VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS
PROGRAMŲ SISTEMŲ BAKALAURO STUDIJŲ PROGRAMA

Lygiagretizuota greitoji Furje transformacija naudojant CUDA

Parallel Fast Fourier Transform using CUDA

Bakalauro baigiamasis darbas

Atliko:	Dominykas Prievelis	(parašas)
Darbo vadovas:	dr. Rokas Astrauskas	(parašas)
Darbo recenzentas:	Irus Grinis	(parašas)

Vilnius – 2022

Santrauka

Greitoji Furje transformacija yra palankesnei sudėtingumo laiko atžvilgiu grupei priklausanti algoritmų šeima, leidžianti paskaičiuoti diskrečiąją Furje transformaciją kur kas greičiau nei matematinė transformacijos formuluotė leistu manyti. Be to, greitoji Furje transformacija pasižymi savybėmis, leidžiančiomis ją lygiagretizuoti. Kadangi šiuolaikiniai grafiniai procesoriai leidžia pasiekti itin didelį instrukcijų pralaidumą ir atminties greitį - grafiniais procesoriais lygiagretizuota greitoji Furje transformacija galės būti atliekama dar greičiau.

CUDA lygiagretaus progamavimo modelio pagalba lygiagretizuoti pagrindo 2 Cooley-Tukey ir pagrindų 2, 4, 8, 16 Stockham algoritmai pasižymi skirtingomis savybėmis bei apribojimais įeities duomenims. Teoriniai rezultatai parodė, jog pagrindo 16 Stockham algoritmas turi didžiausius apribojimus įeities duomenims, tačiau eksperimentiai rezultatai taip pat parodė, jog tai yra našiausias iš sukurtų realizacijų. Ši algoritmo versija taip pat yra greitesnė už industrijoje plačiai naudojamą biblioteką FFTW bei gali pasilyginti su Nvidia grafinių procesorių biblioteka CuFFT.

Raktiniai žodžiai: diskrečioji Furje transformacija, greitoji Furje transformacija, grafiniai procesoriai, CUDA.

Summary

Fast Fourier transform is a family of algorithms, belonging to a more favorable time complexity group and allows for much faster calculation of discrete Fourier transform. Fast Fourier transform also has properties which allow for parallel implementations. As current graphics processors provide high instruction throughput and memory bandwidth - graphics processor accelerated fast Fourier transform can be made even faster.

Using CUDA parallel programming model parallel radix-2 Cooley-Tukey and parallel radix-2, radix-4, radix-8 and radix-16 Stockham algorithms were implemented. These algorithms share different properties and constraints for input data. Theoretical results showed that radix-16 Stockham algorithm has the strictest constraints for input data, however, experimental results showed that it's also the fastest created implementation. This version of the algorithm is faster than widely used FFTW and can compare itself with Nvidias CuFFT library.

Keywords: discrete Fourier transform, fast Fourier transform, graphics processors, CUDA.

TURINYS

Įvadas

Greitoji Furje transformacija (toliau - FFT) yra klasė efektyvių algoritmų diskrečiajai Furje transformacijai (toliau - DFT) paskaičiuoti. FFT laikomas vienu aktualiausių XX amžiaus algoritmų dėl itin plataus pritaikymo spektro. FFT yra panaudojamas daugianarių aritmetikoje, diferencialinių lygčiu sprendime, garso analinėje, radaro signalo analizėje, vaizdo apdorojime ir t.t.

Tuo tarpu šiuolaikiniai grafiniai procesoriai (toliau - GPU) yra pajėgūs pasiekti kur kas didesnį instrukcijų pralaidumą (angl. *instruction throughput*) ir atminties greitį (angl. *memory bandwidth*) nei panašios kainos ir energijos sąnaudų bendro naudojimo procesoriai (toliau - CPU) [NBG+08; Nvi22b]. Šios grafinių procesorių charakteristikos, kartu su jų teikiamu masinio lygiagretumo (angl. *massive parallelism*) savybėmis ir nuolat prieinamesnėmis programavimo paradigmomis (pvz., CUDA) [Nvi22a] leidžia efektyviau spręsti vis platesnį spektrą lygiagretizuojamų uždavinių. Greitoji Furje transformacija yra vienas iš tokių uždavinių, todėl CUDA paremtos realizacijos leidžia pasiekti kur kas geresnius rezultatus lyginant su nuosekliomis ar lygiagrečiomis CPU realizacijomis [GLD+08; LBG08; Tak19].

Darbo tikslas - išnagrinėti skirtingus FFT algoritmus palyginant jų privalumus bei trūkumus GPU lygiagretizavimo kontekste bei sukurti CUDA lygiagretaus programavimo modeliu grįstas algoritmų realizacijas palyginant realizacijų efektyvumą tarpusavyje bei su egzistuojančiais sprendimais.

Keliami uždaviniai ir laukiami rezultatai:

1. Išanalizuoti bendros atminties (angl. *shared memory*) modeliu pagrįstų FFT algoritmų ir jų realizacijų literatūrą siekiant identifikuoti GPU lygiagretizavimui tinkamus algoritmus.
2. Išanalizuoti ir tarpusavyje palyginti kelis identifikuotus FFT algoritmus (ar jų variacijas) siekiant identifikuoti algoritmų pranašumus/trūkumus vienas kito atžvilgiu bei įvertinti skirtingų algoritmų priklausomybę nuo įeities duomenų.
3. Sukurti CUDA modeliu grįstas pasirinktų algoritmų realizacijas siekiant apžvelgti CUDA programavimo modelio gerąsias praktikas bei įvertinti CUDA grįsto FFT realizavimo principus.
4. Palyginti sukurtas FFT realizacijas tarpusavyje bei su egzistuojančiais sprendimais siekiant įvertinti sukurtų realizacijų savybes ir efektyvumą.

1. Furje transformacija

Periodinė (laiko) funkcija $g(t)$ gali būti dekomponuojama į sumą (eilutę) sinusoidų su specifiniais dažniais ir amplitudėmis. Tokį dekomponavimą XIX a. pradžioje sukūrė Žanas Baptistas Furje (*Jean Baptiste Joseph Fourier*) ir pavadino Furje eilute. Tolydžioji Furje transformacija yra išvedama iš Furje eilutės, o jos išeitis yra tolydi dažnių funkcija $\hat{g}(f)$ [Wil05].

1.1. Furje eilutė

Furje eilutė periodinei funkcijai $g(t)$ gali būti išreikšta kaip suma sinusoidų su specifiniais dažniais ir amplitudėmis [ER83]:

$$g(t) = \frac{a_0}{2} + \sum_{f=1}^{\infty} \left(a_f \cos\left(\frac{2\pi f t}{T}\right) + b_f \sin\left(\frac{2\pi f t}{T}\right)\right) \tag{1}$$

Čia T - periodas, o a_f, $b_f \in \mathbb{R}$ - f-tieji Furje koeficientai.

Lygtis (1) gali būti užrašyta trumpiau, naudojant tik vieną Furje koeficientą [ER83]:

$$g(t) = \sum_{f=-\infty}^{\infty} \hat{g}(f) e^{\frac{2\pi i f t}{T}} \tag{2}$$

Kur $\hat{g}(f) \in \mathbb{C}$ yra f-asis Furje koeficientas kompleksinėje formoje.

1.2. Tolydžioji Furje transformacija

Skyrelyje 1.1 apibrėžta Furje eilutė leidžia periodinę funkciją $g(t)$ pervesti iš laiko apibrėžimo srities (angl. *time domain*) į dažnio apibrėžimo sritį (angl. *frequency domain*).

Lygtyje (2) apibrėžta Furje eilutė gali būti transformuota į integralą [ER83]:

$$g(t) = \int_{-\infty}^{\infty} \hat{g}(f) e^{2\pi i f t} df \tag{3}$$

Tuomet $\hat{g}(f)$ gali buti išreikšta taip:

$$\hat{g}(f) = \int_{-\infty}^{\infty} g(t) e^{-2\pi i f t} dt \tag{4}$$

Čia $\hat{g}(f)$ - funkcijos $g(t)$ tolydžioji Furje transformacija, $i^2 = -1$.

1.3. Diskrečioji Furje transformacija

Praktiniuose taikymuose analitinė (simbolinė) nagrinėjamos funkcijos $g(t)$ išraiška dažniausiai nėra žinoma. Tokiais atvėjais egzistuoja baigtinė aibė taškų $t_0, t_1, \cdots, t_{N-1}$ ir yra žinomos $g(t)$ reikšmes tuose taškuose $g_0, g_1, \cdots, g_{N-1}$. Tokiu atveju galima nagrinėti tik baigtinį dažnių intervalą, apibrėžus diskrečiają Furje transformaciją taip [ER83]:

$$\hat{g}_f = \sum_{n=0}^{N-1} g_n e^{\frac{-2\pi i f n}{N}}, \;\; 0 \leq f \leq N-1 \tag{5}$$

Taigi, pritaikius diskrečiąją Furje transformaciją (toliau - DFT), iš aibės realių arba kompleksinių įeities taškų $g_0, g_1, \cdots, g_{N-1}$ yra gaunama aibė kompleksinių išeities taškų $\hat{g}_0, \hat{g}_1, \cdots, \hat{g}_{N-1}$.

Kiekvienas išeities taškas $\hat{g}_f$ teikia informacija apie du dalykus. $|\hat{g}_f|$ nusako dažnio intervalo (angl. *frequency bin*) amplitudę, o arg $\hat{g}_f$ - fazės kampą θ. Dažnio intervalo dydis lygus F_s/N, kur F_s - taškų $g_0, g_1, \cdots, g_{N-1}$ parinkimo dažnis (angl. *sampling frequency*). Fazės kampas θ nusako vieno iš signalą g_n sudarančių sinusoidų poslinkį nuo 0 [MV17].

1.3.1. Sudėtingumas laiko atžvilgiu

Lygtyje (5) kompleksiniai eksponentiniai nariai $e^{-2\pi i f/N}$, kur $0 \leq f \leq N-1$ gali būti paskaičiuojami iš anksto, kadangi jų periodas yra N. Tačiau vistiek reikės paskaičiuoti N^2 kompleksinių sąndaugų ir $N(N-1)$ kompleksinių sudėčių. Taigi sudėtingumas laiko atžvilgiu skaičiuojant DFT tiesiogiai pagal formulę yra $O(N^2)$, kur N - elementų skaičius.

