KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS
KOMPIUTERIŲ KATEDRA

Mykolas Mickus

Trumpųjų bangų sklidimo modelis daugiaprocesorinėje aplinkoje

Magistro darbas

Darbo vadovas
prof. habil. dr. Rimantas Barauskas

Kaunas, 2012

Turinys

1 Įvadas

Akustinių, tampriųjų ir apskritai bangų modeliai jau seniai naudojami siekiant geriau suprasti bangų sklidimo reiškinį. Šios žinios reikalingos tiriant tokius reiškinius kaip žemės drebėjimai [1, 2, 3] arba siekiant įvertinti neardančiųjų medžiagos testavimo (angl. non destructive testing; NDT) metodų [4, 5, 6, 7] ar medicininių ultragarso tyrimų rezultatus [8]. NDT metodai plačiai naudojami tikrinti tiek naujas, tiek ir jau ilgą laiką tarnaujančias metalines konstrukcijas [4, 5, 7].

E. Ginzel išskiria tris NDT kategorijas [4]:

- Paprastas geometrinis (angl. simple geometric) – seniausias „modeliavimo“ tipas, kuomet inžinierius, remdamasis elementariomis žiniomis apie bangas, grafiniu būdu nustato, kurioje vietoje reikia įtaisyti ultragarso daviklius ant bandinio. Dabar labiausiai paplitęs metodas – „Spindulių trasavimas“ (angl. ray tracing) [4]. Tačiau tokie metodai yra labai paprasti ir tik grubiai leidžia įvertinti bangos sklidimą [5].
- Matematiniai skaičiavimai (angl. mathematical computations) – remiasi lygtimis, aprašančiomis nagrinėjamą reiškinį. Bangoms modeliuoti (taip pat ir daugeliui kitų reiškinių) yra naudojama dalinių išvestinių diferencialinė lygtis (DLDI), kitaip dar vadinama kvaziharmonine [9]. Šiai lygčiai spręsti naudojami įvairūs skaitinio integravimo (baigtinių skirtumų (angl. finite difference; FD), baigtinių elementų (angl. finite element; FE), baigtinio integravimo (angl. finite integration) [10]) metodai.
- Vizualizavimas (angl. visualisation) – tam tikras laukas (akustinių bangų, šilumos) sklinda tiriama medžiaga, o procesui „pamatyti“ naudojami „optiniai“ metodai. Šios kategorijos metodai labiausia pasitarnavo ultragarso srityje. Naudojamos dvi pagrindinės technikos: Schlieren metodas bei Fotoelastinė (angl. photo-elastic). [4]

Sprendžiant iš publikuojamų straipsnių kiekio daroma išvada, jog labiausiai paplitęs modeliavimo būdas – baigtinių skirtumų metodas [2, 6, 8, 11, 12]. Pasak Floridos valstijos universiteto mokslinių skaičiavimų departamento mokslininkų, „sprendžiant iš publikuojamos literatūros kiekio, aukštesnės eilės baigtinių elementų metodas yra antra labiausiai naudojama technika sudėtingų trimačių modelių kūrimui po baigtinių skirtumų metodo“ [1].

Visų darbų autoriai akcentuoja, jog tiek FE tiek ir FD metodais modeliuojant bangas neišvengiamai tenka atlikti labai daug skaičiavimų. Taip yra todėl, nes praktinę reikšmę

turinčius modelius sudaro didelis ir tankus tinklelis FD atveju bei didelis kiekis elementų FE atveju.

Galimi trys optimizacijų tipai:

a) Modifikuoti skaičiavimo schemą ir patį modelį. Pavyzdžiui, Barauskas ir Daniulaitis siūlo skaičiavimo pradžioje išskirti aktyvią ir neaktyvią modelio dalį ir skaičiavimus atlikti tik aktyvioje dalyje bei nagrinėjamą sritį skaidyti baigtiniais elementais taip, kad būtų sudaryti reguliaraus tinklelio posričiai, kuriose tektų DLDI spręsti tik vienam elementui, o ne visai sričiai. Autoriai teigia, jog tokios optimizacijos skaičiavimų laiką sutrumpina iki 10 kartų lyginant su įprasta skaičiavimo schema. [5]

b) Skaičiavimams spartinti naudojant kelis procesorius, klasterius ar superkompiuterius. Būtent skaičiavimų lygiagretinimas leido stipriai paspartinti ne tik bangų modeliavimo procesą, tačiau ir skaičiavimus kitose mokslo srityse: astrofizikoje, biologijoje, chemijoje ir pan. [6]

c) Pirmųjų dviejų kombinacija.

Jau 1994 metais R. S. Schechter ir jo kolegos panaudojo CM-5 superkompiuterį ultragarso bangoms modeliuoti FD metodu. Naudojant šį kompiuterį 1 milijono mazgų tinklelis buvo atnaujinamas kas 10,8ms. Darbo autoriai pabrėžia, jog lygiagretieji skaičiavimai smarkiai skiriasi nuo nuosekliųjų. [6]

Tačiau reikia nepamiršti, jog per pastaruosius 15 metų skaičiavimo technika smarkiai patobulėjo. Šiuo metu ypač didelio susidomėjimo tarp mokslininkų sulaukia naujos kartos grafiniai procesoriai (GPU). Taip yra todėl, nes šie procesoriai gali atlikti itin daug slankaus kablelio skaičių operacijų per sekundę (FLOPS) dėl specialios lygiagrečios architektūros [13]. Pavyzdžiui, maksimali teoriškai pasiekiama AMD Radeon HD 6850 procesoriaus skaičiavimo galia – 1,5 TFLOPS [14].

GPGPU (General Purpose GPU computing) era įgavo pagreitį 2006-aisiais metais, kuomet NVIDIA išleido CUDA įrankį, skirtą programuoti šios firmos grafiniams procesoriams [1, 13]. Neatsilikdama nuo konkurentės AMD išleido CUDA alternatyvą savo grafiniams procesoriams – CTM (Close To Metal) [15]. Kadangi abi šios technologijos paremtos C programavimo kalba, tai leido mokslininkams lengviau pritaikyti įvairius algoritmus GPU procesoriams. Tačiau šių dviejų technologijų pagrindinis trūkumas – programos „pririšimas“ prie gamintojo: CUDA parašytos programos gali veikti tik su NVIDIA procesoriais, o CTM – AMD procesoriais.

Šio trūkumo neturi 2008 metais Apple kompanijos sukurtas ir Khronos grupei pavestas valdyti OpenCL (Open Compute Language) standartas, „skirtas modernių procesorių, esančių asmeniniuose kompiuteriuose, serveriuose bei nešiojamuose įrenginiuose, lygiagrečiam programavimui“ [16]. Programos, sukurtos naudojant šį standartą, gali veikti bet kurioje operacinėje sistemoje ar su bet kuriuo procesoriumi, kuris palaiko OpenCL standartą. Didžiųjų procesorių gamintojų (Intel, AMD, NVIDIA) procesoriai jau palaiko šį standartą. Šios kompanijos taip pat teikia įrankius, skirtus programoms rašyti remiantis OpenCL. Šios ir daugelis kitų kompanijų dalyvauja kuriant standartą: per 3 technologijos metus jau yra išleisti 2 standarto papildymai. Tiesą sakant, AMD atstovas Terry Makedon portalui bit-tech.com teigė, jog kompanija yra įsitikinusi, jog būtent OpenCL yra ateities skaičiavimų technologija [17].

2 Bangų modeliavimas lygiagrečiose platformose

Kaip jau minėta anksčiau, akustinių ir tampriųjų bangų modeliavimui pasitelkiami du metodai: baigtinių skirtumų (FD) ir baigtinių elementų (FE).

1994 metais R. S. Schechter ir kitų atliktas darbas – seniausias nagrinėtoje literatūroje minimas lygiagretus akustinės bangos modelis. Autoriai nagrinėja akustinės bangos sklidimą kietoje terpėje. Šios bangos modeliavimui jie naudoja tampriosios bangos lygtį kuri sprendžiama FD metodo bei centrinių skirtumų integravimo schemos pagalba. Autoriai pabrėžia, jog vienodas FD lygties pavidalas bei galimybė medžiagos savybes išreikšti skirtingomis koeficientų reikšmėmis kiekviename tinklelio taške leidžia veiksmus atlikti tuo pačiu metu. Modelis buvo tikrinamas su įvairiais duomenimis (sluoksniuotos medžiagos, pavydžiai su įtrūkimais ar lūžiais ir pan.) naudojant CM-200 kompiuterį, turintį 4096 lustus, 8 GB darbinės atminties. 1 milijono mazgų tinklelis buvo atnaujinamas kas 10,8ms. Tačiau autoriai nesiėmė jokių veiksmų komunikacijai tarp procesorių optimizuoti ir teigia, jog tai padarius geresni rezultatai būtų pasiekti net ir su kuklesnius skaičiavimo pajėgumus turinčiu CM-200 kompiuteriu [6].

Kadangi tiek FD tiek FE metodais paremti modeliai yra naudojami jau seniai, naujausių darbų autoriai dėmesį labiau kreipia į tų modelių pritaikymą lygiagrečioms architektūroms ir siekiai efektyviai spręsti tokias problemas:

- komunikacijos dažnumo ir laiko tarp procesorių ar jų branduolių mažinimas;
- operacijų skaičiau, reikalingo apdoroti elementą FE arba mazgą FD atveju mažinimas;

- skaičiavimo schemos modifikavimas siekiant išvengti sinchronizavimo tarp skirtingų elementų;
- kiek įmanoma efektyvesnis procesoriaus specifinės architektūros ir įtaisų išnaudojimas.
- integravimo žingsnio didinimas naudojant aukštesnės eilės FE elementus, kurie užtikrintų tokį patį tikslumą, naudojant pirmos eilės elementus kartu su mažu integravimo žingsniu.

Kern M. [20] tampriosios bangos modelį realizavo naudodamas MPI (Message Passing Interface) standartą. Siekdamas padidinti tikslumą ir taip įgalinti didesnį integravimo žingsnį autorius naudojo aukštesnės eilės FE elementus bei ketvirtos eilės centrinių skirtumų DLDI integravimo schemą. Nagrinėjama sritis padalinta į posričius, kurių kiekvieną apdoroja atskiras procesorius (toks darbų paskirstymas naudojamas ir kituose nagrinėtuose darbuose). Kiekvieno žingsnio metu, procesoriai atlikę skaičiavimais keičiasi bendrų mazgų duomenimis (panašią komunikaciją naudoja ir Nicholas L. [11] savo akustinių bangų modelyje). Autorius taip pat svarstė ir apie galimybę praplėsti posričius taip, kad jie persidengtų. Toks metodas įgalintų retesnę komunikaciją (kas kelis žingsnius) tarp procesorių, tačiau, autoriaus įsitikinimu, tokiam modeliui reikėtų daug daugiau atminties, ypač trimačiu atveju. Kern M. Pavyko pasiekti beveik 13 kartų pagreitėjimą naudojant 16 procesorių lyginant su sprendimo laiku, kurio prireiktų tą patį uždavinį atlikti su vienu procesoriumi. Tačiau neaišku, kokią naudą davė aukštesnės eilės elementai bei 4-os eilės integravimo schema lyginant su įprastiniais elementais bei antros eilės integravimo schema.

Taborda R. su kolegomis nagrinėjo FE tampriosios bangos didelio žemės drebėjimo sklidimo simuliacijos spartinimo galimybes. Autoriai naudojo Cranegie Mellon universiteto Quake grupės sukurtą įrankį Hercules, skirtą būtent žemės drebėjimų simuliacijoms atlikti [3]. Dėl šios priežasties jie visą dėmesį skyrė elementų standumo matricos koeficientų efektyviam apskaičiavimui. Vieno elemento matrica K^e gali būti apskaičiuojama pagal formulę $K^e = B^T D B$. Kadangi B yra reta matrica (daugelis jos elementų yra lygūs 0), darbo autoriai K^e apskaičiavimą išskleidė į efektyvių operacijų seką (operacijos, kuriose nedalyvauja nuliniai B matricos elementai). Toks sprendimas leido elementarių operacijų (daugyba ir sudėtis) skaičių sumažinti nuo 18 iki 14 (22% mažiau) vienmačiu, nuo 120 iki 48 dvimačiu (60% mažiau) bei nuo 1128 iki 373 trimačiu atveju (67% mažiau operacijų). Toks sprendimas mokslininkams simuliaciją leido paspartinti vidutiniškai 3,2 karto. Autoriai

pripažįsta, jog tokia optimizacija išryškina neefektyvios komunikacijos tarp procesorių problemą, kurią jie ir žada spręsti kituose darbuose.

2.1 Bangos ir GPU

Didžioji dalis modelių veikiančių GPU procesoriuose bei iš jų sudarytuose klasteriuose yra paremti FD metodu [2, 12, 21]. Tuo tarpu pavyko rasti tik vieną darbą, kuriame naudojamas FE metodas ir simuliacija vykdoma GPU klasteryje [1].

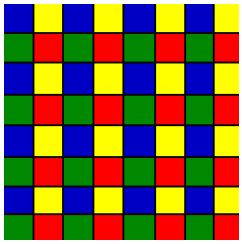
NVIDIA inžinierius Micikevičius P. [12] aprašo apibendrinto trimačio FD metodo realizacijos ypatumus naudojant CUDA programavimo aplinką ir NVIDIA TESLA GPU procesoriams. Autorius naudoja specialią metriką, vadinamą „kreipimosi į atmintį perteklingumas“. Ši metrika parodo, kiek įėjimo elementų nuskaitomi iš pagrindinės GPU atminties, kad būtų galima apskaičiuoti vieną išėjimo reikšmę. Anot autoriaus, naivus būdas vieno FD gardelės mazgo atnaujinimui turi $3k+1$ atminties nuskaitymo perteklingumą. Grupuojant mazgus ir naudojant sparčią grupei bendrą atmintį (angl. shared memory) priklausomai nuo grupės dydžio galima šį rodiklį sumažinti iki 1,5 (32x32 dydžio grupė). Micikevičius pabrėžia, jog norit sumažinti atminties nuskaitymo perteklingumo mato reikšmę trimačiu atveju, GPU procesorius turi per mažai sparčiosios atminties, kurioje būtų galima laikyti informaciją apie visus gretimus mazgus. Todėl jis pasiūlė trimatę erdvę apdoroti dvimačiais langais. Kiekviename žingsnyje atskira gija iš pagrindinės atminties į lokalią (paprastai registrus) atmintį įkelia informaciją apie mazgą, kuris kitame žingsnyje bus apdorojamas. Lokalioje atmintyje taip pat laikomi duomenys apie praėjusiame žingsnyje apdorotą mazgą. Kadangi apie šiuo metu apdorojamą mazgą informacijos reikia ir gretimoms gijoms (jos apdoroja gretimus mazgus), ji laikoma sparčioje bendroje atmintyje.

Micikevičius taip pat pateikia rekomendacijas, kaip reikia atlikti skaičiavimus naudojant 2 ar daugiau GPU procesorių. Duomenų mainus tarp procesorių autorius siūlo paslėpti juos vykdant kartu su skaičiavimais: iš pradžių kiekvienas procesorius atnaujina kraštinius mazgus, tuomet inicijuojami duomenų mainai tarp procesorių. Kol duomenys yra kopijuojami, kiekvienas procesorius atnaujina likusius mazgus.

Komatitsch D. ir Michea D. Micikevičiaus rekomendacijas sėkmingai pritaikė savo darbe modeliuodami seisminę bangą [2]. Jiems pavyko nuo 20 iki 60 kartų pagreitinti modelio veikimą pasitelkus vieną ar kelis NVIDIA GPU procesorius lyginant su nuoseklia jo realizacija. Autoriai papildomai išnaudojo tekstūras (iš principo tai yra paveikslėliai pagrindinėje atmintyje, kuriems nuskaityti GPU turi specialią ir greitą aparatūrą), kurios leido

pasiekti tokį našumą.

Komatitsch D. jau su kitais kolegomis seismines bangos modelį pritaikė dideliam GPU kalsteriui remdamiesi FE metodu [1]. Autoriai naudojo aukštesnių eilių elementus. Modelis, kaip ir [2] darbe, realizuotas naudojant CUDA technologiją bei NVIDIA Tesla procesorius. Lyginant autorių naudotu CPU klasteriu, jiems pavyko pasiekti nuo 12 iki 20 kartų pagreitėjimą. Kaip ir kitų darbų autoriai, šie mokslininkai taip pat komunikaciją tarp procesorių perdengė tuo pačiu metu vykdomais skaičiavimais. Tačiau trimatę erdvę jie suskaidė į kubus o ne dvimates plokštumas, kaip kad siūlė Micikevičius. Autoriai nenaudojo spartinančios atminties, tačiau identifikavo elementus, kurie neturi bendrų mazgų. Tai jiems leido išvengti sinchronizavimo tarp elementų ir atominių operacijų. Toks elementų suskirstymas pavaizduotas 1 paveiksle (aiškumo dėlei pateiktas dvimatis variantas, tačiau trimatė erdvė suskirstoma analogiškai).



