiavimų dalys būtų apskaičiuojamos atskirai [34, 35]:

$$\sin(a + bi) = \sin a \cosh b + i \cos a \sinh b \,, \tag{6}$$

$$\cos(a + bi) = \cos a \cosh b + i \sin a \sinh b \ , \tag{7}$$

Pritaikius šiuos pakeitimus Brekhovskikh modelio koeficientai yra apskaičiuojami pagal (2) formulę, kur:

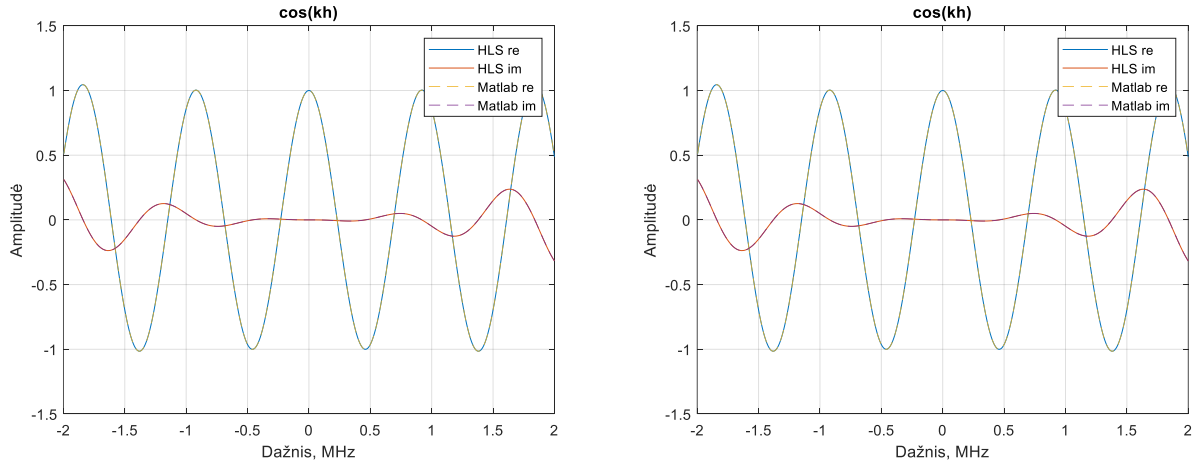
$$\sin(k'h) = \sin(k'_{re}h)\cosh(k'_{im}h) + i\cos(k'_{re}h)\sinh(k'_{im}h) \ , \tag{8}$$

$$\cos(k'h) = \cos(k'_{re}h)\cosh(k'_{im}h) + i\sin(k'_{re}h)\sinh(k'_{im}h) \ , \tag{9}$$

Galutinis rezultatas pritaikius kosinusų ir sinusų identiteto pakeitimus bei išskaidant veiksmus pagal anksčiau minėtą metodą yra štai toks:

```
        alf_1 = hls::abs(freq_axis[i]) / args.freq0;
        alf_2 = hls::pow((float)alf_1, (float)args.n);
        alf = args.alfa0 * alf_2;
        k_real = PI_x_2 * freq_axis[i] / args.c2;
        k_imag = alf;
        real_kh = k_real * args.h;
        imag_kh = k_imag * args.h;
        /* Apply Euler's Cosine Identity replacement */
        cos_kh_real = hls::cos(real_kh) * hls::cosh(imag_kh);
        cos_kh_imag = -1 * hls::sin(real_kh) * hls::sinh(imag_kh);
        cos_kh = ap_complex_32p16(cos_kh_real, cos_kh_imag);
        /* Apply Euler's Sine Identity replacement */
        sin_kh_real = hls::sin(real_kh) * hls::cosh(imag_kh);
        sin_kh_imag = hls::cos(real_kh) * hls::sinh(imag_kh);
        sin_kh = ap_complex_32p16(sin_kh_real, sin_kh_imag);
        denominator = ((ap_fixed_32p16)numerator * cos_kh) - (imag_unit * sin_kh
* denominator_sum_sqr);
        T = numerator / denominator;
```

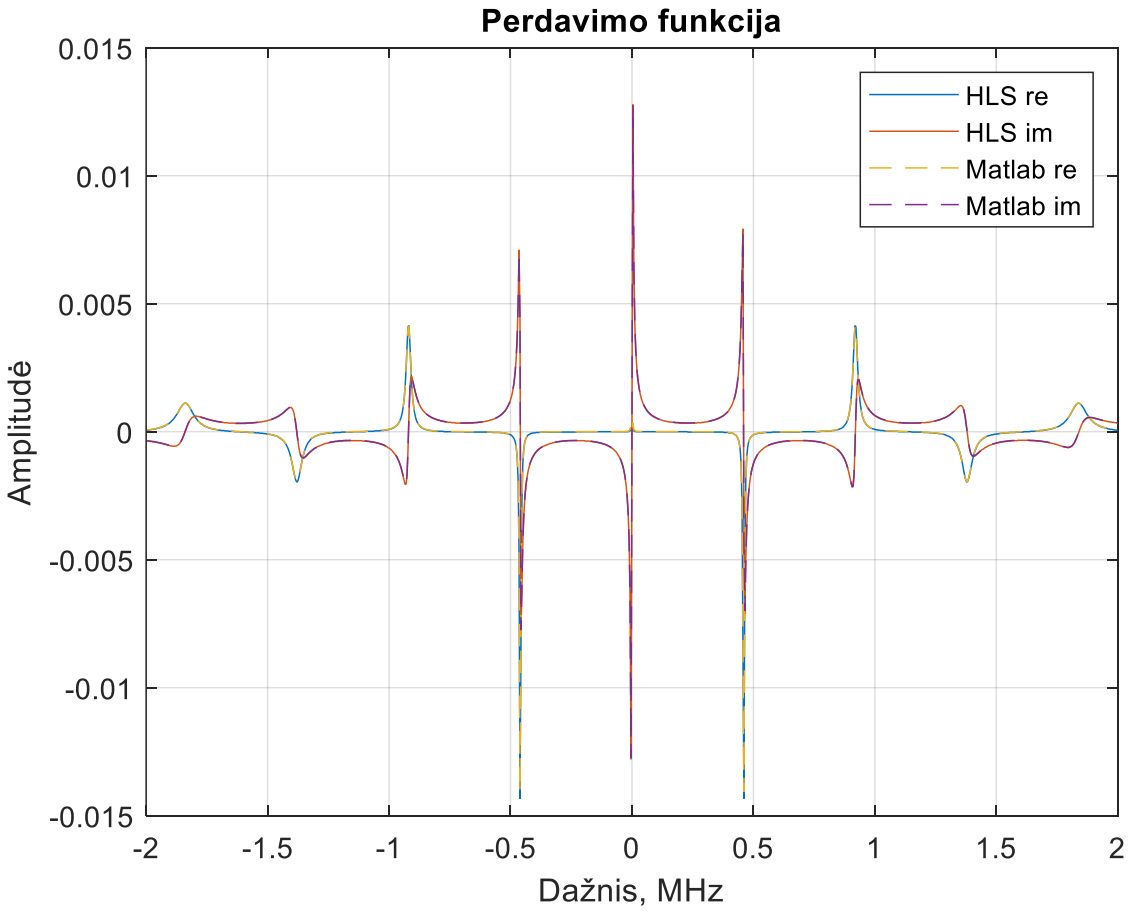
Pritaikius šį pakeitimo metodą gaunamas skirtumas tarp *Matlab* ir *HLS* simuliuojamo kodo yra minimalus. Tarpinių *cos(kh)* rezultatų vidutinė kvadratinė paklaida yra vos $1{,}14*10^{-8}$, o *sin(kh)* – $1{,}13*10^{-8}$. Vertinant *cos(kh)* ir *sin(kh)* tarpinių rezultatų palyginimą vizualiai, matosi, kad ties maksimaliomis vertėmis signalo viduryje atsiranda didžiausi skirtumai. Taip yra todėl, kad *HLS* bibliotekos naudoja verčių lenteles, pagal kurias yra nustatoma *cos*, *cosh*, *sin*, *sinh* vertės.



17 pav. *cos(kh)* ir *sin(kh) Matlab* ir *HLS* simuliuojamo kodo rezultatų palyginimas

3.3.3. Optimizavimas dalinant tarpinius rezultatus

Kuriant ir derinant *HLS* tikslo funkciją buvo pastebėta, kad nors ir tarpiniai rezultatai atitinka *Matlab* gaunamus rezultatus, tačiau galutinis perdavimo funkcijos rezultatas nesutampa. Atliekant programos derinimą, bibliotekų analizę ir eksperimentuojant buvo pastebėta, kad atliekant kompleksinių kintamųjų dalybą yra vykdoma daugybos veiksmai, kurie viršija naudojamų skaičių diapazoną. Šiai problemai spręsti buvo panaudota tarpinių skaičiavimų dalinimas. Toliau bus apžvelgiama šios dalybos įtaka bei paaiškinamos priežastys, kodėl ji buvo taikoma. Čia yra pateikiama sukurto *HLS* algoritmo ir *Matlab* rezultatų palyginimas prieš ir po pritaikant optimizacijas.



18 pav. Perdavimo funkcija gauta su *HLS* ir *Matlab*

Dauginant itin didelius skaičius, kurie išnaudoja beveik visą fiksuoto kablelio diapazoną yra gaunami skaičiai, kurie nebetelpa į atidėtų bitų skaičių ir rezultatas yra gaunamas klaidingas. Šią problemą galima spręsti keliais būdais, pavyzdžiui didinant skaičių diapazoną. Tačiau kuo didesnis skaičių diapazonas, tuo daugiau resursų bus naudojama atliekant matematines operacijas. Taip pat bibliotekos dažnai palaiko tik ribotą bitų skaičių skyrą. Taigi alternatyva yra dalinti tarpinius rezultatus taip, kad galutinis rezultatas nepasikeistų. Savaime suprantama, atliekant tarpinių rezultatų dalybą ypač su fiksuoto kablelio kintamaisiais yra prarandamas tikslumas. Tačiau rezultatas su nedidele paklaida yra geriau už klaidingą rezultatą.