2. Greitoji Furje transformacija

Greitoji Furje transformacija (toliau - FFT) yra klasė efektyvių algoritmų diskrečiajai Furje transformacijai (DFT) paskaičiuoti. Visi FFT algoritmai papuola po $O(N \log N)$ sudėtingumo laiko atžvilgiu klase. Šis sudėtingumas yra pagrindinė FFT populiarumo ir aktualumo priežastis, taip pat leidusi taip plačiai panaudoti DFT kaip matematinę operaciją programinėje įrangoje.

Pirmosios FFT užuominos randamos nepublikuotuose Karlo Frydricho Gauso (vok. *Johann Carl Friedrich Gauß*) darbuose 1805 metais [HJB84], tačiau FFT tapo plačiai žinoma sąvoka deka Jameso Williamo Cooley (angl. *James William Cooley*) ir Johno Wilderio Tukey (angl. *John Wilder Tukey*) publikacijos, išleistos 1965 metais [CT65].

Nuo to laiko pasaulyje atsirado daug įvairių FFT versijų, besiskirenčių realizacijos sudėtingumu, apribojimais įeities duomenims, reikalingu atminties kiekiu, efektyvumų.

Šiame darbe bus nagrinėjami pagrindo-2 (angl. *radix*) mažėjimo laike (angl. *decimation in time*) Cooley-Tukey, Stockham algoritmai. Taip pat bus apžvelgiami aukštesnių pagrindų algoritmai nagrinėjant pagrindo-4 algoritmą kaip pavyzdį.

2.1. Cooley-Tukey FFT

Cooley-Tukey FFT yra pirmasis ir vienas aktualiausių FFT algoritmų (didžioji dalis FFT versijų ir variacijų remiasi Cooley-Tukey algoritmo idėjomis ir/ar savybėmis) [Tak19].

Pagrindo-2 Cooley-Tukey FFT algoritme daroma prielaida, jog duomenų skaičius N tenkina sąlygą $N = 2^L$, $L \in \mathbb{N}$.

Kompleksinis keoficientas apibrėžiamas kaip $\omega_N = \exp(-2\pi i/N)$. Algoritmo išvedimas remiasi sinusoidų savybėmis - simetriškumu ir periodiškumu:
Simetriškumas:

$$\omega_N^{f(N-n)} = \omega_N^{-fn} \tag{6}$$

arba

$$\omega_N^f = -\omega_N^{f+N/2} \tag{7}$$

Periodiškumas:

$$\omega_N^{fn} = \omega_N^{f(N+n)} = \omega_N^{(f+N)n} = \omega_N^{(fn \mod N)} \tag{8}$$

arba

$$\omega_{N/2}^f = \omega_{N/2}^{f+N/2} \tag{9}$$

Tuomet lygtyje (5) N elementų DFT išraiška gali būti padalinama į dvi $N/2$ elementų DFT išraiskas - lyginių ir nelyginių indeksų sumas [Wil05]:

$$\hat{g}_f = \sum_{n=0}^{N/2-1} g_{2n} e^{\frac{-2\pi i f(2n)}{N}} + \sum_{n=0}^{N/2-1} g_{2n+1} e^{\frac{-2\pi i f(2n+1)}{N}} =$$
$$\sum_{n=0}^{N/2-1} g_{2n} e^{\frac{-2\pi i f n}{N/2}} + \sum_{n=0}^{N/2-1} g_{2n+1} e^{\frac{-2\pi i f(n+1/2)}{N/2}} =$$
$$\sum_{n=0}^{N/2-1} g_{2n} e^{\frac{-2\pi i f n}{N/2}} + (e^{\frac{-2\pi i f}{N}}) \sum_{n=0}^{N/2-1} g_{2n+1} e^{\frac{-2\pi i f n}{N/2}} =$$
$$\sum_{n=0}^{N/2-1} g_{2n} \omega_{N/2}^{fn} + \omega_N^f \sum_{n=0}^{N/2-1} g_{2n+1} \omega_{N/2}^{fn}$$

Dabar lyginiu ir nelyginių sumų kompleksinė eksponentė yra ta pati išraiška $\omega_{N/2}^{fn}$. Dėl (8) lygybėje išreikšto periodiškumo, eksponenčių reikšmės pasikartoja po $f = N/2$ (tai reiškia, jog sumų reikšmės pasikartoja po $f = N/2$). O dėl (6) lygybėje išreikšto simetriškumo, sumų išorėje esanti eksponentė ω_N^f taip pat pasikartos po $f = N/2$, tik su minuso ženklu.

Šios sumos gali būti toliau dekomponuojamos į dvi dydžio $N/4$ sumas (kiekvieną $N/2$ sumą atskirai dalinant į lyginių ir nelyginių indeksų sumą su naujais indeksais). Toks dekomponavimas kiekvieną karta vidutiniškai sumažina DFT paskaičiuoti reikalingų operacijų skaičių per puse ir gali tęstis $L = \log_2 N$ kartų, iki kol kiekviena suma atitinks po lygiai vieną elementą. T.y., $N/2, N/4, \cdots, N/2^{L-1}, N/2^L$.

Pavyzdžiui, jei $N = 4$, DFT gali būti dekomponuojama $\log_2 4 = 2$ kartus:

$$\hat{g}_f = \sum_{n=0}^{N/2-1} g_{2n} e^{\frac{-2\pi i f n}{N/2}} + e^{\frac{-2\pi i f}{N}} \sum_{n=0}^{N/2-1} g_{2n+1} e^{\frac{-2\pi i f n}{N/2}} =$$
$$\Big(\sum_{n=0}^{N/4-1} g_{4n} e^{\frac{-2\pi i f n}{N/4}} + e^{\frac{-4\pi i f}{N}} \sum_{n=0}^{N/4-1} g_{4n+2} e^{\frac{-2\pi i f n}{N/4}} \Big) +$$
$$\Big(\sum_{n=0}^{N/4-1} g_{4n+1} e^{\frac{-2\pi i f n}{N/4}} + e^{\frac{-4\pi i f}{N}} \sum_{n=0}^{N/4-1} g_{4n+3} e^{\frac{-2\pi i f n}{N/4}} \Big)$$

2.1.1. Mažėjimas laike ir mažėjimas dažnyje

Cooley-Tukey pagrindu grįsti FFT algoritmai dažnai skirstomi į dvi kategorijas - mažėjimo laike (angl. *decimation in time*) ir mažėjimo dažnyje (angl. *decimation in frequency*). Kiekviena iš šių kategorijų atitinka būda kaip į atskiras mažesnes sumas yra dalinama pagrindinė DFT suma:

- Mažėjimas laike (toliau - DIT) - N elementų DFT suma dalinama į mažesnes N/m elementų DFT sumas renkantis elementus žingsniais mn, $mn + 1$, $mn + 2$, $\cdots$, $mn + (m - 1)$. Čia m - algoritmo pagrindas. Pagrindo-2 Cooley-Tukey mažėjimo laike atveju $m = 2$, todėl pasirenkami lyginiai ir nelyginiai elementai.
- Mažėjimas dažnyje (toliau - DIF) - N elementų DFT suma dalinama į mažesnes N/m elementų DFT sumas pasirenkant elementus intervalais $[0, N/m - 1]$, $[N/m, 2N/m - 1]$, $\cdots$, $[(m - 1)N/m, mN/m - 1]$. Čia m - algoritmo pagrindas.

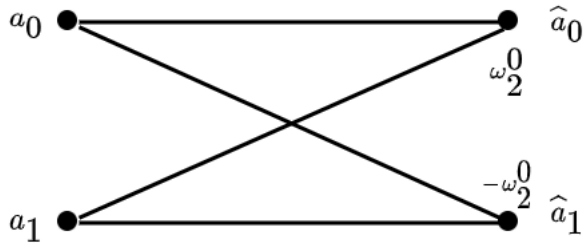
Šiame darbe mažėjimo dažnyje versijos nenagrinėjamos, kadangi skirtumai tarp mažėjimo dažnyje ir mažėjimo laike dažniausiai aktualūs kuriant specifines realizacijas specifinėms architektūroms, tuo tarpu matematinės savybės tarp versijų nesiskiria [Tak19].

2.1.2. FFT operacijų diagrama

Literatūroje gana populiaru vaizduoti FFT atliekamas operacijas drugelio diagramų (angl. *butterfly diagram*) pagrindu [ER83; Tak19; Wil05].

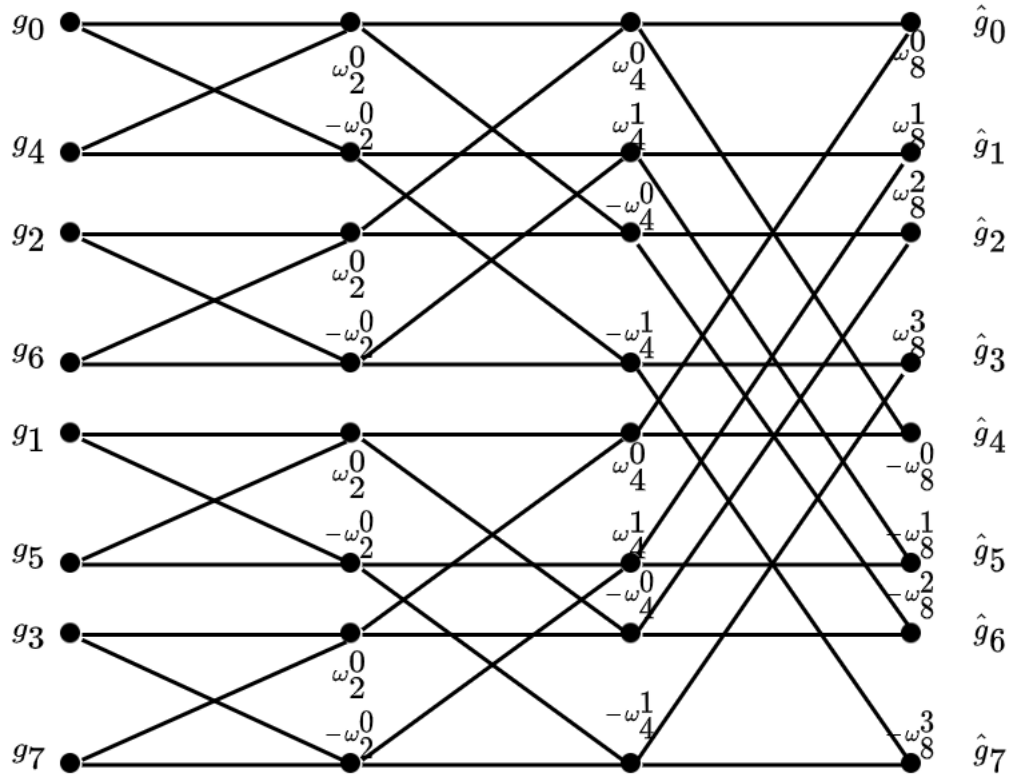
Pavyzdžiui, Pav. 1 pavaizduota diagrama atitinka vieną drugelio operaciją, kuri išreiškiama lygtimis:

$$\hat{a}_0 = a_0 + \omega_2^0 a_1$$
$$\hat{a}_1 = a_0 - \omega_2^0 a_1$$



1 pav. Drugelio (angl. *butterfly*) operacijos diagrama

Tuomet $N = 8$ Cooley-Tukey FFT atliekamos operacijos gali būti pavaizduotos Pav. 2.