1 Paveikslas: [1] šaltinyje siūlomas elementų apdorojimas: ta pačia spalva nuspalvinti elementai neturi bendrų mazgų, todėl gali būti apdorojami lygiagrečiai nenaudojant jokios sinchronizavimo tarp gijų

2.2 Darbo uždaviniai ir tikslas

Iš literatūroje aprašytų darbų matome, jog GPU procesoriai leido paspartinti ne vieną ir ne tik bangų reiškinio simuliaciją. Visų darbų autoriai sutartinai teigia, jog šiems procesoriams tobulėjant, ne tik augs jų teikiama nauda, tačiau jų naudojimas taps paprastesnis dėl gausėjančių įrankių pasiūlos šiems procesoriams programuoti.

Galime pastebėti, jog visi be išimčių darbai, atlikti naudojant GPU procesorius buvo programuojami CUDA aplinkoje. Kai kurie darbai užsimena apie galimybę naudoti kitas technologijas, pavyzdžiui OpenCL [1, 21]. CUDA yra populiari HPC (high performance computing) bendruomenėje, nes tai buvo pirmasis produktas, leidžiantis GPU procesorius programuoti tikra programavimo kalba. Tuo tarpu OpenCL standartas sukurtas tik 2008 metais. Tačiau CUDA turi vieną didelį trūkumą: ji priklauso NVIDIA ir veikia tik su šios kompanijos gaminamais procesoriais. Tuo tarpu OpenCL šio trūkumo neturi.

Reikia paminėti, jog visi apžvelgti lygiagretūs modeliai veikia homogeninėse platformose: skaičiavimams naudojami vienodi procesoriai, o CPU atlieka tik prižiūrėtojo vaidmenį. Todėl kyla klausimas: ar galima apkrauti darbu apkrauti ne tik GPU procesorius tačiau ir juos valdantį CPU, kol jis yra nenaudojamas. Kaip tai padaryti?

Taip pat visi autoriai akcentuoja, jog modeliai pritaikyti heterogeninei medžiagai. Todėl kyla klausimas: kaip lygiagrečioms architektūroms pritaikytus modelius modifikuoti, kad būtų galima pasinaudoti Barausko R. [5] siūlymu dalinti nagrinėjamą sritį į homogeninius vienodų elementų posričius ir taip sumažinti atliekamų operacijų skaičių?

Taip pat reikia pabandyti pritaikyti optimizacijas, naudojamas minėtas FD metodų realizacijose, nes, kaip minėta, literatūroje aprašytas tik vienas FE metodu pagrįstas modelis.

Todėl darbo tikslas – sukurti tampriųjų bangų modelį atviru standartu paremta lygiagretaus programavimo technologija OpenCL ir maksimaliai išnaudoti visus sistemos procesorius (tiek CPU tiek ir GPU) pateikiant atsakymus į anksčiau iškeltus klausimus.

3 Tampriosios bangos modeliavimas baigtinių elementų metodu

3.1 Matematinis elemento modelis

Tampriosios bangos sklidimas tamprioje srityje (kontinuume) aprašomas Navjė lygtimi (kuri yra vienos iš plačiausiai įvairiems kontinuumams aprašyti naudojamos DLDI, vadinamos kvaziharmonine lygtimi, atskiras atvejis). Ši lygtis turi dvi formuluotes.

Stiprioji formuluotė (3.1.1) reiškia, jog diferencialinė lygtis turi būti tenkinama visuose tūriui V priklausančiuose taškuose.

$$\begin{cases} \frac{\delta \sigma_x}{\delta x} + \frac{\delta \tau_{xy}}{\delta y} + b_x = \rho \ddot{u} \\ \frac{\delta \sigma_y}{\delta y} + \frac{\delta \tau_{xy}}{\delta x} + b_y = \rho \ddot{v} \end{cases}, \in V \qquad (3.1.1)$$

Kraštinės kontūro S_t sąlygos užrašomos remiantis vidinių įtempimų ir paskirstytosios išorinės apkrovos jėgų pusiausvyra:

$$\begin{cases} n_x \sigma_x + n_y \tau_{xy} = t_x \\ n_y \sigma_y + n_x \tau_{xy} = t_y \end{cases}, \in S_t \qquad (3.1.2)$$

Čia ρ – medžiagos tankis, n_x, n_y kontūro S_t išorinės normalės projekcijos, $\sigma_x, \sigma_y, \tau_{xy}$ – įtempimų tenzoriaus komponentės, u, v – poslinkio komponentės bet kuriame srities taške, $\ddot{u}, \ddot{v}$ – poslinkio antros eilės išvestinės (pagreičiai).

Silpnoji formuluotė:

$$h\iint_V w_1\left(\delta\frac{\sigma_x}{\delta x}+\frac{\delta\tau_{xy}}{\delta y}+b_x-\rho\ddot{u}\right)dV+h\iint_V w_2\left(\frac{\delta\sigma_y}{\delta y}+\frac{\delta\tau_{xy}}{\delta x}+b_y-\rho\ddot{v}\right)dV=0 \quad (3.1.3)$$

čia $w_1(x,y), w_2(x,y)$ – laisvai pasirenkamos svorinės funkcijos, apibrėžtos srityje V.

Ši silpnoji formuluotė (3.1.3) reiškia, jog jei dydžiai $\sigma_x, \sigma_y, \tau_{xy}, \ddot{u}, \ddot{v}$ (3.1.1) diferencialines lygtis bei (3.1.2) kraštines sąlygas tenkina tik apytiksliai, (3.1.3) lygties kairioji pusė (3.1.1) lygčių kairiųjų ir dešiniųjų pusių neatitikčių (skirtumų), padaugintų iš laisvai pasirinktų svorinių funkcijų w_1, w_2 integralų suma. Radus $\sigma_x, \sigma_y, \tau_{xy}, \ddot{u}, \ddot{v}$ dydžių reikšmes, kurioms esant ši suma lygi nuliui, bus rastas sprendinys, kuris lygtis tenkina su nedidele paklaida. Toks silpnosios formuluotės užrašymas dar vadinamas svertinių neatitikčių metodu.

w_1, w_2 funkcijomis parinkus Galiorkino svorines funkcijas bei atlikus įvairius pertvarkymus gaunama baigtinio elemento (lygties išraiška yra bendra įvairios formos ir mazgų skaičiaus elementams, tačiau darbe naudojamas stačiakampis elementas) dinamikos lygtis:

$$[M^e]\{\ddot{U}^e\}+[K^e]\{U^e\}=\{F^e\}+\{P^e\}+\{S^e\} \quad (3.1.4)$$

čia $[M^e]=\rho\iint_{V_e}[N]^T[N]dv$ – elemento masių matrica (3.1.5)

$[K^e]=\int_V[B^T][D][B]dV$ – elemento standumo matrica;

$\{F^e\}$ – sutelktųjų mazginių jėgų vektorius;

$\{P^e\}$ – tūrinių apkrovų sąlygotas mazginių jėgų vektorius;

$\{S^e\}$ – paviršinių apkrovų sąlygotas mazginių jėgų vektorius;

Matrica $[D]$ yra medžiagos standumo tenzorius, o matrica [N] – elemento formos funkcijų matrica, matrica [B] – susieja deformacijas elemente su jo mazginiais poslinkiais. Elemento matricų išvedimo procedūrą galima rasti [22] vadovėlyje.

Šiame darbe į elemento modelį tūrinių ir paviršinių apkrovų neįtraukiame, tačiau įtraukėme

virpesių slopinimus, kurie išreiškiami matrica $[C^e]=\alpha[M^e]$. Čia α – slopinimo koeficientas. Taigi darbe naudojamo keturkampio baigtinio elemento dinamikos lygties išraiška yra tokia:

$$[M^e]\{\ddot{U}^e\}+[C^e]\{\dot{U}^e\}+[K^e]\{U^e\}=\{F^e\} \qquad (3.1.6)$$

3.2 Centrinių skirtumų skaitinio integravimo schema

Tam, kad (3.1.6) lygtį būtų galima užrašyti išreikštiniu pavidalu – poslinkių pagreičius išreikšti per kitus lygties narius, reikia pakeisti $[M^e]$ matricą. Suintegravus (3.1.5) išraišką gausime *konsistentinę* masių matricą, kurios inversijos rasti negalima. Dėl to elemento matrica keičiama sutelktųjų masių matrica, kuri yra *diagonalioji*, nes kiekvienam elemento mazgui tenka 1/4 stačiakampio elemento masės.

1. $\dot{U}_{n-\frac{1}{2}}, U_n$
2. $\ddot{U}_n \approx [M]^{-1}\{f(U_n, \dot{U}_{n-\frac{1}{2}}, t)\}$
3. $\dot{U}_{n+\frac{1}{2}} = \dot{U}_n + (t_{n+\frac{1}{2}} - t_{n-\frac{1}{2}})\ddot{U}_n$
4. *Kraštinių sąlygų įvertinimas* (*jeigu reikia*)
5. $U_{n+1} = U_n + (t_{n+1} - t_n)\dot{U}_{n+\frac{1}{2}}$

Dabar galime užrašyti centrinių skirtumų skaitinio integravimo schemą (3.1.6) lygčiai:

1. $\dot{U}_{n-\frac{1}{2}}, U_n$
2. $\ddot{U}_n \approx [M]^{-1}\{f(U_n, \dot{U}_{n-\frac{1}{2}}, t)\}$
3. $\dot{U}_{n+\frac{1}{2}} = \dot{U}_n + (t_{n+\frac{1}{2}} - t_{n-\frac{1}{2}})\ddot{U}_n$
4. Kraštinių sąlygų įvertinimas (jei jos yra naudojamos)
5. $U_{n+1} = U_n + (t_{n+1} - t_n)\dot{U}_{n+\frac{1}{2}}$

Žodinis šios skaičiavimo schemos algoritmas yra toks:

1. Turime praėjusiame žingsnyje apskaičiuotus konstrukcijos mazgų greičius.
2. Apskaičiuojame apytiksles mazgų pagreičių reikšmes naudodami minėtas praeitame žingsnyje apskaičiuotas mazgų poslinkių, greičių reikšmes bei įvertiname mazgus veikiančias išorine bei dėl elementų įtempimo atsiradusias jėgas.

3. Suskaičiuojame naujas mazgų greičių reikšmes naudodami praeitame žingsnyje gautas pagreičių reikšmes.
4. Įvertiname, jei reikia, kraštines sąlygas (žinomi mazgų greičiai)
5. Naudodami apskaičiuotas naujas mazgų greičių reikšmes apskaičiuojame naujas mazgų poslinkių reikšmes.

3.3 Diskretizavimo žingsnio laike ir erdvėje parinkimas

Naudojant skaitinio integravimo schemą (3.1.6) lygčiai spręsti, reikia parinkti tinkamus diskretizavimo žingsnius laike ir erdvėje (atstumas tarp elemento mazgų). Maksimaliai galimas žingsnių ribas nusako (3.3.1) nelygybė:

$$\frac{c\cdot\Delta t}{\Delta x}<1 \quad ; \tag{3.3.1}$$

čia c – bangos greitis, $\Delta t\,,\Delta x$ – diskretizavimo žingsniai laike ir erdvėje.

Modeliuojant bangos sklidimą tamprioje terpėje, parenkamos tokios $\Delta t\,,\Delta x$ reikšmės:

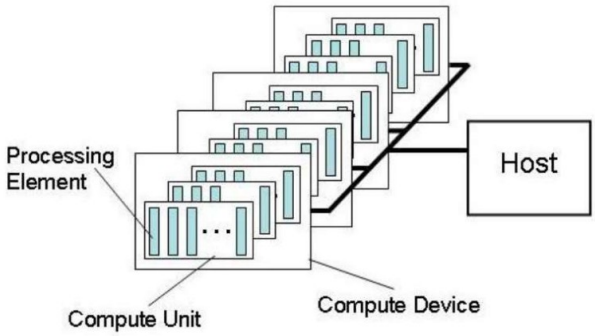
$$\Delta x<\frac{c_{min}}{5\,\omega_{exc}}, \qquad \Delta t<\frac{\Delta x}{2\,c_{max}} \tag{3.3.2}$$

čia $c=\sqrt{\frac{E}{\rho}}$ – išilginės bangos sklidimo greitis, E – medžiagos Jungo modulis, ρ – medžiagos tankis, ω_{exc} - žadinimo dažnis Reikšmės $c_{min}\,,c_{max}$ yra minimalus ir maksimalus išilginės bangos greitis konstrukcijoje.

4 OpenCL platforma

4.1 Platformos modelis

OpenCL (toliau OCL) platformos modelis yra aukšto lygio bet kokios heterogeninės platformos (platforma, susidedanti iš skirtingas funkcijas atliekančių procesorių) „vaizdas“. Šis modelis vaizduojamas 2 paveiksle. OCL platforma visuomet turi vadovaujantį procesorių arba „šeimininką“ (angl. host). „Šeimininkas“ atsakingas už programos komunikaciją su OCL aplinkai svetimais dalykais kaip įvestis ir išvestis ar programos vartotojas.



2 paveikslas: OpenCL platformos modelis

„Šeimininkas“ sujungtas su vienu ar daugiau OCL įrenginių. Čia įrenginys yra tas vienetas, kuriame vykdomi instrukcijų srautai dar vadinami OCL branduoliais (angl. kernel). Todėl šie OCL įrenginiai dar vadinami ir skaičiavimo įrenginiais. Tokiu įrenginiu gali būti CPU, GPU arba bet koks kitas procesorius, esantis sistemoje ir palaikantis OCL standartą.

Šie OCL įrenginiai viduje turi kelis (kelias dešimtis ar daugiau) skaičiavimo branduolius (angl. compute unit), o šie dar yra dalijami į vieną ar daugiau apdorojimo elementų (angl. processing elements). Skaičiavimus įrenginyje atlieka būtent šie elementai.

4.2 Programos vykdymo modelis

Kiekviena OCL programa susideda iš dviejų atskirų dalių: „šeimininko“ programos ir kelių ar daugiau OCL branduolių rinkinio. „Šeimininko“ programą vykdo vadovaujantis procesorius. OCL standartas neapibrėžia, kaip turi veikti ši programa: jis tik nurodo, kaip ji turi sąveikauti su kitais objektais OCL platformoje.

OpenCL branduolius vykdo OCL įrenginiai. Būtent jie ir atlieka didžiąją dalį skaičiavimų programoje. OCL branduoliai yra funkcijos, kurios įeinančius atminties objektus transformuoja į išeinančius atminties objektus. Kitaip tariant OCL branduolys yra funkcija, kuri manipuliuodama pradiniais duomenimis pateikia rezultatą. Galimi dviejų tipų OCL branduoliai:

- OCL branduolys: funkcijos, parašytos OpenCL C programavimo kalba ir kompiliuojamos OCL kompiliatoriumi. Visos OCL implementacijos privalo palaikyti OCL branduolius.
- Nuosavas (angl. native) branduolys: funkcijos, sukurtos nenaudojant OCL standarto ir platformai prieinamos naudojant funkcijų rodykles (angl. function pointer). Pavyzdžiui, šios funkcijos galėtų būti funkcijos, parašytos OCL šeimininko

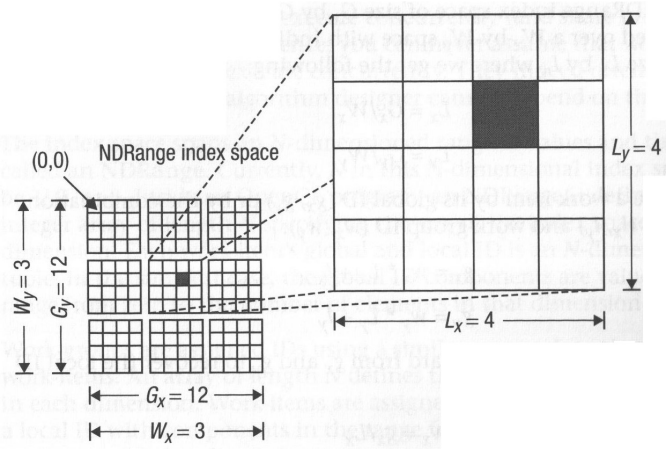
programavimo kalba arba pateiktos išorinėje bibliotekoje. Galimybė vykdyti tokius branduolius yra neprivaloma OCL standarto dalis.

4.2.1 OCL branduolio vykdymas OCL įrenginyje

Branduolys apibrėžiamas „šeimininko“ programoje. Ši programa specialiomis komandomis paruošia branduolį vykdymui OCL įrenginyje. Tuomet OCL sistema sukuria indeksų erdvę. Kiekvienam taškui šioje erdvėje skiriama atskira branduolio kopija. Šios kopijos vadinamos darbo vienetais (angl. work-item). Šie vienetai identifikuojami koordinatėmis indeksų erdvėje. Šios koordinatės dar vadinamos globaliu darbo vieneto identifikatoriumi (angl. global ID; toliau žymėsime G).