Atliekant tarpinių rezultatų dalybą svarbu, kad galutinis rezultatas nepasikeistų. Tai reiškia, kad tiek, kiek kartų yra sumažinamas (2) lygties skaitiklis, tiek kartų reikia sumažinti ir vardiklį. Tai buvo atliekama pritaikant (2, 8, 9) formules ir dalinant iš numatyto *d* dalybos koeficiento:

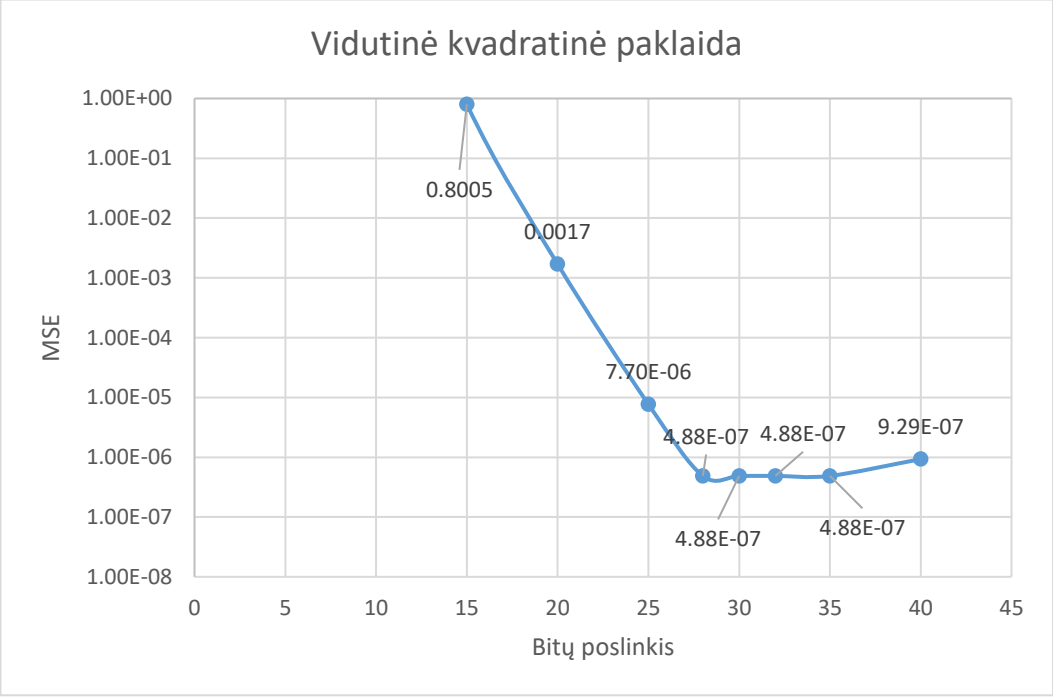
$$T(\omega) = \frac{-Z_1 Z_2}{-2Z_1 Z_2 \cos(k'h) + j(Z_1^2 + Z_2^2)\sin(k'h)} = \frac{-\frac{Z_1 Z_2}{d}}{-\frac{2Z_1 Z_2 \cos(k'h)}{d} + j\left(\frac{Z_1^2}{d} + \frac{Z_2^2}{d}\right)\sin(k'h)}, \quad (10)$$

Programuojant buvo renkamasi taip dalinti kintamuosius, kad būtų išvengiama ketamųjų persipildymo klaidų juos dauginant. Taigi buvo dalinami būtent kintamieji su didžiausia amplitude, kurių nereikia perskaičiuoti kiekvieną ciklą – Z_1 ir Z_2 apskaičiuojant jų sumažintas dalis Z_{1d} ir Z_{2d}. Tuomet yra gaunama tokia perdavimo funkcijos išraiška:

$$T(\omega) = \frac{-Z_1 Z_{2d}}{-2Z_1 Z_{2d} \cos(k'h) + j(Z_1 Z_{1d} + Z_2 Z_{2d})\sin(k'h)}; Z_{1d} = \frac{Z_1}{d}; Z_{2d} = \frac{Z_2}{d}, \quad (11)$$

Tačiau dalybos veiksmai kaip jau anksčiau minėta naudoja *DSP* resursus. Tai galima toliau optimizuoti pakeičiant dalybą bitų poslinkiu, kadangi šiuo atveju nėra svarbu, iš kokio skaičiaus bus dalinami tarpiniai rezultatai, jeigu dalybos konstanta bus išprastinta. Vadinasi galima pakeisti dalybos veiksmą kitu veiksmu – bitų poslinkiu į dešinę. Šiuo būdu galima sutaupyti net 36 *DSP* resursus su kiekviena dalybos operacija, pakeičiant 64 bitų rezoliucijos kintamųjų dalybą bitų poslinkiu.

Lieka klausimas – per kiek bitų reikia perstumti į dešinę kintamuosius? Su kiekvienu bito postūmiu yra prarandama informacija it tikslumas, tačiau užtikrinama ir didesnė atsarga, kad skaičių bitai nepersipildys atliekant daugybą. Siekiant tai įverti buvo modeliuojama perdavimo funkcijos skaičiavimas naudojant įprasto modelio parametrus. Buvo vertinama *Matlab* ir *HLS* skaičiavimo tikslumai naudojant skirtingą skaičių bitų postūmiui. Toliau yra pateikiama, kaip keičiasi perdavimo funkcijos vidutinė kvadratinė paklaida priklausomai nuo bitų postūmio (dalybos koeficiento). Ties 40 bitų postūmiu su *HLS* yra sugadinamas rezultatas ir yra gaunama tik nulinės vertės, tačiau su 28 – 35 bitų poslinkiu yra gaunami tikslūs ir *Matlab* atitinkantys rezultatai.



19 pav. Perdavimo funkcijos vidutinė kvadratinės paklaidos priklausomybė nuo dalybos konstantos (naudojant bitų poslinkį)

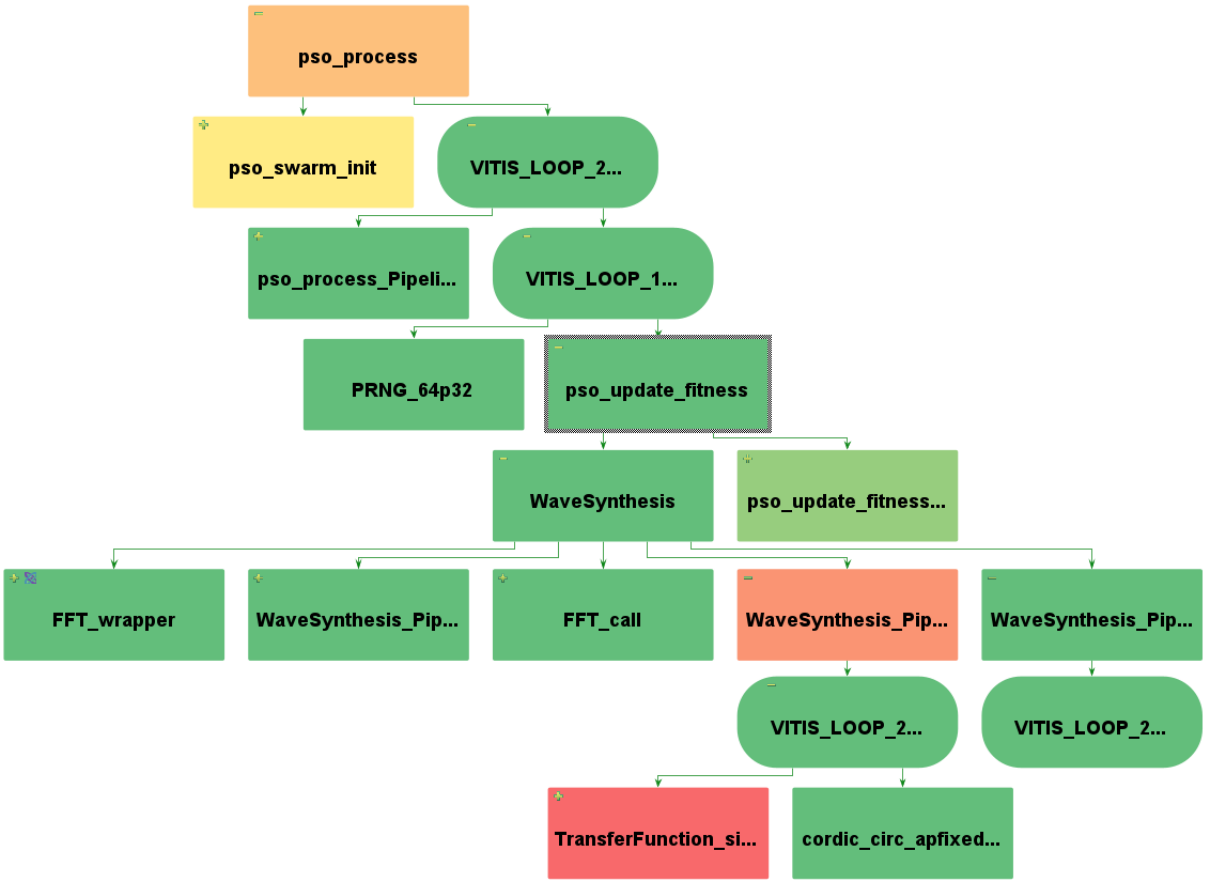
3.4. Sukurto *FPGA* projekto architektūra

Sukūrus *HLS IP* šerdį, kuri teikia norimus rezultatus yra pravartu įvertinti kokia yra sukurtos *IP* šerdies architektūra. Nuo pradinio programos realizavimo, *FPGA* projektas buvo daug optimizuojamas keičiant jo architektūrą. Nuo sukurtos architektūros priklauso kaip yra vykdomos skaičiavimo operacijos ir kokia yra jų seka. Šiame poskyryje apžvelgiama sukurtos *IP* šerdies architektūra, kokia seka ir kaip atliekami skaičiavimai (lygiagrečiai, nuosekliai ar konvejerine struktūra) ir toliau analizuojama, kiek ir kokių resursų sunaudoja *IP* šerdis. Taip pat analizuojama kurios programos dalys ir kurie veiksmai yra imliausi *FPGA* resursams.