2 pav. $N = 8$ Pagrindo-2 Cooley-Tukey FFT operacijų diagrama

2.1.3. Bitų atvirkštinis perrūšiavimas

Pav. 2 kairėje pavaizduotų įeities elementų eilės tvarka neatitinka pradinės įeities eilės tvarkos. Taip atsitinka dėl to, nes kiekviena $N/2$ suma yra toliau rekursyviai dalinama į dvi $N/4$ sumas naudojant naujus indeksus, kurios toliau dalinamos į $N/8$ sumas ir t.t. Tai reiškia, jog nekeičiant įeities elementų eilės tvarkos, išeities elementai bus išsidėstę jų indeksų apversta bitų eilės tvarka (angl. *bit-reverse order*). T.y., jei kiekvieno elemento indeksą paversime dvejetaine jo išraiška ir apversime bitus - gautas indeksas reprezentuos naują poziciją. $N = 8$ pavyzdys matomas lentelėje 1.

Dėl šios priežasties, įeities elementams yra atliekamas bitų atvirktštinis perrūšiavimas, tam, kad kiekvienas išeities elementas $\hat{g}_0$, $\hat{g}_1$,, $\cdots$, $\hat{g}_{N-1}$ atitiktų reikiamą ieities elementą g_0, g_1,, $\cdots$, g_{N-1}.

Šis įeities taškų perrūšiavimo poreikis yra Cooley-Tukey algoritmui būdinga savybė. Dažniausiai būtinybė atlikti perrūšiavimą sukelia neigiamus efektus ir istoriškai iš Cooley-Tukey sekę algoritmai vengia šio perrūšiavimo būtinybės [Tak19]. Vis dėlto, ši savybė leidžia kiekvienos drugelio operacijos tarpinius rezultatus saugoti toje pačioje vietoje, kur yra saugoimi įeities duomenys (nereikia papildomos atminties).

1 lentelė. $N = 8$ elementų indeksų bitų atvirkštinis perrūšiavimas

Įeities elemento indeksas	Dvejetainis indeksas	Bitais apverstas dvejetainis indeksas	Išeities elemento indeksas
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

2.1.4. Operacijų skaičius

Pagrindo-2 algoritmo atveju dalinimas tęsiasi $l \in [1, \log_2 N]$ kartų. Tuomet kiekviename l žingsnyje yra atliekamos $N/2$ drugelio operacijos. Kiekviena drugelio operacija susideda iš 1 kompleksinės sandaugos ir 2 kompleksinių sudėčių. Taigi, bendras operacijų skaičius susideda iš $N/2 \log_2 N$ kompleksinių sandaugų ir $N \log_2 N$ kompleksinių sudėčių.

- Įeities duomenys perrašomi su išeities duomenimis. Tai reiškia, jog nereikia papildomos atminties išskyrimo ir valdymo.
- Atminties prieiga yra žingsniuota (angl. *strided*) ir nėra tolydi (angl. *continious*). Taip nutinka dėl butinybės altikti bitų atvirkštinį perrūšiavimą ir dėl atliekamų drugelio operacijų, kurios kiekviename $l \in [1, \log_2 N]$ žingsnyje sieja elementus nutolusius per 2^{l-1}.
- Nvidia grafiniai procesoriai turi vidines operacijas (angl. *intrinsic operations*) *brev*, *brevll* 32 ir 64 bitų skaičiaus bitų atvirkštiniam atitikmeniui rasti. Vidinės GPU operacijos yra labai

greitos, todėl algoritmą vykdant lygiagrečiai galima šiek tiek sumažinti vieno iš didžiausių algoritmo trūkumų efektą [Nvi22b].

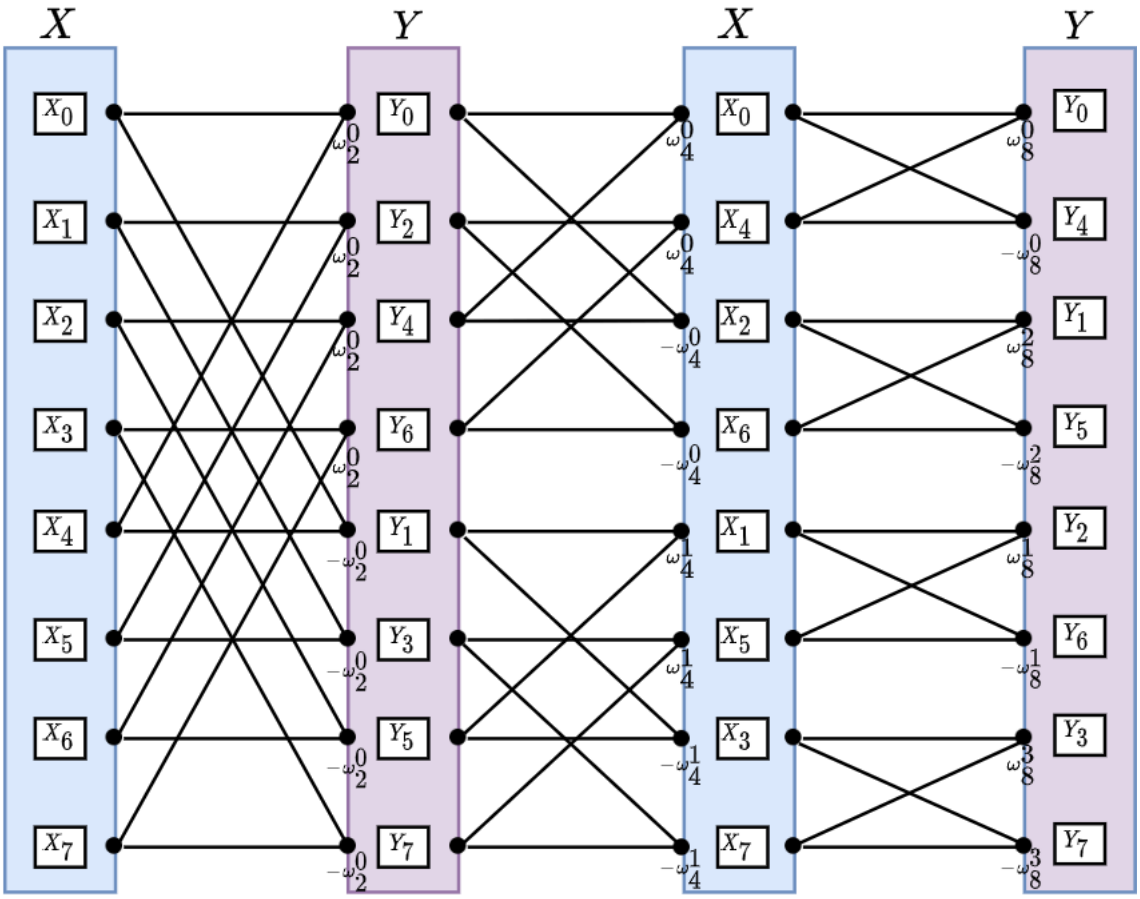
2.2. Stockham FFT

Stockham algoritmas matematine formuluote nuo 2.1 skyrelyje minėto Cooley-Tukey algoritmo nesiskiria. Tačiau, Stockham algoritmas pasižymi realizacijos ypatumais dėl kurių yra plačiau naudojamas lygiagretizavimo kontekste [GLD+08; LBG08; Tak19], kadangi, algoritmas išsprendžia aktualią Cooley-Tukey formuluotės problemą - butinybę atlikti atvirkštinį bitų perrūšiavimą.

2.2.1. Atvirkštinio bitų perrūšiavimo būtinybės išvengimas rūšiuojant elementus

Būtinybės atlikti atvirkštinį bitų perrūšiavimą galima išvengti pasitelkiant antrą masyvą, galintį talpinti N elementų ir kiekviename $l \in [1, \log_2 N]$ žingsnyje atliekant elementų perrūšiavimą kartu su drugelio operacijomis.

Tarkime įeities duomenys egzistuoja masyve X, bei egzistuoja pagalbinis masyvas Y. Abu masyvai gali talpinti N elementų. Tuomet kiekviename l žingsnyje nuskaitant elementus su indeksais i ir $i + N/2$, atliekant drugelio operaciją ir rezultatus išsaugant pagalbiniame masyve atstumais 2^{l-1} gaunamas natūralus elementų išrūšiavimas (kadangi antrojo masyvo deka galima perrašyti įeities duomenis - nebereikia atlikti bitų atvirkštinio perrūšiavimo). Kiekviename l žinsgnyje masyvai X ir Y yra sukeičiami vietomis (t.y., sukeičiamas masyvas iš kurio yra skaitoma ir į kurį yra rašoma). Pagrindo-2 $N = 8$ Stockham pavyzdys drugelio operacijų pavidalu matomas Pav. 3.