Galima sakyti, jog komanda, kuri paruošia branduolį vykdymui sukuria darbo vienetų rinkinį. Kiekvienas iš šių vienetų vykdo tą pačią veiksmų seką, kuri nurodyta branduolio funkcijoje. Nors veiksmų seka tokia pati, darbo vienetų elgesys gali skirtis dėl funkcijoje esančių kodo išsišakojimų arba duomenų, kurie apdorojami skirtingų darbo vienetų.

Darbo vienetai yra grupuojami į darbo grupes (angl. work-group). Šios grupės dalija visą indeksų erdvę į vienodas sritis. Darbo grupei priskiriamas jos identifikatorius (žymėsime W), turintis tiek pat dimensijų kiek ir G. Darbo vienetams taip pat priskiriamas lokalus identifikatorius L. Kadangi indeksų erdvė gali būti vienmatė, dvimatė arba trimatė ji dar vadinama ND-Sritimi (angl. NDRange). Toks indeksų erdės padalijimas vaizduojamas 3 paveiksle.



3 paveikslas: dvimatės indeksų erdvės pavyzdys

Darbo vienetai toje pačioje grupėje vykdomi tuo pačiu metu viename OCL įtaise. Priklausomai nuo OCL implementacijos ir įrenginio, darbo vienetai gali būti vykdomi ir nuosekliai, tačiau dažnai jie vykdomi lygiagrečiai.

4.2.2 Kontekstas

Nors skaičiavimai OCL aplikacijoje atliekami OCL įrenginiuose, „šeimininko“ procesas atlieka svarbų vaidmenį. Būtent čia yra aprašyti OCL branduoliai. „Šeimininkas“ kuria kontekstą, kuriame vykdomi branduoliai. Jis taip pat apibrėžia ir ND-Sritį bei komandų eiles, kurios kontroliuoja kaip ir kada branduoliai yra vykdomi.

Pirmasis „šeimininko“ uždavinys – konteksto sukūrimas. Kontekstas apibrėžia aplinką, kurioje kurioje aprašomi ir vykdomi branduoliai. Kontekstą sudaro šie resursai:

- Įrenginiai: rinkinys OCL įrenginių, kurie bus naudojami skaičiavimams atlikti.
- Branduoliai: OCL branduolio funkcijos, kurios bus vykdomos šių įrenginių.
- Programos objektai: programos išeities tekstas bei vykdymo objektai, kuriuose yra realizuotos branduolio funkcijos.
- Atminties objektai: aibė objektų atmintyje, kurie prieinami OCL įrenginiams ir kuriuose saugomi duomenys, su kuriais operuoja OCL darbo vienetai.

4.2.3 Komandų eilės

„Šeimininko“ bei OCL įrenginių interakcija realizuojama „šeimininkui“ siunčiant komandas į komandų eilę. Šios komandos laukia eilėje, kol jas įvykdys OCL įrenginys. Komandų eilę sukuria „šeimininkas“ ir susieja ją su vienu OCL įrenginiu. Yra trijų tipų komandos:

- Branduolio vykdymo komandos inicijuoja branduolio funkcijų vykdymą apdorojimo elementuose.
- Atminties komandos perkelia duomenis iš „šeimininko“ atminties į OCL atminties objektus bei atvirkščiai, perkelia duomenis tarp skirtingų OCL atminties objektų bei susieja (ir atvirkščiai) OCL atminties objektus su „šeimininko“ atmintimi.

Komandos, nusiųstos į tą pačią eilę, gali būti vykdomos dvejopai:

- Vykdymas iš eilės (angl. in-order execution): komandos pradedamos ir baigiamos vykdyti tokia tvarka, kokia jos buvo įdėtos į eilę.
- Vykdymas ne iš eilės (angl. out-of-order execution): komandos pradedamos vykdyti tokia tvarka, kokia jos buvo įdėtos į eilę, tačiau vėliau įdėtos komandos nelaukia, kol bus įvykdytos ankstesnės komandos. Sinchronizavimas turi būti atliekamas rankiniu

būdu. Šis rėžimas OCL realizacijoms nėra privalomas.

4.3 Atminties modelis

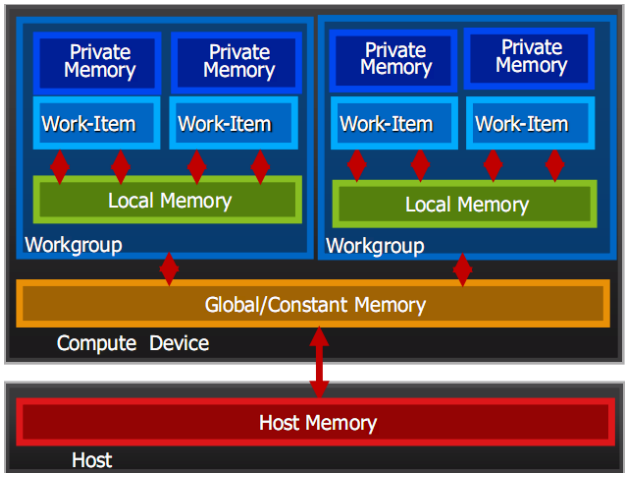
OCL standartas turi dviejų tipų atminties objektus: buferius (angl. buffer object) bei paveikslus (angl. image object). Buferis yra ištisinis atminties blokas, prieinamas OCL branduolyje. Programuotojas pats nusprendžia, kaip duomenys išdėstomi šiame buferyje bei gali prieiti prie šių objektų pasinaudodamas rodyklėms (angl. pointer).

Paveikslo objektai skirti laikyti paveikslėlius ir dažnai yra optimizuoti konkrečiam OCL įrenginiui. Todėl OCL standartas leidžia paveikslų objektus pritaikyti specifinei tam tikro OCL įrenginio architektūrai.

OCL atminties modelis apibrėžia penkis skirtingus atminties regionus:

- „Šeimininko“ atmintis: ši atmintis pasiekiama tik „šeimininko“ procesui. OCL standarte apibrėžti tik būdai, kaip ši atmintis sąveikauja su OCL objektais ir konstrukcijomis.
- Globali atmintis: visi darbo vienetai gali tiek skaityti iš tiek ir rašyti į šią atmintį. Duomenys gali būti talpinami į spartinančiąją atmintį (angl. cache), tačiau tai priklauso nuo konkretaus įrenginio.
- Konstantų atmintis: šiame regione esantys duomenys negali būti modifikuojami, kol yra vykdoma OCL branduolio funkcija. Atminties objektus, esančius šioje srityje, sukuria „šeimininkas“.
- Lokali atmintis: atmintis, matoma darbo grupės viduje. Ši atmintis gali būti naudojama darbo grupės vienetų bendrai naudojamiems duomenims saugoti bei tų vienetų tarpusavio komunikacijai realizuoti. OCL įrenginyje gali būti sumontuoti specialūs atminties įtaisai šiai atminčiai arba gali būti naudojamos globalios atminties sritys.
- Privati atmintis: šis atminties regionas pasiekiamas tik vienam darbo vienetui. Kintamieji, deklaruoti šioje srityje yra nematomi kitiems darbo vienetams.

OCL atminties modelis vaizduojamas 4 paveiksle.



4 paveikslas: OpenCL atminties modelis, atminties sričių hierarchija bei komunikavimo galimybės

4.4 Programavimo modeliai

OCL standartas sukurtas galvojant apie du skirtingus programavimo modelius: užduočių lygiagretumas bei duomenų lygiagretumas. Iš principo įmanoma ir šių dviejų modelių kombinacija.

4.4.1 Duomenų lygiagretumo programavimo modelis

Algoritmai, kurie gerai tinka prie šio modelio, koncentruoja programuotojo dėmesį prie duomenų struktūrų, kurių elementai gali būti apdorojami tuo pačiu metu. Iš esmės ta pati veiksmų seka yra vykdoma su visais struktūros elementais lygiagrečiai.

Šis programavimo modelis natūraliai tinka OpenCL programos vykdymo modeliui. Esminis dalykas čia yra apibrėžta ND-Sritis. Programuotojas turi susieti algoritmo duomenų struktūras su ND-Srities indeksų erdve ir tas duomenų struktūras pateikti OCL įrenginiui naudodamas atminties objektus. O branduolyje nurodomi veiksmai, kuriuos galima lygiagrečiai atlikti su duomenų struktūros elementais, kurie atitinka OCL darbo vienetus.

Sudėtingesnių algoritmų darbo vienetams toje pačioje grupėje gali reikėti dalintis duomenimis. OCL tokias dalybas palaiko panaudojant lokaliąją atmintį. Tačiau kiekvieną kartą, kuomet duomenų mainai yra reikalingi, programuotojas privalo užtikrinti, kad rezultatai bus teisingi nepriklausomai nuo darbo vienetų vykdymo tvarkos. Kitaip tariant reikia sinchronizuoti darbo vienetus toje pačioje grupėje.

Reikia paminėti, jog OCL neturi nuosavo mechanizmo sinchronizavimui tarp darbo vienetų skirtingose darbo grupėse. Jei reikia, tokį sinchronizavimą programuotojas turi sugalvoti pats

arba modifikuoti algoritmą, taip, kad darbo vienetai tarp skirtingų grupių būtų nepriklausomi.

Būtent duomenų lygiagretumas yra išnaudojamas siekiant sukurti lygiagrečius bangų modeliavimo algoritmus tiek nagrinėtuose literatūros šaltiniuose, tiek ir šiame darbe.

4.4.2 Užduočių lygiagretumo programavimo modelis

OCL užduotis apibrėžiama kaip branduolys, kuris vykdomas kaip vienas darbo vienetas nepriklausomai nuo ND-Srities, kurią naudoja kiti OCL programos branduoliai. Toks būdas gali būti naudojamas tuomet, kai programuotojas nori išnaudoti lygiagretumą užduoties viduje. Pavyzdžiui, lygiagretumas gali būti išreiškiamas naudojant vektorines operacijas apdoroti vektoriniams duomenų tipams.

Kitas būdas realizuoti šį modelį – naudoti „ne iš eilės“ vykdomą komandų eilę. Kai užduočių kiekis yra daug didesnis negu skaičiavimo vienetų kiekis OCL įrenginyje, tokia strategija gali būti labai palanki siekiant subalansuoto įrenginio apkrovimo.

Trečias lygiagrečių užduočių organizavimo būdas – jungti jas į užduočių grafus naudojant OCL įvykius (angl. events). Vienos komandos, esančios eilėje, gali generuoti įvykius, kitos – laukti šių įvykių atsiradimo prieš pradedant jas vykdyti. Kartu su „ne iš eilės“ vykdomų komandų eile šios priemonės OCL programuotojui leidžia konstruoti statinius grafus, kurių viršūnės yra užduotys, o briaunos – priklausomybės tarp šių viršūnių.

Daugiau informacijos apie OpenCL platformą, jos programavimo bei veikimo ypatumus galima rasti [18, 19] šaltiniuose.

5 Integravimo algoritmo optimizavimas OpenCL platformoje

5.1 Pradinis nuoseklus algoritmas ir modelio apribojimai

Realizuojant tampriųjų bangų modelio skaičiavimo algoritmą naudojant bet kokią programinę ir techninę įrangą reikia atkreipti dėmesį į faktą, jog tokios pačios formos ir orientacijos elementai, turintys tokį patį medžiagą aprašančių koeficientų rinkinį (Jungo modulis, Puasono koeficientas, medžiagos tankis) aprašomi identiškomis standumo bei įtempimų matricomis. Todėl pakanka apskaičiuoti šias matricas vieną kartą ir jas pakartotinai naudoti perskaičiuojant elementų mazgus veikiančias jėgas. Todėl pats paprasčiausias modelis pasižymi tokiomis savybėmis:

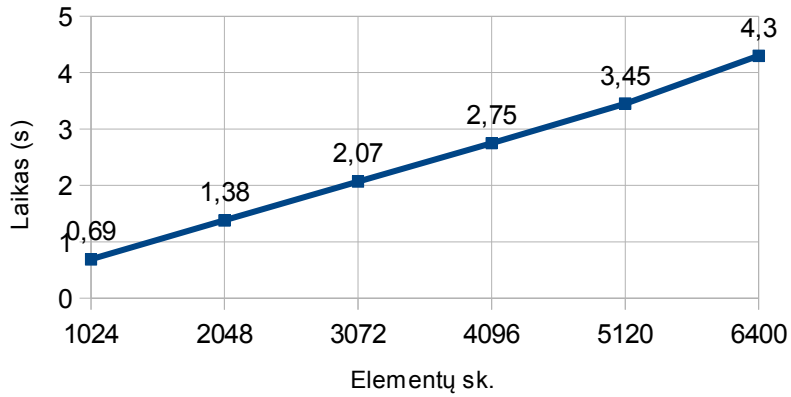
- Kontinuumo sritis yra stačiakampė.

- Sritis yra padalinta vienodais stačiakampio formos elementais.
- Visoje srityje minėtieji medžiagą aprašantys parametrai yra vienodi.

Nuoseklus tokio modelio lygties integravimo algoritmas yra toks:

1. Apskaičiuojamos elemento standumo ir įtempimų matricos. Jos išsaugomos atmintyje.
2. Perskaičiuojamos konstrukcijos mazgus veikiančios jėgos:
   a) mazgų jėgų vektoriui priskiriamos mazgų virpesius slopinančių jėgų reikšmės
   b) jei reikia, prie atitinkamų mazgų jėgų vektoriaus elementų pridedamos išorinės (sužadinimo) jėgos
   c) pridedamos dėl deformuotų elementų atsiradusios įtempimo jėgos
3. Jei baigus iteraciją reikės pamatyti srities įtempimų vaizdą, perskaičiuojami kiekvieno elemento įtempimai.
4. Perskaičiuojami mazgų greičiai, poslinkiai.
5. 2-4 žingsniai kartojami tiek, kiek norima stebėti bangos sklidimo procesą arba kol virpesiai išnyksta.

Reikia pastebėti, jog tokia bazinė implementacija reikalauja saugoti ne tik mazgų poslinkių, greičių, jėgų, elementų įtempimų vektorius, bet dar papildomai ir elementams priklausančių mazgų indeksus, nes CPU per daug laiko praleistų kiekvienam elementui skaičiuodamas šiuos indeksus. Kaip matoma 1-ame grafike, tokia algoritmo realizacija (C++ programavimo kalba) nėra itin sparti.



1 grafikas: Skaičiavimo laikas, kurio reikia atlikti 1000 iteracijų naudojant vieną CPU branduolį bei netaikant jokių specialių optimizacijų ir instrukcijų.

5.2 Sandaugų $[K_e]\{U\}$ ir $[G_e]\{U\}$ apskaičiavimas

Žinoma, jog elemento standumo matricos $[K_e]$ išmatavimai – 8x8, $[G_e]$ – 3x8, o vektoriaus $\{U_e\}$ ilgis – 8 elementai. Sandaugos $[K_e]\{U_e\}$ apskaičiavimui reikia atlikti $8 \cdot 8 = 64$ daugybos operacijas bei $7 \cdot 8 = 56$ sudėties operacijas. Atitinkamai $[G_e]\{U_e\}$ radimui reikia 24 daugybos operacijų bei 21 sudėties operacijos.

Vienas iš būdų šias sumas apskaičiuoti – naudoti ciklus. Tačiau ciklas dar prideda papildomų operacijų: indekso didinimas kiekvienos iteracijos metu, ciklo baigimo sąlygos tikrinimas. Net ir šių nedidelių sandaugų radimui tai dar papildomai pridėtų 72 indekso didinimo ir tiek pat ciklų baigimo sąlygos tikrinimo operacijų $[K_e]\{U_e\}$ atveju bei 32 indekso didinimo ir ciklo baigimo sąlygos operacijų $[G_e]\{U_e\}$ atveju. Žinant, jog šios sandaugos ($[G_e]\{U_e\}$ atliekama tik kas kelis šimtus iteracijų) skaičiuojamos kiekvienam elementui, nesunku pastebėti, jog net ir nedidelis modelis reikalauja atlikti daug perteklinių veiksmų. Pavyzdžiui, jei srityje yra 1000 elementų, tuomet kiekvieną iteraciją procesoriui skaičiuojant $[K_e]\{U_e\}$ sandaugas tenka atlikti 72000 indekso didinimo bei tiek pat sąlygos tikrinimo operacijų. Jei iteracijos metu reikia apskaičiuoti ir elemento įtempimus, tai dar papildomai reikalauja 64000 operacijų. Todėl yra tikslinga šių dviejų sandaugų veiksmus programuoti nenaudojant ciklų. Tokia ciklų eliminavimo procedūra dar vadinama ciklų išskleidimu (angl. loop unrolling). Priklausomai nuo naudojamos programavimo kalbos ar technologijos, šį veiksmą gali atlikti ir preprocesorius arba kompiliatorius.