Sukurta programa susideda iš dviejų dalių: inicializavimo ir iteracinio paieškos metodo. Pirmiausia yra inicijuojama spiečiaus dalelės: sugeneruojama atsitiktinės pozicijų (argumentų) vertės, kiekvienai daleliai priskiriama maksimali tikslo funkcijos vertė, bei priskiriama pradinės greičio ir geriausios pozicijos vertės. Iteracinis procesas susideda iš šių veiksmų:

1. atnaujinama spiečiaus dalelių duomenys:
    a. atnaujinamas dalelių greitis;
    b. atnaujinama dalelių pozicija (argumentai);
    c. apskaičiuojama dalelių tikslo funkcija ir atnaujinama dalių tinkamumas. Šiame etape yra skaičiuojama perdavimo funkcija, sintezuojama ultragarso banga ir skaičiuojamas jos ir išmatuotos bangos vidutinė kvadratinė paklaida;
2. surandama tinkamiausia dalelė su geriausiu tikslo funkcijos rezultatu;
3. nukopijuojama geriausio sprendimo rezultatai į išvesties masyvą.

Šiame iteraciniame procese pirma iteracija yra ypatinga, nes joje yra praleidžiami dalelių greičio bei pozicijos atnaujinimo etapai. Pirmą iteraciją galima priskirti inicializavimo etapui. Toks *PSO* algoritmas yra pritaikytas būtent *FPGA* architektūrai siekiant minimizuoti sintezuojamų modulių kiekį. Žemiau pateiktame paveikslėlyje yra atvaizduojama sukurto *FPGA* projekto funkcijų kvietimo seka. Čia kvadratais stačiais kampais reprezentuoja funkcijas, o apvaliais kampais – ciklus. Objektų spalva atitinka sunaudojamų *DSP* resursų kiekį, kur ryškiai žalia – mažiausia, o ryškiai raudona – daugiausiai *DSP* resursų. Kaip matyti funkcija „*TrasferFunction*“, skirta skaičiuoti perdavimo funkciją sunaudoja daugiausiai *DSP* resursų ir visa architektūra sudaryta taip, kad ši funkcija būtų kviečiama tik vienoje vietoje ir būtų sintezuojamas tik vienas šios funkcijos objektas.



20 pav. *Vitis HLS* atvaizduojama funkcijų kvietimo seka

Programos architektūra yra pasirinkta taip, kad projektui reikalingi resursai nepriklausytų nuo iteracijų ir dalelių skaičiaus. Tai reiškia, kad visi ciklai iteruojantys per visas daleles yra konvejerinės struktūros, o ciklai iteruojantys per dalelių pozicijas ir greitį – yra lygiagretūs. Taip pat pagrindinis „*pso_process*“ ciklas negali būti išlygiagretinamas, nes sekančios iteracijos rezultatai priklauso nuo ankstesnės.

3.4.1. Sukurtos *FPGA* programos resursų sąnaudos

Pagrindinės *FPGA* projekto dalys, kurių resursų sąnaudas galima konkrečiai apibrėžti yra šie:

- perdavimo funkcijos apskaičiavimo procesas (funkcija „*TransferFunction_single*“). Optimizavimo metu, ši funkcija buvo pakeista tokia, kuri apskaičiuoja po vieną perdavimo funkcijos koeficientą vienu kvietimu;
- ultragarsinės bangos sintezavimo procesas (funkcija „*WaveSynthesis*“). Ši funkcija pritaiko *FFT*, *IFFT* ir perdavimo funkcijos skaičiavimus sintezuojant ultragarsinę bangą pagal Brekhovskikh modelį;
- *PSO* algoritmas integruojantis Brekhovskikh modelį (funkcija „*pso_process*“).

Atskirtų dalių ir viso projektų sunaudojamų resursų bei reikalingų ciklų skaičius pateikiama **2** lentelėje. Šioje lentelėje yra pateikiama, kiek *FPGA* ciklų reikia įvykdyti visą programą, kai yra 100 spiečiaus dalelių, 100 iteracijų su 1024 elementų signalu. Kaip atskaitos taškas, buvo pasirinktas *FPGA* komponentas tenkinantis resursų reikalavimus sukurtam projektui. Lentelėje pateikiama, kiek procentų yra sunaudojama visų *FPGA* komponentų resursų sintezuojant tik šiuos modulius. Eilutė „*pso_process* (argumentų suma)“ atvaizduoja kiek resursų sunaudoja *PSO* algoritmas netaikant

Brekhovskikh modelio, o tiesiog sumuojant argumentus. Taip galima įvertinti *PSO* algoritmui reikalingą resursų kiekį.

2 lentelė. Atskirų modulių *FPGA* resursų sąnaudos

Modulis	**Ciklų skaičius**	**BRAM**	**DSP**	**FF**	**LUT**
TransferFunction_single	234	7 (~0%)	129 (17%)	51474 (19%)	58146 (43%)
WaveSynthesis	11767	43 (5%)	224 (30%)	97549 (36%)	108671 (80%)
pso_process (argumentų suma)	151706	50 (6%)	56 (7%)	3988 (1%)	7647 (5%)
pso_process (pilnas modelis)	129241008	83 (11%)	289 (39%)	107041 (39%)	118354 (87%)

Vienas pigesnių *FPGA* komponentų tenkinančių visus *FPGA* resursų reikalavimus yra *AMD Artix-7 XC7A200T-1FBG484C* komponentas [36]. Šis komponentas turi 33650 *SLICE*, 134600 *LUT*, 269200 *FF*, 740 *DSP*, 365 *BRAM* resursus. Pateiktame kataloge komponentas kainuojama 280 € po importo mokesčių. Įprasta mokymui skirta FPGA plokštė „*Digilent Nexys Video Artix-7*“, naudojanti *XC7A200T* FPGA komponentą kainuoja 505 € [37]. Sintezuojant visą *FPGA* projektą daugiausia yra sunaudojama *LUT* elementų – 116143 arba 86 % visų komponento *LUT* išteklių. Sintezuojamas pilnas modelis tenkina laiko reikalavimus su maksimaliu 63 MHz dažniu.

Analizuojant lentelę matosi, kad iš „*TransferFunction_single*“ atlieka pagrindinius tikslo funkcijų skaičiavimus su fiksuoto kablelio tipais. Todėl šis modulis sunaudoja itin daug *DSP* resursų. „*WaveSynthesis*“ modulis pritaiko *FFT* ir *IFFT* operacijas, todėl sunaudoja daugiausiai *LUT* resursų palyginus su kitais programos elementais. Galiausiai, „*pso_process*“ (argumentų suma) parodo, kad nors pats *PSO* algoritmas nereikalauja itin daug resursų, tačiau reikalauja didžiausio ciklų skaičiaus skaičiuojant visas iteracijas ir spiečiaus dalelių parametrus.

Toliau **3** lentelėje yra pateikiami duomenys, kaip *FPGA* programos ciklų skaičius ir resursai priklauso nuo dalelių skaičiaus naudojant 100 iteracijų su 1024 elementų signalu. Analizuojant šią lentelę galima pastebėti, kad kaip ir buvo tikėtasi iš *FPGA* programos architektūros, resursų skaičius neturi priklausyti nuo dalelių skaičiaus, tačiau reikalingas ciklų skaičius proporcingai didėja.

3 lentelė. *FPGA* resursų sąnaudų priklausomybė nuo dalelių spiečiaus dydžio, iteracijų skaičius – 100

Dalelių spiečiaus dydis	**Ciklų skaičius**	**BRAM**	**DSP**	**FF**	**LUT**
50	64620957	83 (11%)	289 (39%)	107014 (39%)	118354 (87%)
100	129241008	83 (11%)	289 (39%)	107041 (39%)	118354 (87%)
200	258481107	83 (11%)	289 (39%)	107068 (39%)	118360 (87%)

4 lentelėje yra pateikiama *FPGA* programos ciklų skaičius ir resursai priklauso nuo iteracijų skaičiaus naudojant 100 spiečiaus dalelių su 1024 elementų signalu. Analizuojant šią lentelę galima pastebėti, kad taip pat kai ir **3** lentelėje, resursų priklausomybės nuo iteracijų skaičiaus nėra, o ciklų skaičius – tiesiogiai priklausomas.

4 lentelė. *FPGA* resursų sąnaudų priklausomybė nuo dalelių spiečiaus dydžio, iteracijų skaičius – 100

Iteracijų skaičius	Ciklų skaičius	BRAM	DSP	FF	LUT
50	64620557	83 (11%)	289 (39%)	107039 (39%)	118354 (87%)
100	129241008	83 (11%)	289 (39%)	107041 (39%)	118354 (87%)
200	258481907	83 (11%)	289 (39%)	107043 (39%)	118356 (87%)

3.5. Programos realizavimas įterptine sistema su *Linux* operacine sistema

Sukurtas *FPGA* projekto kodas buvo pritaikytas programai, kuri veikia *Linux* operacinėje sistemoje. Programą buvo pasirinkta kurti su *Linux* operacine posisteme skirta *Windows*. Programos kompiliavimui buvo panaudotas *Cmake* įrankis su *GCC 11.4* kompiliatoriumi. Programa buvo kuriama siekiant atlikti kuo mažiau pakeitimų ir modifikacijų pradiniam *FPGA* projekto kodui ir palikti kuo panašesnę programos struktūrą. Nuoroda į pilną programos kodą yra pateikta **5** priede.