3 pav. $N = 8$ pagrindo-2 Stockham FFT operacijų diagrama

2.3. Aukštesnių pagrindų Cooley-Tukey ir Stockham algoritmai

Pristatyti 2 pagrindo Cooley-Tukey ir Stockham algoritmai yra tik specifiniai atvėjai, kuomet $N = 2^L$ ir suma $L = \log_2 N$ kartų dalinama į vis mažesnes $N/2, N/4, \cdots, N/2^{L-1}, N/2^L$ elementų sumas. Bendru atveju, jei $N = m^L$, $L \geq 1$, tuomet sumos dekomponavimas gali tęstis $L = \log_m N$ kartų, kiekvieną kartą einamąsias sumas dalinant į vis mažesnes $N/m, N/m^2, \cdots, N/m^{L-1}, N/m^L$ elementų sumas. Tokį apribojimą įeities duomenų skaičiui N turintis ir tokiu dekomponavimu pasižymintis algoritmas vadinamas pagrindo-m algoritmu [Tak19].

Pavyzdžiui, pagrindo-4 atveju, suma $L = \log_4 N$ kartų dalinama į mažesnes $N/4, N/16, \cdots, N/4^L$ elementų sumas renkantis elementus žingsniais $4n$, $4n+1$, $4n+2$, $4n+3$. Vienas pagrindo-4 dekomponavimo žinsgnis matomas (10) lygtyje.

$$\begin{aligned}\hat{g}_f = &\sum_{n=0}^{N/4-1} g_{4n}\omega_{N/4}^{fn} + \omega_N^f \sum_{n=0}^{N/4-1} g_{4n+1}\omega_{N/4}^{fn} + \\ &\omega_N^{2f} \sum_{n=0}^{N/4-1} g_{4n+2}\omega_{N/4}^{fn} + \omega_N^{3f} \sum_{n=0}^{N/4-1} g_{4n+3}\omega_{N/4}^{fn}\end{aligned} \tag{10}$$

Aukštesnio pagrindo algoritmų variacijų pagrindinis trūkumas - stipresnis apribojimas įeities elementų skaičiui N. Tačiau su šiuo apribojimu ateina aktualus privalumas - mažesnis DFT paskai-

čiuoti reikalingas operacijų skaičius. Tęsiant pagrindo-4 pavyzdį, sumų viduje esančių kompleksinių eksponenčių reikšmės (o tuo pačiu ir sumų reikšmes) dėl (8) lygybėje išreiškto periodiškumo pasikartoja po $f = N/4$, $f = N/2$, $f = 3N/4$:

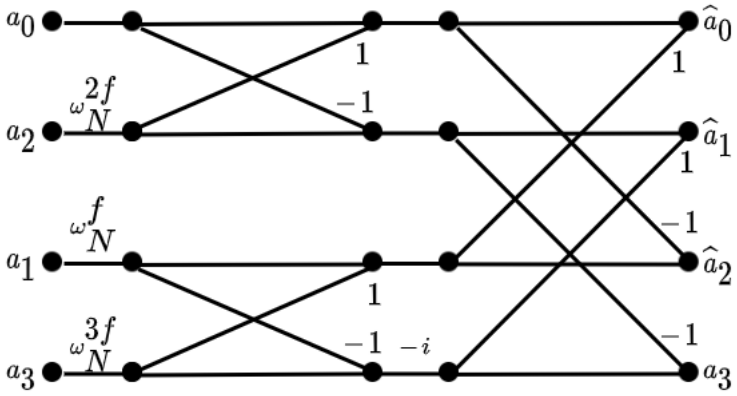
$$\omega_{N/4}^{nf} = \omega_{N/4}^{n(f+N/4)} = \omega_{N/4}^{n(f+N/2)} = \omega_{N/4}^{n(f+3N/4)} \tag{11}$$

Dėl simetriškumo, sumų išorėje esančios eksponentės taip pat pasikartos po $f = N/4$, $f = N/2$, $f = 3N/4$, tik atitinkamai padaugintos iš -1, i arba $-i$:

$$\omega_N^f = i\omega_N^{f+N/4} = -\omega_N^{f+N/2} = -i\omega_N^{f+3N/4} \tag{12}$$

Pagrindo-4 drugelio operacija susideda iš 4 pagrindo-2 drugelio operacijų (tačiau reikia paskaičiuoti mažiau kompleksinių eksponenčių). Pavyzdinė pagrindo-4 drugelio operacija matoma Pav 4.

Remiantis šia logika išvedami ir aukštesnių pagrindų (pvz., 8, 16) algoritmai.



4 pav. Radix-4 drugelio operacijos diagrama

2.3.1. Skaičių sistemos pagrindo m atvirkštinis elementų perrūšiavimas

Pagrindo m Stockham algoritmo atveju jokio išankstinio elementų perrūšiavimo atlikti nereikia, kadangi elementai yra perrūšiuojami kiekvienoje $l \in [1, \log_m N]$ stadijoje taip, kad galutinių išeities elementų indeksai atitinka įeities elementų indeksus. Cooley-Tukey pagrindo m algoritmas reikalauja, jog elementai būtų perrūšiuoti jų indeksų apversta skaičių sistemos pagrindo m eilės tvarka. T.y., jei kiekvieno elemento indeksą paversime jo skaičių sistemos pagrindo m (toliau - SM-m) išraiška ir apversime skaičius - gautas indeksas reprezentuos naują poziciją. Dalinis $N = 4^2 = 16$ pagrindo 4 pavyzdys matomas lentelėje 2.

2 lentelė. $N = 16$ elementų indeksų skaičių sistemos pagrindo 4 atvirkštinis perrūšiavimas

Įeities elemento indeksas	SM-4 indeksas	Apverstas SM-4 indeksas	Išeities elemento indeksas
0	00	00	0
1	01	10	4
2	02	20	8
3	03	30	12
4	10	01	1
5	11	11	5
6	12	21	9
7	13	31	13
...	...	...	...

2.3.2. Pagrindo 4 operacijų skaičius

Pagrindo 4 algoritmo atveju dalinimas tęsiasi $l \in [1, \log_4 N]$ kartų. Tuomet kiekviename l žingsnyje yra atliekamos $N/4$ drugelio operacijos. Kiekviena drugelio operacija susideda iš 3 kompleksinių sąndaugų ir 8 kompleksinių sudėčių. Taigi, bendras operacijų skaičius susideda iš $3N/4 \log_4 N = 3/8N \log_2 N$ kompleksinių sandaugų (25% mažiau nei pagrindo 2) ir $8N/4 \log_4 N = N \log_2 N$ kompleksinių sudėčių (tiek pat kiek pagrindo 2).

Priedo lentelė 4 atvaizduoja operacijų skaičių tarp algoritmo versijų su pagrindais 2, 4, 8, 16.

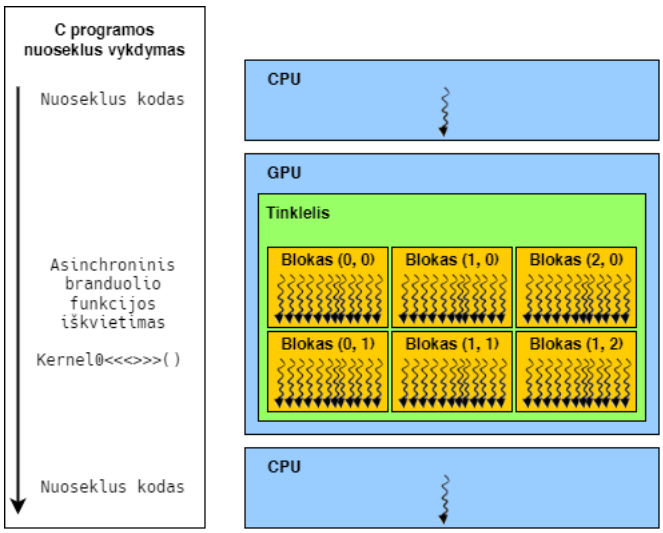
3. CUDA modelis

Didelės paklausos kontekste Nvidia 2007m. sukurė CUDA lygiagretaus programavimo modelį (angl. *parallel programming model*), kuriuo siekiama sumažinti grafinio procesoriaus (toliau - GPU) programavimui reikalingą mokymosi kreivę bei suteikti visas reikiamas abstrakcijas, tuo pačiu metu užtikrinant pakankamai žemo lygio prieigą prie GPU fizinės įrangos (angl. *hardware*). [NBG+08; Nvi22b].

CUDA programavimo modelis yra pateikiamas kaip C/C++ kalbos plėtinys, leidžiantis valdyti GPU gijas (angl. *threads*), vykdyti GPU atminties valdymo operacijas, bei atlikti sinchronizacijos veiksmus. Modelis skatina skaidyti uždavinius į smulkesnius antrinius uždavinius, kuriuos galima būtų spręsti lygiagrečiais gijų blokais (angl. *thread blocks*). Gijų blokai vykdo programuotojo parašytas branduolio funkcijas (angl. *kernel function*), kurios yra rašomos pavieniams duomenų elementams, tačiau yra vykdomos daugumos gijų vienu metu. Toks veikimo principas dar žinomas kaip vienos instrukcijos, daugumos duomenų srautų modelis (toliau - SIMD, angl. *Single Instruction, Multiple Data*) [NBG+08; Nvi22b].

3.1. CUDA lygiagretaus programavimo modelis

Pagrindiniai įrenginiai CUDA programavimo modelyje yra CPU ir GPU, kartu su savo dedikuotais atminties regionais. CPU yra labai gerai pritaikytas nuoseklių instrukcijų vykdymui, tačiau sukuria pralaidumo kliūtis (angl. *bottleneck*) programos veikimo dalyse, kurios yra pritaikomos masyviam lygiagretumui. Kadangi GPU yra pritaikyti masyviam lygiagretumui, optimalus veikimas pasiekiamas heterogeniniu lygiagrečiu programavimu (angl. *heterogeneous parallel programming*), kuomet masyviam lygiagretumui pritaikoma programos dalis yra vykdoma GPU, o nuoseklioji dalis - CPU. Heterogeninio programavimo samprata matoma Pav. 5.