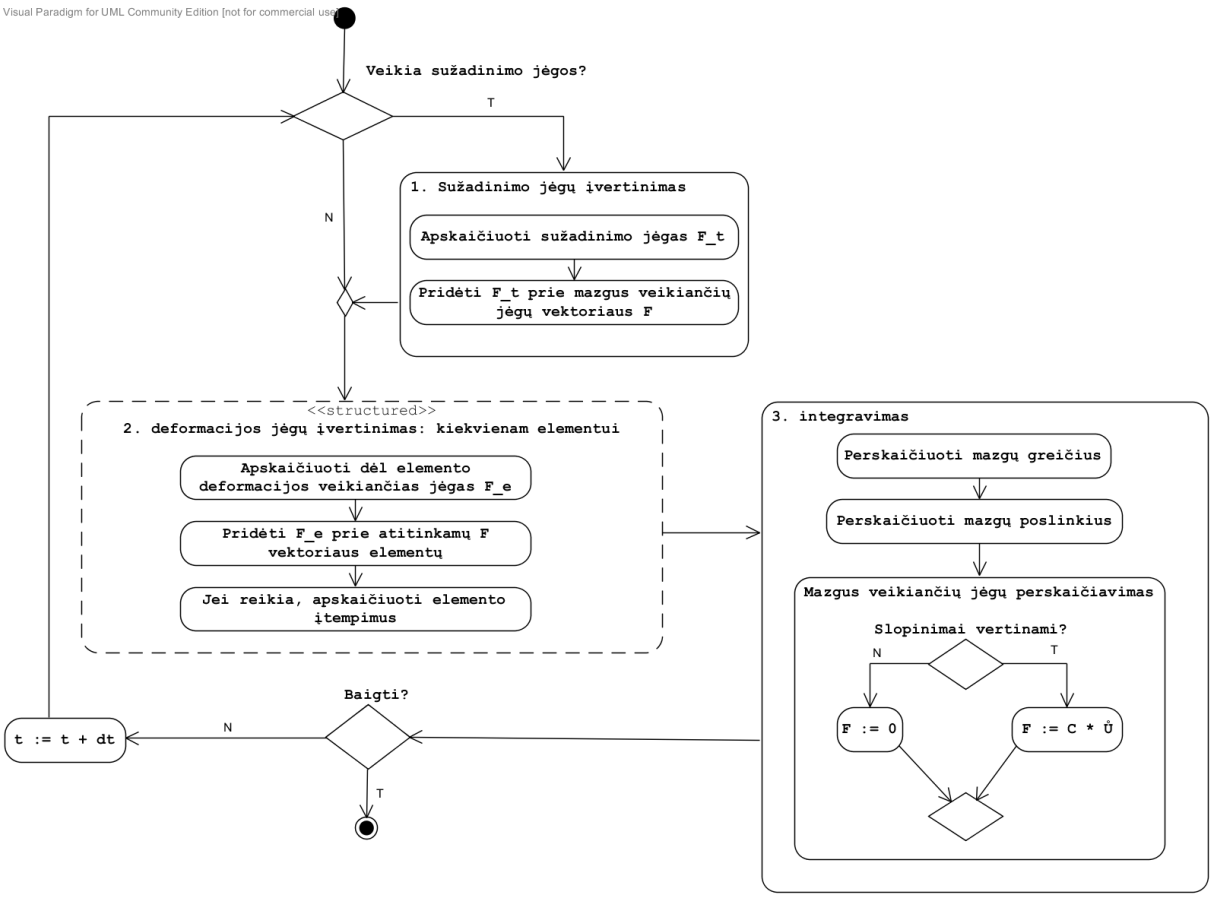
Kyla klausimas, ar negalima kaip nors paspartinti naudingų operacijų vykdymo? CPU ir GPU procesoriai turi specialius vektorinius įtaisus, kurie gali vienu metu tą patį veiksmą (daugybos arba sudėties) atlikti su keliais slankaus kablelio skaičiais iš karto (paprastai 4, 8, 16 ir t.t.). Norint pasinaudoti tokiomis procesorių galimybėmis, tenka naudoti OCL C programavimo kalboje esančius vektorinius duomenų tipus. Kadangi $[K^e]$ ir $[G^e]$ matricos turi po 8 elementus eilutėje, vieną šių matricų eilutę galima sutalpinti į 8 elementų ilgio slankaus kablelio skaičių duomenų tipo kintamąjį (float8). Beje, OCL C kalboje sumos, sandaugos bei kiti operatoriai leidžia labai kompaktiškai užrašyti veiksmus su vektoriniais duomenų tipais, todėl minėtas ciklų išskleidimas net ir rankiniu būdų atliekamas labai paprastai.

5.3 Visos srities lygiagretus apdorojimas be jokio sinchronizavimo tarp elementų

Prieš nagrinėjant srities dinamikos lygties integravimo algoritmo lygiagretinimo galimybes, modifikuokime standartinę CSM schemą:

1. $\dot{U}_{n-\frac{1}{2}}, U_n, F_n$
2. $F_n = F_n + f(U_n, t)$
3. $\dot{U}_{n+\frac{1}{2}} = \dot{U}_{n-\frac{1}{2}} + (t_{n+\frac{1}{2}} - t_{n-\frac{1}{2}})[M]^{-1}\{F_n\}$
4. *Kraštinių sąlygų įvertinimas* $(jeigu\ reikia)$
5. $U_{n+1} = U_n + (t_{n+1} - t_n)\dot{U}_{n+\frac{1}{2}}$
6. $F_{n+1} = f(\dot{U}_{n+\frac{1}{2}})$

Tokia integravimo schema leidžia pakeisti ir integravimo algoritmą:



5 paveikslas: srities skaitinio integravimo algoritmas paremtas modifikuota CSM schema. Čia F – mazgus veikiančių jėgų vektorius, C – mazgų slopinimo koeficientų vektorius, $\dot{U}$ - mazgų greičių vektorius, t – simuliacijos laikas.

Vienas lygties integravimo žingsnis padalintas į 3 etapus:

1. Išorinių jėgų poveikio įvertinimas konstrukcijos mazgams: šis etapas vykdomas tik pradiniuose simuliacijos žingsniuose, todėl jo optimizuoti neverta.
2. Jėgų, atsiradusių dėl elementų deformacijos, apskaičiavimas: šis etapas reikalauja daugiausiai procesoriaus laiko (maždaug 4 kartus daugiau, nei integracijos etapas).

Esminis šio etapo skirtumas nuo kitų dviejų – apdorojami **elementai**, tačiau kiekvienas elementas turi 4 mazgus, ir šiame etape reikia prieiti prie dviejų kiekvieno mazgo kintamųjų: mazgą veikiančios jėgos bei poslinkio.

3. Integravimas: šio etapo metu dėl pasikeitusių jėgų, veikiančių konstrukcijos mazgus, perskaičiuojami jų greičiai, poslinkiai. Taip pat mazgus veikiančių jėgų vektorius yra išvalomas ir apskaičiuojamos slopinimo jėgos. Šiame etape apdorojami **mazgai**, kurie yra nepriklausomi vienas nuo kito, todėl šio etapo lygiagretinimas yra nesudėtingas, galima sakyti, net trivialus.

5.3.1 Jėgų, atsiradusių dėl įtempimų elementuose, įvertinimo lygiagretus algoritmas (1 algoritmas)

Kaip jau buvo minėta literatūros apžvalgoje, Komatitsch D. [1] pasiūlyta algoritmo modifikacija – bendrų mazgų neturinčių elementų apdorojimas vienu metu. Dvimačiu atveju tai reiškia, jog reikia atlikti 4 iteracijas. Kiekvienos tokios iteracijos metu apdorojama vis kita nepriklausomų elementų grupė (6 paveikslas).

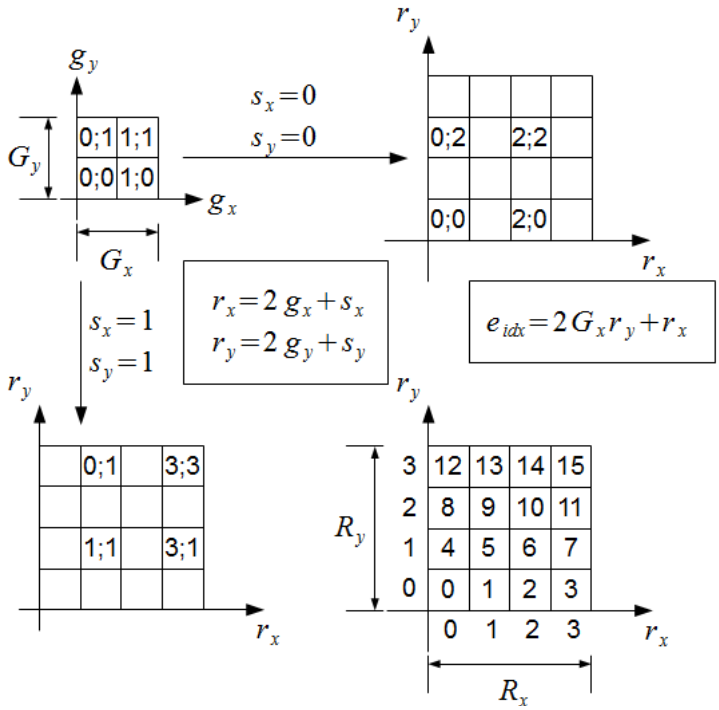
3	2	3	2	3	2	3	2
0	1	0	1	0	1	0	1
3	2	3	2	3	2	3	2
0	1	0	1	0	1	0	1
3	2	3	2	3	2	3	2
0	1	0	1	0	1	0	1
3	2	3	2	3	2	3	2
0	1	0	1	0	1	0	1

6 paveikslas: elementai, pažymėti tuo pačiu skaičiumi apdorojami lygiagrečiai

Toks algoritmas OCL platformoje reiškia, jog:

- Procesorius vykdo tą pačią branduolio funkciją 4 kartus, tačiau apdoroja elementus, turinčius skirtingus indeksus.
- Elementų indeksų erdvė yra 4 kartus mažesnė nei reali modelio elementų indeksų erdvė.

Dėl šių dviejų priežasčių, nustatant gijos apdorojamo elemento indeksą, tenka atlikti perskaičiavimus, kurie parodyti 7-ame paveiksle:



7 paveikslas: elemento indekso skaičiavimo procedūra

Čia G_x, G_y – darbo elementų skaičius x ir y dimensijose, g_x, g_y – darbo elemento koordinatės, R_x, R_y – realaus modelio (srities) išmatavimai elementais, r_x, r_y – elemento koordinatės srityje, e_{idx} – elemento indeksas elementų mazgų vektoriuje (vienmatė indeksų erdvė).

5.3.2 Integravimo etapo lygiagretinimas

Šiame etape atliekami tokie veiksmai su modelio duomenų vektorių elementais:

1. $\dot{U}[i]:=\dot{U}[i]+F[i]\cdot M^{-1}[i]\cdot\Delta t$
2. $U[i]=U[i]+\dot{U}[i]\cdot\Delta t$
3. $F[i]:=C[i]\cdot\dot{U}[i]$

1. $\dot{U}[i]:=\dot{U}[i]+F[i]\cdot M^{-1}[i]\cdot\Delta t$

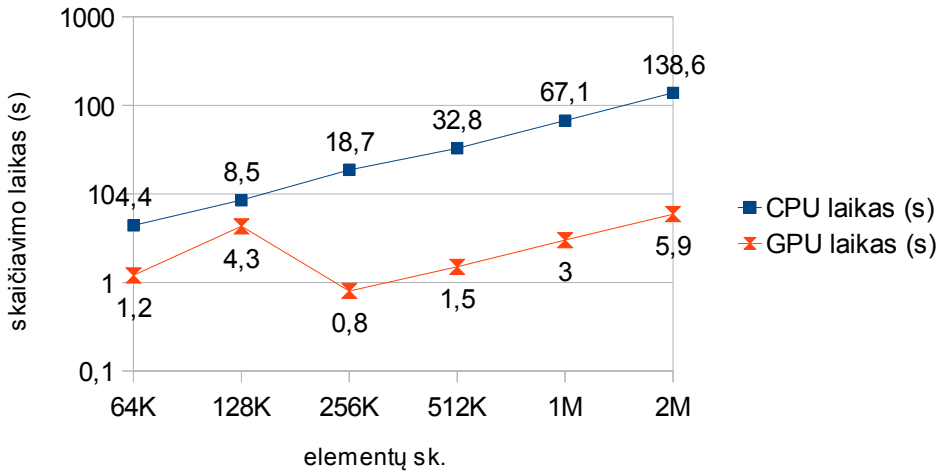
2. $U[i]=U[i]+\dot{U}[i]\cdot\Delta t$

3. $F[i]:=C[i]\cdot\dot{U}[i]$

Kadangi šiame etape apdorojami mazgai ir jie vienas nuo kito nepriklauso, pakanka procesoriui pateikti vienmatėje indeksų erdvėje vykdyti vieną OCL C branduolio funkciją, kurioje šios formulės užrašomos tiesiogiai. Taip kiekvieną mazgą apdoroja vis kita gija.

5.3.3 Pradinio lygiagretaus algoritmo realizacijos OCL platformoje rezultatai

Pritaikius 4.2 skyriuje aprašytą $[K_e]\{U_e\}$ bei $[G_e]\{U_e\}$ sandaugų skaičiavimo metodą bei realizavus aprašytą lygiagretų konstrukcijos lygties integravimo algoritmą OCL platformoje, buvo pasiektas 10 kartų pagreitėjimas naudojant abu CPU branduolius, lyginant su nuoseklia neoptimizuota algoritmo realizacija. Beje, reikia pastebėti vieną svarbų OCL platformos privalumą: tą patį OCL C programos kodą galima naudoti įvairiems sistemoje esantiems procesoriams, o OCL platforma suprojektuota taip, kad automatiškai būtų išnaudojami visi procesoriuje esantys branduoliai. 2-ame grafike yra lyginami CPU ir GPU procesorių skaičiavimų laikai.



2 grafikas: pradinio lygiagretaus algoritmo 1000 iteracijų skaičiavimo laiko priklausomybė nuo elementų skaičiaus

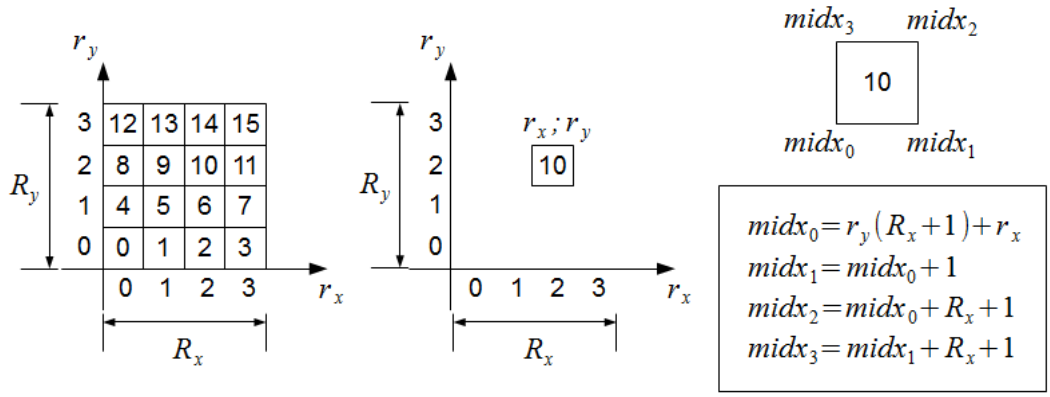
Matome, jog GPU procesorius taikant anksčiau aprašytą algoritmą 1000 iteracijų atlieka maždaug 22 kartus greičiau, nei CPU. Pagrindinė to priežastis – didelis GPU esančių vektorinių branduolių skaičius: Radeon HD6850 jų turi 192 (12 nepriklausomų didesnių branduolių kurių kiekvienas turi tas pačias instrukcijas vykdančius 16 vektorinių branduolių). Svarbu yra ir tai, jog vienas toks nepriklausomas branduolys pakaitomis per 4 taktus apdoroja $16 \cdot 4 = 64$ gijas. Tuo tarpu CPU turi 2 branduolius kurių kiekvienas turi tik 2 vektorinius įtaisus.

Pastebėkime, jog esant nedideliam modeliui (iki 128000 elementų imtinai) GPU našumas yra žymiai mažesnis, nei vykdant skaičiavimus didesniuose modeliuose: šis procesorius CPU lenkia tik maždaug 3 kartus. Toks skirtumas aiškinamas tuo, jog nedidelis modelis nepakankamai apkrauna GPU resursus ir šio procesoriaus našumas krenta. Optimizuojant algoritmus grafiniams procesoriams reikia atkreipti dėmesį į vieną iš svarbiausių šių

procesorių savybių: nors procesorių ir atmintį jungiančios magistralės pralaidumas yra didelis (Radeon HD6850 - 128 GB/s) lyginant su CPU, reikia kelių šimtų taktų (Radeon HD6850 – nuo 300 iki 600), kol duomenys iš atminties patenka į procesoriaus registrus [19]. Taigi šis laiko šuolis atsiranda būtent dėl to, jog procesorius turi per mažai vienu metu veikiančių gijų, kad duomenų mainus tarp atminties ir procesoriaus būtų galima paslėpti skaičiavimais.