Perkeliant kodą iš *Vitis HLS* darbo aplinkos į *Cmake* projektą reikalinga keletas pakeitimų:

– nebereikia naudoti fiksuoto kablelio operacijų, kurios gali tik sulėtinti skaičiavimus. Taigi anksčiau naudoti *ap_fixed* tipai yra pakeičiami *float* ir *double* kintamųjų tipais. Paprasčiausia tai pasiekti tiesiog pakeičiant naudojamų kintamųjų tipų apibrėžimus. Tai leidžia nemodifikuoti likusio kodo, paliekant jau naudojamus kintamuosius *ap_complex_32p16* ir *ap_complex_64p32* tačiau pakeičiant jų apibrėžimą:

5 lentelė. *FPGA* ir *Cmake* projektų kintamųjų tipų atitikmenys

FPGA projekto kintamųjų tipai	Cmake projekto kintamųjų tipai

`typedef ap_fixed<32, 16> ap_fixed_32p16;`
`typedef ap_fixed<64, 32> ap_fixed_64p32;`
`typedef std::complex<ap_fixed_32p16> ap_complex_32p16;`
`typedef std::complex<ap_fixed_64p32> ap_complex_64p32;`
	`typedef float ap_fixed_32p16;`
	`typedef double ap_fixed_64p32;`
	`typedef std::complex<ap_fixed_32p16> ap_complex_32p16;`
	`typedef std::complex<ap_fixed_64p32> ap_complex_64p32;`

– *hls* bibliotekos pakeičiamos į universalias, *C++* bibliotekas pritaikytas procesoriams. Tai reiškia, kad *hls* funkcijos yra pakeičiamos standartinėmis *C++* bibliotekos funkcijomis ir *HLS FFT* biblioteka yra pakeičiama *fftw3* biblioteka [38];
– *Xorshift* atsitiktinių skaičių generavimo algoritmas yra pakeičiamas standartine *C++* funkcija *rand()*.

Projektui yra reikalingos šios programas:

– *Cmake* – pagrindinis įrankis skirtas kompiliuoti programą. Įrašoma į *Ubuntu Linux* su komanda: „*sudo apt install cmake*“.
– *fftw3* – biblioteka skirta atlikti greitąją Furjė transformaciją. Įrašoma į *Ubuntu Linux* su komanda: „*sudo apt install libfftw3-dev*“.
– *gprof* – programos greitaveikos analizės įrankis. Įrašoma į *Ubuntu Linux* su komanda: „*sudo apt install gprof*“.

Norint sukompiliuoti, paleisti ir ištestuoti programos greitaveiką galima tai padaryti rankiniu būdu arba paleisti parašytą „*shell*“ programą „*profile_project.sh*“.

4. Tyrimo rezultatai

Šiame skyriuje yra apžvelgiama ir vertinami rezultatai gauti su *Linux* ir *FPGA* sistemomis. Bus apžvelgta ir palyginta greitaveika ir analizuojama, kiek yra naudojama išteklių bei kokia jų kaina.

Šiame tyrime buvo lyginama ši aparatūra:

- „*AMD Artix-7 XC7A200T-1FBG484C*“ *FPGA* komponentas [36]. Šis komponentas buvo pasirinktas, nes yra vienas iš pigesnių *FPGA* komponentų tenkinančių visus resursų reikalavimus. Šio komponento resursai: 33650 *SLICE*, 134600 *LUT*, 269200 *FF*, 740 *DSP*, 365 *BRAM*. Pateiktame kataloge komponentas kainuojama 280,33 € po mokesčių. Realizuotas algoritmas buvo testuojamas 50 MHz dažniu;
- nešiojamas „*Dell G3 3779*“ kompiuteris su „*Intel i7-8750H*“ procesoriumi, 16 GB *RAM* ir „*NVIDIA GeForce GTX 1050Ti*“ grafine plokšte [39]. Šis kompiuteris buvo pasirinktas dėl esamų išteklių kaip atskaitos taškas lyginant įterptinę sistemą su vartotojam skirtu nešiojamuoju kompiuteriu;
- mini kompiuteris „*Raspberry Pi 4B*“ su „*Cortex-A72*“ procesoriumi ir 4GB *RAM* [40]. Šis mini kompiuteris buvo pasirinktas kaip tipinė *Linux* įterptinė sistema.

4.1. Greitaveikos palyginimas

Tiriant *FPGA* ir *Linux* sistemų greitaveiką buvo lyginama, kiek laiko užtrunka įvykdyti pavyzdinio uždavinio parametrų radimo programą. Tiriama greitaveikos priklausomybė nuo iteracijų ir *PSO* algoritmo dalelių spiečiaus skaičiaus.

Kaip atskaitos taškas buvo pasirinktas 400 iteracijų su 400 spiečiaus dalelių *PSO* parametrų rinkinys teikiantis gerus rezultatus ir nustatantis modelio parametrus su 2% paklaida. Greitaveika buvo testuojama iš pradžių fiksuojant 400 iteracijų ir keičiant PSO spiečiaus dalelių skaičių nuo 100 iki 800 dalelių. Antru bandymu greitaveika buvo testuojama su 400 spiečiaus dalelių keičiant iteracijų skaičių nuo 100 iki 800 iteracijų. Greitaveikos priklausomybės nuo spiečiaus dalelių skaičiaus yra pateikta **6** lentelėje, o gautų rezultatų grafinė reprezentacija atvaizduota **21** pav. Greitaveikos priklausomybės nuo iteracijų skaičiaus yra pateikta **7** lentelėje.

Tiriant greitaveiką buvo tiriami du kraštutiniai nešiojamojo kompiuterio pritaikymo būdai:

- **nešiojamas kompiuteris[1]** - kai kompiuteris yra prijungtas prie maitinimo šaltinio ir veikia greičiausiu režimu, tačiau naudoja daugiausiai galios. Tai atitinka geriausią nešiojamojo kompiuterio greitaveikos scenarijų;
- **nešiojamas kompiuteris[2]** - kai kompiuteris yra atjungtas nuo maitinimo šaltinio ir veikia lėčiausiu režimu siekiant suvartoti kuo mažiau galios. Tai atitinka prasčiausią nešiojamojo kompiuterio greitaveikos scenarijų.

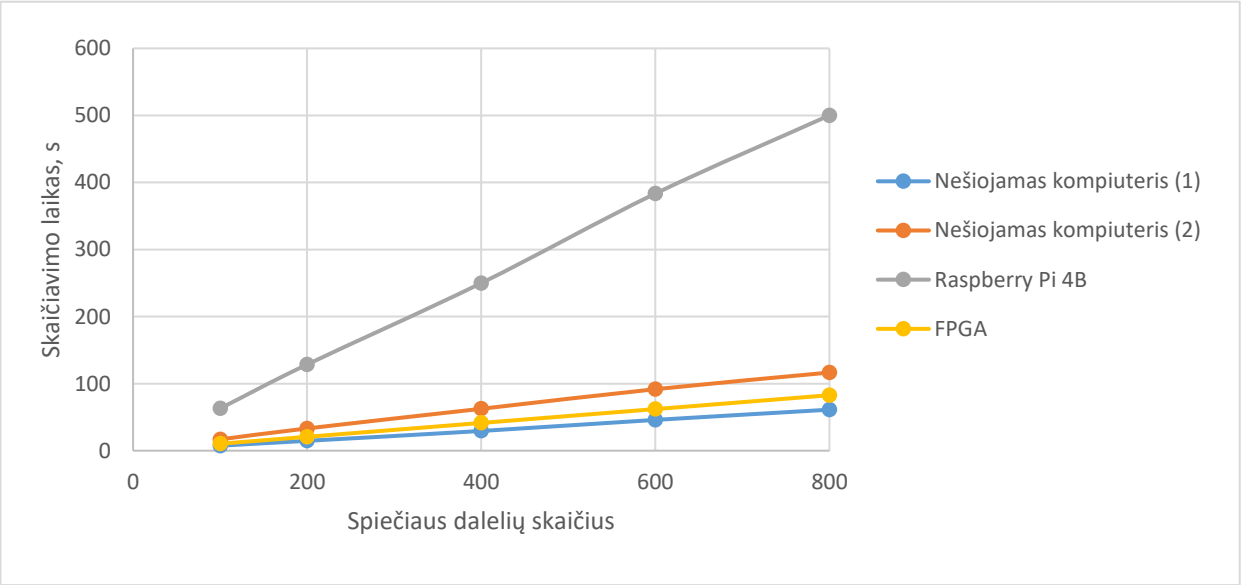
6 lentelė. Aparatūros skaičiavimo laiko priklausomybė nuo spiečiaus dalelių skaičius su 400 iteracijų

	Aparatūros skaičiavimo laikas, s			
Spiečiaus dalelių skaičius	**Nešiojamas kompiuteris[1]**	**Nešiojamas kompiuteris[2]**	***Raspberry Pi 4B***	***FPGA***
100	7,43	16,82	62,93	10,34
200	14,73	33,19	128,59	20,68
400	29,40	62,46	249,94	41,36

600	45,89	91,69	383,45	62,04
800	61,13	116,52	500,15	82,71

7 lentelė. Aparatūros skaičiavimo laiko priklausomybė nuo iteracijų skaičius, 400 spiečiaus dalelių

	Aparatūros skaičiavimo laikas, s			
Iteracijų skaičius	**Nešiojamas kompiuteris[1]**	**Nešiojamas kompiuteris[2]**	***Raspberry Pi 4B***	***FPGA***
100	8,09	14,73	62,92	10,34
200	13,82	30,67	128,08	20,68
400	29,40	62,46	249,94	41,36
600	44,94	87,38	384,57	62,04
800	61,47	119,19	491,35	82,71



21 pav. Aparatūros skaičiavimo laiko priklausomybė nuo spiečiaus dalelių skaičius su 400 iteracijų

Analizuojant gautus rezultatus matosi, kad realizuotas algoritmas yra tiesinio laiko (O(n)), spiečiaus dalelių ir iteracijų atžvilgiu. Tai reiškia, kad didinant spiečiaus dalelių skaičių arba iteracijų skaičių – algoritmo skaičiavimo laikas didėja tiesiškai ir proporcingai. Analizuojant rezultatus taip pat galima pastebėti, kad didinant iteracijų ir spiečiaus dalelių skaičių – skaičiavimo laikas didėja beveik identiškai. Išvardinti bruožai yra būdingi visai aparatūrai.