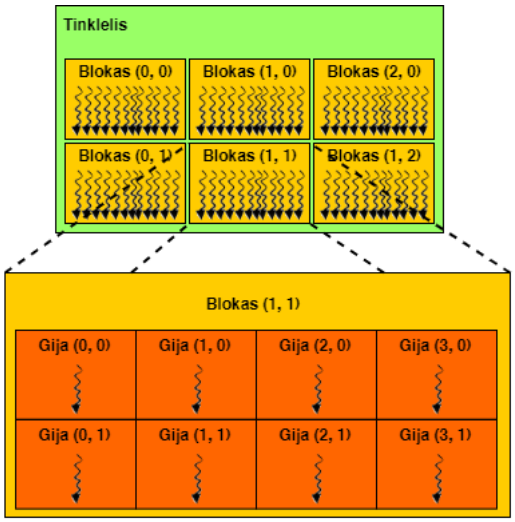
5 pav. Heterogeninio programavimo samprata [Nvi22b].

3.1.1. CUDA gijų grupių hierarchija

Tam, kad loginės gijos galėtų būti susiejamos su fiziniais GPU branduoliais (kitaip dar vadinamais CUDA branduoliais), gijos yra grupuojamos hierarchiniame gijų modelyje. CUDA gijų modelis susideda iš 3 lygių [Nvi22b]:

1. Gijos - žemiausias hierarchijos lygis. Gija atitinka branduolio funkcijos vykdymą ant vieno duomenų elemento. Vieną giją vykdo vienas fizinis CUDA branduolys.
2. Blokai - antras hierarchijos lygis. Aibė gijų sudaro bloką. Blokas gali būti 1, 2 arba 3 dimensijų.
3. Tinkleliai (angl. *grids*) - aukščiausias hierarchijos lygis. Aibė blokų sudaro tinklelį. Branduolio funkcijos iškvietimą atitinka vienas tinklelis, sudarytas iš atitinkamo skaičiaus blokų ir juose esančių branduolių. Tinklelis gali būti 1, 2 arba 3 dimensijų.

CUDA gijų grupių hierarchija taip pat matoma Pav 6.



6 pav. CUDA gijų grupių hierarchija [Nvi22b].

3.1.2. CUDA atminties hierarchija

Kaip ir gijų hierarchijos modelis, CUDA atmieties hierarcijos modelis taip pat susideda iš 3 lygių [Nvi22b]. Šie lygiai matomi lentelėje 3 ir Pav. 7.

3 lentelė. CUDA atminties tipai

Atminties tipas	Apimtis	Gyvavimo laikas
Lokali atmintis (ir registrai)	Viena gija	Gijos darbo pabaiga
Bendrinama atmintis (angl. shared memory)	Visos gijos priklausančios blokui	Bloko darbo pabaiga
Globali atmintis (angl. global memory)	Visos gijos priklausančios tinkleliui	Apibrėžta programuotojo

Siekiant projektuoti ir rašyti optimalias branduolių funkcijas aktualu suprasti atminties hierarchijos išsidėstymą fiziniame lygyje.

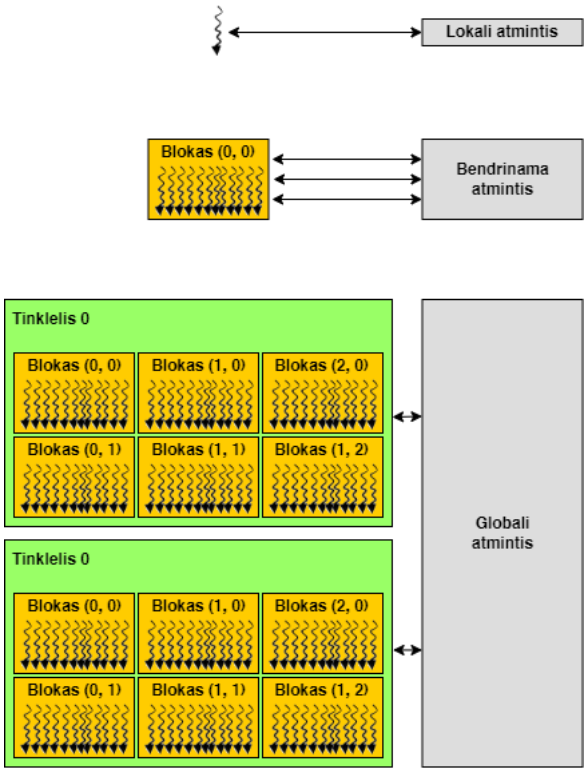
Pav 8. matoma CPU ir GPU dinaminė operatyvi atmintis (toliau - DRAM) sujungta PCIe Bus jungtimi. Globali ir lokali atmintis saugoma GPU DRAM. Bet kokių duomenų tarp CPU ir GPU

perkėlimas turi vykti atliekant atminties kopijavimo operaciją tarp GPU DRAM (globali atmintis) ir CPU DRAM per PCIe Bus jungtį. Fiziniai CUDA branduoliai yra grupuojami į srautinio perdavimo multi-procesorius (toliau - SM, angl. *streaming multiprocessors*).

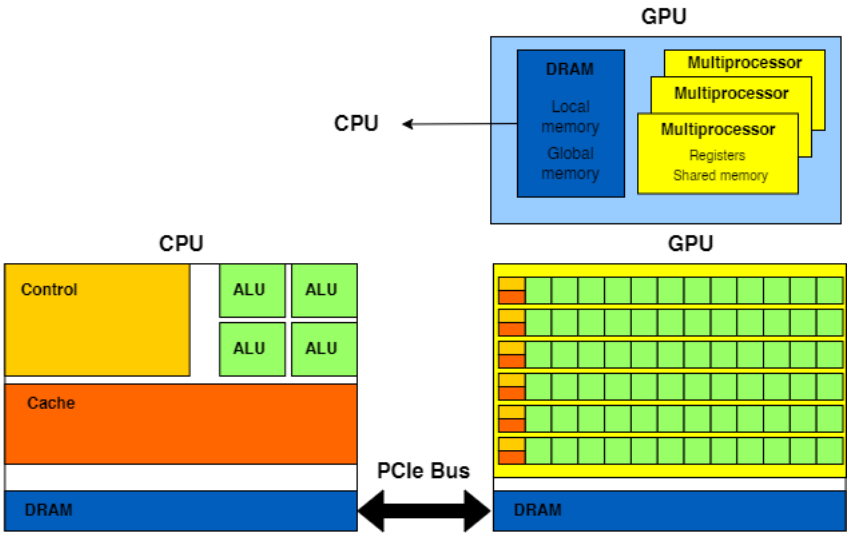
Bendrinama atmintis ir registrai yra vadinama lustine (angl. *on-chip*) atmintimi, kadangi ji randama fiziškai ant SM. Tuo tarpu lokali ir globali atmintis yra vadinama nelustine (angl. *off-chip*) atmintimi, kadangi ji nėra fiziškai randama ant SM [Nvi22a].

Dėl šių aspektų, skirtingo tipo atminties naudojimas lemia skirtingus atminties priegos greičius. Sąryšį tarp jų galima interpretuoti taip:

$$Registrai < Bendrinama \ll Lokali \approx Globali \ll CPU(PCIe) \tag{13}$$



7 pav. CUDA atmienties tipų hierarchija [Nvi22b].



8 pav. CUDA atmienties tipų hierarchija fiziniame lygyje [Nvi22a].

3.2. CUDA programavimo optimizavimo technikos

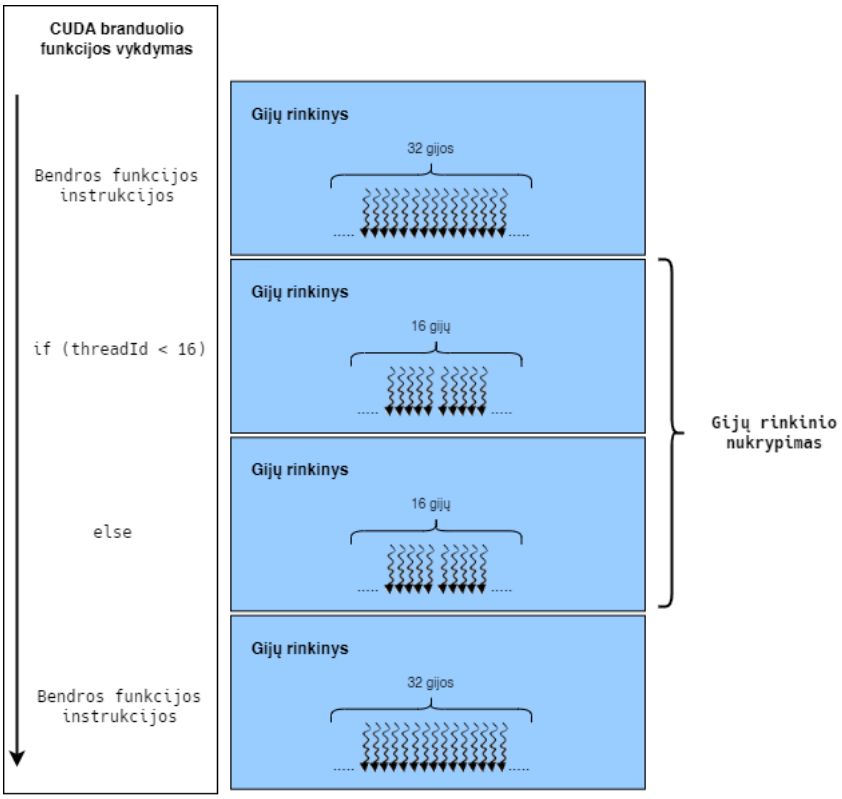
CUDA programavimo kontekste svarbu žinoti fizinės įrangos sukeliamus apribojimus ir potencialius šių apribojimų sukeliamų problemų sprendimo būdus. Šiame skyrelyje pateikiamos kelios CUDA programavime sutinkamos optimizavimo problemos ir potencialūs jų sprendimo būdai.

3.2.1. Gijų rinkinio nukrypimas

Gijos fizinės įrangos lygyje yra grupuojamos į gijų rinkinius (angl. *warps*). Gijų rinkinį sudaro 32 gijos. CUDA garantuoja, jog šios gijos priklausys tam pačiam gijų blokui ir bus vykdomos ant to pačio srautinio perdavimo multi-procesoriaus [Nvi22b].