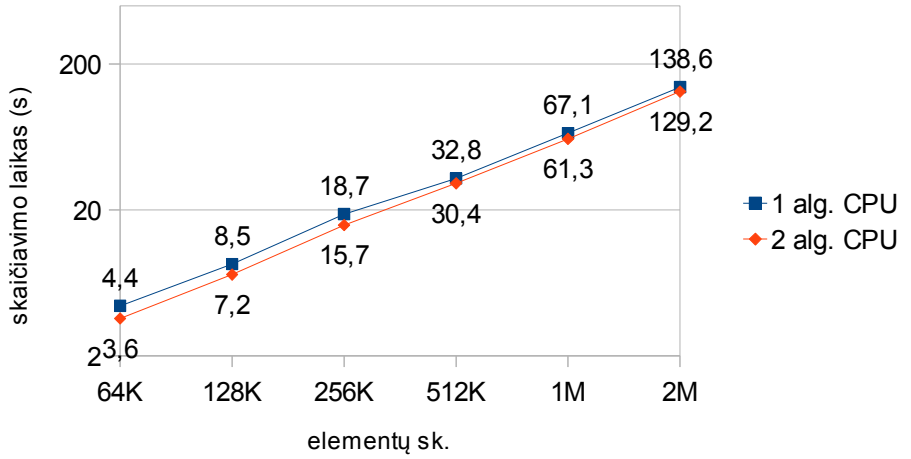
5.3.4 Elementų mazgų indeksų masyvo eliminavimas (2 algoritmas)

Prisiminkime vieną iš darbo pradžioje nustatytų apribojimų nagrinėjamam modeliui: visa sritis padalinta vienodais stačiakampio formos elementais. Šis apribojimas leidžia atsisakyti elementų mazgų indeksų masyvo. Žinoma, tokiu atveju 2-ajame srities dinamikos lygties integravimo etape – įvertinant dėl elementų įtempimų atsiradusias jėgas – tenka papildomai skaičiuoti šiuos indeksus, tačiau nebereikia jų paimti iš atminties. Konkrečiam elementui priklausančių mazgų indeksų apskaičiavimas parodytas 8-ame paveiksle.

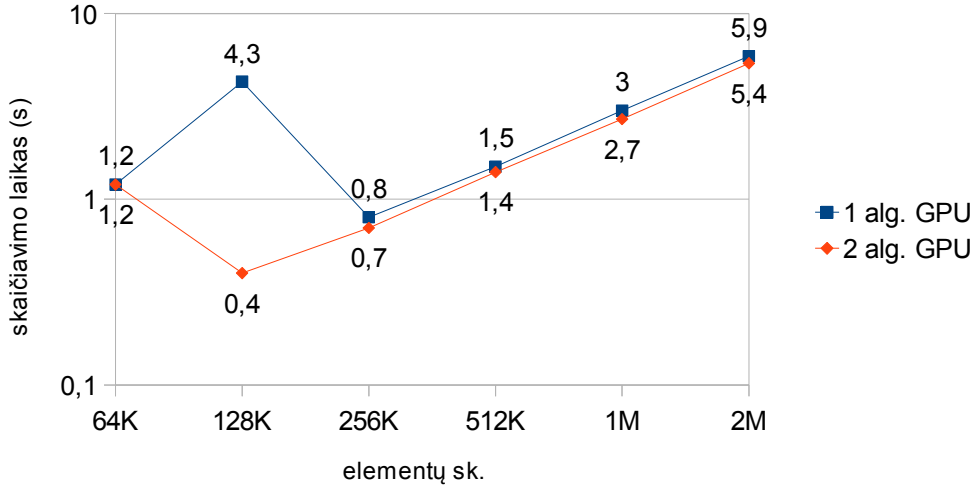


8 paveikslas: elementui priklausančių mazgų indeksų apskaičiavimas

R_x, R_y – realaus modelio (srities) išmatavimai elementais, r_x, r_y – elemento koordinatės srityje, $midx_{[0..3]}$ – elementui priklausančių mazgų indeksai. 3-iame ir 4-ame grafikuose pateikti modifikuoto algoritmo skaičiavimo laikai lyginami su 1-mojo algoritmo rezultatais naudojant skirtingus procesorius.



3 grafikas: 1-ojo ir 2-ojo algoritmo 1000 iteracijų vykdymo laiko priklausomybė nuo elementų skaičiaus (CPU)



4 grafikas: 1-ojo ir 2-ojo algoritmo 1000 iteracijų vykdymo laiko priklausomybė nuo elementų skaičiaus (GPU)

Iš rezultatų matoma, jog tokia nesudėtinga optimizacija leidžia skaičiavimų laiką sumažinti:

- 16% apdorojant sritis iki 256000 elementų naudojant CPU;
- 7% apdorojant sritis didesnes nei 256000 elementų naudojant CPU;
- 8% apdorojant sritis sudarytas iš 256000 elementų ir daugiau naudojant GPU;

Taip pat atkreipkime dėmesį ir į tai, jog ši optimizacija GPU našumo šuolį perstūmė nuo

128000 iki 64000 elementų ribos. Tai aiškinama tuo, jog skaičiuojant mazgų indeksus GPU nereikia jų paimti iš atminties ir dėl to efektyviau išnaudojama atminties magistralė: procesorius per tą patį laiką greičiau gali pasiekti naudingus duomenis.

5.3.5 1-mojo ir 2-mojo algoritmo trūkumai

Šie du algoritmai, nors ir leidžia paprastai ir efektyviai paspartinti skaičiavimus, turi 2 didelius trūkumus:

1. Mazgų duomenys iš atminties nuskaitomi ir rašomi „su tarpais“. Toks duomenų nuskaitymas ir įrašymas nėra optimalus.
2. Kadangi apdorojami elementai, kurie neturi bendrų mazgų, kiekvieno mazgo duomenys nuskaitomi ir atnaujinami 4 kartus. Idealioje situacijoje tos pačios reikšmės turėtų būti nuskaitomos ir atnaujinamos vieną kartą.

Šias problemas (bent dalinai) pašalina 3-iasis algoritmo variantas, aprašytas kitame skyriuje.

5.4 Srities apdorojimas blokais (3-iasis algoritmas)

Norint išvengti skylėto atminties nuskaitymo ir rašymo, tenka skaidyti visą sritį į blokus, ir vienu metu apdoroti šalia esančius elementus. Tokiu atveju neišvengiama sinchronizavimo tarp bloką apdorojančių gijų.

Sprendžiant kitą problemą – perteklinį duomenų nuskaitymą ir rezultatų įrašymą – galima pasinaudoti Micikevičiaus P. pasiūlymu laikyti reikšmes registruose ir jas perstumti [12]. Nors šį būdą autorius naudojo trimatės erdvės skaičiavimuose, toks pats principas gali būti panaudojamas ir dvimatėje erdvėje.

Komatitsch D. [1] pasiūlymas apdoroti bendrų mazgų neturinčius elementus gali būti panaudojamas apdorojant blokus. Principas išlieka tas pats: vienu metu apdorojami nesiliečiantys blokai.

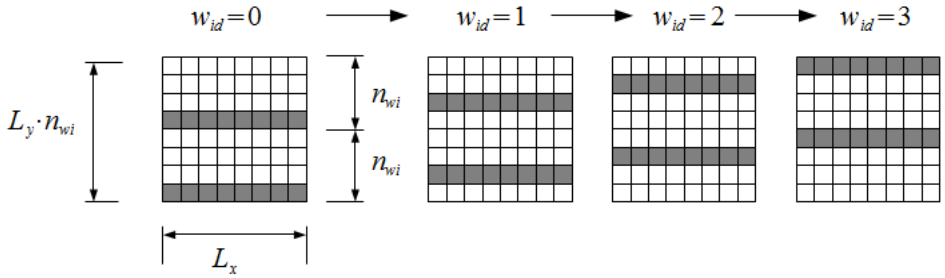
Kaip ir 4.1.1 skyriuje aprašyto algoritmo atveju, taip ir apdorojant sritį blokais tenka skaičiuoti gijai priklausančių elementų indeksus. Tik čia elementų realias koordinates dvimatėje erdvėje bei jų indeksus apskaičiuoti yra šiek tiek kebliau:

$$r_x = (2w_x + s_x) \cdot L_x + l_x;$$
$$r_y = ((2l_y + s_y) \cdot L_y + l_y) \cdot n_{wi} + w_{id}$$

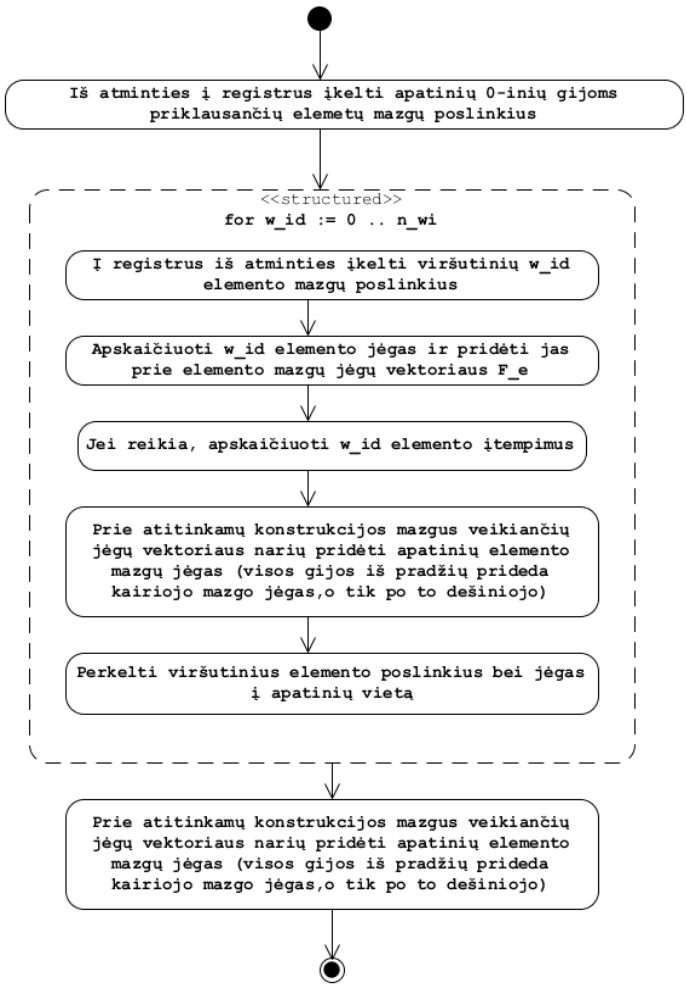
Čia r_x, r_y – elemento koordinatės srityje, w_x – apdorojamo bloko x koordinatė,

$s_x, s_y \in [0,1]$ – bloko postūmis, L_x, L_y – darbo vienetų (bloką apdorojančių gijų) grupės x ir y koordinatės), l_x, l_y – vienos gijos koordinatės gijų grupėje, n_{wi} – vienai gijai tenkantis elementų skaičius, $w_{id} \in [0..n_{wi}]$ – gijos apdorojamo elemento identifikatorius.

9-ame paveiksle pavaizduotas vieno bloko elementų apdorojimo procesas, o 10-ame tokio bloko elementų įtampos jėgų įvertinimo algoritmas.



9 paveikslas: bloko, kurio realus dydis - 8x8 elementų - apdorojimas. Vienu metu skirtingų gijų apdorojami elementai pažymėti pilkšva spalva. Tokiam blokui apdoroti reikia $L_x \cdot L_y = 8 \cdot 2 = 16$ *gijų. Kadangi gijos apdoroja gretimus elementus, iš atminties skaitomi ir į ją rašomi duomenų blokai yra be tarpų.*



10 paveikslas: bloko elementų apdorojimo algoritmas naudojant registrų perstūmimą.

Elementų apdorojimas blokais iškelia klausimą: kiek elementų turi sudaryti bloką? Bloko dydis parenkamas įvertinus techninės įrangos pajėgumus: kadangi Radeon HD6850 procesoriaus kiekvienas branduolys vienu metu apdoroja 64 gijas, natūralu, jog tiek gijų turi apdoroti elementus, priklausančius tam pačiam blokui. Eksperimentai parodė jog jei gijų grupę sudaro tas pats gijų skaičius, jos išmatavimai pastebimo poveikio nedaro.

Toliau tenka nustatyti optimalų vienai gijai tenkantį elementų skaičių n_{wi} . 1-oje lentelėje pateikiami skaičiavimų laikai sekundėmis (naudojant GPU procesorių) naudojant skirtingą n_{wi} reikšmę skirtingo dydžio modeliuose.

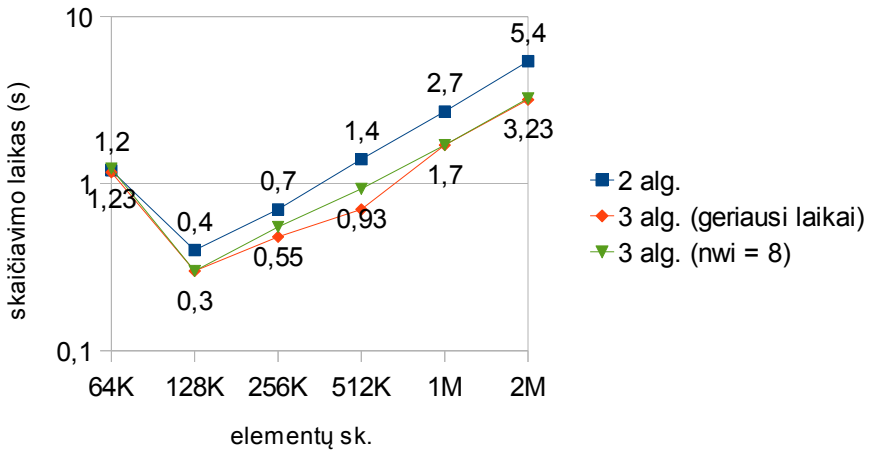
1. lentelė:1000 integravimo iteracijų vykdymo laiko (sekundėmis) priklausomybė nuo vienai gijai tenkančių elementų skaičiaus bei modelio dydžio

n_{wi}	Modelio dydis (elementų skaičius)					
	64K	128K	256K	512K	1M	2M
2	**1,17**	**0,3**	0,55	0,7	1,85	3,63
4	**1,17**	**0,3**	**0,48**	**0,92**	1,72	3,31
8	1,23	**0,3**	0,55	0,93	**1,7**	3,23
16	1,37	0,44	0,57	1,03	**1,7**	3,27
32	1,53	0,59	0,7	0,95	1,8	**3,18**

1-oje lentelėje kiekviename stulpelyje paryškinti mažiausiai procesoriaus laiko reikalaujančios n_{wi} parametro reikšmės. Nors keičiant šį parametrą skirtumai tarp tokio pačio modelio dydžio skiriasi nelabai žymiai (kiek ryškesni skirtumai pastebimi skaičiavimus atliekant mažesnėse srityse), pastebima gana aiški tendencija: kuo didesnis modelis, tuo didesnė n_{wi} reikšmė sąlygoja trumpesnį skaičiavimų laiką. Kitaip tariant, kuo didesnis modelis, tuo didesniais blokais turėtų būti suskaidoma nagrinėjama sritis.

Taip pat reikia pastebėti, jog įmanoma n_{wi} parametrą parinkti per didelį. Kaip jau buvo minėta anksčiau, jei GPU procesorius neturi pakankamai aktyvių gijų, jo našumas krenta. O n_{wi} reikšmės didinimas mažina aktyvių gijų skaičių.

11-ame paveiksle lyginami naujojo (3-iojo) algoritmo skaičiavimų laikai su 4.3.4 skyriuje pateikto algoritmo (2-ojo) skaičiavimo laikais.



5 grafikas: 2-ojo ir 3-iojo algoritmo variantų 1000 iteracijų reikalingo GPU procesoriaus laiko priklausomybė nuo modelio dydžio.

Matome, jog 3-iasis algoritmo variantas yra apie 1,6 karto spartesnis naudojant GPU procesorių lyginant su 2-uoju. Jeigu palygintume naujojo algoritmo GPU procesoriaus

skaičiavimo laikus su 2-ojo algoritmo CPU skaičiavimo laikais, pastebėtume, jog dar kartą optimizuotas algoritmas veikiantis GPU procesoriuje yra ~35 kartus spartesnis už 2-ąjį algoritmą CPU procesoriuje.