Tiesinė skaičiavimo priklausomybė nuo skaičiavimų masto gali pasirodyti nebūdinga *FPGA* matricoms, nes *FPGA* technologija yra paremta konvejeriniais ir lygiagrečiais skaičiavimais, kurie turėtų sutaupyti laiko. Tačiau, bet kokie tolimesni *FPGA* algoritmo optimizavimo metodai suteikia neproporcingai mažus greitaveikos pranašumus itin didelių resursų kaina. Kita vertus algoritmas turi **3.4** aprašytą architektūros privalumą, kad jo resursų skaičius nepriklauso nuo iteracijų ar spiečiaus dalelių skaičiaus.

Analizuojant **6**, **7** lenteles ir **21** grafiką, matosi, kad *FPGA* komponentas yra vidutiniškai 28% procentais lėtesnis nei nešiojamas kompiuteris maitinamas iš šaltinio ir veikiantis didžiausiu

pajėgumu. Tačiau *FPGA* yra 49 % greitesnis nei nešiojamas kompiuteris veikiantis baterijos saugojimo režimu ir net 510 % greitesnis nei *Raspberry Pi 4B* kompiuteris.

Algoritmas realizuojamas *FPGA* smarkiai pralenkia įprastą įterptinę skaičiavimo sistemą, tačiau nusileidžia naudotojams skirtam procesoriui veikiančiu pilnu pajėgumu. Tai atitinka bendrą suvokimą, kad naudotojam skirtas kompiuteris ar debesų kompiuterija gali apdoroti duomenis ir išspęsti uždavinius greičiau negu daiktų interneto pakraščio sluoksnio prietaisas. Tačiau 28% procentai yra palyginus nedidelis skirtumas, kurį gali pašalinti duomenų persiuntimo eliminavimas.

4.2. Skaičiavimo įrangos resursų palyginimas

Šiame poskyryje yra apžvelgiama, kiek testuojama aparatūra sunaudoja resursų, bei kokios yra jų sąnaudos. Taip pat palyginama resursų ir greitaveikos santykis bei energetinis efektyvumas. Taikant daiktų internetą ar perkeliant skaičiavimus iš debesų kompiuterijos į daiktų interneto pakraštį yra itin aktualus finansinis efektyvumas. Net jeigu daiktų interneto pakraštyje realizuotas sprendimas veikia daug greičiau, tačiau yra reikšmingai brangesnis – jis tikriausia nebus realizuojamas. Taigi itin svarbu ne tik sprendimo greitaveika, bet ir finansinis aspektas.

Nešiojamojo kompiuterio veikiančio maksimaliu greičiu *i7-8750H* procesorius pasiekia apytiksliai 4 GHz taktinį dažnį veikdamas *turbo* režimu. Procesorius išnaudoja vidutiniškai 16 % resursų skaičiavimams ir užtrunka 29,40 s skaičiuojant 400 iteracijų su 400 spiečiaus dalelių uždavinį. Procesorius turi 12 branduolių, kuriems paskirsto operacinės sistemos, *Linux* posistemės ir skaičiavimo darbus. Procesorius gali lygiagrečiai skaičiuoti 10 modelių skaičiavimus taip išnaudojant iki 100 % procesoriaus resursų kitus skiriant šalutiniams operacinės sistemos procesams. Tačiau vykdant daugiau nei vieno modelio skaičiavimus, skaičiavimai gali užtrukti net iki dviejų kartų ilgiau.

Nešiojamojo kompiuterio veikiančiu ribotu greičiu ir atjungto nuo maitinimo šaltinio procesorius pasiekia tik 2 GHz taktinį dažnį ir išnaudoja vos 10 % procesoriaus resursų. Tačiau skaičiavimai identiško uždavinio užtrunka 62,46 s. Paleidžiant 10 lygiagrečius modelio skaičiavimus yra panaudojama daugiausiai 60 % procesoriaus resursų ir užtrunka taip pat du kartus ilgiau. Paleidžiant daugiau negu vieną procesą vienu metu, kompiuterio taktinis dažnis yra apriejamas iki 1,4 GHz. Didinant lygiagrečių procesų skaičių, sunaudojamų resursų kiekis neviršija 60 %.

Raspberry Pi 4B skaičiavimams panaudoja 100% vieną iš keturių procesoriaus branduolių ir veikia maksimaliu 1,5 GHz greičiu visą laiką. Tą patį uždavinį *Raspberry Pi* išsprendžia per 249,94 s. *Raspberry Pi* turi keturis procesoriaus branduolius, kurios visus galima panaudoti lygiagretiems skaičiavimams. Paleidus 4 lygiagrečius modelio skaičiavimus skaičiavimai užtrunka 256,88 s, vos 3 % ilgiau.

FPGA komponento *XC7A200T-1FBG484C* išnaudojami resursai vertinami šiek tiek kitaip. Sintezuota *IP* šerdis išnaudoja 11 % *BRAM*, 39 % *DSP*, 39 % *FF* ir 89 % *LUT* resursų. Komponentas tenkina laiko reikalavimus veikdamas 50 MHz taktiniu dažniu. Taigi lieka dar šiek tiek resursų modelio derinimui ir tobulinimui. Tačiau su šiuo *FPGA* komponentu negalima realizuoti daugiau nei vienos *IP* šerdies.

Aprašytos aparatūros greitaveika ir rinkos kainos yra pateikiamos **8** lentelėje. Akivaizdu, kad kuo brangesnis elektronikos prietaisas, tuo daugiau jis turi išteklių ir greičiau atlieka skaičiavimo užduotis. Tačiau įvertinus tai, kad *FPGA* komponento kaina yra per pus mažesnė, už nešiojamojo

kompiuterio ir programuojama logika veikia greičiau už kompiuterį, veikiantį galios taupymo režimu, galima drąsiai teigti, kad *FPGA* turi geriausią greitaveikos ir kainos santykį.

Kita vertus *FPGA* gali atlikti tik vieno proceso skaičiavimus vienu metu, o nešiojamojo kompiuterio procesorius gali vykdyti net 8 ar daugiau procesus lygiagrečiai. Atitinkamai programa gali būti pritaikyta būtent procesoriui ir išlygiagretinta, naudojant kelis branduolius atlikti vieną uždavinį. Taip pat reikia įvertinti, kad nešiojamas kompiuteris yra universalus prietaisas, pritaikomas įvairioms užduotims ir turintis įvairios periferijos. Taigi jeigu būtų vertinama tik procesoriaus *i7-8750H* kaina, kuri yra tik 370,06 Eur [41], kainos ir greitaveikos santykis gerokai pakrypsta į vartotojams skirto procesoriaus pusę.

4.3. Energetinis efektyvumas

Vertinant daiktų interneto pakraščio sprendimų ekonominį tinkamumą yra itin svarbus energetinis efektyvumas. Prietaisai veikiantys daiktų interneto pakraštyje dažnai yra maitinami ne iš elektros tinklo, bet naudojant baterijas. Taigi, energetinis efektyvumas nusprendžia ir kaip ilgai veiks prietaisas su ta pačia baterija, bei koks bus baterijos svoris.

Atlikus modeliavimą naudojant Vivado programinę įrangą buvo apskaičiuota, kad „*XC7A200T-1FBG484C*“ *FPGA* komponentas sunaudoja 1,06 W galios veikdamas pilnu pajėgumu ir atlikdamas skaičiavimus. 0,93 W sudaro dinaminė galia, kuri yra reikalinga skaičiavimams atlikti, o 0,14 W sudaro statinė galia reikalinga užmaitinti prietaisą. Šios galios sąnaudos atitinka realizuojant pavyzdinio uždavinio algoritmą naudojant 400 spiečiaus daleles ir 400 iteracijas su 50 MHz taktiniu dažniu.

Pagal specifikacijas, *Raspberry Pi 4B, Broadcom BCM2711, Cortex-A72* procesoriaus išskiriama šiluminė galia yra 7,5W pagal *TDP* (angl. *Thermal Design Power*) ir 3 W pagal *TDP Down*. Akivaizdu, kad 7,5 W procesorius visą laiką nesunaudoja [42]. Tik veikdamas maksimaliu pajėgumu – pavyzdžiui skaičiuodamas keturis lygiagrečius procesus su keturiais procesoriaus branduoliais, procesorius pasiekia šį galios suvartojimą. Taigi, *Raspberry Pi 4B* procesorius nusileidžia energetiniu našumu ir sparta *FPGA* matricai.

Nešiojamojo Dell kompiuterio procesorius „*Intel Core i7-8750H*“ yra įvertintas 45 W pagal *TDP* ir 35 W pagal *TDP Down*. Tačiau šias galios sąnaudas yra sudėtinga pasiekti. Net paleidus 8 lygiagrečius procesus procesorius išnaudojo 90% visų resursų veikdamas maksimaliu greičiu. Atitinkamai, išnaudojant visus resursus galios taupymo režimu tikėtina, kad procesorius sunaudotų daugiausiai 35 W galios. Tačiau tai yra ryškiai daugiau negu „*Rapsberry Pi*“ ar *FPGA* komponentas.