Gijų rinkiniui priklausančios gijos vykdo po vieną bendrą instrukciją vienu metu (angl. *lockstep*). Jei dalis gijų rinkinio gijų tos pačios instrukcijos vykdyti nebegali (pvz., dėl loginio išsišakojimo sukelto *if* ar *for* sakinių), tuomet įvyksta gijų rinkinio gijų nukrypimas (angl. *warp divergence*) - atitinkama gijų dalis yra trumpam išjungiama, kol bus įvykdyta nesišakojanti dalis. Baigus nesišakojančią dalį, ją vykdžiusios gijos yra trumpam išjungiamos ir darbas pratęsiamas su prieš tai išjungtomis gijomis. Visos gijų rinkinio gijos vėl vienu metu pradeda veikti tik tuomet, kuomet visos gijų rinkinį sudarančios 32 gijos gali vykdyti tą pačią instrukciją (žr. Pav 9.).

Tai reiškia, jog kuriant CUDA branduolių funkcijas reikia stengtis sumažinti logijų gijų rinkį sudarančių gijų išsišakojimą, kadangį, blogiausiu atveju, reikės serializuoti visų gijų rinkinį sudarančių 32 gijų veikimą.



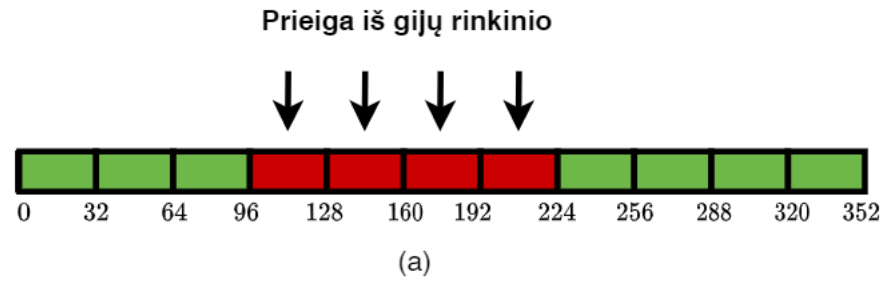
9 pav. CUDA gijų rinkinio nukrypimo pavyzdys.

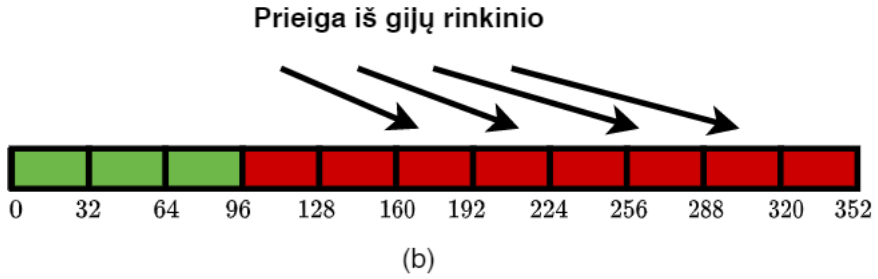
3.2.2. Jungtinė globalios atminties prieiga

CUDA architektūroje globalios atminties prieiga yra automatiškai jungiama (angl. *coalesced*) į mažiausią įmanomą transakcijų skaičių. Bendrai jungtinė globalios atminties prieiga gali būti apibūdinta taip: lygiagreti gijų rinkinyje esančių gijų prieiga prie globalios atminties bus sugrūpuota į transakcijų skaičių, lygų 32-baitų transakcijoms reikalingoms aprūpinti visas gijų rinkinio gijas [Nvi22a].

Pavyzdžiui, jei kiekviena gijų rinkinio gija (iš viso 32 gijos) prašys prieigos prie gretimų 4-baitų segmentų, tuomet keturios 32-baitų transakcijos aprūpins visas gijų rinkinio gijas (t.y., bus atliktos 4, o ne 32 transakcijos), kaip matoma Pav 10. a. Priešingu atveju, jei, pavyzdžiui, kiekviena gijų rinkinio gija prašys prieigos prie kas antro 4-baitų segmento, reikės atlikti aštuonias 32-baitų transakcijas (50% daugiau nei prieš tai), kaip matoma Pav 10. b.

Tai reiškia, jog kuriant CUDA branduolių funkcijas reikia atsižvelgti į globalios atminties prieigos struktūrą (angl. *pattern*), kadangi blogiausiu atveju, reikės atlikti iki 32 transakcijų - kiekvienai gijų rinkinio gijai.





10 pav. Jungtinė gretima (a) ir jungtinė negretima (b) CUDA globalios atminties prieiga [Nvi22a].

3.2.3. Dalybos ir modulio operacijų ekvivalentai

Dalybos ir modulio veiksmai GPU kontekste yra daug laiko užimančios operacijos ir su specifinėmis sąlygomis gali būti pakeistos atitinkamomis bitų operacijomis. Jei N yra skaičiaus 2 laipsnis, tada:

- Dalybos operacija (i/N) ekvivalenti bitų poslinkiui į dešinę $(i >> \log_2 N)$.
- Modulio operacija $(i \mod n)$ ekvivalenti bitų IR operacijai $(i\&(n-1))$.

4. CUDA lygiagretizuoti FFT algoritmai

4.1. Lygiagretizuota Cooley-Tukey FFT

4.1.1. Lygiagretus atvirkštinis bitų perrūšiavimas

Lygiaigrečiam atvirkštiniam bitų perrūšiavimui reikia N gijų, kadangi suradus elemento su indeksu $i \in [0, N/2-1]$ bitų atvirkštinę išraišką negalima garantuoti, jog ji bus intervale su indeksu $j \in [N/2, N-1]$ ir atvirščiai.

64-bitų indekso bitų atvirkštinei išraiškai rasti yra kviečiama GPU vidinė operacija (angl. *intrinsic operations*) $brevll$. Vidinės GPU operacijos yra itin greitos, todėl šią algoritmo dalį taip pat vykdant lygiagrečiai galima šiek tiek sumažinti vieno iš didžiausių algoritmo trūkumų efektą [Nvi22b].

Atlikus operaciją $brevll$, gautą naują indeksą reikia paslinkti per $64 - \log_2 N$ bitus į dešinę, kad būtų gaunamas $\log_2 N$ aktualius bitus turintis indeksas. Lygiagretaus bitų atvirkštinio perrūšiavimo branduolio funkcijos pseudo-kodas matomas Algoritme 1.

Algoritmas 1 Lygiagretaus atvirkštinio bitų perrūšiavimo branduolio funkcija

```
1: procedure BIT_REVERSE(x, n, l)
2:     i ← GLOBAL_THREAD_ID
3:     if i < n then
4:         inv_i ← BIT_SHIFT_RIGHT(BREVLL(i), (64 − l))
5:         if inv_i > i then
6:             SWAP(x[i], x[inv_i])
7:         end if
8:     end if
9: end procedure
```

Kreipiantis į elementus indekse i yra pasiekiama efektyvi jungtinė globalios atminties prieiga, tačiau priega į elementus indekse inv_i yra neefektyvi, kadangi greta vienas kito esantiems i, jų bitų atvirkštinės išraiškos inv_i gali būti stipriai nutolusios viena nuo kitos.

4.1.2. Lygiagrečios drugelio operacijos

Skyriuje 2.1 apibrėžtas pagrindo 2 Cooley-Tukey FFT algoritmas susideda iš $\log_2 N$ išorinių žingsnių. Kiekvieno žingsnio viduje yra atliekama $N/2$ viena nuo kitos nepriklausančių drugelio operacijų. Dėl (6) lygybėje apibrėžto sinusoidų simetriškumo, vienai drugelio operacijai atlikti reikia paskaičiuoti tik vieną kompleksinį keoficientą ω_N^f. Taigi, jei kiekviena gija atliks lygiai vieną drugelio operaciją, tuomet reikalingas bendras CUDA gijų skaičius šioje algortimo vykdymo dalyje bus $N/2$.

Kiekvienoje $l \in [1, \log_2 N]$ iteracijoje kiekviena gija įgauna indeksą $i \in [0, N/2-1]$. Tuomet teigiant, jog $s = 2^{l-1}$, gijai priskiriamas pirmasis drugelio operacijos elementas, kurio indeksas

randamas pagal formulę:

$$j = (i - (i \mod s)) * 2 + (i \mod s) \tag{14}$$

Tuomet antrasis drugelio operacijoje dalyvaujantis elementas randamas indeksu $j + s$.

Vieną pagrindo 2 drugelio operaciją atliekančios branduolio funkcijos pseudo-kodas matomas Algoritme 2.

Algoritmas 2 Lygiagreti Cooley-Tukey pagrindo 2 drugelio operacijos branduolio funkcija

```
1:  procedure INNER_RADIX2_COOLEY_TUKEY(x, n, s)
2:      i ← GLOBAL_THREAD_ID
3:      if i < n/2 then
4:          k ← BIT_AND(i, s − 1)              ▷ k = i mod s, kadangi s yra dvejeto laipsnis.
5:          j ← BIT_SHIFT_LEFT(i − k, 1) +k    ▷ Ekvivalentu (14) formulei.
6:          u_0 ← x[j]
7:          u_1 ← ω_{2s}^k * x[j + s]
8:          DFT2(u_0, u_1)                     ▷ Pagrindo-2 drugelio operacija.
9:          x[j] ← u_0
10:         x[j + s] ← u_1
11:     end if
12: end procedure
```

Algoritme 2 matoma branduolio funkcija veikimo metu nepatiria jokio gijų rinkinio nukrypimo, kadangi funkcijoje nėra jokio loginio išsišakojimo. Taip pat, kreipiantis į elementus j ir $j + s$ pasiekiama efektyvi globalios atminties prieiga (gijos tarpusavio atžvilgiu kreipiasi į nuoseklius elementus).