Pagrindinė priežastis, kodėl nėra 3-čiojo algoritmo skaičiavimo laikų naudojant CPU procesorių yra ta, jog optimizacijos, įsiūtos šiame algoritme yra pritaikytos GPU procesoriui. Todėl CPU toks algoritmo variantas nebetinka: eksperimentai parodė, jog net ir turint nedidelį modelį (64000 elementų), CPU skaičiavimo laikas išauga tiek, jog jo net neverta lyginti su ankstesniais variantais.

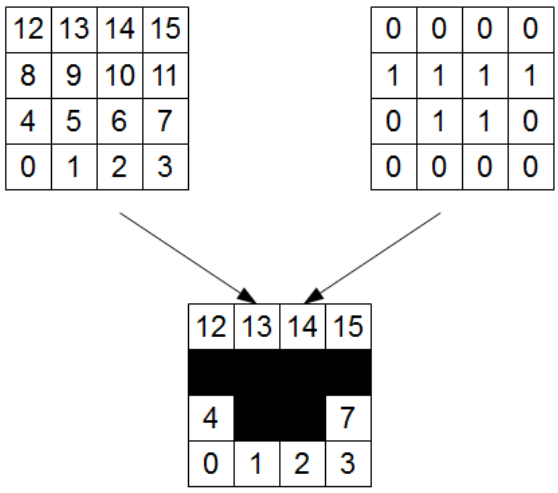
5.5 Įtrūkimų bei nereguliarių elementų įtraukimas į modelį (4-asis algoritmas)

Iki šiol buvo tobulinamas visiškai homogeninės (t.y. visi elementai yra vienodo dydžio, medžiagą aprašantys parametrai – Jungo modulis, Puasono koeficientas, medžiagos tankis, elemento aukštis – yra vienodi) srities integravimo algoritmas. Tačiau toks modelis daug praktinės naudos neturi. Juk visų NDT metodų esmė – įtrūkimų, priemaišų aptikimas. Todėl reikia turimą algoritmą patobulinti taip, kad būtų galima modeliuoti tokias anomalijas.

Logiška pasinaudoti 3-iojo algoritmo išskirtine savybe – elementų apdorojimu blokais. Kiekvienas blokas gali turėti skirtingus medžiagos parametrus. Kitaip tariant kiekvienam blokui galima priskirti vis kitą K^e ir G^e matricų porą: apskaičiuojant šias matricas reikia naudoti skirtingus medžiagą aprašančius parametrus.

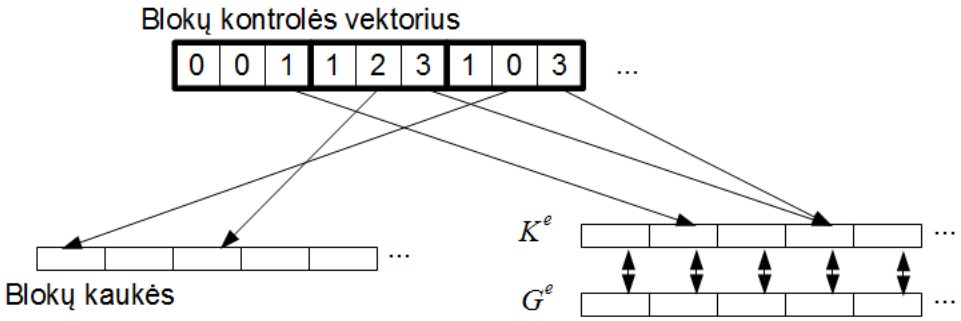
Pastebėkime, jog šių skirtingų matricų naudojimas atskiriems blokams leidžia modeliuoti ir neveikiančių elementų grupes. Iš tiesų jei K^e ir G^e matricos yra nulinės, jomis aprašomi elementai nebeveikia: apskaičiuojant elemento mazgus veikiančias jėgas jos gaunamos nulinės. Taigi šis algoritmo pakeitimas leidžia kurti ir stačiakampio formos skyles nagrinėjamoje srityje. Norint modeliuoti mažesniu įtrūkimus ar bet kokios formos įtrūkimus, reikia kitokio būdo.

Pasirinkime fiksuotą bloko dydį: $16 \cdot 32 = 512$ elementų. Dabar galima sudaryti bloko bitų „kaukę“, kurioje 0 reiškia, jog jį atitinkantis elementas yra įtrūkimo dalis (jis neveikia), o 1 – elementas veikia ir nėra įtrūkimo dalis. Tokiai bitų kaukei aprašyti (pasirinktam bloko dydžiui) reikia 16 sveikųjų skaičių (vienas sveikasis skaičius užima 32 bitus), o tai yra tik papildomi 64 baitai. 11-ame paveiksle pavaizduotas šios kaukės veikimo principas.



11 paveikslas: bloko bitų kaukės veikimo principas: juodai nuspalvinti elementai neveikia (neįtakoja mazgų poslinkių)

Be abejo, tik nedaugelis blokų turi turėti įtrūkimus, todėl įvedamas papildomas blokų kontrolės apdorojimo vektorius, kurio kiekvienas elementas (kurių iš viso reikia tiek, kiek yra blokų) turi 3 sveikuosius skaičius: 0 arba 1 priklausomai nuo to ar blokas turi bent vieną įtrūkimą ar ne, įtrūkimų kaukės indeksas, K^e ir G^e matricų indeksas (12 paveikslas).



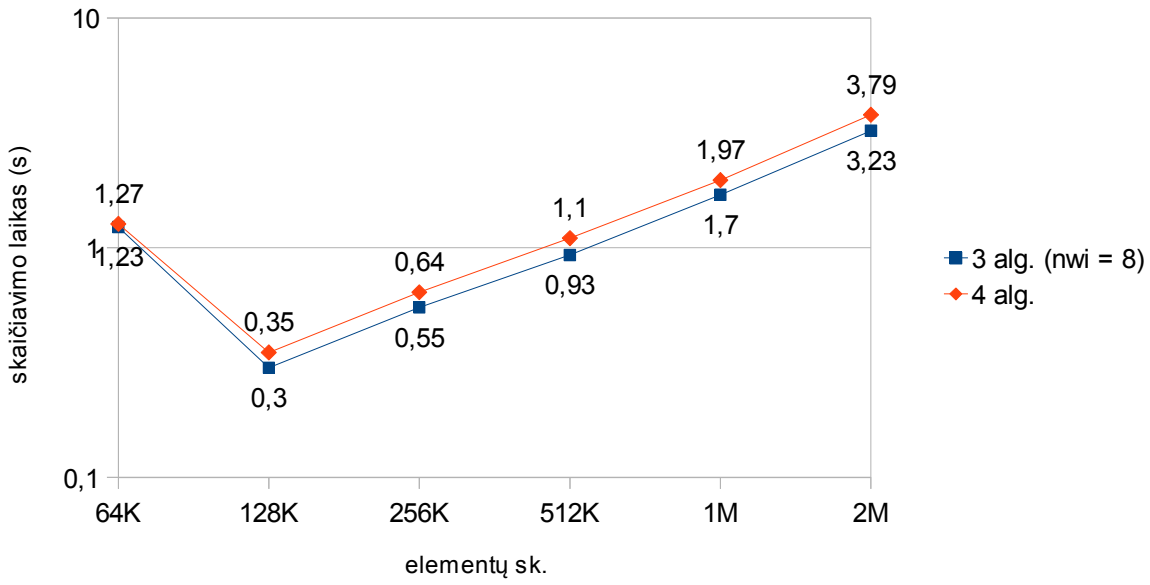
12 paveikslas: blokų kontrolės vektoriaus (masyvo) veikimo principas

Realizuojant tokį algoritmą OCL C kalba GPU procesoriui, reikia atkreipti dėmesį į tai, jog tą patį bloką apdorojančios gijos gali dalintis duomenimis. Kadangi to pačio bloko elementai aprašomi tomis pačiomis K^e ir G^e matricomis, prieš atliekant skaičiavimus jas reikia įkelti į lokaliąją atmintį, kad gijos greičiau galėtų pasiekti duomenis. Tą patį reikėtų padaryti ir su bloko kauke. 2-oje lentelėje ir 6-ame grafike pateikti algoritmo realizacijos OCL platformoje naudojant GPU procesorių skaičiavimo rezultatai.

2 lentelė: 3-iojo ir 4-ojo algoritmų 1000 iteracijų skaičiavimo laikų palyginimas naudojant GPU procesorių

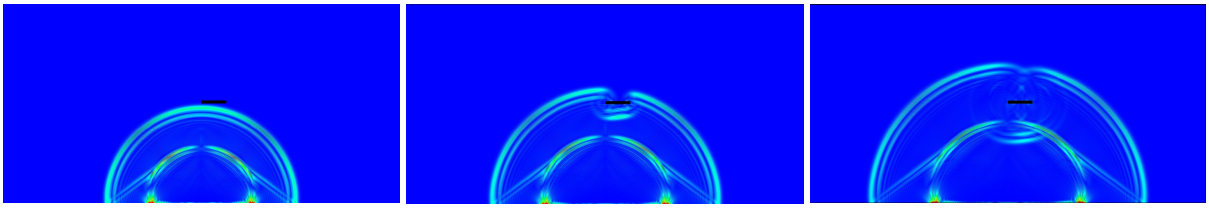
Elementų sk.	3 alg. (n_{wi} = 8)	4 alg. (be įtrūkimų)	4 alg. (su įtrūkimais)
64K	1,23	1,27	1,28
128K	0,3	0,35	0,35
256K	0,55	0,64	0,65
512K	0,93	1,1	1,1
1M	1,7	1,97	1,99
2M	3,23	3,79	3,78

2-osios lentelės 2-ame ir 3-iame stulpeliuose atitinkamai pateikti 4-ojo algoritmo skaičiavimo rezultatai į modelį neįtraukiant jokių įtrūkimų (2-asis stulpelis) ir kuomet visi elementai yra neveikiantys (3-iasis stulpelis). Nors iš pastarojo modelio jokios praktinės naudos nėra, tačiau išmatuoti skaičiavimų laikai šiais dviem atvejais rodo, jog kaukės įkėlimas ar neįkėlimas (priklausomai nuo to, ar blokas turi įtrūkimų ar ne) į atmintį prieš apdorojant elementų bloką neturi jokios įtakos skaičiavimų trukmei. Kadangi K_e ir G_e matricos taip pat įkeliamos į laikinąją atmintį, galima daryti išvadą, jog ši procedūra taip pat pastebimos įtakos skaičiavimo laikui neturi.



6 grafikas: 3-iojo ir 4-ojo algoritmo 1000 iteracijų skaičiavimo laiko priklausomybė nuo elementų skaičiaus naudojant GPU procesorių

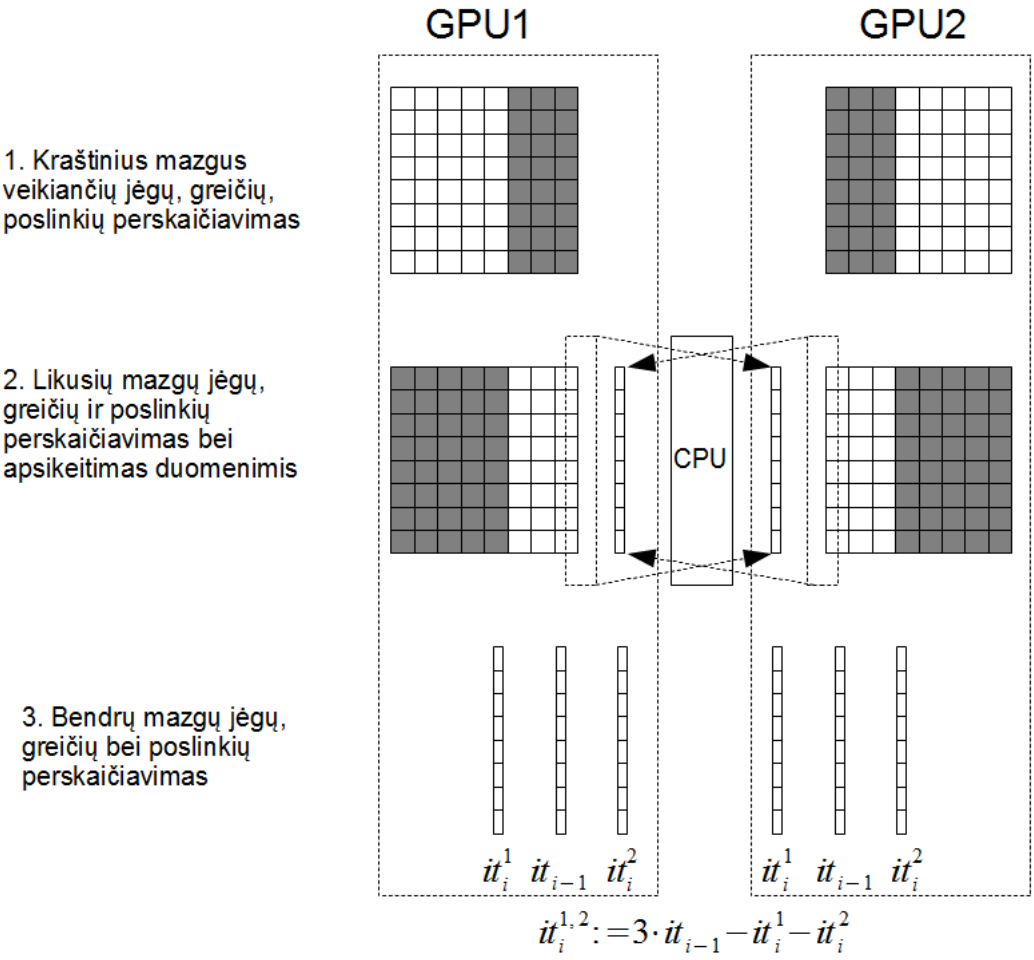
Kaip ir buvo galima tikėtis, 4-asis algoritmas yra šiek tiek lėtesnis (~1,2 karto) lyginant su 3-ioju algoritmu. Tačiau tai nedidelė kaina už padidintą modelio lankstumą.



1. *13 paveikslas: tamprioji banga stačiakampėje plieno plokštelėje su defektu. Modelio dydis – 512K elementų (1mm x 0,5mm), defekto dydis – 64 x 8 elementų (0,064mm x 0,008mm),elemento aukštis – 0,1m integravimo žingsnis – 0,001ns,*

5.6 Skaičiavimų spartinimas panaudojant 2 GPU ir 1 CPU procesorių

Norint skaičiavimus atlikti naudojant 2 GPU procesorius, sritį tenka skelti į 2 dalis. Tokiu atveju po kiekvienos iteracijos procesoriai vienas kitam nusiunčia naujas bendrų mazgų poslinkių, greičių, bei jėgų reikšmes. Apsikeitimo metu duomenys turi būti įrašomi į papildomus masyvus, kad procesoriai neperrašytų savų duomenų. Kiekvienas procesorius dar papildomai turi saugoti praeitos iteracijos persidengiančių mazgų duomenis, kad būtų galima rasti teisingas persidengiančių mazgų poslinkių, greičių bei juos veikiančių jėgų reikšmes. Tam, kad procesoriai tuščiai nelauktų, kol nauji duomenys atkeliaus iš kito procesoriaus (tai gali užtrukti turint galvoje, jog iš abiejų GPU procesorių dedikuotos atminties duomenys turi nukeliauti iki pagrindinės sistemos atminties ir iš ten atgal į GPU dedikuotąją atmintį), P. Micikevičius [12] parodė, jog duomenų mainus įmanoma perdengti skaičiavimais. Pritaikius jo siūlomą metodą šiame darbe tampriųjų bangų simuliacijos spartinimui, skaičiavimo schema kiek pasikeičia. Ji parodyta 14 paveiksle.



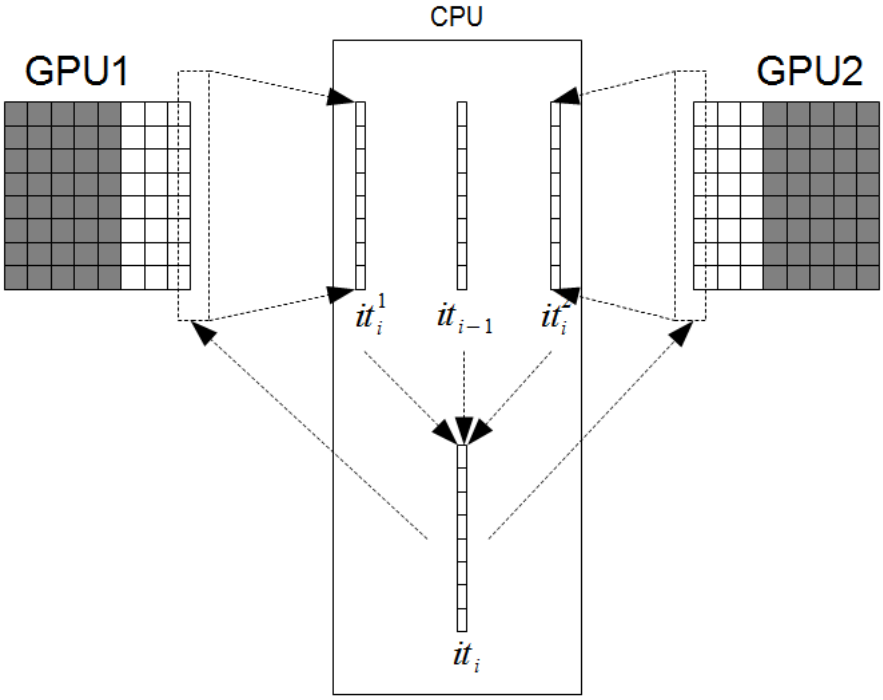
14 paveikslas: P Micikevičiaus pasiūlyta duomenų mainų tarp 2 GPU procesorių schema, pritaikyta darbe nagrinėjamam tamprios bangos modeliui.