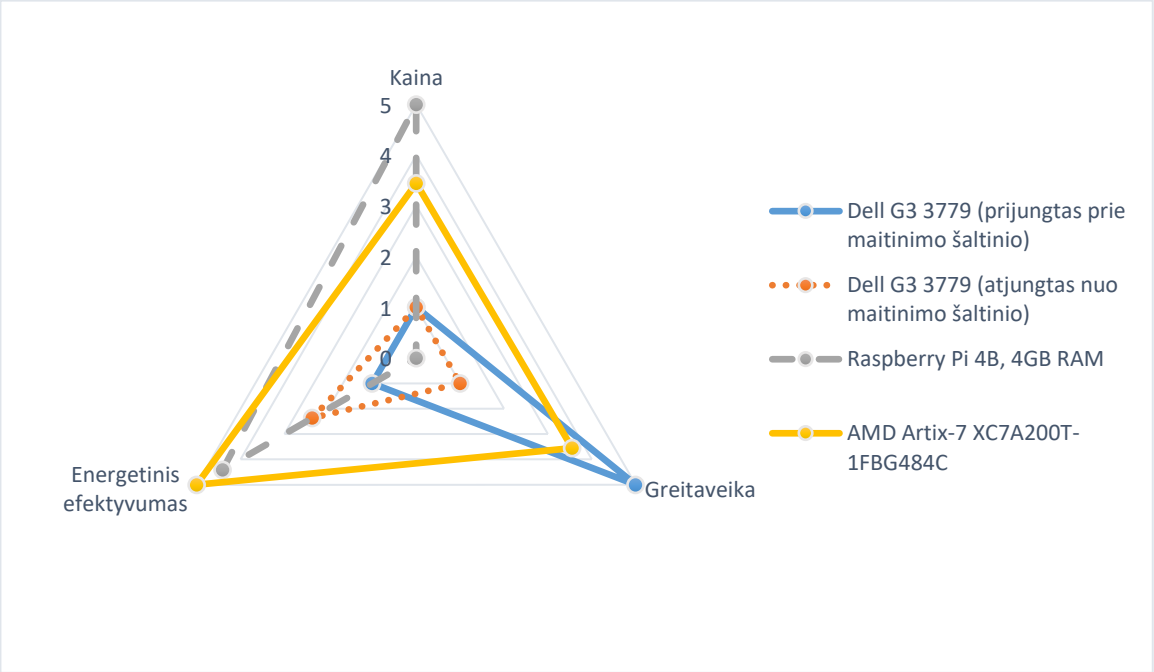
Matosi aiški hierarchija: *FPGA* sunaudoja mažiausiai galios – vos 1 W, „*Raspberry Pi 4B*“ sunaudoja santykinai mažai galios – 7,5 W, nešiojamas kompiuteris veikdamas galios taupymo režimu sunaudoja daugiausiai 30 W ir maksimaliu režimu – 45 W galios. Vertinant greitaveiką akivaizdu, kad *FPGA* neabejotinai efektyviausiai išnaudoja resursus ir teikia geriausią greitaveiką su mažiausiomis galios sąnaudomis. Net jeigu vertintume, kad prireiktų kelių *FPGA* komponentų siekiant kompensuoti „*i7-8750H*“ procesoriaus gebėjimą lygiagrečiai atlikti 8 ar daugiau procesų, *FPGA* energetinis efektyvumas ryškai lenkia procesoriaus efektyvumą. Gauti energetinio efektyvumo rezultatai kartu su kitais rodikliais yra pateikti **8** lentelėje.

8 lentelė. Aparatūros kainos [37, 40, 43], skaičiavimo laiko ir energijos sąnaudų palyginimas (400 spiečiaus dalelių ir 400 iteracijų)

Aparatūra	**Kaina, Eur**	**Skaičiavimo laikas, s**	**Naudojama galia, W**
Dell G3 3779 (prijungtas prie maitinimo šaltinio)	1156	29,40	45
Dell G3 3779 (atjungtas nuo maitinimo šaltinio)	1156	62,46	30
Raspberry Pi 4B, 4GB *RAM*	91	249,94	7,5
AMD Artix-7 XC7A200T-1FBG484C	505	41,36	1

Siekiant rezultatus apibendrinti buvo sukurta radaro tipo diagrama reprezentuojanti aparatūrinių sprendimų savybes: kainą, greitaveiką ir energetinį efektyvumą. Ši diagrama reprezentuoja įrangos santykines savybes normuojant nuo 1 (blogiausia) iki 5 (geriausia). *Raspberry Pi 4B* greitaveika nebuvo vertinama dėl greitaveikos nuokrypio. Ši diagrama suteikia bendrą apžvalgą tirtos aparatūros ir jos privalumų bei trūkumų.

Akivaizdu, kad greitaveikos požiūriu geriausiais sprendimas vis tiek yra nešiojamojo kompiuterio procesorius veikiantis maksimaliu greičiu. Tačiau šis sprendimas nusileidžia visais kitais aspektais, ypač, jeigu veikia galios taupymo režimu. „*Raspberry Pi 4B*“ turi patrauklią kainą ir energetinį efektyvumą, tačiau greitaveika yra išskirtinai prasta. *FPGA* komponentas „*XC7A200T-1FBG484C*“ atrodo kaip puikus kompromisas, per daug neišsiskiriantis, bet tenkinantis visus tris poreikius: konkurencingą kainą, palyginus gerą greitaveiką ir itin gerą energetinį efektyvumą.



22 pav. Santykinis aparatūros kainos, greitaveikos ir energetinio efektyvumo palyginimas penkių balų sistemoje

5. Diskusija

Šiame skyriuje yra aptariama tyrimo apimtis bei aspektai, kurie nebuvo apžvelgti arba kurių buvo pasirinkta neanalizuoti. Taip pat yra trumpai apžvelgiama pagrindiniai iššūkiai su kuriais buvo susidurta realizuojant daiktų interneto taikymo pavyzdį su programuojama logika ir kokia šių iššūkių įtaka sprendimo tinkamumui.

5.1. Rezultatų interpretavimas

Vienas iš šio tyrimo tikslų buvo realizuoti daiktų interneto pakraščio sistemą su programuojama logika. Kaip tyrime pabrėžta – pirmiausia programa buvo sukurta *FPGA* komponentui ir tik vėliau pritaikyta *Linux* operacinei sistemai. Tikėtina, kad pirmiausia pradėjus programą kurti su *Linux* įterptine sistema, programos architektūrą bei sprendimai būtų pasirinkti kitokie bei lemtų šiek tiek kitokius greitaveikos rezultatus. Tačiau tada kiltu problema pritaikant tokią programą su *HLS* programuojamai logikai. Siekiant optimizuoti programą tiek programuojamai logikai, tiek procesoriui, būtų sukurtos skirtingos programos. Taigi buvo keliamas klausimas, ar geriau testuoti kuo panašesnę programą tačiau pritaikytą programuojamai logikai, ar modifikuoti ir iš pagrindų perrašyti programą *C++* kompiliatoriui? Tai reikštų, kad būtų lyginama iš esmės skirtingos programos. Dėl ribotų laiko resursų ir siekiant išlaikyti kuo panašesnes sąlygas buvo pasirinkta naudoti *FPGA* pritaikytą programą. Tačiau perrašius programą galima tikėtis, kad pavyktų panaudoti *OpenMP* biblioteką ar asinchroninius *C++* metodus. Tai leistų išnaudoti kelias procesoriaus šerdis ir geriau įvertinti, procesoriaus greitaveiką atliekant vieną procesą.

Šio tyrimo taikymo pavyzdžio apimtis buvo tik duomenų apdorojimo realizavimas. Literatūros apžvalgoje yra įvardinta, kad reali daiktų interneto pakraščio sistema atlieka šias funkcijas: duomenų surinkimą, apdorojimą ir komunikaciją su kitais prietaisais bei debesų kompiuterija. Šiame tyrime buvo analizuojama tik duomenų apdorojimo dalis. Realizuojant pilną daiktų interneto pakraščio sistemą su programuojama logiką reiktų integruoti ir procesorių atsakingą už duomenų surinkimą, saugojimą ir komunikacijas. Taigi reiktų atlikti daugiau tyrimų, sukuriant pilnai integruotą procesoriaus – programuojamos logikos sistemą, vykdančią visas daiktų interneto pakraščiui būdingas užduotis. Tolimesnių tyrimų tikslas galėtų būti palyginti procesoriaus – programuojamos logikos sistemą su įterptinių sistemų procesoriaus sistema bei debesų kompiuterija paremta sistema analizuojant kokią įtaką bendram sistemos efektyvumui daro programuojamos logikos atliekamas skaičiavimų spartinimas.

5.2. *FPGA* projekto realizavimo procesas

Šiame tyrime nebuvo vertinamas vienas iš kritiškai svarbių faktorių – programos kūrimo laikas. Sistemos tyrimo ir kūrimo laikas galiausiai atsispindi ir yra įskaičiuojamas į galutinę produkto kainą. Kai kuriami daiktų interneto sprendimai yra taikoma plačiai, masto ekonomika leidžia mažiau dėmesio kreipti į programos kūrimui išnaudojamus resursus, nes kiekvienas sutaupytas centas atneša eksponentiškai daugiau pelno. Tačiau jeigu produktas yra nišinis, tai projektavimo ir programavimo laikas sudaro reikšmingą dalį galutinio produkto kainos. Tai aktualu todėl, kad programos kūrimas programuojamai logikai yra gerokai sudėtingesnis procesas reikalaujantis daugiau etapų ir specifinių žinių.

Tyrimo metu buvo skirta eksponentiškai daugiau laiko kuriant ir optimizuojant projektą programuojamai logikai negu pritaikant sukurtą programą Linux įterptinei sistemai. Taip yra todėl, kad kuriant *FPGA* skirtą programą yra reikalingi papildomi etapai:

- nuolatinis resursų optimizavimas;
- nepalaikomų operacijų pakeitimas kitomis;
- *HLS* direktyvų pritaikymas;
- taktinio dažnio ir programos derinimas.

Resursų optimizavimas yra nuolatinis ir pasikartojantis programos kūrimo etapas. Kuriant programuojamos logikos projektą su *HLS* nuolat svarbu sekti, kiek yra panaudojama resursų ir kurios vietos yra problematiškiausios. Iš pirmo žvilgsnio maža problema – skaičiavimas naudojantis šiek tiek daugiau resursų, gali tapti didele problema, kai yra pradedama lygiagretinti skaičiavimus arba jungti į konvejerines struktūras. Todėl nuolat reikia sekti ir vertinti ar pakeitimai nesunaudoja daugiau resursų negu turėtų. Jeigu resursai yra netaupomi, ir operacijos kartojasi, procesorius tiesiog veiks lėčiau, tačiau programuojamos logikos projektas gali nebetilpti į esamus resursus ir gali tapti nebeįmanoma projektą realizuoti su pasirinktu *FPGA* komponentu. Vystant programą taip pat susiduriama su tuo, kad vieni apribojimai pakeičia kitus. Pavyzdžiui realizuojant šio tyrimo pavyzdį iš pradžių buvo optimizuojami *DSP*, vėliau *LUT*, galiausiai įvesties ir išvesties resursai.