4.2. Lygiagretizuota Stockham FFT

Stockham algoritmas yra lygiagretizuojamas beveik tokiu pačiu būdų, kaip ir skyrelyje 4.1 minėtas Cooley-Tukey algoritmas. Kadangi Stockham algoritme nėra atliekamas atvirkštinis bitų perrūšiavimas (deka antro pagalbinio masyvo), indeksai iš kurių kiekviena gija skaito ir į kūriuos rašo skiriasi. Kaip ir ankščiau, apsibrėžiama, jog kiekvienoje $l \in [1, \log_2 N]$ iteracijoje kiekviena gija įgauna indeksą $i \in [0, N/2 - 1]$, o $s = 2^{l-1}$. Tuomet pirmasis ir antrasis drugelio operaciją sudarantys elementai bus randami indeksais i ir $i + n/2$. Atlikus pagrindo-2 drugelio operaciją, tarpiniai rezultatai patalpinami indeksais j ir $j + s$ (čia j paskaičiuojamas pagal (14) formulę). Šis pseudo-kodo skirtumas matomas Algoritme 3.

Po kiekvienos l iteracijos, Algoritme 3 matomi x ir y masyvai yra sukeičiami vietomis.

4.3. Lygiagretizuoti aukštesnių pagrindų Stockham FFT

Lygiagretus auštesnio pagrindo Stockham FFT algoritmo realizavimas pasiekiamas panašiu principu, kaip ir Algoritme 3 yra pasiekiamas pagrindo 2 Stockham FFT realizavimas.

Algoritmas 3 Lygiagreti Stockham pagrindo 2 drugelio operacijos branduolio funkcija

```
1: procedure INNER_RADIX2_STOCKHAM(x, y, n, s)
2:     i ← GLOBAL_THREAD_ID
3:     if i < n/2 then
4:         k ← BIT_AND(i, s − 1)                ▷ k = i mod s, kadangi s yra dvejeto laipsnis.
5:         j ← BIT_SHIFT_LEFT(i − k, 1) +k      ▷ Ekvivalentu (14) formulei.
6:         u_0 ← x[i]
7:         u_1 ← ω_{2s}^k * x[i + n/2]
8:         DFT2(u_0, u_1)                       ▷ Pagrindo-2 drugelio operacija.
9:         y[j] ← u_0
10:        y[j + s] ← u_1
11:    end if
12: end procedure
```

Pavyzdžiui, pagrindo 4 Stockham FFT algoritmas susideda iš $\log_4 N$ išorinių žingsnių. Kiekvieno žingsnio viduje yra atliekama $N/4$ viena nuo kitos nepriklausančių pagrindo 4 drugelio operacijų. Dėl (6) lybybėse apibrėžto sinusoidų simetriškumo ir (12) lygybėse pavaizduoto pasikartojimo, vienai pagrindo 4 drugelio operacijai atlikti reikia paskaičiuoti 3 kompleksinius keoficientus - ω_N^f, ω_N^{2f} ir ω_N^{3f}. Kadangi, kaip ir ankščiau, viena gija atliks vieną drugelio operaciją, tuomet reikalingas bendras CUDA gijų skaičius pagrindo 4 Stockham FFT algoritmui bus $N/4$.

Kiekvienoje $l \in [1, \log_4 N]$ iteracijoje kiekviena gija įgauna indeksą $i \in [0, N/4-1]$. Pirmasis, antrasis, trečiasis ir ketvirtasis drugelio operaciją sudarantys elementai bus randami indeksais i, $i + n/4$, $i + 2n/4$ ir $i + 3n/4$ atitinkamai. Tuomet teigiant, jog $s = 4^{l-1}$, atliktos drugelio operacijos tarpiniai rezultatai patalpinami indeksais j, $j + s$, $j + 2s$ ir $j + 3s$ atitinkamai. Čia j paskaičiuojamas pagal formulę:

$$j = (i - (i \mod s)) * 4 + (i \mod s) \tag{15}$$

Pagrindo 4 indekso j paskaičiavimo formulė nuo prieš tai apibrėžtos (14) formulės pagrindo 2 algoritmo versijai skiriasi tik iš padauginamo skaičiaus. Iš tikrųjų, jei m laikysime algoritmo pagrindu, tuomet bendra formulės išraiška tokiai algoritmo versijai atrodys taip:

$$j = (i - (i \mod s)) * m + (i \mod s) \tag{16}$$

Vieną pagrindo 4 drugelio operaciją atliekančios branduolio funkcijos pseudo-kodas matomas Algoritme 4.

Kaip ir ankščiau, po kiekvienos l iteracijos, Algoritme 4 matomi x ir y masyvai yra sukeičiami vietomis. Remiantis šia logika realizuojami ir aukštesnių pagrindų (pvz., 8, 16) Stockham FFT algoritmai.

Algoritmas 4 Lygiagreti Stockham pagrindo 4 drugelio operacijos branduolio funkcija

```
1:  procedure INNER_RADIX4_STOCKHAM(x, y, n, s)
2:      i ← GLOBAL_THREAD_ID
3:      if i < n/4 then
4:          k ← BIT_AND(i, s − 1)                    ▷ k = i mod s, kadangi s yra dvejeto laipsnis.
5:          j ← BIT_SHIFT_LEFT(i − k, 2) +k          ▷ Ekvivalentu (15) formulei.
6:          u_0 ← x[i]
7:          u_1 ← ω_{2s}^{k} * x[i + n/4]
8:          u_2 ← ω_{2s}^{2k} * x[i + n/2]
9:          u_3 ← ω_{2s}^{3k} * x[i + 3n/4]
10:         DFT4(u_0, u_1, u_2, u_3)                 ▷ Pagrindo-4 drugelio operacija.
11:         y[j] ← u_0
12:         y[j + s] ← u_2
13:         y[j + 2s] ← u_1
14:         y[j + 3s] ← u_3
15:     end if
16: end procedure
```

5. Eksperimentiniai lygiagretizavimo rezultatai

Sukurtos lygiagrečios Cooley-Tukey pagrindo-2 bei Stockham pagrindų 2, 4, 8, 16 realizacijos buvo palygintos tarpusavyje sekant DFT paskaičiuoti reikalingą laiką. Geriausius rezultatus pasiekusi pagrindo 16 Stockham realizacija palyginta su egzistuojančiais populiariais FFT algoritmų sprendimais: CuFFT [Nvi22c] ir FFTW [Mat22]. Į sekamą vykdymo laiką nėra įtraukiamas laikas reikalingas perkelti pradiniu įeities duomenis iš CPU atminties į GPU atmintį ir atvirkščiai - perkelti išeities duomenis iš GPU atminties į CPU atmintį.

5.0.1. Egzistuojančių FFT bibliotekų konfigūracijos

CuFFT ir FFTW buvo konfigūruojami su planais besitikinčiais kompleksinių dvigubo tikslumo slankiojo kablelio skaičių tiek įeityje, tiek išeityje. Papildomai, FFTW buvo nustatytas maksimalus CPU gijų skaičius lygus 8 (toks pasirinkimas grindžiamas tuo, jog autoriaus naudotas procesorius turėjo lygiai 8 fizinius branduolius).

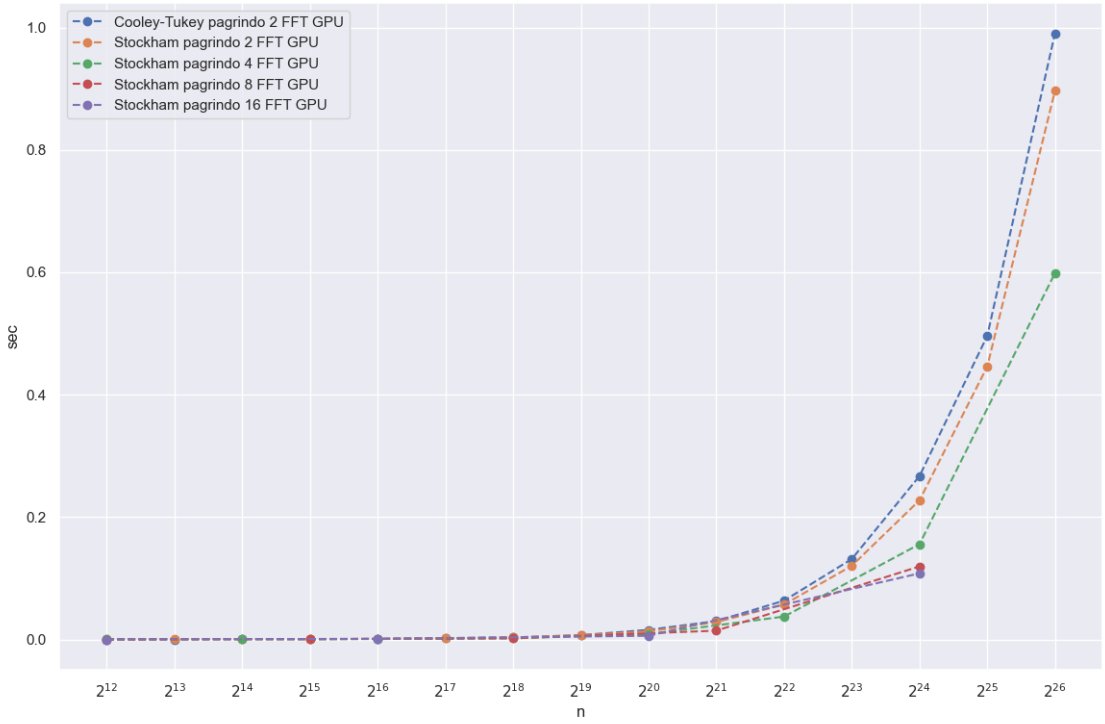
5.0.2. Eksperimentinių skaičiavimų aplinka

Visi algoritmai buvo vykdomi naudojant šią fizinę ir programinę įrangą:

- CPU - Intel(R) Xeon(R) CPU E5-2620
- GPU - Nvidia GTX 1050TI 4GB GDDR5
- Operatyvi atmintis - 32GB 2667 MHz
- Operacinė sistema - Windows 10 Professional 21H1
- C/C++ kompiliatorius - MSVC 19.29
- CUDA SDK - 11.6
- CuFFT - versija siejama su CUDA SDK
- FFTW - 3.3.10