Tokia skaičiavimo schema turi du trūkumus:

1. Ankstesniuose skyriuose buvo parodyta, jog GPU procesorius dirba labai neefektyviai su nedideliais modeliais. Todėl aišku, jog 3 žingsnyje neefektyviai išnaudojami GPU procesoriai, nes veiksmai atliekami tik su bendrais mazgais, o jų yra labai nedaug. Pavyzdžiui, jei kiekvienas procesorius apdoroja sritį, kurios dydis - 2048 x 2048 elementų, iš viso yra tik 2048 bendri mazgai.
2. Pagrindinis procesorius atlieka tik prižiūrėtojo vaidmenį: duomenis perkelia iš vieno procesoriaus į kitą ir taip pat nėra išnaudojamas.

Todėl mes siūlome trečiąjį ir antrąjį šios schemos žingsnį sujungti bendrų mazgų jėgų, greičių bei poslinkių naujoms reikšmėms perskaičiuoti panaudojant CPU procesorių ir tik

šiuos duomenis perduoti GPU procesoriams. 2-ojo ir 3-iojo žingsnių sujungimas parodytas 15 paveiksle.



15 paveikslas: duomenų mainų tarp 2 GPU schema bendrų mazgų jėgų, greičių bei poslinkių perskaičiavimui naudojant CPU

Sprendžiant iš anksčiau aprašytų eksperimentų rezultatų, CPU resursų turėtų pilnai pakakti, kad GPU procesoriai neturėtų laukti naujų duomenų, kai jie baigia apdoroti antrąją srities dalį. Iš tikrųjų, procesoriaus turėtų pakakti taip aptarnauti 4 ar net 8 grafiniams procesoriams ir atitinkamai skaičiavimus pagreitinti dar 4, 8 kartus, tačiau šį teiginį reikėtų patvirtinti eksperimentais.

5.7 Modelio trūkumai ir tolimesni darbai

Pagrindinis sukurto modelio (tiksliau sakant, integravimo algoritmo) trūkumas – prisirišimas prie vienodus išmatavimus turinčių sritį dalijančių keturkampių elementų. Paskutinė integravimo algoritmo modifikacija leidžia modeliuoti nehomogenines sritis bei įtrūkimus. Panaudojant bitų kaukę netgi būtų galima modeliuoti įvairaus kontūro įtrūkimus ar sritis, tačiau kadangi modelyje naudojami tik stačiakampiai elementai, glotnių kontūrų aproksimacija būtų gana grubi ir gauti elementų įtempimai ar mazgų poslinkiai netoli kontūro būtų netikslūs. Tačiau bendrą vaizdą apie bangos sklidimą tokioje srityje vistiek būtų galima susidaryti. Todėl ateityje reikėtų ištirti galimybes į modelį įtraukti ir trikampius elementus, kurie leistų tokius kontūrus žymiai geriau aproksimuoti.

Fiksuotas elemento dydis ir forma taip pat reiškia, jog ne tokios svarbios modeliuojamo bandinio sritys turi būti padalijamos į daug daugiau elementų, nei iš tikrųjų reikėtų. Tai automatiškai didinam modelio elementų skaičių ir skaičiavimų laiką. Todėl reikėtų rasti būdą į modelį paprastai įtraukti kitokios formos elementus.

Modeliuojant didesnius įtrūkimus ir naudojant bitų kaukę neišvengiamai atsirastų visiškai neveikiančių blokų, tačiau jie vistiek būtų apdorojami taip švaistant procesoriaus laiką. Todėl reikėtų panagrinėti galimybes sukurti elementų indeksavimo schemą, kuri leistų tokius blokus apskritai pašalinti iš algoritmo bei duomenų vektorių.

Taip pat pravartu būtų sukurti trimatį modelio variantą, nes trimačiai elementai reikalauja daug didesnių skaičiavimų resursų.

5.8 Eksperimentams atlikti naudota techninė bei programinė įranga

- CPU – Intel Core 2 Duo (2,66 GHz, 2 branduoliai – 2 vienu metu dirbančios gijos)
- RAM – 8 GB DDR2, 800 MHz.
- GPU – AMD Radeon HD 6850 (775 MHz, 12 branduolių po 16 vektorinių elementų – 758 vienu metu dirbančios gijos, 1 GB GDDR5 1 GHz dedikuota atmintis)
- OpenCL platforma – OpenCL 1.2 AMD-APP (923.1)
- Programavimo aplinka – Microsoft Visual Studio 2010

6 Išvados

Šiame darbe buvo nagrinėjamos trumpos tampriosios bangos kietoje terpėje modeliavimo baigtinių elementų metodu bei skaičiavimų spartinimo juos lygiagretinant ir naudojant kelių branduolių CPU ir daugelio branduolių GPU procesorius galimybės. Terpę aprašanti diferencialinė lygtis buvo integruojama pasitelkus centrinių skirtumų metodą (CSM).

CSM skaitinio integravimo schema buvo modifikuota taip, jog skaičiavimų algoritmo vieną iteraciją leido padalinti į 3 etapus: išorinių jėgų poveikio įvertinimas, elementų vidinių jėgų poveikio mazgams įvertinimas, bei mazgų greičių, poslinkių ir juos veikiančių jėgų perskaičiavimas. Siekiant supaprastinti modelį, buvo įvestas apribojimas: nagrinėjama stačiakampė sritis dalijama vienodais stačiakampio formos elementais.

Taip pat pasiūlyta sandaugų $[K_e]U_e$ bei $[G_e]U_e$ skaičiavimams atsisakyti ciklų bei naudoti OpenCL C programavimo kalbos vektorinius duomenų tipus, kurie įgalina

pasinaudoti procesorių vektoriniais įtaisais.

Minėtieji CSM integravimo schemos pakeitimai bei modelio supaprastinimas leido išlygiagretinti skaičiavimus remiantis [1] šaltinyje siūlomu metodu. Skaičiavimus tai leido paspartinti 22 kartus naudojant GPU procesorių vietoje CPU.

Vėliau algoritmas buvo optimizuojamas 2 kartus. Iš pradžių atsisakyta elementų mazgų indeksų masyvo – tai leido skaičiavimus paspartinti apie 20%. Po to buvo pereita prie elementų apdorojimo blokais strategijos. Buvo pasinaudota P. Micikevičiaus siūlymu perstumti registruose laikomą informaciją ir taip sumažinti kreipimųsi į lėta GPU dedikuotą atmintį. Pritaikius šią strategiją (tik GPU procesoriui, nes CPU ji netiko), skaičiavimai paspartėjo dar 1,6 karto.

Galiausiai modelis buvo modifikuotas taip, kad būtų galima modeliuoti įtrūkimus bei iš skirtingų medžiagų sudarytas sritis. Įtrūkimams modeliuoti pasiūlyta naudoti blokų elementų bitų kaukę, kur kiekvienas 0 žymi veikiantį bloko elementą, o 1 – neveikiantį. Kiekvienam blokui pasiūlyta naudoti skirtingas elementų koeficientų matricų $[K_e]$ ir $[G_e]$ poras. Tokie sprendimai sparčiausią algoritmą sulėtino 1,2 karto, tačiau naudojant GPU šis universalesnis algoritmas vis tiek išliko ~30 kartų spartesnis nei optimalus paprasčiausio algoritmo CPU variantas.

Darbo pabaigoje buvo pateiktas pasiūlymas, kaip galima būtų skaičiavimus spartinti naudojant 2 GPU ir 1 CPU. Čia buvo remiamasi P. Micikevičiaus siūlymu duomenų mainus perdengti skaičiavimais. Buvo nustatytas šio siūlymo trūkumas – CPU procesoriaus resursų nepanaudojimas, todėl buvo pateiktas siūlymas, kaip skaičiavimams panaudoti ne tik sistemoje esančius GPU procesorius, bet ir pagrindinį CPU.

Išanalizavus eksperimentų rezultatus, daromos tokios išvados:

- Norint maksimaliai išnaudoti GPU procesoriaus resursus, didinant apdorojamų elementų skaičių reikia didinti ir bloko dydį. Kitaip tariant, didėjant modeliui kiekvienai gijai turi tekti vis didesnis apdorojamų elementų kiekis.
- Kuriant bangų modelį (ir ne tik) lygiagrečiai platformai, tikslinga identifikuoti mažus integravimo schemos pakeitimus, kurie galėtų supaprastinti skaičiavimų algoritmo pritaikymą tokiai platformai.
- Daug operacijų reikalaujančias ir dažnai skaičiavimuose dalyvaujančias algoritmo vietas tikslinga optimizuoti atsisakant ciklų ir panašių programavimą lengvinančių

struktūrų. Toks sprendimas reikalauja daugiau dubliuoto kodo. $[K_e]U_e$ bei $[G_e]U_e$ sandaugų apskaičiavimo atveju kiekvienam skirtingam elementui tektų programuoti skirtingas procedūras.

- Universalesnis modelis ar algoritmas yra lankstesnis jo vartotojo požiūriu, tačiau nėra toks našus, kaip specializuotas tam tikromis savybėmis pasižyminčios srities integravimo algoritmas. Todėl programuojant itin didelius modelius, jų suskaidymas į reguliarius blokus, kaip pasiūlė R. Barauskas [5], sprendžiant iš rezultatų, gali skaičiavimų laiką sumažinti iki 30 kartų skaičiavimams atlikti naudojant GPU procesorius.

7 Naudota literatūra

1. KOMATITSCH, D. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster // Journal of Computational Physics. T. 229 (2010), p. 7692-7714.
2. KOMATITSCH, D. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards // Geophysical Journal International. T. 2 (2010), p. 389-402.
3. TABORDA R., Speeding Up Finite Element Wave Propagation for Large-Scale Earthquake Simulations; Carnegie Mellon University. Pittsburgh, 2010.
4. GINZEL E. NDT Modelling An Overview. Materials Research Institute, Waterloo, Ontario, Kanada.
5. BARAUSKAS R., DANIULAITIS V. Simulation of the Ultrasonic Wave Propagation in Solids // Ultragarsas. – 2000, Nr. 4(37), p. 34-39.
6. SCHECHTER R. S. Real-Time Parallel Computation and Visualization of Ultrasonic Pulses in Solids // Science. T 265 (1994), p 1188-1192.
7. GACHAGAN A. Analysis of Ultrasonic Wave Propagation in Metallic Pipe Structures Using Finite Element Modelling Techniques // World Conference on NDT. Montreal, Kanada, 2004.
8. KARAMALIS A. GPU Ultrasound Simulation and Volume Reconstruction // Miuncheno technikos universitetas, Miunchenas, Vokietija, 2009.
9. ZIENKIEWITCZ O. C. The Finite Element Method, Fifth Edition. T. 1, 2, 3: Buterworth-Heineman, 2000.
10. BIHN M., WEILAND T. Numerical Simulation of Ultrasonic Wave Propagation in Inhomogeneous, General Elastically Anisotropic Media // Ultrasonic Symposium, 1998.
11. NICHOLAS L., ARMSTRONG C. Parallel Acoustic Wave Simulation in Two Dimensions for Mediums of Varying Density // University of Alaska Anchorage.
12. MICIKEVICIUS P. 3D Finite Difference Computation on GPUs using CUDA // Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. New York, JAV, p. 79-84. ISBN 978-1-60558-517-8.

13. OWENS D. GPU Computing // Proceedings of the IEEE. T 96 (2008), p. 879-899.
14. AMD Radeon™ HD 6850 Graphics Overview. [žiūrėta 2011-11-25]. Prieiga per internetą: http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/hd-6850/Pages/amd-radeon-hd-6850-overview.aspx#2
15. A Brief History of General Purpose (GPGPU) Computing [žiūrėta 2011-12-03]. Prieiga per internetą: http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/gpgpu-history.aspxs
16. OpenCL standarto puslapis: http://www.khronos.org/opencl/
17. AMD Betting Everything on OpenCL. Iš bit-tech [interaktyvus]. 2011, gegužė [žiūrėta 2011-12-15]. Prieiga per internetą: http://www.bit-tech.net/hardware/cpus/2011/05/30/amd-betting-everything-on-opencl/1
18. MUNSHI A. OpenCL Programming Guide: Adison-Wesley, 2011. 603 p.
19. AMD Accelerated Parallel Processing OpenCL Programming Guide // Advanced Micro Devices, 2011. Prieiga per internetą: http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf
20. KERN M. Parallel Solution of the Wave Equation Using Higher Order Finite Elements. Le Chesnay C´edex, Prancūzija.
21. ZAMITH P. M. Simulation of Wave Propagation in Semi-Infinite Domains Using The Finite Difference Method on a GPU Based on Cluster // Mecánica Computacional. T 29 (2010). p. 7147-7157.
22. BARAUSKAS, R. Baigtinių elementų metodo pagrindai. Vilnius: Technika, 2004. 612p. ISBN 9986-05-792-2

8 Santrauka

Trumpųjų bangų sklidimo modelis daugiaprocesorinėje aplinkoje

Santrauka

Tampriosios bangos (arba akustinės ar bet kokios kitos bangos) sklidimo tyrimai yra svarbūs tokiose srityse kaip seismologija arba neardantis medžiagos testavimas. Tamprioje srityje šis reiškinys aprašomas tampriosios bangos dinamine diferencialine lygtimi. Tačiau šios lygties sprendimas naudojant tokius skaitinius metodus kaip baigtiniai elementai reikalauja sritį padalinti į milijonus elementų. Naujų skaičiavimo technologijų kaip bendros paskirties grafiniai procesoriai (GPU) atsiradimas skaičiavimų laiką leidžia ženkliai sumažinti, tačiau algoritmai turi būti specialiai pritaikomi.

Todėl šiame darbe koncentruojamasi į trumpos tampriosios bangos baigtinių elementų modelio sukūrimą ir algoritmų tobulinimą naudojant GPU bei pagrindinį procesorių (CPU). Lygties integravimui buvo pasirinktas centrinių skirtumų metodo (CSM) schema. Ši integravimo schema buvo modifikuota taip, kad būtų galima išskirti tris integravimo algoritmo etapus: išorinės jėgos įvertinimas, elementų deformacijos sąlygotų jėgų įvertinimas bei magų poslinkių, greičių ir jėgų perskaičiavimas. Remiantis strategija pasiūlyta [1] šaltinyje, buvo sukurti lygiagretūs algoritmai 2 ir 3 etapo skaičiavimams atlikti. Toliau antrojo etapo algoritmas buvo optimizuotas 2 kartus. Pirmiausia buvo atsisakyta elementų mazgų indeksų masyvo: tai skaičiavimo laiką sumažino 20%. Po to algoritmas buvo modifikuotas taip, kad elementus būtų galima apdoroti blokais kaip siūloma [12] ir [22] šaltiniuose. Skaičiavimo laiką tai leido sumažinti dar 60%. Toliau algoritmas buvo modifikuotas dar kartą taip, kad būtų galima į modelį įtraukti bandinio defektus ir medžiagos nereguliarumus. Tačiau ši modifikacija skaičiavimo laiką padidino tik 20%. Galiausiai buvo pateiktas siūlymas, kaip efektyviai išnaudoti 2 GPU ir 1 CPU.

Atlikus darbą ir išanalizavus rezultatus buvo padarytos tokios išvados:

- Didinant modelį taip pat reikia didinti ir elementų, tenkančių vienai gijai, skaičių. Tikslus elementų skaičius, tenkantis vienai gijai, nustatomas eksperimentais.
- Tikslinga modifikuoti integravimo schemą taip, kad būtų lengviau sukurti lygiagretų skaičiavimo algoritmo variantą.
- Daug operacijų reikalaujančias ir dažnai skaičiavimuose dalyvaujančias algoritmo vietas tikslinga optimizuoti atsisakant ciklų ir panašių programavimą lengvinančių struktūrų.
- Universalus modelis ir algoritmas nėra toks efektyvus skaičiavimo laiko prasme lyginant su specializuotu modelio variantu, tačiau nedideli universalumą didinantys pakeitimai nereikalauja didelių skaičiavimo laiko sąnaudų.