Bene daugiausiai laiko pareikalavęs aspektas – fiksuoto kablelio operacijos. Slankiojančio kablelio operacijos naudoja neproporcingai daug *DSP* resursų, todėl yra beveik neišvengiama jas keisti į fiksuoto kablelio operacijas. Tačiau tai smarkiai apriboja programos lankstumą. Programa tampa pritaikyta prie tam tikrų kintamųjų diapazonui ir iš šių apribojimų išėjus – gaunami klaidingi rezultatai. Šis aspektas prailgina programos derinimo etapą, nes reikia įsitikinti, kad visos operacijos tenkina pasirinktą kintamųjų rezoliuciją.

Vystant *HLS* projektą reikalaujantį sudėtingų skaičiavimų yra nuolat susiduriama su operacijomis, kurios yra nepalaikomos *HLS*. Tai gali būti kintamųjų kėlimas laipsniu ar kompleksinių skaičių operacijos. Šie iššūkiai prailgina programos kūrimo etapą, nes reikia ieškoti alternatyvių skaičiavimo metodų ir funkcijų.

Pritaikant *HLS* direktyvas yra atsiremiama į resursų bei aparatūrinius apribojimus. Šiame etape tenka arba modifikuoti pačią programą arba keisti direktyvas taip, kad jos atitiktų aparatūrinius apribojimus. Dažnai tai lemia programos modifikavimą ir grįžimą prie ankstesnių etapų.

Išvados

1) Realizuota daiktų interneto pakraščio sistema su programuojama logika ir *Linux* įterptinėmis sistemomis, leidžianti vertinti aparatūros privalumus ir trūkumus atliekant skaitmeninių signalų apdorojimą ir iteracinius paieškos algoritmus.
2) Programuojamos logikos sistema optimizuota taikant fiksuoto kablelio skaičių formatą ir aukšto lygio sintezei pritaikytas bibliotekas. Taikant konvejerines ir lygiagrečias struktūras, optimizuota programuojamos logikos realizacija ir pasiektas dalelių spiečiaus optimizavimo algoritmo tiesinio laiko sudėtingumas, dalelių ir iteracijų atžvilgiu.
3) Programuojamos logikos greitaveika teikia kompromisą tarp įterptinės sistemos ir debesų kompiuterijos greitaveikos. Tiriant taikymo pavyzdį, programuojama logika yra vidutiniškai 28 % lėtesnė nei *Intel i7-8750H* procesoriaus branduolys, veikiantis maksimaliu greičiu, tačiau 49 % procentais greitesnė, kai procesorius veikia galios taupymo režimu. Programuojama logika yra 510% greitesnė nei *Raspberry Pi 4B* procesoriaus branduolys.
4) Programuojamos logikos energetinis efektyvumas smarkiai lenkia tirtų procesorių energetinį efektyvumą. Vykdant skaičiavimus, FPGA komponentas sunaudoja vos 1,06 W galios. Tai yra 7 kartus mažiau nei *Raspberry Pi 4B* procesorius, gebantis vykdyti 4 lygiagrečius procesus, ir 42 kartus mažiau nei *Intel Core i7-8750H* procesorius, galintis vykdyti 10 lygiagrečių procesų.

Literatūros sąrašas

1. E. Sisinni, A. Saifullah, S. Han, U. Jennehag and M. Gidlund, "Industrial Internet of Things: Challenges, Opportunities, and Directions," in IEEE Transactions on Industrial Informatics, vol. 14, no. 11, pp. 4724-4734, Nov. 2018, doi: 10.1109/TII.2018.2852491.
2. A. Karmakar, N. Dey, T. Baral, M. Chowdhury and M. Rehan, "Industrial Internet of Things: A Review," 2019 International Conference on Opto-Electronics and Applied Optics (Optronix), Kolkata, India, 2019, pp. 1-6, doi: 10.1109/OPTRONIX.2019.8862436.
3. Lionel Sujay Vailshery, „Number of IoT connected devices worldwide 2019-2021, with forecasts to 2030“, Nov 22, 2022; Prieiga per internetą: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/
4. J. P. Tomas, „Thames Water: A Strategic Move into Predictive Planning“, 2017, Prieiga per internetą: https://wiprodigital.com/cases/progressive-metering-a-utilitys-strategic-move-into-predictive-planning/
5. S. S. Mishra and A. Rasool, "IoT Health care Monitoring and Tracking: A Survey," 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), Tirunelveli, India, 2019, pp. 1052-1057, doi: 10.1109/ICOEI.2019.8862763.
6. Muhammad Khan, Rabia Khalid, Sharjeel Anjum, Numan Khan, Seungwon Cho, Chansik Park, „Tag and IoT based safety hook monitoring for prevention of falls from height", Automation in Construction, Volume 136, 2022, 104153, ISSN 0926-5805, https://doi.org/10.1016/j.autcon.2022.104153
7. Luigi Atzori, Antonio Iera, Giacomo Morabito, „The Internet of Things: A survey“, Computer Networks, Volume 54, Issue 15, 2010, Pages 2787-2805, ISSN 1389-1286, https://doi.org/10.1016/j.comnet.2010.05.010.
8. Xu R, Jin W, Kim D. Environment Optimization Scheme Based on Edge Computing Using PSO for Efficient Thermal Comfort Control in Resident Space. Actuators. 2021; 10(9):241. https://doi.org/10.3390/act10090241
9. Serkan Ayvaz, Koray Alpay, Predictive maintenance system for production lines in manufacturing: A machine learning approach using IoT data in real-time, Expert Systems with Applications, Volume 173, 2021, 114598, ISSN 0957-4174, https://doi.org/10.1016/j.eswa.2021.114598.
10. W. Yu et al., "A Survey on the Edge Computing for the Internet of Things," in IEEE Access, vol. 6, pp. 6900-6919, 2018, doi: 10.1109/ACCESS.2017.2778504.
11. I. Zyrianoff, A. Trotta, L. Sciullo, F. Montori and M. Di Felice, "IoT Edge Caching: Taxonomy, Use Cases and Perspectives," in IEEE Internet of Things Magazine, vol. 5, no. 3, pp. 12-18, September 2022, doi: 10.1109/IOTM.001.2200112.
12. Y. Wang, M. Sheng, X. Wang, L. Wang and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling", IEEE Trans. Commun., vol. 64, no. 10, pp. 4268-4282, Oct. 2016.
13. R. P. Weicker, "An overview of common benchmarks," in Computer, vol. 23, no. 12, pp. 65-75, Dec. 1990, doi: 10.1109/2.62094.
14. C. P. Kruger and G. P. Hancke, "Benchmarking Internet of things devices," 2014 12th IEEE International Conference on Industrial Informatics (INDIN), Porto Alegre, Brazil, 2014, pp. 611-616, doi: 10.1109/INDIN.2014.6945583

15. D. Lee, S. Lee, S. Oh and D. Park, "Energy-Efficient FPGA Accelerator With Fidelity-Controllable Sliding-Region Signal Processing Unit for Abnormal ECG Diagnosis on IoT Edge Devices," in IEEE Access, vol. 9, pp. 122789-122800, 2021, doi: 10.1109/ACCESS.2021.3109875.
16. N. Sulaiman, Z. Obaid, M. H. Marhaban, M. N. Hamidon, „Design and Implementation of FPGA-Based Systems -A Review“. Australian Journal of Basic and Applied Sciences. 3, 2009.
17. C. Xu et al., "The Case for FPGA-Based Edge Computing," in IEEE Transactions on Mobile Computing, vol. 21, no. 7, pp. 2610-2619, 1 July 2022, doi: 10.1109/TMC.2020.3041781.
18. R. Ferdian, R. Aisuwarya and T. Erlina, "Edge Computing for Internet of Things Based on FPGA," 2020 International Conference on Information Technology Systems and Innovation (ICITSI), 2020, pp. 20-23, doi: 10.1109/ICITSI50517.2020.9264937.
19. C. Zhao, C. Xiao and Y. Liu, "A Real-time Reconfigurable Edge computing System in Industrial Internet of Things Based on FPGA," 2021 IEEE 16th Conference on Industrial Electronics and Applications (ICIEA), 2021, pp. 480-485, doi: 10.1109/ICIEA51954.2021.9516225.
20. B. Zhou, M. Egele and A. Joshi, "High-performance low-energy implementation of cryptographic algorithms on a programmable SoC for IoT devices," 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2017, pp. 1-6, doi: 10.1109/HPEC.2017.8091062.
21. J. Hochstetler, R. Padidela, Q. Chen, Q. Yang and S. Fu, "Embedded Deep Learning for Vehicular Edge Computing," 2018 IEEE/ACM Symposium on Edge Computing (SEC), Seattle, WA, USA, 2018, pp. 341-343, doi: 10.1109/SEC.2018.00038.
22. Intel Neural Compute Stick 2 NCS dirbtinio intelekto skaičiavimų akseleratoriaus specifikacija ir aprašas, Prieiga per internetą: https://www.intel.com/content/www/us/en/developer/articles/tool/neural-compute-stick.html
23. S. Y. Nikouei, Y. Chen, S. Song, R. Xu, B. -Y. Choi and T. R. Faughnan, "Real-Time Human Detection as an Edge Service Enabled by a Lightweight CNN," 2018 IEEE International Conference on Edge Computing (EDGE), San Francisco, CA, USA, 2018, pp. 125-129, doi: 10.1109/EDGE.2018.00025.
24. B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas and Q. Zhang, "Edge Computing in IoT-Based Manufacturing," in IEEE Communications Magazine, vol. 56, no. 9, pp. 103-109, Sept. 2018, doi: 10.1109/MCOM.2018.1701231.
25. T. Gizinski and X. Cao, "Design, Implementation and Performance of an Edge Computing Prototype Using Raspberry Pis," 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), 2022, pp. 0592-0601, doi: 10.1109/CCWC54503.2022.9720848.
26. A. Das, S. Patterson and M. Wittie, "EdgeBench: Benchmarking Edge Computing Platforms," 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 175-180, doi: 10.1109/UCC-Companion.2018.00053
27. P. Tervydis, L. Svilainis, Ž. Nakutis, A. R. Martínez, "CloudEdgeAssetOptimizer: Tool to optimize the Cloud-Edge computing network resources at given requirements of processing delay, battery capacity and cost", SoftwareX, Volume 26, 2024, 101714, ISSN 2352-7110, https://doi.org/10.1016/j.softx.2024.101714.