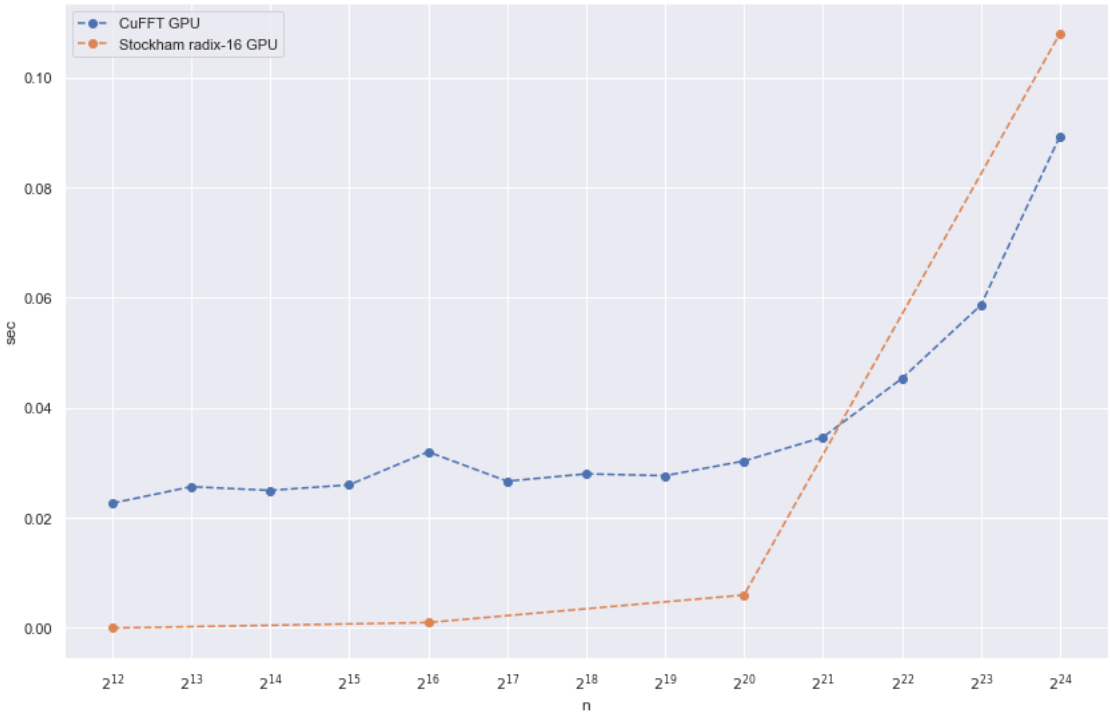
5.0.3. Eksperimentinių skaičiavimų rezultatai

Atlikti skaičiavimų eksperimentai rodo, jog pagrindo 2 Stockham algoritmas yra šiek tiek greitesnis už pagrindo 2 Cooley-Tukey algoritmą. Tai yra tikimasis rezultatas, kadangi Stockham algoritme nėra atliekamas atvirkštinis bitų perrūšiavimas. Vis dėlto, pagreitėjimas nėra žymus, dalinai deka skyrelyje 4.1.1 aprašyto lygiagretaus atvirkštinio bitų perrūšavimo naudojančio vidines GPU operacijas. Didžiausiais skirtumas ir pagreitėjimas matomas pagrindo 16 Stockham algoritme. Priedo lentelėje 4 matomi teoriniai operacijų skaičiai atitinka matomą pagreitėjimą (pagrindo 16 Stockham algoritmas daugiau nei 50% greitesnis už pagrindo 2 algoritmą). Šie rezultatai matomi Pav. 11.

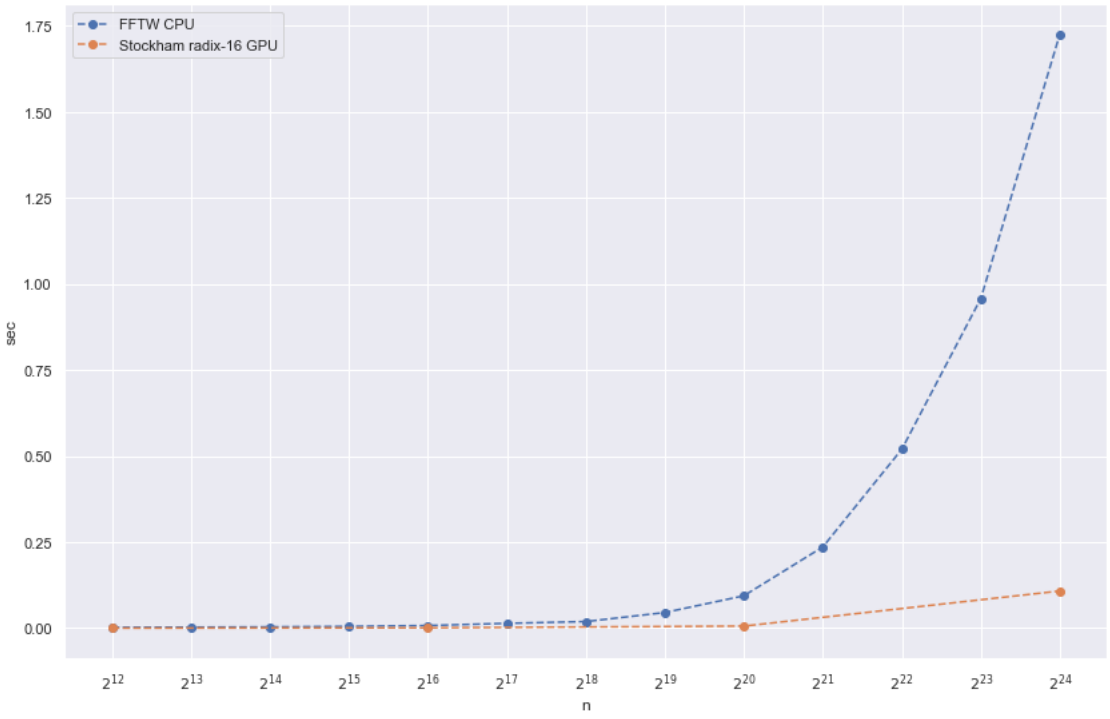


11 pav. GPU CUDA algortimų realizacijų vykdymo laikų palyginimas

Pagrindo 16 Stockham realizaciją lyginant su CuFFT matomas pranašumas iki tam tikro N. Šis pranašumas greičiausiai atsiranda dėl to, jog CuFFT turi sukurti algoritmo vykdymo planą, kurio kūrimas yra įtraukiamas į bendrą užtrunkamo laiko skaičiavimą. Šie rezultatai matomi Pav. 12. Tuo tarpu pranašumas prieš lygiagrečią CPU biblioteką FFTW yra itin didelis. Pagrindo 16 Stockham ir FFTW palyginimo rezultatai matomi Pav. 13.



12 pav. Pagrindo 16 Stockham realizacijos palyginimas su CuFFT



13 pav. Pagrindo 16 Stockham realizacijos palyginimas su FFTW

Rezultatai ir išvados

Rezultatai:

1. Identifikuoti ir matematiškai išnagrinėti GPU lygiagretizavimui tinkami FFT algoritmai.
2. Išsirinkti algoritmai ir jų versijos palygintos tarpusavyje identifikuojant jų tarpusavio pranašumus/trūkumus bei priklausomybę nuo įeities duomenų.
3. Sukurtos CUDA modeliu grįstos pasirinktų algoritmų realizacijos.
4. Realizuoti algoritmai palyginti tiek tarpusavyje, tiek su egzistuojančiais sprendimais.

Išvados:

1. Didesnį apribojimą įeities duomenims suteikiantys FFT algoritmai pasižymi didesniu našumu.
2. FFT yra itin stipriai lygiagretizuojamas uždavinys, todėl net nesufistikuotos lygiagrečios GPU realizacijos turi didelį pranašumą prieš industrijoje lydinčias lygiagrečias CPU realizacijas.

Literatūra

[CT65] James W. Cooley ir John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[ER83] Douglas F. Elliott and K. Ramamohan Rao. *Fast Transforms: Algorithms, Analyses, Applications*. Academic Press, Inc., USA, 1983. ISBN: 0122370805.

[GLD+08] Naga K Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith ir John Manferdelli. High performance discrete fourier transforms on graphics processors. *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 1–12, 2008.

[HJB84] Michael T. Heideman, Donald E. Johnson ir C. Sidney Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, 1:14–21, 1984.

[LBG08] D Brandon Lloyd, Chas Boyd ir Naga Govindaraju. Fast computation of general fourier transforms on gpus. *2008 IEEE international conference on multimedia and expo*, p. 5–8, 2008.

[Mat22] Steven G. Johnson Matteo Frigo. The fastest fourier transform in the west (fftw) library. `https://www.fftw.org`, version 3.3.10, 2022. accessed 2022-05-30.

[MV17] V. Mathuranathan ir M. Viswanathan. *Digital Modulations Using Matlab: Build Simulation Models from Scratch*. Independently Published, 2017. ISBN: 9781521493885.

[NBG+08] John Nickolls, Ian Buck, Michael Garland ir Kevin Skadron. Scalable parallel programming with cuda: is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[Nvi22a] Nvidia. Cuda c++ best practices guide. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`, version 11.6.1, 2022. accessed 2022-03-09.

[Nvi22b] Nvidia. Cuda c++ programming guide. `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, version 11.6.1, 2022. accessed 2022-03-09.

[Nvi22c] Nvidia. Cuda fast fourier transform library. `https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf`, version 11.6.1, 2022. accessed 2022-03-09.

[Tak19] Daisuke Takahashi. *Fast Fourier Transform Algorithms for Parallel Computers*. Springer Publishing Company, Incorporated, 1st leid., 2019. ISBN: 9789811399640.

[Wil05] Barry Wilkinson. *Parallel programming : techniques and applications using networked workstations and parallel computers*. C. Michael Allen, editor. Pearson/Prentice Hall, Upper Saddle River, NJ, 2nd ed. Ed., 2005.

Priedas nr. 1
Teorinio palyginimo rezultatai

4 lentelė. Pagrindų 2, 4, 8, 16 Cooley-Tukey FFT algoritmų operacijų skaičius

	Pagrindo 2	Pagrindo 4	Pagrindo 8	Pagrindo 16
Kompleksinės sandaugos	$\frac{N}{2}\log_2 N$	$\frac{3N}{8}\log_2 N$	$\frac{7N}{24}\log_2 N$	$\frac{15}{64}\log_2 N$
Kompleksinės sudėtys	$N\log_2 N$	$N\log_2 N$	$N\log_2 N$	$N\log_2 N$