9 Santrauka užsienio (anglų) kalba

Development of the model of short wave propagation by using multi-processor environment

Summary

Understanding elastic wave (or acoustic or any other type of wave for that matter) phenomenon is of great importance in areas such as seismology or non destructive testing (NDT). This phenomenon in case of elastic environment is described by dynamic elastic differential equations. However, computational models like finite element method consumes huge amounts of computational power as even for relatively small problems require dividing area of interest into millions of elements. In the advent of general purpose GPU computing new opportunities for speeding up computations as well as challenges for developing high performance algorithms suited for new kinds of processors arise.

Therefore this work concentrates on developing a finite element based short elastic wave propagation model on GPU as well as CPU. Central difference explicit wave equation integration scheme has been chosen. It then was slightly modified in order to separate integration algorithm into three phases: external force evaluation, evaluation of forces that occur due to stresses of elements and recalculation of node shifts, speeds and forces. A parallel algorithm has been developed for executing third and seconds phases, based on strategy suggested in [1]. Then the algorithm of the second phase has been optimized 2 times: at first the array of element node indices was eliminated yielding 20% performance boost, then modifications have been made to process elements in blocks by using strategy described at [22]. This caused another 60% reduction of computational time. Then the algorithm was modified again to allow introduction of cracks and irregularities in the model. This caused only 20% of computational time increase. Finally suggestions for effectively using 2 GPUs and 1 CPU has been made.

The following conclusions has been drawn after analysing the results:

- When model size increases so should the work performed by one thread. In other words, the bigger the model, the more elements must be processed to achieve maximum processor efficiency. The exact amount of work performed by one thread can only be determined by experiments.
- It is beneficial to try modifying whatever integration scheme is used to make parallelization of an algorithm easier.
- Small parts of integration algorithm that contain most of calculations performed, should be additionally optimized by unrolling loops and avoiding similar programming simplifications to reduce number of operations performed.
- More universal models and algorithms are not as efficient as algorithms developed for wave simulations in the environment with some specific restrictions.

10 Priedai

10.1 3-iojo srities integravimo etapo realizacija OpenCL C kalba

```
kernel void integrate(
   const float dt,
   const global float2 * node_masses,
   const global float2 * node_dampness,
   global float2 * node_forces,
   global float2 * node_speeds,
   global float2 * node_shifts)
{
   uint g_id = get_global_id(0);
   float2 speed = node_speeds[g_id]
                  + node_forces[g_id] * node_masses[g_id] * dt;
   float2 shift = node_shifts[g_id] + speed * dt;
   float2 force = node_dampness[g_id] * speed;
   node_speeds[g_id] = speed;
   node_shifts[g_id] = shift;
   node_forces[g_id] = force;
}
```

10.2 Elementų įtempimo jėgų įvertinimo algoritmų OpenCL C funkcijos

10.2.1 1-asis algoritmas

```
#define sum_v8(v) ((v.s0 + v.s1) + (v.s2 + v.s3))
                     + ((v.s4 + v.s5) + (v.s6 + v.s7))

kernel void apply_external_forces(
   int force_type,
   float force_strength,
   global float2 * node_forces)
{
   const uint g_id = get_global_id(0);
   if (force_type == FORCE_TYPE_EDGE) {
      node_forces[g_id].y += force_strength;
   }
   else {
      const uint g_size_half = get_global_size(0) / 2;
      if (g_size_half == g_id) {
         node_forces[g_id].y += force_strength;
      }
   }
}

kernel __attribute__((reqd_work_group_size(8, 8, 1)))
void apply_element_forces(
   const int calculate_stresses,
   const uint2 elem_idx_shift,
   constant float8 * rows_K,
   constant float8 * rows_G,
   const global uint4 * elem_nodes,
   const global float2 * node_shifts,
   global float2 * node_forces,
   global float * elem_stresses)
{
   const uint elem_x = 2 * get_global_id(0) + elem_idx_shift.x;
   const uint elem_y = 2 * get_global_id(1) + elem_idx_shift.y;
   const uint elem_idx = elem_y * get_global_size(0) * 2 + elem_x;
```

```
   const uint4 elem_node_indices = elem_nodes[elem_idx];

   float8 U = 0.0f;
   U.s01 = node_shifts[elem_node_indices.x];
   U.s23 = node_shifts[elem_node_indices.y];
   U.s45 = node_shifts[elem_node_indices.z];
   U.s67 = node_shifts[elem_node_indices.w];

   float8 KxU = 0.0f;
   float8 tmp = 0.0f;
   tmp = rows_K[0] * U; KxU.s0 = sum_v8(tmp);
   tmp = rows_K[1] * U; KxU.s1 = sum_v8(tmp);
   tmp = rows_K[2] * U; KxU.s2 = sum_v8(tmp);
   tmp = rows_K[3] * U; KxU.s3 = sum_v8(tmp);
   tmp = rows_K[4] * U; KxU.s4 = sum_v8(tmp);
   tmp = rows_K[5] * U; KxU.s5 = sum_v8(tmp);
   tmp = rows_K[6] * U; KxU.s6 = sum_v8(tmp);
   tmp = rows_K[7] * U; KxU.s7 = sum_v8(tmp);

   node_forces[elem_node_indices.x] -= KxU.s01;
   node_forces[elem_node_indices.y] -= KxU.s23;
   node_forces[elem_node_indices.z] -= KxU.s45;
   node_forces[elem_node_indices.w] -= KxU.s67;

   if (calculate_stresses == 1) {
      float4 SIGMA = 0.0f;
      for (int row = 0; row < 3; row++) {
         float8 tmp = 0.0f;
         tmp = rows_G[0] * U; SIGMA.s0 = sum_v8(tmp);
         tmp = rows_G[1] * U; SIGMA.s1 = sum_v8(tmp);
         tmp = rows_G[2] * U; SIGMA.s2 = sum_v8(tmp);
      }
      elem_stresses[elem_idx] = length(SIGMA.s01);
   }
}
```

10.2.2 2-asis algoritmas

```
#define sum_v8(v) ((v.s0 + v.s1) + (v.s2 + v.s3))
                     + ((v.s4 + v.s5) + (v.s6 + v.s7))

kernel __attribute__((reqd_work_group_size(8, 8, 1)))
void apply_element_forces_cnidx(
   const int calculate_stresses,
   const uint2 elem_idx_shift,
   constant float8 * rows_K,
   constant float8 * rows_G,
   const global uint4 * elem_nodes,
   const global float2 * node_shifts,
   global float2 * node_forces,
   global float * elem_stresses)
{
   const uint elems_in_row = get_global_size(0) * 2;
   const uint nodes_in_row = elems_in_row + 1;
   const uint elem_x = 2 * get_global_id(0) + elem_idx_shift.x;
   const uint elem_y = 2 * get_global_id(1) + elem_idx_shift.y;
   const uint elem_idx = elem_y * elems_in_row + elem_x;

   const uint4 elem_node_indices = 0;
   elem_node_indices.x = elem_y * nodes_in_row + elem_x;
   elem_node_indices.y = elem_node_indices.x + 1;
```

```
   elem_node_indices.zw = elem_node_indices.yx + nodes_in_row;

   float8 U = 0.0f;
   U.s01 = node_shifts[elem_node_indices.x];
   U.s23 = node_shifts[elem_node_indices.y];
   U.s45 = node_shifts[elem_node_indices.z];
   U.s67 = node_shifts[elem_node_indices.w];

   float8 KxU = 0.0f;
   float8 tmp = 0.0f;
   tmp = rows_K[0] * U; KxU.s0 = sum_v8(tmp);
   tmp = rows_K[1] * U; KxU.s1 = sum_v8(tmp);
   tmp = rows_K[2] * U; KxU.s2 = sum_v8(tmp);
   tmp = rows_K[3] * U; KxU.s3 = sum_v8(tmp);
   tmp = rows_K[4] * U; KxU.s4 = sum_v8(tmp);
   tmp = rows_K[5] * U; KxU.s5 = sum_v8(tmp);
   tmp = rows_K[6] * U; KxU.s6 = sum_v8(tmp);
   tmp = rows_K[7] * U; KxU.s7 = sum_v8(tmp);

   node_forces[elem_node_indices.x] -= KxU.s01;
   node_forces[elem_node_indices.y] -= KxU.s23;
   node_forces[elem_node_indices.z] -= KxU.s45;
   node_forces[elem_node_indices.w] -= KxU.s67;

   if (calculate_stresses == 1) {
      float4 SIGMA = 0.0f;
      for (int row = 0; row < 3; row++) {
         float8 tmp = 0.0f;
         tmp = rows_G[0] * U; SIGMA.s0 = sum_v8(tmp);
         tmp = rows_G[1] * U; SIGMA.s1 = sum_v8(tmp);
         tmp = rows_G[2] * U; SIGMA.s2 = sum_v8(tmp);
      }
      elem_stresses[elem_idx] = length(SIGMA.s01);
   }
}
```

10.2.3 3-iasis algoritmas

```
#define sum_v8(v) ((v.s0 + v.s1) + (v.s2 + v.s3))
                  + ((v.s4 + v.s5) + (v.s6 + v.s7))

#define fetch() \
   U.s67 = node_shifts[node_idx_top];  \
   U.s45 = node_shifts[node_idx_top + 1]

#define write() \
   node_forces[node_idx_bottom    ] += F.s01;
barrier(CLK_GLOBAL_MEM_FENCE); \
   node_forces[node_idx_bottom + 1] += F.s23; barrier(CLK_GLOBAL_MEM_FENCE)

// register shifting: move data at top to bottom
#define shift() \
   U.s0123 = U.s6745; \
   F.s0123 = F.s6745; \
   F.s4567 = 0.0f; \
   node_idx_bottom = node_idx_top;

#ifndef L_x
#define L_x 32
#endif
```

```
#ifndef L_y
#define L_y 2
#endif

#ifndef WI_SIZE
#define WI_SIZE 8
#endif


kernel __attribute__((reqd_work_group_size(L_x, L_y, 1)))
void apply_element_forces(
   const int calculate_stresses,
   const uint2 group_shift,
   constant float8 * rows_K,
   constant float8 * rows_G,
   const global float2 * node_shifts,
   global float2 * node_forces,
   global float * elem_stresses)
{
   const uint elems_in_row = get_global_size(0) * 2;
   const uint nodes_in_row = elems_in_row + 1;
   const uint elem_x = (get_group_id(0) * 2 + group_shift.x) * L_x
                        + get_local_id(0);
   uint elem_y = ((get_group_id(1) * 2 + group_shift.y) * L_y
                                              + get_local_id(1)) * WI_SIZE;

   uint node_idx_bottom = 0.0f;
   uint node_idx_top = 0.0f;
   float8 U = 0.0f;
   float8 F = 0.0f;
   float8 KxU = 0.0f;
   float8 tmp = 0.0f;

   // fetch the bottom data
   node_idx_top = elem_y * nodes_in_row + elem_x;
   fetch();
   shift();

   for (uint wi_id = 0; wi_id < WI_SIZE; wi_id++) {
      node_idx_top = (elem_y + 1) * nodes_in_row + elem_x;
      fetch();

      tmp = rows_K[0] * U; KxU.s0 = sum_v8(tmp);
      tmp = rows_K[1] * U; KxU.s1 = sum_v8(tmp);
      tmp = rows_K[2] * U; KxU.s2 = sum_v8(tmp);
      tmp = rows_K[3] * U; KxU.s3 = sum_v8(tmp);
      tmp = rows_K[4] * U; KxU.s4 = sum_v8(tmp);
      tmp = rows_K[5] * U; KxU.s5 = sum_v8(tmp);
      tmp = rows_K[6] * U; KxU.s6 = sum_v8(tmp);
      tmp = rows_K[7] * U; KxU.s7 = sum_v8(tmp);

      F -= KxU;

      if (calculate_stresses == 1) {
         float4 SIGMA = 0.0f;
         for (int row = 0; row < 3; row++) {
            float8 tmp = 0.0f;
            tmp = rows_G[0] * U; SIGMA.s0 = sum_v8(tmp);
            tmp = rows_G[1] * U; SIGMA.s1 = sum_v8(tmp);
            tmp = rows_G[2] * U; SIGMA.s2 = sum_v8(tmp);
         }
```

```
            elem_stresses[elem_y * elems_in_row + elem_x] = length(SIGMA.s01);
         }

         write();
         shift();
         elem_y++;
      }
      write();
   }
```

10.2.4 4-asis algoritmas

```
#define fetch() \
   U.s67 = node_shifts[node_idx_top];  \
   U.s45 = node_shifts[node_idx_top + 1]

#define write() \
   node_forces[node_idx_bottom    ] += F.s01;
barrier(CLK_GLOBAL_MEM_FENCE); \
   node_forces[node_idx_bottom + 1] += F.s23; barrier(CLK_GLOBAL_MEM_FENCE)

// register shifting: move data at top to bottom
#define shift() \
   U.s0123 = U.s6745; \
   F.s0123 = F.s6745; \
   F.s4567 = 0.0f; \
   node_idx_bottom = node_idx_top;

#define L_x 16
#define L_y 4
#define WI_SIZE 8


kernel __attribute__((reqd_work_group_size(L_x, L_y, 1)))
void apply_element_forces(
   const int calculate_stresses,
   const uint2 group_shift,
   constant uint4 * control_data,
   const global uint16 * crack_mask_data,
   global float8 * g_rows_K,
   global float8 * g_rows_G,
   const global float2 * node_shifts,
   global float2 * node_forces,
   global float * elem_stresses)
{
   const uint group_x = get_group_id(0) * 2 + group_shift.x;
   const uint group_y = get_group_id(1) * 2 + group_shift.y;
   const uint l_x = get_local_id(0);
   const uint l_y = get_local_id(1);
   const uint elems_in_row = get_global_size(0) * 2;
   const uint nodes_in_row = elems_in_row + 1;
   const uint elem_x =  group_x * L_x + l_x;
         uint elem_y = (group_y * L_y + l_y) * WI_SIZE;

   uint node_idx_bottom = 0.0f;
   uint node_idx_top = 0.0f;
   float8 U = 0.0f;
   float8 F = 0.0f;
   float8 KxU = 0.0f;
   float8 tmp = 0.0f;
```

```
local uint crack_mask[L_x];
local float8 rows_K[8];
local float8 rows_G[3];

uint wi_mask = 0;
const uint4 control = control_data[group_y * get_num_groups(0) * 2
                      + group_x];
#define matrix_index control.z
#define group_has_cracks control.x
#define mask_idx control.y

event_t evt[2];
evt[0] = async_work_group_copy((local float8 *) rows_K, g_rows_K +
                                          (matrix_index * 8), 8, NULL);
evt[1] = async_work_group_copy((local float8 *) rows_G, g_rows_G +
                                          (matrix_index * 3), 3, NULL);
wait_group_events(2, evt);


if (group_has_cracks) {
   event_t evt = async_work_group_copy(
      (local uint16 *) crack_mask, crack_mask_data + mask_idx, 1, NULL);
   wait_group_events(1, &evt);
   // grab the right 8 bits as a crack mask for current work item
   wi_mask = crack_mask[l_x] >> (WI_SIZE * l_y);
}

// fetch the bottom data
node_idx_top = elem_y * nodes_in_row + elem_x;
fetch();
shift();

for (uint wi_id = 0; wi_id < WI_SIZE; wi_id++) {
   node_idx_top = (elem_y + 1) * nodes_in_row + elem_x;
   fetch();

   tmp = rows_K[0] * U; KxU.s0 = sum_v8(tmp);
   tmp = rows_K[1] * U; KxU.s1 = sum_v8(tmp);
   tmp = rows_K[2] * U; KxU.s2 = sum_v8(tmp);
   tmp = rows_K[3] * U; KxU.s3 = sum_v8(tmp);
   tmp = rows_K[4] * U; KxU.s4 = sum_v8(tmp);
   tmp = rows_K[5] * U; KxU.s5 = sum_v8(tmp);
   tmp = rows_K[6] * U; KxU.s6 = sum_v8(tmp);
   tmp = rows_K[7] * U; KxU.s7 = sum_v8(tmp);

   const float mask_elem = (wi_mask >> wi_id) ? 0.0f : 1.0f;
   F -= mask_elem * KxU;

   if (calculate_stresses == 1) {
      float4 SIGMA = 0.0f;
      for (int row = 0; row < 3; row++) {
         float8 tmp = 0.0f;
         tmp = rows_G[0] * U; SIGMA.s0 = sum_v8(tmp);
         tmp = rows_G[1] * U; SIGMA.s1 = sum_v8(tmp);
         tmp = rows_G[2] * U; SIGMA.s2 = sum_v8(tmp);
      }
      elem_stresses[elem_y * elems_in_row + elem_x] =
                              mask_elem * length(SIGMA.s01);
   }

   write();
```

```
            shift();
            elem_y++;
        }
        write();
    }
```