28. L. Svilainis et al., "Air-Coupled Ultrasound Resonant Spectroscopy Sensitivity Study in Plant Leaf Measurements," 2021 IEEE Sensors, Sydney, Australia, 2021, pp. 1-4, doi: 10.1109/SENSORS47087.2021.9639612.
29. L. M. Brekhovskikh, O. A. Godin, Acoustics of Layered Media I. Plane and Quasi-Plane Waves. Springer-Verlag, 1990.
30. Gad, A.G. „Particle Swarm Optimization Algorithm and Its Applications: A Systematic Review“. Arch Computat Methods Eng 29, 2531–2561 (2022). https://doi.org/10.1007/s11831-021-09694-4
31. Vitis HLS klaidų dokumentacija, planavimo problema. Prieiga per internetą: https://docs.amd.com/r/2023.1-English/ug1448-hls-guidance/Unable-to-Schedule
32. Vitis HLS masyvų skaidymo dokumentacija. Prieiga per internetą: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Array-Partitioning
33. „Xilinx“ HLS vartotojam skirta dokumentacija apie FFT algoritmą. Prieiga per internetą: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/FFT-IP-Library
34. Oilerio kosinuso identiteto formulė. Prieiga per internetą: https://proofwiki.org/wiki/Cosine_of_Complex_Number
35. Oilerio sinuso identiteto formulė. Prieiga per internetą: https://proofwiki.org/wiki/Sine_of_Complex_Number
36. AMD Artix-7 FPGA komponento XC7A200T-1FBG484C katalogas. Prieiga per internetą: https://www.digikey.lt/en/products/detail/amd/XC7A200T-1FBG484C/3925796
37. Digilent Nexys Video Artix-7 FPGA programavimo mokymui skirta plokštė su Xilinx Artix-7 XC7A200T-1SBG484C FPGA komponentu. Prieiga per internetą: https://digilent.com/shop/nexys-video-artix-7-fpga-trainer-board-for-multimedia-applications/
38. M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," in Proceedings of the IEEE, vol. 93, no. 2, pp. 216-231, Feb. 2005, doi: 10.1109/JPROC.2004.840301 Bibliotekos prieiga per internetą: https://www.fftw.org/
39. Dell G3 3779 nešiojamojo kompiuterio dokumentacija. Prieiga per internetą: https://www.dell.com/support/manuals/en-lt/g-series-17-3779-laptop/dell-g3-3779-setupandspecifications/
40. Raspberry Pi 4 techninė specifikacija, Prieiga per internetą: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
41. Intel Core i7-8750H procesoriaus specifikacija ir rekomenduojama kaina. Prieiga per internetą: https://ark.intel.com/content/www/us/en/ark/products/134906/intel-core-i7-8750h-processor-9m-cache-up-to-4-10-ghz.html
42. Raspbrerry Pi 4B procesoriaus BCM2711 specifikacija. Prieiga per internetą: https://www.cpu-monkey.com/en/cpu-raspberry_pi_4_b_broadcom_bcm2711
43. Dell G3 3779 nešiojamojo kompiuterio rinkos kaina. Prieiga per internetą: https://www.kilobaitas.lt/kompiuteriai_ir_komponentai/nesiojami_kompiuteriai/nesiojami_kompiuteriai_notebook/dell_g3_17_3779_black_173_quot_ips_full_hd_1920_x_1080_pixels_matt_intel_core_i7_i7-8750h_8_gb_dd/product.aspx?itemid=366665

Priedai

1 priedas. Sukurtas *HLS C* kalbos kodas realizuojantis *PSO* algoritmą

```c
/* INCLUDE */
/*____________________________________________________________________________*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>

/* DEFINE */
/*____________________________________________________________________________*/
#define SWARM_SIZE 20
#define DIMENSION 2
#define MAX_ITERATIONS 20
#define INERTIA_WEIGHT 0.5
#define PERSONAL_WEIGHT 1
#define GLOBAL_WEIGHT 1
#define DIMENSION_MAX 1000

/* TYPEDEF */
/*____________________________________________________________________________*/

typedef struct sParticle_t {
   float position[DIMENSION];
   float position_best[DIMENSION];
   float velocity[DIMENSION];

   float current_fitness;
   float best_fitness;
} sParticle_t;

/* PROTOTYPES */
/*____________________________________________________________________________*/

/* FUNTIONS */
/*____________________________________________________________________________*/

/* y = x^2 + y^2 */
float Fitness(float x, float y) {
   return x * x + y * y;
}

/* Xorshift32 */
static uint32_t RNG_seed = 12345;
float PRNG() {
   uint32_t x = RNG_seed;
   x ^= x << 13;
   x ^= x >> 17;
   x ^= x << 5;
   RNG_seed = x;
   float randomFloat = (float) x / UINT32_MAX;
   return randomFloat;
}

/* default */
float RNG_default(void) {
   return (float) rand() / RAND_MAX;
```

```
}

/* PSO FUNCTIONS */

void swarm_init(sParticle_t swarm[SWARM_SIZE]) {
   for (int i = 0; i < SWARM_SIZE; i++) {
      for (int j = 0; j < DIMENSION; j++) {
         swarm[i].position[j] = PRNG() * DIMENSION_MAX; // [0 - DIMENSION_MAX]
         swarm[i].position_best[j] = swarm[i].position[j];
         swarm[i].velocity[j] = 0.0;
      }
      swarm[i].current_fitness = Fitness(swarm[i].position[0], swarm[i].position[1]);
      swarm[i].best_fitness = swarm[i].current_fitness;
   }
}

void swarm_update(sParticle_t swarm[SWARM_SIZE], float globalBest[DIMENSION]) {
   float rand1 = 0;
   float rand2 = 0;
   for (int i = 0; i < SWARM_SIZE; i++) {
      // Update particle parameters
      for (int j = 0; j < DIMENSION; j++) {
         rand1 = PRNG(); // [0-1]
         rand2 = PRNG(); // [0-1]

         swarm[i].velocity[j] = INERTIA_WEIGHT * swarm[i].velocity[j] +
         PERSONAL_WEIGHT * rand1 * (swarm[i].position_best[j] - swarm[i].position[j])
+
         GLOBAL_WEIGHT * rand2 * (globalBest[j] - swarm[i].position[j]);

         swarm[i].position[j] += swarm[i].velocity[j];
      }

      swarm[i].current_fitness = Fitness(swarm[i].position[0], swarm[i].position[1]);
      if (swarm[i].current_fitness < swarm[i].best_fitness) {
         swarm[i].best_fitness = swarm[i].current_fitness;
         for (int j = 0; j < DIMENSION; j++) {
            swarm[i].position_best[j] = swarm[i].position[j];
         }
      }
   }
}

void PSO_HLS(float* position_X, float* position_Y, float *fitness, const char
itterations) {
   static sParticle_t swarm[SWARM_SIZE];
   static float global_best[DIMENSION];
   static float global_best_fitness = 9999;

   swarm_init(swarm);

   for (int iter = 0; iter < itterations; iter++) {

      // Update the global best
      for (int i = 0; i < SWARM_SIZE; i++) {
         if (swarm[i].best_fitness < global_best_fitness) {
            global_best_fitness = swarm[i].best_fitness;
            for (int j = 0; j < DIMENSION; j++) {
               global_best[j] = swarm[i].position_best[j];
            }
         }
      }
```

```
        swarm_update(swarm, global_best);

        /* Update output */
        *position_X = global_best[0];
        *position_Y = global_best[1];
        *fitness = global_best_fitness;
    }
}
```

2 priedas. Sukurtas *HLS C* kalbos kodas realizuojantis *PSO* algoritmo testavimą

```c
/* INCLUDE */
/*_____________________________________________________________________*/
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>

/* DEFINE */
/*_____________________________________________________________________*/
#define SWARM_SIZE 5
#define DIMENSION 2
#define MAX_ITERATIONS 20
#define INERTIA_WEIGHT 0.5
#define PERSONAL_WEIGHT 1
#define GLOBAL_WEIGHT 1
#define DIMENSION_MAX 1000

/* PROTOTYPES */
/*_____________________________________________________________________*/
float Fitness(double x, double y);
/* FUNTIONS */
/*_____________________________________________________________________*/
int main(int argc, char *argv[]) {

   float position_X = 10;
   float position_Y = 10;
   float fit = 99999;


   for (int i = 0; i < 100; i = i + 5) {
      PSO_HLS(&position_X, &position_Y, &fit, i);
      printf("Iter%3d: Fit:%.4f  Pos:{%.3f;%.3f}\n", i, fit, position_X, position_Y);
   }

   return 0;
}
```

3 priedas. Sukurtas *Vitis HLS* projektas

Prieiga per internetą: https://github.com/MindaugasPunys/FPGA_PSO_Brekhovski_model

4 priedas. *Matlab* kodas skirtas modeliuoti ultragarsinį signalą

Prieiga per internetą: https://github.com/MindaugasPunys/Matlab_Brekhovski_model_sim

5 priedas. Sukurtas *Cmake* projektas skirtas *Linux* ir *WSL* platformoms

Prieiga per internetą: https://github.com/MindaugasPunys/Linux_PSO_Brekhovski_model